Designing Real-Time 3D Graphics for Entertainment

SIGGRAPH '96 Course #33

Organizer:

James Helman
Silicon Graphics

Lecturers:

Andy Bigos
3Dlabs

Philippe Tarbouriech
Electronic Arts

Eric Johnston
LucasArts

Scott Watson
Walt Disney Imagineering

Steve Rotenberg
Angel Studios

Designing Real-Time 3D Graphics for Entertainment

Organizer:
James Helman
Silicon Graphics

SIGGRAPH '96 Course

Abstract

This course covers the issues of creating real-time 3D games on platforms ranging from home game consoles up to high-performance image generators used in theme parks. Topics include the hardware architectures of various game platforms, visual simulation tricks, 3D modeling, real-time character animation, game prototyping and programming. The authors draw examples from the development of actual games, tools and game development environments.

The course has two components. The first part covers the graphics and programming techniques available to make the best use of graphics technology for high-quality, real-time renderings. The topics include hardware and software architectures, graphics optimization, database tuning and other tricks of the trade. The visual simulation roots of many of these hardware and software techniques is also covered.

In the second part, developers discuss the use of those techniques as one component in creating interactive 3D experiences, whether for home game consoles or for location-based entertainment or theme park installations. The topics covered include tools and methods for content generation, software frameworks, and animation systems.

Speakers

James Helman
Silicon Graphics

Jim Helman works in Silicon Graphics' Advanced Graphics Division as a member of the engineering team for IRIS Performer, SGI's real-time graphics toolkit. Before coming to SGI, he was a student in the Applied Physics department at Stanford University where he worked on his PhD in data visualization. His interests include virtual environments, game design, and keeping large green cars running.

Silicon Graphics	Email:	jimh@sgi.com
Advanced Graphics Division	Phone:	(415) 933-1151
2011 N. Shoreline Blvd. MS 8U-590	FAX:	(415) 965-2658
Mountain View, CA 94043 USA		

Andy Bigos
3Dlabs

Andy has been on the engineering staff at 3Dlabs (formally DuPont Pixel) for 5 years. As part of the GLINT & PERMEDIA core architecture teams he helped bring workstation class 3D graphics to the PC platform. As well as working on core architectures he's being closely involved with porting and optimizing OpenGL and Direct3D for 3Dlabs hardware. Andy is currently working with game developers to enable console class games on PC. He holds a Masters degree in Computer Graphics and a Bachelors degree in Engineering, his interests include real-time physically based modelling as well as play testing the latest and greatest games.

Andy Bigos	Email:	andy.bigos@3Dlabs.com
3Dlabs	Phone:	44 1784 476 636
Meadlake Place, Thorpe Lea Road	FAX:	44 1784 470 699
Egham, Surrey, TW20 8HE, UK		

Philippe Tarbouriech
Electronic Arts

Philippe Tarbouriech works in Electronic Arts as a member of the Advanced Technology Group. He manages the conversion of ShockWave to Playstation and PC while working on other projects. He was the software designer, aerial photographer and one of the game designers of the original ShockWave on 3DO. Prior to joining Electronic Arts, he worked in diverse unfit startup companies. His interests include evolution. He holds a M.S. in electrical engineering and computer science from ENSEA (France).

Electronic Arts	Email:	ptarbouriech@ea.com
1450 Fashion Island Blvd.Phone:	Phone:	(415) 513-7191
San Mateo, CA 94404	FAX:	(415) 571-1893

Eric Johnston
LucasArts

Eric Johnston is currently the technical lead for LucasArts Entertainment s 3D console development group. Previously at Spectrum HoloByte, as head of their VR group, he developed Onyx-based games for location-based entertainment applications. As a Macintosh games programmer, his credits include the Mac versions of Rebel Assault, Indiana Jones and the Fate of Atlantis, Monkey Island 1 and 2, Loom, Pipe Dream, and Putt-Putt Joins the Parade. Eric graduated from U.C. Berkeley, with a B.S. in EE and CS. A former windsurfing instructor, he currently spends too much of his spare time on the flying trapeze.

LucasArts	E-mail:	ej@lucasarts.com
1600 Los Gamos Drive	Phone:	(415) 444-8437
San Rafael, CA 94903	FAX:	(415) 444-8240

Scott Watson
Walt Disney

Scott Watson is Walt Disney Imagineering's VR Studio Technology Director. Scott started programming at 9 years of age. In 5th grade, IBM loaned him their first portable (75lbs) computer, the 5100 in exchange for writing games and demos to show it off. In his college days, when 8-bit machines where all the rage, Scott wrote multi-tasking OSes, device drivers, cross compilers and RF communications stacks as a day job. His free time was dedicated to his band "The Loved Ones" and Fanzine, "The Pig Paper."

Upon joining Disney's R&D department, his first assignment was to write the control software for the "Indiana Jones Ride Vehicle." An eclectic spectrum of projects has followed. Examples range from creating audio and image processing technology for theme-park films to helping design a computer keyboard for dolphins. For several years "Disney.com" was the machine on his desk. Since the beginning of Disney's exploration of Virtual Reality, Scott has been at the heart of the technology and is the principal designer of the "Disney*Vision Player." Disney*Vision is an interactive VR story development system that supports real-time Disney quality character animation and the SAL scripting language. Scott holds several patents and is an avid fan of The Monkees.

Walt Disney Imagineering
1401 Flower Street
Glendale, CA 91221

E-mail: scott@disney.com
Phone: (818) 544-6790
FAX: (818) 544-4544

Steve Rotenberg
Angel Studios

Steve Rotenberg joined Angel Studios in the fall of 1992 to redesign and expand Angel Studios' proprietary 3D animation software and is now Angel Studios' Director of Software Development. Having embarked on his programming career at the age of six, Steve has amassed extensive experience designing software for a variety of platforms. During the past two years, Steve's natural interest in problem solving and his object-oriented approach to software design has proven integral to Angel Studios' development of ANGEL ARTS(tm), advanced real-time software for both the location-based and home entertainment markets. Using his invaluable experience in such software development, Steve has lead teams in developing VR experiences such as Dr. Megow's Mad Cap Ornithon and F-1 Net Race and, more recently, Buggie Boogie for Nintendo's next generation home game platform, the Ultra 64.

Angel Studios
5962 La Place Courte, Suite 100
Carlsbad, CA 92008

Email: steve@angel.com
Phone: (619) 929-0700
FAX: (619) 929-0719

Additional Course Notes Contributors

Sharon Clay
Silicon Graphics

Sharon (Fischler) Clay is a member of the IRIS Performer engineering team in the Advanced Graphics Division at Silicon Graphics where she specializes in performance issues for system implementation and real-time graphics applications. She was a member of the original design team, and before that, was a member of the Graphics Software group where she worked on the development team for the VGX graphics platform. Her interests, besides a real need for speed, include user interfaces, plants and fish (simulated and real). She studied using natural language in graphical user interfaces at the University of California at Santa Cruz where she received her Masters degree in Computer Science. Her Bachelors degree is in Mathematics and Linguistics from the University of California at Berkeley.

Silicon Graphics
Advanced Graphics Division
2011 N. Shoreline Blvd. MS 8U-590
Mountain View, CA 94043 USA

Email: src@sgi.com
Phone: (415) 933-1002
FAX: (415) 965-2658

Wes Hoffman
Paradigm Simulation

Wes Hoffman is the founder of Paradigm Simulation Inc. Paradigm Simulation was started five years ago and has positioned itself as a leader in the real-time 3D market. The database and programs he has worked on are well known by those in the industry, such as the Performer town database and the Magic Edge location based entertainment experience. Mr. Hoffman is currently leading the database development effort for Paradigm's Nintendo Ultra 64 game. Before working at Paradgim he worked at Merit Technology building a real-time simulation toolkit. Mr. Hoffman graduated from Syracuse University with a Bachelor of Fine Art and a major in computer graphics. His other interests include making crop-circles.

Paradigm Simulation
15280 Addison Road Suite 360
Dallas, TX 75248

Email: wes@paradigmsim.com
Phone: (214) 960-2301
FAX: (214) 960-2303

Michael Jones
Silicon Graphics

Michael Jones works in Silicon Graphics' Advanced Graphics Division where he manages the IRIS Performer engineering team. He has worked with high-performance visual simulation systems for the last 5 years. Prior to joining Silicon Graphics, he worked in diverse areas, including high-performance visual simulation, color conversion of monochrome films and serials, optimization of cellular telephone antenna placement, and the nationwide routing of delivery trucks. He has been a professional computer programmer since the seventh grade. His personal interests include work and sleep.

Silicon Graphics
Advanced Graphics Division
2011 N. Shoreline Blvd. MS 8U-590
Mountain View, CA 94043 USA

Email: mtj@sgi.com
Phone: (415) 933-1455
FAX: (415) 965-2658

Michael Limber
Angel Studios

Michael Limber is COO at Angel Studios and serves as its Production Director. He has a degree in Architecture from U.C. Berkeley and a Masters in Industrial Design from Pratt Institute. His professional computer graphics career began in 1985 after getting a job at Digital Productions in Los Angeles. Beginning as a modeler and progressing to the position of Technical Director, Michael spent two intense years at the trailblazing computer graphics company, using a Cray XMP and state-of-the-art proprietary software to create commercials, film effects, visualizations, and music videos. After serving as Head Animator at the fully digital Post Perfect, in 1989, he moved to San Diego to work as Director of Computer Animation at Angel Studios with another Digital Productions veteran and colleague, Brad Hunt.

Michael worked as Animator and Technical Director on "The Lawnmower Man" and Peter Gabriel's MindBlender, among a multitude of other unique projects.Since that time, Angel Studios has become a significant developer of real-time interactive entertainment for companies like Sega, Hasbro, and most recently Nintendo.

Angel Studios
5962 La Place Courte, Suite 100
Carlsbad, CA 92008

Email: michael@angel.com
Phone: (619) 929-0700
FAX: (619) 929-0719

Agenda

8:30 **Introduction**
Architecture and Performance of Entertainment Systems
Jim Helman - *Silicon Graphics*

10:00 **Break**

10:15 **Exploiting Consumer Class 3D Hardware Acceleration for Real-Time Entertainment**
Andy Bigos - *3Dlabs*

11:00 **Tuning to the Metal**
Philippe Tarbouriech - *Electronic Arts*

12:00 **Lunch**

1:30 **Prototyping and Portability of Hardware-Assisted 3D Games**
Eric Johnston - *LucasArts*

2:15 **Action!**
Scott Watson - *Walt Disney*

3:00 **Break**

3:15 **Creating Compelling Real-Time Content**
Steve Rotenberg - *Angel Studios*

4:15 **Panel Discussion, Q & A**
All

5:00 **Close**

Contents

Architecture and Performance of Entertainment Systems

James Helman
Silicon Graphics Computer Systems

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

In the beginning there was the Drive In
And Walt said, "Let there be a Park."
And lo, there was Pirates of the Caribbean
Thus did Walt beget the Experience Industry
And Walt saw that it was Good...
- Michael Krantz Figure [Kranz94]

1 Introduction

The use of real-time 3D computer graphics in interactive entertainment has grown dramatically recently. These entertainment applications include better arcade games, 3D-capable home game consoles, more sophisticated multi-player games for location-based entertainment (LBE) centers, virtual actors on TV driven by puppeteers with motion capture devices, and even virtual interactive theatres where the "player" can assume the role of a character in a story and alter the course of the plot.

This chapter of the course notes tries to provide a general background into the elements that go into creating a real-time 3D graphics entertainment application and the basic performance levels required to meet human factors requirements. Subsequent chapters fill in the details of content generation, programming and graphics techniques that can be used to meet those performance requirements across platforms ranging from home $250 game consoles to image generators for multiplayer LBEs costing $100,000 and above.

2 What's New?

The initial wave of real-time, 3D graphics for is hitting the entertainment market at many different levels. At the high end, theme park based systems, such as Epcot Center's Aladdin VR experience which opened last summer, are running on high-end graphics workstations. Such systems can support a quality of content that approaches the production values of TV or film with complex scenes covered with hand-painted imagery and complex characters animated in real-time. The same set of underlying graphics capabilities that make this possible, most notably texture mapping, high polygonal complexity and 3D character animation can be seen moving into less expensive systems produced for location-based entertainment use such as Magic Edge, W Industries, Iwerks and Virtual World Entertainment. With the latest generation of arcade machines and home game consoles like the Sega Saturn, 3DO Multiplayer, Atari Jaguar and, of course, the Nintendo Ultra64, many of these same capabilities are beginning to appear in the home.

With most LBE sites, many arcade and home games taking on a "virtual" moniker, entertainment is often called the "killer-app" for virtual reality. Alternatively, one could say that the same improvements in technology that enable VR are also enabling new applications of computer graphics in entertainment, some which are immersive in the VR sense and some of which are not.

Whatever one's perspective, the technological forces behind this movement can be seen by looking back at 3D computer graphics over the last decade. Two developments are key: the evolution of graphics hardware and the creative skills to use 3D computer graphics effectively.

Hardware Evolution

One could divide relevant CG applications into areas with different performance and cost requirements.

- computer generated imagery (CGI) for film and broadcast.
  very high image quality → low frame rate @ high cost.
- modeling, animation production, MCAD and data visualization.
  medium frame rates → low image quality @ medium cost.
- 2D video games.
  low cost, high frame rates → low image quality
- visual simulators.
  high frame rate, medium image quality → high cost

The equation that has constrained high and medium quality rendering to the realm of frame-by-frame CGI is very roughly:

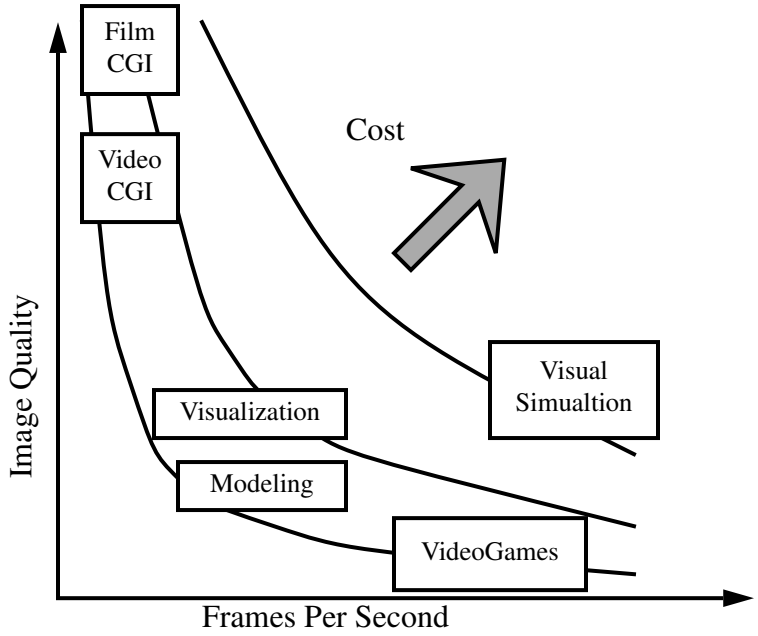


FIGURE 1. Frame Rate versus Image Quality versus Cost

$$cost \propto frame\ rate\ X\ image\ quality \quad \textbf{(EQ 1)}$$

Based on their requirements, applications find different sets of tradeoffs between cost, image quality and frame rate attractive as shown in Figure 1

Until very recently, realistic 3D graphics has fallen into two camps. One could produce imagery with sufficiently high quality to meet Hollywood standards, but with rendering times measured in minutes per frame. Or using high-end visual simulation equipment, you could produce marginally realistic graphics at 12 to 60 frames per second, but at costs per display channel in the $200,000 to $1,000,000 range.

On the technical side, what's changing is the continuing improvement in the price performance of computers and graphics hardware. This decreases the proportionality constant every year and moves the cost curves back towards the origin for a particular combination of quality and frame rate. For example, a system suitable for visual simulation system that might have cost $200,000 per channel might now cost 1/10th or 1/20th of that. The result is that real-time 3D graphics becomes practical for more uses in entertainment systems all the way from 3D texture mapped video games to high-quality theme park attractions.

So that now it's becoming possible to have:

high frame rate *and* high image quality @ low cost

Frame Rate (frames/sec)	Application	Quality	Cost
frame-by-frame	Film CGI	very high	very high
.001 - 1 fps	Video CGI	high	high
interactive	modeling tools	low	medium
5-10 fps	motion capture	low	medium
	data visualization	low	medium
real-time	visual simulations	medium	medium
15-60fps	video games	low	low
	LBE	medium	medium
	broadcast	high	high

TABLE 1. Applications grouped by frame rate requirements

Creative Skills

"Movies did not flourish until the engineers lost control to artists"
-Paul Heckel [Heck91]

In addition to the technical developments, the second enabling element is the formation of creative talent with the knowledge and expertise to produce content for this new medium. The production of a top-notch entertainment experience requires a large set of skills. The team often consists of:

- story/game designers
- CGI animators and modelers
- visual simulator developers

The creation of compelling 3D scenery and characters draws heavily on experience that is found primarily among traditional animators and those working in the industry built around non-real-time computer generated imagery for film and video. These people have the ability to create compelling scenery and bring characters to life. In moving to the domain of real-time graphics, the main challenge is how to live within a limited budget of geometry and texture imagery without destroying the visual effect.

The integration of the many technical elements into a real-time system requires experience from the visual simulation industry which is familiar with the programming and integration of real-time processors, texture-mapped graphics hardware, sound systems, displays, motion platforms, input devices, etc.

The Result:

This merging of new technologies and new talent is facilitating the production of a new form of entertainment, perhaps even a new medium on a par with film or video. For lack of a better term, I'll refer to it using Krantz's: "realies". While overall realism and character quality are still somewhat limited, the interactivity and immersivity alone make it qualitatively different from the media which spawned it. It's characteristics are:

- more realistic than video games
- more story and character than video games
- more interactive than ride films
- more immersive and first-person than film or TV

	Realistic	Interactive	Immersive	Detailed Character
Video Games	No	Yes	No	No
Ride Films	Yes	No	Yes	Yes
Film & TV	Yes	No	No	Yes
3D "Realies"	Yes	Yes	Yes	Yes

TABLE 2. Characteristics of various entertainment media.

3 Platform Hardware and Software

Constructing a complete entertainment system requires many pieces including hardware, software and database. Usually these fall into two sets. The platform components are the low-level building blocks which are not highly specific to any particular game or experience. On top of this platform lie layers of increasing specificity. The next layer might be a software run-time manager that could control the system for a particular class of game experiences. The final layer then would be the content which is specific to a particular game: characters, 3D geometric models, scenery, game logic, behaviors, animations, AI for autonomous characters, in short the game application and its associated content.

Each of these functions requires a hardware component and a software component to provide a useful interface for driving it. Together they provide a common set of capabilities which can be used for running many different games, much as a movie projector provides a platform for the showing of films. Figure

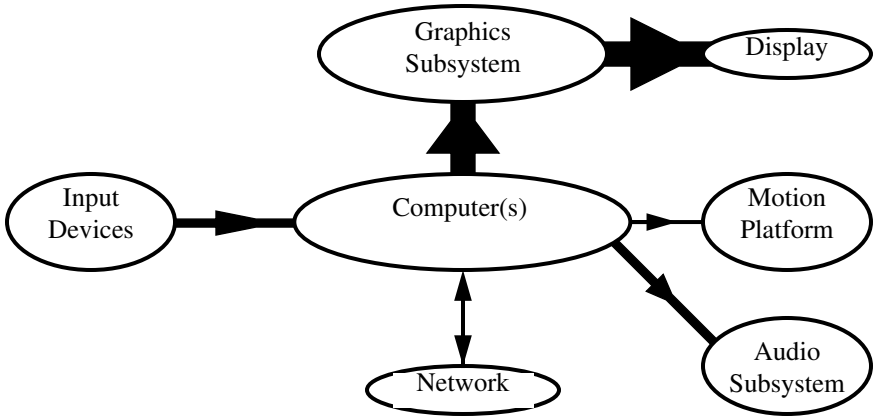


FIGURE 2. Common Hardware Components

2 shows a common set of hardware elements.At the lowest level, the platform can be divided into six areas:as shown in Table 3.

Function	Hardware Example	Software Example
general processing	workstation CPU(s)	Unix™
visual	graphics subsystem CRTs head-mounted displays	IRIS Performer™
audio	MIDI synthesizer	audio drivers
motion	motion platform	dynamics model motion drivers
input	joysticks trackers	device drivers
output	LED displays real bells & whistles	

TABLE 3. Low-level run-time functions and hardware/software examples

Hardware

In entertainment systems, image quality and frame rate are not goals in themselves. It's not about how many antialiased, textured, bump mapped, polygons per second you can draw, or about how many sound sources, diffusers and reflectors your 3D spatial sound system can render. Entertainment systems, whether in a home, arcade or theme park, need to achieve sufficient fun per dollar to pay for themselves either directly or as an attraction.

> *"Super Mario didn't sell millions of copies because the mushrooms were texture mapped" - Tom Garland, SGI*

But it takes certain capabilities to make something compelling and fun. And the standards go up every year. Pong was quite a hit in 1972. But if some neo-

Bushnell were to install a game with similar graphics quality in an arcade today, would he have problems with it shorting out from filling up too fast with quarters? Probably not. And theme park standards must be even higher to warrant the longer distances and greater expense. So there is an as yet unsated thirst for quality which is unlikely to be met until real-time rendering can approach the production quality of CGI for film and video. And judging by the visual quality which current graphics hardware can achieve at real-time rates of 20-30 fps, we've still got a ways to go before we have graphics power to spare.

The table below lists a number of factors which contribute to the quality of the experience and the hardware involved.

Capability	Enabling Hardware
visual complexity	GFX, displays
frame rate	GFX
character animation	CPU + GFX
motion realism	motion platform
audio fidelity	synthesizers, spatializers, speakers
player environment	pod, auxiliary displays, input devices
collision detection	CPU

TABLE 4. Capabilities and the associated hardware which enables (and limits) them

Often, the graphics subsystem is the single most expensive component.

While seeking quality to support richer content and distinguish themselves, developers of entertainment systems are also very sensitive to cost per player. Revenues for these systems typically range from the 25cents/play of arcade video games to the dollar/minute or more charged by many location-based entertainment installations. Total system costs can range roughly from $10,000 to perhaps as high as $100,000 per player. In many of these systems, the computing and graphics hardware are the largest factors in cost.

Unless many players share a single display, as in a large motion cab with a projection screen, these price constraints lie well below the cost per visual channel of traditional visual simulation image generators and are even a stretch for many workstation-based graphics subsystems.

Thus, developers are caught between their hunger for visual quality and a thin wallet, but are constrained by. Nothing except an infinite budget can eliminate the constraint imposed by (EQ 1). Every additional polygon costs something in the bottom line. But careful attention to the design of the visual database and implementation of the run-time system can be used to maximize the visual

impact and frame rate at a particular cost, effectively reducing the proportionality constant in the equation and improving cost/performance.

Architectural Design

Once the performance and content requirements of the system have been determined. The architecture of the hardware system should be designed with three main things in mind:

- processing power - how much does a particular subsystem require?
- bandwidth - how much data needs to get in and out of each subsystem
- latency - what sort of delays are allowable in data generation and transfer

Graphics Hardware

Most graphics hardware capable of rendering reasonably high scene complexity at high frame rates scan convert polygons for rendering, rather than using ray tracing or volume rendering techniques. Architectures in this domain include like RealityEngine [Akeley93] or LEO [Deeri93].

While the fundamental tradeoff is between frame rate, visual complexity and cost, the details of a particular graphics architecture are often important to gaining maximum performance from it. Chapter 9 of these course notes, a reprint of [Akeley89], provides a good introduction to this class of graphics hardware. Depending on the graphics architecture used, one may be limited in the number or size of polygons that can be rendered, the number of pixels that can be drawn, the amount of data transferred between the host and graphics subsystem. Methods for optimizing rendering to these constraints are described in Chapter 4. Some techniques such as texture mapping (discussed in Chapter 2) can dramatically increase the perceived visual complexity without increasing the polygonal complexity of the database.

Host to Graphics Connection

Traditionally one of the largest consumers of bandwidth has been the connection between the visual database and the graphics subsystem. For this reason, many systems, such as the image generators used in visual simulation, have had the database reside in the graphics subsystem itself. This has advantages in that it allows hardware specific optimization of rendering and requires a much lower bandwidth between the host computer and the graphics subsystem (commands such as “move object #15 North 10 meters”). But such optimizations (e.g. binary space partitioning) often place restrictions on the dynamism of the data and because the rendering engine owns the database, any new features, such as character animation, must be coded directly in the rendering hardware rather than in the friendlier host development environment.

Most workstation and many PC graphics systems take a different approach and have the rendering traversal and sometimes the first stages of the rendering performed on the host CPU. This allows more flexibility in rendering, but requires a much larger bandwidth between the host CPU and the graphics subsystem, since every polygon vertex, normal and color must be transferred each frame.

Software

This platform layer consists of the core run-time functionality that could be used by many different games. This level of software should provide a high-performance layer which isolates the game developer from the details of the underlying hardware. Typically, it comes in the form of a toolkits or software libraries layered on top of an operating system.

Real-Time OS?

One approach to achieving real-time performance is to assemble a custom hardware dedicated and perfectly tuned to the task as with VIDEOPLACE which Myron Krueger has developed over the last 18 years. He meticulously "timed software modules with a logic analyzer" to be certain about the performance characteristics [Krueg90]. This is the level of certainty we desire, and predictability and high performance must be designed into the hardware and software platform. But developing a fully custom OS or even utilizing one of the standard "real-time" kernels available for embedded systems requires a very substantial amount of work. The development of hardware graphics drivers in itself would be prohibitive for many projects which are deployed in small numbers.

The fast CPUs, good development environment, flexibility and graphics performance of UNIX workstations makes it an attractive OS for high end game systems. One can prototype, develop, code, model, paint and debug all on the same or similar systems. But is it good for deployment? UNIX has a bad reputation for scheduling and interrupt latency.

In order to achieve consistent and predictable performance on a workstation, one needs to insure that the desired application processing is not interrupted by daemon or ancillary interrupt driven kernel activity. Multiprocessing workstations can provide this functionality by allowing processors to be restricted to certain tasks so that applications and real-time device drivers have guaranteed response times. The REACT extensions to Silicon Graphics' IRIX operating system supports these features.

Other issues in trying to achieve constant frame rates and constant latencies lie in the application domain. Frequently gaming systems are so complex that it may not be feasible to exhaustively simulate all contingencies. To prevent the

system from failing under stress, the platform software should also provide mechanisms for monitoring performance and respond gracefully when overloaded preferably without dropping its frame rate. Most often this involves decoupling various processing tasks and being able to provide extrapolated values when actual information is delayed.

Synchronization and Communication

Sometimes the most cost effective solution requires having a single powerful computer driving the experiences of multiple players. For example, a high-end, multiprocessing workstation could drive six game pods playing the same or different games as shown in the left side of Figure 3. In this case, synchronizing the different players is a trivial matter of communication through shared memory.

But when multiple systems need to be networked to connect players into a common game or environment, as shown in the right side of Figure 3, synchronization is required to ensure that the

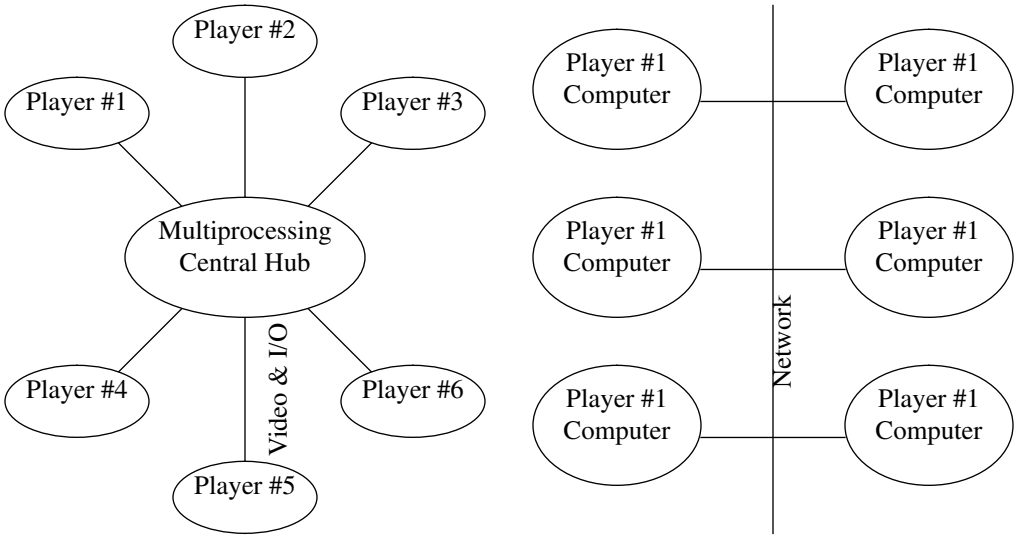


FIGURE 3. Centralized vs. decentralized systems

global state (e.g. locations and conditions of players) for the game is the same for everyone. This becomes complicated when the network connecting the systems has high latencies, and is particularly a problem for fast-paced games in which objects are moving rapidly. Maintaining accurate information about player location requires all systems to have an accurately synchronized notion of time and a method for extrapolating position information. and correcting for extrapolation issues such as the paper [Sing94] reprinted in Chapter 10.The accuracy of synchronization in multiplayer simulation is largely a function of

network bandwidth and latency. Depending on requirements and budget, solutions range from relatively high-latency connections such as modems over telephone lines to low-latency reflective shared-memory systems such as SCRAMNet (as used in the CAVE [Cruz93]).

When multiple machines drive the same display (e.g. panoramic, multiprojector display), very fine synchronization is required. Typically this is done through synchronizing the video clock and buffer swap mechanisms of the different systems rather than through a standard network.

Domain Specific Software

For example, on the graphics side, one might use a toolkit such as IRIS Performer to provide a graphics and database processing, as well as a framework for multiprocessing. But a complete system requires comparable low-level software for handling input devices, networking, audio, dynamics, collision response, and so on. Some higher-level platform software covers more than one of these areas (e.g. Paradigm Simulation's Vega and Division's dVS), but currently no single solution covers the entire set of needs listed above. Consequently, a number of different software suppliers and a fair amount of custom coding are often used.

4 Artistic Content

The other major component is the content of a particular game, which in the projector analogy corresponds to what's on the film being projected. In many ways, producing a game is like producing a film. First and foremost comes the concept, plot, characters, scenarios and game activity. Jordan Weisman, the creator of Virtual Worlds Entertainment, designed role playing board games for many years before venturing into location-based entertainment. Game designers have much more experience to draw on today than when Chris Crawford developed Excalibur at Atari a decade ago, but his comments on the state of the art still sound true, especially as advances continue to change the design medium and range of possible content.

> *Computer games constitute a new and as yet poorly developed art form that holds great promise for both designers and players.*
>
> *-Chris Crawford [Craw84]*

For the most part, these issues lie outside the graphics and engineering focus of this course. Some aspects of the design process are covered in later chapters in this course. Crawford's book [Craw84] and section in Laurel's book [Laurel90] give many practical insights for designing computer games. [Laurel91] and [Thom81] also provide essential background material. In his review of PC,

CD-ROM and video game technology[Morr94], Morrison also briefly describes the design considerations in the making of several current games. Moses Ma, the designer of Spectre VR, a 3D simulation game for PC and Macintosh, sums up his ideas on game design as follows:

Moses' Ten Commandments of Game Design

1. High Concept: A story that's hot, original, obvious, leading edge on technology, emotional
2. Barriers: Non-limiting, challenging — like flying through closing barn doors. "Difficult landings", try thread the needle, barriers *must* move and change
3. Evolving Enemies: Interesting characterizations, show inner conflict! The enemy keeps you from getting what you want. Interaction on 3 levels: world, personal, inner.
4. Dazzling Graphics: I mean really... Is it breathtaking? Is there a consistent cinematic image system? Does it have a look?
5. Emotion producing sound.
6. Action well-timed and rising/falling action and boredom detection - 20 minute episodes — sweat drenching — opening hook, conflict, crisis, resolution.
7. Tight registration of feedback: Both collision detection, joystick and visual meter feedback.
8. Positive Monitoring in Learning Curve: Use a flight log on data disk, easy to learn / a challenge to master.
9. Infinite Replayability: Create data disk format/ flightlog - levels of skill.
10. Make the gaming simulation world real, authentic (not cliche). Interesting characterizations! Know the world, details count.

Implementation

After moving from concept to detailed design, next comes the implementation. As with a film, sets must be constructed and populated with characters. This requires the modeling of geometry, the generation of imagery for texturing and the specification of the data necessary to animate the characters. Because this medium is new, there is a relative scarcity of production tools.

Component	Examples
visual database	geometry, texture, animated models
audio database	samples, data for synthesis
behavioral models	animation control, collision response, AI players

Component	Examples
visual database	geometry, texture, animated models
motion platform	motion models
user interface	interface models
game logic	scenario handling
application	overall control, story, coordination, continuity

Geometric Data

Typically the geometric database is constructed with a 3D modeling tool. The desired output is a collection of polygons suitable for real-time rendering. Commercial modeling tools currently tend to fall into two categories, modelers with origins in visual simulation and those with origins in non-real-time computer animation. Visual simulation modelers tend to focus on supporting the construction of objects polygon by polygon or through terrain height fields, combined with lofting, rotation and extrusion. These methods of construction are appropriate for real-time systems since polygon count must be minimized to achieve reasonable frame rates. Such modelers also provide real-time features such as level-of-detail switching and precomputed animation sequences as discussed in Chapter 2.

Modelers used for producing frame-by-frame animations usually have a richer set of surface construction tools, but often lack real-time concepts such as ranges for level of detail switching and prerendered animation sequences. Higher level primitives such as spheres, cylinders and NURBS are powerful but can be dangerous for real-time systems. While they make the construction of multiple levels of detail much easier, they also make it easy to generate far to many polygons for real-time rendering.

Image Data

Image data can be produced by many different means: photographs of objects or hand paintings, use of computer paint tools, procedural generation, etc. When game is based on an existing property such as a film or TV show, much of the image material may already exist. This image data is then mapped onto geometry, usually using the same tools that were used to generate the geometry.

Animation & Motion

Introducing dynamism is the great challenge in designing content for real-time systems. A deserted scene in which nothing moves or changes probably won't make a good game. Defining paths for object to move through a scene is quite straightforward as is specifying jointed articulation of static elements such as the motion of a crane or the wheels on a car. But complex animations require

more than this. Traditionally in visual simulation, such animations have been modeled as flip-card sequences of preanimated models. Flip-card animation sequences eliminate the run-time computational load of complex animations. The next step up is to use geometric morphing to interpolate between steps in the sequence. One is still limited to the prerendered sequencing, but at least the motion can be smooth and existing animation packages can generate the sequences of models.

Total animation of a character requires both articulation and geometric morphing as discussed briefly in Chapter 2. Depending on the type of animation, some combination of key frame animation, live motion capture, goal-directed motion [Badl91] and dynamics modeling can be used. Since deGraf and Wahrman showed "Mike the Talking Head" at SIGGRAPH '88, a growing amount of work has been done on generating character animations at interactive frame rates. But the computational and graphics requirements make this difficult, and commercial tools for translating animations into forms suitable for real-time rendering remain scarce.

Some types of geometry and imagery are well-suited for procedural generation and animation [Ebert92][Prus90]. Extensive procedural generation of geometry (e.g. fractals, plants) and texture images is rarely done at run time because the large computational burden required. With a few notable exceptions such as Pixel-Planes 5 [Fuchs89], most graphics hardware systems do not accelerate procedural texture generation.

So, most procedural animation used in real-time systems takes much simpler forms such as defining a mathematical model for the motion of waves. Such models usually end up being defined in application code rather than in the database partially for efficiency, but mainly due to lack of tools and database formats to support them.

Special Effects

> *"Nintendo informed Jaleco that the exploding hamster had to be deleted in future cartridges." - David Sheff [Sheff93]*

The creation of special effects whether sparkling pixie dust or something more visceral, often falls outside the capabilities of standard modelers. Often some combination of modeling and procedural generation at run-time are used. The run-time capabilities available for use include flip-card animation of textures and geometric models, morphing, texture coordinate animation *and simple* particle systems.

5 The Director

The tar pit of software engineering will continue to be sticky for a long time to come. - Fred Brooks [Brook82]

We've covered two obvious elements: artistic content and the platform for showing it on. Using the film analogy, on one side we have the actors, the set and the script and on the other we have the projector, or at least a motor, bulb and lens. Unfortunately, the technology for making a "realie" is nowhere near as well developed as film or video. Each developer must put together their own system for bridging the gap between the artistic content and the platform. Several years or more may elapse before any standard solutions are settled on. Until then, one of the major challenges of assembling a system is writing this software director in a way that it can be reused for different experiences.

This "Director" software is the run-time engine that coordinates all activity. On one side the director is soaking up information from all of the input devices, and following the script, tells the camera operator how to shoot, the audio system what to say to the player, the characters how to move.

One problem is the script. It appears to have no simple representation. It covers a large range of activities: gaming logic, autonomous characters, complex animations, scenery shifts, scenario tuning based on player's responses, collision response, etc. How can one hope to embody something so ill-defined? Software, of course. It's the programmability of our machines that makes rapid progress possible even in the face of problems on the scale of trying to create virtual theatres and gaming worlds.

The topic of "Director" software and other elements of game construction are covered in Chapters 5-7.

6 Conclusions

Never mistake a clear view for a short distance. - Paul Saffo [Saff90]

The current situation as far as tools for creating content, hardware for producing the sound, motion and graphics, and the software for integrating the content to the hardware indicates that this industry is still in its infancy. Modeling and animation tools do not yet fully reflect the requirements for producing scenery and characters that can be rendered in real-time. With large gains every year, graphics hardware is now able to produce much more compelling scenes than were available to the first developers of computer games. But it still falls far short of being able to unable to render everything that our artists would like to at the real-time frame rates required for interactive gaming and theatre. And

while software platforms are improving along with the hardware, there are as yet no standard or fully reusable run-time “projection” systems.

No doubt the available tools and software will improve as the market and our understanding of the problem improve. Until then, those wishing to create such games or experiences have a lot work to do ranging from building their own sets, cameras, and even projectors.

As with other entertainment media, the most important element lies in the design of compelling and feasible concept. Beyond that, the largest problem is trying to squeeze the execution of that concept into the 15 to 50 milliseconds we have to think about and render each frame.

Graphics is currently one of the most expensive elements in the run-time system, and nothing can save developers from the quality vs. frame rate vs. cost tradeoff embodied in (EQ 1). But if the entire game design process from modeling and animation to the run-time software proceeds knowing that every polygon, every character animation and every special effect costs something in the bottom cost/performance line, we can at least shift the balance in favor of more visual impact. This becomes more important as the producers of these new entertainment systems seek to distinguish themselves. Chapters 2-4 of these notes discuss various software and modeling methods for use in this effort.

7 Bibliography

[Akeley89] Kurt Akeley, "The Silicon Graphics 4D/240GTX Superworkstation", *Computer Graphics and Applications,* Vol. 9, No. 4, July 1989, pp. 71 - 83.

[Akeley93] Kurt Akeley, "RealityEngine Graphics", *Proceedings of SIGGRAPH '93*, July 1993, pp. 109-116.

[Appi92] P.A. Appino, J.B. Lewis, L.Koved, D.T. Ling, D.A. Rabenhorst, and C.F. Codella. An architecture for virtual worlds. *Presence*, 1(1), Winter 1992, pp. 1-17.

[Badl91] Norman Badler, Brian Barsky and David Zeltzer, eds. *Making Them* Move: Mechanics, Control and Animation of Articulated Figures. Morgan Kaufmann, San Mateo, California, 1991.

[Brook82] Frederick P. Brooks, *The Mythical Man-Month*. Addison-Wesley. 1982.

[Craw84] Chris Crawford. *The Art of Computer Game Design*. Undated manuscript. 1984(?).

[Cruz93] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. "Surround-screen projection -based virtual reality: The design and implementation of the cave," *Proceedings of SIGGRAPH '93*, July 1993, pp. 135-142

[Deeri93] Michael F. Deering, "Leo: A System for Cost Effective 3D Shaded Graphics", *Proceedings of SIGGRAPH '93*, July 1993, pp. 101-108.

[Ebert92] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin. *Pro*cedural Modeling and Rendering Techniques. Course Notes. SIGGRAPH 1992.

[Fuchs89] Henry Fuchs, et. al. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System. Using Processor-Enhance Memories. *Proceedings SIG*GRAPH '89. Computer Graphics 23(3), July 1989, pp. 79-88.

[Heck91] Paul Heckel. *The Elements of Friendly Software Design*. Sybex, San Francisco, 1991.

[Kranz94] Michael Krantz. Dollar a Minute. *Wired* 2.05, May, 1994.

[Krueg90] Myron W. Krueger. Simulation versus artificial reality. In E.G. Monroe, editor, Proceedings of the 1990 Image V Conference (Phoenix, 19-22 June 1990), Image Society, 1990, pp. 147-155

[Laurel90] Brenda Laurel, ed. *The Art of Human Computer Interface Design.* Addison-Wesley, Reading, Massachusetts, 1990.

[Laurel91] Brenda Laurel. *Computers as Theatre*. Addison-Wesley, Reading, Massachusetts, 1990.

[Morr94] Mike Morrison. *The Magic of Interactive Entertainment*. Sams, Indianapolis, Indiana, 1994.

[Prus90] Przemyslaw Prusinkiewicz. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[Saff90] Paul Saffo, “The Zen of Information Technology”, EE380 Seminar, Stanford University, November 28, 1990.

[Sheff93] David Sheff. *Game Over*. Random House, New York, 1993.

[Sing94] Sandeep K. Singhal and David R. Cheriton, Technical Report: STAN-CS-TR-94-1505, Stanford University, Computer Science Dept., February 1994

[Thom81] Frank Thomas and Ollie Johnston. *Disney Animation: The Illusion of Life*. Abbeville, New York, 1981.

Appendix A. Performance Requirements and Human Factors

A 1 Visual Perception

Of all the sensory information we need to provide, the visual component is certainly the most important in producing the illusion of a virtual environment. The ideal would be to provide simulated input with fidelity which matched or exceeded the limits of human visual perception in all aspects. Fortunately, in practice, this is not a prerequisite for producing a usable virtual environment. But in designing a system, we need to always keep the capabilities of the human visual system in mind because the limitations of technology will force many tradeoffs to be made for the foreseeable future.

The human visual system is complex and even basic perceptual thresholds defy simple characterization, usually depending on a variety of factors. The following briefly discusses some of the perceptual limits and how they have been addressed in traditional simulators.

Visual Acuity

Spatial resolution limits can be measured in many ways and depend strongly on many factors including brightness, color, contrast, off-axis eccentricity, length of exposure and retinal illumination.

Visual acuity is commonly measured in terms of the angle subtended at the eye. For reasonably bright objects, on-axis, the limit is around 1 minute of arc. This corresponds roughly to a 20/20 result in a standard vision test, which indicates the ability to recognize letters which subtend 5 minutes of arc. Sensitivity

to spatial frequencies is measured by the contrast required to perceive vertical bars and is largest at around 5 cycles/degree [Buse92]. This indicates that resolutions of around 5 minutes of arc are required to get into the region of peak sensitivity, a limit important for creating sharp edges.

Acuity falls off rapidly as the object moves outside the central 2 degree region. At 10 degree of off-axis eccentricity, acuity drops to around 10 arcmin.

For comparison, a first generation LCD-based head-mounted display (HMD) with around 185 RGB triads across an 75 degrees field of view per eye [Robi91] yields a resolution of around 24 arcmins, or around 20/480 vision. To make resolution matters worse, typically optical blurring was often employed to help fuse the red, green and blue pixels [Teit90].

Even a “high-resolution” display with 1280 pixels across a narrower 60 degrees field of view per eye achieves only around 3 arcmins.

Standards set for visual simulators are valuable guideposts because they embody years of experience. In this case, the standard for out-the window commercial flight simulators (Level C) set by the Federal Aviation Administration (FAA) requires 3 arcmin resolution [FAA89].

Temporal Resolution

Peak sensitivity to temporally modulated illumination occurs around 10Hz to 25Hz, with the frequency increasing with luminance. The frequency at which modulation is no longer perceptible, the *critical flicker fusion frequency*, varies from 15Hz up to around 50Hz for high illumination levels [Wysz82]. On CRT displays, which fill a large field of view and have non-sinusoidal temporal profiles, some individuals still perceive flicker at the common 60Hz refresh rate, which has lead workstation manufacturers to introduce video formats in the 66Hz to 76Hz range. Bright displays with large fields of view can require refresh rates of 85Hz or more [Padm92].

Luminance

Including dark adaptation the eye has a dynamic range of around 7 orders of magnitude, far greater than any current display device. The eye is sensitive to ratios of intensities rather than absolute differences, and at high illuminations the eye can detect differences in luminance as small as 1% [Wysz82]. Thus a CRT with a dynamic range of around 100 can display no more than log100/log1.01 = 463 perceptible levels.

For reference, Padmos indicates that contrast ratios of 10:1 to 25:1 are sufficient. The FAA standard for commercial flight simulators requires a 5:1 contrast ratio for scenery and 25:1 for light points [FAA89].

Color

The human eye can perceive light in the range of 400nm to 700nm in wavelength. Fortunately for the creators of simulated environments, the human eye can't perceive the exact spectrum of light emanating from an object. According to the tristimulus theory, our perception of color starts in three different types of receptors on the retina. Each type of color receptor has a different spectral response with a single dominant peak. The resulting three-dimensional color space can be mapped in many ways.

The CIE chromaticity space factors out luminous energy to yield a two-dimensional color gamut which serves as a benchmark tristimulus based color generation. Most reproduction methods whether printing inks, film layers or phosphors used in CRTs, only cover a portion of the color gamut [Fole90].

But the tristimulus-based color gamut is not the final word. Colors are perceived differently depending on their context, for example we perceive an apple to have the same color even under a variety of illuminations. This same adaptation that allows colors to "look right" under varying illumination also tends to cause limited color ranges to appear richer than we would expect from the tristimulus theory. Land in his retinex theory [Land83] showed that stimulation by a combination of only two spectral sources can give the impression of a surprisingly wide range of colors. This could be relevant some displays such as the two-color version of the BOOM [McDo90].

Stereopsis and Depth

The limit of stereo vision typically occurs for a binocular disparity of 12 arcsec which translates into perceiving the depth ordering of objects separated by 0.1cm at a distance of 1m, 9cm at 10m, and 56cm at 25m [Buse92].

When looking at computer-generated imagery, the eyes' focus (accommodation) and vergence often do not match as they must focus at the screen or the image plane defined by the optics of an HMD but converge at an angle dictated by the rendered images. And without proper calibration or in a monoscopic system such as a dome, neither focus nor convergence may reflect the actual position of the virtual object relative to the viewer. This causes errors in accommodation (defocus) and vergence (fixation disparity) [Hung94]. Of even greater concern in systems intended for public use is the possibility that oculomotor problems can persist after participation in a virtual environment as a form of simulator sickness.

Out-the-window simulators often use collimated optics to place the images at infinity thereby making convergence and focus match closely for distant scenery.

Many users of stereoscopic systems have trouble fusing stereo images, perhaps because of these inconsistencies. Computer graphics applications which do not need to accurately depict the scale of depth can artificially adjust the parallax to allow more comfortable viewing [Hibb91].

But for virtual environments requiring close-up manipulation of objects and especially for those requiring accurate registration of virtual objects with the real world, the false depth cues generated by this intentional decalibration are unacceptable. The best that can be done is to calibrate and consider all variables affecting stereo viewing [Deer92] and carefully choose the size, overlap and image distance of the system to match the task and operating distance.

The blurring of objects due to depth-of-field effects poses another challenge. Depth of field can be rendered using multi-pass techniques [Haeb90]. But even if methods for measuring or inferring eye accommodation existed, the 2X to 4X performance penalties for simulating depth of field are probably unacceptable. Real-time holography solves this particular problem, but brings a few too many of its own.

Field of View

Each eye has approximately a 150 degrees horizontal field of view (60 degrees towards the nose and 90 degrees to the side) and 120 degrees vertically (50 degrees up and 80 degrees down) [Buse92]. The FAA standard for commercial out-the-window flight simulators (Level C) requires 75 degrees horizontal and 30 degrees vertical fields of view [FAA89].

It's important to remember that visual acuity limited to only a few degrees around the axis of gaze direction. Whether in a head-mounted display or a projection system, vast amounts of rendering power are wasted drawing high resolution imagery where you can't see it, because you're looking at something two or more degrees away.

Binocular overlap when focused at infinity is approximately 120 degrees. Overlap varies substantially among different HMDs binocular with some supporting variable overlaps. With the relatively small fields of view (40 degrees to 60 degrees of HMDs used in simulation), large overlaps of more than 50% have been found useful. Because of other problems such as blending of brightness at the borders of overlap, one report found that 100% overlap produced best performance on a visual search task [Edga91].

Motion

Motion of bright objects can be perceived down to 0.3 min arc/sec [Buse92]. In out-the-window simulators, the maximum displacement of an object per update which gives an impression of continuous motion is 15 arcmin

[Padm92]. If carried directly over to HMD usage, a head slew rate in excess of 180 degrees/sec translates into an unattainable 720 Hz. At such high motion rates, temporal antialiasing, i.e. motion blur, could help simulate the temporal averaging of the visual system, but the appropriateness of motion blur for an object also depends on whether the gaze is fixed on the moving object.

Motion of the visual field causes a sense of motion even without corresponding physical body motion. For example, in a simulator without a motion platform the participant gets an impression of self-motion from the motion of objects on the screen.

For rotations, this visually-induced motion varies depending on the axis and the physical orientation of the subject, presumably because of conditioning by gravity. The sense of rotation occurs for roll, pitch and yaw, with sense of yaw being the strongest and roll the weakest [Howa87]. At very high rates of change in yaw, the sense of motion begins to saturate at 60 degrees/sec [Mooi87].

This visually-induced sense of motion is tied to field of view becoming “effective” at around 60 degrees and most effective at 180 degrees[Mooi87].

This sense of motion is accompanied by a perception of a change in orientation, even in the absence of a physical change in orientation. A sense of change in orientation occurs in the opposite direction of the rotation of the visual image and can range up to 45 degrees [Howa87].

A 2 Temporal Artifacts In Simulated Displays

Field-Sequential Artifacts

Ideally, each pixel should be accurate for the moment it is scanned out on the display device including motion blur (temporal antialiasing) for the period between screen refreshes. In practice this is not feasible and would also unrealistically blur moving objects which the gaze is fixed on. For most applications, the performance penalties for doing full scene motion blur outweigh the benefit, and brute force per-pixel temporal alignment is several orders of magnitude more costly. The bottom line is that relatively few applications can even update the screen at the display refresh rate, let alone worry about these higher quality issues.

Repeating Frames

Most displays run at 60Hz or higher field refresh rates to prevent flicker, but many visual simulations run at lower update rates. This means that a single

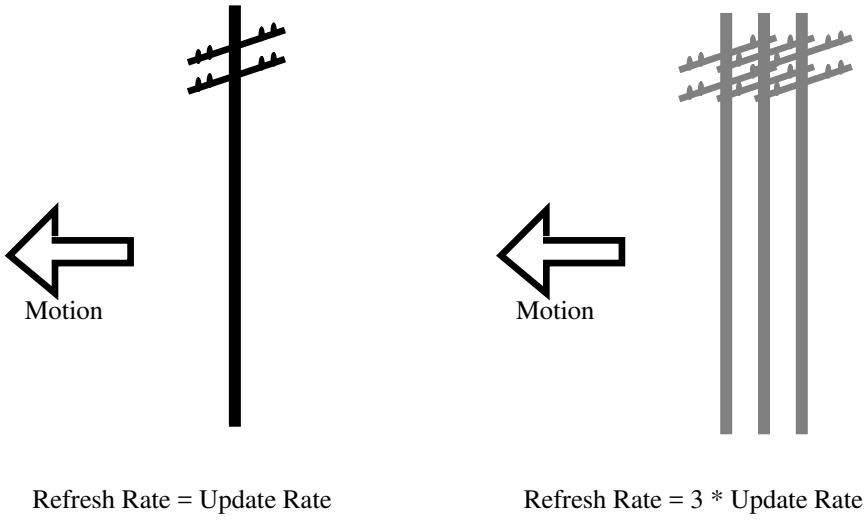


FIGURE 1. Multiple image artifact when fixating on moving object.

image may be scanned out several times before being changed. The sense of motion at the lower update rate is not as smooth, especially when imagery is moving rapidly. In addition, the repetition of a frame means that the image is temporally inaccurate for motion. Real moving objects do not stay in one place for a couple frame times and then move. The result is that when one fixates on a moving object, it appears to split into multiple copies along the direction of motion with the number of ghosts equaling the refresh rate divided by the update rate. So a simulation running at 20Hz update on a display refreshing at 60Hz, the object will appear tripled as shown in Figure 1. On large objects such as horizon silhouette, the effect manifests itself as multiple edges.

Motion blur would mitigate this effect, but poses other problems. In practice, smooth motion and the absence of ghosting are best achieved by an update rate which equals the display refresh rate.

Interlacing

One way to make the update rate equal the refresh rate is to interlace the video so that even and odd line fields are drawn at 60Hz. The graphics hardware can then render the scene with at 30Hz. This suffers from problems similar to those above. But in this case, the temporal inaccuracy for motion causes combed ghosting with edges breaking up into combs due to the interlacing.

Another option is to render with half-resolution at 60Hz (just the even or odd field lines). This has some advantages since rendering latency is decreased by 1/60th sec, motion is smoother and combed ghosting is reduced. As this requires rendering all the geometry in the scene twice, it is only possible for

scenes which are strongly limited by pixel fill rates. But many highly-textured databases with simple geometry falls into this category.

Sequential Stereo

Most common CRT-based stereo viewing systems operate by sequentially presenting left and right eye images. For moving objects the common approach of rendering both the left and right eye with the same positions for moving objects is temporally incorrect and produces visual artifacts, including erroneous stereo depth cuing in low refresh rate systems. Lipton reports that the artifacts which are noticeable at 30 fields/sec/eye are not perceptible at updates of 60 fields/sec/eye[Lipt91].

Sequential Color

Color monitors typically have lower contrast than monochrome monitors because of the shadow mask. Shadow masks also make the manufacture of small, high-resolution color CRTs extremely difficult. Thus a system which uses a monochrome monitor to sequentially displaying each color channel with an accompanying change in color by a shutter has advantages for use in head-mounted displays over low-contrast LCD displays. However, the sequential display of color also generates artifacts. For example, a monochrome monitor may be driven at a field rate of 180Hz thus creating a full RGB image at 60Hz. But if the frame buffer is only updated at a rate of 60Hz, the positions of moving objects will only be correct for one of the three color scans. This leads to an effect similar to poorly registered color printing plates, especially when the eye is tracking an object of high contrast. Updating the frame buffer at 180Hz might alleviate this somewhat, but few if any graphics platforms are configured to drive visuals at this rate, and a factor of three increase in the required rendering is likely to be unacceptable.

Frame Rate Variations

The aforementioned discussions assume a constant frame rate. Varying frame rates pose a number of additional perceptual and practical issues. Fixed frame rates are required in the design of most visual simulators.

A varying frame rate tends to distract the user from the task at hand, particularly tasks involving manipulation of the world or accurate perception of velocities. In particular when going from 60Hz update to 30Hz update, the factor of two is quite noticeable in the quality of motion. Frame rate changes can also false training, if for example a trainee relied on a change in frame rate as an indication of some otherwise hidden feature, such as a large installation hidden behind a hill.

Unanticipated frame rate variations also cause temporal inaccuracies, because the frame does not appear at the time for which it was planned and all visual latencies through the system change as well.

A 3 Latency

Latency is the time measured from the setting of an input until the corresponding output is manifested. Many factors contribute to latency: input devices, software architecture, rendering time, motion response. Different portions of a system may have different latencies, e.g. the response of the visuals to changes in eye point might differ from the collision response which might in turn differ from the response of the motion system. Any path from an input to an output in Figure 2 could take a different amount of time.

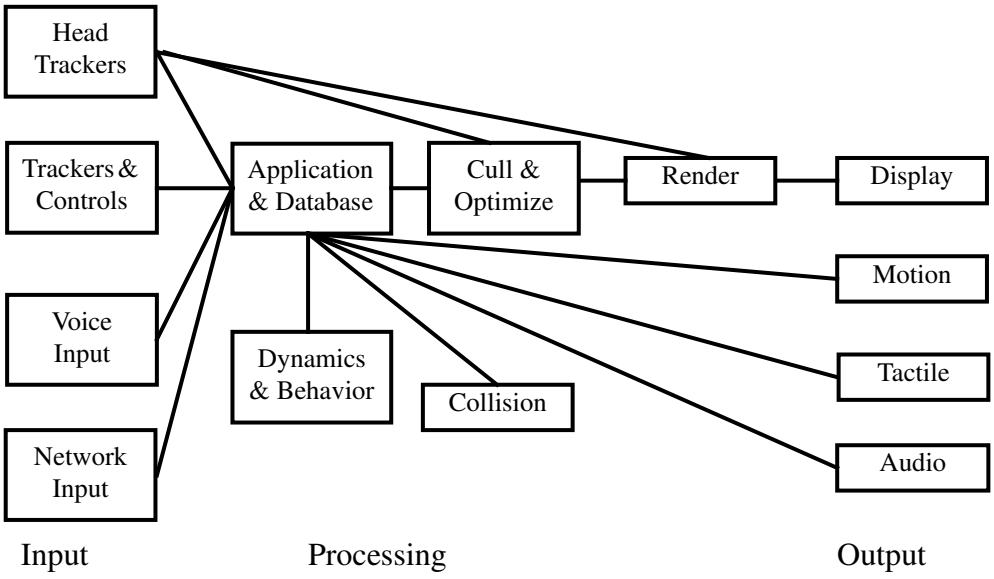


FIGURE 2. Sources of latency in a hypothetical virtual environment system.

For the rendering portion of the system, the latency is typically taken as the time after some value, such as the eyepoint, is set until the last pixel of the corresponding frame is scanned out by the display device.

Typical Simulator Latencies

Reports from a flight training system indicate that the user-perceived quality of the simulation degrades steadily for total latencies over 100ms [Beck92]. Other studies have indicated that latencies below 100ms have little effect [Card90]. According to Padmos, in out-the-window simulators, latencies typically range

from 40-80ms for driving simulators to 100-150ms for low-maneuverability flight simulation [Padm92]. The FAA latency requirement for commercial flight simulators (Level C) is 150ms [FAA89].

Latencies in the 200-350ms range significantly decrease human response rates at measured by transfer functions in the frequency domain [Hess83]. As is well known from aircraft control theory, this can lead to over-compensation and oscillations induced by the operator.

A 4 Simulator Sickness

The term *simulator sickness* refers to a variety of maladies that result from the use of simulators, both with and without real body motion. The prevalence of simulator sickness in out-the-window simulators indicates that it could seriously impact the usage of VEs. If the first virtual environments to enter mainstream use make users sick, it could damage their acceptance into real-world use.

Virtual Environments vs. Simulators

Virtual environments share many attributes with traditional simulators:

- A visually-induced sense of motion either without corresponding physical accelerations or imperfectly simulated by a motion platform.
- Latencies between events or actions and the manifestation of their results whether visual, auditory or physical.
- Displays with wide fields of view.
- Visual artifacts from inadequate temporal and spatial aliasing.
- Visual artifacts due to inadequate frame rates.
- Display resolutions which at best barely meet human resolution limits.

While most virtual environments may not involve the sort of gut-wrenching maneuvers which fighter pilots go through, VEs offer other challenges not common in traditional visual simulation:

- The slew rates for HMD use are substantially larger than those encountered on most out-the-window visual simulators.
- In HMDs, fixating while turning the head poses latency and update challenges greater than in out-the-window simulation.
- Head-tracked displays are subject to errors in 6 degree-of-freedom trackers.

Many virtual environments involve viewing objects up close. In the real world, such viewing is accompanied by focus distance and depth-of-field effects which are difficult to simulate. Out-the-window applications can often closely simulate the optical configuration of physical world by using collimated optics to place the imagery at infinity.

Proposed uses of virtual environments span many diverse population groups. Unlike pilots these groups are unlikely to have become accommodated through extensive simulator usage and have not gone through a winnowing process which strongly selected for resistance to motion sickness.

Simulator sickness is surprisingly common with 20% to 40% of fighter pilots using simulators suffering ill effects. Some simulators have incidence rates of up to 87% [Mone91b].

This despite the facts that:

- Fighter pilots are a population group highly selected for resistance to motion sickness, and are better accommodated to simulator use than the general public.
- The simulators have stringent human factors requirements
- Most simulators are out-the-window, i.e. CRT, projection or dome based, so the visuals need only track the vehicle motion, and not rapid head motion.

Symptoms

While nausea is a significant and perhaps most obvious feature of motion and simulator sickness, it is not the only or even the principal symptom [Kenn90]. Symptoms include visuomotor dysfunctions (e.g. eyestrain, blurred vision, difficulty focusing), mental disorientation (e.g. difficulty concentrating, confusion, apathy) and nausea (e.g. stomach awareness, nausea, vomiting). Drowsiness, fatigue, eyestrain and headache are among the more common symptoms [Kenn87]. In pilot training, one of the main concerns is the persistence of some of these effects for many hours, requiring a recovery period before the pilot is ready for actual flight.

Causes

The lack of much formal research in HMD visually induced motion sickness means we must mostly extrapolate from military and commercial simulators both with and without real body motion.

It’s generally agreed that simulator sickness has two prerequisites, a functioning vestibular system and a sense of motion [Hett92]. The vestibular system is the set of canals, tubes and sacs in the inner ear give us our sense of orientation

and acceleration. Individuals without functioning vestibular systems do not exhibit simulator sickness either with or without real body motion [Eben92].

One hypothesis is that simulator sickness arises from a mismatch between visual motion cues and the vestibular system. This would commonly occur when visual motion does not match physical motion either because no motion platform is used or because the motion platform lags the visuals and cannot match all the accelerations and orientations. In virtual environments, simulator sickness can be expected both in motion based systems (e.g. game pods) and physically static ones (e.g. HMD user seated in a chair).

Wired for Cookie Tossing

Why did this unfortunate response to simulated experience develop? Evolutionarily, it's postulated that in nature the disruption and inconsistency of perceptions which trigger simulator sickness would likely be the side effect of ingesting poisons, and so vomiting is a useful response [Mone91a].

Contributing Factors

Since a sense of visual motion is required for simulator sickness, it's reasonable to expect that many features which make a virtual environment convincing will contribute to simulator sickness. This is born out by studies which have found that bright imagery is more likely to induce sickness than night time scenes, and that wide fields of view cause more problems than narrow ones. Also domes projection systems seem to elicit fewer symptoms than CRT based systems. In motion systems, roll in the 0.2Hz region is particularly nauseogenic.

One author proposes a set of workarounds for pilots using simulators [Mone91b]. Some of these bits of wisdom are paraphrased here for application to virtual environments:

- Don't suggest to users that they will get sick or let them see someone else vomiting. It's contagious.
- Don't go into a VE when your are hung over or have an upset stomach.
- Adaptation is the best fix. Do the VE every day.
- Don't do the real thing the same day you do it in a VE.
- Get set before turning the VE on.
- Set the VE up for night flying.
- Don't roll or pitch too much.
- Don't move your head too much.
- Turn off the VE before getting out.

Until further research is done, we won't know how much additional problems are caused by inadequate resolution, frame rate and latency in HMDs or to what extent BOOM mounted displays [McDo90] are affected. The initial response to the CAVE [Cruz93] do not indicate that surround scene projection systems with standing observers are particularly prone to inducing simulator sickness.

But since problems occur even in visual simulators with good visual quality and no head tracking issues, e.g. dome projection systems, we will always be confronted with the prospect of simulator sickness in some types of virtual environments, particularly those with large visually induced motions.

A 5 Conclusions

The human factors requirements of head-mounted displays are in many ways harder to meet than those for traditional out-the-window visual simulation. Many virtual environments today are far from meeting even the standards for traditional out-the-window simulators, which themselves still suffer from many human factors problems. So as virtual environments become more convincing to our senses, it's reasonable to expect that the problems arising from lags and inconsistencies in the sensory input provided by virtual environments will cause significant problems in their use in the real world, particularly in fast-paced applications such as games and entertainment.

While we have some rough guidelines to work with, much more research and actual experience are needed before we will know how to design systems that will not potentially render a significant fraction of their users sick or disoriented. Until that time, careful attention to the design and extensive testing are a prerequisite to the fielding of any virtual environment.

A 6 Bibliography

[Beck92] P.Beckett. Effective cueing during approach and touchdown - comparison with flight. In *Piloted Simulation Effectiveness AGARD Conference Proceedings 513* (Brussels, 14-17 Oct 1991), pp. 30.1-30.11,

[Buse92] P.Buser and M.Imbert. Vision. MIT Press, Cambridge, Massachusetts, 1992.

[Card90] F.M. Cardullo. Virtual system lags: The problem, the cause, the cure. In E.G. Monroe, editor, *Proceedings of the 1990 Image V Conference (Phoenix, 19-22 June 1990)*, Image Society, 1990, pp. 31-42.

[Cruz93] C.Cruz-Neira, D.J. Sandin, and T.A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. *Computer Graphics Annual Conference Series '93 (Proceedings of SIGGRAPH '93)*, August 1993, pp. 135-142.

[Deer92] M.Deering. High resolution virtual reality. *Computer Graphics (Proceedings SIGGRAPH '92)*, 26(2), August 1992, pp. 195-202.

[Eben92] S.M. Ebenholtz. Motion sickness and oculomotor systems in virtual environments. *Presence*, 1(3), Summer 1992, pp. 302-305.

[Edga91] G.K. Edgar, K.T. Carr, M.Williams, J.Page, and A.L. Clarke. The effects upon visual performance of varying binocular overlap. In *Helmet Mounted Displays and Night Vision Goggles, AGARD Conference Proceedings 517* (Pensacola, Florida, 2 May 1991), pp. 8.1-8.15.

[FAA89] Airline Simulator Qualification. Advisory Circular 120-40B, Federal Aviation Administration, May 1989.

[Fole90] J.D. Foley, A.van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Massachusetts, 1990.

[Haeb90] P.Haeberli and K.Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (Proceedings SIGGRAPH '90)*, 24(4), August 1990, pp. 309-318.

[Hess83] R.A. Hess. Effects of time delays on systems subject to manual control. *Journal of Guidance, Control and Dynamics*, 7(4):416-421, 1983.

[Hett92] L.J. Hettinger and G.E. Riccio. Visually induced motion sickness in virtual environments. *Presence*, 1(3), Summer 1992, pp. 306-310.

[Hibb91] E.A. Hibbard, M.E. Bauer, M.S. Bradshaw, D.G. Deardorff, K.C. Hu, and D.J. Whitney. On the theory and application of stereographics in scientific visualization. *In Proceedings of Eurographics '91 (Vienna, Austria, 2-6 Sept 1991)*, pp. 1-21.

[Howa87] I.P. Howard and B.Cheung. Influence of vection axis and body posture on visually- induced self-rotation and tilt. In *Motion Cues in Flight Simulation and Simulator Induced Sickness, AGARD Conference Proceedings 433 (Brussels, 29 Sept/1 Oct 1987)*, pp. 15.1-15.8.

[Hung94] George K. Hung and Kenneth J. Ciuffreda. Sensitivity Analysis of Relative Accommodation and Vergence, *IEEE Transactions on Biomedical Engineering. 41(3), March 1994. pp 241-248.*

[Kenn87] R.S. Kennedy, K.S. Berbaum, G.O. Allgood, N.E. Lane, M.G. Lilienthal, and D.R. Baltzley. Etiological significance of equipment features and pilot history in simulator sickness. In *Motion Cues in Flight Simulation and Simulator Induced Sickness, AGARD Conference Proceedings 433 (Brussels, 29 Sept/1 Oct 1987)*, pp. 2.1-2.15.

[Kenn90] R.S. Kennedy and J.E. Fowlkes. What does it mean when we say that simulator sickness is polygenic and polysymptomatic. In E.G. Monroe, editor, *Proceedings of the 1990 Image V Conference (Phoenix, 19-22 June 1990),* Image Society, pp. 45-44.

[Land83] E.Land. Recent advances in retinex theory and some implications for cortical computations: Color vision and the natural image. *Proceedings of the National Academy of Sciences*, Vol. 80, 1983, pp. 5163-5169.

[Lipt91] L.Lipton. Temporal artifacts in field-sequential stereoscopic displays. In *Proceedings of SID '91 (Anaheim, CA, 6-10 May 1991)*, pp. 834-835.

[McDo90] I.E. McDowall, M.Bolas, S.Pieper, S.S. Fisher, and J.Humphries. Implementation and integration of a counterbalanced crt-based stereoscopic display for interactive viewpoint control in virtual environment applications. In *Stereoscopic Displays and Applications*, *SPIE Proceedings*, Vol. 1256, February 1990, pp. 136-146.

[Mone91a] K.E. Money. Signs and symptoms of motion sickness and its basic nature. *Motion Sickness: Significance in Aerospace Operations and Prophylaxis (Toronoto, 7-8 Oct 1991) AGARD Lecture Series 175.*

[Mone91b] K.E. Money. Simulator sickness. *Motion Sickness: Significance in Aerospace Operations and Prophylaxis (Toronoto, 7-8 Oct 1991) AGARD Lecture Series 175.*

[Mooi87] H.A. Mooij. Technology involved in the simulation of motion cues: The current trend. In *Motion Cues in Flight Simulation and Simulator Induced Sickness, AGARD Conference Proceedings 433 (Brussels, 29 Sept/1 Oct 1987)*, pp. 2.1-2.15.

[Padm92] P.Padmos and M.Milders. Checklist for outside-world images of simulators. In *Proceedings of ITEC '92, International Training Equipment Conference and Exhibition (Luxembourg, 7-9 April 1992)*, pp. 2-14.

[Robi91] W.Robinett and J.P. Rolland. A computational model for the stereoscopic optics of a head mounted display. In *Stereoscopic Displays and Applications II, pp. 140-160. SPIE Proceedings* Vol. 1457, May 1991.

[Teit90] M.A. Teitel. The eyephone, a head mounted stereo display. In *Stereoscopic Displays and Applications, SPIE Proceedings* Vol. 1256, February 1990, *pp. 168-171.*

[Wysz82] G.Wyszecki and W.S. Stiles. *Color Science.* John Wiley and Sons, New York, 1982.

Lessons Learned from Visual Simulation

Michael Jones
Silicon Graphics Computer Systems

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

1 Introduction

Interactive computer graphic simulation of virtual environments has been business-as-usual in the visual simulation industry since the 1960's. The use of real-time image generation for military aircraft and armor training, commercial pilot training, and astronaut training is so pervasive that finding persons skilled in these activities who have not been trained in simulators would be problematic. As costs of simulation systems have declined, the number of applications using these techniques has increased, assisting in training people to drive automobiles and trucks, to pilot ships, to operate cargo and construction cranes, to serve as railroad engineers, to perform surgical procedures, and to perform other tasks where skill development is important and especially where the unavoidable student mistake can prove fatal.

Significant advances have been made in the design of hardware and software systems that provide simulation as well as in the understanding of human perceptions that govern the sense of realness and thus training effectiveness that such systems can provide. Computer graphics systems are beginning to be used in interactive entertainment applications where many, if not all, of the lessons learned from visual simulation apply. This section of the SIGGRAPH '96 course "Designing Real-Time Graphics for Entertainment" surveys many of the key insights developed by real-time graphics pioneers as they developed the first visual simulation applications and suggests how these innovations can be incorporated in the design and implementation of current and future entertainment projects.

Preliminaries

Responding to the fact that the efforts of real-time image generation systems are directed at human observers, the first subject to understand is human physiology as it controls and defines observer perceptions. This is a deep and richly researched topic of which much is known (and of which much remains to be discovered). Human perceptions of motion, the spatial perception of sound, and the nature of our visual system all mandate requirements that interactive systems must meet to be truly interactive and enable a sense of realness. These topics are discussed in companion sessions of this course.

Overview

This paper outlines and reports on the major techniques used in visual simulation systems. These techniques are collected into six major groups, each of which covers several related topics. The groupings and their topics are:

- The importance of low-latency in image generation, which includes issues of frame rate and latency, both real and perceived. Latency can not be avoided, but its effects if identified and understood can be minimized not only by attention to hardware design and software structure, but also through subtle interactions of viewpoint control and graphics system features.
- The need for consistent frame rates and approaches that enable them. The goal of a fixed frame rate is central to visual simulation. Achieving this goal is very difficult since the very nature of the problem implies using a fixed graphics resource to view images of varying complexity. Designing for constant frame rates demands concessions in both hardware, database, and application design that must be understood if the goal is to be met.
- The sense of realness and training effectiveness that rich scene content provides. Since complex, detailed, and realistic images are nearly always desired by the same customers who want high update rates and low system cost, providing interesting and natural scenes is largely a matter of tricks and half-way measures since a naive implementation would be prohibitively expensive in terms of machine resources.
- The significance and importance of texture mapping in realistic visual simulation and entertainment applications cannot be overstated. Texture processing is arguably the single most important incremental capability of real-time image generation systems. The presence and sophistication of texture processing continues to define the “major” and “minor” leagues of visual simulation technology.

- The difficulty and complexity of database construction for visual simulation systems. One of the key notions of image generation systems is the fact that they are programmed largely by their databases. This programming includes the design and specification of several autonomous actions for later playback by the visual system.
- The advent of real-time character animation capability in entertainment systems is based on features and capabilities originally developed for high-end flight simulation applications. Creation of compelling entertainment experiences hinges on the ability to introduce engaging synthetic characters. This capability is just now beginning to be introduced and the issues surrounding its use are discussed.

Analysis

In addition to reviewing the lessons learned from the last 30 years of visual simulation, this paper also identifies how these lessons are being applied in several major visual simulation based entertainment projects and observes the influence that the entertainment market is exerting on the visual simulation industry.

2 Low-Latency Image Generation

The issue of latency is critical to comfortable perception of moving images under interactive control. Our human experience of such simple actions as looking to our left or right is that the image we see moves smoothly and instantly in reaction to our own motion. Sadly, this is rarely the case in simulated visual environments. In fact, the usual response to head movement or other control input is a discrete series of frames that represent point-sampled images generated at fixed time intervals. Even worse, the image resulting from a movement often is not presented until several frame intervals have elapsed, creating a very unnatural latency due to technical issues involved in generating and displaying the images.

A common human reaction to these unnatural images is a nausea commonly known as *simulator sickness*. The comprehensive surveys by Kennedy and coworkers [Kenn90][Kenn92] summarize more than thirty reports on simulator sickness and conclude that transport delay is a major factor, particularly with regard to motion of the simulated horizon.

In visual simulation the terms “latency” and “transport delay” refer to the same thing—the time elapsed between stimulus and response. Confusion can enter the picture because there are several important latencies in visual simulation

and often it's not clear which one is being discussed, that is, which stimulus and which response.

The most general measure is the *total latency*, which measures the time between user input (such as a pilot moving a control) and the display of a new image computed using that input. An example would be a sudden roll after smooth level flight. How long does it take for a tilted horizon to appear?

The total time required is the sum of latencies of components within the processing path of the simulation system. The basic component latencies include:

1. Input device measurement and reporting latency
2. Vehicle dynamics computation latency
3. Image generation computation latency
4. Video display system scan-out latency

This is the latency that matters to the user of the system, since the overall latency is what controls the sense of realness the system can provide.

Despite the utility of a total system measure, vendors of subsystems can only provide latency measures for their component. The exception to this is the image generation system, since the video latency is implied by the video output format of the image generation hardware. The combined image generation computation latency and display system latency measure is known as the *visual latency.*

Questions of latency in visual simulation applications usually refer to either total latency or visual latency. The application developer will select the scope (in the sense of the tasks enumerated above) of the application, and then the latency will be decided by the choice of image generation mode, frame rate, and video output format, as shown below.

Calculating Latency

To calculate the visual latency in visual simulation applications we must define the stimulus and response to be used to define the interval.

The proper stimulus for the visual system measurement is the end of application processing, since it is the point in time at which control is handed to the visual system for the computation of the next image. The computational expense which precedes this point is application time: time spent in input device measurement, in intersection processing, and in vehicle dynamics computation. This time is important and contributes to total latency, but must be counted separately.

The response typically considered to terminate the latency calculation in visual simulation is the point in time at which the last pixel of the first field of the new

image is displayed. Understanding this requires a brief review of video output terminology.

A complete video image is known as a *frame*, and a frame is composed of one or more *fields*. The two most common video formats used by visual simulation applications are the 60 Hertz non-interlaced display typical of computer workstations and the 30 Hertz frame, 60 Hertz field, interlaced NTSC video timing used by broadcast television.

In the non-interlaced case, there is one field per frame, so raster display lines are scanned sequentially from top to bottom. This means that at a 60 Hz frame rate, each frame (which in this case is synonymic with a field) requires 1/60th second to display. In this situation the "last pixel of the first field" metric outlined above will be reached 1/60th second after the start of the display of a frame. Presuming that one field is sufficient for all drawing, the actual visual latency in this case is 16.67 msec for image generation and 16.67 msec for video display, which is 33.33 msec in total, as is indicated in Figure 1.

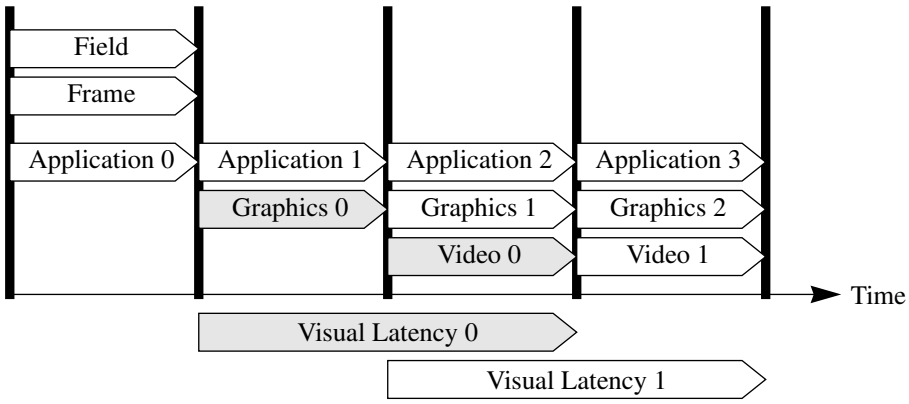


FIGURE 1.

Experience shows that other circumstances such as graphics hardware structure or software multiprocessing implementations may lengthen the graphics stage of this pipeline beyond one field thereby raising the total latency.

Interlaced displays are different in that they make two passes through the image, displaying the even-numbered scan-lines on one trip and the odd-numbered lines on the other. Each of these passes is known as a field, and since there are two fields per frame, the field rate is twice the frame rate. In the NTSC example, the frame rate is 30 Hz, but the field rate is 60 Hz, so the "last pixel of the first field" event will also occur 1/60th second after the start of the display of a frame. Presuming that one frame (twice as long as in the previous example) is sufficient for image generation, the latency in this case is 33.33

msec for this task and 16.67 msec for the video display of the first field. Combining these two time intervals results in 50 msecs of total visual latency.

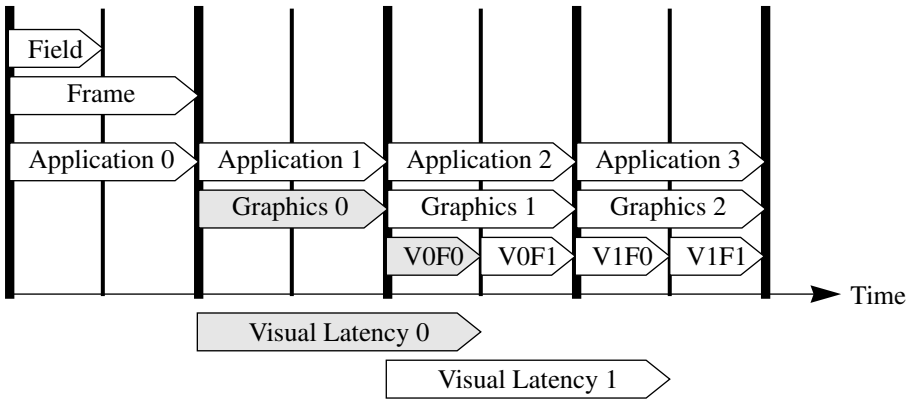


FIGURE 2.

Note that each frame in Figure 2 consists of two fields and that the display of the two fields is shown as individual events on the Video timeline, where ‘V’ represents video scan-out and ‘F’ indicates field 0 or 1. Notice also that the visual latencies are shown at the bottom of the diagrams of both Figure 1 and Figure 2.They measure the time elapsed between the end of application processing and the time when the last pixel of the first field of the new image is displayed. This is the true visual latency.

The proceeding discussion describes the factors that define the actual visual latency of an image generation system. The actual total latency is defined by simple extension of these arguments to include the other sources of latency present in the system. Common sources of latency include:

1. Input devices such as head trackers, gloves, and joysticks
2. Communication between input devices and the simulation computer
3. Operating system delivery of input data to the simulation processes
4. Algorithmic complexity of vehicle simulation after input processing
5. Communication between simulation process and graphics hardware
6. Processing time within the graphics hardware
7. Video scan out of computed images

Minimizing the total latency (and thus the visual system latency as well) is a worthy goal, but can be expensive to attain since it usually means allocating more hardware resources to the task.

Perceived Latency

In many situations the perceived latency can be much less than the actual latency. This is because the human perception of latency can be reduced by anticipating the control inputs that the user is about to make. This means that reducing perceived latency is largely a matter of accurate prediction.

As an extreme example, a program capable of precognition could start drawing images at just the right time so that the human observer would never perceive more than one video field of latency no matter how far in advance (minutes or hours) the rendering had been begun. Writing such clairvoyant programs can be difficult.

Fortunately, these extreme measures are not required in practice since actual latencies tend to be short by human standards. Actual visual latencies as defined above range from 33.3 to 50 msec in current lowest-latency systems and human motor skills define upper limits on how much motion can be achieved in such short intervals. Testing of military pilots has shown that maximum rates of head motion are approximately 100 degrees per second, which reduces to a 3 to 5 degree change during best-case latency intervals for worst-case viewpoint motions.

Using well-known techniques of prediction, extrapolation, and filtering [Kalm61] based on rates and acceleration, the position of joysticks, head tracking systems, and other inputs sampled at the start of the latency interval can be estimated forward in time to the instant when the frame will be displayed. If the difference between the predicted and actual control effect is slight, the perception will be one of reduced latency.

Researchers have shown that prediction alone can not resolve all latency and perception problems. The results of Cardullo and Yorke [Cardul90] suggest that even with comprehensive approaches such as the McFarland position prediction method [McFarl88], actual system latencies greater than 100 to 120 msec can be insurmountable.

For the prediction approach to be most effective, changes to several simulation components must be extrapolated, including the observer's eyepoint and gaze direction and the position and orientation of any moving objects within the scene. Implementors of systems with rotation extrapolation have found quaternion interpolation to be the primary mechanism for robust solutions.

Two other approaches to reducing perceived latency that work well are based on the notion of making changes to image parameters during the course of rendering and display. These changes are interjected into the drawing pipeline between the culling and drawing stages in the first approach, and between the drawing and video stages in the second. These methods are not appropriate in

all environments since they require a very tight coupling between user processing and graphics processing.

The first approach uses eyepoint and gaze direction prediction to guess where the user will be looking as outlined above. Rather than culling the database to the minimal viewing frustum however, an expanded viewing volume which is a few degrees larger in horizontal and vertical view extent is used. As mentioned above, the viewer is not expected to be able to look more than 5 degrees further in any direction in typical situations. This would enlarge a nominal 50 degree horizontal FOV into a 60 degree wide culling volume. Then, after the database is culled to this enlarged volume and just before the drawing task is begun, a presumably more accurate view specification is obtained from more recent inputs available during cull processing. Use of the more accurate view prediction will reduce the perceived visual latency. This is a very effective approach on systems where updated viewing information becomes available during the culling interval, where applications can introduce last-moment view information after culling is complete, and where the extra culling and drawing expense incurred by the expanded frustum is affordable.

A second method that relies on a video output hardware feature to reduce the perceived latency is described by Riner and Browder in [Riner92]. In this approach, which can be used along with the previous one, the image rendered into a frame buffer slightly larger than that which will be displayed. For example, when producing 1280x1024 images, a 1300x1044 image might be rendered. This would be an image that is ten pixels larger on all four sides.

An example of this is shown in the left portion of Figure 3, where the heavy box represents the total rendered area and the centered inner box represents the expected view and display area. The viewing specification used to produce this situation is computed such that the inner 1280x1024 portion to be exactly that which would be viewed under normal circumstances. In other words, the inner area of the larger image contains the same pixels as would be rendered normally, while the border area represents image regions just slightly “off screen”.

After this larger image has been rendered and before the video scan-out commences, the view information is again sampled and extrapolated, and a determination made about the up-down and left-right extent of the positional error contained in the rendered image. These offsets are used to specify a video panning of the frame buffer which repositions the display surface within the window in an attempt to correct for prediction error.

The right portion of Figure 3 completes the example of this processing. In that figure, the updated eyepoint information has indicated that the image region above and to the right of the expected display area should be displayed. The video output component of the graphics hardware is then adjusted to cause the

display of this sub-region on the display surface. This image-space panning operation must be performed incrementally for each video field that intervenes between the computation of images from new eyepoints.

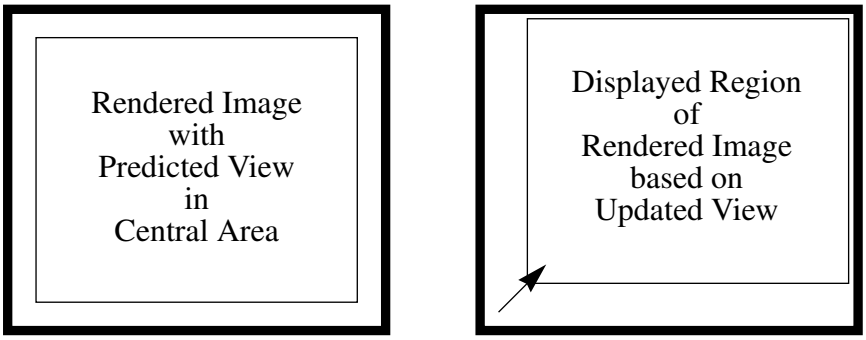


FIGURE 3.

This two-dimensional panning process introduces parallax errors into the image but can be used sparingly to provide an additional and quite effective technique for reduction of perceived latency.

3 Consistent Frame Rates

Interactive graphics applications, and immersive virtual environments in particular, are especially dependent on a consistent frame rate for observer acceptance. Human perceptions are attuned to continuous update from natural scenes but seem tolerant of discrete images presented at rates above 20 frames per second in many cases. Tolerant that is, when the frame rate is consistent. When latency grows large or frame rates waver the result is certain: the headaches and nausea of simulator sickness.

The attainment of a constant frame rate for a constant scene is easy. What's hard is maintaining a constant frame rate when scene content and complexity vary wildly, as is often the case. Several approaches have been used in image generation systems to ensure or at least attempt the enforcement of a constant, programmer-selected, frame rate.

The first and most basic method is to only draw scenes of such complexity as can be safely viewed from any location at the chosen frame rate. This conservative approach is much like always driving in low-gear just in case a hill were to be encountered. Implementing it simply means identifying and planning for

the worst case situation of graphics load. Although this may be reasonable in some cases, in general it's unacceptably wasteful of system resources. This is particularly so in applications where the observer can view a database from different heights, with different fields of view, or where independent moving vehicles might converge unexpectedly. If a database is sparse enough so that viewing it from high altitudes achieves the goal frame rate, it is certain to be nearly featureless when viewed from ground level.

A second approach is to discard (cull) database objects that are positioned completely outside the frustum of vision. This requires a pass through the visual database to compare scene geometry (or simpler bounding volumes that represent collections of geometry) with the current frame's viewing volume. Any objects completely outside the frustum can be safely discarded. When this approach is used with databases that have a recursive spatial partitioning and hierarchical bounding volumes at each level, the culling process becomes tractable even for large databases and is a necessary first step in limiting scene complexity. A hierarchical, grid, or cellular culling mechanism is the basis of view processing in nearly-all real-time visual simulation systems. A recent implementation of this approach is detailed in [SGI94].

When simple view-volume culling as outlined above is insufficient to keep scene complexity constant, it may be necessary to compute a potential visibility for each object during the culling process by taking into consideration other objects within the scene that may potentially occult the test object [Airey90] [Teller91]. High performance image generation systems have used comparable occlusion culling tests to reduce the polygon filling complexity of real-time scenes. Approaches have included half-space tests based on eyepoint position and hardware depth sorting and front-to-back rendering combined with tests for fully-covered pixels.

Additional methods can be used to adjust the graphic complexity (and thus the processing time requirements) of those database portions that pass the culling test. Several software-oriented approaches to this task have been found to be effective and are outlined in a later section on level of detail selection.

Approaches to fixed frame rates that are based on hardware features have been used as well. One method is based on the use of special video monitors that can adjust video line timing on a per-frame basis. This allows frames to be rendered with slightly varying frame rates and provides an additional measure of safety in meeting the desired frame rate. The expense of CRT display systems with this capability suggests that solutions appropriate for entertainment applications must be sought elsewhere.

The most certain (and certainly the most Draconian) of the hardware-based fixed frame rate measures is the absolute frame rate nature of one vintage

image generator. This machine used a double buffered frame buffer that autonomously swapped buffers at the programmed frame rate without regard to the state of rendering based in interrupts from a real-time clock. Even when polygons that had passed the culling test remained to be drawn or if the current polygon had only partially been rendered, the machine would swap buffers at the end of the designated frame interval and begin display of whatever image was in the frame buffer. This situation would likely include at least one partially rendered polygon.

A clever approach was taken by the designers of this system to mitigate the effect of these incomplete scenes. The machine operated in an interlaced video display mode and the frame buffers consequently held only a single field. Taking advantage of this the order of polygon filling toggled for each field generated: it was top-to-bottom on one field and bottom-to-top in the next. This allowed raster interlace display artifacts and human visual system integration of sequential fields to partially hide the incomplete polygons.

4 Rich Scene Content

Level of Detail Selection

All graphics systems have finite capabilities that affect the number and type of geometric primitives that can be displayed per frame at a specified frame rate. Because of these limitations, maximizing visual cues while minimizing scene complexity is the fundamental science of database construction for real-time simulation. The concept of level of detail selection is one of the most beneficial tools available for managing database complexity for the purpose of improving display performance.

Object Level of Detail

The basic premise of LOD processing arises from the observation that objects that seem small because they are located at a great distance from the observer's eyepoint or that are barely discernible because atmospheric conditions are reducing visibility, don't require much polygonal complexity. Alternately, these same objects might require many polygons and textures to appear equally correct when viewed at close range, in clear air, or when using a narrow field of view. Combining these two observations into a general approach, we note that each model in databases can be built as a set of alternate representations, each of which is designed to be viewed at a certain distance or screen-space size.

For example, consider an automobile model and imagine a parking lot packed with rows and rows of exactly the same car model. If you were to stand in the

middle of the parking lot and examine the car closest to you, you would be able to see many small features such as rear-view mirrors, raised letters on the tires, door handles, and license plates. Turning your attention to the very last row of cars off in the distance you would recognize the car from its general shape, but would not be able to see any of these small details. (Or else you imagined a much smaller lot than the author.) That this would be the case follows easily from the fixed resolving power of the human retina. Sampling theory suggests the same dichotomy applies to computer graphics because the rendered size of polygons viewed in perspective decreases with range and as the polygon size becomes small compared to the fixed pixel spacing, the polygons no longer contribute to the scene.

It should be clear that the complexity of a car model designed for viewing at a two to three foot range might be vastly different than that for cars on highways when viewed from aircraft. In this latter case a five polygon box of the appropriate color would be sufficient while in the former there is almost no limit to the number of polygons that could be gainfully employed. What may be less clear is that an entire spectrum of cars could be defined, each designed for viewing at intermediate distances between these two extremes.

Control

Having produced multiple versions of an object and specified the ranges or other criteria used to select which version of an object to display in each circumstance, the LOD process is ready to begin. The entire ensemble is treated as a single object and translated or rotated as desired. During the culling phase of frame processing, the distance from the eyepoint to the object is computed and used to select which version of the model should be viewed. As the eyepoint approaches an ever less distant car, each of the versions would be displayed in turn: the simplest, then the one slightly more detailed, and so on until at close range, the most detailed version would be selected for viewing.

Distance is one of several criteria that can be used to select an appropriate level of detail when rendering an object. Early efforts used the area of a screen aligned bounding box about an object to determine the LOD. This approach has been largely abandoned based on experience with object variation due to rotation (but not completely, see [Wern94] for a recent revival). The difficulty with this method arises when a slender object, such as a pencil or runway, rotates on the screen. The area of a screen aligned bounding box can vary tremendously and this causes the LOD chosen for the object to waver unacceptably. Better results are obtained when the distance to the object's centroid is used. Factors commonly used to adjust the LOD distance include field of view, image resolution in pixels, the optical accuracy of the display system, atmo-

spheric effects of haze and fog, and specific offset or scale factors used to influence LOD selection for load management.

This decision process causes the available polygon budget to be spent wisely in that it assures that each polygon drawn contributes significantly to the richness of the scene. Much like culling, level of detail processing is a basic and central feature of high-performance real-time visual simulation systems.

Popping

The level of detail selection approach described above is not without fault, however. The problem occurs at the point of transition between one version and its neighbor. Since the transition happens abruptly between two frames, there is a visual "popping" from one version to the next. Unfortunately, this type of step discontinuity in visual images is one that the human visual system is keenly adept at discerning. Several mechanisms have been developed to avoid this distracting popping artifact, including fading and morphing.

Fading

Much more subtle than the distracting popping approach is to perform the transition from one version to its neighbor over a distance range rather than at a single point. The transition is done by drawing *both* versions during the transition zone, using complementary transparency to make the switch nearly undetectable.

For example, if the higher LOD car model is to be drawn at ranges out to 100 meters, and the lower LOD model at greater ranges, we might define a 20 meter transition range and specify that the blending or fading of the images of each model be a function of the relative distance within the transition ranges. The complementary fading produced by this example is shown in Figure 4.

Distance	High LOD	Low LOD
90 Meters	100% Opaque	0% Opaque
95 Meters	75% Opaque	25% Opaque
100 Meters	50% Opaque	50% Opaque
105 Meters	25% Opaque	75% Opaque
110 Meters	0% Opaque	100% Opaque

FIGURE 4.

It is important to note that each model is itself drawn as opaque and the resulting images are subsequently blended according to the schedule shown above. Drawing the actual higher or lower LOD versions as partially transparent does

not work since self-occlusion relationships within the models must be maintained. In the automobile example given above, drawing the higher LOD version as 75% opaque (and thus 25% transparent) would allow the tires to be seen through the hood of the car, destroying image integrity.

Using just three intermediate blend levels as shown above is not enough to provide a smooth transition. Typical real-time image generation systems provide between 8 and 33 levels of complementary blending and users have found this sufficient for most applications.

Intermediate distance is not the only mechanism for these transitions. One other popular transition technique is time based. In this model the switch points are identified as ranges as previously described, but the process of LOD transition is implemented over a selectable transition time. Such *time-based range-triggered* transitions can be very important when an object may stay at a fixed distance from the eyepoint for extended intervals, as is common on the downwind, base, and final approach legs of aircraft landings—in each case the control tower and terminal area are usually at a similar distance from the pilot and the sustained fade expense can be prohibitive.

As alluded to in the previous paragraph, the primary weakness of the fade-LOD transition approach is that it requires two models to be drawn during the transition interval. This increases the processing complexity of the scene and can also prove awkward when picking or intersection calculations are performed on a model in transition.

Terrain Level of Detail

The presumption made in the previous discussion of discrete level of detail selection is the notion that the *entire* model should transition at the same time. It's quite reasonable that features of an automobile such as door handles could transition out of the scene at the same time even though the passenger door might be slightly more distant than the driver's door. It is much less clear that this would be workable for very large objects such as an aircraft carrier or a space station, and it's clearly not acceptable for objects that span a huge extent, such as a terrain surface.

Attempts to handle large-extent objects with the discrete LOD tools focus on breaking the big object into myriad small objects and treating each small object independently. This works in some cases but often fails at the junction between two or more independent objects where cracks or seams exist when different detail levels apply to the objects. Some terrain processing systems have attempted to provide a hierarchy of crack filling geometry that is enabled based on the LOD selections of two neighboring terrain patches. This "digital grout"

becomes untenable when more than a few patches share a common vertex, and like it's bathroom brethren, is often unsightly in appearance.

An alternate approach that is currently in vogue in the real-time visual simulation community [Cos90][Ferg90] treats terrain as a single connected surface. In many ways, this system is like the well known and widely implemented fractal terrain notion seen in the film "Vol Libre" and described by Fournier, Fussell, and Carpenter in [Fourn82].

The essence of their approach is that a triangle representing terrain is judged to be either sufficiently accurate or else is recursively subdivided. The general manner of subdivision is to introduce a vertex at the midpoint of each side of the original triangle. Connecting these new vertices to make a triangle leads to the subdivision of the original triangle into four triangles. After subdivision, the algorithm perturbs the vertex elevations. If the four resulting triangles are not yet sufficiently fine, the subdivision process is applied to each one in turn.

This subdivision process can be applied to terrain level of detail needs by making slight modifications to the implementation outlined above. The first change is to replace the random vertex elevation process with a simpler sampling of the terrain surface. Secondly, an interpolation mechanism is used to move from one level of tessellation to the next during LOD transition zones. Here's how it's done. The triangle labeled 'A' in Figure 5 below represents the triangle before the LOD evolution process is begun.

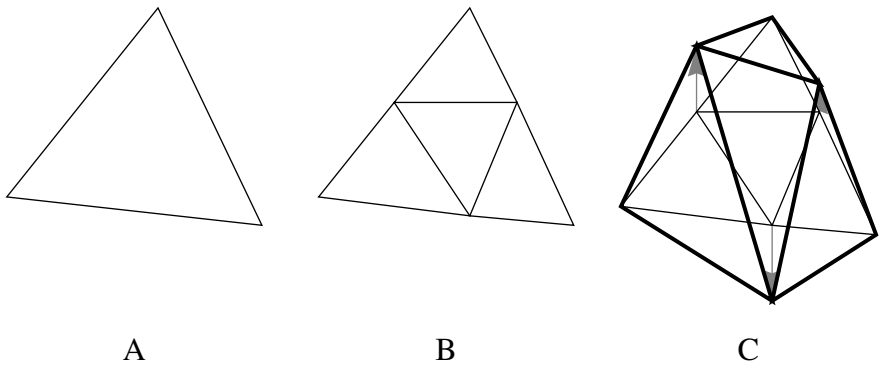


FIGURE 5.

As before, when the eyepoint moves close enough that an LOD transition is required, a transition phase is entered. This time, however, no blending is used. First the single triangle is replaced with four triangles. This is indicated by step 'B' of Figure 5. These four triangles are a simple tessellation of the original. Since the newly introduced vertices are in the plane of the original triangle and

share all material properties, the observer sees no change in the image at the instant where the three new vertices are introduced. Since there is no visual change, the replacement can be performed instantly without introducing visual artifacts or incurring the expense of rendering two versions of the terrain.

As the eyepoint moves closer to these polygons, the elevations of the three new vertices are interpolated from their starting place in the plane of the lower-LOD triangle to their final destinations as sampled from the true terrain surface. This is indicated in step ‘C’ of Figure 5 where the four heavy triangles are the fully interpolated locations of the triangles introduced in step ‘B’. The vectors show the direction of interpolation and the thin triangles are for reference. They show the locations of the three edge midpoint vertices before the elevations were interpolated.

If the eyepoint moves even closer to one of the four new triangles, this interpolated subdivision process can be continued in that triangle as well. This subdivision also requires a companion subdivision of at least one neighboring cell to avoid ‘T-vertices’, which suggests that the tessellation is actually an edge operation. If these same three steps are applied to the lower-left triangle in step ‘C’ of Figure 5 to produce an even more accurate terrain surface in the area near the lower-left vertex, the results will be as shown in Figure 6.

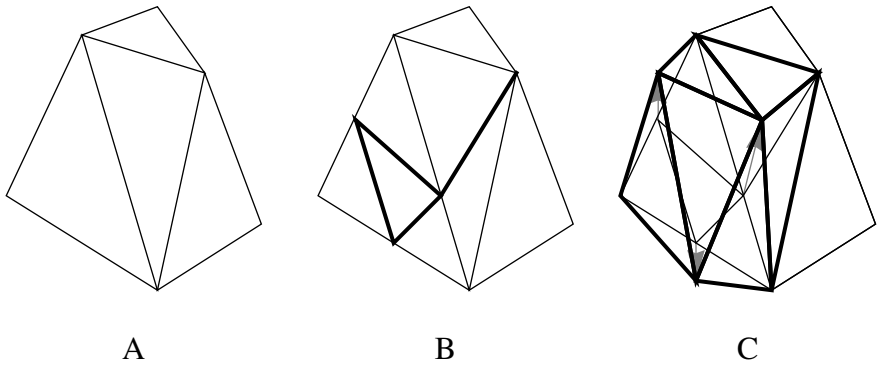


FIGURE 6.

Notice that in addition to the major recursive subdivision and interpolation of the lower-left triangle, a minor subdivision and interpolation of the central triangle was required as well. When the central triangle of a four-triangle must be subdivided this minor subdivision and interpolation operation is required at all three neighboring triangles. Although the management of this system can be complicated, the efficiency and accuracy of terrain so represented is widely thought excellent enough to justify the effort.

Arbitrary Morphing

Terrain level of detail by layered interpolation is a restricted form of the more general notion of object morphing. Morphing of models such as the car in a previous example can simply involve scaling a small detail to a single point and then removing it from the scene. Morphing is possible even when the topology of neighboring pairs do not match. The paper by Kent, Carlson, and Parent [Kent92] introduces and surveys the subject. Both models and terrain can have vertex, normal, color, and appearance information interpolated between two or more representations. Several benefits accrue when this approach is used. The advantages include: reduced graphics complexity since blending is not used, constant intersection truth for collision and similar tasks, and monotonic database complexity that makes system load management much simpler. The computational resources required to perform general interpolation of this nature to large portions of a scene at interactive frame rates are just now becoming available. Further examples and details of this mechanism are provided in a later section of this paper that deals with character animation.

System Load Management

Level of detail selection can be biased at will. That is, transitions can be made to occur either closer or further from the eyepoint than is specified by the level of detail parameters encoded in the database. This is a very powerful opportunity provided by level of detail processing, since it enables the construction of a closed-loop control system for maintaining a specified frame rate.

In it's simplest form, the control system measures the time required to generate a frame and compares this value with the chosen frame interval. As the frame time begins to approach the frame interval the level of detail selection bias is increased to encourage earlier than usual level of detail transitions. When the rendering time returns to a desired level, the bias factor can be returned to normal or perhaps even set to allow higher than expected visual quality. This type of closed-loop feedback system is shown in Figure 7.

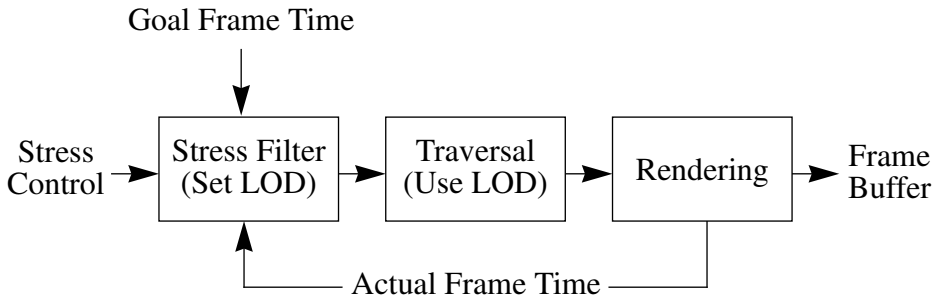


FIGURE 7.

The IRIS Performer real-time visual simulation system developed by the author and coworkers [SGI94] implements load management with a proportional-control feedback scheme as shown above. In this system, the desired frame time (such as 1/30th second) as well as the actual frame time of previous frames are used as input to a stress computation function that computes a new level of detail range scale factor based on selectable stress control parameters. Our experience has shown that designers of stress filter functions must use great care [Lathi74] to design a quickly reacting and well damped control-system.

Further extensions and elaborations of this concept have been explored. A recent paper by Funkhouser and Sequin [Funk93] explores an alternate level of detail selection mechanism designed to minimize variations in image rendering time.

Billboard Objects

Many of the objects in databases can be considered to have one or more axes of symmetry. Trees, for example, tend to look nearly the same from all horizontal directions of view. An effective approach to drawing such objects with less graphic complexity is to place a texture image of the object onto a single polygon and then rotate the polygon during simulation to face the observer. These self-orienting objects are commonly called *billboards* even though the phrase "sandwich board" is much more accurate. The images in Figure 8 show how this rotation keeps the billboard geometry perpendicular to the viewing direction.

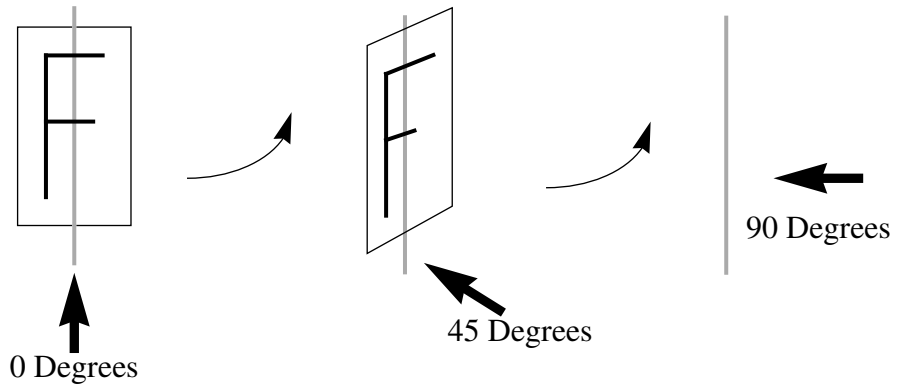


FIGURE 8.

There are a number of implementation choices for billboards. For example, they can be made to always face the eyepoint, to always align with the observ-

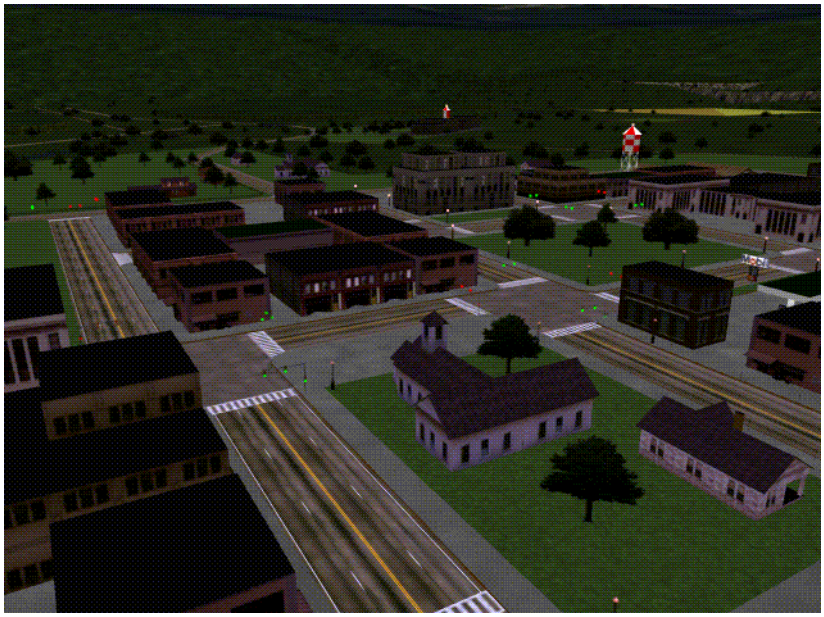


FIGURE 9.

er's gaze direction, or to align to an arbitrary vector. Also, since the billboard rotates, there is the question of what axis to rotate about. The answer for trees is simple since gravity works to keep tree trunks remarkably vertical. Since trees are such common billboard objects, some systems in fact allow only a vertical axis of rotation.

It is also feasible and useful to allow the billboard image to rotate about two axes rather than one. When this is done, it's possible to achieve the effect of a screen-aligned two-dimensional image that varies in size as a function of range. An additional billboard complexity adds automatic scaling as a function of range to the dual axis of rotation system, and is able to achieve the *bit-blt* sprite effect that forms the mainstay of low-cost graphic games, but does so at a surprisingly significant computational expense.

When combined with texture, the billboard approach is effective in enabling the simulation of trees in high-performance applications. The image in Figure 9 is from a real-time simulation that contains 709 billboard trees, each of which is rotated to face the eyepoint every frame. Notice how real the trees look, particularly in the middle to far distance ranges where billboards are best utilized.

The image in Figure 10 shows the same scene with texturing disabled. The rotation of billboard objects to face the viewpoint can be easily detected in the tree polygon located at the lower right third of the image.

FIGURE 10.

Animation Sequences

Often animated events in simulation environments have a sequence of stages that follow each other without variation. Where this is the case it's possible to define this behavior in the database during database construction and allow the behavior to be implemented by the real-time visual system without intervention by the application process.

An example of this would be illuminated traffic signals in a driving simulator database. There are three mutually exclusive states of the signal, one where the green lamp is illuminated, one with the amber, and one with the red. The duration of each state is known and can be recorded in the database. Having specified these intervals during the database construction process, simulations can be performed without requiring the simulation application to cycle the traffic signal from one state to the next. The traffic signals in Figure 9 were modeled in just this way using commercial database modeling tools. The rate of their advance is completely specified in the database as a function of time and they maintain their duty cycle without regard to the update rate of the visual system. If the operation of these lights is adjusted in the database then any application that views them will provide the adjusted operation automatically.

Early image generators were stand-alone subsystems with somewhat low communication rates with their host computer. Latency and bandwidth considerations of interactions with these side-car systems led to the notion of an *active*

database. The action in an active database is simply the specification of behaviors to be performed by the image generation system—behaviors that can proceed without application intervention.

The simplest type of animation sequence is known as a *geometry movie*. It is a sequence of exclusive objects that are selected for display based on elapsed time from a trigger event, whose advance is tied to frames rather than time, or where advancement is based on specific events within the database. It is useful to provide for cyclic looping and bidirectional shuttling of these sequences as well as comprehensive positioning, rate, and repeat count specification.

When the evaluation of objects within the sequence is more general than selecting geometry, a much more powerful mechanism results. Typical image generators allow the construction of hierarchical animation sequences. Recent real-time image generation software architectures [SGI94] have expanded on this concept considerably, allowing user-coded animation functions with arbitrary capability to be invoked in addition to providing full support for simple predefined animation sequences.

Antialiasing

The advent of antialiased image generation in visual simulation was a significant milestone in terms of image quality. The difference, though subtle in some cases, has very significant effects on the sense of realness and the suitability of simulators for training. Military simulators often center on the goal of detecting and recognizing small objects on the horizon. Aliased graphics systems produce a “sparkle” or “twinkle” effect when drawing small objects. This artifact is completely unacceptable in these training applications since the student will come to subconsciously expect such effects to announce the arrival of an opponent and this unfulfilled expectation can prove fatal. Situations where false cues are delivered to students often result in such *negative training*. This is especially true in aircraft landing situations where dark nighttime landings and bright runway edge and end illumination lights are commonplace. Such scenes, with slow perceived motion of single-pixel white points on a nearly black background can serve to test the quality of any filtering technique.

To review briefly, the idea is for image pixels to represent an average or other convolution of the image fragments within a pixel’s area rather than simply be a sample taken at the pixel’s center. This idea is easily stated but difficult to implement with high performance. Only recently have workstation-class systems been able to provide antialiasing along with the other expected features such as Z-buffering at full speed. In contrast to the newness of this feature in lower-cost systems, traditional image generators have provided high-quality antialiasing since the 1970’s—in fact the 1981 SIGGRAPH video review contains an excellent example of antialiased real-time image generation in a flight

simulation context. Several approaches to hardware antialiasing of polygons have been used in real-time systems. These fall into three broad classes and each is described in the following sections.

The original approach was the *coverage mask* method. It's similar to the well known A-buffer method introduced a few years later by Carpenter in [Carp84]. The coverage mask method is the one used by the image generation system featured in the video review mentioned above.

Each pixel in coverage mask systems has an associated coverage mask that is set to zero before rendering. The frame buffer is also reset to black at the beginning of each frame. Each bit in the coverage mask corresponds to a sample point within its associated pixel. Traversal logic in the system assures that polygons are drawn in front to back order. As each pixel of each polygon is rendered into the frame buffer, the polygon is tested against the pixel's sample points and a pixel coverage mask is computed. Comparing the new fragment's coverage mask with the existing coverage mask, and then counting the number of newly-covered sub-pixels touched provides a weighting factor indicating the incremental contribution that new polygon makes to the pixel. The polygon color is scaled by this value and the result is added to the frame buffer. When all polygons have been drawn, the frame buffer contents represent the box-filtered color contributions of all visible polygons incident to each pixel.

This approach produces excellent images and has proven amenable to efficient hardware implementations. Most of the classic real-time image generators used this method, and modern variations are still being developed. See [Schill91] for a recent example of a quality-sensitive approach and implementation.

The severe limitation of this approach is the rigid requirement to render all geometry within a scene in a strict front to back order. Although techniques such as cluster priority [Green74], binary space partitioning trees [Fuchs80], and polyhedral priority meshes make ordered database traversal tractable for static databases, handling complicated dynamic databases with large numbers of articulating moving objects is a surprisingly difficult task. Failure to recognize the difficulties implicit in the effort has been the bane of several well-intended simulator development projects.

The effort required to build large and realistic databases with these prioritization techniques can also add considerably to the cost and time required to build a complete visual simulation system. It is these complexities, rather than any perceived weakness in image quality, that have generally ended the development of new general-purpose systems based on the coverage mask approach.

The second approach is termed *supersampling* or *multisampling* [Crow81]. In this system, each pixel is considered to be composed of multiple subpixels, much like the sample points used in the coverage mask method. Unlike that

method, though, there is a complete set of pixel information stored for each of the several subpixels. This information includes color and transparency, and most importantly, includes a Z-buffer value as well. Providing multiple independent Z-buffered subpixels (the so-called *sub-pixel Z-buffer*) per image pixel allows opaque polygons to be drawn in an arbitrary order since the subpixel Z-comparison will implement proper visibility testing. Converting the multiple color values that exist within a pixel into a single result can either be done as each fragment is rendered into the mutisample buffer or after all polygons have been rendered. For best visual result, transparent polygons are rendered after all opaque polygons have been drawn. As described in [Akeley93] the multisampling approach is the basis of recent designs from visual simulation hardware vendors.

A third class of real-time antialiasing approaches is based on accumulation in the context of non-overlapping polygons. A simple example can be imagined in the rendering phase of a system that uses some form of “cookie cutter” approach to determining object visibility where subdivision assures that the output polygons do not overlap in screen space. This means that simple area-weighted color accumulation will render nicely sampled images. A version of this non-overlapping accumulation approach has been implemented in hardware, and its comparative advantages and limitations is described in [Gish91].

5 *Texture Mapping*

The most powerful incremental feature of image generation systems beyond the initial capability to draw geometry is *texture mapping*, the ability to apply textures to surfaces. These textures consist of synthetic or photographic images that are applied to geometric primitives in order to modify their surface appearance, reflectance, or shading properties. This process was developed by Catmull [Cat84] and elaborated by Blinn and Newell [Blinn76] and is now widely known and explored within the computer graphics community.

Texture mapping capability has been implemented by real-time image generation systems for more than a decade. These systems have achieved a diverse range of effects via texture mapping, many of which are described below.

Surface Appearance

The most obvious use of texture mapping is to generate the image of surface details on geometric objects. Examples of teapots, donuts, and other objects having the surface appearance of tile, wood, bricks, or flowers can be seen in any of the SIGGRAPH proceedings since Catmull’s germinative publication.

One valuable and widely used addition to these texture processing features is the concept of transparency as a per-texture-element attribute. An example of this has already been seen in Figure 9 where each tree is simply a photograph of a tree mapped to a flat rectangle. Figure 11 shows one of the three tree texture maps used in Figure 9 more clearly.

FIGURE 11.

The areas "outside" the tree but within the rectangle are parts of the texture image that are marked as being fully transparent. Portions of the billboards used as the geometry for this tree texture are not drawn at all, as can be seen by comparing Figure 9 and Figure 10.

Environment Mapping

In their 1976 paper, Blinn and Newell [Blinn76] also suggested that textures could be used to simulate reflections by using the viewing vector and the geometry's surface normal to compute each screen pixel's index into the texture image. The texture image used for this process, the *environment map*, contains a suitably encoded image of the environment to be reflected. This idea was elaborated by Greene in [Greene86]. Recent results by Voorhies and Foran [Voor94] show that the encoding step may not be necessary in future systems.

The fist step in the creation of an environment map is shown in Figure 12. This image was formed by concatenating six independent images taken in the "IRIS Performer Town" database shown in Figure 9. The six views are along both the positive and negative directions of the principal axes.

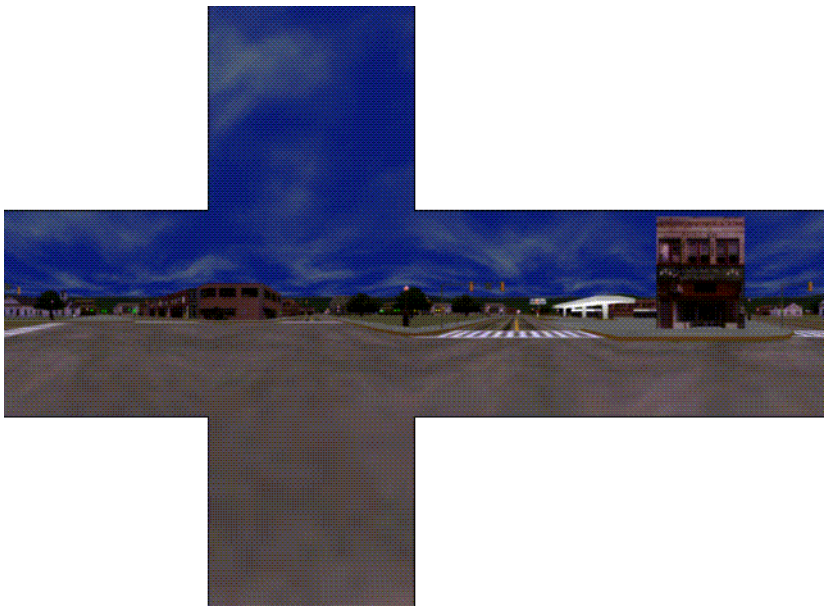


FIGURE 12.

The next step is to convert the images into the encoded form expected by the computation mentioned previously, as shown in Figure 13.

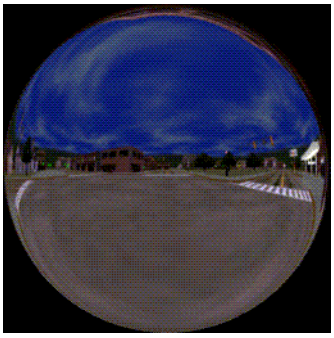


FIGURE 13.

The final step is to use the encoded texture to modify the surface appearance of a geometric object to simulate reflection. This can be seen in Figure 14 where the image of the Performer Town intersection from Figure 12 has been environment mapped onto a model of a recreational vehicle using the texture map shown in Figure 13.

FIGURE 14.

Sophisticated Shading

The environment mapping technique can be used to implement lighting equations by noting the fact that the environment map image represents the image seen in each direction from a chosen point. If this image is interpreted as the illumination reflected from an incident light source as a function of angle, then the *intensities* rather than the colors of the environment map can be used to scale the colors of objects in the database in order to implement complex lighting models with high performance. This method can been used to provide elaborate lighting environments in systems where per-pixel shading calculations would not otherwise be available.

Projective Texture

Texture mapping can also be used to project images such as aircraft landing lights and vehicle headlights into images. These projective texture techniques, when combined with the ability to use Z-buffer contents to texture images, allows the generation of real-time images with true 3D cast shadows. This is essentially the Shadow-buffer technique described by Lance Williams [Will78] as modified to meet the limitations of real-time image generation. These cast shadows provide a superior simulation environment than more widely known methods for rapid shadow generation, such as that described by Blinn in [Blinn88].

6 Character Animation

Betweening

The previous discussion of level of detail described a methodology for continuous terrain level of detail processing based on interpolation between two elevations for vertices, a process also known as *betweening*. Real-time image generation hardware capable of the interpolation of vertex position, colors, normal vectors, and texture coordinates between two versions of a model now exists in the form of multiple processor computer graphic workstations. In addition to this hardware, existence of both the software to drive these systems [SGI94] and the database development tools needed to specify the targets and their associations [Cos92] makes the betweening process a practical one not just for terrain level of detail, but for general model morphing as well.

Generalized Betweening

Simple pair-wise betweening is not sufficient to give animated characters life-like emotional expressions and behavior. What is needed is the ability to model multiple expressions in an orthogonal manner and then combine them with arbitrary weighting factors during real-time simulation.

The idea of using these techniques in the context of human facial animation are well described by Parke [Parke72]. One current approach to implementing Parke's method is to build a geometric model of an expressionless face, and then to distort this neutral model into an independent target for each desired expression. Examples would include faces with frowns and smiles, faces with eye gestures, and faces with eyebrow movement. Subtracting the neutral face from the smile face gives a set of smile displacement vectors [Parke75] and increases efficiency by allowing removal of null displacements.Completing this process for each of the other gestures yields the input needed by a the real-time system: a base or neutral model and a collection of displacement vector sets.

In actual use, the data is processed in a straightforward manner. The weights of each source model (or corresponding displacement vector set) are specified before each frame is begun [Parke82]. For example a particular setting might be "62% grin and 87% arched eyebrows" for a clownish physiognomy. The algorithmic implication is simply a weighted linear combination of the indicated vectors with the base model.

The processing steps outlined above are made more complicated in practice by the performance-inspired need to execute the operations in a multiprocessing environment. Parallel processing is needed because users of this technology:

- need to perform hundreds to thousands of interpolations per character;

- desire several characters in animation simultaneously;
- prefer animation update rates of 30 or 60 Hertz; and,
- generate multiple independent displays from a single system.

Taken together, these demands can require more than 30 MFLOPS, even when only vertex coordinates are interpolated. When colors, normals, and texture coordinates are also interpolated, and especially when shared vertex normals are recomputed, the computational complexity is correspondingly increased. The computational demands can be reduced when the rate of betweening is less than the image update rate. The quality of the interpolated result can often be improved by applying a non-linear interpolation operation such as the eased cosine curves and splines found useful in other applications of computer animation. The deeper implementation details of these and related issues are described in [Rohlf94] and in other sections of this course.

Skeleton Animation

A successful concept in computer-assisted 2D animation systems is the notion of *skeleton animation* as defined and described by Burtnyk and Wein [Burt71] for increased fidelity interpolation of key-frame drawings. The nature of this method is to interpolate a defining skeleton and then position artwork relative to interpolated skeleton. In essence, the skeleton defines a “deformation” of the original 2D plane, and the original image is transformed by this mapping to create the interpolated image.

This process can be extended directly into the three-dimensional domain of real-time computer image generation systems and used for character animation in entertainment applications. Early research in this area was performed by Fetter [Fetter82], whose “Fourth Man” can be quite human in his movements. In this context, the skeleton is a hierarchical 3D graph of nested transformations. The vertices or control points of the character’s geometric definition are associated with one of the coordinate reference frames defined by the composition of skeleton transformation elements. These matrices are updated during real-time simulation and the corresponding vertices are then transformed.

An example of these ideas used in a real-time application can be seen in the serpentine neck of the simulated pteranadon of Figure 15. It was animated using a version of the skeleton animation technique with a spline curve defining the creatures spine and with the vertices defining the creature’s skin defined in planes tangent to that spline. The application was developed by Greystone Technology and Angel Studios using the software described in [Rohlf94] as a SIGGRAPH 1993 exhibit-hall presentation.

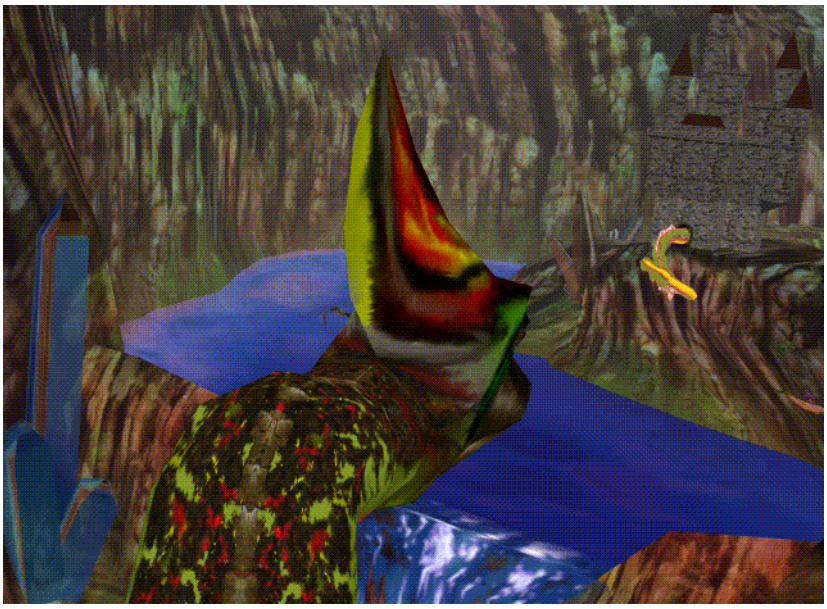


FIGURE 15.

Total Animation

These two techniques—generalized betweening and skeleton animation—can be used in conjunction to create advanced entertainment applications with life-like animated characters. One order of application of the two methods is to first perform a generalized betweening operation that builds a character with the desired pre-planned animation aspects, such as eye or mouth motion and then to set the matrices or other transformation operators of the skeleton transformation operation to represent hierarchical motions such as those of arms or legs. The result of these animation operations is a freshly posed character ready for display. It is expected that total animation techniques will soon become ubiquitous with visual simulation in entertainment applications.

7 Database Construction

One difficulty that visual simulation pioneers have noted but been unable to completely resolve is the complexity of database construction. Special tools have been built to address this need. Tools that allow interactive design of geometry, easy editing and placement of texture images, flexible file-based instancing, and many other operations. Special-purpose tools also exist to aid in the design of roadways, instrument panels, and terrain surfaces. Even with this support, however, the process is often more arduous than expected.

The reward of building complex databases that accurately and efficiently represent the desired virtual environment is great, however, since the value of real-time image generation systems is essentially the datasets that are explored using them. As Chu Hsi said in [Chu]:

> The Ultimate Reality may be likened to a mirror, which, before anything is reflected upon, is simply vacuous, …but as for its function, what is most vacuous in it can be made to show signs of all that is lovely and of all that is homely in the world.

While Chu's "Ultimate Reality" was certainly not a forward reference to any buzz-words of *our* century, the idea of graphics systems as mirrors seems apt and underscores the importance of database modeling.

Summary

The lessons learned in the last 30 years of high-end visual simulation are easily seen to be directly applicable to entertainment graphics systems now being developed. The major insights include:

- The importance of low latency and high frame rates;
- The necessity of consistent image update rates;
- The role of level of detail modeling, billboards, and animation sequences to enhance scene detail and fidelity;
- The use of closed-loop control systems to maintain desired frame rates;
- The visual enhancements provided by antialiasing and texture mapping;
- The combination of generalized betweening and skeleton animation to meet the demand for animated characters in entertainment systems; and,
- That database construction is hard.

These visual simulation techniques are now available in workstation environments through a combination of advanced graphics hardware systems [Akeley93] and visual simulation oriented control software [SGI94]. These systems are widely used in entertainment development projects; applications using these techniques are underway for use in home, arcade, location-based, and destination theme park venues.

The sophisticated requirements of these entertainment systems are causing technologies from classic image generation systems to be propagated to very low cost systems. The size of the entertainment market and the advanced techniques it desires are forcing the development of new architectures and technologies for the visual simulation products of the future.

8 References

[Airey90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr., "Towards image realism with interactive update rates in complex virtual building environments", *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2), 1990, pp. 41-50.

[Akeley93] Kurt Akeley, "RealityEngine Graphics", *Computer Graphics*, Proceedings of SIGGRAPH '93, July 1993, pp. 109-116.

[Blinn76] James F. Blinn, and Martin E. Newell, "Texture and Reflection in Computer Generated Images", *CACM*, 19(10), October 1976, pp. 542-547.

[Blinn88] James F. Blinn, "Me and my (Fake) Shadow", *IEEE Computer Graphics and Applications*, 9(1), January 1988, pp. 82-86.

[Burt71] N. Burtnyk and M. Wein, "Computer-generated key-frame animation", *Journal of the Society Motion Picture and Television Engineers,* 80(3), 1971, pp. 149-153.

[Cardul90] Frank M. Cardullo and Yorke J. Brown, "Visual System Lags: The Problem, The Cause, The Cure", *Proceedings of the 1990 IMAGE V Conference*, June 1990, pp. 31-42.

[Carp84] Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method", SIGGRAPH 1984 Proceedings, *Computer Graphics*, Vol 18, Number 3, July 1984, pp. 103-108.

[Cat84] Edwin A. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces", Ph D. Thesis, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, UT, December 1974.

[Chu] Chu Hsi, "Essay on the *Symbolic Punishment* according to the code of Emperor Shun".

[Cos90] Michael A. Cosman, Allen E. Mathisen, and John A. Robinson, "A New Visual System to Support Advanced Requirements", *Proceedings of the 1990 IMAGE V Conference*, June 1990, pp. 371-380.

[Cos92] Michael A. Cosman, "Mission Rehearsal Modeling", *Proceedings of the 1992 IMAGE VI Conference*, July 1992, pp. 413-425.

[Crow81] Franklin Crow, "A comparison of anti-aliasing techniques", *IEEE Computer Graphics and Applications,* 1981, 1(1), pp. 40-48.

[Ferg90] R.L. Ferguson, R. Economy, W. A. Kelley, and P. P. Ramos, "Continuous Terrain Level of Detail for Visual Simulation", *Proceedings of the 1990 IMAGE V Conference*, June 1990, pp. 145-151.

[Fetter82] W. A. Fetter, "A Progression of human figures simulated by computer graphics", IEEE Computer Graphics and Applications, 2(9), 1982, pp. 9-13.

[Fourn82] Fournier, Fussell, and Carpenter, "Computer Rendering of Stochastic Models", *CACM* 25(6), June 1982, pp. 371-384.

[Fuchs80] Henry Fuchs, Kedem, Z, and Bruce Naylor, “On visible surface generation by a priori tree structures”, *Computer Graphics*, Vol 14, Number 3, 1980, pp. 124-133.

[Funk93] Thomas A. Funkhouser, and Carlo H. Séquin, “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments”, *Computer Graphics*, 1993, pp. 247-254.

[Gish91] Walter Gish and Allen Tanner, “Antialiasing Without Supersampling”, *Proceedings of the 13th Interservice/Industry Training Systems Conference*, December 1991, pp. 262-270.

[Green74] Donald P. Greenberg, “Computer Graphics in Architecture”, *Scientific American,* May 1974.

[Greene86] Ned Greene, “Environment Mapping and Other Applications of World Projections”, *IEEE Computer Graphics and Applications*, Vol 6., No. 11, November 1986, pp. 21-30.

[Kalm61] R. E. Kalman and R. S. Bucy, “New Results in Linear Filtering and Prediction Theory”, *Transaction ASME (Journal of Basic Engineering)*, 83D, 1961, pp. 95-108.

[Kenn90] Robert S. Kennedy and Jennifer E. Fowlkes, “What does it mean to say that simulator sickness is *polygenic* and *polysymptomatic*?”, *Proceedings of the 1990 IMAGE V Conference*, June 1990, pp. 45-59.

[Kenn92] Robert S. Kennedy, Kevin S. Berbaum, Martin G. Smith, and Lawrence J. Hettinger, “Differences in simulator sickness symptom profiles in different simulators: application of a field experiment method”, *Proceedings of the 1992 IMAGE VI Conference*, July 1992, pp. 29-39.

[Kent92] James R. Kent, Wayne E. Carlson, and Richard E. Parent, “Shape Transformation for Polyhedral Objects”, Computer Graphics, Vol 26, Number 2, July 1992, pp. 47-54.

[Lathi74] Bhagwandas Pannalal Lathi, *Signals, Systems, and Controls*, Intext, New York, 1974, ISBN 0-7002-2431-9.

[McFarl88] R. E. McFarland, “Transport Delay Compensation for Computer Generated Imagery Systems”, NASA JM-100084, 1988.

[Parke72] F. I. Parke, “Animation of faces”, *Proceedings of the ACM Annual Conference*, Volume 1, 1972.

[Parke75] F. I. Parke, “A model for human faces that allows speech synchronized animation”, *Computers and Graphi*cs, Pergamon Press, 1(1), 1975, pp. 1-4.

[Parke82] F. I. Parke, “Parameterized models for facial animation”, *IEEE Computer Graphics and Applications*. 2(9), 1982, pp. 61-68.

[Riner92] Bruce Riner and Blair Browder, “Design Guidelines for a Carrier-Based Training System”, *Proceedings of the 1992 IMAGE VI Conference*, July 1992, pp. 65-73.

[Rohlf94] John Rohlf and James Helman, “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics”, *Proceedings of SIGGRAPH ‘94*, July 1994.

[Schill91] Andreas Schilling, “A New Simple and Efficient Antialiasing with Subpixel Masks”, *Computer Graphics*, Vol. 25, Number 4, July 1991, pp. 133-141.

[SGI94] Silicon Graphics, *IRIS Performer Programming Guid*e, 1994, Document Number 007-1680-020.

[Teller91] Seth Teller and Carlo Séquin, “Visibility Preprocessing For Interactive Walkthroughs” *Computer Graphics*, Vol. 25, Number 4, July 1991, pp. 61-69.

[Voor94] Douglas Voorhies and Jim Foran, “Reflection Vector Shading Hardware”, Proceedings of SIGGRAPH ‘94, July 1994.

[Wern94] Josie Wernecke, *The Inventor Mentor: Programming Object-oriented 3D Graphics with Open Inventor, Release 2*, Addison-Wesley, ISBN 0-201-62495-8, 1994.

[Will78] Lance Williams, “Casting Curved Shadows on Curved Surfaces”, *Computer Graphics, Proceedings of SIGGRAPH 1978*, pp. 270-274.

Optimization for Real-Time Entertainment Applications on Graphics Workstations

Sharon Rose Clay
Silicon Graphics Computer Systems

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

1 Introduction

Real-time entertainment applications are very sensitive to image quality, performance, and system cost. Graphics workstations provide full product lines with a full range of price points and performance options. At the high end, they provide many traditional Image Generator features such as real-time texture mapping and full scene antialiasing. They can also support many channels, or players, per workstation to offset the cost of getting the high-end features. At the low end, they have entry prices and performance that are often competitive with PCs. Graphics workstations can provide a very powerful, flexible solution with a rich development environment. Additionally, because of binary compatibility across product lines and standards in graphics APIs, graphics workstations offer the possibility of portability of both applications and databases to different and future architectures. However, this power and flexibility increases the complexity for achieving the full quoted performance from such a machine. This paper presents a strategy for performance for developing and tuning real-time graphics applications on graphics workstations.

The following topics are covered:

- typical application requirements for graphics workstations
- multi-processing issues for graphics subsytems
- graphics workstation pipelines and performance trade-offs
- strategies for diagnosing pipeline bottlenecks

- database structure for traversal
- designing and tuning a real-time application
- run-time diagnostics and load-management strategies
- tools for debugging graphics performance

Developing a designed-for-performance application requires understanding the potential performance problems, identifying which factors are limiting performance, and then making the trade-offs to achieve maximum frame rate with the highest quality scene content.

2 Background

Tuning both application and database is an essential part of the development process on a graphics workstation. Traditionally, both high-end image generators and low-end PC graphics platforms have been very restrictive in the type of application that can be supported and the scene content that can be rendered at a given rate. Graphics workstations offer:

- a wide range of graphics subsystems tightly coupled with today's fastest CPUs,
- high-bandwidth connections between the main CPU and the graphics subsystem (1.2GBytes/sec for the Silicon Graphics RealityEngine system bus),
- scalable (binary compatible) product lines and graphics standards for portability to different architectures, and hopefully future architectures,
- native, optimized, standard rendering libraries, such as OpenGL,
- the flexibility of being able to make performance trade-offs to optimize performance, maximize scene content, and minimize cost — careful tuning can yield tremendous results resulting in a flexible, low cost solution with high scene content,
- sophisticated development environments.

Why Tune?

The down-side of these features is that the tuning process is not only essential, it can be complex. Tuning an application to be performance-portable to different architectures is additionally complex. Unfortunately, tuning is one of those tasks that is often put off until the point of crisis.

Top 10 rationalizations for tuning avoidance:

10. The machine specifications should take into account a "real" application, so tuning should not be necessary.
9. We can worry about performance after implementation.
8. If we design correctly, we won't have to tune.
7. We will tune after we fix all of the bugs
   (also known as: The next release will be the performance version)
6. CPUs are going to be faster by the time we release so we don't have to tune our code.
5. We will always be limited by "that other thing" so tuning won't help.

4. The compiler should produce good code so we don't have to
3. We have this guru who will do all of the performance tuning for us.
2. The demo looks pretty fast.
1. Tuning will destroy our beautiful code.

An understanding of tuning issues and methods should hopefully make the above rationalizations unnecessary.

Tune early. Tune often.

Getting Started: Assessing Requirements and Predicting Performance

The first step, both in designing a real-time application and tuning an existing application, is to assess the requirements for image quality and predict the time required to produce that image quality. If there is a large gap between these calculations, trade-offs may have to be made in the application to balance scene content with performance to produce the most compelling result.

One of the most important parameters in the effectiveness of a simulated environment is *frame rate* — the rate at which new images are presented. The faster new frames can be displayed, the smoother and more compelling the animation will be. Constraints on the frame-rate can determine how much time there is to produce a scene.[i]

Entertainment applications typically require a frame rate of at least 20 frames per second (fps.), and more commonly 30fps. High-end simulation applications, such as flight trainers, will accept nothing less than 60fps. If, for example, we allow two milliseconds (msecs.) of initial overhead to start frame processing, one msec. for screen clear or background, and two msecs. for a window of safety, a 60pfs. application has, optimistically, about 11 msecs. to process a frame and a 30fps. application has 28 msecs.

Another important requirement for Visual Simulation and Entertainment applications is minimizing *rendering latency* — the time from when a user event occurs, such as change of view, to the time when the last pixel for the corresponding frame is displayed on the screen. Minimizing latency is also very important for the basic interactivity of non-real time applications.

A Typical Frame

The basic graphics elements that contribute to the time to render a frame are:

i. There are also additional requirements associated with frame-rate, such as low variability of frame rates and the handling of overload conditions when frame rates are missed. These issues are discussed later in Section 9.

screen clear (color and z-buffer clear or reset),

amount of data transferred to the graphics subsystem,

selected attributes for geometry, such as lighting, texturing, atmospheric effects and number of different attribute sets,

viewing transformations of geometry,

the number of pixels produced for the frame (resolution multiplied by *depth complexity*),

video refresh to output final image from framebuffer memory.

An estimation of expected performance should take into account all of these frame components, plus possible overhead due to interactions between components. An estimation of the time in milliseconds required to render a frame will then translate into an expected frame rate.

Screen clear time is like a fixed tax on the time to render a scene and for rapid frame rates, may be a measurable percentage of the frame interval. Because of this, most architectures have some sort of screen clear optimization. For example, the Silicon Graphics RealityEngine™ has a special screen clear that is well under one millisecond for a full high-resolution framebuffer (1280x1024). Video-refresh also add to the total frame time and is discussed in Section 4.

The size and contents of full databases vary tremendously among different applications. However, for context, we can guess at reasonable hypothetical scene content, given the high frame rates required for real-time graphical applications and current capabilities of graphics workstations.

The number of polygons possible in a 60fps. or 30fps. scene is affected by the many factors discussed in this paper, but needless to say, it can be quite different than the peak polygon transform rate of a machine. Current graphics workstations can manage somewhere between 1500 and 5000 triangles at 60pfs. and 7000-10,000 triangles at 30fps. Typical attributes specified for triangles include some combination of normals, colors, texture coordinates, and associated textures. For entertainment applications, the amount of dynamic objects and geometry changing on a per-frame basis is probably relatively high. For handling general dynamic coordinate systems of moving objects, matrix transforms are most convenient. Such objects usually also have relatively high detail (50-100 polygons). These numbers imply that we can easily imagine having half to a full megabyte of just geometric graphics data per frame.

Depth-complexity is the number of times, on average, that a given pixel is written. A depth-complexity of one means that every pixel on the screen is touched one time. This is a resolution-independent way of measuring the fill requirements of an application. Visual simulation applications tend to have depth-complexity between two and three for high-altitude applica-

tions, and depth-complexity between three and five for ground-based applications. Depth-complexity can be reduced through aggressive database optimizations, discussed in Section 7. Resolutions for visual simulation applications also vary widely. For entertainment, VGA(640x480) resolution is common. A 60fps. application at VGA resolution with depth-complexity five will require a fill rate of 100 million pixels per second (MPixels/sec.). In a single frame, there may can easily be one-two million pixels that must be processed.

The published specs of the machine can be used to make a rough estimate of predicted frame rate for the expected scene content. However, this prediction will probably be very optimistic. Performance prediction is covered in detail in Section 5. An understanding of the graphics architecture enables more realistic calculations.

Graphics Architectures

The type of system resources available and their organization has a tremendous effect on the application architecture. Architecture issues for graphics subsystems are discussed in detail in Section 3 andSection 4.

On traditional image-generators, the main application is actually running on a remote host with a low-bandwidth network connection between the application running on the main CPU and the graphics subsystem. The full graphics database resides in the graphics subsystem. Tuning applications on these machines is a matter of tuning the database to match set performance specifications. At the other extreme, we have PCs. Until recently, almost all of the graphics processing for PCs has traditionally been done by the host CPU and there has been little or no dedicated graphics hardware. Recently, there have been many new developments in this area with dedicated graphics cards developed by independent vendors for general PC buses. Some of these cards have memories for textures and even resident databases.

Graphics workstations fall between these two extremes. They traditionally have separate processors that make up a dedicated graphics subsystem.They may also have multiple host CPUs. Some workstations, such as Silicon Graphics, have a tightly coupling between the CPU and graphics subsystems with system software, compilers, and libraries. However, there are also independent vendors, such as EvansSutherland, Division, and Kubota, producing both high and low end graphics boards for general workstations.

The growing acceptance and popularity of standards for 3D graphics APIs, such as OpenGL™, is making it possible to develop applications that are portable between vastly different architectures. Performance, however, is typically not portable between architectures so an application may still require significant tuning (rewriting) to run reasonable on the different

platforms. In some cases, the standard API library may have to be bypassed altogether if it is not the fastest method of rendering on the target machine. For the Silicon Graphics product line, this has been solved with a software application layer that is specifically targeted at real-time 3D graphics applications and gives peak performance across the product line [Rohlf94]. Writing/tuning rendering software is discussed in Section 5.

A common thread is that multiprocessing of some form has been a key component of the high-performance graphics platforms and is working its way down to the low-end platforms.

3 Multi-Processing for High-Performance Graphics

A significant part of designing and tuning an application is determining the best way to utilize the system processing resources and determining if additional system resources will benefit performance. Any tuning strategy requires an understanding of how the different components in a system interact to affect performance. There are many elements to system performance beyond the guiding frames-per-second:

Throughput — maximizing frame rates is equivalent to maximizing throughput: producing the most output possible per unit of time.

Bandwidth — The major datapaths through the system must have sufficient bandwidth or entire parts of a system may be under-utilized. The connections of greatest concern would be 1) that between the host computer and the graphics subsystem, 2) the paths of access to database memory and 3) disk access for the application and the graphics subsystem. It is particularly important that bandwidth specs not assume tiny datasets that will not scale to a real application.

Processor utilization — Will the system get good utilization of the available hardware or will some processors be sitting idle while others are overloading (will you get what you paid for). Good processor utilization is essential for a system to realize its potential throughput. Achieving this in a dynamic environment requires load balancing mechanisms.

Scalability — If performance is a problem, will the system support the addition of extra processors to improve throughput, and will improved performance scale with the addition of new processors. Additionally, as new processors are added, will load-balancing enable a real application to see the improved performance, or will it only show up in benchmarks.

Latency — What is the maximum interval of time from when a user initiated an input and the moment the final pixel of the corresponding new frame is presented. Low latency is critical to interactive real-time entertainment applications.

Synchronization overhead — how much overhead is incurred when tasks communicate information. This is particularly an issue for the very dynamic database of an interactive, real-time entertainment application: both the main application and the graphics subsystem need efficient access to the current state of the database.

These measures of performance can be applied to both the system as a whole, and to individual subsystems.

Methods of Multiprocessing

Because graphics applications have many very different tasks that must be executed every frame, they are well suited to division among multiple tasks, and multiple processors if available. Multiprocessing can also be used to achieve better utilization and throughput of a single processor.

The partitioning and ordering of the separate tasks has direct consequences on the performance of the system. A task may be executed in a pipelined, or in a concurrent fashion. *Pipelining* uses an assembly-line model where a task is decomposed into stages of operations that can be performed sequentially. Each stage is a separate processor working on a separate part of a frame and passing it to the next stage in the line. *Concurrent* processing has multiple tasks simultaneously working on different parts of the same input, producing a single result.

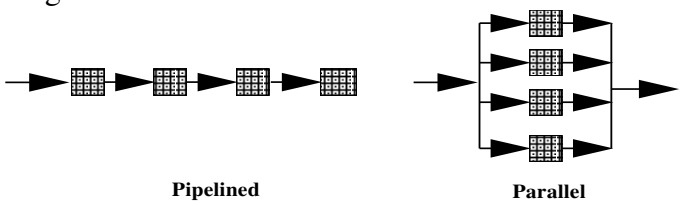


FIGURE 1. Pipelined vs. Parallel Processors

Both the host and graphics subsystems may employ both pipelining and parallelism as a way of using multiple processors to achieve higher performance. The general theoretical multiprocessing issues apply to both graphics applications and graphics subsystems. Additionally, there are complexities that arise with the use of special purpose processors, and from the great demands of graphics applications.

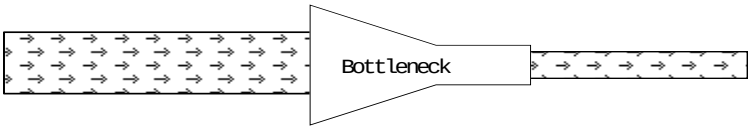
Many graphics tasks are easily decomposed into pipelined architectures. Typically, there is a main graphics pipeline, with parallelism within stages of the pipeline. Individual pipeline stages may themselves be sub-pipelines, or have parallel concurrent processors. Additionally, there may be multiple parallel graphics pipelines working concurrently.

Pipelining vs. Parallelism

Pipelined systems need minimal synchronization because each task is working on its own data for a different frame, or part of a frame, in an ordered fashion so synchronization is implicit. Longer pipelines have increased throughput — producing new results in quick succession because the task has been broken up into many trivial stages that all execute quickly. However, each stage in a pipeline will add latency to the system

because the total amount of time through the pipeline is the number of stages multiplied by the step time, which is the speed of the slowest stage. While every step produces a new output, the total amount of time to produce a single output may get longer. The addition of a new pipeline stage will presumably decrease the step time, but probably not enough to avoid overall increased latency.

Pipelined systems will always run at the speed of the slowest stage, and no faster. The limiting stage in a pipelined system is appropriately called a *bottleneck*.



Pipeline tuning amounts to determining which stage in the pipeline is the bottleneck and reducing the work-load of that stage. This can be quite difficult in a graphics application because through the course of rendering a frame, the bottleneck changes dynamically. Furthermore, one cannot simply take a snapshot of the system to see where the overriding bottleneck is. Finally, improving the performance of the bottleneck stage can actually reduce total throughput if the another bottleneck results elsewhere. Bottleneck tuning methods are discussed in Section 5.

Tune the slowest stage of the pipeline.

Concurrent architectures do not suffer from the throughput vs. latency trade-off because each of the tasks will directly produce part of the output. However, synchronization and load-balancing are major issues. If processors are assigned to separate tasks that can be run in parallel, then there is the chance that some tasks will take very little time to complete and those processors will be idle. If a single task is distributed over several processors, then there is the overhead of starting them off and recombining the output results. However, the latter has a better chance of producing an easily-scalable system because repetitive tasks, such as transforming vertices of polygons, can be distributed among multiple concurrent processors. Concurrent parallel architectures are also easier to tune because it is quite apparent who is finishing last.

SIMD vs. MIMD

The processor organization in the system also needs to be considered. There are two types of processor execution organization: SIMD or MIMD. SIMD (single instruction multiple data) processors operate in lock-step where all processors in the block are executing the same code. These processors are ideal for the concurrent distributed-task model and require less

overhead at the start and end of the task because of the inherent constraints they place on the task distribution. SIMD processors are common in graphics subsystems. However, MIMD (multiple instruction multiple data) do better on complex tasks that have many decision points because they can each branch independently. As with pipelined architectures, the slowest processor will limit the rate of final output.

In actual implementation, graphics architectures are a creative mix of pipelining and concurrency. There may be parallel pipelines with the major pipeline stages implemented as blocks of parallel processors.

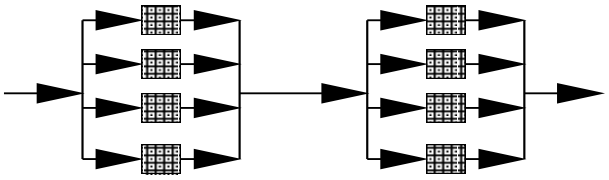


FIGURE 2. Parallel Pipeline

Individual processors may then employ significant sub-pipelining within the individual chips. Systems may be made scalable by allowing the ability to add parallel blocks.

4 Performance Issues in Graphics Pipelines

This section briefly reviews the basic rendering processes in the context of presenting the basic computational requirements. Additionally, ways in which the rendering task can be partitioned for implementation in hardware and corresponding performance trade-offs are also discussed. Tuning an application to a graphics pipeline is discussed in detail in Section 5.

The task of rendering three-dimensional graphics primitives is very demanding in terms of memory accesses, integer, and floating-point calculations. There are impressive software rendering packages that handle three dimensional texture-mapped geometry and can generate on the order of 1MPixels/sec on current CPUs. However, the task of rendering graphics primitives is very naturally suited to distribution among separate, specialized pipelined processors. Many of the computations that must be performed are also very repetitive, and so can take advantage of parallelism in a pipeline. This use of special-purpose processors to implement the rendering process is based on some basic assumptions about the requirements of a typical target application. The result can be orders of magnitude increases in rendering performance.

The Rendering Pipeline

The rendering process naturally lends itself to a simple pipeline abstraction. The *rendering pipeline* can generally be thought of as having three main stages:

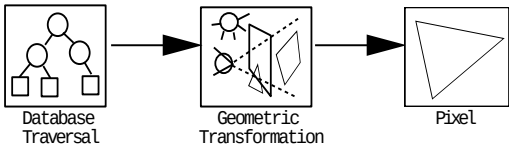


FIGURE 3. The Rendering Pipeline

Each of these stages may be implemented as a separate subsystem. These different stages are all working on different sequential pieces of rendering primitives for the current frame. A more detailed picture of the rendering pipeline is shown in Figure 4. An understanding of the computations that occur at each stage in the rendering process is important for understanding a given implementation and the performance trade-offs made in that implementation. The following is an overview of the basic rendering pipeline, the computational requirements of each stage, and the performance issues that arise in each stage[i][Foley90,Akeley93,Harrell93,Akeley89].

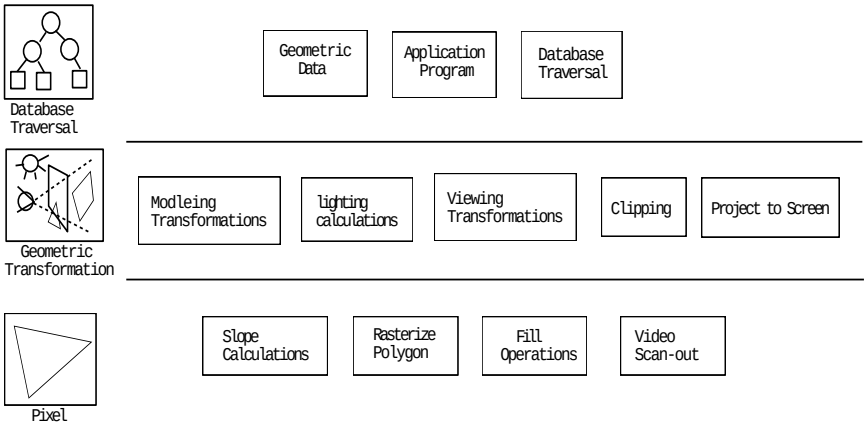


FIGURE 4. The Detailed Stages of the Rendering Pipeline

i. See [Foley90], Chapter 18, for a detailed discussion of this topic.

The CPU Subsystem (Host)

At the top of the graphics pipeline is the main real-time application running on the host. If the host is the limiting stage of the pipeline, the rest of the graphics pipeline will be idle.

The graphics pipeline might really be software running on the host CPU. In which case, the most time consuming operation is likely to be the processing of the millions of pixels that must be rendered. For the rest of this discussion, we assume that there is some dedicated graphics hardware for the graphics subsystem.

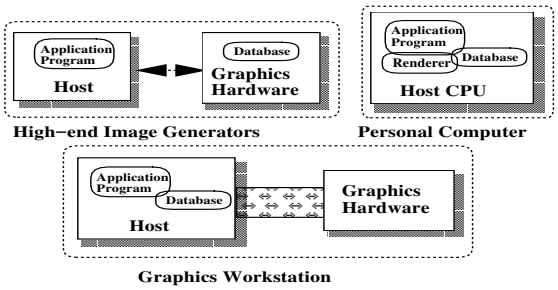


FIGURE 5. Host-Graphics Organizations

Database Traversal

The application may itself be multiprocessed and running on one or more CPUs. The host and the graphics pipeline may be tightly connected, sharing a high speed system bus, and possibly even access to host memory. Such buses currently run at several hundred MBytes/sec, up to 1.2GBytes/sec. However, in many high-end visual simulation systems, the host is actually a remote computer that drives the graphics subsystem over a network (SCRAMnet, 100 Mbits/sec, or even ethernet at 10Mbits/sec).The first stage of the rendering pipeline is traversal of the database and sending the current rendering data on to the rest of the graphics pipeline. In theory, the entire rendering database, or *scene graph*, must be traversed in some fashion for every frame because both scene content and viewer position are dynamic. Because of this, there are three major parts of the database traversal stage: processing to determine current viewing parameters (usually part of the main application), determining which parts of the scene graph are contained within the viewing frustum (culling), and the actual drawing traversal that issues rendering commands for the visible parts of the data-

base. These components form a traversal pipeline of three stages: Application, Cull, and Draw:

FIGURE 6. Application Traversal Process Pipeline

Possibilities for the application processes are discussed further in Figure 6. This section will be focussing on the drawing traversal stage of the application.

Some graphics architectures impose special requirements on the drawing traversal task, such as requiring that the geometry be presented in sorted order from front to back, or requiring that data be presented in large, specially formatted chunks as display lists.

There are three main types of database drawing traversal:

immediate mode,
display list mode,
retained data.

Immediate Mode Drawing Traversal

In the first two, the rendering database lives in main memory. For immediate mode rendering, the database is actually shared with the main application on the host, as shown in Figure 7. The application is responsible for traversing the database and sending geometry directly to the graphics pipeline. This mode is the most memory efficient and flexible for dynamic geometry. However, the application is directly responsible for the low-level communication with the graphics subsystem.

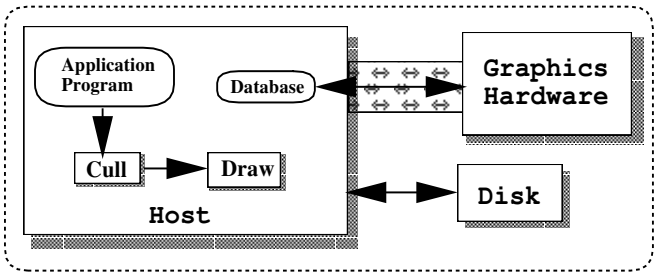


FIGURE 7. Architecture with Shared Database

Display List Traversal

In display-list mode, pieces of the database are compiled into static chunks that can then be sent to the graphics pipe. In this case, the display list is a separate copy of the database that can be stored in main memory in an optimized form for feeding the rest of the pipeline. The database traversal task is to hand the correct chunks to the graphics pipeline. These display lists can usually be edited, or re-created easily for some additional performance cost. For both of these types of drawing traversals, it is essential that the application be using the fastest possible API for communication with the graphics subsystem. An inefficient host-graphics interface for such operations as issuing polygons and vertices could leave the rest of the graphics pipeline starved for data.

Use only the fastest interface and routines when communicating with the graphics pipeline.

There is potentially a significant amount of data that must be transferred to the graphics pipeline every frame. If we consider just a 5K triangle frame, that corresponds to

(5000 tris) * (3 vertices/tri * (8 floats/vertex) * (4bytes/float)) = 480KBytes

→ 28.8 MBytes/sec for a 60fps. update rate.

for just the raw geometric data. The size of geometric data can be reduced through the use of primitives that share vertices, such as triangle strips, or through the use of high-level primitives, such as surfaces, that are expanded in the graphics pipeline (this is discussed further in Section 7). In addition to geometric data, there may also be image data, such as texture maps. It is unlikely that the data for even a single frame will fit a CPU cache so it is important to know the rates that this data can be pulled out of main memory. It is also desirable to not have the CPU be tied up transferring this data, but to have some mechanism whereby the graphics subsystem can pull data directly out of main memory, thereby freeing up the CPU to do other computation. For highly interactive and dynamic applications, it is important to have good performance on transfers of small amounts of data to the graphics subsystems since many small objects may be changing on a per-frame basis.

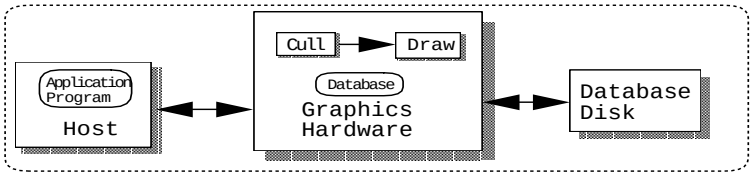


FIGURE 8. Architecture with Retained Data

Retained Database Traversal

If the database, and display list, is stored in the graphics pipeline itself, as shown in Figure 8, separate from main memory, it is a retained display list. Retained display lists are traversed only by the graphics pipeline and are required if there is very low bandwidth between the host and the graphics subsystem. The application only sends small database edits and updates (such as the new viewing parameters and a few matrices) to the graphics pipe on a per-frame basis. Pieces of the database may be paged directly off local disks (also at about 10MBytes/sec.). Retained mode offers less much flexibility and power over editing the database, but also can remove the possible bandwidth bottleneck at the head of the graphics pipeline.

The use of retained databases can enable additional processing of the total database by the graphics subsystem. For example, partitioning the database may be done order to implement sophisticated optimization and rendering techniques. One common example is the separation of static from moving objects for the implementation of algorithms requiring sorting. The cost may be additional loss of power and control over the database due to limitations on database construction, such as the number of moving objects allowed in a frame.

Graphics Subsystems

The Geometry Subsystem

The second two stages of the rendering pipeline, Figure 3, are commonly called *The Geometry Subsystem* and *The Raster Subsystem*, respectively. The geometry subsystem operates on the geometric primitives (surfaces, polygons, lines, points). The actual operations are usually per-vertex operations. The basic set of operations and estimated computational complexity includes [Foley90]: modeling transformation of the vertices and normals from eye-space to world space, per-vertex lighting calculations, viewing projection, clipping, and mapping to screen coordinates. Of these, the lighting calculations are the most costly. A minimal lighting model typically includes emissive, ambient diffuse, and specular illumination for infinite lights and viewer. The basic equation that must be evaluated for each color component (R, G, and B) is [OpenGL93]:

RGBemissive_mat +
RGBambient_model*RGBambient_mat +
RGBambient_mat*RGBambient_light +
RGBdiffuse_mat*RGBdiffuse_light * (light_vector. normal)

Specular illumination adds an additional term (exponent can be approximated with table lookup):

$$RGBspecular\_light*RGBspecular\_mat*(half\_angle.\ normal)^{shininess}$$

Much of this computation must be re-computed for additional lights. Distance attenuation, local viewing models and local lights add significant computation.

A trivial accept/reject clipping step can be inserted before lighting calculations to save expensive lighting calculations on geometry outside the viewing frustum. However, if an application can do a coarse cull of the database during traversal, a trivial reject test may be more overhead than benefit. Examples of other potential operations that may be computed at this stage include primitive-based antialiasing and occlusion detection.

This block of floating-point operations is an ideal case for both sub-pipelining and block parallelism. For parallelism, knowledge about following issues can help application tuning:

MIMD vs. SIMD,

how streams of primitives are distributed to the processors,

is the output of these processors remain separate in parallel streams for the next major stage, or re-combined into a single output stream for re-distribution.

The first issue, MIMD vs. SIMD, affects how a pipeline handles changes in the primitive stream. Such changes might include alterations in primitive type, state changes, such as the enabling or disabling of lighting, and the occurrence of a triangle that needs to be clipped to the viewing frustum. SIMD processors have less overhead in their setup for changes. However, since all of the processors must be executing the same code, changes in the stream can significantly degrade processor utilization, particularly for highly parallel SIMD systems. MIMD processors are flexible in their acceptance of varied input, but can be more somewhat more complex to setup for given operations, which will include the processing state changes. This overhead can also degrade processor utilization. However adding more processors can be added to balance the cost of this overhead.

The distribution of primitives to processors can happen in several ways. An obvious scheme is to dole out some fixed number of primitives to processors. This scheme also makes it possible to easily re-combine the data for another distribution scheme for the next major stage in the pipeline. MIMD

processors could also receive entire pieces of general display lists, as might be done for parallel traversal of a retained database.

The application can affect the load-balancing of this stage by optimizing the database structure for the distribution mechanism, and controlling changes in the primitive stream.

Order rendering to benefit the most expensive pipeline stage.

After these floating point operations to light and transform geometry, there are fixed-point operations to calculate slopes for the polygon edges and additional slopes for z, colors, and possibly texture coordinates. These calculations are simple in comparison to the floating point operations. The more significant characteristic here is the explosion of vertex data into pixel data and how that data is distributed to downstream raster processors.

The Raster Subsystem

There is an explosion of both data and processing that is required to rasterize a polygon as individual pixels. Typically, these operations include depth comparison, gouraud shading, color blending, logical operations, texture mapping and possibly antialiasing. These operations require accessing of various memories, both reading for inputs to comparisons and the blending and texturing operations, and writing of the updated depth and color information and status bits for logical operations. In fact, the memory accesses can be more of a performance burden than the simple operations being computed. Of course, this is not true if complex per-pixel shading algorithms, such as Phong Shading, are in use. For antialiasing methods using super-sampling, some of these operations (such as z-buffering) may have to be done for each sub-sample. For interpolation of pixel values for antialiasing, each pixel may also have to visit the memory of its neighbors. Texture interpolation for smoothing the effects of minification and magnification can also cause many memory accesses for each pixel. An architecture might choose to keep some memory local to the pixel processor, in which case, fill operations that only access local processor memory would probably be faster.

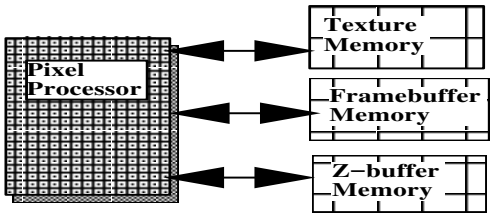


FIGURE 9. Pixel Operations do many Memory Accesses

The vast number of pixel operations that must be done would swamp single-processor architectures, but are ideal for wide parallelism. The Silicon Graphics RealityEngine™ has 80 Image Processors on just one of up two four raster subsystem boards. The following is a simplified example of how parallelism can be achieved in the Raster Subsystem. The screen is subdivided into areas for some number of rasterizing engines that take polygons and produce pixels. Each rasterizer has a number of pixel processors that are each responsible for a sub-area of its parent rasterizing engine and writes directly into framebuffer memory. Thus, the Raster Subsystem may have concurrent sub-pipelines.

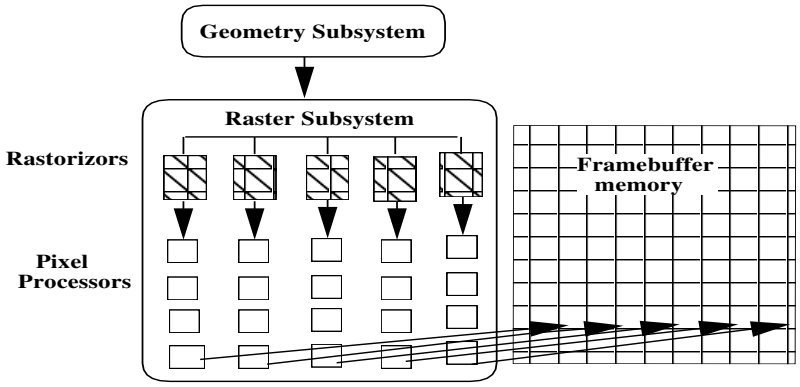


FIGURE 10. Parallelism in the Raster Subsystem

In some architectures, such as the Silicon Graphics VGX ™, the rasterizers get interleaved vertical spans on the screen and the pixel processors get interleaved pixels within those spans. The Silicon Graphics RealityEngine™ uses a similar scheme, but with more complex interleaving for better load balancing and many more pixel processors[Akeley93].

Certain operations that can cause an abort of the writing to a pixel, such as a failed z-buffer test, can be used short-circuit further more expensive pixel operations. If the application can draw front to back, or draw large front polygons first, a speedup might be realized.

Depending on the distribution strategy, MIMD processors in this stage might show more benefit from such short-circuit operations. The distribution strategy typically employs an interleaved partitioning of the framebuffer. This optimizes memory accesses and promotes good processor utilization. The possible down side is that most processors will need to see most primitives. The complexity in figuring out which primitive go to which processors may cause processors to receive input for which they do no work. Because of this overhead, small polygons can have less efficient fill characteristics.

Bus Bandwidth

The bottleneck of a pipeline may not be one of the actual stages, but in fact one of the buses connecting two stages, or logic associated with it. There may be logic for parsing the data as it is comes off the bus, or distributing the data among multiple downstream receivers. Any connection that must handle a data explosion, such as the connection between the Geometry and Raster subsystems, is a potential bottleneck. The only way to reduce these bottlenecks is to reduce the amount of raw data that must flow through it, or to send data that requires less processing. The most important connection is the one that connects the graphics pipeline to the host because if that connection is a bottleneck, the entire graphics pipeline will be under-utilized.

The use of FIFO buffers between pipeline stages provides necessary padding that protects a pipeline from the affects of small bottlenecks and smooths the flow of data through the pipeline. Large FIFOs at the front of the pipeline and between each of the major stages can effectively prevent a pipeline from backing up through upstream stages and sitting idle while new data is still waiting at the top to be presented. This is useful important for fill-intensive applications which tend to bottleneck the very last stages in the pipeline. However, once a FIFO fills, the upstream stage will back up.

Fill the pipeline from back to front.

Video Refresh

The final stage in the frame interval is the time spent waiting for the video scan-out to complete for the new frame to be displayed. This period, called a *field*, is the time from the first pixel on the screen until the last pixel on the screen is scanned out to video. For a 60Hz video refresh rate, the time could be as much as 16.7msecs. Graphics workstations typically use a double-buffered framebuffer so that for an extra field of latency, the system can achieve frame-rates equal to the scan-out rate. A double-buffered system will toggle between two framebuffers, outputting the contents of one framebuffer while the other is receiving rendering results. The framebuffers cannot be swapped until the previous video refresh has completed. This will force the frame rate of the application to run at a integer multiple of the video refresh rate. In the worst case, if the rendering for one frame completed just after a new video refresh was started, the application could theoretically have to wait for the entire refresh period, waiting for an available framebuffer to receive rendering for the next frame.

A double-buffered application will always have a frame rate that is an integer multiple of the video refresh rate.

Dealing with Latency

The time for video refresh is also the lower bound on possible latency for the application. The typical double-buffered application will have a minimum of two fields of latency: one for drawing the next frame while the current one is being scanned, and then a second field for the frame to be scanned. This assumes that the frame rate of the application is equal to the field rate of the video. In reality, a double-buffered application will have a latency that is at least

$$2 * N * field\_time$$

where N is the number of fields per application frame.

One obvious way to reduce rendering latency is to reduce the frame time to draw the scene. Another method is to allow certain external inputs, namely viewer position, into later stages of the graphics pipeline. An interesting method to address both of these problems is presented in [Regan94]. A rendering architecture is proposed that handles viewer orientation after rendering to reduce both reduce latency and drawing. The architecture renders a full encapsulating view around the viewer's position. The viewer orientation is sampled after rendering by a separate pipeline that runs at video refresh rate to produce the output RGB stream for video. Additionally, only objects that are moving need to be redrawn as the viewer changes his orientation. Changes in viewer position could also be tolerated by setting a maximum tolerable error in object positions and sizes. Complex objects could even be updated at a slower rate than the application frame rate since their previous renderings still update correctly with viewer orientation.

These principles of sampling viewer position as late as possible, and decoupling of object rendering rate from viewer update rate can also be applied to applications.

Graphics Architectures

It is always fascinating to see the creative ways in which basic concepts can be applied to produce vastly different architectures. The following are brief examples of a few different past and current graphics architectures with different price/performance characteristics.

Apollo DN10000[Voorhies89] — A complete graphics workstation. RISC concepts were applied to produce a RISC graphics-capable CPU that could be combined in a multiprocessor configuration. All traversal and geometry processing are done on the CPU(s).

SGI Extreme[Harrell93] — desktop workstation with 500K triangles/sec and 80MPixels gouraud-zbuffered. The geometry subsystem is 8 SIMD parallel processors, raster subsystem is 2 parallel processing blocks. Hyper-pipelining in the raster subsystem is used to increase performance of pixel generation. Texture-mapping done in software on the host.

Silicon Graphics high-end graphics workstations — three generations of graphics pipelines native to a multiprocessor host:

GTX[Akeley89] — has a highly parallel Geometry Subsystem with five stages, and a raster subsystem that includes parallelized stages for scan-conversion and pixel processing.

VGX[Akeley90] — achieves high polygon rates (first workstation to do 1 Million triangles) with a simpler 3-stage geometry subsystem where two of the stages use four SIMD processors each. Data was combined at the output of the geometry subsystem for re-distribution to the raster subsytems which were MIMD, parallelized and extendable. Texture mapping is supported at reduced fill rates. VGX-Skywriter supported multiple graphics pipelines in one (large) box that could act as independent parallel pipelines, or could be used together as frame-interleaved pipelines to effectively double throughput (but with additional latency).

RealityEngine[Akeley93] — has a simpler geometry subsystem that is 12 MIMD processors wide and has an even more extendable raster subsystem. Up to three independent pipelines may exist in a single box. High polygon rates (1.2M, 1.8M on RE2) were in balance with high fill rates for complex fill algorithms, including tri-linear texture mapping and real-time full-scene antialiasing. Also supports per-window stereo.

Kubota Denali(93) — scalable graphics subsystem that sits on the main DEC Alpha system bus. Has main stages of 3-stage Geometry Transformation Module and Framebuffer Module(SIMD); both are extendable for greater performance. Has low to high end configurations with 200K and 1.2M triangles/sec, respectively, and supports texture mapping at reduced fill rates.

Sun Leo[Deering93] — desktop workstation capable of 210K 100 pixel triangles/sec. Geometry subsystem uses four parallel custom floating point processors and the raster subsystem uses five parallel processors. Leo is VRAM-access limited, and thus fill-limited, even for 100 pixel triangles. Supports per-window stereo.

Evans & Sutherland Image Generators — use a remote networked host and retained database traversal. They have special support for many visual-simulation features, including special terrain traversal and rendering algorithms with dynamic Level of Detail (LOD), landing lights, and various atmospheric affects. The ESIG4000(high-end) has special support for the placement of LOD geometry on LOD terrain and distortion correction for projection onto curved display surfaces. Restrictions on databases include: the databases must be built with special modeling tools and use separating-plane construction for peak performance and an associated limited number of moving models allowed and animated forms of dynamic objects.

PixelPlanes[Fuchs89] explores the use of massively parallel processors for the rendering task. A MIMD array of geometry processors sorts polygons into screen-based region buffers (128x128 pixels) and SIMD pixel processors apply pixel-based rendering algorithms, such as Phong shading. These sophisticated shading calculations are deferred until an end-of-frame point, after zbuffering, so they only have to be done once per pixel and are not affected by depth-complexity. All geometry processing must be done before the deferred shading pass. Screen-based allocation does not provide good load-balancing.

PixelFlow[Molnar93] improves this load balancing. It is an image composition architecture where rendering of primitives is distributed over an extendable, massively parallel array of complete renderers — each able to draw to the entire framebuffer, but still rasterizes in 128x128 pixel patches. As in PixelPlanes, deferred

shading is applied after the renderer images are composited. PixelFlow can run in either a retained database mode, or in an immediate mode where it is hosted by a parallel computer with processors directly connected to the renderers.

5 Optimizing Performance of a Graphics Pipeline

This section discusses tuning an application for a graphics pipeline. The *Law of Diminishing Returns* definitely applies here: for an application designed with performance in mind, it is typically fairly easy to achieve about 60% of expected optimal performance. A bit of work can get you to 75% or 80% and then it starts to get more difficult. A major reason for this is the pure complexity of having so many parameters interacting which further affects performance behavior. The key is in identifying and isolating the current problems. The goal is a balanced pipeline where no one stage is an overwhelming bottleneck.

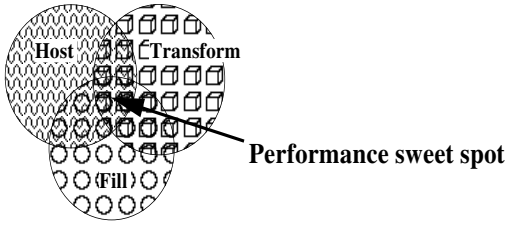


FIGURE 11. A Balanced Pipeline

The focus of this section is the development of the following basic tuning strategy:

1. Design for performance
2. Estimate expected performance
3. Measure and evaluate current performance
4. Isolate performance problems
5. Balance operations
6. Repeat

Design For Performance

The full system pipeline should be kept in mind when designing the application. Graphics features should be chosen to balance the pipeline and careful estimations of expected performance for target databases should be made during the design phase as well as the tuning phase.

Selecting Features

Combinations of rendering features should be chosen to produce a balanced pipeline. An advantage of graphics workstations is the power to make trade-offs to maximize both performance and scene quality for a

given application. If, for example, a complex lighting feature is required that will bottleneck the geometry subsystem, then possibly a more interesting fill algorithm could be used to both require less polygons being lit and achieve overall higher scene quality.

Beware of features that use multi-pass algorithms because pipelines are usually balanced with one pass through each stage. There are many sophisticated multi-pass algorithms incorporating such techniques as texture-mapping, Phong-shading, accumulation antialiasing, and other special effects, that produce high-quality images. Such features should be used sparingly and their performance impact should be well understood.

Designing Tasks

The application should also be designed with multiprocessing in mind since this is very hard to add after-the-fact. Large tasks that can be run on separate processors (preferably with minimal synchronization and sharing of data) should be identified. For ease of debugging, portability, and tuning (discussed further in Section 8) the application should support both a single process mode, and a mode where all tasks are forced into separate processes.

Design with multiprocessing in mind.

The tasks also need to be able to non-invasively monitor their own performance, and need to be designed so that they will support measurements and experiments that will need to be done later for tuning. The rendering task (discussed later in this section) must send data to the graphics pipeline in a form that will maximize pipeline efficiency. Overhead in renderer operations should be carefully measured and amortized over on-going drawing operations.

Estimating Performance for a Pipeline

Making careful performance estimations greatly enhances your understanding of the system architecture. If the target machine (or similar machine) is available, then this should be done in tandem with the analysis of current application performance and the comparison to small benchmarks until the measurements and estimations agree.

As should not be surprising by this time, estimating performance of an application for a pipeline is much more than looking at peak quoted numbers for a machine and polygon totals for a database. The following are basic steps for estimating performance:

1. Define the contents of a worst-case frame, including number of polygons and their types, number of graphics modes and changes, and average polygon sizes
2. Identify the major stages of the graphics pipeline

3. For each major stage, identify the parts of the frame that are significant for that stage
4. Estimate the time that these frame components will spend in each stage of the pipeline (if possible, verify with small benchmarks)
5. Sum the maximum stage times computed in (4) for a very pessimistic estimation
6. When the drawing order can be predicted (such as for screen clear which may be expensive in the fill stage and will always come first) more optimistic estimations can be made by assuming that time spent in upstream stages for later drawing will be concurrent with the downstream work, and thus gotten for free.

First, consider the geometry subsystem: the significant operations might be triangles (with better rates for meshed triangles), lighting operations, clipping operations, mode changes, and matrix transformations. Given this information, one can compile the following type of information:

- Triangles: percentage of triangles in meshes of different lengths
- Graphics modes: percentage of the triangles that are lit
- Mode changes: number of texture changes per frame
- Viewing Matrix transformations: number per frame
- Clipping: percentage of triangles that are trivial accept, are trivial reject, and those that intersect the viewing frustum

Given the scene characteristics, one should first write small benchmarks to get geometry subsystem performance statistics on individual components. Then, write an additional benchmark that mimics the geometry characteristics of a frame to evaluate the interactions of those components.

We then similarly examine the raster subsystem. We first need to know the relevant frame information:

- Depth complexity of the frame and target resolution
- Number of triangles in different fill modes and a few typical sizes
- Number of raster mode changes
- Time for screen clear

Again, if possible, benchmarks should be written to verify the fill rates of polygons and the cost of raster mode changes. From these estimates, one can make a best guess about the amount of time that will be spent in the raster subsystem.

We can now make coarse-grained and fine-grained estimations of frame time. An extremely pessimistic approach would be to simply add the bottleneck times for the geometry subsystem and the raster subsystem. However, if there is a sufficient FIFO between the geometry and raster subsystems, much of the operations in the geometry subsystem should overlap with the raster operations. Assuming this, a more optimistic

coarse-grained estimation would be to sum the amount of time spend in the raster subsystem and the amount of time beyond that required by the geometry subsystem. A fine-grain approach would be to consider the bottlenecks for different types of drawing. Identify the parts of the scene that are likely to be fill-limited and those that are likely to be transform-limited. Then sum the bottleneck times for each.

Measuring Performance and Writing Benchmarks

Being able to make good performance measurements and write good benchmarks is essential to getting that last 20% of performance. To achieve good timing measurements, do the following:

1. Take measurements on a quiet system. Graphics workstations have fancy development environments so care must be taken that background processes, such as a graphical clock ticking off seconds, or a graphical performance monitor, etc., are not disrupting timing.
2. Use a high-resolution clock and make measurements over a period of time that is at least 100x the clock resolution.
3. If there are only very low resolution timers (less than 1millisecond) then to accurately time a frame, pick a static frame (freeze matrix transformations), run in single-buffered mode, and time the repeated drawing of that frame.
4. Make sure that the benchmark frame is repeatable so that you can return to this exact frame to compare the affects of changes.
5. Make sure that pipeline FIFOs are empty before starting timing and then before checking the time at the end of drawing. When using OpenGL, one should call **`glFinish()`** before checking the clock.
6. Verify that you can rerun the test and get consistent timings.

A generally good technique for writing benchmarks is to always start with one that can achieve a known peak performance point for the machine. If you are writing a benchmark that will do drawing of triangles, start with one that can achieve the peak triangle transform rate. This way, if a benchmark seems to be giving confusing results, you can simplify it to reproduce the known result and then slowly add back in the pieces to understand their effect.

Verify a known benchmark on a quiet system.

When writing benchmarks, separate the timings for operations in an individual stage from benchmarks that time interactions in several stages. For example, to benchmark the time polygons will spend in the geometry subsystem, make sure that the polygons are not actually being limited by the raster subsystem. One simple trick for this is to draw the polygons as 1-pixel polygons. Another might be to enable some mode that will cause a very fast rejection of polygon or pixels after the geometry subsystem.

However, it is important to write both benchmarks that time individual operations in each stage, and those that mimic interactions that you expect to happen in your application.

Finding the Bottlenecks

Over the course of drawing a frame, there will likely be many different bottlenecks. If you first clear the screen and draw background polygons, you will start out fill-limited. Then, as other drawing happens, the bottleneck will move up and down the pipeline (hopefully not residing at the host). Without special tools, bottlenecks can be found only by creative experimentation. The basic strategy is to isolate the most overwhelming bottleneck for a frame and then try to minimize it without creating a worse one elsewhere.

Isolate the bottleneck stage of the pipeline.

One way to isolate bottlenecks is by eliminating work at specific stages of the pipeline and then check to see if there is a significant improvement in performance. To test for a geometry subsystem bottleneck, you might force off lighting calculations, or normalization of vertex normals. To test for a fill bottleneck, disable complex fill modes (z-buffering, gouraud shading, texturing), or simply shrink the window size. However, beware of secondary affects that can confuse the results. For example, if the application adjusts what it draws based on the smaller window, the results from just shrinking the window without disabling that functionality will be meaningless. Some stages are simply very hard to isolate. One such example is the clipping stage. However, if the application is culling the database to the frustum, you can test for an extreme clipping bottleneck by simply pushing out the viewing frustum to include all of the geometry.

6 Tuning the Application

Applications usually plan on pushing graphics past their limits. If the rendering traversal is part of the application, then this traversal must be optimized so that it keeps the graphics subsystem busy. On a multiprocessing system, other operations for scene management formatting of data can be moved out of the renderer and into other processes, preferably running on other CPUs. Finally, a key part of real-time rendering is *load management*, providing a graceful response to overloading the graphics subsystem, which is discussed later in Section 8.

Tuning the Renderer

Efficient Coding

There is no escape from writing efficient code in the renderer. Immediate mode drawing loops are the most important parts since code in those loops are executed thousands of times per frame. For peak performance from these loops, one should do the following:

- Minimize the loop overhead and decision logic in the polygon loops. Unroll per-vertex code and duplicate code, as opposed to using per-vertex if-tests.
- Use a flat data structure for the draw traversal - you want to minimize the number of memory pages you need to touch in a given loop.
- Disassemble code to examine loop overhead (and to check that the compiler is doing what you expect).

Display list rendering requires less optimization because it does not require the tight loops for rendering individual polygons. However, this is at the cost of more memory usage for storing the display list and less flexibility in being able to edit the geometry in the list for dynamic objects. The extra memory required by display lists can be quite significant because there can be no vertex sharing in display lists. This can restrict the number of objects you can hold in memory and will also slow the time to page in new objects if the graphics display lists must be re-created. Additionally, display lists may need to be of a certain minimum size to be handled efficiently by the system. If there are many small moving objects in the scene, the result will be many small display lists. If you have the choice, given the option between immediate mode rendering and database paging, you might choose to use at least some immediate mode, particularly for dynamic objects.

Don't let the host be the bottleneck

IRIS Performer™, a Silicon Graphics toolkit for developing real-time graphics applications, uses a fairly aggressive technique for achieving high-performance immediate-mode rendering. Data structures for geometry enforce the use of efficient drawing primitives. Geometry is grouped into sets by type and attribute bindings (use of per-vertex or per-polygon colors, normals, and texture coordinates). For each combination of primitive and attribute binding, there is a specialized routine with a tight loop to draw the geometry in that set. The result is several hundred such routines but the use of macros makes the code easy to generate and maintain. IRIS Performer also provides an optimized display list mode that is actually an immediate mode display list and shares the application copy of data instead of copying off a separate, uneditable copy. This is discussed in [Rohlf94], and [PFPG94]. Host rendering optimization techniques 24are also discussed in detail in [GLPTT92].

Multiprocessing

Multiprocessing can be used to allow the renderer to devote its time issuing graphics calls while other tasks, such as scene and load management can be placed into other processes. There are several large tasks that are obvious candidates for such course-grained multiprocessing:

- the real-time application — processes inputs from IO, calculates new viewing parameters, positional parameters for objects, and parameters for dynamic geometry,
- scene management — culling out parts of the scene graph that are not in the viewing frustum, calculating LOD information, generating a display list for the rendering traversal,
- dynamic editing of geometric data,
- IO handling — polling external devices, database paging,
- intersection traversals for collision detection,
- complex simulations for various vehicles.

A combination of pipelining and parallelism can be used to get the right throughput/latency trade-off for your application and the target machine. IRIS Performer™ provides a process pipeline:

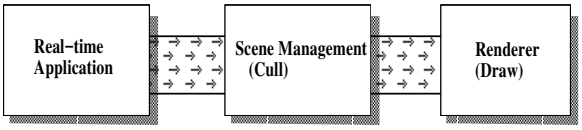


FIGURE 12. IRIS Performer Process Pipeline

This process pipeline, described in [Rohlf94], is re-configurable to allow:

- a pipeline where the cull and draw are parallel processes (app->cull/draw),
- a model where the cull and draw are performed by a single process that culls and renders simultaneously (app->cull_draw),
- a minimal-latency model where all tasks are performed by a single process (app_cull_draw).

Multiprocessing also allows additional tasks to be done that will make the rendering task more efficient, such as:

- generating a per-frame, optimized display list for the rendering task so that the drawing traversal does not need to traverse the original database,
- sorting geometry by mode to minimize mode changes,
- host backface removal (and removal of backfaced objects) to save additional host bandwidth,
- flattening of dynamic transformations over objects of only one or two polygons.

It is important to identify which tasks must be real-time, and which can run asynchronously and extend beyond frame boundaries. Real-time tasks are those that must happen within a fixed interval of time, and severe consequences will result if the task extends beyond its frame. The main applica-

tion, cull, and draw tasks are all real-time tasks. However, it might not be so traumatic if, for example, some of the collision results are a frame late. The polling of external control devices should probably be done in a separate, asynchronous process — if those results are late, extrapolation from previous results is probably better than waiting. Real-time tasks are discussed further in Section 8.

A real-time process should not poll an external device.

7 Database Tuning

Tuning databases is as important (and often just as much work) as tuning the application. The database hierarchy needs to be structured to optimize the major traversal tasks. Information cached in the database hierarchy can reduce the number of dynamic operations that must be done by traversals. Finally, the modeling of the database geometry should be done with an understanding of the performance characteristics of the graphics pipeline.

Spatial Hierarchy Balanced with Scene Complexity

The major real-time database traversals are the cull and collision traversals. Both benefit by having a database that is spatially organized, or is coherent in world space. These traversals eliminate parts of the scene graph based on bounding geometry. If a database hierarchy is organized by grouping spatially near objects, then entire sub-trees can be easily eliminated by the bounding geometry of a root node. If most nodes have bounding geometry that covers much of the database, then an excessive amount of the database will have to be traversed.

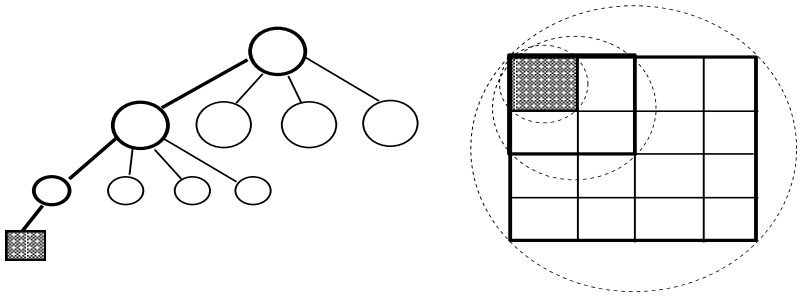


FIGURE 13. Scene-graph with Spatial Hierarchy

It is additionally helpful to have a hierarchy based on square areas so that simple bounding geometry, such as bounding spheres, can be used to optimize the traversal test.

The amount of hierarchy put in the database should balance the traversal cost of nodes with the number of children under them. A node with few children will be able to eliminate much of the database in one step. However, a deep hierarchy might be expensive to maintain as objects change and information must be propagated up the tree.

Database Instancing

Instancing in a database is where multiple parents reference a single instanced child which allows you to make performance/memory trade-offs.

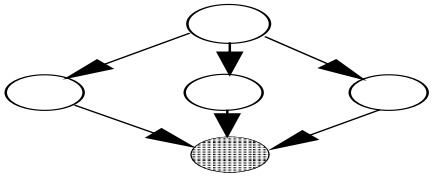


FIGURE 14. Instanced Node

Instancing saves memory but prevents a traversal from caching traversal information in the child and also prevents you from flattening inherited matrix transformations. To avoid these problems, IRIS Performer™ provides a compromise of *cloning* where nodes are copied but actual geometry is shared.

Balancing the Traversals

The amount of geometry stored under a leaf node will affect all of the traversals, but there is a performance trade-off between the spatial traversals and the drawing task. Leaf nodes with small numbers of polygons will provide a much more accurate culling of objects to the viewing frustum, thus generating fewer objects that must be drawn. This will make less work for the rendering task; however, the culling process will have to do more work per polygon to evaluate bounding geometry. If the collision traversal needs to compute intersections with actual geometry, then a similar trade-off exists: fewer polygons under a leaf node means fewer expensive polygon intersections to compute.

Modeling to the Graphics Pipeline

The modeling of the database will directly affect the rendering performance of the resulting application and so needs to match the performance

characteristics of the graphics pipeline and make trade-offs with the database traversals. Graphics pipelines that support connected primitives, such as triangle meshes, will benefit from having long meshes in the database. However, the length of the meshes will affect the resulting database hierarchy and long strips through the database will not cull well with simple bounding geometry.

Objects can be modeled with an understanding of inherent bottlenecks in the graphics pipelines. Pipelines that are severely fill-limited will benefit from having objects modeled with cut polygons and more vertices and fewer overlapping parts which will decrease depth complexity.

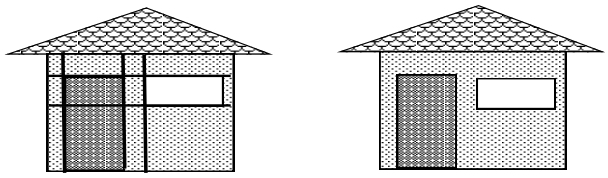


FIGURE 15. Modeling with cut polygons vs. overlapping polygons

Pipelines that are easily geometry or host limited will benefit from modeling with fewer polygons.

There are a couple of other modeling tricks that can reduce database complexity. One is to use textured polygons to simulate complex geometry. It is especially useful if the graphics subsystem supports the use of *alpha textures* where a channel of the texture marks the transparency of the object. Texture can be made as cut-outs for things like fences and trees. Textures are also useful for simulating particles, such as smoke. Textured polygons as single-polygon billboards are additionally useful. *Billboards* are polygons that are fixed at a point and rotated about an axis, or about a point, so that the polygon always faces the viewer. Billboards are useful for symmetric objects just as light posts and trees, and also for volume objects such as smoke. Billboards can also be used for distant objects to save geometry. However, the managing of billboard transformations can be expensive and impact both of the cull and draw processes.

3D Database modeling techniques like these have been in use for a long time in Visual Simulation applications.

8 Real-Time On a Workstation

Graphics workstations are attractive development platforms because they have rich and user-friendly development environments. They are not tradi-

tionally known for their real-time environments. However, extensions can be made to the basic operating system to support a real-time mode, as was done with the Silicon Graphics REACT™ extensions to IRIX™. The REACT extensions are really ways to push UNIX aside for real-time operation. It guarantees interrupt response time on a properly configured system and enables the user to have control of the scheduling of specified processors and therefore be exempt from all UNIX overhead. REACT also includes a real-time frame scheduler that can be used on the real-time processors. Finally, REACT offers time-stamps on system and user operations for real-time system performance feedback.

Running an application in performance-mode might be quite different from running it in development mode. Most obviously, a real-time application needs fast timers to be able to monitor its performance for load-management, as well as having accurate time-based animations and events. A real-time application also needs to be guaranteed worst case behavior for basic system functions such as interrupt response time. It also needs to have control over how it is scheduled with other processes on the system, and how its memory is managed. In addition, the main application needs to synchronize frame boundaries of various tasks with the graphics subsystem.

Put the system and application in real-time mode for real-time performance.

Managing System Resources for Real-Time

One type of organization is to put the rendering process on its own processor, isolated from other system activity and synchronization with other tasks. This is the organization used in IRIS Performer™[Rolf94]. To do this, the rendering process should also have its own copy of data to minimize synchronization and conflict over pages with other processors. On a general-purpose workstation, one CPU will need to be running basic system tasks and the scheduler. Additionally, a distinction should be made between tasks that must be real-time (happen reliably at fixed intervals), and those processes that may extend past frame boundaries in generating new results. Non-real-time tasks can be given lower priorities and share processors, perhaps even the system CPU.

Real-Time Graphics

Getting steady, real-time frame rates from the graphics subsystem can be a challenge on any system. One problem is handling overload conditions in the graphics subsystem. Another is the synchronization of multiple graphics systems.

Load Management

High-end image generators have frame control built into the graphics subsystem so that they can simply halting drawing at the end of a frame time. This can produce an unattractive result, but perhaps one that is less disturbing than a wild frame rate. Getting a graphics subsystem to stop based on a command sent from the host and placed in a long FIFO can be a problem. If the graphics subsystem does not have its own mechanism for managing frame rate control (as currently only high-end image-generators do) then the host will have to do it. This means leaving generous margins of safety to account for dynamic changes in graphics load and tuning the database to put an upper bound on worst-case scenes. However, some method of load management will also be required.

Load management for graphics is a nice way of saying “draw less.” One convenient way to do this is by applying a scaling factor to the *levels of detail* (LODs) of the database, using lower LODs when the system is overloaded and higher LODs when it is under-utilized. A hysteresis band can be applied to avoid thrashing between high and low levels of detail. This is the mechanism used by IRIS Performer™[Rohlf94]. This technique alone is quite effective at reducing load in the geometry subsystem because object levels of detail are usually modeled with fewer vertices and polygons. The raster subsystem will see some load reduction if lower levels of detail use simpler fill algorithms, however, they will probably still require writing the same number of pixels. If the system supports it, variable screen resolution is one way to address fill limitation — though this is traditionally only available on high-end image generators. Another trick is to aggressively scale down LODs as a function of distance so that distant objects are not drawn. A fog band makes this less noticeable. However, since they will be small, they may not account for very many pixels. LOD management based on performance prediction of objects in various stage of the graphics pipeline[Funk93] can aid in choosing appropriate levels of detail. Since the computation for load management might be somewhat expensive (calculating distances, averages over previous frames, etc.) it is best done in some process other than the rendering process.

Multiple Machines

Entertainment applications typically have multiple viewpoints that must be rendered and may require multiple graphics systems. If it is desired that these channels display synchronously, then the graphics output must be synchronized, as well as the host applications driving them. There is typically some mechanism to synchronize multiple video signals. However, double-buffered machines must swap buffers during the same video refresh period. This can be done reasonably well from the front end via a high-speed network such as SCRAMnet, as was done in the CAVE environment[CN93], or with special external signals, as is done on the RealityEngine™.

9 Tuning Tools

The proper tuning tools are necessary to simply evaluate an application's performance, let alone tune it. This section describes both diagnostics that you might build into your application, and some tools that the author has found to be useful in the past.

Graphics Tuning Tools

A couple of standard tools for debugging and tuning graphics applications found to be useful on Silicon Graphics machines are **`GLdebug`** and **`GLprof`**, described in detail in [GLPTT92]. **`GLdebug`** is a tracing tool that allows you to trace the graphics calls of an application. This is quite useful because most performance bugs, such as sending down redundant normals or drawing things twice, have no obvious visual cue. The tool can also generate C code that can be used (with some massaging) to write a benchmark for the scene. **`GLprof`** is a graphics execution proffer that collects statistics for a scene and can also simulate the graphics pipeline and display pipeline bottlenecks (host, transform, geometry, scan-conversion, and fill) over the course of a frame. The **`GLprof`** statistics include counts for triangles in different modes, mode changes, matrix transformations, and also the number of polygons of different sizes in different fill modes.

System Tuning Tools

Some of the tools in the standard UNIX environment are also very useful. **`prof`**, a general profiler which does run-time sampling of program execution, allows you to find *hot spots* of execution.

Silicon Graphics provides some additional tools to help with system and real-time tuning. **`pixie`** is an extension to **`prof`** and does basic block counting and supports simulation of different target CPUs. **`par`** is a useful system tool that allows you to trace system and scheduling activity. Silicon Graphics machines also have a general system monitoring tool, **`osview`**, that allows you to externally monitor detailed system activity, including CPU load, CPU time spent in user code, interrupts, and the OS, virtual memory operations, graphics system operations, system calls, network activity, and more.

For more detailed performance monitoring of individual applications, Silicon Graphics provides a product called WorkShop that is part of the CASEVision™ tools which is a full environment for sophisticated multiprocess debugging or tuning[CASE94]. For monitoring of real-time performance of multiprocessed applications, there is the WindView™ for IRIX product based on the WindView™ product from WindRiver. WindView

works with IRIX REACT to monitor use of synchronization primitives, context switching, waiting on system resources, and tracks user-defined events with time-stamps. The results are displayed in a clear graphical form. Additionally, there is the Performance Co-Pilot™ product from Silicon Graphics that can be used for full-system real-time performance analysis and tuning.

Real-Time Diagnostics

The most valuable tools may be the ones you write yourself as it is terribly difficult for outside tools to non-invasively evaluate a real-time application. Real-time diagnostics built into the application are useful for debugging, tuning, and even load-management. There are four main types of statistics: system statistics, process timing statistics, statistics on traversal operations, and statistics on frame geometry.

Applications should be self-profiling in real-time.

System statistics include host processor load, graphics utilization, time spent in system code, virtual memory operations, etc. The operating system should allow you to enable monitoring and periodic querying of these types of statistics.

Process time-stamps are taken by the processes themselves at the start and end of important operations. It is tremendously useful to keep time-stamps over several frames and then display the results as timing bars relative to frame boundaries. This allows one to monitor the timing behavior of different processes in real-time as the system runs. By examining the timing history, one can keep track of the average time each task takes for a frame, and can also detect if any task ever extends past a frame boundary. The standard deviation of task times will show the stability of the system. Process timing statistics from IRIS Performer™ are shown in Figure 16. Geometry statistics can keep track of the number of polygons in a frame, the ratio of polygons to leaf nodes in the database, frequency of mode changes, and

average triangle mesh lengths. IRIS Performer™ displays a histogram of tmesh lengths, shown in the statistics above in Figure 16.

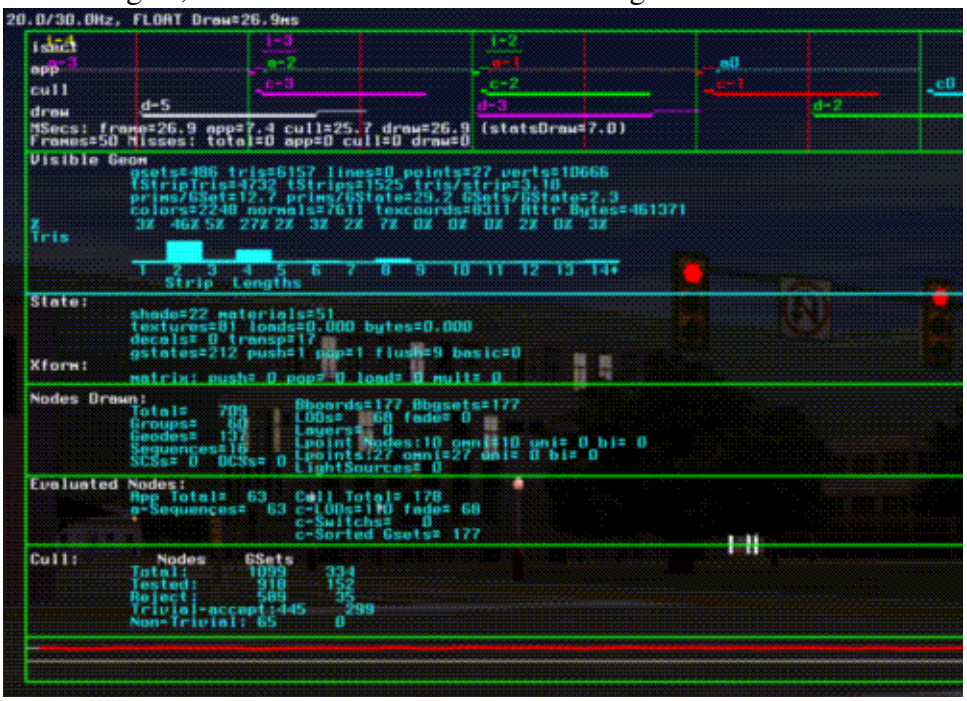


FIGURE 16. Process and Database Statistics

Traversal and geometry statistics do not need to be real-time, and may actually slow traversal operations. Therefore, they should only be enabled selectively while tuning the traversals and database. Traversal statistics can keep track of the number of different types of nodes traversed, the number of different types of operations performed, and perhaps statistics on their results. The culling traversal should keep track of the number of nodes traversed vs. the number that are trivially rejected as being completely outside the viewing frustum. A high number of trivial rejections means that the database is not spatially well organized because the travsersal should not have to examine many of those nodes.

Additionally, IRIS Performer™ supports the display of depth complexity, where the scene is painted according to how many times pixels are touched.The painted framebuffer is then read back to the host for analysis of depth complexity. This display is comfortably interactive on a VGX™ or RealityEngine™ due to special hardware support for logical operations and

stenciling. Thus, you can actually drive through your database and examine depth complexity in real-time.

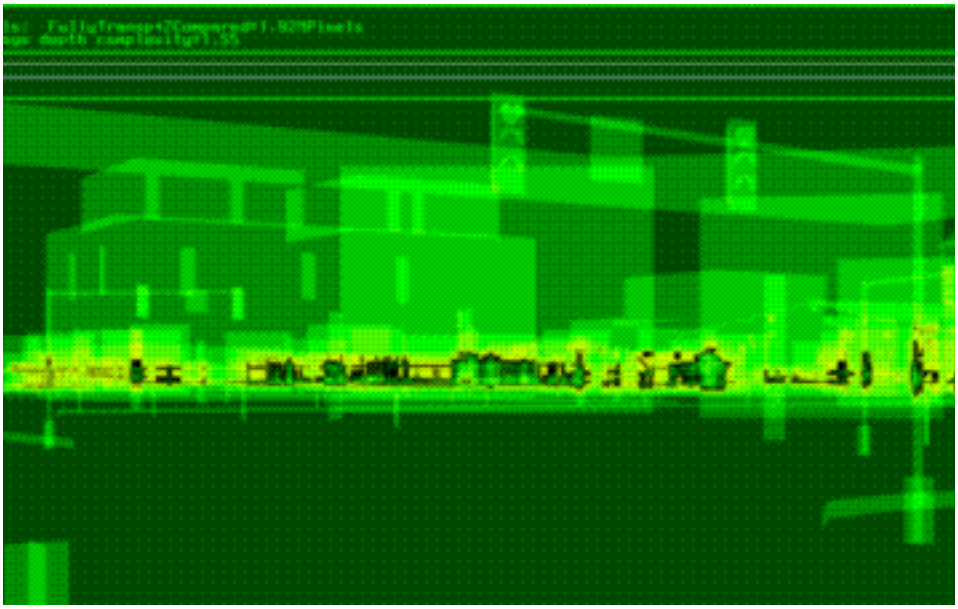


FIGURE 17. Pixel Depth Complexity Profile

10 Conclusion

Graphics workstations offer a wide array of options for balancing cost, performance, and image quality. Rich development environments plus real-time image-generator features like texture mapping and full scene anti-aliasing make graphics workstations attractive platforms for both the development and deployment of entertainment applications. Understanding the target graphics architecture and designing for performance will enable you to get the scene quality and performance you paid for.

11 Acknowledgments

The helpful comments and careful review by Rosemary Chang and Kevin Hunter, both of Silicon Graphics, greatly contributed to the quality of this paper. Their efforts and time are very much appreciated.

12 References

[Akeley89] Kurt Akeley, “The Silicon Graphics 4D/240GTX Superworkstation”, *CG&A*, Vol. 9, No. 4, July 89, pp. 71-83.

[Akeley93] Kurt Akeley, “RealityEngine Graphics”, *Proceedings of SIGGRAPH ‘93*, July 93, pp. 109-116.

[CASE94] *CASEVision™/WorkShop User’s Guide*, Document #007-1523-040, Silicon Graphics., 1994.

[CN93] C. Cruz-Neira, D. J. Sandin, and T.a A. DeFanti. “Surround-screen projection -based virtual reality: The design and implementation of the cave,” *Proceedings of SIGGRAPH ‘93*, July 93, pp. 135-142.

[Deering93] Michael F. Deering, “Leo: A System for Cost Effective 3D Shaded Graphics”, *Proceedings of SIGGRAPH ‘93*, July 93, pp. 101-108.

[Foley90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics Principles and Practice, 2nd. Ed.*, Addison-Wesley, 1990.

[Fuchs89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Israel, “Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories”, *Proceedings of SIGGRAPH ‘89, July 89*, pp. 79-88.

[Funk93] Tom Funkhouser and Carlo Sequin, “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments”, *Proceedings of SIGGRAPH ‘93*, July 93, pp. 247-254.

[GLPTT92] *Graphics Library Programming Tools and Techniques*, Document #007-1489-010, Silicon Graphics., 1992.

[Haeberli90] Paul Haeberli, Kurt Akeley, “The Accumulation Buffer: Hardware Support for High Quality Rendering”, *Proceedings of SIGGRAPH ‘90, Aug. ‘90,* pp. 309-318.

[Harrell93] Chandlee B. Harrell and Farhad Fouladi, “Graphics Rendering Architecture for a High Performance Desktop Workstation”, *Proceedings of SIGGRAPH ‘93*, July 93, pp. 93-99

[Molnar92] Steven Molnar, John Eyles, and John Poulton, “PixelFlow: High-Speed Rendering Using Image Composition”, *Proceedings of SIGGRAPH ‘92, July 92*, pp. 231-240.

[OpenGL93] Jackie Neider, Tom Davis, Mason Woo, OpenGL™, Programming Guide, Addison Wesley., 1993.

[PFPG94] *IRIS Performer™ Programming Guide,* Document #007-1680-020, IRIS Performer™, Silicon Graphics., 1994.

[Potmesil89] Michael Potmesil and Eric M. Hoffert, “The Pixel Machine: A Parallel Image Computer”, Proceedings of SIGGRAPH ‘89, July 89, pp. 69-70.

[Rohlf94] John Rohlf and James Helman, “IRIS Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics”, Proceedings of SIGGRAPH ‘94, July 94, pp. 381-394.

[Regan94] Mathew Regan, Ronald Pose, “Priority Rendering with a Virtual Reality Address Recalculation Pipeline.”, Proceedings of SIGGRAPH ‘94, July 94, pp. 155-162.

[Voorhie89] Doug Voorhies, “Reduced-complexity Graphics”, *CG&A*, Vol. 9, No. 4, July 89, pp. 63-70.

Exploiting Consumer Class 3D Hardware Acceleration for Real-Time Entertainment

Andy Bigos
3Dlabs

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

Abstract

This paper describes the new breed of consumer class 3D graphics accelerators now becoming available on the PC platform to accelerate mainstream, or pervasive 3D applications from games to web browsers. Aimed at game developers, this paper highlights the typical feature sets and performance characteristics of these new accelerators and discusses some of the issues involved in fully exploiting them. This paper will also be of interest to anyone wanting to come up to speed on this new class of accelerator.

1 Introduction[i]

This year, 1996, sees the arrival of a new class of PC video accelerator; the consumer class 3D and multimedia accelerator. While 3D acceleration has been available for some time on PC's, it has generally been targeted at the professional CAD and simulation markets.

The latest generation of low cost 3D accelerators are being aimed at the entertainment and pervasive multimedia markets on PC's and are being enabled by strong 3D API support under Windows95 from Direct3D[DirectX] and OpenGL[OGL1/2]. One of the significant motivations for this market has been

i. All trademarks acknowledged.

to enable ‘console class’ games on the PC. The latest game consoles, e.g. the Sony PlayStation and Nintendo64 all have dedicated 3D acceleration hardware, and with the advent of PC based 3D accelerators the PC is now able to deliver the performance necessary performance to create similarly compelling 3D experiences.

This tutorial attempts to describe, in some detail, the architecture, features and performance of a typical system and some of the keys to fully exploiting it. In this discussion I’m remaining API neutral and concentrating on generic 3D accelerator features and system functionality, parts of which are not always supported by all API’s. Taking this approach can give a better insight into the technology, leading to better use of the higher level API’s.

After reading this paper you should have a clear understanding of what consumer class 3D hardware acceleration on the PC is capable of, and have a feel for the trade-offs involved in fully exploiting the features and performance of these systems.

2 Hardware: Overview

The Geometry Pipe

At the risk of treading well worn ground, the standard 3D graphics geometry pipe [Foley90], consists of the following stages:

- Database Traversal - Deciding what objects to display in the current scene.
- Modelling Transforms - Placing the objects in the scene.
- Lighting - Lighting and coloring objects appropriate to the environment.
- View Transforms - Placing the virtual camera in the scene.
- View Volume Clipping - Restricting the primitives to the current view.
- Homogeneous Division - Mapping the scene onto the 2D viewing plane.
- Primitive Assembly - Constructing the rendered primitives.
- Setup Calculations - Evaluation of the parameters required for rasterization.
- Rasterization (Scan Conversion) - Painting the pixels on the screen.

Consumer level PC 3D accelerators target the lower end of the pipeline, as this tends to be where the main system bottlenecks occur:

- Primitive Assembly - This is the process of constructing primitives from a stream of vertices processed by the higher levels of the geometry pipe. Full support for primitive assembly is essential in hardware accelerators which perform setup calculations because it allows advantage to be taken of meshed primitives such as triangle strips and fans which share vertex data. Shared vertex data in meshed primitives is an effective way to reduce the amount of data required to draw primitives, and thus reduce bus traffic - which, as shown later, can be a bottleneck in some rendering situations. Backface culling and screen clipping can also be implemented at this stage where necessary.
- Setup Calculation - This stage calculates the parameters required by the rasterizer to interpolate color and texture values over a primitive. As well as being a computationally intensive task it handles conversions from floating point to fixed point representation.
- Rasterization - This is the stage at which pixels are generated, and effects such as shading, texturing and fogging are applied.

Alternative Architectures

A number of different approaches to providing cost effective 3D on the PC are possible. This section identifies the three main areas of technology currently seen.

Cost Engineered Workstation Technology

This class of system has its heritage in workstation 3D graphics. It is typified by having a rich and mature 3D feature set, broad API support and fully integrated GUI and video playback capabilities, it is typified by 3Dlabs' PERMEDIA [PERMEDIA] and GLINT Delta[Delta] class accelerators.

Novel New Techniques

These systems use novel methods to reduce the cost of providing 3D or add extra functionality and quality. Whilst offering impressive proprietary 3D solutions they tend to have eclectic feature sets and often limited support for standard API's and 2D.

2D GUI Transitional (FreeD)

These systems offer limited 3D functionality bolted-on to an existing 2D chip. These often suffer from patchy 3D API support and fail to perform better then fast software 3D solutions. They do however deliver good 2D GUI performance.

The rest of this document describes the features and performance of consumer class accelerators based on the first group identified above, those based around

architectures found in the workstation world. Because these systems have broad API support, are based on sound architectures, and have a well known pedigree, they are likely to become the most widely adopted.

3 Hardware: Performance

This section is intended to give a feel for the performance levels achievable from consumer class PC acceleration and the performance characteristics of these systems. First common terms used in real-time graphics to measure the complexity of a scene are described, along with the key performance figures quoted by hardware vendors. Finally the performance characteristics of these systems are discussed.

To give this section a grounding in reality, performance figures quoted are from the 3Dlabs PERMEDIA chip[i].

Performance: Metrics

Three standard measures of graphical complexity are encountered in real-time programming:

- Scene Complexity, which refers to the number of polygons wholly or partially visible in the viewing volume per frame. This number is typically significantly lower than the total number of polygons in the database.
- Depth Complexity, referring to the number times a pixel is drawn per frame. This is very dependent on the type of game and figures can range from less than 1.0 to greater than 5.0.
- Pixel Complexity, a measure of the number of graphical effects applied to a pixel. Unlike depth complexity and scene complexity this is a qualitative measure. A pixel with high complexity would be one with depth testing, blending, texturing and fogging applied, whereas a flat shaded non depth buffered pixel would have a low pixel complexity.

These metrics allow meaningful estimates of system performance to be made, as well as comparisons between different systems to be performed reliably.

i. PERMEDIA performance figures are based on chip. While every care has been taken in the preparation of performance figures in this document, 3Dlabs accepts no liability for any consequences of their use. 3Dlabs products are under continual improvement and 3Dlabs reserve the right to change specifications without notice. Contact 3Dlabs for the most current information.

Performance: Figures

Performance figures fall into two areas; fill rates and primitive throughput. Primitive throughput limits the number of primitives the system can handle in unit time, and is typically measured in thousands of triangles per second ktps. Fill rate describes the maximum rate at which pixels can be drawn with various pixel complexities, typically measured in millions of pixels per second, commonly referred to simply as MPixels.

Primitive Throughput

Primitive throughput is sensitive to both pixel complexity and host performance.

The following figures assume an infinitely powerful host to perform the setup calculations.

Primitive Type[a]	25 Pixel Triangles (ktps)	50 Pixel (ktps)
Gouraud	600	350
Gouraud, Depth	350	250
Gouraud, Texture	500	300
Gouraud, Texture, Depth	300	200

a. Screen resolution 640x480, 16bpp, 16bit depth.

TABLE 1. PERMEDIA Primitive Throughput Rates 25 and 50 Pixel Triangles

These figures can give a good idea of the peak performance levels achievable, they are however not a sound basis to make good estimates of game performance. Game level benchmarks, as described on page 3-7, give a much better feel for realistic performance by taking into account possible asynchronous processing between the host and accelerator as well as other host loading, including geometry, lighting and game mechanics.

Primitive throughput is sensitive to host performance so running benchmarks on a range of hosts systems can highlight areas which may later become bottlenecks.

Pixel Fill Rates

Pixel fill rates are typically very sensitive to pixel complexity but relatively insensitive to host performance. Fill rates of interest are:

- Flat shaded, non depth buffered. This is the lowest pixel complexity and is of interest to see the fastest fill rate an accelerator can achieve. In systems with VRAM, SGRAM etc., this figure can be very high when block filling is used. In some systems additional per-pixel effects can be applied while maintaining very high fill rates, for example stipple testing. This

type of information is critical to getting the most out of an accelerator, and can be sometimes be gleaned from documentation, sometimes from benchmarking.

- Gouraud shaded, non depth buffered. This should be tested in Ramp and RGB color modes which can give an interesting insight into the architecture of the chip.
- Textured, non depth buffered. This figure is sensitive to various texture options enabled, as well as the properties of the actual mapping (i.e. if the texture is being minified or magnified, if perspective correction is enabled, etc.). It is sometimes quoted in MTexels rather than MPixels to emphasise that it's a texture mapping rate. For games which do not use depth buffering this is the key fill rate figure.
- Textured, depth buffered (pass and fail). Two figures are of interest in the case of depth buffered rendering, the rate at which pixels are plotted when the depth test passes (entailing a read and write of the depth buffer) and secondly, the rate at which pixels can be rejected when failing the depth test. Often a system is designed to perform the depth test before other potentially expensive operations such as texture mapping, so the depth fail performance can be significantly higher than the depth pass performance[i]. If this is the case the game can be structured to take advantage of it as described in "Early Depth Testing" on page 3-21

Pixel Complexity[a]	Pixel Fill Rate (MPixels)	Notes
Flat Shaded, No Depth	1600	Block Fill
Flat Shaded, No Depth	50	No Block Fill
Gouraud Shaded, No Depth	25	Single Interpolant (8bpp)
Gouraud Shaded, No Depth	25	RGB Interpolants
Texture, No Depth	25	Fastest Texture Mode
Texture, Depth (Win)	16.6	Depth Read and Write
Texture, Depth (Lose)	25	Depth Read

a. Screen resolution 640x480, 16bpp, 16bit depth.

TABLE 2. PERMEDIA Fill Rate Figures

Using these figures it's possible to make ballpark estimates of performance levels achievable on an accelerator. The following table uses these figures to

i. Some systems can perform deferred shading, that is, only doing full pixel processing for pixels which are known to be visible. This is the equivalent of depth buffered rendering in strictly front to back order, as described in "Depth Buffered Hidden Surface Removal" on page 3-21.

show frame rates for a pixel fill rate of 25MPixels for a number of screen resolutions and depth complexities.

Screen Resolution	Depth Complexity	Frame Rate[a] (Hz)
320x240	1.5	217
320s240	2.5	130
512s384	1.5	84
512s384	2.5	50
640x400	1.5	65
640x400	2.5	40
640x480	1.5	54
640x480	2.5	32

a. Assuming zero cost double buffering.

TABLE 3. Frame Rate and Depth Complexity for 25MPixel Fill Rate

This is a relatively crude estimate, ignoring everything except fill rate, however these figures can be useful at the feasibility stage of a project to identify target screen resolutions.

Game Level Benchmarks

In an ideal world turning on extra features on an accelerator would have no impact on performance - however this is rarely the case in practice! Trade-offs between cost and performance must be made during the implementation of any feature, for example many effects can be implemented in a serial or parallel manner, serial often being cost effective, parallel providing highest performance. This type of trade-off can lead to unexpected performance hiccups when some combinations of features are used, making performance predicting difficult.

As soon as possible in a project's development game level benchmarks should be used to measure performance. These should include realistic databases and non graphics related CPU loading, game mechanics, sound, user input etc., and, critically, be repeatable. At this stage benchmark figures can become game specific and qualitative, measured in monsters per second or simply "Wow this plays well". As soon as this type of benchmark is available it allows database designers and modellers to get real feedback and start to tune the system. It also allows potential system bottlenecks to be identified early.

Performance: Hardware vs. Software

Although well designed 3D hardware typically gives significant improvements in graphics performance over software, in some cases the improvements can be much less and occasionally fast software can go faster than hardware.

There is an overhead involved in passing data to the accelerator and performance for small primitives can be very sensitive to the amount of data needed by the hardware to describe the primitive. If, for example, a primitive only covers a few pixels, and a dozen parameters must be passed to the hardware to describe the primitive, then it can be slower to pass the parameters to the hardware than for a software rasterizer to render them. If this is sustained for a large number of primitives in a scene then the hardware is being poorly utilised.

Figure 1 (below) shows performance curves of Pixels per Triangle vs. Fill Rate for software, and hardware (with and without setup processing).

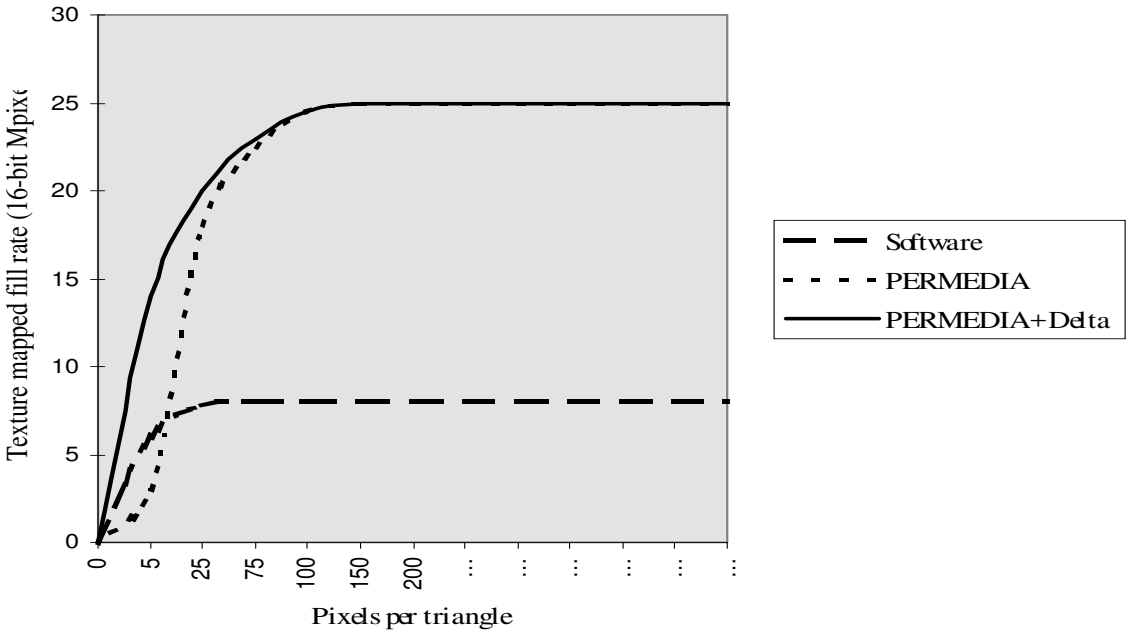


FIGURE 1. Hardware vs. Software Fill Rates

This curve identifies the area in which software can render pixels faster than hardware (without triangle setup) because less pixels are being plotted than data being transferred to the accelerator to describe the primitive. Hardware with setup processing requires less data to describe a primitive, and removes a significant amount of processing (slope calculations and floating to fixed conversion) and will usually beat software in all cases.

Figure 2 shows for small triangles the primitive throughput is sensitive to the amount of data transferred to the accelerator, even though the fill rates are identical.

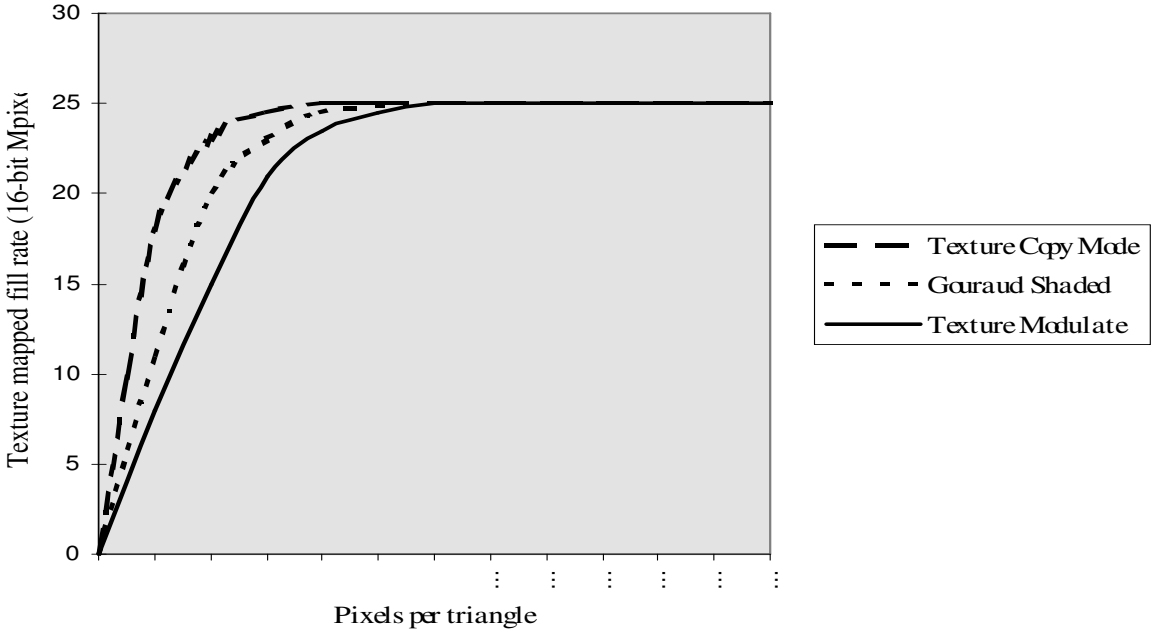


FIGURE 2. Fill Rate And Pixel Complexity

The slope of the curve is sensitive to the amount of data needed to draw the primitive:

- Texture (Copy Mode) requires two interpolants[i] for the texture coordinates (or three for perspective correct).
- Gouraud (RGB) requires three interpolants, one for each color component.
- Texture (Modulate) requires five interpolants, two texture coordinates, three color components

This graph gives a feel for the characteristic primitive throughput of a typical accelerator. As the triangle size gets larger the fill rate approaches the maximum. For small primitives however the gradient of the curve is high and performance varies greatly for small changes in polygon size and pixel complexity.

---

i. Interpolants are the values interpolated over a primitive, and include, color, depth, fog, texture, etc. To interpolate a value over the primitive 3 different values are required per interpolant, delta (or gradient) values in the x direction and along the edge of the primitive, and a start value. This is in addition to the data required to describe the geometry of the primitive itself, typically in the order of 20 words of data are required to render an RGB depth buffered triangle in a system without setup support.

In figures 1 and 2 you should always aim to work in the area of the curves were the highest fill rate is achieved. If your database lies on the first part of the curve you are not fully utilizing the hardware, and performance becomes erratic because small changes in primitive size and complexity can result in large changes in performance.

The moral of the story is that you must leverage work onto the accelerator to fully exploit its performance and quality improvements. This means that you should be biased towards using large polygons (trying to sustain peak fill rates) and adding complexity to a scenes via per-pixel effects (such as fogging) rather than adding geometry. This is less true for systems which support setup in hardware.

4 Hardware: Features

This section describes some of the features found on a typical accelerator along with some discussion of the performance issues involved in using them.

Color Modes

Ramp Mode (Color Index) Rendering

Ramp mode rendering uses a single color interpolant in conjunction with a fixed palette, typically with 256 entries (8 bit). Ramp mode rendering is computationally efficient but suffers from a number of limitations:

- High quality effects such as filtering and fogging are difficult to perform with a fixed palette. Great care has to be taken in constructing palettes to allow these effects to work properly.
- The limited number of palette entries can soon become exhausted when many textures are required in conjunction with lighting, fogging, etc.

One advantage of ramp mode rendering over RGB rendering is that because the palette is constructed by hand, it can be optimised to suit the game. Converting an optimised 256 entry palette into equivalent 8 bit RGB colors generally looses color fidelity. Typically a 16 bit RGB mode must be used to maintain color fidelity - thus doubling the required frame buffer memory.

Note also, that Direct3D supports accelerated rendering in RGB color space only - ramp mode rendering is performed in software.

RGB Mode Rendering

Support for RGB is one of the big advances in quality enabled by hardware accelerated 3D. Rendering in RGB color space makes performing operations

on color values a well defined and simple operation - at the cost of additional computational load (three color components instead of one). Most accelerators will handle the three color components in parallel at no per-pixel cost - this should be verified through benchmarking though.

RGB color depths supported are typically 8bit (3bits red, 3bits green and 2bits blue), and 16bit (5bits red, 6bits green and 5bits blue). Dithering is essential for RGB rendering at low color resolutions and should be available at no cost.

Performance Considerations: The extra color interpolants required for RGB rendering can cause extra host and accelerator loading. For example gouraud shading in RGB mode requires three interpolants, in ramp mode it requires only one. Flat shading requires no color interpolants and can often be performed at very high speed (see Table 2).

Texture Mapping

High performance texture mapping is a key enabler to allow console class games on PC's. The following sections describe the kind of features available and performance considerations to bear in mind when using them.

Texture Formats

Textures can generally be supplied in a variety of formats to allow trade-offs to be made between color fidelity and memory usage. RGB textures should be available in 8, 16 and 24 bit formats.

Palette Textures (Compressed Textures)

Palette textures add a level of indirection into the process of mapping texel values to RGB colors. Rather than a texel representing a color, it represents an index into a palette of colors. Two benefits arise from this indirection. Firstly it achieves a degree of memory compression, and secondly, the indexed colors are typically of high color resolution.

Supported table sizes typically include 2,4,8,16 and 256 entries, 16 entry (4 bit) has been found in practice to offer the best compromise between compression and fidelity for many textures.

Eight bit palette textures provide an excellent mechanism for porting games based on ramp mode rendering to an RGB system. The texture palette is set to the ramp palette and the frame buffer is set to RGB mode. This allows rapid porting of an existing game and allows hardware features such as fogging and filtering to be added trivially. Significantly, it also means that the original artwork can be used

Performance Considerations: Because palette textures have a small memory foot print they can have a very good cacheing behaviour. Check if the texture

palette is stored in memory or locally on the hardware. The latter is the most efficient.

Application Modes

Texture application modes define how texels read from the texture map are transformed into pixel colors (after any palette indirection). Typical modes either simply replace the pixel color with the texel color (copy mode) or combine (modulate) the texel color with the pixel color, which can originate from flat or gouraud shading:

Copy Mode: C(r,g,b)= T(r,g,b)

Modulate Mode: C(r,g,b) = P(r,g,b) * T(r,g,b)

C(r,g,b) is the pixel color after texture application.

T(r,g,b) is the texel color

P(r,g,b) is the pixel color before texture application

Performance Considerations: The primitive throughput and fill rates of these combinations should be measured (in conjunction with any other effects that are required, e.g. fogging). The fill rates can be sensitive to particular combinations of modes, and the primitive rate is sensitive to the number of interpolants required, especially if the texture is being modulated with a base polygon color, see Figure 2. Typically copy mode texturing will, in practice, be faster than modulate because modulate requires additional color information.

Using pre-lit textures can be a simple way to add environmental effects to a scene while using copy mode textures.

Wrapping Modes

The behavior of a texture at it's boundaries is defined by texture wrap modes. Three modes are typically supported; repeat, mirror and clamp. Repeat and mirror modes allow textures to be tiled across large areas, clamp mode, as it's name implies, causes a texture to be clamped on a surface without any repeat. Wrap modes are typically controllable independently in each dimension.

Repeat and mirror modes are effective ways to add texturing to a scene without excessive use of texture memory.

Bi-Linear Filtering

Software rasterizers typically point-sample textures, that is, the nearest texel to the texture sample location is selected for display. Texture filtering attempts to produce a higher quality image by performing a weighted average of four adjacent texels. The additional texture accesses required to perform texture filtering make it difficult to implement efficiently in software. It is however amenable to hardware acceleration.

The visual effect of bi-linear texture filtering is to reduce the pixelated effect visible on higher magnified point sampled textures.

Performance Considerations: The performance of filtering operations can be sensitive to the ratio of texel size to pixel size, i.e., if the texture is minified or magnified. Magnification can make effective use of textures caches whereas minification can reduce cache efficiency. If this is the case then filtering can be enabled/disabled based on the zoom factor.

Mip-Mapping

Mip-mapping performs level of detail (LOD) management of textures based on the ratio of the projected areas of screen pixels and texels. This serves to reduce sampling errors and improve visual quality. Several copies of the texture are stored at different resolutions and in the texel selection process texels are selected from the map which produces the least distortion in the texel/pixel mapping. This map selection can be made on a per primitive or per-pixel basis.

Mip-mapping has a number of different texel selection possibilities:

- Nearest map, Nearest texel; in this mode the mip-map nearest to the required resolution is selected and the nearest texel to the required sample point is selected.
- Nearest map, Linear texel; in this case the nearest map is picked and a bi-linear filtering operation is used to generate a texture color.
- Linear map, Nearest texel; the two nearest mip-maps are selected, texels are selected from each map without filtering are combined in as a linear combination.
- Linear map, Linear texel; the works, linear on and between maps, commonly referred to as tri-linear filtering.

These are roughly in order of increasing cost[i].

Performance Considerations: The additional lower resolution maps required for mip-mapping can use up precious texture memory (33% extra texture memory is required). An extra interpolant, the LOD parameter must also be calculated and interpolated in order to select the appropriate map. Often when texture memory is scarce a reasonable approximation to mip-mapping can be performed in the game domain. This typically uses two textures (a high resolution texture and low resolution, low frequency texture) with map selection

i. Most accelerators use ‘inverse’ texture mapping, that is texture coordinates are interpolated in screen space and the mapping ‘back’ to texture space is performed to locate the appropriate texel in the texture map. Some systems use ‘forward’ mapping, that is, the texture is parsed in texture space and the corresponding pixels in screen space are located. Mip mapping can actually increase the performance of forward texture mapping which can be dependent on the size the texture being rendered.

being performed on the host on a per primitive basis. This technique gives many of the benefits of mip-mapping without requiring explicit mip-mapping support.

Perspective Correction

Perspective correction enables rasterized parameters to be interpolated linearly in screen space (i.e. cheaply) even though the mapping from 3D space to screen space is a non-linear one. Although this distortion is present in all interpolated parameters (e.g. color components, fog, etc.) it is most noticeable on texture mapped primitives which can contain high frequencies.

Perspective correction adds most value for large polygons near the eye which suffer the most distortion, often a small percentage of polygons in a typical scene.

Two techniques commonly used to implement perspective correction are a per pixel correction, or a subdivision technique (which can be performed at the primitive or scan-line level). It is useful to consider the two techniques in more detail (see [Wolberg90] and [Heckbert91] for a thorough discussion).

Rational Interpolation

The main burden of this technique is seen at the per-pixel stage which adds an extra interpolant and a division per pixel. However, additional host loading is also seen because two additional multiplies are required per vertex and one interpolant per primitive.

Primitive Subdivision

An alternative technique to remove the perspective distortion by subdividing primitives until the distortion is reduced to a tolerable level (i.e. a linear approximation is adequate). One advantage of this technique is that it can be implemented on a system which does not support per-pixel perspective correction, or in a system where per-pixel correction is too expensive.

Performance Considerations: Although perspective correct texturing may be available with no impact to fill rate, the per primitive overhead can be significant. The optimum use of perspective correction depends on a specific circumstances, e.g. if you are fill or geometry limited. Generally small primitives are the most inefficient to renderer perspective correct, and gain the least by it, and correction should be disabled for these primitives if possible. Perspective correction is also an appropriate candidate for LOD management.

Local Textures

Local texture storage refers to textures which are stored in the same address space as the frame and depth buffers. This type of storage makes resource allocation straightforward and allows trade-offs between screen, texture and depths resolutions to be made easily. However, it also allows a number of

effects which are difficult to achieve with other memory architectures, including:

- Antialiasing: Full scene antialiasing can be performed by rendering a scene at a high resolution and then using texture filtering to average into a lower resolution display buffer.
- Rendered textures: In a similar vein, textures can be created through rendering operations. For example, a reflection in a mirror can be created by rendering the scene to create the mirror view, then using this image as a texture to apply to the polygons representing the mirror. Although this is possible without support for local textures, the overhead of copying the rendered image into texture memory can make the scheme impractical for real-time use.
- Scaled rendering: By rendering at a lower resolution than the final display and zooming to it via a texture operation, back buffer memory can be saved, at the cost of a more expensive buffer swap operation.

Depth Buffering

Depth buffering is an essential feature required to support pervasive 3D applications and is becoming more important in the game world as the complexity of game environments increases[i]. Direct3D is optimized for depth buffered rendering.

Depth buffering tests a pixels depth value against the current value in the depth buffer. If the tests passes, the value in the depth buffer is updated with the pixels depth value, if the test fails, the pixel is discarded and the depth buffer is left unchanged. The sense of the test is controlled by the depth test function. The depth test function is typically one of the following; Never, Less, Equal, Less or Equal, Greater, Greater or Equal, Not Equal, or Always. Although the number of depth functions may seem excessive, they can be used to great effect, as described in “Depth Buffered Hidden Surface Removal” on page 3-21.

A depth buffer of 16 bits is sufficient for most games. Deeper buffers mean additional input precision is required to fully utilize them and this is generally not justified in the consumer market. Although depth buffering makes hidden surface removal a trivial operation, it comes at the price of extra memory usage, a 16bit depth buffer at a 640x480 resolution takes 0.6MB of memory (or alternatively eighteen 256x256 four bit index textures) - see Table 4 below.

i. Id Software’s Quake uses depth buffering.

The section “Hidden Surface Removal” on page 3-21 discusses some of the issues involved in hidden surface removal and compares depth buffered rendering and depth priority (sorted) rendering.

Performance Considerations: Depth buffering requires an additional interpolant to be calculated and transferred to the accelerator as well as the memory required for the buffer itself. The depth buffer must also be cleared before starting to render a new frame (this can be avoided in some circumstances, see “Buffer Clearing” on page 3-22) and this must be factored into depth complexity calculations. If clearing the depth buffer is required it can often be hidden under other activities such as waiting for frame blank during double buffering, however this may not always be possible. Triple buffering attempts to do useful work during this potentially idle period and is described in “Double and Triple Buffering” on page 3-27.

Special Effects

One key requirement to exploiting rasterization acceleration is to leverage as much work as possible onto the accelerator. This is made practical when high quality per-pixel effects are available in hardware. To this end ‘special effects’ can be the key to fully exploiting hardware acceleration, by allowing scene complexity to be replaced by pixel complexity.

Fogging

Fogging is the process of blending a pixel color with an arbitrary user supplied color to model atmospheric and special effects. In high quality systems the fog will be added in a linear interpolation operation on a per-pixel basis:

$$C(r,g,b) = f*P(r,g,b) + (1-f)\ F(r,g,b)$$

C(r,g,b) is the pixel color after fog application

P(r,g,b) is the pixel color after fog application

F(r,g,b) is the fogging color

f is the fog index

The fog index value is sometimes linked to the depth value, but is most flexibly supplied as an extra interpolant.

An example of the aggressive use of fog would be to clear the background of the scene to the fog color (using a high fill rate mode) and setting the far clipping plane to coincide with the fog color and render a scene with a depth complexity of less than 1.0. Pulling the far clipping plane forward in this scenario, while keeping it coincident with the fogging color, is an effective way to reduce scene complexity by leveraging work onto the hardware.

Fogging can also serve to reduce the ‘popping’ sometimes associated with level of detail management (“Level of Detail Management” on page 3-24.) changes by performing transitions in partially fogged regions.

Performance Considerations: Benchmarks for fogged fill rates are essential to see if fogging can be utilised to full effect, ideally adding fogging to a scene should have no impact on the fill rate.

Transparency

Transparency can be accomplished using a screen-door technique or through a blending operation.

Screen Door Transparency (Stipple Testing)

Screen-door transparency works by rejecting a percentage of the pixels in a primitive. The degree of transparency being the ratio of retained to rejected pixels. The rejection is based on a user supplied pattern (or a fixed internally generated pattern) and is fixed relative to window coordinates. This means that with appropriately designed patterns transparency can be achieved without the need to sort polygons.

Performance Considerations: Because the pixel rejection can take place high in the pixel processing path it can significantly effect fill rates - it can be useful to measure the fill rate when all pixels are being rejected to get an idea of the win. In some systems stipple testing can be combined with very fast flat shaded performance to allow shadows and transparent effects to be achieved at very high speeds.

Blended Transparency

Blended transparency produces higher quality results than screen door transparency but requires the transparent objects to be rendered in back to front order (even when depth buffering is enabled), as well as requiring a read from the frame buffer. It works by combining a new pixel color with the value already in the frame buffer. A typical blend operation is of the form[i]:

C(r,g,b)=P(r,g,b) * P(a) + D(r,g,b) * (1 - P(a))

C(r,g,b) is the pixel color after blending

P(r,g,b) is the pixel color before blending

P(a) is the pixel alpha(transparency) value

D(r,g,b) is the destination pixel color

Performance Considerations: Because blended transparency requires a read from the frame buffer to calculate the new blended color it can have a significant impact on performance. The blend operation uses requires similar arith-

i. OpenGL supports a much larger range of blending functions, however many rely on the frame buffer storing alpha values and this is not commonly the case for low cost solutions.

metic to fogging and the same hardware elements are often used to perform these operations, thus resulting in a performance hit if blending and fogging are enabled simultaneously.

Lighting Effects

Lighting effects can add significant visual cues to a scene, but are often difficult to apply using the standard texture modulation mode. To assist the lighting of textured surfaces some hardware supports additional lighting parameters to be interpolated with a primitive and combined with the pixel color after texture application (but before the application of fog).

The most common effect supported is the application of a white specular highlight to a surface. This is simply an addition operation:

$$C(r,g,b) = P(r,g,b) + S(m)$$

C(r,g,b) is the pixel color before specular lighting, but after texture application

P(r,g,b) is the lighting effect after specular lighting

S(m) is the specular index (white light)

Performance Considerations: For a white light a single interpolant is required whereas colored lights require three interpolants. Texture mapping with a single channel of specular lighting is an effective way to add environmental effects to a scene without the need to interpolate both color and texture channels.

Video Support

Supporting video is essential for general multimedia acceleration, but this can also be used to great effect in games for video textures, etc.

YUV Color Space Conversion

The basic support for video is to treat YUV as a native color format and provide YUV to RGB conversion. If the YUV conversion is fully integrated in the 3D functionality of the chip-set then it can be subject to all the common pixel processing operations, for example video textures can be depth buffered and specularly lit.

Chroma Testing

Chroma testing is a pixel rejection test which works by comparing a pixel color with a reference value, and accepting or rejecting the pixel based on the outcome of the test. The test is performed independently on all color channels (RGB and A) and a tolerance is provided for each[i].

i. Chroma testing using RGBA and tolerance is a superset of the OpenGL Alpha Test.

Chroma testing can be effective for ‘blue screen’ composition of video footage, but as part of the standard pixel processing path can be utilised to great effect for cut-out textures and 3D sprites.

Miscellaneous

The following are miscellaneous features which can be of use.

Sub-pixel Accurate Rendering

Sub-pixel accurate rendering is essential to allow the seamless application of textures to polygon meshes. This process calculates parameter information to a high precision and samples all parameters at the centre of screen pixels. This ensures that no stitch marks are created at the join of polygons and that the shading of primitives is of a consistent high quality.

Performance Considerations: Sub-pixel correction can add an additional load on the host, and is potentially a candidate for quality/speed trade-offs.

2D Clipping

Generally clipping to the 3D view volume is performed in software as part of the geometry pipeline, however 2D clipping is often supported in hardware. Two levels of support are typically available; primitive level and pixel level. Primitive level clipping modifies primitives to avoid off screen pixels being generated. Pixel level clipping allows off screen pixels which have been generated in the rasterizer to be efficiently discarded.

Performance Considerations: For primitives with a large number of pixels off screen it may be quicker to clip yourself than to use a per pixel clipping scheme, for small primitives it may be quicker to let the hardware do the rejection.

Extent Checking

Some accelerators feature the ability, at no cost, to record the rasterized area over an arbitrary number of primitives. This functionality is especially useful if it takes into account depth testing (and any other pixel rejection tests). This can be the basis for implementing depth visibility testing as described in “Depth Visibility Testing” on page 3-22.. In the more simple case it can be used to minimise the area of the screen repaired during clearing and copying (including buffer swapping).

Stencil Testing

Stencil testing allows rendering to be limited to a region defined by the stencil buffer contents, a reference value, and the current stencil test function. This is most commonly used to mask out arbitrary shaped regions during rendering,

typically to protect a non-interactive part of the display, for example, a control panel.

See [OGL1/2] for a thorough discussion of stencil testing along with applications.

Performance Considerations: The stencil buffer is often packed into the same area of memory as the depth buffer, in which case, when used in conjunction with depth buffering, each additional bit of stencil buffer means one less bit of depth. Generally a one bit stencil buffer is the most useful, leaving fifteen bits of depth.

High Resolution Timer

Access to a high resolution, low cost timer, can be useful to allow accurate synchronisation of game events. By synchronising this timer with the vertical refresh period it can be used to aid implementation of a constant frame rate system.

DMA Mastering

DMA mastering enables the graphics system to suck data out of system memory leaving the host to do other work. This is a key technique to enable overlap of work between the host CPU and the graphics system. DMA works best when the system can batch work up into a series of buffers with roughly equal graphics loading and operate a scheme which automatically sends buffers to be processed while freeing exhausted buffers (see below).

Performance Considerations: The memory required for DMA operations is special in that it must be physically contiguous, which 'vanilla' virtual memory may not be. This requirement often means that data must be copied into a DMA buffer before a DMA operation can take place. In some cases the cost of this copy operation can outweigh the benefits of DMA.

Interrupt Services

Interrupt services can be used to increase the overlap of work between the host and graphics system. The following two interrupt services are most useful:

- DMA Complete: This interrupt signifies that a DMA operation mastered by the graphics system is complete and another can be started. This is best implemented by keeping DMA buffers in a circular list simply adding and re-filling buffers as required. To get the best out of this system the game must try to load the buffers with roughly equivalent amounts of work to get a reasonable overlap.

- Vertical Retrace: This interrupt is generated when the vertical blank period begins and operations can be performed in the front buffer without being visible. Ideally this interrupt should be able to be placed on an arbitrary line of the display, this can be tuned so that the cost of actually servicing the interrupt can be taken into account. As well as being used to indicate when to swap buffers in a double buffered system, the interrupt is useful to the game because it indicates that rendering into the back buffer can now commence.

5 Hardware: System Issues

This section describes some high level issues which should be considered when porting or designing games for 3D accelerators. Some of these issues are naturally specific to particular accelerators, however they are typical of the considerations which need to be made to get peak performance out of a system.

Hidden Surface Removal

Two methods of hidden surface removal are typically available; hardware depth buffering and depth priority rendering. The decision to use one or the other depends on the game and the accelerator.

Depth Buffered Hidden Surface Removal

Depth buffering fits into the philosophy of leveraging work onto the accelerator, however, the use of a depth buffer is probably one of the most significant decisions in architecting a game and needs careful consideration. The cost of depth buffering in terms of memory usage is shown in Table 4, and in terms of fill rates it is seen in Table 2. This section attempts to describe some of the advantages and disadvantages of depth buffering. See also [Akeley] for some interesting uses of the depth buffer and a discussion of depth precision.

Early Depth Testing

Depth testing can often be done early in the pixel processing path, this can avoid costly per-pixel effects (high quality filtering, fogging etc.) being applied to a pixel which will fail the depth test. This means that the depth pass fill rate in Table 2 is effectively a worst case figure, in practice the fill rate will be somewhere between the pass and fail rates. Careful design can maximise the number of depth fails in a scene to increase performance, as described in the next section.

In some cases depth testing cannot be performed early. This situation occurs if a pixel rejection operation takes place based on the result of per-pixel operations, e.g., chroma or fog testing. In this case, if the pixel were rejected after the depth test was performed it would leave the incorrect value in the depth

buffer. This emphasises the need to measure fill rates for the exact set of primitives and modes used in the final game to avoid any unexpected dips in performance.

Arbitrary Rendering Order

One advantage of depth buffered rendering is that objects can be rendered in any order. This can be exploited in a number of ways:

- Depth priority rendering requires objects in a scene to be rendered in back to front order. That is, the objects closest to the viewer, potentially the most important in a scene, are rendered last. If the game is trying to maintain a constant frame rate by adjusting the graphic load in a scene then the most important objects in a scene may be forced to have lowest detail while objects in the background have high detail. Using a depth buffer means that objects can be rendered in order of visual importance, with foreground objects rendered first. Using this technique objects in the background can have detail lowered or be skipped completely to maintain a constant frame rate without large visual anomalies.
- Advantage can be taken of early depth testing by rendering objects in roughly front to back order to maximise the number of depth failures.
- If objects contain expensive state changes then rendering order can be based on grouping primitives with like state together. For example, if changing textures is an expensive operation then all the primitives which share the same texture can be batched and rendered together.

Buffer Clearing

In depth buffered games which update each pixel in every frame the cost of clearing the depth buffer can be avoided. This involves sacrificing one bit of depth buffer precision to effectively split the depth buffer in two halves. Alternate frames are rendered into each half of the buffer, and by manipulating the depth test function and the mapping of depth coordinates to depth values, it can be guaranteed that each pixel rendered in a new frame always win the depth test when being compared to values from the previous frame. Thus eliminating the need to clear the buffer each frame.

This technique is described more fully in[Bigos96].

Unconditional Depth Updates

Often the depth priority of certain elements in a scene is known intrinsically, for example, “the sky can never obscure the ground”, however these elements must have correct depth values in the depth buffer to be able to interact with other more complex elements in the scene. In this scenario the rendering of the sky can be accelerated by rendering it first and unconditionally writing depth values into the buffer, thus avoiding a read of the depth buffer (this is done by setting the depth test function to always, which indicates that no depth read is required).

Partially Depth Buffering

Not all objects in a scene need be depth buffered. If for example a complex object always appears in the foreground of a scene and it can be efficiently ren-

dered with a depth priority algorithm such as a BSP tree then no depth buffering is required. If other objects do interact with this object then it may be efficient to use a BSP tree to enable strict front to back rendering (as described above) to fill the depth buffer with correct values with the optimum number of depth failures.

Depth Visibility Testing

Depth visibility testing leverages additional viewing volume culling onto the accelerator. During the culling stage when a bounding volume is found to lie within the viewing volume the bounding box itself is rasterized into the depth buffer, keeping track of depth pass/fail results while disabling writes to the depth buffer. If all the depth tests fail, then no object contained in the bounding volume can be visible on screen and the bounding box can be rejected. This technique is described more fully in [Greene93] and [DirectX].

This technique is most effective in scenes with a high depth complexity.

Support for this requires extent checking, as described in "Extent Checking" on page 3-19., and the ability to perform depth tests, but disable updates of the depth buffer (usually available via some form of write mask).

Depth Buffering Without a Depth Buffer

It is possible to perform a two pass rendering operation which produces depth buffered scenes without the need for a depth buffer. This technique works by temporarily using the color buffer as a depth buffer. In the first pass the scene is rendered as normal but the pixel colors are discarded and the frame buffer is treated as the depth buffer. In the second pass the scene is rendered and the depth test is used to see if the current pixel has the same depth value as the value in the depth buffer, if it's the same, then this pixel is the closest to the viewer and the color value is placed into the buffer, if the depth value is different the color is discarded. One bit of depth and frame buffer resolution is lost in this scheme to keep track of whether, in the second pass, the buffer contains depth or color values.

The two-pass nature of this technique means its performance suffers when compared to having a dedicated depth buffer. The technique also only provides a 'less than or equal' type test, which means that co-planar polygons may not be treated consistently.

Dynamic and Complex Environments

The complexity of hidden surface removal in highly dynamic and complex environments can be difficult and costly using a depth priority algorithm.

Consider an animated figure with many articulated joints, constructing and managing the rendering of such a figure can be very difficult using a sorting algorithm. However depth testing makes the job trivial by allowing simple ball joints to be used, which makes modelling trivial.

Depth Priority Hidden Surface Removal

With the long list of benefits in using a depth buffer, why would anyone choose not to use one? Memory, memory and memory! Fill rate, fill rate, fill rate!

Memory Architectures

The most common architecture is one which uses a shared memory space for frame buffer, local buffer and texture buffer. The following table gives an idea of the trade-offs involved in selecting screen resolution and depth in a system with 2MB of installed memory (a base level system) for some common screen resolutions.

Screen Resolution	Colour Resolution (bpp)	Buffering	Depth Buffering (16bpp)	Texture Space (Kb)[a]
512x384	8	Double	No	1700 (48)
512x384	8	Double	Yes	1310 (37
512x384	16	Double	No	1310 (37)
512x384	16	Double	Yes	917 (26)
640x400	16	Double	No	1073 (30)
640x400	16	Double	Yes	561 (16)
640x480	16	Double	No	868 (25)
640x480	16	Double	Yes	250 (7)

a. Number in parenthesis represent the number of 256x256 four bit index textures would fit in the free texture space, assuming that the texture palette is stored on chip.

TABLE 4. Memory Allocations on a 2MB System

Clearly depth buffering at high resolutions can significantly impact the texture memory available.

Hi-Level Optimization

This section briefly describes some techniques to improve graphics performance. A more in-depth examination of some of these issues is found in [Rohlf94].

Primitive Selection

Primitive selection can have a significant impact on performance, benchmarking should be used to ascertain the most efficient primitive to use. Generally meshed primitives (triangle and quad strips), are the most effective and can reduce the amount of data sent to the accelerator. Even on systems which support triangles as the base rendering primitive it's worth experimenting, some-

times a system can take advantage of the connectivity of the two triangles forming a quad to reduce the amount of data transferred to the system.

Level of Detail Management

This is a standard technique for trading the visual quality of an object against its rendering cost based on some ‘importance’ criteria, usually the projected area of the object on the screen, or its distance from the eye. This technique is nearly always effective, but be sure to measure the system performance for the different LOD’s, especially if they contain mode changes. Some state changes can adversely impact performance causing a ‘fast’ LOD to give little or no benefit[i].

Using LOD management to implement constant frame rate updates is described in [Funkhouser93].

Remember LOD is also applicable to collision detection and other non graphic related operations which can be degraded gracefully based on some importance criteria.

View Volume Culling

View volume culling is another standard technique to reduce the number of primitives processed in the geometry pipe. This technique will almost always pay dividends, however some care has to be taken in database design for it to work most effectively. Consider a rectangular mesh of polygons used to represent a terrain; this is best represented as a set of smaller rectangular sections for culling, as opposed to say a series of horizontal/vertical strips whose culling efficiency will be highly dependent on the viewing direction.

See also “Depth Visibility Testing” on page 3-22..

Model Hierarchies

Effective use of hierarchical descriptions of a database can lead to very efficient culling strategies, allowing large amounts of data to be removed from a database very quickly.

Database Design

Database design can have a significant effect on performance. Typically the database designer often has the option of adding visual interest and complexity to scene by increasing the polygon count or by adding more per pixel effects

i. An example of this was seen in a system which reduced a polygon model of suspension bridge to a line model at a low LOD. The environment was fogged but fogged lines had not been optimized, resulting in this LOD running slower than a more complex polygon based one!

(fog, texture etc.). The former technique tends to reduce the size of polygons in a scene, the latter tends to increase the size.

Consider building a model of a road with line markings. Possible methods are:

- Use non textured polygons for the road, adding line details by layering additional polygons onto the base polygon.
- As above but tile the polygons to reduce depth complexity and co-planar polygons (which can cause problems with both depth buffered and sorting hidden surface algorithms).
- Use textures polygons with a line markings added as texture details.

The first technique has a low pixel complexity but high depth complexity and may require state changes to change the color of the primitives. The second technique reduces the depth complexity but increases scene complexity. The final technique reduces the scene and depth complexities while increasing the pixel complexity. The most appropriate choice is dependent on the particular circumstances, however this is a good example of the trade-offs required to engineer a high quality real-time database. To make the correct decision you must know if you are likely to be geometry bound or pixel bound, and the real system performance of the different pixel complexities.

State Shadowing

In large object oriented and modularly structured applications redundant mode switching is often encountered. This is a result of creating self contained routines which save and restore state to provide a clean interface to other modules. A good way to find out if this is a problem is to create a trace of state calls made and parameters and check for repeated redundant calls. Adding a thin layer to code to shadow state changes and remove unnecessary ones can sometimes be a big win.

Benchmarking state changes can also be illuminating.

Texture Map Management

Current hardware typically has a fixed amount of local texture memory and textures must be loaded into this memory before they can be used. A game may have many more textures than is possible to store in the local texture memory simultaneously - thus the game programmer is responsible for ensuring all the textures required for a particular frame are present when needed.

Techniques to help manage this include:

- Ensuring that you use the optimum texture format for a particular texture. This generally means selecting the shallowest texel format (to reduce memory) that is aesthetically possible. Generally this means using palette textures.
- Scale the textures to fit into memory (use the hardware to do this when possible to get filtering etc.). Color resolution as well as texture dimensions can be scaled.
- Ensure you're taking advantage of mirroring and repeating modes to get best memory usage.

Inevitably it may become necessary to download textures to the accelerator in real-time, ideally using DMA.

Double and Triple Buffering

Hardware double buffering can be implemented in a number of ways, the most common are described below. Most of these buffer swapping operations should be done by rigging the appropriate operation to the vertical retrace period interrupt to avoid polling and maximise the overlap of work between the accelerator and the host.

Copy Double Buffering

Copy double buffering involves physically copying the pixels from the back buffer into the front buffer. One case when this technique can be used to effect is when the back buffer has a different dimension to the front buffer and the copy is actually performed as a texturing operation including scaling and filtering.

The cost of this technique is high so it should generally be avoided unless the scaling and filtering operations out weigh the cost.

Full Screen Double and Triple Buffering

Full screen double buffering (page flipping) is the lowest cost solution, but as it's name implies can only be used when the full screen is being swapped. The technique works by simply telling the RAMDAC to display a different area of memory, and so happens instantaneously.

When very high, consistent frame rates are required, one way to achieve this is to use triple buffering. Triple buffering uses the additional buffer to allow rendering to begin on a the next frame before the previous frame has been displayed in the front buffer. This removes the need to wait for the swap buffers to take place before starting to render the next frame thus maximising the available rendering time, while minimising the chance of being quantised by the vertical blank period[i].

Color Space Double Buffering

Color space double buffering works by defining each pixel to have a front and back component. An example of this is a 24 bit RGB pixel with the top half allocated to one buffer and the lower half to the other buffer. The RAMDAC is programmed to use either the top or bottom halves to from. The hardware simply has to be able to write into the appropriate nibble without destroying the other nibble (during retrace period).

Per-Pixel Double Buffering

Per-pixel double buffering works in a similar way to color space double buffering except that control of which part of a pixel to display is based on (as it's name implies) a per pixel basis.

Setting and resetting the control bit is generally performed using a block fill operation in conjunction with a write-mask, thus it can be achieved at very high speed.

Because this technique works on a per-pixel basis it can be used to provide some interesting effects. Consider implementing a fighting type game with a high quality rendered background which is panned around in the 'back' buffer, while combatants are rendered directly into the 'front' buffer setting the buffer display bit as they are rendered. The pixel depth of this scenario can be less than 1.0 and the system can run at very high frame rates (60-70Hz). This effectively uses the control bit to provide overlay like functionality.

Tuning Tips

These are a few tips which can be of use when tuning graphics performance, in addition to standard techniques, such as profiling:

- NOP out all the 3D calls to see how fast the system runs, this provides an upper limit of possible performance and helps identify host bottlenecks.
- Run in single buffer mode to see the cost of buffer swapping and if any vertical blank quantization is occurring.
- Run in wire frame mode to get a feel for the geometry processing in a scene when the fill rate is minimal.

i. Remember that in a system which synchronizes buffer swaps with the vertical blanking period you will always run a multiples of the blanking rate, e.g. at a 60Hz blanking rate possible frame rates are: 60, 30,20, 15, 12 etc.

- Calculate the depth complexity by calculating the screen area of all primitives (this can be done in OpenGL using feedback) and dividing by the screen resolution. At this stage you can also produce a histogram of the polygon sizes which can be useful to get a feel for accelerator utilization as shown in Figure 1.
- Always try to reconcile actual performance with theoretical performance.

6 Conclusions

Fully exploiting hardware acceleration is a significant challenge. The strengths and weaknesses of hardware accelerators must be borne in mind when porting and architecting games to run on them. Generally more than a simple port from an existing software 3D renderer is required to extract maximum performance from a 3D accelerated system. It is essential to leverage as much work as possible onto the accelerator, this means the feature set and performance of the system must be understood intimately. The trade-offs and costs of each feature must be considered in terms of accelerator and host loading and how to balance the two - benchmarking achievable system performance is essential.

7 References and Resources

References

[Akeley] Kurt Akeley. The Hidden Charms of Z-Buffer. Iris Universe, Number 11. (Published by SGI).

[Bigos96] Andrew Bigos. Avoiding Buffer Clears. ACM Journal of Graphics Tools. Volume 1, No. 1.

[Delta] GLINT Delta Architecture Overview. 3Dlabs Inc. 181 Metro Drive, Suite 520, San Jose, CA 95110, USA.

[Direct3D] Microsoft. DirectX II SDK (CDROM).

[Foley90] Foley, van Dam, Feiner and Hughes. Computer Graphics: Principles and Practice. Second Edition. Addison-Wesley Publishing Company. ISBN 0-201-12110-7.

[Funkhouser93] Thomas Funkhouser and Carlo Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Environments. Computer Graphics Proceedings, Annual Conference Series, 1993.

[Greene93] Ned Greene, Michael Kass and Gavin Miller. Hierarchical Z-Buffer Visibility. Computer Graphics Proceedings, Annual Conference Series, 1993.

[Heckbert91] Paul Heckbert and Henry Moreton. Interpolation for Polygon Texture Mapping Shading. In State of the Art in Computer Graphics: Visualisation and Modelling, Editors, Rogers David, Earnshaw Rae. 1991 Springer-Verlag, New York Inc. ISBN 0-387-97560-8.

[OGL1] The OpenGL Programming Guide - The Official Guide to Learning OpenGL Release 1, Addison-Wesley Publishing Company, 1993. ISBN 0-201-63274-8.

[OGL2] The OpenGL Reference Manual, Addison-Wesley Publishing Company, 1993. ISBN 0-201-63276-4.

[PERMEDIA] PERMEDIA Architecture Overview. 3Dlabs Inc. 181 Metro Drive, Suite 520, San Jose, CA 95110, USA.

[Rohlf94] John Rohlf and James Helman. IRIS Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics. Computer Graphics Proceedings, Annual Conference Series, 1994.

[Wolberg90] George Wolberg. Digital Image Warping. Los Alamitos, CA: IEEE Computer Society Press, 1990.

Resources

Developer details regarding the PERMEDIA(tm) SDK are available on the 3Dlabs web site, http://www.3Dlabs.com/.

Information about DirectX and Direct3D can be found on the Microsoft web site http://www.microsoft.com/.

http://www.sgi.com/Technology/OpenGL is the OpenGL WWW centre which contains links to OpenGL related information.

Tuning to the Metal

Philippe Tarbouriech
Electronic Arts

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

1 Abstract

Stages in the development of a console game, from concept, prototyping to product. Working within speed and memory constraints. How differences between platforms impact program structure and tuning. Elements of game design and playability.

2 Introduction

In the home, video games have until two years ago been limited to sprite animations or very crude 3D. New more powerful game platforms, such as 3DO, Playstation, Ultra64, Saturn or personal computers (Mac or PC) have raised the level of performance and the available storage space. These platforms now allow real-time 3D texture mapping, involving computer graphics technology reserved until recently to high end graphic workstations.

Game developers on consumer platforms, however, have to struggle with hardware under much stronger economic constraints than graphic workstations. Options such as adding rendering hardware or memory do not exist. In order to compete in a market driven largely by a gee-whiz audience demanding the highest performance and coolest looking games on any given platform, developers have to find ways to use the available hardware in the most efficient way,

under very difficult conditions (lack of stable hardware or OS, primitive tools, Christmas, ...).

Beyond the problems attached to implementing a specific project on a specific platform, game developers are also faced with a growing choice of target platforms and skyrocketing development costs. Those new constraints, added to the volatile demands of a hit oriented business invite flexible game designs that can be prototyped and developed independently from the target platform(s).

There is no magic solution to designing hit video games, when in doubt, make it fun and make it look cool!

3 Shockwave: a case study

Shockwave is a 3D textured mapped action flight simulator for 3DO. It shipped in June 94 and has since sold more than 200 000. It has since then been ported to Playstation, Saturn, Mac and Windows95. A sequel shipped in Christmas 95 for 3DO.

.

FIGURE 1. Shockwave

Original concept

When offered a new platform, game producers are faced with an alternative. Moving an old property to the new platform or developing an original that takes better advantage of the new platform. In June 92, it appeared that the suspected performance of 3DO would allow better textured mapped 3D than ever possible before on a home console. A dream had been floating around Electronic Arts for a while: people wanted a game in which they would be able to fly over the bay area and see their house. That was it for an early design. Several attempts had been already made.

We realized quickly that the memory requirements for textures was way beyond 1 Mb of memory 3DO had at that time. In order to represent a house with one pixel, the resolution of the texture map had to be such that a pixel would represent about 20m x 20m. 1 Mb of RAM could hold a square of texture of approximately 20 Km x 20 Km in 8 bits/pixel representation. That translates in the horizon (how far one can see) being 10 Km away. A CD-ROM is of course a large storage device, but even though we could theoretically store a 200 Km * 300 Km texture on the 660 Mb CD there are 2 limitations to how you can use it: bandwidth and seek time. On such CD players, bandwidth is 300 Kb/s and seek time depends on how much the head has to travel and on the error rate but varies between 1/10 second and 1 second. The classical caching alternative if of course inappropriate with the limited amount of RAM. The CD-ROM bottleneck in effect would limit the freedom of motion of the player way below what is commonly accepted in a flight simulator. 3 Mb in the shipping 3DO is far greater but does not change the nature of the problem as texture is not the only thing we need to fetch, 3D objects, elevations, sounds... and all sorts of other things that make a game share RAM, and disc access with terrain textures.

This early exploration was indeed important as it focused us on the real core of the design, which we kept until the end. What the producer really wanted was not necessarily to see his house, what he wanted was a 3D flight simulator that would be as intense as possible (lots of things to shoot at), as photo realistic as possible (fully texture-mapped) and featuring the highest-quality video.

Early hardware

Unlike PCs or workstations, backward compatibility is not a design goal in game consoles. To some extent, the opposite is true, manufacturers need to differentiate as much as possible their new hardware from their old one in order to push consumers into buying more software. Furthermore, more and more companies compete by introducing constantly new platforms. On the flip side, as software becomes more complex and most importantly, as the volume of artwork becomes enormous, development time increases! The consequence of

those two opposing trends is that development rarely happens on a stable, solid platform. On the contrary, tools are usually very crude, documentation non-existent, benchmarks are projections, and development systems are hard to get. Game development very often involve pioneering.

ShockWave was pipelined with the development of 3DO, and some of our biggest limitations came from having to work with hardware, OS and tools under development:

Hardware put us on a roller coaster: RAM went from 1 Mb to 2 and then 3 in the shipping product, OS memory requirements went up from 200 Kb to some 800 Kb and back down to 660 Kb, math hardware was introduced, CPU speed was supposed to double from 12.5 MHz to 25 but never did, hardware went through at least 5 revisions, etc. The operating system and libraries were on a class of their own: every revision brought changes to the interfaces, had its set of bug fixes and new bugs. Tools went from very crude to quite good, but in that respect, we were in a very good positions, thanks to our own internal tools group. After all this pioneering, developing for 3DO now is relatively easy, but on 3DO's second generation hardware, M2, tools are still primitive...

Maybe the most critical aspect of game development is art creation, and optimization in that area requires understanding in depth the capabilities of the hardware. It took us 2 years to fully understand all we could do with the different graphic formats and options. More importantly, it took two years to educate the artists and to provide them with tools that would allow them to take advantage of those formats.

The main challenge for graphic artists is to produce artwork that looks really cool, takes up very little memory, and renders very fast. (for audio artists, that translates into: sounds really good, takes up very little memory and takes as few DSP cycles as possible). On 3DO, the choice of graphic formats supported by the hardware is pretty large: it includes 1, 2, 4, 6 and 8 bit palette representation, 8 and 16 bit direct RGB. Each mode has its own limitations and specific capabilities: for example, 8 bit palette mode really only uses a 5 bit color palette, the remaining 3 bits in the source are used to refer to shaded versions of the 32 palette entries. Each entry in the palette (to 15 bit color representation) can use one of 2 (you would not want to waste the 16th bit) mode of frame buffer operation (additive, transparent, translucent, opaque...). Another very tricky aspect is translucency: each cel can in most cases have two levels of translucency. Going beyond those 2 levels in order to simulate a real alpha channel requires associating several overlaid cels.

This choice of graphic formats is very confusing as those formats are direct reflections of what the hardware can work with, and as in general, this hard-

ware is irregular due to processor design optimizations. Developing the right tools is critical to producing quality artwork in large quantities.

The only solution to developing on such early hardware is to have a flexible design, one that can move down to accommodate lower performance than expected or move up when things get better. Video game console hardware is always available early, and hardware designers are late for Christmas too...

Limitations and implementation tradeoffs

When any aspect of a game seems at one point to be less demanding than what the hardware can do, a cool new feature is on its way to fill that void... For that reason, the development of a game often hits every possible limit the game console offers: available memory, CD-ROM real estate, CD-ROM bandwidth and seek time, rendering speed, audio channels, ...

Memory

The two major technical limitations ShockWave hit were RAM and CD-ROM access. The main reason was that in order to achieve our goal of intense photo realistic rendering, we needed to work with lots of high resolution textures to load new images often. Our trade off turned out to be: freedom of movement vs. intensity and quality of action. We could either give the player a complete freedom of motion, but then we could not really script missions precisely, and we would have to provide for potentially very long seeks, or we could make the motion completely linear, and just playback a stream of video in the background with overlaid 3D objects

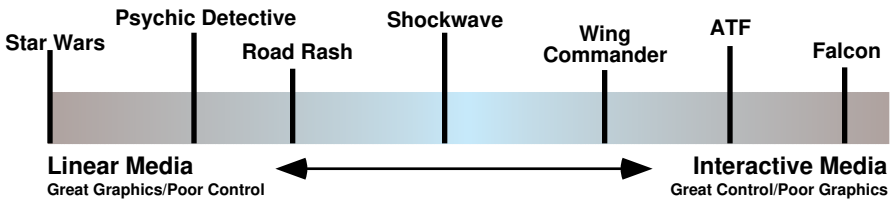


FIGURE 2. Trade off control vs. quality

ShockWave, after several iterations, fell somewhere in between. It provides 3D freedom of motion within a wide path. The world in which the game takes place is a wide band of terrain that allows some amount of flying backwards. This design allowed us to create an engine that would use the available memory as efficiently as possible by reducing the amount of cache, and would spend the least time seeking on the CD-ROM as all the data could be pre laid out in order. Precisely, all data for a ShockWave mission comes from a single

file, and the player controls the rate at which this file is read (and at which obsolete pieces of data are disposed of).

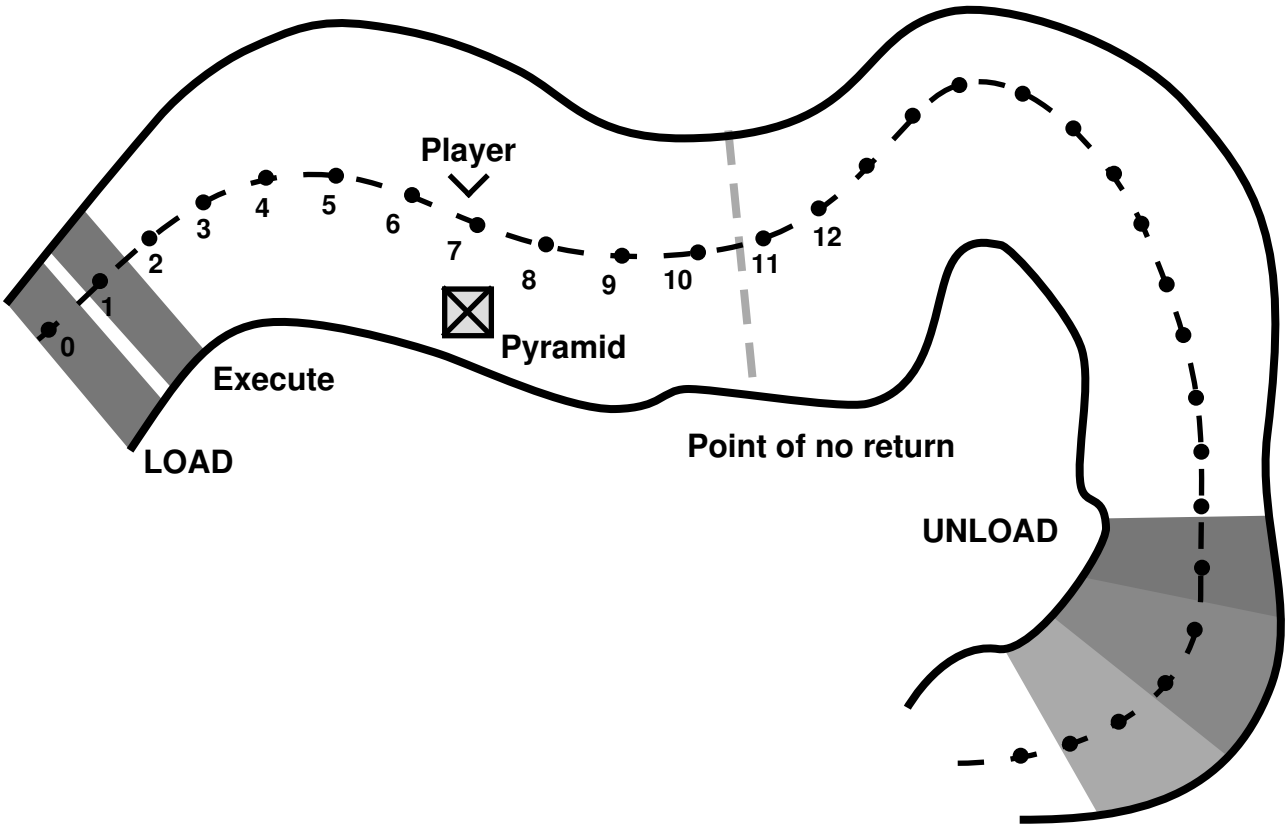


FIGURE 3. Linear flight path

The two major technical limitations were in this case overcome by a design change. By limiting the freedom of the player to a path, we were able to eliminate seek and to optimize RAM usage

Performance

Performance is obviously critical to game play quality. The two main components are latency (how much delay there is between joypad input and feedback) and frame rate (how often is the screen refreshed). The first part of the equation has a straightforward solution: read the player's input very often and give some feedback as soon as possible. In ShockWave, our decision was to synchronize the control model on vertical blanks (VBL). Every VBL, the main rendering and simulation task is interrupted to allow reading the joypad, modifying the control model accordingly, and giving the player some audio feed-

back. This architecture limits latency to 1 VBL, which is quite acceptable for the human race. The second part, frame rate, is more complex as there is a tradeoff between complexity of each scene (rendering and simulation) and frame rendering time. Our experience has shown that above 20 frames per second, the quality of gameplay is very good. At 15 fps[i], there is a slight degradation, and below, one can sense the choppiness of motion. An additional issue is variability: depending on the player's direction, more or less objects are visible, and depending on the mission layout, more or less objects have to be simulated.

We had then to decide whether the frame rate would be constant or variable, and whether the simulation would be frame rate dependent or not. We decided to go for a variable frame rate because even though we always wanted to maintain more than 15 fps, we knew some situations would require more rendering than others, and in order to accommodate those worst cases without dropping objects[ii], we would have to limit the constant frame rate to unacceptable levels. Simulation quality, on its side, only shows in the game play through rendering. As long as behaviors are frame rate independent, a variable frame rate simulation does not cause any perceivable artifacts. On the contrary, this variability makes the integration of acceleration into speed and then speed into position slightly chaotic, and gives a welcome unpredictability to some of the objects' behaviors[iii]. A fixed simulation rate below rendering rate would look quite strange, and above rendering speed, would increase the time it takes to simulate every frame without bringing any perceptible advantages.

For all those reasons, we decided to have a main simulation and rendering loop that would take a short (generally less than 4 or 5) but variable number of VBLs to execute. This also provided for a smooth PAL conversion (adding a 6th VBL every count of 5), and greatly simplifies porting to other platforms.

---

i. In order to avoid flickering, video games often use double buffering, swapping the two frames during a vertical blank. On NTSC, a vertical blank happens every 1/60th of a second. Rendering a frame during less time yields 60 frames per second (fps), rendering in 2 VBLs gives 30fps, and so on 15, 12, 10... For PAL, those frame rates become 50, 25, 16.6, 12.5, 10...

ii. Because we use the painter's algorithm (to render 3D objects from most distant to closest), the worst cases would force us not to render some of the closest, most visible objects, which is quite unacceptable when the player is typically trying to destroy those same objects.

iii. Chaos can be good, but should not introduce discontinuity. One cheap and effective way to reintroduce continuity without adding much overhead is to filter those chaotic variables, for example with a simple filter such as $U_n = \frac{3U_{n-1} + I_n}{4}$ (U is the filtered value, I the chaotic value)

3.4. Optimizations in rendering and simulation

What shortcuts can we find to raise the intensity and quality of action without reducing frame rate, with as little design implication as possible... The following four examples are shortcuts we took:

Bend game design to hardware

A typical problem faced by flight simulator is objects or terrain popping into view. Haze is a good meteorological solution to that problem: blending ground and objects with the sky, in the distance limits how far into the 3D database the rendering engine has to reach. Control over translucency on 3DO allowed us to perform similar functions but proved to be extremely costly in rendering time as the graphic engine needs to read the current frame buffer in order to blend it with each pixel being written. Another feature of 3DO's cel engine allows us, in some modes, to mix any cel with gray (0 to 31). Rendering a grayed cel does not take any longer than normal cels as it does not require transferring more data over the bus. We wished hard that the sky would be gray instead of blue, so that we could use this graying mechanism instead of the translucency. We made it so! The script introduced adequately a poison gas (gray as could be) that the aliens had used to put humans into shock (hence the title of the game). This 3DO specific optimization allowed us to cut rendering time of the terrain with haze almost in half, but it was only possible by adapting game design to technical specifications.

Shortcut 3D

In typical Orwellian style, the original game design mentioned walking tripods which would create havoc in human cities. Tripod legs were quite costly: we needed at least 3 quads[i] per section, 2 sections per leg, and at least 3 legs. That amounted to 18 quads for little surface on the screen. Furthermore, these 3 sided legs looked quite boxy and their apparent width would change a lot depending on the viewing angle. We introduced a new sort of face in our 3D format: cylindrical faces. Each section of a leg would be rendered by one of those faces. Each requires 2 anchor points (the joints), 2 diameters (to allow legs that change width from end to end) and 2 colors (ambient and diffuse). With this information, we compute a sort of specialized billboard which approximates the projection of a long narrow cylinder on the screen (the ends are of course incorrectly straight). The texture we use for this face is a sub portion of a regular gradient texture between the ambient and diffuse color.

i. 3DO only renders rectangular textures into 2D quadrilaterals.

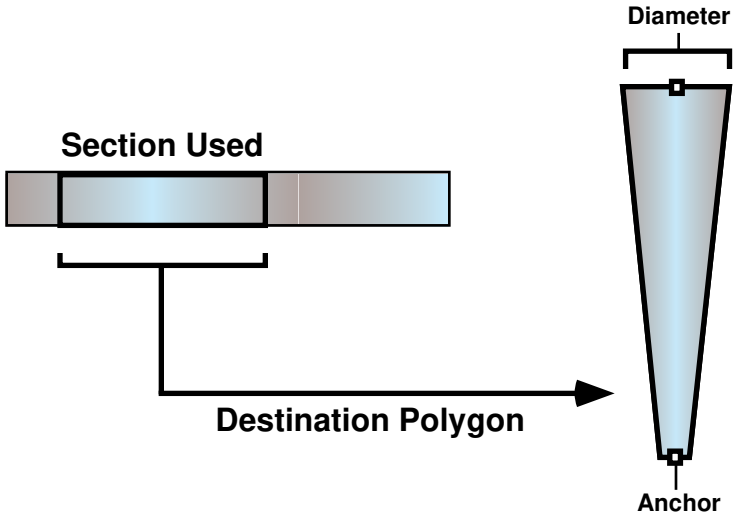


FIGURE 4. Cylindrical textures

Compute the minimum required accuracy

A very common function in a simulation is distance calculation, in 2D or 3D. Over time, lots of approximations have been invented in order to avoid the costly square root function call. The rule for using one approximation vs. another is to use the fastest one (less than 1 pixel error after projection on the screen...). Again, some of the non-linearity in some of those approximations (manhattan, infinite norm...) introduces some spicy chaos to behaviors. The problem comes when calculating object matrices for example, when a good accuracy is needed to renormalize vectors, in order to avoid having uncontrolled squash and stretch...We figured that we needed about 8 bits of accuracy (the screen is 320 wide) in our transformations matrices to avoid visible artifacts.

So we wrote a little distance calculation function that uses a small 2D look up table (512 entries) to give approximately 5% accuracy, and then corrects the error by using an linear (and power of 2) approximation of the circle's slope for the particular look-up table entry.

That error correction (5% accuracy on the error) reduces the overall error to less than 0.25%, and takes about 30 cycles, which was satisfactory for our purpose.

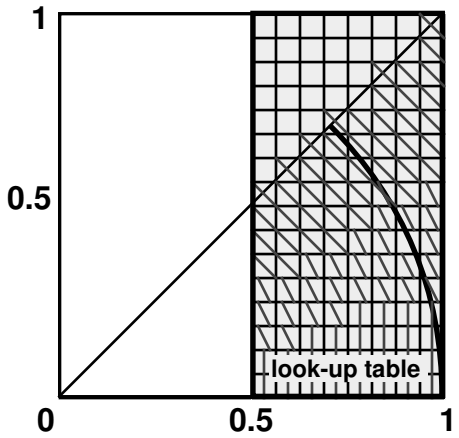


FIGURE 5. Distance approximation.

Preprocess the data

3DO's rendering hardware uses a forward[i] non filtered texture mapping. This presents us with several problems:

- Rendering speed is not only related to destination size, but also to source size (this entire source is parsed, even if it has to be rendered in a single pixel)
- Low res textures stretched to fill a large surface look blocky, or pixellated
- High res textures rendered in a small surface look aliased, especially in motion as each frame will be rendered using different pixels sampled somewhat randomly from the same source
- Geometry is projected properly but textures are rendered in 2D space. This leads to some annoying inconsistencies, especially with large close objects or with the ground.

We came to the classical conclusion that in order to limit aliasing, pixellation, and to optimize rendering time, it was better to use textures that would be projected in a surface roughly similar to their sizes. In other words, we wanted to have a pixel to texel ratio as close to 1 as possible. In a 3D environment, this means using mipmaps instead of bitmaps.

i. A forward texture mapping as on 3DO parses texels and figures what quadrilateral in frame buffer has to be filled with that texel's color.

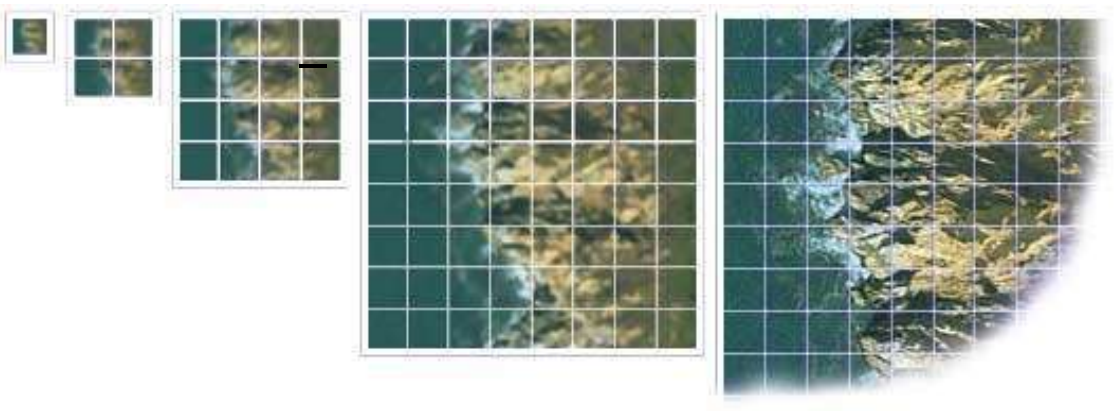


FIGURE 6. Terrain mipmaps

To compensate for the absence of filtering, and to reduce aliasing on highly compressed textures, we also low-pass filtered the lower resolution versions of our textures. The price to pay for a better rendering quality and increased speed is memory, but the surface of the textures drops with the square of resolution, and therefore the added memory requirement turns out to be only around 30%.

This solution does not address the texture distortion issue, but it introduces one of the most efficient ways to reduce rendering time in a 3D application: variable level of detail (LOD).

Level of detail

In a game, the only engineering that counts is one that affects the player's experience. Translated into 3D, this tells us not to worry too much about very distant or invisible things, but instead to focus the console's resources on the same things the player does: large and close objects.

We redirect resources from distant objects to close ones by degrading the quality of representation of distant objects. This degradation is achieved by both simplifying the geometry and lowering the texture's resolution. In ShockWave, costly objects (many faces, large textures...) come in two flavors, a low-res and a high-res. Each object is tested in order to determine the most appropriate transition distance. When rendering an object, its distance to the camera is compared to the transition distance in order to decide which version to use. When frame rate drops, that distance is lowered progressively in order to preserve gameplay at the cost of rendering quality.

Beyond rendering, lots of other operations take time, and the LOD approach can be applied to those as well. In ShockWave, z-sorting and detection colli-

sion are degraded in the distance. The terrain geometry serves as a basis for a bucket sort used to order the objects and compute collisions. As the terrain renderer is multi-resolution (levels of geometry and texture resolution), the quality of object sorting and collision detection drops with the distance to camera.

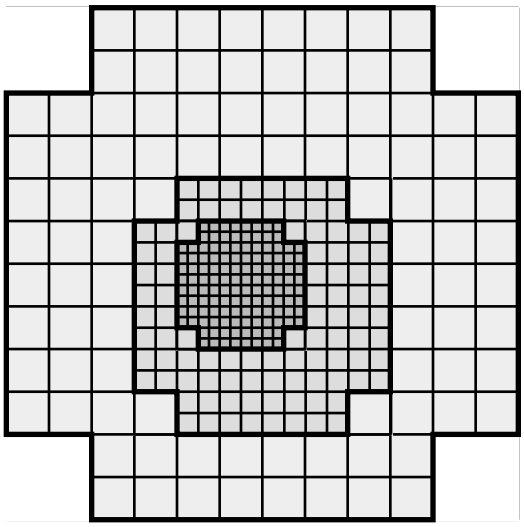


FIGURE 7. Terrain recursion

Conclusion

We took some shortcuts but left many more out. Most shortcuts, though, fall from the same general philosophy: game play is the most important, and there is not just a single correct solution. What makes game software architecture both so interesting and so challenging is the freedom it gives from classical math or physics.

4 Game console issues

New game consoles show up on the market at an alarming rate, and choosing the right one is a difficult challenge. A successful property is often followed by sequels, but is most importantly ported to as many other platforms as possible. This multi-platform environment poses at least two problems: choosing an initial platform, reducing the cost and risk of porting.

4.1.Classes of consumer game platform

There is a multitude of game platforms on the market (even more are extinct) but they can generally be organized in three major classes.

16 bit consoles

This class of machines is represented in large parts by Super Nintendo (SNES) and Sega Genesis. These machines handle sprite manipulation through specialized processors. They generally have a 16 bit general purpose CPU and a few hundred kilobytes of memory.

The strength of these machines is in 2D animation: they handle scrolling planes very efficiently, and render sprite display lists with no cost on the CPU. Those machines are generally ROM based which provides a fast access to all game data.

Sound is very primitive (some simple FM synthesis), colors are few and processing power is low. The only relative strength of these machines is in scrolling planes: most newer machines (except Sega Saturn) handle only polygon texturing, and this makes scrolling an expensive operation in bus bandwidth.

The installed base of 16 bit machines is huge (> 100 million), but it has become very much a hit only business, only top 5 games sell well. This concentration of business has driven most smaller developers to newer platforms and only the biggest companies continue to sequel their games on 16 bit machines.

There is disagreement as to when exactly those machines will disappear, but they are rapidly moving out of the picture.

Home computers

Sales of home computers have exploded in recent years. A large portion of those support CD-ROMs and audio synthesis: they are often referred to as multimedia personal computers. This large installed base (more than 20 million new multimedia home computers in 95) is attractive to game developers, especially since there is no license fee associated to shipping a Mac or PC game.

Game consoles of one type rarely come in different flavors: all SNES games work on all SNES machines (whether Nintendo or a third party edits it). Personal computers, on the other side, come from many vendors, and even Macintoshes have clones now. This variety poses a compatibility problem. Educational or ‘multimedia’ titles often have limited technical requirements, and run on most multimedia personal computers. Action games, on the other side, are more performance oriented and can only run at acceptable frame rates on faster machines with appropriate graphic and audio subsystems.

The difficulty is to define a set of technical requirements that does not limit the market too much while allowing the game to push the envelope of what customers expect from their computers. For example, most 3D games on PCs today require a pentium processor or a PowerPC. 3D games that could still run on a 286 processor would look very poor compared to the state of the art, would not have any technological appeal, and most probably would not sell. On the opposite, shipping today a P6 only game would not make sense as the installed base is still very small.

Almost every subsystem in a PC comes in different flavors: CPU (speed, type), CD-ROM (bandwidth, seek, DMA), audio board (sample playback, FM synthesis, ...), bus (ISA, EISA, PCI, local bus), OS (DOS, OS/2, Win3.1, Windows95), memory size, screen (VGA, SVGA, 8 or 16 bits...), display accelerator (2D, 3D, texture mapping). Specifying a minimal set of requirements is a difficult art. Still, one has to make bets, and trust the biggest players, such as Intel or Microsoft. They claim a typical system for Christmas 96 will run Win95, have a Pentium 90, 8 or 16Mb of RAM, quad speed CD-ROM, 16 bit audio board, PCI display board), support for 8 and 16 bit display up to 800*600 pixels.

This specification has to be very clear to the customer, but it is only the first part of the problem. Even more important are testing and installation. A universal installer is almost impossible to achieve as the variety of systems is so large, and as new devices are introduced almost every day. The only solution is to support the market leaders (SoundBlaster, ...) and make sure a large enough share of customers will be able to install and use the game. As a matter of facts, most PC games are updated rapidly after the first release, sometimes to fix bugs, but more often to support systems the initial product left out.

When a game is installed on a customer system, it has to assess what the system is capable of. Most DOS products go straight into a setup screen in which the customer is supposed to input his system's setup (IRQ, DMA, Joystick...). This sort of interface is very repulsive to most new users, and in order to keep the returns to a minimum, more elegant solutions should be used. On Macs, the system provides information to the application through a standard API and Windows95 has finally a similar database on system installation. These informations allow a game to run, but they do not directly give performance related information. Such information is necessary to scale rendering quality and frame-rate in order to provide the player with the best possible experience. Most often, the best approach is to benchmark the system at run-time, by rendering some test cases, and to degrade quality, bit depth, or screen resolution for the frame rate to be acceptable (between 15 and 30 fps). This automatic setup should of course be overridable by the player.

The most notable change in PC game landscape will come from 3D accelerators. A growing number of board manufacturers are announcing or shipping 3D acceleration boards and they all have different features, performance, interface... and once again game developers are faced with the issue of selecting an API, defining a specification, etc. directX on Windows 95 is now the defacto standard for games on PC and direct3D, the 3D graphics component of directX, will probably be a very common interface to 3D accelerators. In 97, we should expect very few DOS games.

32 and 64 bit consoles

All these machines are successors to SNES and Genesis. They have introduced much more powerful CPUs (RISC based) and typically use CD-ROMs as storage media instead of more costly cartridges. Along with the main CPU, they have specialized coprocessors to handle 3D math, audio, and rendering into a 16 bit frame buffer. 3DO and Jaguar shipped first, and were followed by Saturn and Playstation. Ultra64 and M2 are to come next.

However similar, these consoles have major architectural differences. Each one has a number of very specific optimizations: Playstation has a 2D frame buffer in which textures and frame buffers have to be stored, Saturn has an array of processors, memory banks, buses... Those architectural differences often put specific limitations on resource allocation: for example as to how many textures can be used at a time, how many audio samples, etc. Each limitation has to be worked around specifically in the case of a port.

Other critical differences include features (Gouraud shading, MIDI music, alpha channels, CD-ROM streaming) and also tools (compilers, libraries, debuggers, graphic and audio editors), but very often, the most complex problems when porting from a console to another arise because of platform shortcuts. For example, texture mapping on Playstation is linear triangle based, and this approximate texture mapping introduces unacceptable distortions on quads that looked fine on 3DO. Math computation on Playstation are 4.12 accuracy instead of 16.16, which is not enough in many situations. Saturn is an even more complex console, and many developers predict it will take years to get the most out of the 7 or so processors it contains.

The best thing about those proprietary consoles is that a game is always a breeze to install for the gamer, and is guaranteed to work. The worst thing is that such proprietary designs are generally very irregular and sometimes plain weird. The challenge is to make games that take advantage of each platform, and are easy to port. Part of the solution lies in knowing early in the design which specifications are fundamental for the game to work, and which are for eye candy. This way, it is possible to implement the same core game on all consoles, and scale the different implementations on the eye candy part.

Choice of target

Of these 3 classes, as cartridge sales for 16 bit game consoles are declining fast, 32 bit game consoles and personal computers are really to be preferred as targets for new games, especially if these involve real-time 3D computer graphics. However, PC gamers and console gamers are two very different crowds, and for most purposes can be considered independently. PC gamers generally favor realistic simulations, strategy games, and reflection games while console gamers prefer action, racing and fighting games.

A necessary step: prototyping

It is very hard to predict the success of a new game concept and it is very hard to predict the success of a game platform. There is no magical formula for building successful games, or for predicting which console will prevail. Nevertheless, it is possible to reduce the risk first by evaluating game design as early as possible, potentially before implementation, and second by delaying the choice of a target platform, until the market is better understood. Prototyping a game on a high level system, abstracted from most gruesome implementation and optimization details, seems like an elegant solution. The risk is in ignoring fundamental limitations, such as memory space, CD-ROM bandwidth, rendering speed, etc. and in ending up designing unimplementable games. The only way to avoid such pitfalls is to initially decide on a set of specifications (for example: 2 Mb DRAM, CD-ROM, triangle based texture mapping, MIDI sound, 320x240 16 bit frame buffer), and then to perform reality checks on critical aspects: memory map, number of polygons per frame, complexity of calculations,... along the way.

Beyond reducing the risks, validating a game design on a workstation also improves productivity, by giving artists a shorter turnaround time when visualizing their work, and by allowing tuning of behaviors, physics, and gameplay in a powerful, comfortable and stable programming environment.

5 Conclusions

Computer graphics in video games are reaching new levels of performance with accelerated 3D and texture mapping. To improve even more the quality of graphics and frame rate, many classical computer graphic shortcuts can be taken, but new ones also have to be invented. In particular, as correctness is not an objective in video games, all compromises are valid as long as they improve gameplay.

Focusing on gameplay also encourages developing functional prototypes before tuning to the metal. This abstraction, if safe guarded by a good under-

standing of game platform capability, reduces some of the risk associated with game development and helps shorten production cycles.

Gameplay is a very important key both to find powerful optimizations, and to simplify game development across multiple platforms.

Database Design for Visual Simulation and Entertainment

Wes Hoffman
Paradigm Simulation

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

Abstract

Within the last year technical advances in hardware and software have dropped the price of 3D graphics and opened the entertainment market to technology previously available only to high-end simulation. Companies approach this new technology from two directions. Game companies struggle with technical issues while simulation companies try to tackle game content and game play. As with software and hardware, database engineers must learn new tricks and construction techniques that apply to this developing market.

The technical issues of database design created by low end 3D entertainment are much the same as those of high end simulation. Concepts such as frame rate, LOD, culling, and transformation rate are common to both worlds. Database engineers from the high end market are very familiar with these problems and their impact on database design. Game developers must learn the tricks of the 3D trade and the artists who were currently limited to 2D content must now learn 3D and learn all the associated performance considerations.

These course notes will address the technical issues of database design as well as the subjective nature of content and appearance. The technical problems covered range from understanding hardware specs and their impact on the database to optimizing geometry and textures for culling, drawing, and query. The artistic aspects include how to best use polygons for maximum effect, database consistency, and the art of simulating reality.

1 Types of Databases

Geospecific vs. Geotypical

"Geospecific" describes a database that has been modeled to correlate at some level to a `real' of terrain. There are several methods of creating geospecific databases. These databases are used primarily for military applications, however Microsoft Flight Simulator and a few other games have built real-world cities and terrain. Geospecific databases can be generated from data sources such as DTED and DFAD automatically, although the resulting database is generally less than ideal.

"Geotypical" databases do not represent an actual area of the world but are built to give the impression of a type of terrain. Geotypical databases are much more common than geospecific. A database modeler has a higher level of control and can, ironically, generate a more believable reality when they are not restricted to reproducing a real place. To be able to build a successful geotypical database the modeler must have a well developed creative sense.

Synthetic vs. Hand-Modeled

Synthetic databases may be created with various programs that involve a high level of interactivity between the user and the resulting database. Parameters entered by the modeler control fusion of data sources, polygon densities, and phototexture application. Emphasis in methods of this type is placed on a front-end user interface which carefully controls the resulting database. The ability to twiddle some parameters and pass elevation and cultural data to a program that spits out a perfect database is attractive. Many attempts have been made to write a program that does this with varying degrees of failure. Software that makes use of fractal equations is also frequently used to generate synthetic databases. Striking geometrical representations often result from these equations, though care must be taken to assure that the resulting geometry has the characteristics desired. Synthetic databases have the advantage of not requiring large amounts of memory to store pre-rendered geometry but leave the resulting database at the whim of a random number generator. Furthermore, while synthetic databases allow for unlimited terrain, the quality and performance will ultimately suffer.

Hand-modeled databases give the modeler the most control over database content and structure while providing the greatest freedom for creativity. This is the most labor intensive construction technique, but is necessary when the runtime environment is limited and the database determines the speed and realism of a game. Hand modeling a database will be the focus of these course notes.

2 What's in a Database

Terrain

Most databases for games and military applications use some sort of geometrical environment through which to navigate. For these purposes, such environments will be referred to as terrain. Terrain typically consists of all ground planes, large and small area features (i.e., forests, oceans, roads), and anything else which will remain static in a given application.

Models and Cultural Features

Models and cultural features include everything in a database from factories to animated eggs on skis. They generally are modeled to a high degree of detail from which subsequently lower levels of detail are derived to control overall scene density. Models allow the modeler to instance an object throughout a database.

Textures

Textures are rectangular arrays of RGB data (texels) that can be applied to the surface of geometry to create the illusion of detail. Textures may be derived from photographs, synthesized from programs, created using paint packages, or virtually anything else which will output an image which may be mapped to a polygon.

Materials

In addition to texture, some hardware platforms support the rendering of specialized material properties in real time. A polygon may be given the attributes of a desired material which will control its appearance with regard to lighting. Common material properties include transparency and lighting properties. Lighting properties include the materials ambient, diffuse, emissive, and specular reflectivity.

Application Information

All geometry in a database may include information which helps an application control the scene. This information is embedded in the database and processed by the game to improve or control aspects of the database not related to appearance. Typical information includes bounding boxes, values for controlling scene content in overload conditions, surface codes which may be queried to aid in special types of collision detection, or motion constraints that control the animation of an object.

3 Generating Database Specifications

Hardware Limits

The first step to building a real-time 3D database is to understand the limits of the target hardware. The performance numbers for graphics hardware make statements about polygons per second, pixels per second, and CPU speed. The Sony PlayStation quotes 500,000 polygons per second, the Sega Saturn 900,000, and an Silicon Graphics ONYX benchmarks over a million polygons per second. These numbers are useless when generating a database design. First one must define a polygon as a triangle; polygon limits do not refer to anything other than triangles. Second, rendering a polygon is a combination of two unique graphics steps. The vertices of a polygon must be transformed into screen space, and the resulting surface must be filled. One can increase the number of polygons drawn by optimizing meshing or caching (depending on the hardware). It is generally a good idea to run benchmark test to determine what sort of geometry and scene density the target hardware can handle.

To generate a database specification one must know not only the realistic polygon throughput of the target hardware, but the desired framerate, field-of-view (FOV) and viewing distance (far clip) of the application. A high speed flight simulation requires a larger far clip, larger overall database, and medium to low terrain detail. By contrast, a ground based simulation with a slow, rumbling tank would use a smaller far clip, greater detail, and a smaller database. By defining your type of application, you necessarily define the parameters of your viewing frustum from which follow the calculations required to maximize the performance of your hardware.

Scene Complexity and Polygon Budget

A common term used in simulation is scene complexity, which refers to the number of visible polygons. Scene complexity is related directly to the transform and fill characteristics of the specific hardware. Most applications have specific scene and depth complexity requirements. When you begin designing your database you should be able to compute an approximate number of terrain surfaces and models that can be seen in a single frame based on the desired scene complexity.:In the following example the polygon budget for a typical helicopter is calculated. A total polygon budget of 7000 polygons at 30 frames/second (Hz) is assumed. The first step is to break a scene down into its individual elements. This example helicopter simulation will require the following features:

1. dynamic models (planes, tanks, etc.)
2 static models (house, trees, etc.)
3. special effects (rotor blades, missile trails)

4. instrumentation
5. terrain

Each of these scene elements requires polygons and must be include in the final calculations. The number of polygons used by dynamic and static models must be calculated based on the number of objects in a scene and the current level-of-detail (LOD):

	polygons	**vertices**	**# visible**	**total polys**
LOD0	100	200	4	400
LOD1	75	126	20	750
LOD2	30	15	40	1200
LOD3	5	10	70	350
TOTAL				2700

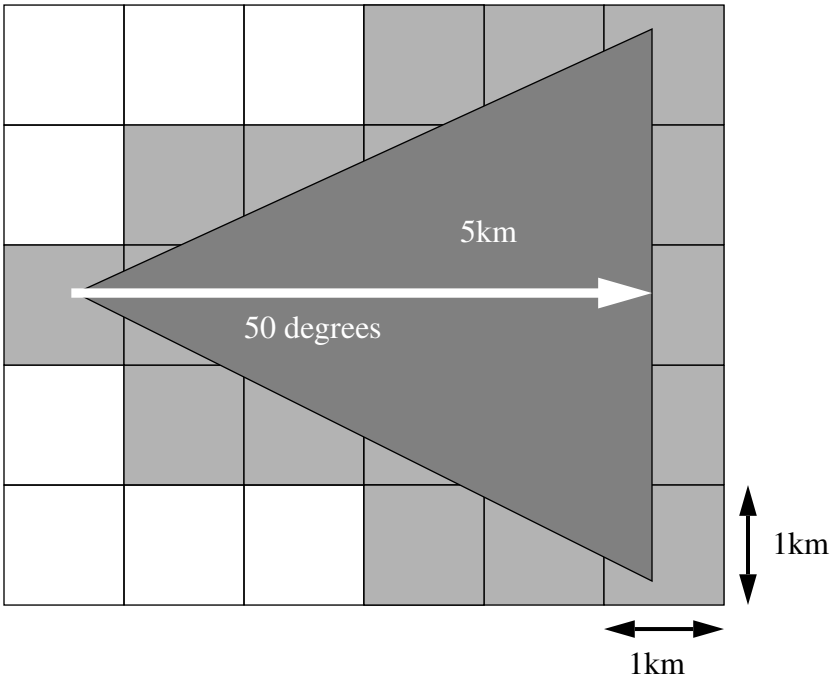
Static and Dynamic Polygon Counts

Special effects and instrumentation will be allocated 1000 polygons, leaving 2300 polygons for the terrain. The next step is to compute the polygon density of the terrain given the remaining polygons. Figure 1 demonstrates how to calculate terrain density and budget polygons based on a 50 degree FOV and 5km far clip.:

By precomputing the number of surfaces you can use to create each area and model you can begin to get an idea of what you will be able to represent with the number of available polygons. By knowing the characteristic geometry of your viewing frustum and the maximum number of vertices your hardware can transform into 3D space, you can calculate LODs with ranges based on average densities for models and terrain at each level of detail.

4 Making a Database Perform

The process of optimizing a database has many variables. The database must take advantage or not only the hardware but also the software driving the simulation. Obviously the frame rate of a simulation is a combination of hardware, software, and database. Understanding how the hardware/software works, enables the database engineer to “tune” a database during its construction so that the simulation software can function efficiently. There are three basic functions that a 3D simulation must perform: drawing, culling, and collision detection. As discussed below the requirements of these functions unfortunately contradict each other.



A = Field-of-View		50 degrees	
B = far plane		5 km	
C= avail polys for terrain		2300 polys/frame	
km^2/frame	=	tan(A/2)*B*B	= 11.66
avg polys/km^2	=	C/(tan(A/2)*B*B)=	197.3

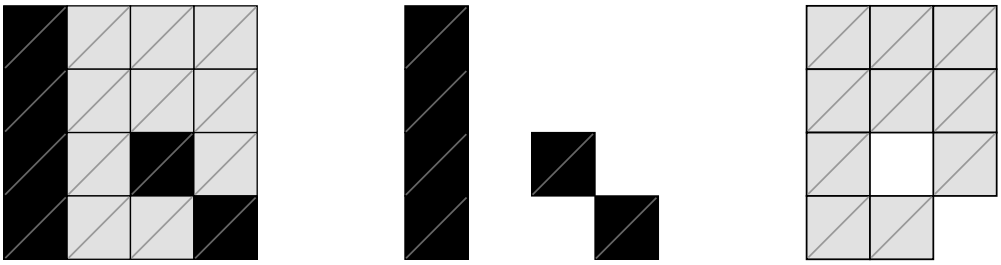
FIGURE 1. Field Of View Extent

Optimizing for Drawing

The render speed of the target hardware is the most obvious feature to optimize. The graphics performance issue a database must address are transformation and fill rate. A more subtle, but no less important, hardware restriction is graphics mode switching.

Because hardware can only transform so many vertices a second, limiting the number of transformations that occur is the first place to start. As mentioned previously, polygon performance is not a reliable benchmark; the number of vertices used by the polygons is much more accurate. There are rendering algorithms available that attempt to reduce vertex transformation while maintaining polygon counts, namely tri-mesh strips or fans. By including as many polygons as possible in a strip one can achieve substantially more polygons

while not increasing vertex count. The modeler should be aware of this feature since the methods use to construct geometry and apply texture will dramatically effect the mesh efficiency of the run-time database. For a vertex to be include in a mesh it must share all the aspects of the other vertices in the mesh and the polygons they creates must have the same texture. material, and draw mode. A vertex consists of the x,y,z position, and optional u,v texture coordinates, normals, or colors. If any of this data is not shared by two polys at a common edge the mesh must break. Sharing as much vertex data as possible can increase the polygons they create by over 200%. In the following figure a simple patch of a terrain polygons are combined to achieve the best rendering performance by minimize the number of vertices and mode switching:



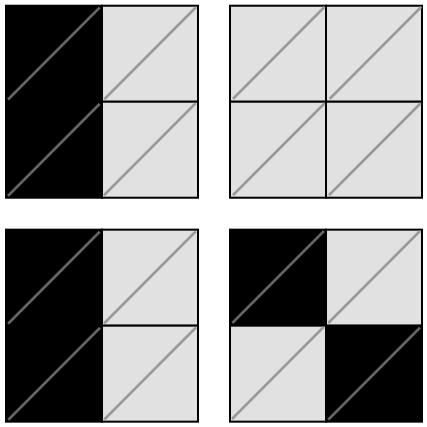
polygons	meshes	vertices	mode switches
32	7	46	2

The second major limiting factor of hardware is fill rate. This is a more difficult feature to optimize since the depth complexity (the average number of times each pixels is written into) of a database varies eyepoint position. However the database designer can make conscious decisions when building geometry that can reduce depth complexity and optimize fill. An example of this is the construction of lower LODs. The modeler knows the object will occupy a small number of pixels so it may be more advantageous to concentrate on mesh efficiency or reducing mode changes.

A hidden hardware limitation is graphics mode switching. This include switching texture, materials, draw modes (backface/frontface), and transparency. In order to reduce mode changing it is best to group like polygons by mode. Reducing graphics mode switching is a problem best solved by the application software, however the database designer should consider this bottleneck since the minimum number of texture, materials, and draw modes used is defined by the models they create.

Optimizing for Culling and Intersection

A critical component of real-time simulation is efficient culling. Efficient culling requires the application software to reduce the entire database to only those polygons visible in a single frame. However, because of the optimization methods used for drawing, polygons become grouped into meshes that can extend outside the visible FOV. An efficient culling and query database should be spatially grouped (similar to a quad tree). This enables the software to quickly determine which groups can be seen. To increase the complexity of the problem, the modeler must also consider the query (or collision) aspects of the database. The following figure shows the previous patch of terrain optimized for culling.

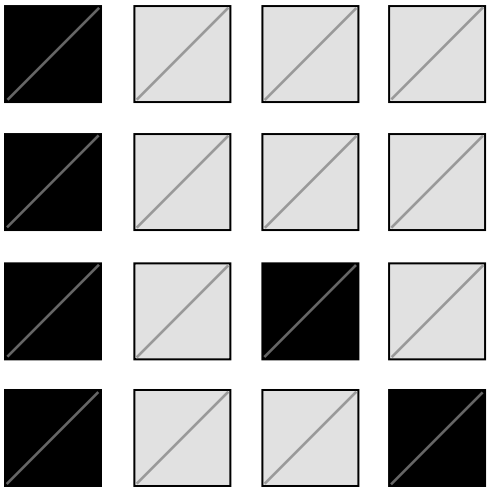


Optimal culling configuration

polygons	meshes	vertices	mode switches
32	10	52	9

For most simulation the same polygons used to represent the visual database are used mathematically to determine terrain intersection and object collision. If the smallest primitive in a database is a mesh, then the intersection testing can only reduce the possible target geometry to that in the mesh. At this point, CPU-intensive algorithms must take over and check each triangle in the mesh for a possible intersection. Obviously, the more vertices in a strip the more surfaces must be checked. The best database for culling and intersection would

not contain meshes at all but individual surfaces. Here is the same terrain patch optimized for intersection and collision testing:



Optimal intersection configuration

polygons	meshes	vertices	mode switches
32	16	64	16

This paradox would seem to imply the necessity of two databases, one for drawing / culling, and one for intersections. The conflicting goals of culling and drawing is the most frustrating problem a database engineer must conquer.

The Proper Balance

A well trained database engineer should understand the algorithms used to draw, cull, and perform intersections with their database. Too much emphasis on optimizing the databases draw time will more than likely hurt the application's culling and query operation. Every vertex and texture added will impact all aspects of the game. For this reason the best constructed databases strike a proper balance between the performance restrictions of each software and hardware function.

5 Making a Database Attractive

As technology improves, more techniques and possibilities are available to the real-time database artist. Games are now three dimensional with all the trappings of high end simulation. Suddenly the prospect of making a database that

closely resembles reality is realized and the database artist must struggle to understand all the options now at their fingertips

The Impressionist Nature of Modeling

Building a successful database is much like creating a impressionist painting. The resemblance to the object being modeled is implied. The talent behind a believable database lies in the database artist's ability to capture the essence of a building through minimal polygons and textures. If you put your eye close to the canvas of a Monet or a Van Gogh painting you lose the realism you had when you viewed the painting from a distance. The same is true of polygons and textures. The polygons and textures in a database must be built to be viewed from a reasonable distance. This approach solves 99% of the problem since the odds of one viewing a house or a tree from a distance are much greater than the chance that the eyepoint will end up glued to the objects surface. The bulk of the artist's efforts should be concentrated around the big picture approach, though some effort should be paid to subtle detail. The more detailed an object, the closer the eyepoint can approach before the impression is lost. In short, don't model the bricks, model the wall.

Consistency

The most important visual aspect of a database is consistency of style and detail throughout a scene. For example, a database may use only intensity texture and color, the viewer will accept these liberties as realistic only if the entire database in modeled to he same level of realism. If a photo-realistic house is put in the same database, the illusion is shattered. The consistency of style is refers to this relationship between all visual aspects within a scene. Slight style difference between one database artist and another may also ruin the illusion of reality.

To generate consistent detail throughout a database a modeler must select a level of realism that fits within their polygon budget and stick to it. If one starts with too much detail they will run out of polygons before they run out of database to draw; suddenly the rest of the database becomes surprisingly simple. Conversely, if too little detail is created the database will not be as realistic as it could given the polygon budget.

Imitating Nature by Grouping

In all high end simulation the goal it to simulate reality as close as possible. Unfortunately, reality rarely cooperates with the designs of the database modeler. In reality objects are grouped together, there is no even distribution of features, so maintaining uniform scene density becomes almost impossible. One of the talents of the real-time database artist is dispensing database density

consistently but allowing for natural concentrations of data without dead space. The spacing and density of data groupings should depend on the field-of-view and far clipping plane of the game eyepoint. The following figure the grid represents a terrain grid, the dots are concentrations of detail, and the FOV is the triangle. The grouping on the left provides evenly spaced areas of detail to avoid overload conditions. The grouping on the right my cause frame rate problems due to the concentration of detail in the center of the terrain:

efficient and inefficient grouping of detail

6 Putting It All Together

A successful database is a combination of art and science. The quest for realism is a constant struggle between appearance and performance. A skilled real-time database engineer must understand all the technical issues as well as their impact on the imagery they can create. The bottom line of database modeling is the ability to use polygon, vertices, and textures effectively. There are two concepts a professional modeler must understand in order to get the most visual impact from a limited amount of data:

Avoid hard to model features

Some terrain or model features translate better to polygons and textures that others. Luckily, much of the world fits well into this medium. There are large amounts of replication and flat surfaces in features such as roads, warehouses, farm land, office buildings, and so forth. These are much easier to create realistically than features such as industrial sites, trees, coastline, or rivers. While building a realistic tree may be a modeling challenge, do not wastes the polygons and textures if the tree is not a critical element in the application. Save your graphics for what is important.

Don't over-model

The tendency to over-model a feature is hard to avoid. Over modeling refers to the process of putting to much detail in an object than will ever be seen. An object may have twice the polygon count necessary to represent it effectively or a texture may be double the size it will ever be seen. Additionally LOD switch ranges are easy to correct when the switch is too close, making it noticeable, but not if the switch range is too far. The database modeler will never know if a feature has too much detail, since there are no visual cues to signal a problem. A database may fit within all the specifications originally proposed, but because so much detail was wasted on unnecessary information the database is lacking in precious visual information.

Making an efficient database that takes advantage of all the hardware's performance and making it visually appealing are the job of the database engineer/artist. The ability to balance all the conflicting requirements of database design and created an environment that allows the software to push the edge of the hardware limitations is the art of the profession.

Prototyping and Portability of Hardware-Assisted 3D Games

Eric Johnston
LucasArts Entertainment

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

1 Introduction

Until very recently, the use of three-dimensional graphics in computer and video games has been limited entirely to what could be done in software. At SIGGRAPH '96, we see the introduction of a variety of products which claim to bring dazzling, eye-popping real-time 3D graphics into the home.

As pixel counts and poly-per-second specifications fly to and fro, game developers are presented with the task of creating entertaining content which will survive the hardware and API battles. This part of the course will attempt to introduce a few of the major issues, and present one possible battle-proven solution.

2 Competition

No matter how beautifully it is rendered, reflected, shadowed, or shaded, a whirling teapot-ahedron is not sufficient to motivate most game players to spend five hundred dollars on a new display board. Manufacturers of graphics accelerators, armed with this knowledge and some really nice teapots, have been courting computer game developers for months, asking them to produce special versions of their games that derive some advantage from the hardware.

One primary problem with the situation so far has been the following: If the developer modifies an existing game, designed to run entirely in software, then

the market expansion due to hardware acceleration (the number of people who would buy the game because of this feature) is extremely small. Most of the early adopters of these 3D boards are likely to already own fast computers, on which the original game needs no acceleration. If, on the other hand, the developer goes full out in support of the hardware, writing a game which is not playable without it, the developer risks a staggeringly low sales figure for the game, reliant entirely on the hardware's performance.

Simply put, it takes roughly twelve to eighteen months to develop a good computer game (or even a bad one). Graphics accelerator manufacturers need great games on their hardware in order to have game players widely adopt it. Game companies need to know that a graphics accelerator has been widely adopted in order to commit the time to develop great games for it.

3 Software-Only Rendering

There are plenty of 3D games available for most computers and game machines. In the absence of graphics acceleration hardware, game programmers have drawn on creative methods for years, producing all-software methods of rendering real-time 3D games.

In order to keep adequate speed, game-specific shortcuts are taken, as in the following categories:

Flight Simulators

Flight engines are very good at drawing a horizon and some airplanes, with extremely limited transform and sorting needs, and no hierarchies.

Dungeon-wall Engines

Using specialized trapezoid-drawing routines, the texture-mapped world consists of vertical walls and horizontal floors, with bitmap-based character graphics.

Sprite Zoomers

Flat bitmaps are scaled up as they scream towards the camera.

In addition to these and other de-generalized graphics techniques, faster computers and more clever programming techniques have led to rapid recent improvements in software-based 3D engines. There are, however, strict limits on what can currently be done in software. To crank out 200,000 textured, z-

buffered polygons per second into a buffer bigger than 320x200 pixels, the CPU is going to need some help.

4 Hardware Assistance

The only display hardware commonly found on desktop computers is a pixel buffer. Cartridge-based game machines usually add sprites and scrolling bit-planes, to help draw rapidly-moving game characters.

Hardware-assisted 3D texture engines were placed into the hands of game developers just a few years ago, and are only now making a debut in the consumer arena.

1992 SGI and E&S Supercomputers (RealityEngine, ESIG 2000)

1995 Consoles (Sony, Sega, Nintendo, 3DO)

1996 Desktop Accelerators for Mac & PC

5 Division of Hardware Types

Truth in advertising

“3 billion pixels per second”

“500,000 lit, texture-mapped triangles per second”

“Phong shading”

As flashy terms and figures are tossed around by hardware manufacturers, it is extremely important for developers to keep in mind that there are many different types of 3D graphics accelerators. The following is a list of some of the most fundamental divisions, which have a large impact on the development process, and the maximum performance which can actually be obtained.

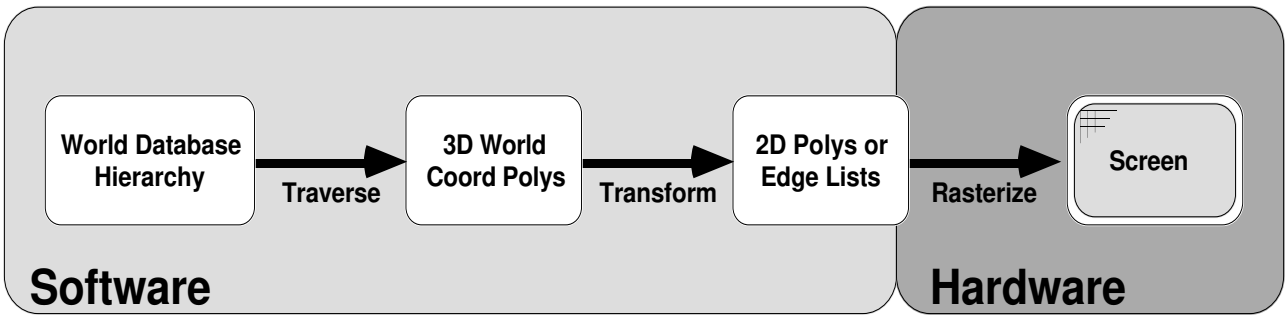
Major Division 1: Type of acceleration

The most basic question to ask a piece of graphics acceleration hardware is “Exactly what do you accelerate?” The answers vary widely, and have a direct

impact on many aspects of development. They can, however, typically be divided into three distinct types, as follows:

Type 1: 2D Polygon Rasterization

Although this is currently the most common type of "3D graphics" board, there is really no 3D involved. This hardware gets polygons in screen coordinates, usually pre-clipped, and renders them to the screen. This type of board will often boast extremely high polygon counts, under the assumption that you have written magic software which can transform, light, and clip polygons as fast as the board can spit them out.



Type 1 Graphics Accelerator

Advantages:

Inexpensive

Common

More app control & flexibility

Disadvantages:

Speed

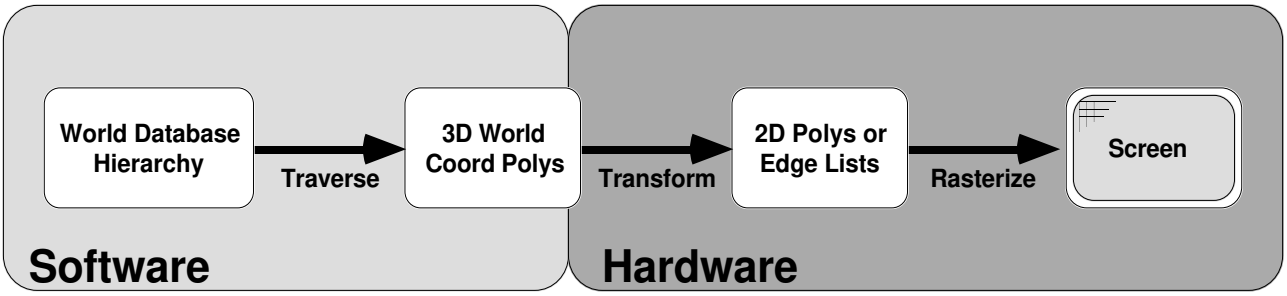
Bus traffic

Complete lack of lighting or other 3D facilities

Most of the work is still done in software

Type 2: Transformation Stack-Based Renderers

GL Programmers are familiar with type 2 engines, as most SGI machines fall into this category. The graphics hardware contains a model matrix stack, a viewing matrix stack, and some rendering attributes. It is then handed polygons, vertex by vertex, in world coordinates.



Type 2 Graphics Accelerator

Advantages:

Speed

Transform assist in hardware

Light, fog, etc. may be in hardware

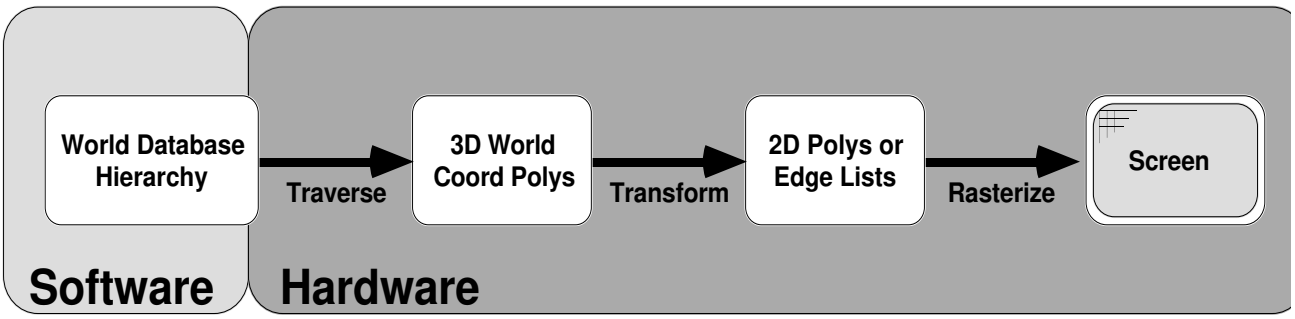
Disadvantages:

More expensive

Slightly less app control & flexibility

Type 3: Fully Independent Database Rendering

As a superset of Type 2 above, this type adds the ability to traverse a display tree on its own, rather than having the CPU hand over one polygon at a time. Matrix pushes and pops, as well as display state changes, are stored in a form which the traversal engine can accept. This model follows SGI's Performer paradigm.



Type 3 Graphics Accelerator

Advantages:

Speed! Speed! Speed!

Extremely light bus traffic

Parallel operation

Disadvantages:

More expensive

Least app control & flexibility

Major division 2: Occlusion handling

(*not* collision handling)

An extremely large amount of the complexity of a 3D game involved occlusion handling. Most software-only engines must use pre-sorting, octants, or BSP trees (provided the world holds still). However...

Z-buffered versus everything else

An increasing number of graphics acceleration products are introducing z-buffering in hardware. The development time and CPU time saved by efficient use of z-buffers will come back in the form of better, faster, more creative games. It's worth it.

Major division 3: Floating point

It's there, but is it in the pipeline, or just on the CPU? Few to none of the consumer products currently available have 32-bit floating point support in the rendering hardware. This simply means that polygon vertices and transformation matrices must be converted to the hardware's native fixed-point format in order to be useful.

The only real floating-point issue, then, is how happy the CPU is about performing object logic in floats.

Major division 4: Texture filtering

This may seem like a minor point, but it's not. The ability of a graphics engine to perform filtering of oversized texels has formed a razor-sharp dividing line between platforms.

This feature is entirely dependent on the hardware, as it cannot be done in software. A machine which has the ability to filter and blur upscaled textures also inherits the advantage of cleaner mip-mapping, resulting in less pop-and-flicker (an artifact which has become a trademark of software-based texture engines).

6 Graphics API Advantages and Caveats

In the increasing mayhem of 3D graphics acceleration, a little standardization is a good thing. If the API (Application Programmers' Interface) is sound and functional as a means to access the hardware, then it is certainly better than forcing the developer to learn the hardware's intimate secrets. Example software, written to the API, is typically easy to read, and shows that the hardware company is serious about its speed and generality claims.

The other common use for graphics API's is to fill in the gaps left by the hardware. Referring to the hardware acceleration types above, a type 1 board can become more generally useful when some software is added to perform lighting, transformation, and clipping, effectively transforming the whole package into a type 2 system, with the 3D stage existing in software.

Doubtless, some developers will go around the API, on the grounds that it is too thick, or that they can implement it better themselves. In many cases, if the API software people have done their job, this assumption will be wrong. The people whose jobs center around making the API as efficient as possible are well motivated by the competition.

SGI's Performer

A solid example of a well implemented API currently in use is Performer from SGI. Performer takes a RealityEngine (or other type 2 SGI machine) and transforms it into a type 3 machine. The group at SGI responsible for this software took a good look at the set of things that real-time simulations have in common with one another, and wrote a solid, canonical implementation which runs as efficiently as possible on each SGI hardware configuration.

Companies need to find their emphasis

Some API's currently available have grown from being methods of accessing one piece of hardware into proposals for cross-platform standards. The previous year's API wars have seen companies attempting to "sell" their API's to companies for large sums of money, on the idea that the API will be operational on all new platforms as they come out.

The primary problem with this is that the hardware companies which originated these API's in the first place are not set up to serve as a Ministry of Standards, and will be unready and unable to help all of the developers who have adopted their methods, especially on competing hardware.

Why some API's won't work on some machines

Consider a not-so-hypothetical type 3 API which not only allows the programmers access to the vertex, polygon, and matrix data, but requires them to manipulate it directly. The API has a "native" database format, which it knows how to traverse. This native format was probably derived from the format that its original graphics accelerator needed.

A problem occurs when the API is moved over to a real type 3 accelerator. This graphics engine absolutely needs the database in its native format, or it will be unable to traverse and draw. It is unlikely that the ported API will be re-converting the entire database from its own copy frame by frame, so the unsuspecting developer must rewrite volumes of code in order to make the game work on the new platform. Sort of defeats the purpose of using a "cross-platform" API in the first place.

7 A Cross-Platform Development Approach

Designing for a Type 3 System

So, how does one construct a game which will survive the transition to various hardware platforms? One extremely stable method of doing this is by designing and implementing the game for a type 3 system in the first place.

Components: Data Structures

Scene Tree

Platform Specific

This is the equivalent of a Performer tree. It contains geometry, transformation matrices, switches, and rendering attributes.

Attribute Lattice

Platform Independent

Attribute lattices contain a "stickman" connectedness diagram of the world. They are used for AI navigation, visibility culling, collision handling, fog, ambient sound, and a variety of other spatially determined world characteristics.

Object Data

Platform Independent

Every object in the game, from spaceship to timer to teapot, needs state information. Using a type 3 API abstraction, this data can be nearly 100% cross-platform.

Keyframe/Motion Data

Platform Independent

Camera Structs

Platform Independent

Sound/Music Data

Platform Independent

Components: Software Modules

World Manager

Platform Specific

The World Manager is, among other things, the game's interface to any given type 3 API. It is responsible for knowing the native data structures and processes, so that the individual objects don't have to.

Object Code

Platform Independent

Every object in the game has program code to govern its behavior. Using a type 3 API abstraction, this code can be nearly 100% cross-platform.

Keyframe/Motion Manager

Platform Independent

Camera/Handoff Manager

Platform Independent

Sound/Music Manager

Platform Specific

8 Conclusion

In order to assure portability among 3D graphics accelerators, a game should be written for type 3 acceleration, only dipping below for special effects which will not travel among platforms. When it is brought to a type 1 of 2 machine, a layer of interface software should be written, if it does not exist already, to bring the "effective" platform back up to a type 3.

While this does restrict some operations, it by no means requires content of a game to be "watered down" to the lowest common denominator. On the contrary, it will allow full usage of each of the target platforms, without the necessity to fundamentally re-engineer the game.

They key point which sums up all of this chapter's lessons is the following: When writing a game for more than one platform, take some time to develop a razor-sharp definition of what data will and will not travel across platforms, and hold to it.

GUF: Grand Unified File-Format Level 0 Format Description

Scott Watson
Walt Disney Imagineering

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

1 Background

GUF[tm] is a backwards compatible data exchange format developed at Walt Disney Imagineering. GUF is the second generation of the VR Studio's in-house file format, EGG. EGG was used in the production of the Aladdin Virtual Reality attraction at EPCOT.

Disney's application required that they support features not found in any common exchange file format, namely vertex morphing, skeletal (vertex level -vs- polygon level transforms) and animation channels for animating these features.

GUF incorporates all of these features and is designed with extensibility and backwards compatibility in mind.

Disney is making the GUF technology available for consideration as a standard. In "legalese", these are the terms:

> Use of GUF[tm] technology by recipient shall not grant to Disney any rights in recipients intellectual properties or other business assets; the grant of use by Disney shall not grant to recipient any rights to the name Disney in any form or any fanciful characters of Disney. Disney retains all rights in and to the trademark GUF and it may not be used without prior written consent from The Walt Disney Company.

There is a GUF mailing list! To join, send a message to:

guf-request@disney.com

2 Description:

GUF is primarily designed as a backwards compatible exchange format. Ideally, exchange formats should possess clear semantics and be simple to parse. To the extent possible tools, should continue to operate in predictable ways, even when applications introduce new objects. “GUF level 0” is primarily for the exchange of DATA OBJECTS, not procedural information.

In designing GUF we had the following goals:

Easy to parse
SINGLE PASS parsing and object construction (** Network/stream friendly)
Extensible (Self describing)
Backwards compatible

GUF consists of rules for constructing legal GUF “forms”, primitive data types and a set of primitive forms.

GUF is not an object system, it is primarily a syntax. Secondarily, it is a base set of well known (and non-controversial) forms like “(polygon...)”, “(vertex ...)” etc. which can be further specialized by your applications.

3 Key Concepts

There is only one tree in the parser, the “form tree”. This tree which informs the parser which forms (constructors) can be substituted for others. The parser’s form tree is constructed at run time by “def-form”. Type checking is assumed to be performed by the primitive constructors.

Pseudo BNF:

<ident>	:=	[legalchars]+	
<slot-id>	:=	<ident>=;; ie: x= , color=	
<string>	:=	“.*”	;; including newline
<int>	:=	[-][0-9]+	
<real>	:=	[-][0-9]\.[0-9]	;; ie: .9 1.9 1.
<form>	:=	(<ident> <arglist>)	
<slot/value>	:=	<slot-id> <prim>;	; ie: key= 12
	|=	<slot-id> <form>	;; ie: key= (* 3.1415.)
<arg>	:=	<form>	
	|=	<prim>	

```
                |=      <slot/value>
<arglist>       :=      <arg>
                |=      <arglist> <arg>
<symbol>        :=      ‘<ident>
<prim>          :=      <string>
                |=      <int>
                |=      <real>
                |=      <symbol>
                |=      <bool>
<bool>          :=      “#t”
                |=      “#f”
<comment>:      =       ;; .*                   ;; ignore to EOL

legalchars =    alpha “~!@#$%^&*-_+=<>?/”;; legal symbol chars
```

4 *Primitive/Special Forms*

```
;; storing and retrieving values
   (set!  <ident> <arg>)        ->      <obj>
   (set!! <ident> <arg>)        ->      <obj>
   <ident>                      ->      <obj>

   ;; Form definition
   (def-form <symbol>
         [ is=  <symbol>]       ;; “parent form”
         [ has= ‘( ... )]       ;; defaults
         [ <symbol> ]* )        ;; (additional) field names
                                ->      <::undefined::>

   (guf <arglist>) -or-         ->      <arglist>
   ‘( <arglist )                ->      <arglist>
```

<obj> above is the result of "evaluating" arg. Primitives are "self" evaluating", for example 12.34 evaluates to 12.34, 'foo evaluates to 'foo, identifiers evaluate to whatever they have been set to, etc.

Note:

The simple version of (ref ...) can only reference previously defined values. See below for explanation of forward reference behavior.

==========

In normal forms, arguments are evaluated first, from RIGHT -> LEFT. This rule extends to the form name as well. Forms evaluate to a typed value.

Forms that function primarily through side-effects, may evaluate to "<::undefined::>", if they do, they will be ignored, and not passed to the enclosing form.

Objects returned by forms are assumed to support a minimal API consisting of a default constructor, and a slot assignment function. Also, besides returning values, forms are also required to return the <type> of the value (or return objects which support dynamic type-id) in order to allow enclosing forms to perform type checking.

A naked identifier evaluates to the value bound to it via set! or set!!, assuming it has already been bound.

Optionally, arbitrary topologies can be reconstructed if we add a general purpose forward reference mechanism. One simple (parser level) method for doing this would be to extend the identifier evaluation function as follows. If the identifier being referenced has not been bound yet, the parser creates a "fix-up structure" and places it in a work-list. The fix-up structure contains: a reference to the current object being initialized, the slotname that would be have been paired with the returned value (if it had been bound) and the identifier itself. This is something like scheme's (letrec ...).

Example of forward and circular reference:

```
(set! poot
      (group name = "some-group" poot)))
```

There are two distinct types of "set" statements, (set! ...) and (set!!...). The first form of set creates bindings with lexical scope the second creates TOP-LEVEL bindings (or parser global) scope.

5 Issues/Features

Binary GUF

BINARY GUF is semanticly equivalent to ASCII GUF. The only difference is that the data stream is pre-tokenized and that primitives (ie: <real>s) are sent in their binary form. There is NO GLOBAL TOKEN enumeration. Each application can generate it's own token/symbol pairs and even add to it on the fly, the only constraint is that prior to sending pre-tokenized data that the sending application dump it's symbol to the receiver. Ask me about SYMBOLS!

Setting DEFAULTS

We can _easily_ associate default arguments (slot-name/value pairs) per form. It may be useful to be able to modify the defaults in the middle of the data stream as a way of expressing shared state. We've added a special form called "form-has"; it looks like (form-has <formname> '(....)).

```
(form-has polygon
        '( color= red double-sided= #t ) )
```

which is 100% equivalent to:

```
(form-has polygon
        (guf color= red double-sided= #t ) )
```

after which all polygons IN THE CURRENT SCOPE will default to being two sided and red when constructed. Please note that this capability is supplied by the parser alone, no modification to the underlying object system is required.

In fact, this fits nicely in def-form:

```
(set! red (rgba 1. 0. 0. 1.))
(def-form redpoly
        is= polygon
        has=    '( color= red double-sided= #t )
)
```

The (form-has ...) function follows lexical scoping rules. This means that definitions at the top-level are GLOBAL defaults and internal defaults disappear when their enclosing scope is left. Since GUF evaluates in a standard order (left -> right) we can nest defaults and get predictable (and desirable!) behavior.

```
;; constrain Z
(def-form vtx3 is= guf ‘x ‘y ‘z)
(def-form xy-vtx is= vtx3 has= ‘( z= 5. ))
(vtx-pool
        (xy-vtx 1. 1.)   ;; -> 1. 1. 5.
        (xy-vtx 2. 2.)   ;; -> 2. 2. 5.
        (form-has xy-vtx ‘( z= 3. ))
        (xy-vtx 3. 3.)   ;; -> 3. 3. 3.
         (def-form x-vtx is= xy-vtx has= ‘( y= 4.))
         (x-vtx 4. )      ;; -> 4. 4. 3.
        )
(vtx-pool
         (xy-vtx 1. 1.)  ;; -> 1. 1. 5.
        )
```

6 *Example #1 - Backwards compatibility*

```
test-file.guf
//
// The following is an example of how GUF supports backwards com-
patibility
//
// definitions
//
(def-form rgb
  is= guf
  ‘r ‘g ‘b )                ;; base form

(def-form rgba              ;; “derived from rgb”
  is= rgb
  ‘a
  has= ‘( a= 1.0 );; default alpha is 1.
  )

// etc...
```

```
(circle
 color = (rgba 1. a = .5);; translucent red circle
 )
```

Assumptions:

A system which does support rgb but not type rgba when fed “test-file.guf” would do the following:

Enter circle form
Create a default circle and assign to obj
note that the next value should be assigned to the slot-name “color”
Enter rgba form
since this parser doesn’t support rgba it punts and
uses the “rgb” form.
Create a default rgb and assign to obj
obj.set_slot w/ slotname=”r”,type=<real>,val=1.
/* the red field is set to 1. */
obj.set_slot w/ slontame=”a”,type=<real>,val=.5
/* rgb ignores this information */
exit rgba form
obj.set_slot w/ slotname=”color”,type=<rgb>,val=(rgb 1. 0. 0.)
Exit circle form

The constructors themselves should type check prior prior to setting the slots.

Obviously - the best we can do in this situation is be “wrong in a good way”. The idea is that, in many cases behavior like the above may be good enough to get the job at hand done.

==========

===== NOTE =====

==========

This is currently under revision! Your comments are actively pursued!

Please mail all comments to: scott@disney.com or call at 818-544-6790.

Lexically guf is identical to scheme.

7 Sample GUF File: stickman.guf

```
;; stickman.guf (GUF v.9)
;; Info--
;; Source File: stickman-morph
;; Created on:  Mon Feb 20 23:21:51 1995
;;         by:  alias2guf -vfmw Stickman stickman-
morph
;; Pathalias--
;; stick:"/fat/people/aladdin/user_data/test/pix/"
;;

(dart
 name= "Stickman"
 (dcs #t)

 ;; Base of skined object
 (skel-morph

  (set! Polyset#2
(m-vtxPool
 (m-vtx 0.449956 16.337658 -13.723756 )
 (m-vtx 0.987580  16.305292 -13.427006 )
 (m-vtx 1.204791  16.282518 -11.054466 )
 (m-vtx 0.082079 16.345008 -10.489552 )
 (m-vtx 0.951188 15.675961 -13.429715 )
 (m-vtx 1.168399 15.653186 -11.057175 )
 (m-vtx 0.045687 15.715676 -10.492261 )
 (m-vtx -1.169168 15.785950 -10.497708 )
 (m-vtx -1.132776 16.415281 -10.494999 )
 (m-vtx 0.413564 15.708327 -13.726465 )
 (m-vtx -0.801291 15.778600 -13.731912 )
 (m-vtx -0.764899 16.407932 -13.729203 )
 ))
  (set! Polyset#3
(m-vtxPool
 (m-vtx 0.086105 16.388311 10.518361
(m-uv 0.000000 0.450000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
)
 (m-vtx 1.209545 16.333132 11.082589
(m-uv 0.000000 0.650000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
)
 (m-vtx 0.994807 16.376303 13.455070
(m-uv 1.000000 0.650000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
(d-xyz 4 1.348694 0.000000 0.529844 )
)
(m-vtx 0.457432 16.410153 13.752105
(m-uv 1.000000 0.450000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
(d-xyz 4 1.348694 0.000000 0.529844 )
)
 (m-vtx 1.174476 15.703774 11.090867
(m-uv 0.000000 0.850000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
)
 (m-vtx 0.959738 15.746945 13.463348
(m-uv 1.000000 0.850000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
(d-xyz 4 1.348694 0.000000 0.529844 )
)
 (m-vtx -1.128888 16.456094 10.524504
(m-uv 0.000000 0.250000
     (d-uv  6 1.0 0.0 )
     (d-uv  7 0.0 1.0 )
     )
)
 (m-vtx -1.163957 15.826735 10.532782
(m-uv 0.000000 1.250000
     (d-uv  6 1.0 0.0 )
```

```
    (d-uv  7 0.0 1.0 )
    )
)
 (m-vtx 0.051036 15.758953 10.526639
(m-uv 0.000000 1.050000
    (d-uv  6 1.0 0.0 )
    (d-uv  7 0.0 1.0 )
    )
)
 (m-vtx -0.792630 15.848578 13.766526
(m-uv 1.000000 1.250000
    (d-uv  6 1.0 0.0 )
    (d-uv  7 0.0 1.0 )
    )
(d-xyz 4 -1.204191 0.000000 0.000000 )
)
 (m-vtx 0.422363 15.780795 13.760383
(m-uv 1.000000 1.050000
    (d-uv  6 1.0 0.0 )
    (d-uv  7 0.0 1.0 )
    )
(d-xyz 4 1.348694 0.000000 0.529844 )
)
 (m-vtx -0.757561 16.477936 13.758248
(m-uv 1.000000 0.250000
    (d-uv  6 1.0 0.0 )
    (d-uv  7 0.0 1.0 )
    )
(d-xyz 4 -1.204191 0.000000 0.000000 )
)
 )
)
  (set! Polyset
(m-vtxPool
 (m-vtx -1.000000 9.015649 -0.008487
(m-uv 0.248649 0.418305
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 9.015649 -2.017877
(m-uv 0.073227 0.418305
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 11.015649 -2.017877
(m-uv 0.073227 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 11.015649 -0.008487
(m-uv 0.248649 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 9.015649 -2.017877
(m-uv 0.926773 0.418305
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 11.015649 -2.017877
(m-uv 0.926773 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 4.517996 2.601173
(m-uv 0.441587 0.226323
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 0.020343 2.601173
(m-uv 0.441587 0.034341
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 0.020343 0.591784
(m-uv 0.335045 0.034341
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 4.517996 0.591784
(m-uv 0.335045 0.226323
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 4.527383 -0.508713
(m-uv 0.175103 0.226724
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 0.039117 -0.508713
(m-uv 0.175103 0.035142
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 0.039117 -2.518102
(m-uv 0.060165 0.035142
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 4.527383 -2.518102
(m-uv 0.060165 0.226724
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 9.015649 2.000903
(m-uv 0.573763 0.418305
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 11.015649 2.000903
```

```
(m-uv 0.573763 0.503675
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 11.015649 -0.008487
(m-uv 0.751351 0.503675
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 9.015649 -0.008487
(m-uv 0.751351 0.418305
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 11.015649 2.000903
(m-uv 0.426237 0.503675
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 9.015649 2.000903
(m-uv 0.426237 0.418305
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 0.020343 2.601173
(m-uv 0.558413 0.034341
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 0.020343 0.591784
(m-uv 0.664955 0.034341
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 4.517996 2.601173
(m-uv 0.558413 0.226323
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 4.517996 0.591784
(m-uv 0.664955 0.226323
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 0.039117 -0.508713
(m-uv 0.824897 0.035142
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 0.039117 -2.518102
(m-uv 0.939835 0.035142
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 4.527383 -0.508713
(m-uv 0.824897 0.226724
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 4.527383 -2.518102
(m-uv 0.939835 0.226724
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 12.015649 -1.999097
(m-uv 0.077435 0.540680
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 17.015649 -1.999097
(m-uv 0.086231 0.756044
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 17.015649 2.000903
(m-uv 0.413832 0.756044
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 12.015649 2.000903
(m-uv 0.422624 0.540680
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 17.409076 -1.233409
(m-uv 0.125572 0.772754
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 17.869200 -1.233409
(m-uv 0.128446 0.792297
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 17.869200 1.167193
(m-uv 0.367174 0.792297
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx -1.000000 17.409076 1.167193
(m-uv 0.370040 0.772754
     (d-uv  5 0.0 1.0 )
     )
)
 (m-vtx 1.000000 12.015649 2.000903
(m-uv 0.568979 0.551676
     (d-uv  5 0.0 1.0 )
     )
```

```
)
 (m-vtx 1.000000 11.015649 2.000903
(m-uv 0.573763 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 11.015649 2.000903
(m-uv 0.426237 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 17.015649 2.996205
(m-uv 0.439223 0.756044
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 17.015649 3.991508
(m-uv 0.453402 0.756044
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 15.017003 3.991508
(m-uv 0.460753 0.671154
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 15.017003 2.996205
(m-uv 0.448505 0.671154
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 12.015649 -1.999097
(m-uv 0.930966 0.551676
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 11.015649 -1.999097
(m-uv 0.926179 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 17.015649 2.000903
(m-uv 0.559633 0.764523
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 17.015649 -1.999097
(m-uv 0.940318 0.764523
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 11.015649 -1.999097
(m-uv 0.073821 0.503675
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 15.017003 -2.985010
(m-uv 0.949254 0.679632
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 15.017003 -3.970923
(m-uv 0.961288 0.679632
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 15.017003 -3.970923
(m-uv 0.039442 0.671154
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 15.017003 -2.985010
(m-uv 0.051675 0.671154
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 15.416910 -6.979815
(m-uv 0.978567 0.696618
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 15.416910 -9.988708
(m-uv 0.984977 0.696618
    (d-uv  5 0.0 1.0 )
    )
(d-xyz 2 0.871974 -1.027684 0.000000 )
)
 (m-vtx -1.000000 15.416910 -9.988708
(m-uv 0.016581 0.688139
    (d-uv  5 0.0 1.0 )
    )
(d-xyz 2 -1.121110 -0.778548 0.000000 )
)
 (m-vtx -1.000000 15.416910 -6.979815
(m-uv 0.023640 0.688139
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 17.015649 -2.985010
(m-uv 0.060983 0.756044
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx -1.000000 17.015649 -3.970923
(m-uv 0.046826 0.756044
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 17.015649 -2.985010
(m-uv 0.958979 0.764523
```

```
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 17.015649 -3.970923
(m-uv 0.968865 0.764523
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx -1.000000 17.015649 6.996196
(m-uv 0.472893 0.756044
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx -1.000000 17.015649 10.000883
(m-uv 0.480943 0.756044
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx -1.000000 15.416910 10.000883
(m-uv 0.483439 0.688139
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx -1.000000 15.416910 6.996196
(m-uv 0.476414 0.688139
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 15.017003 2.996205
(m-uv 0.550569 0.679632
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 15.017003 3.991508
(m-uv 0.538520 0.679632
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 17.015649 2.996205
(m-uv 0.540875 0.764523
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 17.015649 3.991508
(m-uv 0.530978 0.764523
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx -1.000000 18.323315 -1.497176
(m-uv 0.115785 0.811585
      (d-uv  5 0.0 1.0 )
      )
(d-xyz 1 0.000000 0.131757 -2.004587 )
)
 (m-vtx -1.000000 21.081348 -1.497176
(m-uv 0.130630 0.928730
      (d-uv  5 0.0 1.0 )
      )
(d-xyz 1 0.000000 2.908063 1.364625 )
(d-xyz 3 0.000000 1.661971 0.000000 )
)
 (m-vtx -1.000000 21.081348 1.416799
(m-uv 0.365000 0.928730
      (d-uv  5 0.0 1.0 )
      )
(d-xyz 1 0.000000 2.908063 -1.374036 )
(d-xyz 3 0.000000 1.661971 0.000000 )
)
 (m-vtx -1.000000 18.323315 1.416799
(m-uv 0.379842 0.811585
      (d-uv  5 0.0 1.0 )
      )
(d-xyz 1 0.000000 0.197635 2.136344 )
)
 (m-vtx 1.000000 17.409076 1.167193
(m-uv 0.590686 0.781233
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 17.869200 1.167193
(m-uv 0.586189 0.800776
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 17.409076 -1.233409
(m-uv 0.913256 0.781233
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 17.869200 -1.233409
(m-uv 0.917639 0.800776
      (d-uv  5 0.0 1.0 )
      )
)
 (m-vtx 1.000000 21.081348 -1.497176
(m-uv 0.960134 0.937209
      (d-uv  5 0.0 1.0 )
      )
(d-xyz 1 0.000000 2.908063 1.364625 )
)
 (m-vtx 1.000000 21.081348 1.416799
(m-uv 0.542027 0.937209
      (d-uv  5 0.0 1.0 )
      )
(d-xyz 1 0.000000 2.908063 -1.374036 )
)
 (m-vtx 1.000000 18.323315 1.416799
(m-uv 0.569106 0.820064
```

```
    (d-uv  5 0.0 1.0 )
    )
(d-xyz 1 0.000000 0.197635 2.136344 )
)
 (m-vtx 1.000000 18.323315 -1.497176
(m-uv 0.934186 0.820064
    (d-uv  5 0.0 1.0 )
    )
(d-xyz 1 0.000000 0.131757 -2.004587 )
)
 (m-vtx 1.000000 17.015649 -9.988708
(m-uv 0.987488 0.764523
    (d-uv  5 0.0 1.0 )
    )
(d-xyz 2 1.058826 0.903116 0.000000 )
)
 (m-vtx -1.000000 17.015649 -9.988708
(m-uv 0.019080 0.756044
    (d-uv  5 0.0 1.0 )
    )
(d-xyz 2 -0.903116 1.121110 0.000000 )
)
 (m-vtx -1.000000 17.015649 -6.979815
(m-uv 0.027169 0.756044
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 17.015649 -6.979815
(m-uv 0.982133 0.764523
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 17.015649 10.000883
(m-uv 0.512497 0.764523
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 15.416910 10.000883
(m-uv 0.515005 0.696618
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 15.416910 6.996196
(m-uv 0.521383 0.696618
    (d-uv  5 0.0 1.0 )
    )
)
 (m-vtx 1.000000 17.015649 6.996196
(m-uv 0.517826 0.764523
    (d-uv  5 0.0 1.0 )
    )
)
 )
)
```

```
;;
;; Skeleton Data
;;

(joint
 name= “RootJoint”
 (transform
  matrix= (mat4x4
   1.000000 0.000000 0.000000 0.000000
   0.000000 1.000000 0.000000 0.000000
   0.000000 0.000000 1.000000 0.000000
   0.000000 0.000000 0.000000 1.000000
  )
 )
 (joint
  name= “spine1”
  (transform
   matrix= (mat4x4
    1.000000 0.000000 0.000000 0.000000
    0.000000 1.000000 0.000000 0.000000
    0.000000 0.000000 1.000000 0.000000
    0.000000 10.019320 -0.043357 1.000000
   )
  )
  (m-vtxRef
   1 3 4 6 16 17 18 19 38 39 45
   48
   pool= Polyset
  )
  (joint
   name= “rhip”
   (transform
    matrix= (mat4x4
     1.000000 0.000000 0.000000 0.000000
     0.000000 1.000000 0.000000 0.000000
     0.000000 0.000000 1.000000 0.000000
     0.000000 -0.031612 1.043555 1.000000
    )
   )
   (m-vtxRef
    15 20
    pool= Polyset
   )
   (joint
    name= “rknee”
    (transform
     matrix= (mat4x4
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 -4.994824 0.537589 1.000000
)
     )
     (m-vtxRef
      7 8 9 10 21 22 23 24
```

```
      pool= Polyset
      )
     )
    )
   (joint
    name= “lhip”
    (transform
     matrix= (mat4x4
      1.000000 0.000000 0.000000 0.000000
      0.000000 1.000000 0.000000 0.000000
      0.000000 0.000000 1.000000 0.000000
      0.000000 -0.063225 -0.917064 1.000000
      )
     )
    (m-vtxRef
     2 5
     pool= Polyset
     )
    (joint
     name= “lknee”
     (transform
      matrix= (mat4x4
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 -4.931599 -0.505966 1.000000
)
      )
     (m-vtxRef
      11 12 13 14 25 26 27 28
      pool= Polyset
      )
     )
    )
   (joint
    name= “spine2”
    (transform
     matrix= (mat4x4
      1.000000 0.000000 0.000000 0.000000
      0.000000 1.000000 0.000000 0.000000
      0.000000 0.000000 1.000000 0.000000
      0.000000 1.485803 0.063246 1.000000
      )
     )
    (m-vtxRef
     29 30 31 32 33 36 37 44 46 47 73
     75
     pool= Polyset
     )
    (joint
     name= “spine3”
     (transform
      matrix= (mat4x4
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.491367 5.050761 -0.031623 1.000000
)
      )
     (m-vtxRef
      34 35 74 76
      pool= Polyset
      )
     (joint
      name= “rshoulder”
      (transform
matrix= (mat4x4
 1.000000 0.000000 0.000000 0.000000
 0.000000 1.000000 0.000000 0.000000
 0.000000 0.000000 1.000000 0.000000
 -0.000938 -0.031601 2.466585 1.000000
 )
)
      (m-vtxRef
40 41 42 43 65 66 67 68
pool= Polyset
)
      (joint
name= “relbow”
(transform
 matrix= (mat4x4
  1.000000 0.000000 0.000000 0.000000
  0.000000 1.000000 0.000000 0.000000
  0.000000 0.000000 1.000000 0.000000
  0.000000 0.000000 3.541763 1.000000
  )
 )
(m-vtxRef
 1 2 3 4 5 6 7 8 9 10 11
 12
 pool= Polyset#3
 )
(m-vtxRef
 61 62 63 64 85 86 87 88
 pool= Polyset
 )
)
      )
     (joint
      name= “lshoulder”
      (transform
matrix= (mat4x4
 1.000000 0.000000 0.000000 0.000000
 0.000000 1.000000 0.000000 0.000000
 0.000000 0.000000 1.000000 0.000000
 -0.000938 -0.031601 -2.434962 1.000000
 )
)
```

```
      (m-vtxRef
49 50 51 52 57 58 59 60
pool= Polyset
)
      (joint
name= “lelbow”
(transform
 matrix= (mat4x4
  1.000000 0.000000 0.000000 0.000000
  0.000000 1.000000 0.000000 0.000000
  0.000000 0.000000 1.000000 0.000000
  0.000938 0.031601 -3.573386 1.000000
  )
 )
(m-vtxRef
 1 2 3 4 5 6 7 8 9 10 11
 12
 pool= Polyset#2
 )
(m-vtxRef
 53 54 55 56 81 82 83 84
 pool= Polyset
 )
)
      )
     (joint
      name= “neck”
      (transform
matrix= (mat4x4
 1.000000 0.000000 0.000000 0.000000
 0.000000 1.000000 0.000000 0.000000
 0.000000 0.000000 1.000000 0.000000
 0.042202 1.421949 0.000000 1.000000
 )
)
      (m-vtxRef
69 70 71 72 77 78 79 80
pool= Polyset
)
      )
     )
    )
   )
  )

 ;;
 ;; Polygons
 ;;

 (p-group
  name= “Polyset#2”
  (ct-poly
   (rgba 0.903194 0.762680 0.458617 1.000000 )
   (m-vtxRef 4 3 2 1 pool= Polyset#2 )
   )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 3 6 5 2 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 9 8 7 4 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 4 7 6 3 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 8 11 10 7 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 7 10 5 6 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 12 1 10 11 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 1 2 5 10 pool= Polyset#2 )
 )
(ct-poly
 (rgba 0.903194 0.762680 0.458617 1.000000 )
 (m-vtxRef 9 4 1 12 pool= Polyset#2 )
 )
)
(p-group
 name= “Polyset#3”
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 4 3 2 1 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 3 6 5 2 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 1 9 8 7 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 2 5 9 1 pool= Polyset#3 )
  )
```

```
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 9 11 10 8 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 5 6 11 9 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 10 11 4 12 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 11 6 3 4 pool= Polyset#3 )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:urn1.rgb”)
  (m-vtxRef 12 4 1 7 pool= Polyset#3 )
  )
 )
(p-group
 name= “Polyset”
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 4 3 2 1 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 3 6 5 2 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 10 9 8 7 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 14 13 12 11 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 18 17 16 15 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 5 6 17 18 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 16 17 6 3 4 19 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 2 14 11 1 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 19 4 1 20 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 9 22 21 8 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 7 23 15 20 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 8 21 23 7 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 23 24 18 15 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 21 22 24 23 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 18 24 10 1 pool= Polyset )
  )
 (ct-poly
  (rgba 1.000000 1.000000 1.000000 1.000000 )
  (txt “stick:640X480.rgb”)
  (m-vtxRef 24 22 9 10 pool= Polyset )
  )
```

```
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 1 10 7 20 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 13 26 25 12 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 11 27 18 1 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 12 25 27 11 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 27 28 5 18 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 25 26 28 27 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 5 28 14 2 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 28 26 13 14 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 32 31 30 29 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 36 35 34 33 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 39 38 37 32 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 43 42 41 40 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 38 45 44 37 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 37 44 47 46 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 48 29 44 45 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 52 51 50 49 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 39 32 29 48 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 56 55 54 53 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 29 30 57 52 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 52 57 58 51 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 30 47 59 57 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 57 59 60 58 pool= Polyset )
 )
```

```
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 44 49 59 47 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 49 50 60 59 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 29 52 49 44 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 64 63 62 61 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 32 37 65 43 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 43 65 66 42 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 37 46 67 65 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 65 67 68 66 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 31 40 67 46 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 40 41 68 67 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 32 43 40 31 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 72 71 70 69 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 31 46 73 36 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 36 73 74 35 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 46 47 75 73 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 73 75 76 74 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 30 33 75 47 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 33 34 76 75 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 31 36 33 30 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 71 78 77 70 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 35 74 79 72 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
```

```
 (m-vtxRef 72 79 78 71 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 74 76 80 79 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 79 80 77 78 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 34 69 80 76 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 69 70 77 80 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 35 72 69 34 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 55 82 81 54 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 51 58 83 56 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 56 83 82 55 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 58 60 84 83 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 83 84 81 82 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 50 53 84 60 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 53 54 81 84 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 51 56 53 50 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 63 86 85 62 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 42 66 87 64 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 64 87 86 63 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 66 68 88 87 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 87 88 85 86 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 41 61 88 68 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 61 62 85 88 pool= Polyset )
 )
(ct-poly
 (rgba 1.000000 1.000000 1.000000 1.000000 )
 (txt “stick:640X480.rgb”)
 (m-vtxRef 42 64 61 41 pool= Polyset )
 )
 )
)
```

```
;;
;; Run-time structure
;;
;;> (ls stickman)
;;Stickman: <DCS>
;;   RootJoint: <DCS>
;;      spine1: <DCS>
;;         rhip: <DCS>
;;            rknee: <DCS>
;;         lhip: <DCS>
;;            lknee: <DCS>
;;         spine2: <DCS>
;;            spine3: <DCS>
;;               rshoulder: <DCS>
;;                  relbow: <DCS>
;;               lshoulder: <DCS>
;;                  lelbow: <DCS>
;;               neck: <DCS>
;;   Polyset:  <Morph>
;;   Polyset#3:  <Morph>
;;
```

8 Sample GUF File: stickman-chan.gif

This file animates the stickman using varying weights for geometric morphs and transformations.

```
;; stickman-chan.guf (GUF v.9)
;; Info--
;; Source File: stickman-morph
;; Created on:  Wed Feb 22 13:15:41 1995
;;         by:  alias2egg -A 8 35 -r 24 -vfmwx Stick-
man stickman-morph
;;

(table
 name= “stickman-morph”
 ;;
 ;; This animation nominally runs at 24 hz
 ;;
 (form-has s$-anim ‘( fps= 24 ) )

 (bundle
  name=“Stickman”
  (table morph
 4= (s$anim
    fps= 60
    0.929005 0.910140 0.889054 0.865745
0.840214
    0.812459 0.782481 0.750278 0.715851
0.679198
    0.640320 0.599216 0.555885 0.510327
0.462541
    0.412527 0.360285 0.305813 0.249112
0.190180
    0.129018 0.065625 0.000000 0.065625
0.129018
    0.190180 0.249112 0.305813
    )
 2= (s$anim
    0.278591 0.326998 0.372684 0.413925
0.448999
    0.476185 0.493759 0.500000 0.495704
0.483407
    0.464000 0.438370 0.407407 0.372000
0.333037
    0.291407 0.248000 0.203704 0.159407
0.116000
    0.074370 0.035407 0.000000 -0.035407 -
0.074370
    -0.116000 -0.159407 -0.203704
    )
 1= (s$anim
    fps= 60
    0.352000 0.425250 0.500000 0.574750
0.648000
    0.718250 0.784000 0.843750 0.896000
0.939250
    0.972000 0.992750 1.000000 0.994400
0.979200
    0.956800 0.929600 0.900000 0.870400
0.843200
    0.820800 0.805600 0.800000 0.805600
0.820800
    0.843200 0.870400 0.900000
    )
 3= (s$anim
    0.846949 0.940728 1.000000 0.973396
0.900964
    0.793772 0.662891 0.519389 0.374336
0.238802
    0.123855 0.040564 0.000000 0.010974
0.068345
    0.162133 0.282358 0.419040 0.562198
0.701854
    0.828026 0.930734 1.000000 1.011866
0.959000
    0.856353 0.718878 0.561528
    )
 ;; testpattern v_offset
 5= (s$anim
```

```
    0.000000
    )
 ;; purpleurn u_offset
 6= (s$anim
    0.000000
    )
 ;; purpleurn v_offset
 7= (s$anim
    0.250000
    )
 )
  (table
   name= “RootJoint”
   (table
    name= “spine1”
    xform= (xfm$anim-s$
    order= “sphrt”
    p= (s$anim 0.000000 )
    h= (s$anim
-5.000000 -5.000000 -5.000000 -5.000000 -
5.000000
-5.000000 -5.000000 -5.000000 -5.000000 -
5.000000
-5.000000 -5.000000 -5.000000 -5.000000 -
5.000000
-4.607822 -3.783414 -2.942845 -2.043612 -
1.089536
-0.107509 0.952867 2.041862 2.640695
3.175418
3.721582 4.275097 4.830310
)
    r= (s$anim 0.000000 )
    x= (s$anim
0.248904 0.302465 0.360530 0.422953 0.489590
0.560295 0.634921 0.713325 0.795361 0.880883
0.969746 1.061805 1.156913 1.254927 1.355700
1.459087 1.564943 1.673122 1.783480 1.895869
2.010146 2.126165 2.243781 2.362848 2.483220
2.604753 2.727300 2.850718
)
    y= (s$anim
9.976032 9.966717 9.956618 9.945763 9.934174
9.921877 9.908898 9.895263 9.880996 9.866122
9.850668 9.834658 9.818117 9.801071 9.783545
9.765565 9.747155 9.728342 9.709149 9.689603
9.669729 9.649551 9.629097 9.608390 9.587456
9.566319 9.545007 9.523542
)
    z= (s$anim -0.043357 )
    )
    (table
     name= “rhip”
     xform= (xfm$anim-s$
     order= “sphrt”
     p= (s$anim
 -1.523597 -1.542782 -1.559485 -1.573052 -
1.596908
 -1.620063 -1.637997 -1.661881 -1.679385 -
1.706109
 -1.730555 -1.756496 -1.780765 -1.806791 -
1.833449
 -1.661993 -1.289203 -0.911182 -0.514315 -
0.098536
 0.318942 0.755956 1.189645 1.379995 1.526316
 1.659842 1.776088 1.878180
 )
     h= (s$anim 0.000000 )
     r= (s$anim
 -18.343496 -18.359762 -18.364038 -18.349073 -
18.326811
 -18.281366 -18.216114 -18.130636 -18.021236 -
17.887325
 -17.724611 -17.532825 -17.314930 -17.059496 -
16.766092
 -16.502930 -16.357655 -16.139011 -15.895080 -
15.612192
 -15.277560 -14.914935 -14.496670 -13.896578 -
13.219555
 -12.477647 -11.668720 -10.770175
 )
     x= (s$anim 0.000000 )
     y= (s$anim -0.031612 )
     z= (s$anim 1.043555 )
     )
     (table
      name= “rknee”
      xform= (xfm$anim-s$
      order= “sphrt”
      p= (s$anim 0.000000 )
      h= (s$anim 0.000000 )
      r= (s$anim
  -32.304001 -32.272007 -32.245796 -32.232113
-32.218952
  -32.225239 -32.240345 -32.271233 -32.317402
-32.375412
  -32.454475 -32.552692 -32.672127 -32.816460
-32.985798
  -33.179615 -33.244534 -33.393238 -33.546055
-33.711102
  -33.923157 -34.121185 -34.363049 -34.757530
-35.209526
  -35.703876 -36.237640 -36.861679
  )
      x= (s$anim 0.000000 )
      y= (s$anim -4.994824 )
      z= (s$anim 0.537589 )
      )
     )
    (table
```

```
    name= “lhip”
    xform= (xfm$anim-s$
    order= “sphrt”
    p= (s$anim
 7.617012 7.619936 7.623580 7.631106 7.632080
 7.634013 7.639951 7.642865 7.648116 7.651617
 7.655164 7.658770 7.660182 7.661975 7.663314
 7.641713 7.596849 7.571074 7.559945 7.566250
 7.591846 7.638159 7.706904 7.801176 7.909174
 8.031247 8.167560 8.318274
 )
    h= (s$anim 0.000000 )
    r= (s$anim
 28.332661 28.069313 27.782797 27.436253
27.128870
 26.776413 26.391304 25.986361 25.555332
25.100582
 24.619764 24.117092 23.594872 23.043789
22.465258
 21.968834 21.623940 21.239628 20.863728
20.485325
 20.096973 19.725706 19.349688 18.801037
18.225519
 17.640720 17.050995 16.449024
 )
    x= (s$anim 0.000000 )
    y= (s$anim -0.063225 )
    z= (s$anim -0.917064 )
    )
    (table
     name= “lknee”
     xform= (xfm$anim-s$
     order= “sphrt”
     p= (s$anim 0.000000 )
     h= (s$anim 0.000000 )
     r= (s$anim
  -49.170937 -49.274048 -49.381676 -49.492794
-49.597561
  -49.719643 -49.833103 -49.941154 -50.045387
-50.145561
  -50.236568 -50.319210 -50.388172 -50.442451
-50.483807
  -50.487072 -50.549500 -50.554752 -50.562351
-50.564911
  -50.535816 -50.522396 -50.485184 -50.416153
-50.334702
  -50.242012 -50.145077 -50.015690
  )
     x= (s$anim 0.000000 )
     y= (s$anim -4.931599 )
     z= (s$anim -0.505966 )
     )
     )
    )
    (table
     name= “spine2”
     xform= (xfm$anim-s$
     order= “sphrt”
     p= (s$anim 0.000000 )
     h= (s$anim
  -4.739519 -4.571962 -4.388557 -4.276102 -
4.154028
  -3.953095 -3.727014 -3.492117 -3.236127 -
2.960482
  -2.666229 -2.359208 -2.043515 -1.705087 -
1.344553
  -1.356505 -1.782538 -2.206850 -2.676051 -
3.188410
  -3.716250 -4.313536 -4.928152 -5.000000 -
5.000000
  -5.000000 -5.000000 -5.000000
  )
     r= (s$anim -5.000000 )
     x= (s$anim 0.000000 )
     y= (s$anim 1.485803 )
     z= (s$anim 0.063246 )
     )
     (table
      name= “spine3”
      xform= (xfm$anim-s$
      order= “sphrt”
      p= (s$anim -5.000000 )
      h= (s$anim -5.000000 )
      r= (s$anim
   -3.732379 -3.336253 -2.944697 -4.464797 -
5.000000
   )
      x= (s$anim 0.491367 )
      y= (s$anim 5.050761 )
      z= (s$anim -0.031623 )
      )
      (table
       name= “rshoulder”
       xform= (xfm$anim-s$
       order= “sphrt”
       p= (s$anim 35.000000 )
       h= (s$anim 10.000000 )
       r= (s$anim
    -15.034025 -15.331027 -15.610883 -14.040801
-13.397036
    -13.230670 -13.039773 -12.818527 -12.571033
-12.281788
    -11.961925 -11.594760 -11.196093 -10.743291
-10.237941
    -9.699205 -9.123150 -8.477436 -7.769794 -
6.989098
    -6.121908 -5.174033 -4.126870 -2.968911 -
1.696448
```

```
  -0.291246 1.255142 2.940875
  )
     x= (s$anim -0.000938 )
     y= (s$anim -0.031601 )
     z= (s$anim 2.466585 )
     )
     (table
name= “relbow”
xform= (xfm$anim-s$
order= “sphrt”
p= (s$anim 0.000000 )
h= (s$anim
   78.859573 78.808548 78.781532 78.851692
78.881157
   78.877037 78.864265 78.858955 78.850548
78.826508
   78.811569 78.791504 78.777191 78.749817
78.722496
   78.665100 78.633308 78.557358 78.477692
78.383987
   78.261909 78.126579 77.959740 77.739990
77.493706
   77.199074 76.862633 76.469879
   )
r= (s$anim 0.000000 )
x= (s$anim 0.000000 )
y= (s$anim 0.000000 )
z= (s$anim 3.541763 )
)
)
      )
     (table
      name= “lshoulder”
      xform= (xfm$anim-s$
      order= “sphrt”
      p= (s$anim -35.000000 )
      h= (s$anim -10.000000 )
      r= (s$anim
  -11.403259 -12.118526 -12.854723 -11.558334
-11.320373
  -11.679234 -12.087089 -12.488034 -12.916216
-13.378163
  -13.848054 -14.333160 -14.825439 -15.336758
-15.866801
  -16.409222 -16.929785 -17.469416 -18.024492
-18.590534
  -19.165464 -19.752264 -20.348715 -20.941864
-21.539541
  -22.148142 -22.768255 -23.382957
  )
     x= (s$anim -0.000938 )
     y= (s$anim -0.031601 )
     z= (s$anim -2.434962 )
     )
     (table
name= “lelbow”
xform= (xfm$anim-s$
order= “sphrt”
p= (s$anim 0.000000 )
h= (s$anim
   -51.151478 -51.393791 -51.666893 -
51.775043 -51.983051
  -52.233376 -52.539711 -52.829578 -53.135273
-53.486210
  -53.819321 -54.186211 -54.539146 -54.904430
-55.277077
  -55.677170 -56.011391 -56.368813 -56.732471
-57.094101
   -57.447613 -57.801929 -58.148254 -
58.527699 -58.896053
   -59.262192 -59.622498 -59.953053
   )
r= (s$anim 0.000000 )
x= (s$anim 0.000938 )
y= (s$anim 0.031601 )
z= (s$anim -3.573386 )
)
)
      )
     (table
      name= “neck”
      xform= (xfm$anim-s$
      order= “sphrt”
      p= (s$anim
  -4.594393 -4.363284 -4.070470 -3.635263 -
3.286030
  -2.952685 -2.740646 -2.421130 -2.108602 -
1.920341
  -1.647288 -1.376064 -1.177094 -0.960684 -
0.764448
  -0.550226 -0.055481 0.313736 0.681619
1.043723
  1.334680 1.662633 1.934788 2.031734
2.089536
  2.087597 2.037874 1.985049
  )
     h= (s$anim -20.000000 )
     r= (s$anim
  -20.000000 -20.000000 -20.000000 -16.776329
-15.049335
  -13.994863 -12.834098 -11.659429 -10.433983
-9.127547
  -7.814590 -6.483043 -5.089135 -3.696455 -
2.289534
  -0.875278 0.393568 1.706650 2.995140
4.255712
  5.505949 6.700747 7.869171 9.043299
10.184815
  11.292832 12.374558 13.365005
  )
```

```
      x= (s$anim 0.042202 )
      y= (s$anim 1.421949 )
      z= (s$anim 0.000000 )
      )
      )
     )
    )
   )
  )
 )
)

;;
;; Run-time structure
;;
;;> (lsr stick-bundle)
;;stick-bundle
;;  RootJoint
;;    spine1
;;      lhip
;;        lknee
;;      rhip
;;        rknee
;;      spine2
;;        spine3
;;          rshoulder
;;            relbow
;;          neck
;;          lshoulder
;;            lelbow
;;  morph
>
```

Creating Compelling Real-Time Content

Michael Limber
Angel Studios

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

Course Concept

The ever expanding universe of computer graphics and processing technology has spawned extraordinary new markets in real-time interactive experiences. As the promise of virtual reality begins to come true, many realize that new skills are required to take advantage of these emerging technologies. At this point in the evolution of the field groups of artists and scientists are required to generate compelling virtual experiences. The structure of these groups is unique, and its collaborative success depends on the careful integration of computer technology and creative content.

1 Introduction

Human beings, among all the animals, possess a most unique form of consciousness. Information about the world is provided by our senses in the form of perceptions. These perceptions are integrated into concepts by our higher brain functions to produce the phenomenon of human thought and consciousness. These faculties serve not only to guide us through reality, but also enable us to recombine concepts into new ones, using retained information to form the building blocks for new ideas. The expression of these ideas, and the actions and feelings resulting from them, is essential to the evolution of human communication and knowledge.

Language is a vital form of communication, requiring the systematic use of our conceptual faculties. Written language offers a very efficient way of storing and transferring knowledge. A few specific marks on a page can convey complex thoughts and emotions that completely consume the reader. This ability to transport the mind to another time and context is as unique as the desire itself. All of the world's art forms are the re-creation of some aspect of reality according to the artist's internal view of the world.

The advent of photography, film and recorded sound allowed an unprecedented new range of expression. Reality could be recorded, then edited and rearranged to present a synthetic but very believable series of events. Theater and cinema are really very early forms of virtual reality. The audience sits in a darkened room where, for a while, they are mentally transported to another dimension. The written word hasn't been the same since.

Film and television, however powerful, are generally confined to the bounds of "real" reality. Various special effects are used to represent ideas and events that are intangible or impossible. The advent of three dimensional computer graphics set an undeniable precedence in the ability to portray fantastic yet realistic forms. Finally one could travel to the ends of the galaxy or peruse the invisible workings of the molecule. This is a medium where imagination can take form, and project the appearance and authority of truly existing. But traditional computer animation is still a passive experience. Viewers watch and listen while their bodies lay motionless. Thoughts and feelings can be evoked in this way, but a true virtual reality experience involves full interaction.

With the advent of virtual reality we are now poised to create experiences that completely immerse the senses -- all of them. Sight, sound, touch, smell and even taste can be manipulated synthetically to produce a transcendental effect close to dream-like. This new real-time interactive technology should prove to be a very dramatic form of expression. Entwined in media hyperbole, enshrined by computer age philosophers, and dreaded by suspicious technophobes, virtual reality promises to change the way everything from science to art is expressed and perceived. The open question for this potent new medium is the same basic question for all forms of expression -- what content shall this form express? It is a crucial question to ask.

To this point complex computer graphics has required the skills of numerous scientists and programmers. Elaborate computer generated scenes involve painstaking communication to the computer at a very low level. For this reason (coupled with the speed of the available hardware) significant synthetic imagery is rarely seen, and then usually only for brief moments of graphic spectacle. The reputation of computer graphics' visual impact and style has overshadowed concerns about its creative content and thematic substance. Until recently, the medium has been the message.

This view is changing. Computers offer a unique opportunity for the creative and technical side of human nature to collaborate. New advances in hardware performance and object-oriented programming are giving us a first glimpse at the shores of a new moldable frontier. We can shape our new worlds any way we want. The shape we choose will reflect on us and our values.

This course will attempt to show the importance of taking a Renaissance approach to the development of real-time interactive experiences; integrating the skills of not only scientists, technicians, and engineers but also writers, designers, and artists. After establishing a brief historical context we will examine a model for developing, creating, and producing compelling virtual content.

2 Historical Background

Much of the initial development of 3D computer graphic science, back in the mid 1960's, was for real-time military flight simulators. The cost of training fighter pilots in real planes far exceeded the cost of developing a synthetic approach. The esoteric nature of this new technology gained popularity in the sciences where the field of Scientific Visualization emerged. Complex intangible phenomena could be made visible to the naked eye. Since the computational cost of real-time simulation was high, computer animation technology developed enabling high-resolution, frame by frame output of lengthy visualizations.

The photo-realistic appearance of this visual technology was soon apparent to the commercial community. By the late 1970's broadcast ID's, corporate logos, and the occasional commercial and film effect debuted a new art form to large audiences. By the late 1980's, computer graphics was not only a familiar workhorse of the broadcast television market, but was finally embraced by the cinema as a viable tool. Concurrently millions of people were exposed to interactive experiences through personal computers and cartridge based video games.

Since that time the entertainment market has become a major force in the development of graphics technology, including virtual reality. The popularity of books by authors like William Gibson and science fiction films like "The Lawnmower Man" have helped fuel a growing VR fever. Largely fiction until now, the potential of virtual reality is about to be actualized. The gaming industry has already embraced it. The venture capital community, which waited out the initial media storm, is beginning to investigate the economic potential of the fledgling technology. The growth curve of development and interest appears to be exponential.

3 New Opportunities

Film and Television. These new computer technologies are creating opportunities in film and television. Automated animation techniques will make long-form computer generated projects feasible. We have all seen the recent big screen computer graphics achievements in films like “Jurassic Park”. Faster hardware and increased software control will soon give computer generated effects longer and longer screen times, and more significant characters will be digitally generated. There are already at least three fully computer generated films in production, and several weekly TV series are using extensive digital scenery in many shots. In the near future many shows will be entirely produced on the computer, including all characters and action. The real revolution will come when these shows are produced in real-time.

Video Arcades. Traditional video arcades are going to make a most dramatic change in the near future. New 3D interactive rides and experiences will soon replace most of the now popular 2D sprite games. Affordable motion-base units, high resolution head mounted displays, and a wide range of 3D real-time interactive games will change the small standing room only video parlors into large interactive gaming centers. Many owners are teaming up with mall and theater operators to widen the appeal of their locations. The large video game companies are using these arcades as test beds for new approaches to 3D gaming.

Consumer Platforms. Current consumer video game profits surpass those of feature film box-office intake; it topped $15 billion in 1994. The market for the new real-time 3D products will easily surpass that. The teenaged gamers of 10 years ago have grown up--and they’re still playing games. Only their tastes have matured. All of the major consumer game are currently manufacturing new 32-bit platforms that boast 3D capability. Some have collaborated with real-time hardware makers to produce the next generation machines. Developers are rushing to learn the new skills required to exploit the new medium. The fruits of their labors are already becoming abundantly apparent on games store shelves.

Location Based Entertainment. Dynamic location-based-entertainment (LBE) events are slowly displacing costly amusement park iron rides. Eventually, networked full-immersion simulation centers will replace most traditional centralized theme parks and arcades. The real-time software experiences can be updated and changed frequently, giving a much longer appeal to the locations. Because most of the motion-base systems are relatively small, as compared

with roller-coasters, simulation centers can be set up almost anywhere. If fact, many portable systems exist. Advanced versions may soon radically change the shape of county fairs and sports events. From Las Vegas casinos and big theme parks, to shopping malls and ocean cruise lines, operators are finding that virtual reality is drawing more than just teenage interest. The market's demographic age range is rapidly widening.

4 New Technology Developments

Computer Hardware. New developments in hardware are already raising the standards and expectations for computer generated images and experiences. Personal computer manufacturers are consistently doubling chip speeds each year. Some PCs, with add-on processing hardware, are approaching the performance of some high-end workstations -- at a fraction of the cost. Game companies are about to release new 32 and 64-bit hardware with moderate 3D capacity, around 500-1500 un-antialiased un-textured polygons per frame. The cost is somewhere between $200 to $500. The high-end manufacturers are also quickly raising performance. Some real-time image generator hardware is currently capable of generating elaborate real-time interactive scenarios -- about 3000-5000 anti-aliased textured polygons per frame running at 60 Hz. In 1994 these machines used to cost between $100,000 to $250,000. In 1995 that same power is available in the $25,000 to $50,000 range.

Input Devices. A wide range of new input devices are allowing for unprecedented data capture. From 3D laser scanners and 3 space digitizing devices, to full body motion capture systems and facial expression sensing units, data from the real world can be captured and used to develop the new digital "unreal" estate.

Output Devices. Like input devices, output devices are multiplying in complexity and interactive function. There are many potential uses of reliable high resolution head mounted displays. From training simulators and medical applications, to CAD workstations and real-time consumer gaming platforms, these new output devices allow the user full immersion. New light weight designs do away with screens altogether and project images directly onto the retina of the eye. Also, varieties of motion base units and force feedback devices are multiplying in number, providing convincing synchronized physical responses to virtual events.

Networks. Probably one of the most significant developments in the advent of real-time interactivity is the establishment of large scale fiber-optic networks. Many international governments and private companies are committed to developing a fiber-optic infrastructure. With its superior speed and band-width

it will surely uncork the bottle on sophisticated real-time interaction and communication.

Applications. Simulation and Artificial Intelligence sciences are maturing rapidly. Complex physical dynamics and kinematics software libraries are growing. The ability to compute the effects of gravity and wind, and the behavior of fabric and skin is opening the door on elaborate dynamic forms, all running in real-time. Real-time particle systems make commonplace the creation of believable flocks of birds, smoke and dust, and meandering bubbles. New developments in automating distinct human and animal motion and behavior are promising to populate virtual reality with believable counterparts.

5 The Necessary Skills

For obvious reasons the successful control of this new medium requires a wide range of specialized skills. Knowledge of content, form, reality and "virtuality" must be extensive. Creative groups must have experience creating novel concepts and fiction, designing varieties of detailed artwork, building elaborate 3D models and environments, and creating evocative character expression and behavior. Programmers need to be familiar with everything from real-time 3D simulation and complex physical dynamics, to intricate multi-player interactivity and numerous device interfaces and networks. The learning curve for these skills is understandably long, but when combined, they can be used to assemble expansive synthetic worlds and experiences unlike the general public has ever imagined. Within the next few years, nearly everyone will have experienced them first hand.

Certain companies already possess many of the necessary talents and can take advantage of this window of opportunity. And many of them have begun to make remarkable inroads. Looking for commercial applications of military technology, the defense industry simulation companies are beginning to explore the entertainment market and develop 3D interactive concepts. They are very familiar with 3D graphics and real-time simulation, but have some things to learn about the creative requirements of entertainment. The large video game companies are also upgrading their technical skills. They are familiar with creating content and 2D artwork for loads of interactive entertainment on small computers, but the tasks involved in dealing with large quantities of dynamic 3D data, behavior and technology are a new discipline.

There is another type of company well poised to take advantage of this new technology -- the traditional high-end computer graphics facilities. The more experienced studios have had to rely on strong in-house software development teams to produce work of any lasting significance. Their software departments

are having to constantly invent more efficient techniques of dealing with high-resolution data and complex scene descriptions. High commercial production values have instilled their creative departments with consistent yet flexible talents, and many are large enough to produce intricate, fully digital, long-form projects.

But, market pressures are tough. The large capital investment in equipment required along with the expensive talents of the individuals needed, make the task almost prohibitive. High resolution computer graphics production is mostly service work. Companies rarely have any equity interest in the projects they produce; a marginal income and a good reputation are signs of success. If these types of organizations can make the leap to developing their own content, many may enjoy fiscal success as well.

Aside from the necessity of getting the creative team to develop proprietary content, it is essential for the software department to make some critical changes as well. They must build a robust library of diverse simulation software. In general, real-time and physical dynamics programming is a natural progression toward speeding up and automating otherwise painstaking animation production. But it adds the complexity of generating and keeping track of dynamic interactions between entities in the virtual world. This leap into object-oriented real-time programming and artificial intelligence is essential to be competitive in the new interactive markets.

6 Real-Time Content Production

With few exceptions, the production of real-time content follows a similar pattern as the production of more conventional 3D computer graphics. A fundamental difference at this early stage of the industry is the necessity for in-house development of the creative content as well as the necessary 3D real-time techniques. Companies must not only invent the technology, but because of its unique nature, they must also invent creative applications for it. Because of the high expenses, projects tend to be driven by commercial demands. In general, most production companies are hired to solve someone else's creative problems. Initiating an original idea for a virtual experience, a real-time game, or an interactive fiction is a new concept for many otherwise experienced production groups.

So, who will develop these creative properties? How will they do it?. The following sections outline a production model for real-time virtual environments and interactive content. The background for this information comes from my experience as Chief Operating Officer and Head of Production for Angel Studios of Carlsbad, California. Beat know for its work on "The Lawnmower

Man:, and Peter Gabriel's "Kiss that Frog" Music Video, Angels Studios is now exclusively developing content for 3D real-time interactive entertainment.

7 Human Resource Breakdown

The most vital asset of any organization is the synergy created by its individuals. Getting the entire team to collaborate effectively is the key to creating content of lasting quality. The following is a list of the major team members involved in interactive content production. Some organizations may use different titles for individuals who perform similar functions.

Although they will not be covered here, every production team has a capable group of colleagues supporting the other aspects of development. The vital importance of everyone from the sales, marketing, and public relations groups, to the business administrators, operations managers and production assistants should not be overlooked. All are critical to the successful creation of a quality real-time interactive product.

The Management Team

Producer. The Producer is the overseer and integrator of all aspects of a project. From initial bids and final budgets, to personnel and resource scheduling, the Producer serves as the tip of the production pyramid. They are the bridge between the creative, the production, and the software teams. If developing content for an outside client, the Producer is also the liaison for all interactions with the customer. When developing content in-house they serve as a link between the content distributor and the production department. The Producer also plays a major role in selecting the team members for any given project, and in generating a complete and realistic schedule for the allocation of resources and the timely execution of the work. As companies get larger and the projects increase in number and scope, the Producer may have several Assistant Producers working with them to distribute the work. The Producer is one of the most vital members of any successful development team.

Creative Director. Working closely with the Producer and other Project Directors, the Creative Director manages the creative team and helps define a consistent aesthetic approach for each interactive project. From concept formation and content selection, through progressive refinements and final interactive testing, the Creative Director is the aesthetic cornerstone of the interactive production studio.

Production Director. The Production Director supervises all in-house production. Responsibilities include deciding which production approach to take, ensuring standards and specifications for each part of a project, and coping

with the inevitable unexpected production dilemma. They work closely with creative and software teams throughout a project, assuring the smooth progress of each stage. Production Directors need to be experienced Technical Directors themselves to supervise the overlapping production of diverse projects.

Software Director. The Software Director defines the entire direction of the company's programming technology. Real-time applications are pushing the limits of available computer technology, and pioneering virtual experiences require revolutionary software techniques to progress. The Software Director determines what aspects of an application to emphasize for each project, and which programmers should preform what tasks.

The Creative Team

Game Designer. Since most current real-time development revolves around interactive entertainment, another essential team member is the Game Designer. Serving as the creative team leader, the Game Designer usually comes up with the initial creative concept. All aspects of game design, from concept and game-logic, to artwork and game-play, come under the guiding hand of the Game Designer. Generally creating the fiction for the experience, they work closely with the Art Director and Production Artists to outline and design the project.

Art Director. Once the concept has been outlined the Art Director steps in to visualize the idea. Concept sketches are produced and the creative team uses them to stimulate further dialogue and development. They are responsible for all of the project's aesthetic concerns, from start to finish, and ensure continuity and consistency for all artwork. A talented Art Director can make an interactive experience both visually exciting and creatively compelling.

Production Artist. Once a project is in production there is generally much artwork to be generated. Included in the Production Artist's duties are everything from complete maps of virtual worlds and object construction drafts, to model sculptures and artwork for texture maps. Supervised by the Game Designer and the Art Director, the Production Artist brings to full life all of the previous conceptual work, and provides the production material for the look and feel of the real-time experience.

The Production Team

Technical Director. The Technical Director must integrate all of the different aspects of a production team's effort into a cohesive whole ready for the proposed application. From organizing models and setting up pre-computed sequences, to tuning the application and managing varieties of data, the TD organizes, produces and refines the project. The Software team members work

closely with the Technical Director to make sure real-time experiences are as engaging and attractive as traditional passive events, like television and film.

Modeler. All virtual worlds require extensive model building. Modelers take design concept sketches and storyboards, and create 3D digital models using a variety of sophisticated construction tools. Some objects are static and fairly straightforward to make. Others, like detailed dynamic characters and elaborate animated devices, require similar talents as that of an architect or sculptor. Making models for real-time applications requires creating well proportioned low detail objects. Making 50,000 polygons look good isn't difficult. Making 500 look good is.

Sound Designer. A most important aspect of any real-time experience is a convincing sound design. The Sound Designer uses a wide range of tools to create interactive theme music, sampled voice-overs, and holophonicly placed sound effects. Otherwise well done projects may fail because of poor sound design. A good Sound Designer's work tends to integrate the entire project, completing the effect, and permitting the audience to truly immerse themselves.

The Software Team

Application Programmer. Each application tends to build off an existing library of solutions. Since each project is also unique the Application programmer is in charge of developing and maintaining the structure and content of each real-time program. He or she integrates the efforts of the rest of the software and production teams, and assembles a single piece of executable code that comprises the entire interactive experience.

Simulation Programmer. So essential for a convincing simulation, the Simulation and Dynamics Programmer is the physics and behavior expert on the team. This person needs to have strong skills in physics, mathematics and dynamics and be able to combine these disciplines effectively. Real-time inverse and forward kinematics, particle dynamics, collision detection, and character automation are all concerns of the Simulation Programmer.

Interface Programmer. The final success of the project is assured if the human interface is intuitive and effortless. The work of all the other team members can be lost if the interaction interface is clumsy and complex. From on-screen information and selection menus, to sequence scheduling and user input processing, the Interface Programmer is responsible for maintaining the quality of the interaction between man and machine.

Device Programmer. All interactive experiences require interfacing with numerous devices and accessories. From joysticks, data-gloves, and digital audio equipment, to head mounted displays, body trackers, and 6 degree-of-freedom motion bases, the Device Programmer is an invaluable player in inte-

grating physical devices with the interactive application and producing the virtual reality effect. Since interactivity between users is an essential component of VR, the Device Programmer also handles all networking tasks.

8 The Software Application

Equally as important as the development the creative content is the development of the technical content. Advanced computer programming talent is necessary to exploit the creative potential of powerful new computers. The software application, more than ever, executes control over the description and behavior of a virtual scene. Since virtual worlds by their nature need to have autonomous aspects, the software used in building and managing these real-time environments must have a sophistication well beyond that which is currently commercially available. Developing robust in-house software code and libraries is essential for breaking new ground and limits.

For the now traditional method of producing computer graphics there are a number of fairly complete third party software solutions available. Offering a high level of scene management and control with numerous sophisticated effects and procedures, these programs make the creation of high resolution imagery more accessible than ever before. But many of the computationally expensive software techniques in these packages are not well suited to high speed real-time software requirements. It will be a number of years before we see complete and integrated off-the-shelf real-time packages.

The following is a specification outline for Angel Studios Advanced Real-Time Simulation software (ARTS). It is by no means all inclusive, but it is a good presentation of the main features and techniques necessary for the production of real-time interactive content. Its content represents thousands of man hours of experimentation and exploration. Angel Studios believes it is a minimum specification for dealing with the complex problems associated with creating real-time interactive content.

Angel Studios Arts

ARTS is a collection of high performance tools for developing state-of-the-art, real-time, 3D interactive experiences. The system is built on a highly optimized library of C++ programs, which form the basis of a complete simulation package.

Main Features. ARTS is written entirely in optimized C++, it is designed and implemented to take advantage of today's high speed graphics platforms. ARTS supports a variety of hardware, from 32-bit home systems and PCs to high performance graphics engines. It has transparent support for commercial

photo-realistic rendering packages, as well as a complete compliment of general data processing tools, along with low level vector, matrix and math operations.

Scene Management. ARTS features a hierarchical scene description structure. It supports dynamic load management, multiple cameras and lights, custom and automatic culling, and level of detail features. ARTS takes advantage of new multi-processing technology for simulation, drawing, culling, and custom applications. It supports multiple viewports and hardware pipelines. A necessity for gaming scenarios and large production environments, ARTS has built-in distributed networking capabilities.

Graphics Features. Along with supporting a variety of rendering packages and techniques, ARTS has real-time options for generating radiosity lighting effects, edge on fading to simulate gaseous objects, and multi-pass rendering for high quality shading and special effects like real-time bump maps. It also features optimized display list processing for non-linear deformations and n-dimensional shape interpolation.

Dynamics Simulation. One of the most important features of any real-time system is how well it deals with physical dynamics of objects and entities in a virtual environment. ARTS is well equipped for this, with optimized forward and inverse kinematics (with integrated motion capture), hierarchical collision detection using surface geometry and/or primitives, and real-time non-linear spring-mass networks. It also supports multiple dynamic particle systems which can interact with global forces and other objects, efficient height above terrain and range finding, and eye path collision avoidance.

Behavioral Simulation. Populating virtual worlds with intelligent agents and entities is critical in developing compelling interactive experiences. Using an extensible object-oriented approach ARTS provides controls for creating varieties of autonomous friends and foes. In support of photo-realistic animations ARTS allows for imported keyframe data from third party animation packages, as well as a variety of motion capture systems, like the Polhemus Fast-Track and Ascension's Flock-of-Birds.

Database Generation. ARTS construction tools offers a full functioned point, line, polygon, and patch database editor, as well as an interface for object from third party products. The extensible nature of the ARTS environment allows for dynamic integration of newly developed advanced or specific modeling tools, with a generic graphical user interface builder. In this way, new software techniques can be integrated and released to the entire studio in a matter of minutes. ARTS also provides an extensive library of floating point image processing tools for image manipulation and texture generation and management.

Audio Features. ARTS recognizes the importance of sound in the virtual world, and it supports a full MIDI standard. 3D placement of localized sounds, ambient sounds, and virtual microphones, all help drive the illusion home. ARTS offers an efficient tool kit of digital sound editing and imaging tools, and is configurable to support a variety of hardware/software audio interfaces.

Hardware Interface. It is essential in this new field to be able to import and export data from a variety of input and output devices. Two to six degree of freedom motion bases, stereoscopic and head mounted displays, and full body motion tracking systems are supported with a customizable networked input event queue.

Extension Packages. Because of the extensible nature of the object oriented approach used in ARTS a number of specialized extension packages can be developed within the ARTS environment. Certainly a generic library of objects and environments is a must for efficient production, as well as a feature based environment generator. Many applications require extensive vehicle dynamics with elaborate weapons and explosion handling capabilities. Software libraries for synthetic actors and crowds can be added along with feature based expression editors with automated lip-sync support. All of these features can be used for high resolution photo-realistic animation production systems. Even efficient security encryption, production management and tracking can be integrated into the ARTS system.

These specifications and suggestions are an example of what is required to produce truly compelling interactive content. If the user is to get deeply involved in a simulation of a virtual world that depiction must be complete, consistent, and coherent. Advanced programming techniques must be used if one is to achieve a willing suspension of disbelief.

9 Conclusion

The preceding sections have attempted to demonstrate the full range of disciplines and techniques necessary to create compelling interactive content. A dynamic balance between strong creative and technical teams, a thriving research and development effort, and a genuine enthusiasm for the work, are the ingredients most likely to produce an engaging product.

For now the task is still fairly complicated and the industry is still in its infancy. As these techniques and procedures become more refined the depiction of more lush and fleshed out worlds will be possible. Not until the look and feel of the interactive experience is at least as good as the cover art for the product, and the content has the depth and appeal of some of our best art forms,

will true real-time interactive experiences fulfill the complete promise of virtual reality.

One day it may be possible for a creative designer, writer and director to sit together in a special room and develop interactive experiences merely by discussing and describing the content to a very sophisticated computer. Their words and ideas would be interpreted and turned into representations in real-time, as well as any changes and modifications they might have. A virtual reality "black box" of this kind would let the human designer operate at a high level, much like a film director currently directs actors. The idea of the "Holo-Deck" on the Starship Enterprise, in the Star Trek TV series, may not be so far fetched.

Most of the opportunities we have discussed pertain mostly to leisure and entertainment. That is only temporary. High costs have limited this technology's wide application to areas of high visibility and profit. As the price/performance ratio of the equipment and techniques involved improves, we will see more and more educational, medical and industrial uses developing. This is only the beginning in a new era of expression and communication.

No matter what the application, the development of real-time interactive technology will always require the combined efforts of artists and scientists. During the Renaissance, a driving curiosity about nature and a benevolent respect for the human mind created a fertile environment for new technological inventions and creative ideas. Many of those artistic developments and technological innovations are still admired today because of their universal appeal and their timeless qualities. Today, the emerging field of real-time interactive entertainment may be the surprising catalyst to once again make both the arts and sciences accessible, exciting, and meaningful to everyone.

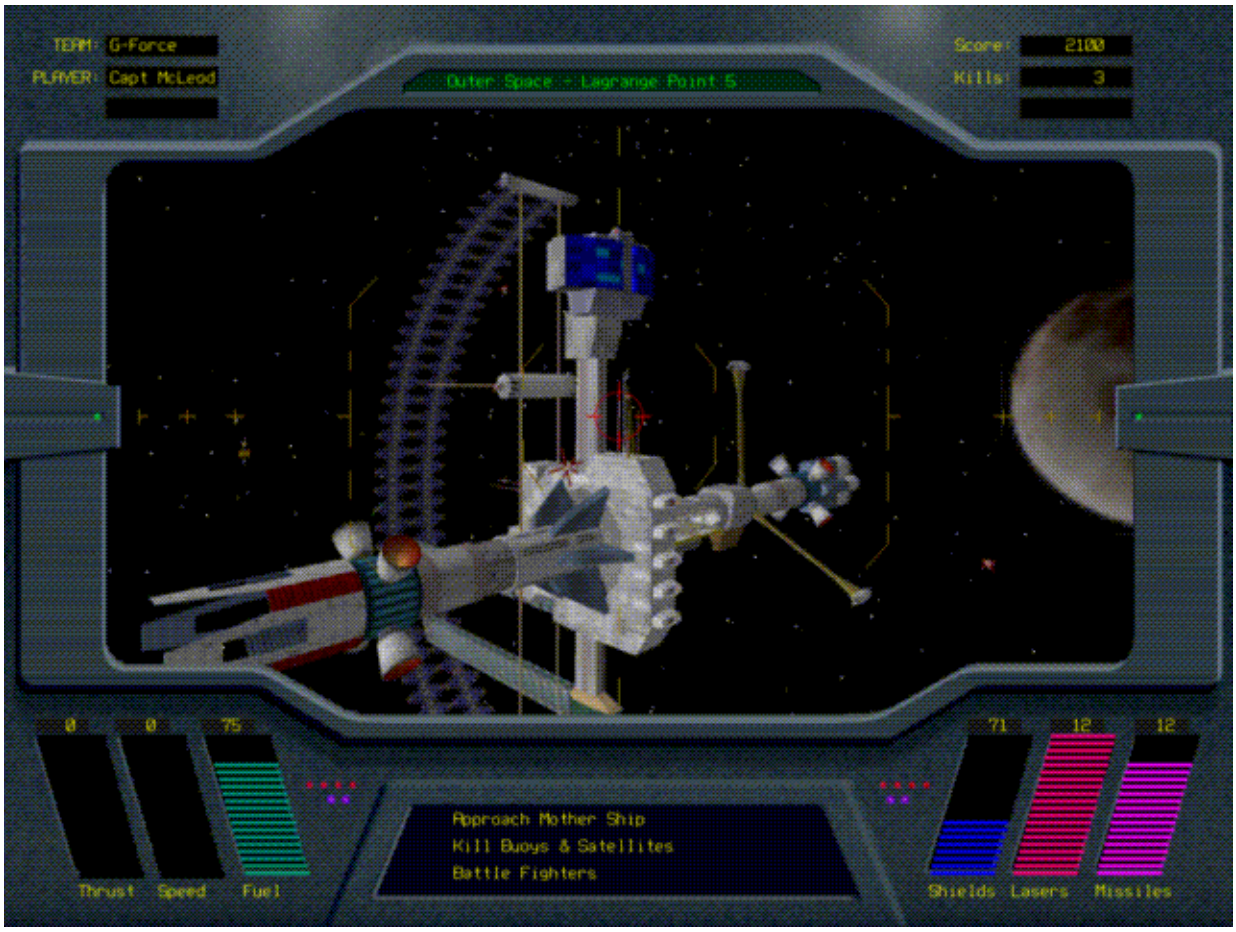


FIGURE 1. Orbit Defenders: mother ship exterior with control overly.

10 Figures

The images in this section are from two projects. Orbit Defenders and the Pteranadon demonstration ride. Angel produced Orbit Defenders using a Ball 944 Image Generator using ARTS™. The Pteranadon was produced in conjunction with GreyStone Technology on an Onyx RealityEngine using IRIS Performer.

FIGURE 2. Orbit Defenders: exterior battle sequence.

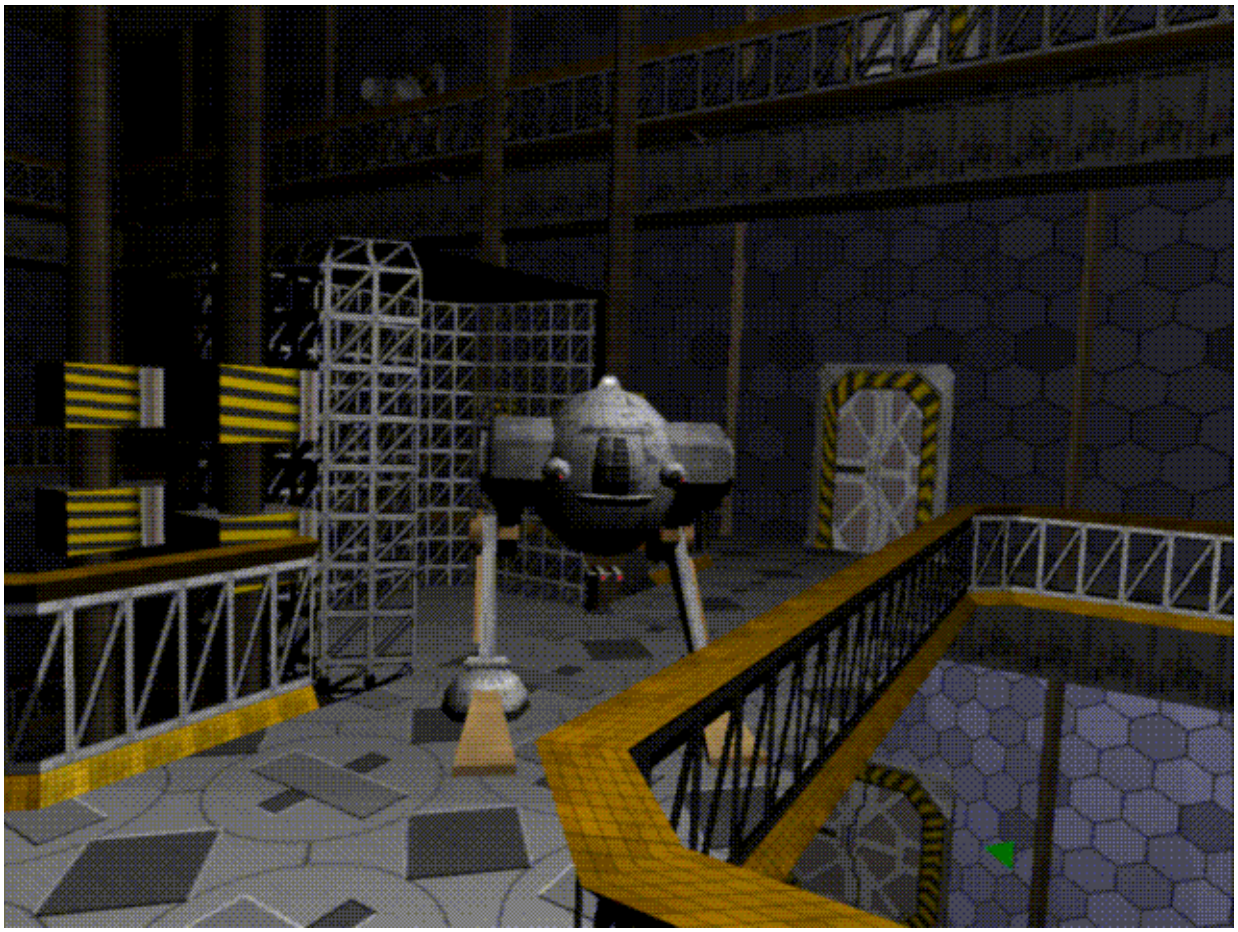


FIGURE 3. Orbit Defenders: interior intruder robot confrontation.

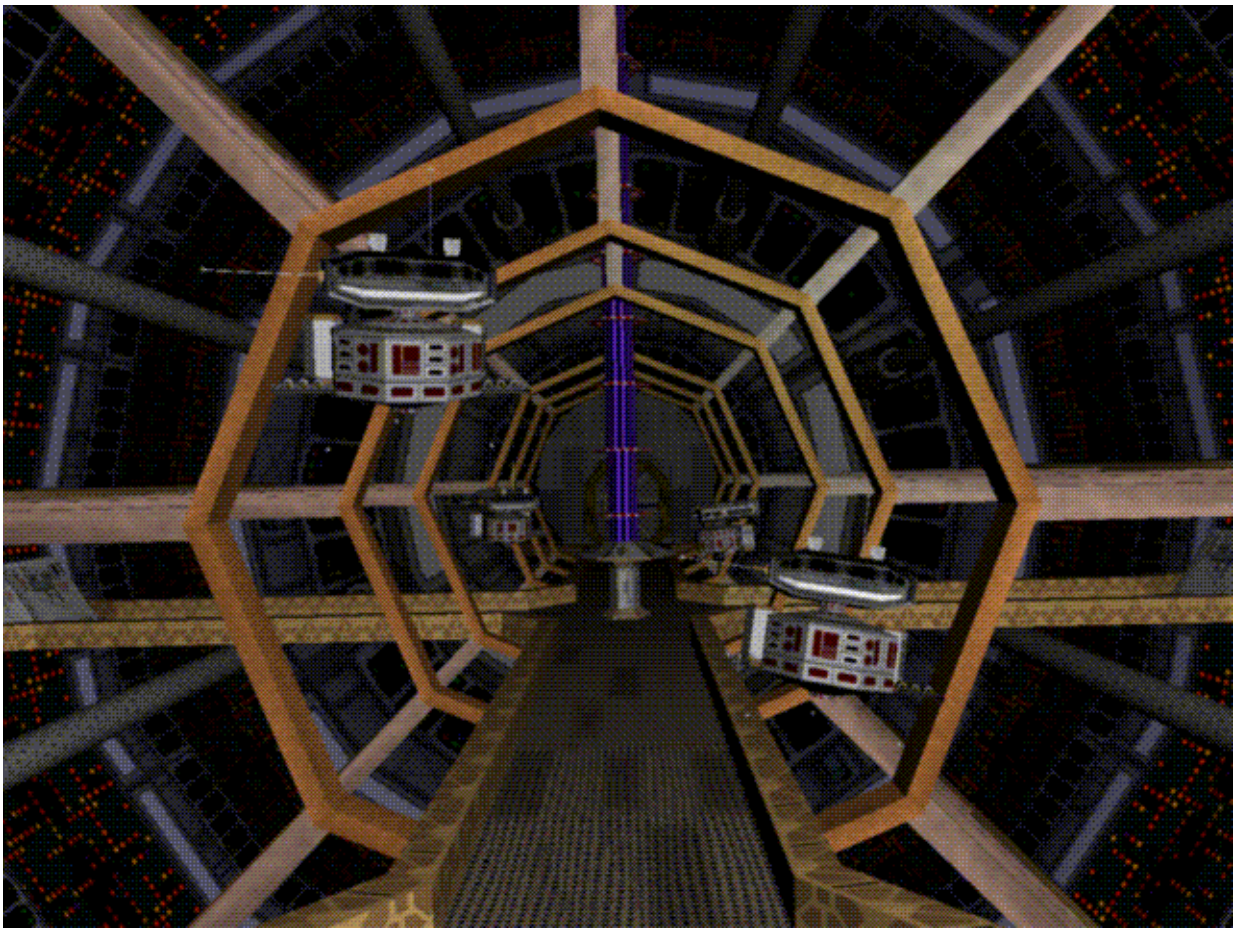


FIGURE 4. Orbit Defenders: computer room with defensive droids.

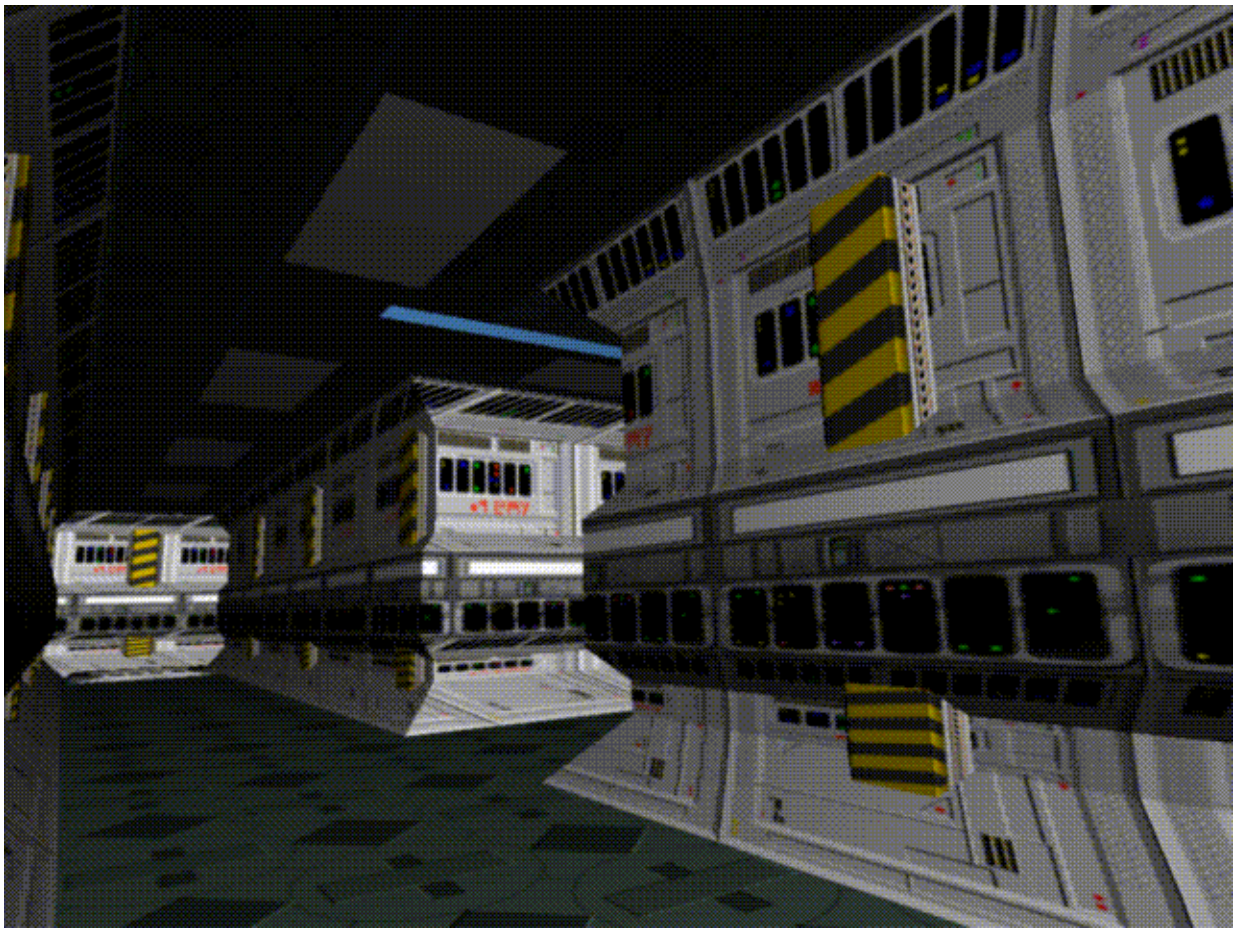


FIGURE 5. Orbit Defenders: interior corridor maze section.

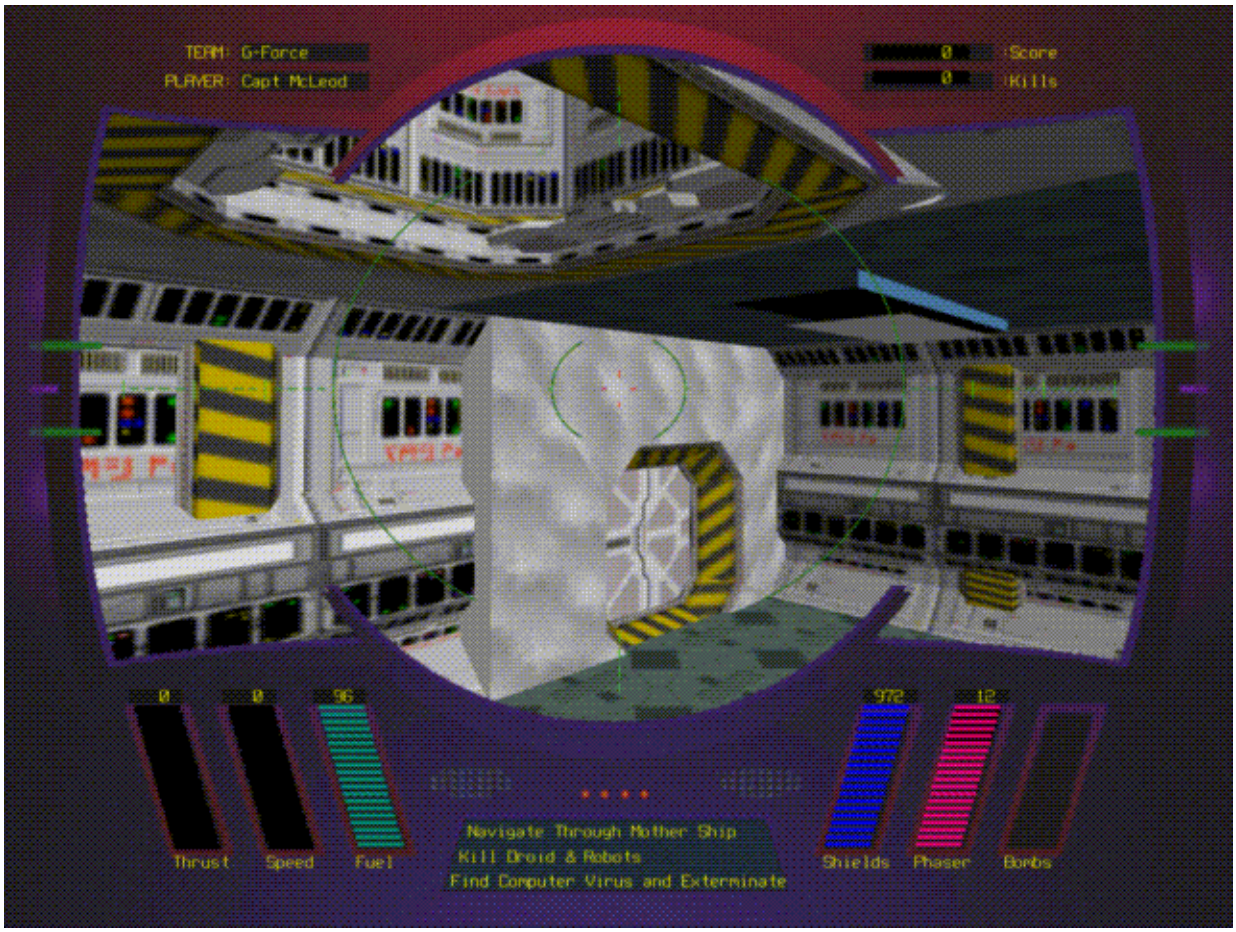


FIGURE 6. Orbit Defenders: maze airlocks with control overlay.

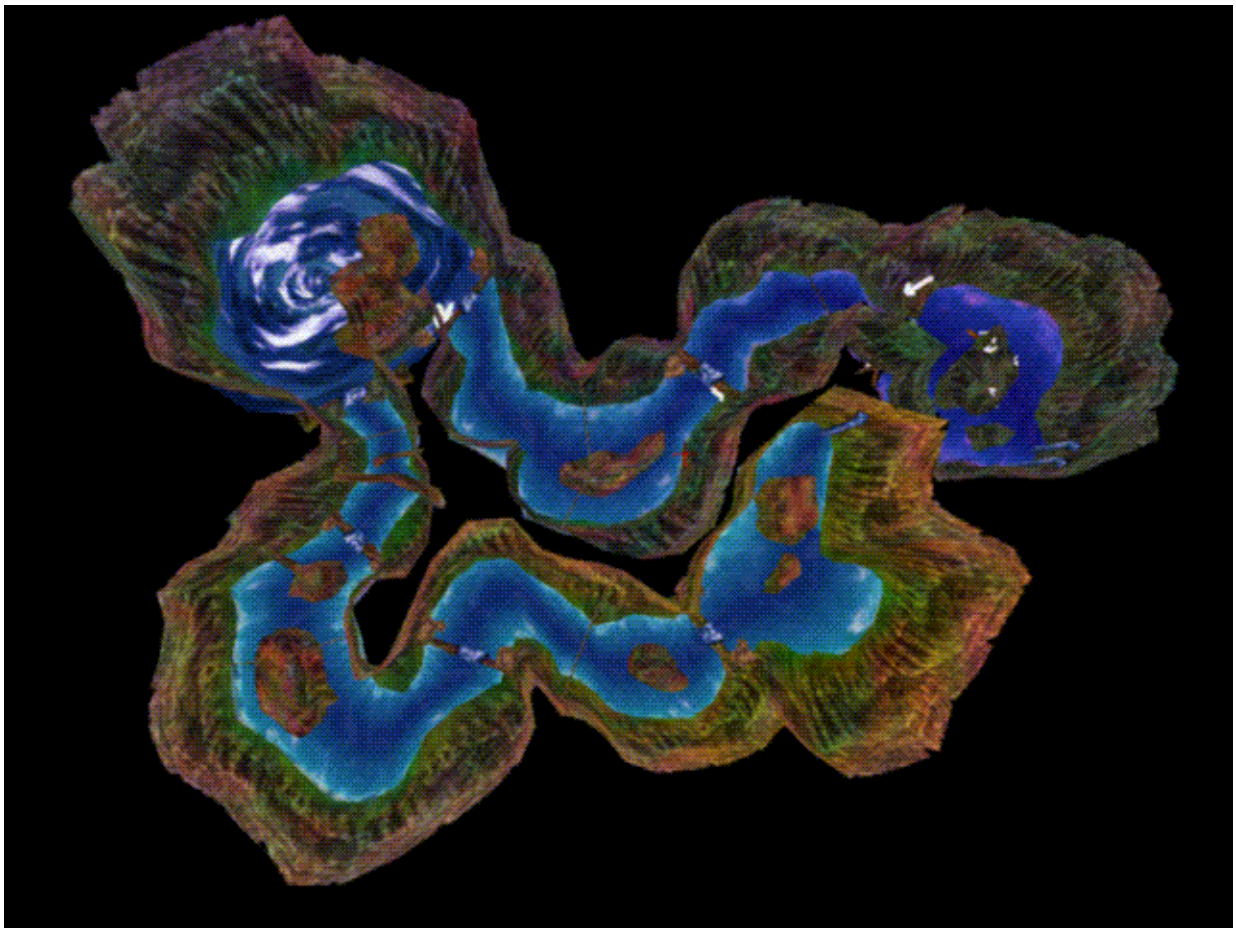


FIGURE 7. Plan view of the Pteranadon canyon environment.

FIGURE 8. Pteranadon flying a low pass over animated leviathan.

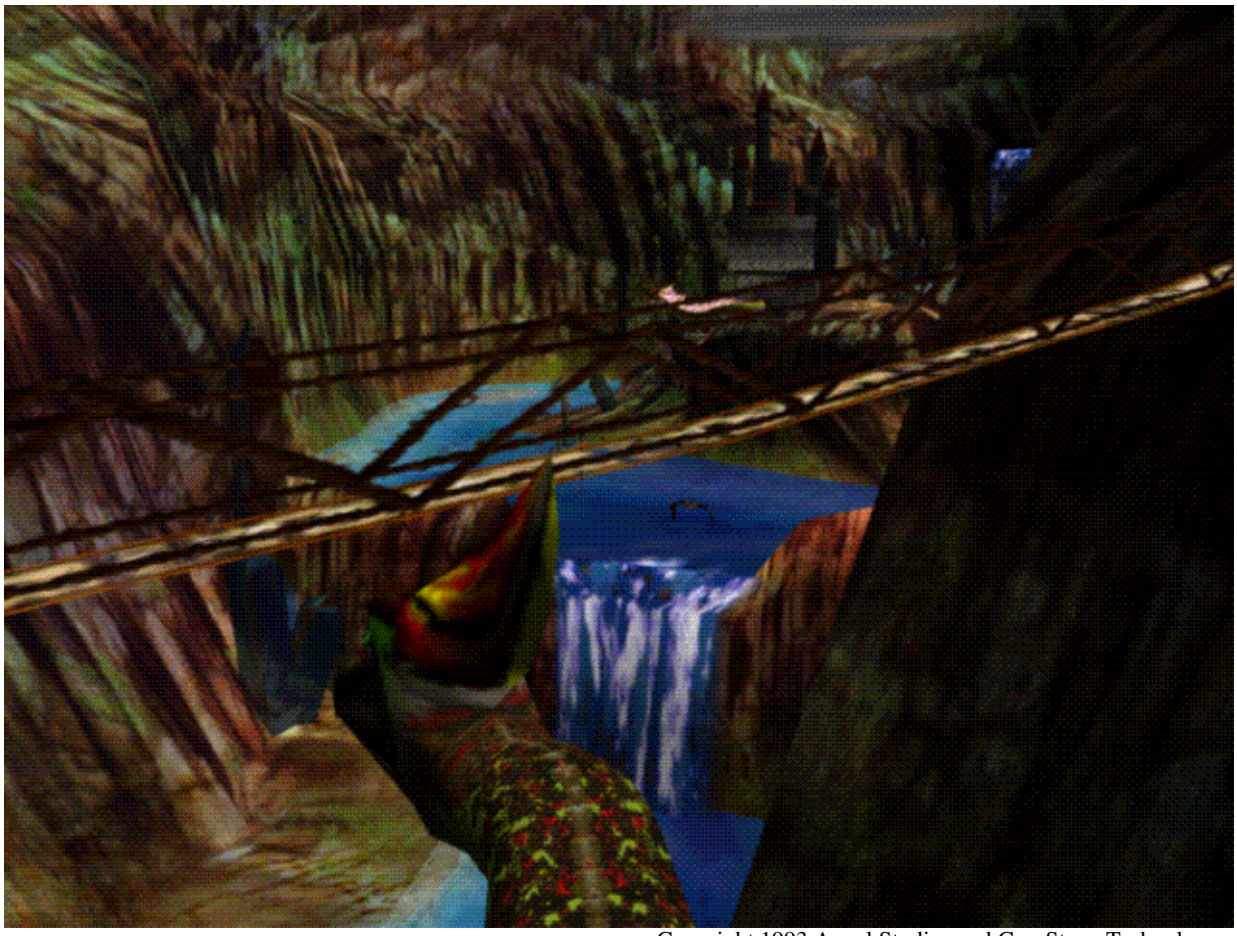


FIGURE 9. Pteranadon approaching castle with rope bridge in foreground.

Paper Reprint: IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3-D Graphics

John Rohlf and James Helman
Silicon Graphics Computer Systems

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '95 Course

This paper describes the principal design features of a real-time 3D graphics toolkit written by the authors of the first three chapters in these course notes.- The toolkit has been used in location-based entertainment (e.g. Magic Edge by Paradigm Simulation, Aladdin Virtual Reality Ride by Disney Imagineering) and in game prototyping and development for other platforms. The paper describes the methods IRIS Performer uses for multiprocessing, optimizing rendering and implementing visual simulation features on workstation-class machines.

Published: Proceedings SIGGRAPH '94

IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics

John Rohlf and James Helman
Silicon Graphics Computer Systems*

Abstract

This paper describes the design and implementation of IRIS Performer, a toolkit for visual simulation, virtual reality, and other real-time 3D graphics applications. The principal design goal is to allow application developers to more easily obtain maximal performance from 3D graphics workstations which feature multiple CPUs and support an immediate-mode rendering library. To this end, the toolkit combines a low-level library for high-performance rendering with a high-level library that implements pipelined, parallel traversals of a hierarchical scene graph. While discussing the toolkit architecture, the paper illuminates and addresses performance issues fundamental to immediate-mode graphics and coarse-grained, pipelined multiprocessing. Graphics optimizations focus on efficient data transfer to the graphics subsystem, reduction of mode settings, and restricting state inheritance. The toolkit's multiprocessing features solve the problems of how to partition work among multiple processes, how to synchronize these processes, and how to manage data in a pipelined, multiprocessing environment. The paper also discusses support for intersection detection, fixed-frame rates, run-time profiling and special effects such as geometric morphing.

Keywords: Real-time graphics, multiprocessing, visual simulation, virtual reality, interactive 3D graphics

CR Categories and Subject Descriptors: I.3.2 Graphics Systems; I.3.3 Picture/Image Generation; I.3.4 Graphics Utilities, Application Packages, Graphics Packages; I.3.7 Three-Dimensional Graphics and Realism

1 Introduction

Recently, multipurpose workstations have attained graphics performance levels that have customarily been the province of expensive, special-purpose image generators (IGs). Consequently, many visual simulation applications are migrating from IGs to graphics workstations. Additionally, the decrease in the cost/performance ratio of current-generation workstations has opened the door to non-traditional visual simulation applications such as virtual reality and location-based entertainment. These applications are often very cost-sensitive and so demand every drop of speed from the machine.

*2011 N. Shoreline Blvd., Mountain View, CA 94043 USA
jrohlf@sgi.com, jimh@sgi.com.

1.1 Motivation

In our experience, application developers often have problems extracting graphics performance due to inexperience with the system and ignorance of the "new set of rules", some of them quite arcane, which must be followed for peak performance on each new graphics platform. Also, applications often forgo multiprocessing simply because the development of a multiprocessed application proves too difficult or time-consuming. The resulting single-threaded applications sequentially process all tasks, leaving an expensive graphics subsystem idle while the application carries out non-graphics processing.

Existing general purpose 3D libraries and toolkits tend to address different problems. Immediate-mode rendering libraries such as OpenGL[9], Starbase[6], and XGL provide an efficient interface to hardware, but leave the definition of geometry, scene content and multiple eye points to the application. Object-oriented toolkits such as PHIGS+[13], HOOPS, Doré[7] and IRIS Inventor[12] provide scene structures based on display lists and objects, but for most efficient rendering they retain an internal copy of the geometric data. Since applications often need access to the original data for other purposes, a second inaccessible copy inside the toolkit can substantially increase memory usage. In addition, when the application dynamically changes geometry, the retained data must be edited or rewritten. Depending on the toolkit, this can increase program complexity, degrade performance, or both.

Most importantly, none of the aforementioned toolkits addresses multiprocessing. And from our experience, retrofitting a retained-database toolkit with efficient multiprocessing support and parallel traversals proves difficult at best.

In addition to demanding maximum performance, visual simulation and virtual reality applications have real-time requirements and must run at fixed frame rates to avoid the distractions and artifacts caused by frame rate variations. To achieve reasonable performance, these applications require efficient database culling to the viewing frustum, scene complexity management through level-of-detail switching, intersection testing, and run-time profiling for application and database tuning. Toolkits written specifically for visual simulation such as VisionWorks[10] and GVS[8] partially address many of these issues, but neither offers a fully multiprocessed solution.

1.2 Purpose

The fundamental design goal of the toolkit is to provide a software development layer that delivers the greatest possible performance from the graphics workstation, freeing the application developer to concentrate on other matters. We achieve this primarily through:

- Graphics optimizations
- Multiprocessing

Another goal is to simplify the development of virtual reality and visual simulation applications by providing intrinsic support for common graphics and database operations such as multiple views, level-of-detail switching, morphing, intersection testing, picking,

and run-time profiling. However, the toolkit does not provide direct support for I/O devices, audio, or motion systems since these are not directly related to the core functions of a rendering platform or a multiprocessing framework. Some applications, such as the fly-through system shown in Figure 17, have added their own device support to IRIS Performer, as have developers of toolkits for particular application domains, e.g. dVS[5] and WorldToolkit[8].

The graphics optimizations and multiprocessing features of the toolkit are targeted for workstations which support immediate-mode graphics and small-scale, symmetric, shared memory multiprocessing.

1.3 Overview

The toolkit's core consists of two libraries: **libpf** and **libpr**. **libpr** consists primarily of optimized graphics primitives as well as intersection, shared memory, and other basic functions. **libpf** is built on top of **libpr** and adds database hierarchy, multiprocessing, and real-time features. This arrangement is illustrated in Figure 1 below:

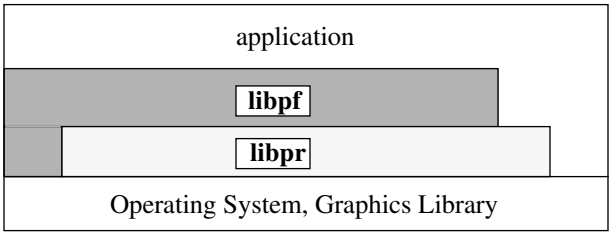


Figure 1. Library Layering

The two-library approach allows developers to choose which layer they wish to program to and also avoids "black box" limitations to flexibility by allowing an application which uses **libpf** to access the underlying **libpr** primitives. An application is also free to access the immediate-mode graphics library and operating system directly for customized rendering or control.

In keeping with our bottom-up design methodology, we discuss **libpr** first, then follow with **libpf** and finish with a description of run-time profiling utilities which facilitate performance tuning.

2 libpr - Efficient Rendering

The **libpr** library provides the high-performance foundation for IRIS Performer. Its specialized graphics primitives are designed to squeeze the highest level of performance from the graphics pipeline by efficiently managing geometry and graphics state for immediate-mode rendering. In addition, **libpr** supports intersection and shared memory utilities that facilitate a multiprocessed visual application.

2.1 pfGeoSet - Efficient Geometry Primitive

In our experience, the data structures used to represent geometry and the code which transfers that data to the graphics hardware very often make or break an immediate-mode graphics application. Scattered memory organizations can result in poor cache behavior and inefficient rendering loops can starve a fast graphics pipeline.

Immediate Mode vs. Display List Mode

The pfGeoSet's purpose is to achieve maximum immediate-mode performance for 3D geometry. In *immediate mode*, the host CPU must feed the graphics subsystem with primitive, vertex, and attribute commands. An alternative to immediate mode is *display list mode* which compiles a list of commands into a data structure that can be very efficiently transferred to the graphics subsystem. However, display list mode has some significant disadvantages that immediate mode does not have:

- A display list is a closed data structure. Geometry data must be duplicated at substantial memory penalty for database queries like intersections which require read access.
- Display lists are costly to compile. This generally requires that geometry be static. Techniques requiring vertex manipulation such as animation do not lend themselves to display list mode.

pfGeoSets utilize application-supplied arrays for attributes such as coordinates and colors, consequently avoiding these disadvantages. Applications are free to modify these arrays for dynamic effects without experiencing degraded rendering performance.

A pfGeoSet is a collection of geometric primitives of a single type defined by its:

- primitive type: points, lines, line strips, triangles, quads, or triangle strips
- attribute lists: coordinates, colors, normals, texture coordinates
- attribute bindings: per-vertex, per-primitive, overall, off.

Figure 2 illustrates a pfGeoSet consisting of two triangles with a per-primitive color binding: the first is red and the second is blue.

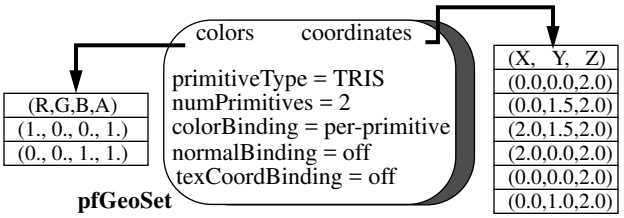


Figure 2. pfGeoSet Structure

On high-end machines in particular, care must be taken to ensure that immediate-mode data transfer is efficient or else the graphics hardware will be starved. pfGeoSets guarantee efficient data transfer by enforcing an *a priori* grouping of geometry by type that facilitates the use of customized, extremely tight rendering loops. Since all primitives within a pfGeoSet are homogeneous, a single, well-tuned rendering routine that is tailored to the specific pfGeoSet type can quickly transfer the primitives with a minimum of overhead. For example, if a pfGeoSet is a collection of triangles which have colors defined per-primitive (i.e., one color per triangle), its corresponding rendering routine doesn't waste precious if-tests determining whether or not a color should be sent down with each vertex. Over 700 of these specialized rendering routines exist (macro-generated) to handle all combinations of primitive types and attribute bindings, and all are indirectly accessed through the single pfDrawGSet() routine.

pfGeoSet Construction

Developers may find pfGeoSet construction messy and may sometimes generate pfGeoSets with sub-optimal performance, e.g., pfGeoSets with a small number of primitives may suffer from excessive setup overhead when transferring them to the graphics subsystem. Or an application may fail to use triangle meshes where possible. Connecting triangles together into a mesh can significantly reduce the amount of data transfer from the CPU to the graphics subsystem as well as the amount of processing required in the graphics hardware. Unfortunately, most databases do not utilize triangle meshing and automatic meshing algorithms are complex. To avoid these pitfalls, the pfuBuilder utility functions provide convenient meshing and performance-oriented construction of pfGeoSets. The application simply feeds independent, potentially concave polygons to a pfuBuilder which returns sorted, meshed,

and optimized pfGeoSets on request.

2.2 Efficient Graphics State Management

Unlike geometry, graphics state commands do not modify the frame buffer; they do not "draw" anything, but instead configure the graphics hardware with a particular mode (e.g. shading model) or attribute (e.g. texture) that modifies the appearance of geometry. Like geometry, efficient management of graphics state is required for optimal graphics performance.

In **libpr** there are 3 ways to set graphics state, each of which offers significant performance advantages:

- Immediate mode
- Display list mode
- Encapsulated mode

In general, applications use immediate mode to set global state such as enabling fog and use encapsulated mode to specify the appearance of geometry at database creation time. Display list mode is primarily intended for use by the **libpf** library to accommodate multiprocessing.

2.2.1 pfState - Immediate Mode

The state management provided by the pfState object is useful for avoiding redundant mode changes. A pfState object maintains all current and previous graphics state in a state stack. The set of managed graphics state is that which can be modified through **libpr** routines and is a subset of that provided by the graphics library. Graphics state is partitioned into:

- Modes such as backface culling, gouraud shading, wireframe on/off
- Attributes such as texture, material parameters

Modes are generally simple integer values that are set by single commands such as pfShadeModel() while *attributes* are objects like pfTexture that encapsulate many graphics characteristics. Modes are "set" and attributes are "applied" by their immediate-mode routines: pfShadeModel() and pfApplyTex() for example.

By shadowing the state of the graphics hardware, a pfState can eliminate costly mode changes. For example, if the current shading model is FLAT then a subsequent attempt at setting a FLAT shading model should be intercepted before being sent to the graphics hardware. Avoiding mode changes is especially useful for parallelized geometry engines which become essentially single-threaded during a mode change because mode changes must be broadcast to all engines. Redundant mode changes become particularly prevalent if the database is sorted by mode (See Section 3.1.3).

2.2.2 pfDispList - Display List Mode

The primary purpose of the pfDispList is to capture an entire frame's worth of data for use in multiprocessing. It captures and buffers **libpr** rendering commands such as pfShadeModel() and pfApplyTex(). As will be discussed in Section 3.2.2, two processes can communicate via a pfDispList to increase throughput. One *producer* process fills the pfDispList and a *consumer* process draws it by traversing it and sending appropriate commands to the graphics subsystem. Throughput is enhanced because the producer process off-loads expensive database processing from the time-critical consumer process which performs immediate-mode rendering. A pfDispList may be configured as a FIFO or ring buffer for concurrent producer/consumer configurations.

A pfDispList is different from a typical display list in that it captures only references to **libpr** objects and does not contain individual vertex or primitive commands; instead the **libpr** objects themselves contain and transfer these commands. Consequently a pfDispList can be quickly built and traversed. Additionally, a pfDispList is somewhat editable (it may be reused and appended to) and can also contain references to function callbacks for user-defined rendering.

2.2.3 pfGeoState - Encapsulated Mode

The pfGeoState object provides the primary mechanism for specifying graphics state in an IRIS Performer application. It encapsulates all state modes and attributes managed by **libpr**. For example, a pfGeoState may be configured to enable lighting and reference a wood pfTexture and a shiny pfMaterial. Then after it is applied to the graphics subsystem, subsequent geometry will have the appearance of a finished wood surface. A pfGeoState can be attached to a pfGeoSet so that together they define geometry with a specific appearance.

The pfGeoState has some special features that either directly or indirectly enhance rendering performance:

Locally Set vs. Globally Inherited State

It is possible to specify every **libpr** mode and attribute of a pfGeoState, in which case the pfGeoState becomes a true graphics context that fully defines the appearance of geometry. However, a full graphics context is fairly expensive to evaluate and is almost never required. The key observation is that many state settings apply to most geometry in the database. For example: fog, lighting model, light sources and lighting enable flag are often applied to the entire scene since they are global effects by nature. Conversely, attributes such as materials and textures are likely to change often within a database. pfGeoStates support these two kinds of state by distinguishing between *globally inherited* and *locally set* state respectively. By globally inheriting state, a pfGeoState can reduce the amount of state it sets, i.e.- it becomes sparse. A sparse pfGeoState is more efficiently managed because fewer pieces of state need be examined. State is inherited simply by not specifying it. However, an important point discussed below is that state is *never* inherited between pfGeoStates. As an important result, pfGeoState rendering becomes order-independent.

Order Independence

In many immediate-mode graphics libraries, geometry inherits previously set graphics modes. As a result, rendering is order-dependent; graphics state and geometry must be organized in a specific order to produce the desired appearance. Order dependence is undesirable for high-level database manipulations such as view culling and sorting which frequently modify rendering order.

To ensure order independence, the application must either completely specify the graphics state of all geometry or it must be aware of the current graphics state and change state when necessary. The former solution seriously compromises performance if the graphics context is non-trivial and the latter is a bookkeeping nightmare.

pfGeoStates guarantee order independence for rendering as a direct consequence of not inheriting state from each other. When applied, a pfGeoState implicitly saves and restores state so that its state modifications are insulated from other pfGeoStates. Furthermore, if a global state element is modified by a pfGeoState, it will be restored for those pfGeoStates which inherit that element.

Lazy Push/Pop

If a pfGeoState explicitly pushed and popped all graphics state, significant performance would be lost due to unnecessary mode setting. Instead, a pfGeoState pushes only those global state elements that it needs to change and pops only those global state elements that it needs to inherit and that were changed by a previously-applied pfGeoState. Lazy popping eliminates useless mode changes since a mode is not restored if a pfGeoState is going to change it anyway.

2.3 Multiprocessing Support

The **libpr** library is designed to fully support, but not require, a multiprocessing environment. To this end, **libpr** provides mechanisms for creating and maintaining shared data.

2.3.1 Shared Memory

libpr provides mechanisms for sharing memory between related (forked from the same image) and unrelated processes. Allocations are reference counted to support operations such as deletion in a multiprocessed environment (See Section 3.2.3).

2.3.2 pfMultibuffer - Multibuffered Arrays

When a process needs to modify a piece of data for consumption by other processes, data must be passed or multiple copies (buffers) must be maintained. To facilitate this, **libpr** provides multiprocessing constructs such as queues and multibuffered memory. The pfMultibuffer object provides data synchronization and data exclusion for multi-stage software pipelines by managing multiple copies of a single data array. pfMultibuffer is particularly useful for dynamic and morphing geometry. A global index for each process indicates the currently active pfMultibuffer buffer, e.g., process A may be working on buffer0 while process B is simultaneously working on buffer2. By changing the global index, processes can "pass" work to each other, simulating a processing pipeline. Since buffers are recycled rather than copied, the mechanism is efficient regardless of the amount of data which changes and independent of the number of consuming processes. When the contents of a pfMultibuffer stop changing, the most recent version is copied into each buffered instance so the application does not need to write every pfMultibuffer every frame.

2.4 Database Intersection

Most applications require intersection testing for purposes such as picking and collision detection. Since the target of these tests is often the visual data already represented in pfGeoSets, **libpr** provides the ability to intersect line segments against the polygons inside a pfGeoSet, thereby avoiding expensive duplication of the database. We chose line segments as the first primitive to implement because the tests are fast and they provide the most natural expression of common queries such as picking, line-of-sight visibility, and terrain following. Many simple collision detection mechanisms can be implemented by intersecting a set of line segments that describe the swept volume of a moving object with the database. The racing car simulator shown in Figure 15 uses two segments for following the track height and four segments for detecting collisions with walls and other cars. Several line segments can be grouped into a single intersection request to reduce processing overhead. Performance may be further improved by specifying an optional bounding cylinder which encompasses all line segments and by caching plane equations for static pfGeoSets.

pfSegsIsectGSet() returns the nearest or farthest intersection along each line segment. Applications can use a *discriminator callback* to examine each intersection individually during traversal of the geometry. Discriminator callbacks can direct the intersection traversal and/or modify the intersecting line segments for fine-grained intersection control. Intersection information available to the application includes the actual triangle within the hit pfGeoSet, the intersection position and geometric normal.

3 libpf - Adds Database Hierarchy and Automated Multiprocessing to libpr

Representing a visual database involves more than just geometry and its associated graphics state. A higher-level library, **libpf**, built on top of **libpr** provides a hierarchical scene graph of nodes which organizes **libpr** geometry for improved modeling and processing efficiency.

IRIS Performer accomplishes most database processing through *traversals* of the scene graph hierarchy. Much of **libpf**'s programming interface handles traversal configuration and control. Typically, an application updates scene graph and viewing parameters for a frame and then activates one or more processing traversals. For improved performance on multiprocessor systems, **libpf** can automatically execute these traversals in parallel with little extra programming burden on the application.

3.1 Database

A *scene graph* consists of *nodes* connected in a directed, acyclic fashion. Geometry lies at the leaves of the scene graph while internal nodes support notions such as grouping, transformation, selection, and sequencing as well as special operations such as level-of-detail switching, and morphing.

3.1.1 Class Hierarchy

While both **libpf** and **libpr** libraries are object-oriented, the flat class hierarchy of **libpr** allowed us to write it in C. However, the natural expression of scene graph nodes requires a deeper class hierarchy as shown in Figure 3. Consequently **libpf** is written in C++.

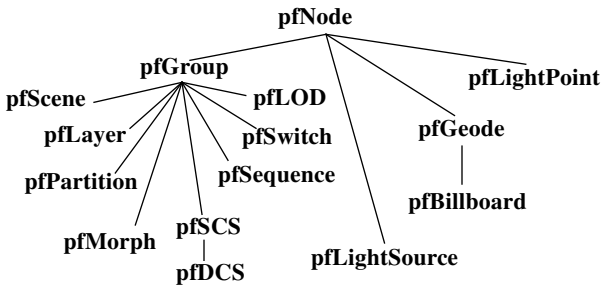


Figure 3. Node Class Hierarchy

Nodes fall into three groups: abstract, internal and leaf. pfNode is the abstract base class for all nodes and is itself derived from an internal class called pfUpdatable which creates and maintains multiple copies of the node for multiprocessing as described in Section 3.2.

The internal node types are:

- pfGroup: references pfNodes as children
- pfScene: group that roots a scene graph.
- pfSwitch: group with none, one, or all children active
- pfSequence: sequences through its children for animation effects
- pfSCS: applies an unchangeable transformation (static coordinate system) to its children
- pfDCS: applies a changeable transformation to its children (dynamic coordinate system)
- pfLayer: renders coplanar geometry, e.g. pictures on a wall.
- pfLOD: selects one or more children based on distance to eye, viewport pixel size, and field-of-view (level-of-detail).
- pfMorph: interpolates geometry, color, etc. between models
- pfPartition: spatially partitions geometry beneath it into an efficient data structure

The leaf node types are:

- pfGeode: references zero or more pfGeoSets
- pfBillboard: rotates pfGeoSets to face the eyepoint

- pfLightPoint: draws visible, but non-illuminating points of light, e.g. stars, runway lights
- pfLightSource: non-visible but illuminating light source

In a scene graph, pfGeodes typically contain most of the visual geometry. Each pfGeode references a set of **libpr** pfGeoSets. Specialized geometry is contained within pfLightPoints and pfBillboards.

3.1.2 Node Hierarchy

State Inheritance

In addition to providing organizational and instancing capability, a hierarchy of nodes (scene graph) also allows state inheritance. Within a scene graph, inheritance is strictly top-down. The absence of any left-right or bottom-up inheritance allows arbitrary pruning of the scene graph during traversal. This also facilitates parallelization of a single traversal because subgraphs of the scene graph can be traversed independently. The primary type of inherited state is 3D transformations, although user callbacks may also affect inherited state during traversals. Graphics state such as that defined by the pfGeoState primitive is *not* inherited through the scene graph. Grouping the primary specification of graphics state with leaf geometry rather than with internal nodes of the scene graph greatly facilitates tasks such as sorting by graphics mode.

Bounding Volume Hierarchy

The node hierarchy also defines a hierarchy of bounding volumes which are used to accelerate intersection and culling. Each node has a bounding sphere which encloses the node as well as any children it may have. The toolkit automatically recalculates these bounding volumes when geometry or scene graph topology changes.

All node types except pfScene retain parent lists. This allows a change to a child in a scene graph, such as a bounding volume change, to be propagated to all its ancestors in the scene graph. To eliminate redundant updates, internal state is marked using dirty bits which are propagated to the root of the scene graph so the cleaning of dirty state can be deferred until required.

3.1.3 Traversals

After the application configures the scene graph and viewing parameters, three basic traversals may process the scene graph:

- Intersection traversal (ISECT) — processes intersection requests for collision detection and terrain following.
- Culling traversal (CULL) — rejects objects outside the viewing frustum, computes level-of-detail switches, sorts geometry by modes
- Drawing traversal (DRAW) — sends geometry and graphics commands to the graphics subsystem.

TABLE 1. Traversal Characteristics

	ISECT	CULL	DRAW
Controller	pfSegSet	pfChannel	pfChannel
Global Activation	pfSegsIsectNode	pfCull	pfDraw
Modes	pfSegSet mode	pfChanTravMode	pfChanTravMode
Masks	pfNodeTravMask pfSegSet mask	pfNodeTravMask pfChanTravMask	pfNodeTravMask pfChanTravMask
Process Callback	pfIsectFunc	pfChanCullFunc	pfChanDrawFunc
Node Callbacks	pfNodeTravFuncs	pfNodeTravFuncs	pfNodeTravFuncs

Table 1 lists the **libpf** routines which define major characteristics of these 3 traversals

The default CULL and DRAW traversals are completely automatic and are triggered by pfFrame() (See Section 3.2.2). However, pfFrame() first triggers a partial traversal of the scene graph which cleans the internal state of the scene graph. Portions of the scene graph may have already been cleaned if the application called a routine which attempted to read a piece of state which was dirty.

Cull Traversal

The CULL traversal precedes the DRAW and uses many techniques to improve rendering performance by reducing load on both the DRAW traversal and on the graphics subsystem:

- Culling to the viewing frustum (pfChannel)
- Computing state specific to a pfChannel, e.g. level-of-detail
- Sorting for performance and visual quality
- Generating a simple display list (pfDispList) for the DRAW traversal

For applications with an eye point in the midst of the database, culling to the viewing frustum can reject the majority of geometry, substantially reducing the amount of data sent to the graphics subsystem. Viewing state and frustum are encapsulated by the pfChannel object. IRIS Performer supports multiple views, e.g. stereo, through multiple pfChannels which may view the same or different pfScenes.

The CULL traversal uses the hierarchical bounding volumes provided by the scene graph (See Section 3.1.2). Bounding spheres are used within the scene graph because they are fast to update, transform and test against. Axially aligned bounding boxes are used for each pfGeoSet to provide tighter bounds around the actual geometry.

During the CULL traversal the bounding sphere of each node is transformed as necessary and compared against the viewing frustum. The action taken depends on the result of the bounding volume test as follows:

- Completely outside the frustum: traversal continues without traversing any of the node's children — the node is *pruned*
- Completely inside the frustum: continue down the scene graph with no further culling tests
- Partially or potentially intersecting: continue testing and traversing down the scene graph

The ultimate output of the CULL traversal is the geometry and graphics state information to be sent to the graphics hardware. When enabled to do so, the CULL traversal first generates sorted lists of the pfGeoSets to be rendered. Each frame, these lists are sorted by graphics mode to increase rendering performance by minimizing expensive graphics mode changes such as transformation and texture changes. It is here that the order-independence offered by pfGeoStates (see Section 2.2.3) is especially useful. Next, the CULL traversal converts these sorted lists into a single pfDispList which eventually contains the entire frame. Transparent geometry is placed into the display list last, after a limited depth sort which improves both pixel-fill performance and the visual quality of the transparency. In our experience, mode sorting can significantly improve rendering throughput, sometimes more than 50%.

Draw Traversal

For each visual channel, the DRAW traverses the display list generated by its associated CULL traversal and sends commands to the graphics subsystem. The DRAW traversal differs from the CULL and ISECT traversals in that it does not involve traversing the actual scene graph. We designed the pfDispList format to be

very simple, so the DRAW traversal has very little work other than issuing graphics calls. The scene graph traversal overhead is absorbed by the CULL which increases rendering throughput when multiprocessing. When not multiprocessing, we can combine the CULL and DRAW traversals into a single traversal which both culls and issues graphics commands to avoid the small overhead of pfDispList generation.

Traversal Control

Nodes have separate traversal masks for each traversal type to allow the application to "mask off" subgraphs of the scene for traversal. A node is only traversed if the logical AND of the traversal mask and the node mask is non-zero. This allows multiple databases to coexist in the same scene graph. For example, a scene graph may contain simpler geometry for collisions than for rendering in order to reduce intersection times. In this case, the DRAW traversal mask for the collision geometry and the ISECT traversal mask for the visual geometry would both be zero.

Traversal Callbacks

Traversal callbacks provide even finer control on traversals. Each node can have its own pre- and post-traversal callbacks corresponding to each traversal type. These allow the application to prune or terminate the traversal at any time. The pre-CULL callback also allows the application to specify the result of the cull test for customized culling. The application may use the pre- and post-DRAW callbacks for custom rendering using **libpr** or the underlying graphics library, or to change and restore the graphics state for a portion of the scene graph. Figure 16 shows a real-time video effects program which uses DRAW callbacks to apply video texturing.

Intersection Traversal

ISECT traversals differ from the CULL and DRAW in that they are not automatic but are directly invoked by the application. Currently, intersections are based entirely on sets of line segments. The pfSegSet structure embodies an intersection request as a group of line segments, an intersection mask, discriminator callback, and traversal mode. The traversal consists of testing the pfSegSet against the hierarchical bounding volumes in the scene graph. Intersection "hits" can be returned for pfNode bounding volumes, pfGeoSet bounding boxes and the actual geometry inside pfGeoSets. In addition to the traversal callbacks described above, intersections also provide a discriminator callback so that the application can examine each "hit" during traversal and accept or reject the intersection as well as terminate traversal. Because ISECT traversals usually require a pfSegSet to be tested against many triangles, the traversal transforms the pfSegSet into local object coordinates rather than transforming the bounding volumes and pfGeoSets into world coordinates. Since intersections do not modify the database, applications may invoke many intersection requests in parallel.

Efficiency of Bounding Volume Hierarchy

The efficiency of both CULL and ISECT traversals is largely dependent on the depth and balance of the scene graph hierarchy. For example, a scene graph arranged as a balanced octree will cull more quickly than a flat scene graph. A scene graph with poor spatial hierarchy can be rearranged as a result of database profiling as described in Section 4.2 or be imposed with an improved secondary partitioning with pfPartition as described in Section 3.1.4.

3.1.4 Performance Optimizations

pfFlatten - Eliminating Transformations

Taking a single model and placing it under multiple static transformations (e.g. trees, houses) in the scene graph is convenient for modeling, but not always necessary at run time. During rendering, a transformation typically requires the hardware matrix stack to be pushed, the new transformation applied, the geometry drawn and then the matrix stack to be popped. For small models, these matrix operations can consume as much time or more than the actual rendering. pfFlatten() can improve graphics performance at a cost in memory usage by duplicating static, instanced geometry, applying the current static transform to the geometry, and setting all static coordinate systems (pfSCSes) to the identity matrix.

pfLOD - Level of Detail

Next to view frustum culling, the most important mechanism for reducing and managing the graphics load is level-of-detail (LOD) switching. When an object is only a few pixels large on the screen, it's wasteful to render a model with a high polygon count; rather, a coarser model with a lower level-of-detail should be rendered instead. The pfLOD node uses distance to the eye point, field-of-view, viewport pixel size, and graphics stress (see Section 3.3.2) to select among models of varying geometric complexity.

To make LOD changes as inconspicuous as possible, the pfLOD node can gradually fade between two models when switching. A drawback to fade LOD is that it requires rendering both models during the transition which temporarily increases the graphics load. An alternative LOD mechanism provided by the pfMorph node is described in Section 3.1.5 and can avoid this penalty by smoothly migrating vertices from one LOD to another.

pfSequence - Animation Sequences

Most high-quality animation requires moving vertices every frame. But for the highest performance with minimal CPU loading, most real-time applications make extensive use of precomputed animation sequences such as a sequence of textures to simulate a flickering torch. The pfSequence node supports this by automatically sequencing through its children. Each child is assigned a period of time, rather than a number of frames, during which it should be displayed so that the sequence is immune to frame rate variations. An example of pfSequence use is the dragon seen in the background of Figure 13.

pfBillboard - Billboarded Geometry

Rotating geometry, usually a single textured polygon, so that it always faces the eye is a trick from visual simulation used for axially and radially symmetric objects such as trees, clouds and special effects such as smoke or fire. Using a billboarded polygon instead of a full three-dimensional model reduces both geometry and pixel fill demands on the graphics pipe. A pfBillboard can be constrained to rotate about an axis or a point. The trees and lamp posts in Figure 14 are examples of pfBillboards.

pfPartition - Spatial Data Structure

IRIS Performer relies on the hierarchical bounding volumes of a scene graph to accelerate intersection and culling traversals. However, a user-constructed scene graph may exhibit poor spatial arrangement, obviating the benefits of hierarchical bounding volumes. In this case a specialized spatial data structure imposed on the default scene graph can provide much higher performance, particularly for intersections. The pfPartition group node analyzes geometry underneath it at database load time and partitions pfGeoSets into a 2D grid with multiple membership. During the intersection traversal, line segments in a pfSegSet are scan converted onto the grid to quickly determine which pfGeoSets need to be tested against. Other types of spatial data structures may be added in the future.

3.1.5 Special Features

pfMorph - Morphing

The pfMorph node provides a mechanism for interpolating geometry between many sources. A pfMorph takes a set of input arrays and weights and places the linear combination of the input arrays

into an output array. Typically, the morphed arrays are the vertex, color, normal or texture coordinate arrays of a pfGeoSet in the scene graph beneath the pfMorph node. The two main applications are for continuously varying animated geometry such as the head of the creature in the foreground of Figure 13 and for continuous LOD switching [3]. The latter allows nearly invisible LOD transitions and can be more efficient than fade LOD if the cost of morphing is small compared to the cost of drawing two models during a fade transition.

3.1.6 Database Importation

IRIS Performer is strictly a runtime programming interface with an in-memory scene representation and currently has no database file format. An application calls toolkit routines to create and assemble a scene graph from various elements such as pfNodes, pfGeoSets and pfGeoStates. Because the task of creating pfGeoSets can be tedious, a utility library built on top of the toolkit provides routines (pfuBuilder) to simplify the construction and triangle meshing of pfGeoSets. Using these, database loaders have been written for various database formats including Autodesk DXF, Wavefront OBJ, Software Systems FLT, Coryphaeus DWB, and LightScape LSB. Database formats with a hierarchical scene graph and visual simulation extensions (e.g. level-of-detail, billboards) map directly to the toolkit scene graph. For those database formats without any hierarchy, the utility library provides spatial octree-based breakup of geometry (pfuBreakup) so that even large, monolithic models can be organized into a scene graph for efficient culling and intersecting.

3.2 Multiprocessing

A fundamental design criterion of the toolkit was to improve performance through multiprocessing while hiding the programming complexities that multiprocessing creates. This section describes our solutions to the following multiprocessing problems:

- How to partition work among multiple processes
- How to synchronize process execution
- How to manage data in a pipelined, multiprocessing environment

3.2.1 Pipelined Multiprocessing

IRIS Performer employs a *coarse-grained*, *pipelined*, multiprocessing scheme, i.e., a relatively small number of processes work concurrently on different stages of one or more processing pipelines. This configuration favors workstations with a relatively small number of processors (tens) over massively parallel systems (thousands). The partitioning of work into multiple processes is based on *processing stages*. A processing stage is a discrete section of a processing pipeline and encompasses specific types of work. Processing stages are tightly coupled to the scene graph traversals described in Section 3.1.3. The ISECT, CULL, and DRAW processing stages consist of zero or more intersection, culling, and drawing traversals respectively in addition to application-specific processing that is accessed through function callbacks. An additional processing stage, the APP, consists primarily of application code as well as database, viewpoint, and system modifications made through toolkit routines. Together, these four stages define two kinds of processing pipelines:

- rendering pipeline: APP → CULL → DRAW
- intersection pipeline: APP → ISECT

pfPipe - Rendering Pipeline

The APP stage is the head of all pipelines and controls their execution. A rendering pipeline consists of the CULL and DRAW stages and is encapsulated by the pfPipe primitive. An application may use one or more parallel pfPipes that each renders zero or more viewpoints into a single graphics window. The multipipe feature is provided for machines with multiple graphics subsystems and includes support for time-multiplexing the output of multiple hardware renderers to a single display. The intersection pipeline consists of the ISECT stage. Only one intersection pipeline is supported.

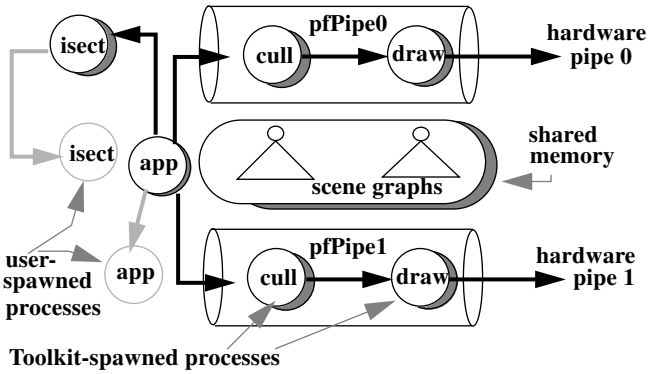


Figure 4. Multiprocessing Multipipe Configuration

Multiprocess Partitioning - pfMultiprocess

Multiprocessing in IRIS Performer is achieved by splitting the rendering and intersection pipelines at stage boundaries into multiple processes. For example, the APP and CULL stages may be combined into a single process while the DRAW stage is split into a separate process, resulting in a 2-process configuration which is suitable for a 2-processor machine. The application specifies this partitioning through pfMultiprocess(), allowing applications to choose a process partitioning based on the number of available CPUs. Figure 4 illustrates a processing configuration consisting of two rendering pipelines and an intersection pipeline where each stage has been split into a separate process. Figure 5 illustrates different multiprocess partitionings of the rendering pipeline that range from 1 to 3 processes.

Multiprocessing With Shared vs. Non-shared Address Space

All pipelined processes are created by pfConfig() using the fork() mechanism. We chose fork() over mechanisms which allow a fully shared virtual address space so we could selectively share memory and support multiple graphics pipes, since not all immediate-mode graphics libraries allow multiple rendering contexts within a single virtual address space. Synchronization for all processes created by pfConfig() is handled internally.

Additional Multiprocessing

Additional multiprocessing is easily acquired if the application itself creates extra processes. The ISECT and APP stages particularly lend themselves to this kind of multiprocessing. For example, multiple ISECT processes may concurrently execute calls to pfSegsIsectNode() which intersects a set of line segments with a scene graph (see Section 3.1.3). However, synchronization for these processes is the responsibility of the application. The stippled circles in Figure 4 depict these user-spawned processes.

3.2.2 Process Synchronization

Process synchronization defines the execution order of multiple processes. It is responsible for enforcing periods of mutual exclusion between processes and for ensuring concurrent execution of processes. Most process synchronization in the toolkit is achieved through well-known mechanisms such as semaphores and locks.

Throughput vs. Latency

IRIS Performer enforces pipelined synchronization of processes created by pfConfig(). Pipelined multiprocessing trades increased throughput for increased latency. *Rendering latency* is defined as

the time elapsed from viewpoint specification until the display is completed for that viewpoint. *Rendering throughput* is defined as the amount of geometry processed in unit time. The size of the throughput vs. latency trade-off is dictated by the number of processes in the pipeline (its *depth*) and increases with process count. Pipeline depth is configurable and can range from 1 to 3. For example, a configuration combining the APP and CULL into a single process and separating the DRAW will generate a rendering pipeline whose depth is 2. If all pipeline stages are well-utilized, performance can be increased over the single-processed case by a factor equal to the pipeline depth.

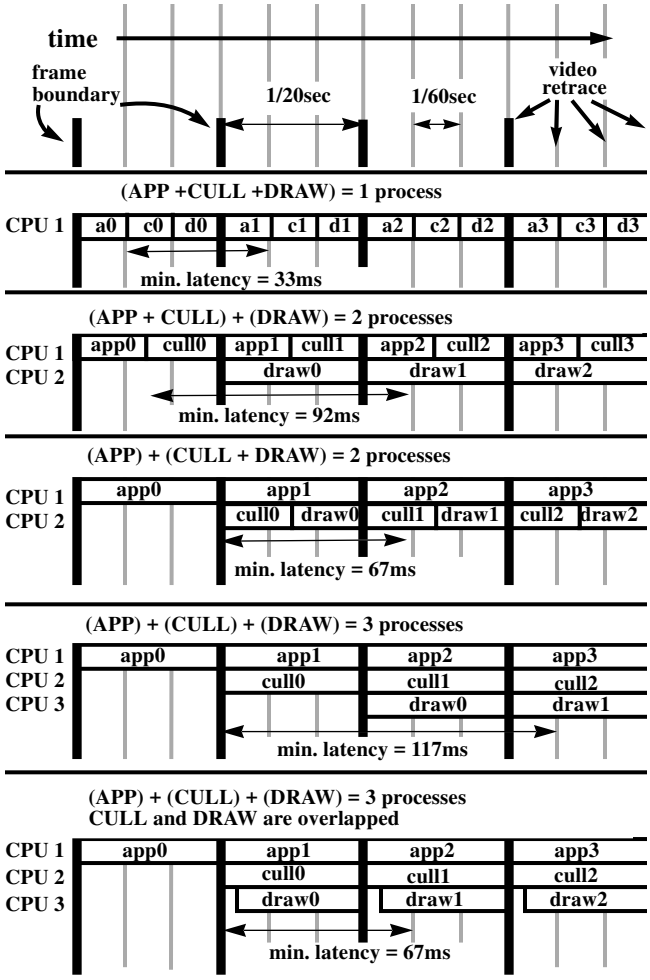


Figure 5. Multiprocess Partitioning and Timing Diagram

Figure 5 illustrates timing diagrams for different multiprocess configurations ranging from 1 to 3 processes that are running at 20Hz. Boxes represent the execution time of individual stages and each row of boxes corresponds to a single process. Thus, multiple rows of timing boxes illustrate parallel execution of pipeline stages. The text inside the boxes specify the stage or stages that the process handles while the numbers indicate the frame that the process is currently working on. Notice how the amount of time available to each stage (throughput) increases as the number of processes (pipeline depth) increases.

Frame Control

The toolkit typically synchronizes the application to a user-specified frame rate, e.g. 30Hz. This frame rate defines a series of frame boundaries that demarcate the beginning and ending of a frame. The APP stage is responsible for synchronizing to the specified frame rate and for triggering all processing pipelines once per frame by calling pfSync() and pfFrame() respectively.

pfSync() suspends the calling process until the next frame boundary and is discussed in more detail in Section 3.3.1. pfFrame() indicates that all rendering and intersection pipelines should begin processing a new frame. If a pipeline stage is not ready to begin processing a new frame because the processing time for the previous frame exceeded the allotted frame time, the stage has *frame-extended*. In this event, pfFrame() does not block but returns control to the application. If the APP process frame-extends, then pfFrame() is not called often enough and the update rate drops even if the rendering pipeline can keep up. For this reason, application processing *must* be kept to within a frame time.

Improving Latency

Certain applications like “man-in-the-loop” flight simulation and virtual reality applications utilizing a head-tracked display require very low latencies [14]. The latencies listed in Figure 5 are timed from the end of the APP processing until video scanout of the last pixel. To ensure this minimal latency even in cases when the APP takes less than its full allotment of time, the toolkit allows latency-critical updates such as the viewpoint to be made just before kicking off the CULL traversal with pfFrame(). Figure 6 depicts a close-up view of how pfSync() and pfFrame() work together to synchronize process execution. Latency-critical updates are made in the shaded portions of the APP processing time and may reduce throughput by delaying the triggering of the processing pipelines.

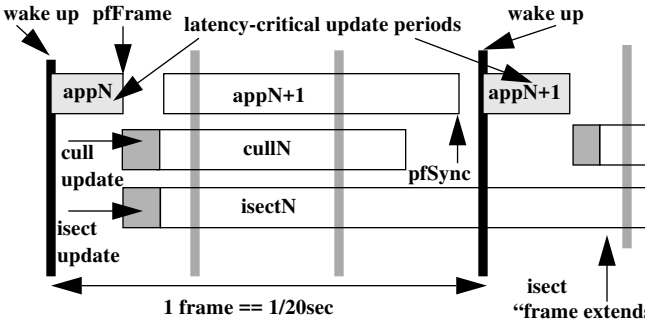


Figure 6. pfSync and pfFrame

The following pseudo-code fragment illustrates the use of pfSync() and pfFrame() in a typical simulation loop:

```
while(!Done)
{
    updateSim();    /* Make non-latency-critical updates */
    pfSync();       /* Sleep until next frame boundary */
    updateView();   /* Read input devices and update eyepoint */
    pfFrame();      /* Trigger new frame */
}
```

A special multiprocessing mode illustrated by the last timing diagram of Figure 5 eliminates an entire frame of latency by overlapping the CULL and DRAW processes that are working on the same frame. The two processes communicate via a FIFO which stalls a process on empty and full conditions. Although the DRAW has to wait for the CULL to begin filling the FIFO and will stall if it is faster than the CULL, in practice neither of these drawbacks are significant. In this overlapped case, latency is reduced to a single frame, generally the lowest possible. When CULL and DRAW are not overlapped, latency can still be reduced to a single frame by culling to a slightly larger viewing volume and sampling a new viewing position just before drawing.

A lower latency alternative to pipelined multiprocessing would be a single, multithreaded scene graph traversal. We chose against this method due to the *much* higher complexity and overhead arising from the necessary fine-grained synchronization. Also, the threads

would have to be single-threaded when the application makes random access modifications to the database and when rendering, if the graphics pipeline does not allow multiple writers.

Pipeline Bottlenecks

Ideally, each process in the pipeline takes exactly one frame time to complete its work. This situation indicates a balanced pipeline that is getting maximum utilization of its processors and is the one depicted in Figure 5. An out-of-balance situation arises when a particular process takes longer than all other processes in the pipeline and becomes a bottleneck. In most graphics intensive applications, the process handling the DRAW stage is the bottleneck. In this case, draw times can be reduced through the stress management techniques described in Section 3.3.2. If the bottleneck is due to the CULL stage, times can be reduced by disabling one or more culling modes. Bottlenecks due to the APP stage are largely the responsibility of the application.

Process Callbacks

By default, IRIS Performer performs all rendering processing when triggered by pfFrame(); culling and drawing functions are carried out in "black box" fashion. *Process callbacks* provide the user with the ability to execute custom code both before and after default processing, and to execute the code in the appropriate process when multiprocessing.

Process callbacks are provided for the ISECT, CULL, and DRAW stages. Default processing for these stages is triggered by pfSegsIsectNode, pfCull, and pfDraw respectively. If a callback is specified, default processing is disabled and must be explicitly triggered by the callback. This arrangement allows the user to "wrap" default processing with custom code, allowing save/restore, before/after, and multipass rendering methods which use techniques such as projective textures [11]. Figure 12 is from an application which uses multipass renderings with projective textures to simulate a spotlight with real-time shadows. In practice, the DRAW callback is often used for 2D graphics, textual annotations and specialized rendering that requires the full flexibility of the underlying graphics library. A typical DRAW callback is illustrated below:

```
void
drawCallback(pfChannel *chan, void *data)
{
    clearFrameBuffer();
    pfDraw();
    drawSpecialStuff();
}
```

3.2.3 Data Management

Three problems plague data management in a pipelined multiprocessing environment:

1) Data visibility. Processes need to share data.
2) Data exclusion. A process must not modify data while other processes are simultaneously reading and/or writing it.
3) Data synchronization. Data modifications must be propagated down processing pipelines in a "frame-accurate" fashion.

1) is handled by the shared memory mechanisms described in Section 2.3.1. **2)** can be handled with hardware spin locks but fine-grain locking becomes expensive and as we shall see, the data exclusion problem is solved by the solution to **3)**. First, let us examine the data synchronization problem more closely.

Data Synchronization

In the toolkit's multiprocessing pipelines, multiple processes work on different frames at the same time. For example, the APP process works on frame 33 while the DRAW is on frame 31. Suppose a single matrix in shared memory represents the position of a database model. If the APP process updates this matrix while the DRAW process is sending it to the graphics hardware, the matrix might be partially updated when sent to the graphics, resulting in an unintended combination of two matrices. Alternatively, the model might be drawn at the position it should have at frame 33, rather than frame 31. In this case we say that the matrix update is not *frame-accurate* since it does not affect the displayed model at the appropriate time.

Note that hardware pipelines exemplified by graphics subsystems such as RealityEngine[1] solve the data synchronization problem by copying the entire database down through the pipeline. While wide, fast data paths make this practical for hardware pipelines, software pipelines do not have this luxury and require another approach.

Multibuffering

We solve the problem of data exclusion and data synchronization with a technique called *multibuffering*. Multibuffering employs multiple copies of data structures known as pfUpdatables (or updatables) that are logically partitioned into buffers known as pfBuffers. All **libpf** objects including pfNodes are pfUpdatables so that each pfBuffer contains a full copy of the scene graph. A pfBuffer is associated with a single process and that process may access only those pfUpdatables in its pfBuffer, thereby solving the data exclusion problem.

Modifications made to pfUpdatables by the APP process are recorded in an update list. Each frame these updates are applied to all downstream pfUpdatables so the updates propagate down all pipelines in frame-accurate fashion, thereby solving the data synchronization problem. Propagating only database modifications significantly reduces the amount of data that flows through the processing pipelines.

This update-based multibuffering mechanism is most useful when making sparse modifications to largely static data structures. This is in contrast to the pointer-switching type of multibuffering provided by pfMultibuffer (see Section 2.3.2) which is most suitable for data structures with large changes, such as vertex arrays used in morphing. In this case, swapping pointers is much more efficient than copying large amounts of data.

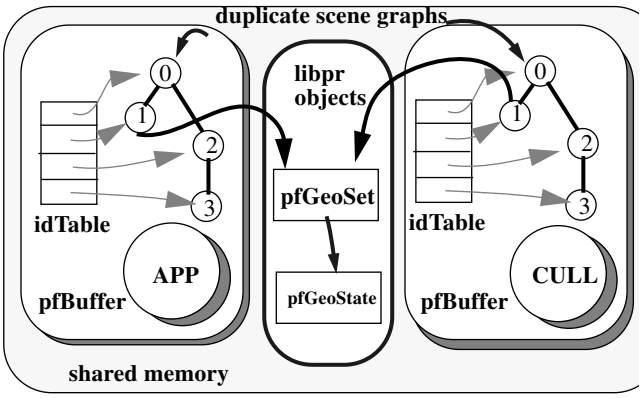


Figure 7. Multibuffering of a Scene Graph for APP and CULL

pfBuffer and pfUpdatable

In addition to forking multiple processes, pfConfig() creates and associates a pfBuffer with each process (except the DRAW as is discussed below). Each pfBuffer has an id table which associates the address of a pfUpdatable with its id. When created, a pfUpdatable is assigned a unique integer id and is added to the id table of the creating process' pfBuffer. Then during the period when updates are exchanged, corresponding pfUpdatables are created in all downstream pfBuffers. Figure 7 depicts the referencing of two copies of the scene graph (one each for the APP and CULL processes) through the pfBuffer's idTable.

Selective Multibuffering

The net result is that N "images" of each pfUpdatable are created: one for each pfBuffer in use. At first glance this may seem to be an extravagant use of memory. However, only pfUpdatables are multibuffered and only **libpf** objects are pfUpdatables, e.g. pfNodes, pfChannels. Thus all **libpr** primitives such as pfGeoSets and pfGeoStates are *not* multibuffered and do not suffer the memory penalty that multibuffering introduces. This design decision relies on the following assumptions:

- Geometric primitives like pfGeoSets and pfGeoStates represent the vast majority of database memory. Thus, duplicating only the scene graph skeleton does not drastically increase memory usage.
- Most geometry is static and does not require the frame-accurate behavior provided by multibuffering. (In Figure 7 the pfUpdatable numbered "1" is a pfGeode that references a non-multibuffered pfGeoSet.)

Although the first assumption has proven reasonable in most circumstances, we are currently exploring a "copy-on-write" extension to the multibuffering mechanism which would create extra copies only when an updatable is modified. The second assumption however, is restrictive in applications which use sophisticated morphing techniques like continuous terrain level-of-detail that require vertex-level manipulations of geometry [3]. Without multibuffering, the APP process may modify geometry at the same time the DRAW is sending the geometry to the graphics subsystem, resulting in cracks between adjacent polygons. To solve this problem we have offered a solution with the pfMultibuffer primitive described in Section 2.3.2.

Data Exclusion Revisited

In addition to frame-accurate behavior, multibuffering provides data exclusion which is essential to robust multiprocessing. Since each process is guaranteed exclusive access to updatables in its pfBuffer, it need not worry for example, that the APP process has removed a node from the scene graph. Otherwise, the process might collide with the modification and dereference a bad pointer with disastrous results.

Update List

An *update* consists of an updatable id and another integer id which defines what has changed. For example an update of [31, 12] might mean "update the transform of the pfDCS whose id is 31." Recording updates by reference has significant advantages over recording updates by value, which in the above example would mean copying the transformation matrix into the update list:

- Updates are homogeneous, thereby simplifying code and data structures
- Updates are small, resulting in quick recording and memory conservation
- Updates have a unique key which allow them to be efficiently managed by a hash table. Specifically, duplicate updates are discarded, keeping the update list from growing without bound.

The primary disadvantage of this update form is that it requires blocking the upstream process during the update period described below.

In order to provide frame-accurate behavior, updates must propagate in an orderly fashion down all processing pipelines. This propagation period occurs during pfFrame(). At this point all processes downstream of the APP (all CULL and ISECT processes) traverse the update list generated by the APP process and update their pfUpdatables. Each update consists of copying a portion of a pfUpdatable in the upstream pfBuffer into the corresponding pfUpdatable in the downstream pfBuffer. For the pfDCS example mentioned above, we would copy only the transformation matrix between pfDCS copies. At the end of the update period, all pfUpdatables in the downstream pfBuffer are identical to those in the upstream pfBuffer.

During the update period, the upstream process (the APP) must be blocked so that it cannot modify updatables in its buffer and possibly corrupt the update data exchange; we must ensure data exclusion. This update period is illustrated in Figure 6 as the shaded portions of the CULL and ISECT processes.

Figure 8 illustrates an APP feeding two pipelines: one intersection and one rendering pipeline. In this case there are three pfBuffers - one each for the APP, ISECT, and CULL processes.

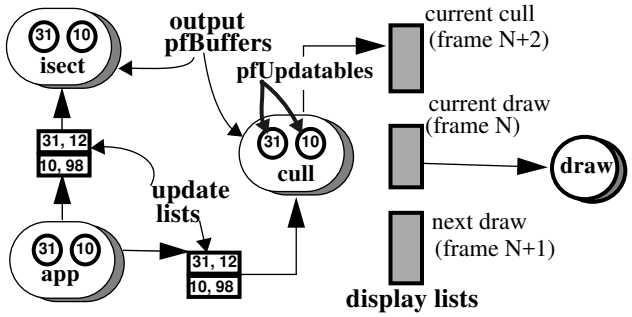


Figure 8. Interprocess Communication for Processing Pipelines Using Update Lists and Display Lists

Pipeline Frame Extension

The APP pfBuffer maintains an update list for each processing pipeline and appends all updates to all update lists. If a downstream pipeline is not ready to accept the update list when pfFrame() is called because it has frame-extended, the APP does not block but continues with the next frame. In this case, the update list corresponding to the frame-extended pipeline is not reset so that further updates are appended to the list and previous updates are not lost; they will be consumed later when the pipeline is ready. If the APP is feeding multiple pipelines, all ready pipelines update themselves in parallel.

Cull/Draw Communication

Note however that the DRAW process in Figure 8 does not have a pfBuffer and uses a different communication mechanism with the upstream CULL process. This is not precluded by the pfBuffer/pfUpdatable mechanism but was chosen to reduce memory requirements and performance degradation. When the CULL and DRAW stages are in separate processes, the CULL process traverses the scene graph and renders visible geometry into a **libpr** display list (See pfDispList in Section 2.2.2). This is very important because it off-loads scene graph traversal overhead from the time-critical DRAW process. However, this means that there is no need for a scene graph in the DRAW process. Also, maintaining a pfBuffer in the DRAW process would require an update period that would steal precious drawing time.

As illustrated in Figure 8, the CULL and DRAW communicate via three display lists. In a perfectly balanced pipeline, only two display lists would be required — the classic double-buffered configuration. However, both CULL and DRAW processes may frame-extend. As a result, a third display list is required to keep the non-extending process from waiting until the extending process is finished with its display list.

pfDelete - Object Deletion

Deletion of a hierarchical scene or subgraph that supports instancing can be tricky. Care must be taken to ensure that an object's memory is not freed until all references to it are removed. To do

otherwise would open the possibility of corrupted memory and ungraceful program cessation. IRIS Performer employs a reference counting scheme to avoid such results.

Whenever an "attachment" is made between two objects, the reference count of the "attachee" is incremented by one. Reference count modifications are locked to ensure data exclusion between multiple processes. pfDelete() deletes objects whose reference counts are non-positive and follows all reference chains, deleting objects until it reaches one whose reference count is greater than zero. The reference count of a pfNode is simply the number of its parents. User-allocated memory such as the attribute arrays of pfGeoSets (See Section 2.1) are reference-counted if the memory is allocated by **libpr** routines since they maintain internal reference counts.

Multiprocessed Delete

Unfortunately, multiprocessing adds another dimension to reference counting. Non-multibuffered objects such as pfGeoSets are "referenced" by the processes which are accessing them. For example, the ISECT and DRAW processes may be concurrently intersecting with, and rendering a given pfGeoSet. Consequently, a simple reference counting scheme is inadequate.

One possibility would be for processes to reference/dereference objects as they need them. This is unacceptable from a performance standpoint since locks are not free and the number of objects needing locking is large. IRIS Performer's solution takes advantage of its pipelined configuration. An object is not immediately deleted; rather, a frame-stamped deletion request is added to a special list. Meanwhile, the back ends of all pipelines (ISECT and DRAW processes) record the frame count of their most-recently-completed frame. Then when pfFrame() is called, each deletion request on the list is examined. If its frame stamp is less than the frame counts of all pipelines, the deletion request is safely carried out since all pipelines have flushed themselves of the object.

3.3 Achieving Real-Time

3.3.1 Achieving Real-time Synchronization

Real-time behavior is often required of graphics applications, both for human and hardware (sensor) perception. Real-time in this context implies more than a reasonable frame rate. Equally important is a *fixed frame rate* which ensures a solid, consistent update rate without glitches or hiccups. In fact, many visual simulation applications sacrifice peak frame rates for a fixed frame rate.

The first step in achieving real-time behavior is accessing a timer that runs at wall-clock time, i.e., it runs at the same rate as the clock on your office wall. Since the graphics update rate is restricted to integral fractions of the video refresh rate, the video clock provides a natural real-time clock for synchronizing a graphics application.

pfVClockSync - Synchronizing to Video Retrace

The kernel maintains a video retrace counter and also provides a synchronizing feature that is accessed through the pfVClockSync() call. This routine takes two arguments, [*interval, offset*] that together specify the frame synchronization boundary. Put arithmetically, pfVClockSync() puts the calling process to sleep until the video retrace count modulo the *interval* equals the *offset*. For example, if pfVClockSync() is called with arguments of [3, 0] when the current video clock is 658, the process will sleep until the video clock is 660.

An application specifies its desired fixed frame rate and synchronizes the APP process to that rate by invoking pfSync() which calls pfVClockSync() to sleep until the next frame boundary. Note that pfSync() alone does not guarantee a fixed frame rate. First, the APP cannot take longer than a frame time because it would then synchronize to an integral multiple of the desired field rate such as 30Hz dropping to 15 Hz or even 10Hz. Second, the processing pipelines must be able to complete their work within a frame time as is discussed in more detail below.

3.3.2 Achieving A Fixed-Frame Rate

Once synchronization to wall-clock time is achieved, the next step in attaining real-time behavior is to ensure a fixed frame rate. Many things can compromise a fixed frame rate on a multiuser workstation:

1) Graphics context switching
2) Process context switching
3) Process frame extension (e.g. APP, CULL extensions)
4) Graphics pipeline frame extension (DRAW extension)

1) can be remedied by ensuring that only the application of interest is running: no clocks or performance meters allowed. **2)** may be solved by running the application with super-user privileges and using OS commands to isolate and restrict processors. **3)** is more difficult to solve and requires rearranging database hierarchies, disabling of modes, and further multiprocessing to unload the burdened process(es). **4)** is often the most prevalent enemy to a fixed frame rate and it is that which we address in this section.

Graphics pipelines have hard limits on the amount of geometry they can process in a given time. Ideally, the throughput of a graphics pipeline is always enough to render the desired amount of scene geometry in the desired amount of time. In this case a fixed frame rate is easily achieved. However, most scenes have varying geometric complexities due to varying scene density and/or moving models which may come into view. If a frame rate is chosen such that the view of highest complexity may be rendered within a frame time, then the expensive graphics hardware will be underutilized for less complex scenes. On the other hand, if a higher frame rate is chosen, complex scenes will take longer to render than the allowed frame time and distracting visual anomalies, technically referred to as "hiccups", will occur. Consequently, many applications choose a frame rate that can handle the average scene and rely on other mechanisms to artificially reduce more complex scenes so that they can be rendered within a frame time.

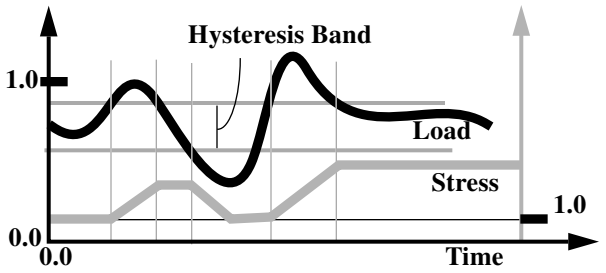


Figure 9. Stress Feedback Filtering

Stress management is the technique used to reduce scene complexity that relies on the level-of-detail mechanism described in Section 3.1.4. When the system is in stress, LODs are artificially reduced; coarser than normal models are chosen, so that overall graphics load is reduced. Stress is based on *load*, the fraction of a frame time taken to render a frame, and increases as load exceeds a user-specified threshold. The load for frame N is used in conjunction with user-specified parameters to define the stress value for frame N+1, thus defining a feedback network. As discussed in [4], this method works reasonably well for relatively constant scene densities but suffers because the stress is always a frame late and can exhibit oscillatory behavior. As illustrated in Figure 9, a hysteresis band can reduce stress oscillations but a more sophisticated stress management technique such as that described in [4] has bet-

ter characteristics.

3.3.3 Overload Management

While stress management seeks to fit DRAW processing into a frame time, *overload management* dictates what happens when stress management has failed and the DRAW exceeds a frame time — it has frame extended. The application may choose differing overload behavior by selecting the *phase* of the DRAW process. Phase dictates the type of synchronization used by the DRAW process: if the phase is *locked*, the DRAW process is guaranteed to begin only on a frame boundary. Thus if the DRAW takes just slightly longer than a frame time, the aggregate frame rate will drop in half. If the phase is *floati*ng, a frame-extended DRAW will start to draw again as soon as it can (at the next vertical retrace) and try to "catch up", relying on stress management to reduce scene complexity. In practice, floating phase is used more often than locked phase since it does not sacrifice an entire frame time if the DRAW takes just slightly longer than a frame. However, locked phase offers deterministic latencies and can produce a steadier frame rate.

4 Run-Time Profiling

Without proper profiling and diagnostic utilities, it is difficult to ascertain the performance of a given application. "Is it running as fast as it can go?" is the most pertinent question. To answer, the developer must be able to answer other questions concerning potential bottleneck areas:

- CPU processing, e.g., is the APP taking longer than the DRAW?
- CPU to graphics transfer, e.g., is the bus saturated or is the DRAW suffering from overhead due to small pfGeoSets?
- Geometry transform, e.g., are excessive mode changes thrashing the Geometry Engines? Are my triangle strips too short?
- Geometry fill, e.g., is the pixel depth complexity too high?

To further complicate matters, bottlenecks change and shift as the visual scene changes, making them moving targets for the tuner.

To aid application and database tuning, IRIS Performer provides extensive profiling information that is collected at run-time and may be graphically displayed for easy comprehension. Run-time collection provides a display of up-to-date information as you fly through the database, facilitating an interactive and time-saving approach to tuning. Figure 10 is the statistics display for the scene in Figure 14 and shows both process and database statistics measurements that are examined in the following sections.

4.1 Process Statistics

Due to the concurrent, time-dependent nature of multiprocessing, it is often difficult to understand the behavior of a multiprocessed application. IRIS Performer records the times spent by each processing stage and displays the results in a timing diagram which quickly exposes any bottlenecks. In Figure 10, the upper portion of the display defines a timing diagram analogous to those in Figure 5. Vertical lines indicate vertical retrace and frame boundaries. Horizontal lines indicate the processing times for different stages and their color indicates the stage's frame count.

Example Analysis

From Figure 10 we see that the application is configured as 4 processes, one each for ISECT, APP, CULL and DRAW, which all run in parallel. Additionally, the processing times for CULL and DRAW are roughly equivalent and occupy most of a frame time indicating that 30Hz is a reasonable frame rate and load balancing is good. (Note that the time required to draw the statistics display itself pushes the draw time over 1/30 sec.) However, the APP and ISECT stages take little time so we could free a CPU by combining these two stages into a single process.

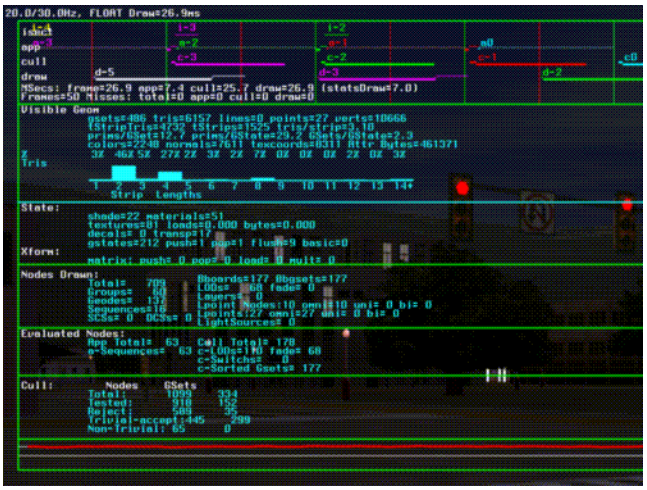


Figure 10. Display of Process and Database Statistics

4.2 Database Statistics

Although the toolkit strives to achieve maximum performance with a given database, a significant amount of performance gain may lurk within the database itself. For example, a scene graph without hierarchy will suffer from poor intersection and culling performance, both of which rely on hierarchical bounding volumes to accelerate processing. Also, a pfGeoSet which contains few triangles will suffer from overhead in pfDrawGSet(). These problems and more can be easily inferred from the statistics display of Figure 10.

Example Analysis

The ratios of primitives to pfGeoState (12.7) and pfGeoSets to pfGeoState (2.3) are reasonably high, indicating that pfGeoSet and pfGeoState overhead is not likely a problem. However, the average number of triangles per strip is low at 3.1 which indicates that the hardware geometry processing stage may be a bottleneck. This fragmentation of the database is likely due to the large number of textures (81) since a strip cannot span multiple textures.

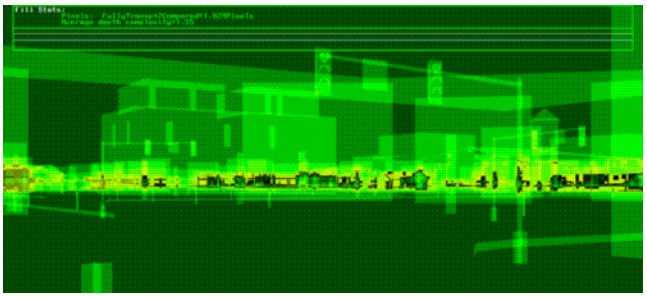


Figure 11. Profiling Display Depicting Pixel Depth Complexity

Although Figure 10 reveals much about the database, it says nothing about the pixel fill bottleneck which is the most important one for the majority of full-screen applications. The toolkit provides a special mode for visualizing pixel *depth complexity*, the number of times each pixel is touched. Figure 11 is a false-color visualization of the depth complexity for the scene of Figure 14. Depth complexities of up to 7 are represented by colors of increasing brightness (some areas have complexities > 7 and wrap). Additionally, the total number of pixels rendered and the average depth com-

plexity is displayed. All of these statistics are computed and displayed at a run-time, albeit at a reduced frame rate.

4.3 Future Work

Our design approach has been to focus on the performance and structure of the toolkit's rendering and multiprocessing core. Because of this, we believe the toolkit provides a good foundation for additional functionality.

Database Paging

Many applications use databases which are too large to fit in main RAM memory or even a 32-bit virtual address space so portions of the database must reside on disk. The avoidance of distracting pauses when loading from disk requires a quick-loading database format as well as run-time logic which anticipates the viewpoint so the toolkit can begin paging database regions before they come into view.

Traversals

While the current 3-process rendering pipeline (APP, CULL, DRAW) is adequate for most applications, some require extensive application and cull processing. The addition of an APP traversal would allow user callbacks to be invoked each frame to control object behavior or trigger activity outside the toolkit. And currently, each pipeline's CULL traversal is restricted to a single process. Implementing parallelized traversals for both APP and CULL, where multiple processes concurrently carry out the same traversal, would improve throughput for both. The strict top-down inheritance of state in the scene graph eases this task since multiple processes can traverse individual subgraphs without requiring state information from other subgraphs. However, load balancing issues and allowing APP processing to be conditional on the results of visibility and level-of-detail computations are problematic since these computations are currently made *after* APP processing.

Collision Detection

While intersecting with line segments is useful for terrain following and simple collisions, collisions between objects of substantially different sizes and more detailed interference checking can require very large numbers of segments for adequate spatial coverage. Graph-to-graph intersections of volumes, geometry, and line segments represented by nodes within the scene graph would greatly benefit applications such as MCAD.

5 Conclusions

In this paper, we have presented a toolkit with a novel architecture for building high performance, multiprocessed graphics applications. We have described how the toolkit extracts maximal performance from multiprocessor, immediate-mode graphics workstations primarily through:

- geometric data structures designed for efficient immediate-mode data transfer
- reduction of graphics mode changes
- pipelined multiprocessing for parallel scene graph traversal
- efficient host-based view frustum culling
- stress modified level-of-detail switching
- run-time database and process statistics for tuning

By emphasizing immediate-mode performance without caching, the toolkit lends itself to techniques such as character animation and morphing which require intensive vertex-level modifications.

In the course of writing the toolkit, we developed a number of useful techniques for efficient task and data synchronization in a pipelined, multiprocessing system including a configurable software pipeline with update-driven multibuffering.

Without these performance optimizations, expensive hardware can be substantially underutilized. Since the optimizations described in this paper are non-trivial to implement, providing this functionality in a layered toolkit makes it substantially easier for application and other toolkit developers to reap significant performance benefits.

6 Acknowledgments

We would like to thank both present and past IRIS Performers: Michael Jones, Sharon Fischler, Chris Tanner, Allan Schaffer, Rob Mace, Ben Garlick and especially Craig "Crusty" Phillips and George Kong for their contributions. We would also like to thank Wade Olsen for the video application in Figure 16, Computer Arts and Entertainment of Madrid for the race simulator in Figure 15, Angel Studios and GreyStone Technology for Figure 13, and Paul Mlyniec of Software Systems for Figure 17.

7 References

1. Akeley, Kurt. Reality Engine Graphics. Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, 109-116.
2. Craig, Edgar Phillip. Micropoly 2 vs. Stargazer: the Search for an Exoteric Image Generator. *Proceedings of the 1996 Image VIII Conference,* Dallas, Texas, 26-29 June, 1996. 370-371.
3. Ferguson, Robert, et al. Continuous Terrain Level of Detail for Visual Simulation. In *Proceedings of the 1990 Image V Conference, Phoenix, Arizona, 19-22 June, 1990*, 144-151.
4. Funkhouser, Thomas and Carlo Sequin. Adaptive Display Algorithms for Interactive Frame Rates During Visualization of Complex Virtual Environments. Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, 247-254.
5. Grimsdale, Charles, dVS - Distributed Virtual Environment System. In *Proceedings of Computer Graphics '91 Conference*, London, 1991.
6. Hewlett-Packard Company, *Starbase Graphics Techniques and Display List Programmer's Guide,* Hewlett-Packard, Fort Collins, Colorado, 1991.
7. Kaplan, Michael. The design of the Doré graphics system, *Advances in Object-Oriented Graphics I, Konigswinter, Germany, 6-8 June 1990.* Springer-Verlag, 1991. 177-198.
8. Kawalsky, Roy, *The Science of Virtual Reality and Virtual Environments,* Addison-Wesley, Wokingham, England, 1993.
9. Neider, Jackie, Tom Davis and Mason Woo, *OpenGL Programming Guide*, Addison-Wesley, Reading, Mass, 1993.
10. Paradigm Simulation Inc., *VisionWorks Programming Guide*, Paradigm Simulation, Dallas, Texas, 1992.
11. Segal, Mark, et al. Fast Shadows and Lighting Effects Using Texture Mapping, Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In *Computer Graphics* 26,2 (July 1992, 249-252.
12. Strauss, Paul and Rikk Carey, *An Object-Oriented 3D Graphics Toolkit*, Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, 341-349.
13. van Dam, Andries, et al., PHIGS+ Functional Description Revision 3.0, *Computer Graphics* 22, 3 (July 1988), 124-218.
14. Ward, Mark, et al. A Demonstrated Optical Tracker with Scalable Work Area for Head-Mounted Display Systems, Proceedings of 1992 Symposium on Interactive 3D Graphics (Cambridge, Massachusetts, March 29 - April 1, 1992), 43-52.

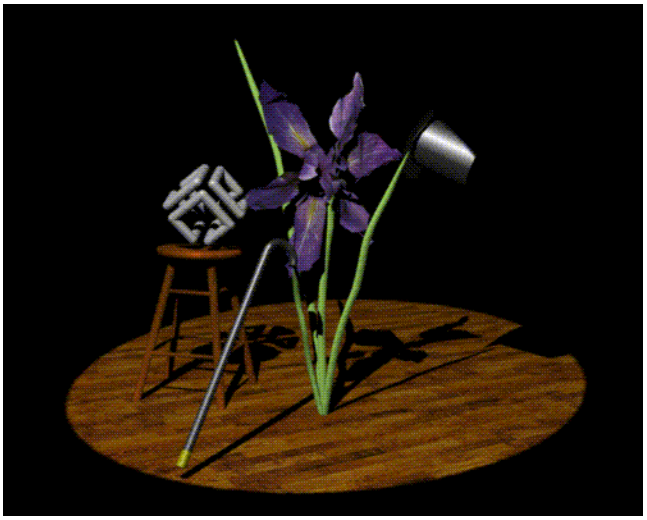


Figure 1. Real-Time Shadows Using Multipass Rendering

Figure 2. Precomputed and Dynamic Geometry Animations

Figure 3. Visual Simulation Scene

Figure 4. Racing Simulator with Collision Detection

Figure 5. Video Special Effects Using Draw Callbacks

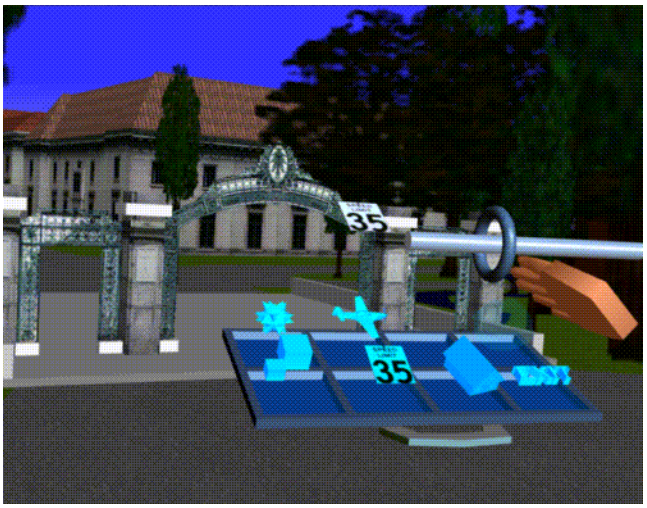


Figure 6. Fly Through with Virtual Reality Interface

Paper Reprint: The Silicon Graphics 4D/240GTX Superworkstation

Kurt Akeley
Silicon Graphics Computer Systems

Designing Real-Time 3D Graphics for Entertainment
SIGGRAPH '96 Course

Although this paper was published in 1989, the discussions of task partitioning and stages in rendering pipelines transcend the particular hardware implementation and still provide useful background for anyone seeking to maximize hardware utilization while constructing software frameworks or designing graphics databases for real-time rendering.

Published in IEEE Computer Graphics and Applications, July, 1989.

1 Introduction

A new class of machines, known both as superworkstations and graphics supercomputers, has become available. These workstations provide computing performance previously available only on supercomputers, and graphics performance at levels never before available. Their spectacular graphics performance is due both to advances in hardware technology and to tight integration between graphics and CPU subsystems.

Although superworkstation architectures share a commitment to graphics performance, and all evidence tightly integrated graphics subsystems, they differ fundamentally in their approaches to the partitioning of graphics related computations, specifically in their support of the *transformation* process. These different partitionings, while seemingly new, are in fact well known options to designers of high-performance graphics systems.

This paper presents a model that differentiates graphics architectures based on graphics computation partitioning, and places superworkstation architectures within this framework. It then presents the reasoning that resulted in the architecture of the Silicon Graphics 4D/240GTX superworkstation. Finally, the architecture and performance of the 4D/240GTX are discussed.

2 Architectural Alternatives

Both traditional and contemporary high-performance graphics systems use parallelism to achieve their performance goals[1-3]. Fortunately, graphics computations are easily partitioned into functional tasks, which can then be assigned to multiple processors.

Graphics Task Partitioning

All programmable, geometry-based raster graphics systems implement five fundamental tasks:

1. *Generation*. Generate (create, acquire, or modify) graphics data. Organize these data into a structure.
2. *Traversal*. Traverse the graphics data structure, delivering graphics data to the appropriate processor(s).
3. *Transformation*. Transform graphics data from object-space coordinates into eye-space coordinates. Do whatever shading operations are appropriate in eye-space. (Lighting calculations are an example.) Transform eye-space coordinates to clip-space, clip these coordinates, and project the resulting coordinates to screen-space.
4. *Scan Conversion.* Render the resulting screen-space objects (points, lines, and polygons) into raster memory. Iterate parameters as appropriate throughout this process. Do whatever shading operations are appropriate in screen-space.
5. *Display.* Scan the resulting pixels out of raster memory to a display monitor.

Because these tasks are executed in order (*transformation* precedes scan conversion, etc.) they are always implemented as a pipeline of processes. The choice of which tasks are executed on the application-programmable CPU, and which on special-purpose graphics processors, differentiates high-performance graphics architectures.

Display is a simple but performance sensitive task. It is simple and regular because raster data are always transferred to the monitor in scan-line order, whether from a single raster or from a small set of sub-screen rasters. It is per-

formance sensitive because monitors must be provided complete image data at least 60 times per second if irritating flicker is to be avoided. Thus a 1280x1024 pixel color monitor, typical of superworkstations, demands data at the sustained rate of 236 Mbytes/sec (1280 x 1024 pixels/frame x 60 frames/sec x 3 bytes/pixel). While this data rate would significantly burden any general-purpose processor, its regularity allows support to be provided in the form of a simple, special-purpose processor. For these reasons the *display* task is always executed on such a processor. (Systems that scan convert into non-displayable memory require an additional copy operation. This copy may be supported by an application-programmable processor.)

Generation, while more forgiving of performance variation, is demanding in its algorithmic complexity. Interactive simulations, for example, generate graphics data by culling and selecting object models based on viewer position and direction. Stress analysis requires that new colors be generated for each vertex in a mechanical model. Position and length of each bond are changed in the model of a molecule. Multiple windows, each with its own specific requirements, are maintained simultaneously. Because *generation* tasks are complex, widely varied, and application specific, they are always executed on the application-programmable processor.

Traversal, transformation, and scan conversion, however, have all been successfully implemented both on application-programmable processors and on special-purpose processors. Some examples will clarify these alternatives.

Architecture Examples

Bitmap workstations such as the Xerox Alto[4] and the original SUN systems supported generation, *traversal*, and *transformation* (typically in just 2 dimensions), completely with their application-programmable processors. While some hardware support was provided to increase the performance of scan conversion (BitBlt microcode on the Alto, a 32-bit barrel shifter on the SUN workstation) this hardware was driven directly by code executing on the application processor. Only the *display* task executed on a separate, special-purpose processor. We refer to such an architecture as GTXS-D, indicating that generation, traversal, transformation, and scan conversion (GTXS-) execute on the application-programmable processor, *display* (-D) on a separate, special-purpose processor.

Tradition host-terminal implementations, on the other hand, offer a very different assignment of graphics tasks. The DEC VAX/E&S PS-390 combination, for example, supports only the *generation* task on the application-programmable VAX processor. Traversal, transformation, scan conversion, and *display* are all executed on special-purpose processors within the PS-390 terminal. While some of the PS-390 processors are user programmable, their special-purpose

design clearly distinguishes them from the general-purpose, application-programmable VAX. We refer to such host-terminal architectures as G-TXSD, indicating that only the *generation* process executes on the application-programmable processor.

Many contemporary systems, including superworkstations, shun the architectural extremes of GTXS-D and G-TXSD. Graphics accelerator boards that include display, scan conversion, and sometimes *transformation* processors are widely available for machines ranging from personal computers to engineering workstations. With the addition of such accelerators, GTXS-D systems are modified to become either GTX-SD or GT-XSD systems.

While superworkstations have sufficient application-programmable CPU power to generate, traverse, transform, and scan convert with surprisingly high performance, all currently support one or more of these tasks with special-purpose processors. Superworkstations available from Apollo, Ardent, and Stellar implement variations of GTX-SD architectures. All Silicon Graphics workstations, including the 4D/240GTX superworkstation, are variations of GT-XSD architectures.

The assignment of graphics tasks to special-purpose processors is a compromise of factors such as cost-performance, generality, configurability, and user coding style. Before investigating the merits of various partitionings as they pertain to superworkstations, we simply observe that all of the alternatives have legitimate application. Examples of each recur as graphics architectures evolve[5].

3 The Silicon Graphics Task Partitioning

As we have seen, the issue facing a graphics system designer is not *whether* to partition graphics tasks among various processors, but rather *how* to partition these tasks. Our choice to support *generation* and *traversal* on our application-programmable CPU, *transformation*, scan conversion, and *display* in dedicated hardware, was made six years ago when our first workstation product was developed. Since that time the internal details of both the CPU and of the dedicated graphics systems have changed dramatically. Application-programmable CPUs have gone from 0.5 MIPS CISC machines (68010) to 20 MIPS RISC machines (MIPS R3000), and from single to multiprocessor organizations. Dedicated graphics processors have evolved from hard-wired engines (such as the original Geometry Engine™) to the 40 MFLOP (sustained) programmable floating-point subsystem of the 4D/240GTX. But the fundamental GT-XSD task partition has remained unchanged.

What are the characteristics of GT-XSD task partitioning that allow us, and in fact encourage us, to continue to design with it? We consider both the options of increasing (G-TXSD) and of decreasing (GTX-SD) the number of graphics tasks executed on special-purpose processors.

Why not Traverse in Hardware?

The arguments against a G-TXSD partition boil down to just one issue: immediate mode graphics. Moving the *traversal* task from the application-programmable environment of the general-purpose CPU into dedicated graphics hardware offers the promise of increased performance, but at the price of imposing hard-wired traversal algorithms, predefined data structures, and clumsy modeling interaction. When the *generation* task is ignored (as it often is in graphics demonstration programs) machines with G-TXSD architectures shine. Their inflexible traversals and costly model changes do not affect performance, because simple traversals are adequate, and the model is not being changed. Thus host-terminal G-TXSD systems are capable of displaying complex models with rapid viewpoint changes, but offer limited support of either user imposed or computed changes to the model being viewed.

The VAX/PS-390 combination, often used in molecular modeling applications, offers a suitable example. Users of this system quickly learn that, while molecule stick figures can be quickly viewed from any direction, and in fact 'wiggled' to improve depth perception, changes to the structure of the molecule are not interactive. Thus individual molecules can be viewed, but studies of the interaction of neighboring molecules are limited.

It is sometimes thought that the limitations of host-terminal systems are the result only of insufficient bandwidth on the 'thin wire' connecting the host to the terminal. While low bandwidth does reduce interactivity, all G-TXSD systems, whether connected by an RS-232 line or a 500 Mbyte/sec backplane, remain limited by predefined *traversal* algorithms, fixed data structures, and clumsy, often slow, modifications to geometry. Only when *generation* and *traversal* execute on the same application-programmable processor (or processors) are these limitations overcome.

Immediate mode graphics is graphics driven entirely, i.e. *generated* and *traversed*, by an application program executing on the programmable CPU. While it imposes additional requirements on the dedicated *transformation* graphics hardware (section 5.1) it frees the graphics programmer from the constraints of predefined display lists, display list traversal, and from inefficient change mechanisms to model definitions. The *generation* task of an immediate mode molecular modeling program, for example, computes bond lengths and angles resulting from an interactive condition, then changes them by simply editing its own data structures. The subsequent *traversal* takes its data from the shared

data structures, effecting the changes to the image automatically. Alternately, a real time simulation *traversal* can modify its own operation, culling and making model detail decisions as it operates based on the actual time elapsed during the construction of a frame. A *generation* task could make these decisions only in the abstract, based on viewer position and direction. Ultimately, an immediate mode workstation performs well not only on 'canonical demo' programs, but also on **interactive** applications, applications that change both model data and traversal as a function of user input.

Before leaving the issue of application-driven *traversal*, it is important to consider the PHIGS graphics standard. PHIGS, and its proposed extension PHIGS+, define a retained-model graphics standard. *Traversal* of the graphics data retained in PHIGS structures is defined by the standard, and is therefore not under direct application control. Thus PHIGS offers the software architectural equivalent of a G-TXSD hardware architecture. Two points must be made:

1. Because PHIGS inherits the limitations of G-TXSD architectures, a high-performance system should offer an immediate mode graphics interface in addition to its PHIGS/PHIGS+ interface.
2. PHIGS and PHIGS+ are easily and efficiently implemented on a GT-XSD platform. An immediate mode graphics interface, on the other hand, cannot be efficiently implemented on a G-TXDS platform.

Why transform in graphics hardware?

Tight coupling between model generation and traversal is critical to the performance of superworkstation machines; it is a characteristic that is shared by all superworkstation architectures. These architectures differ, however, in their implementation of the *transformation* task. What is the case for doing *transformation* in special-purpose processor hardware? We consider the following issues:

1. Impact of graphics requirements on CPU architecture.
2. Efficiency of hardware use during anticipated operation.
3. Configurability.
4. Feature/performance trade-off.

We will see that the first three considerations argue for special-purpose *transformation* processing, and that only the last argues against it.

Impact on CPU Architecture

The impact of including the *traversal* process on the general-purpose CPU is low. The CPU must be able to traverse data structures and transfer data at the

rate that they are consumed by the *transformation* process. While this rate is substantial in a modern workstation, and some hardware support in the CPU may be required to accommodate it, there is no need to radically alter the organization of the chosen CPU.

Transformation, on the other hand, makes tremendous demands of its supporting hardware. The 4D/240GTX, for example, executes approximately 100 floating-point operations per transformed vertex, which it generates at the rate of 400,000 per second. Thus its dedicated transformation engine operates at a sustained rate of 40,000,000 floating-point operations per second (40 MFLOPS). This computation rate is simply not achievable using currently available general-purpose CPU technology. If it is to be sustained, either a special-purpose processor must be designed to support it (e.g. a Geometry Engine™), or the application-programmable CPU must itself be designed as a special-purpose system (e.g. a vector processor). Such a special-purpose CPU will compromise performance across the wide variety of non-graphics application programs. It may also become separated from the mainstream of CPU technology, increasing the difficulty of tracking future hardware advances.

Efficiency of Hardware Use

It would seem that cost-performance is optimized when no hardware in the machine goes unused. Thus dedicated graphics hardware, when graphics computations are not being performed, is a poorly utilized resource. However, cost-performance is also reduced when hardware systems are underutilized. A single-precision graphics MFLOPS can be supported in special-purpose hardware at a substantially lower cost than it can be in the (typically double-precision) floating-point hardware of an application-programmable processor. The ratio on the 4D/240GTX, for example, is greater than 10:1. Thus, while graphics hardware in a GT-XSD machine is poorly utilized when graphics computations are not being performed, CPU hardware in a GTX-SD machine is poorly utilized when graphics computations *are* being done. Again, the choice is not whether to compromise, but how.

GTX-SD workstations compromise simultaneous computation and graphics performance because the graphics *transformation* process consumes a disproportionate, often total, fraction of the available application processor(s). When computation and graphics performances are specified independently, they create a misleading impression of GTX-SD interactive capability. The GT-XSD 4D/240GTX, on the other hand, efficiently executes computationally intensive applications and real-time 3D geometric graphics simultaneously. *Traversal* at full graphics performance consumes only a fraction of the power of 1 of its 4 application processors. Thus over 3/4 of its CPU performance is available for *generation* and other computation tasks.

Stress analysis, for example, is a computationally intensive task. The 4D/240GTX can apply almost all of its CPU performance to this problem, utilize its immediate mode graphics capability to update the display model, and support simultaneous viewing of a constantly re-colored image.

Configurability

Because only a small fraction of the 4D/240GTX CPU power is required to support full-performance graphics, its graphics and CPU performance are effectively decoupled. Simply removing the special-purpose graphics processors creates a powerful and efficient application engine (4D/240S). Alternatively, application processors can be removed (2-processor 4D/220GTX) or run at lower clock rates without affecting graphics performance.

The 16 MHz, 2-processor 4D/120GTX, for example, provides a cost-effective solution in embedded simulation environments. While the *generation* task is typically shared between the 4D/120GTX and another controlling processor, reducing the computational requirements on the superworkstation, full graphics performance, including *traversal*, is still required.

Feature/Performance Trade-off

The disadvantage of special-purpose hardware is its reduced flexibility. We expect a more rigid feature/performance trade-off from a dedicated graphics processor than from an application-programmable processor. Indeed, for the same reasons that special-purpose application processors compromise performance when executing general-purpose algorithms, dedicated special-purpose processors also operate best when executing their intended algorithms. Also, unlike special-purpose application processors, dedicated processors are inconvenient, or simply impossible, to program from the application level. Thus a dedicated *transformation* processor must be provided by its manufacturer with a wide variety of graphics algorithms if it is to exhibit gradual feature/performance trade-offs.

The single-precision floating-point *transformation* processor used in the 4D/240GTX, while not application programmable, is driven by RAM-based microcoded controllers. Each software release adds features and capabilities to its large initial set. Although its hardware design anticipated some specific requirements of the algorithms for which it was tuned, it has proved to be readily adaptable, and its performance on new and original algorithms is comparable.

Our recent implementation of non-uniform rational B-spline (NURBS)[6] provides a useful example. NURBS generation is both algorithmically complex and computationally intensive. By implementing the (complex) trimming algorithms in software on the application-programmable processor, and the float-

ing-point intensive mesh expansion in the special-purpose *transformation* processor, we achieved high performance on a feature that was not intended when the *transformation* processor was designed.

Summary

The advantages of application-programmable traversal and immediate mode graphics compel us to keep the *traversal* process located on the CPU. We have chosen the advantages of CPU independence, cost-effective simultaneous computation and graphics operation, and configurability demonstrated by a dedicated *transformation* processor over the potential benefits of feature/performance trade-off exhibited by CPU *transformation* machines. This given, we have designed our dedicated *transformation* engine to deliver high performance across a broad range of graphics algorithms.

4 Computing System Architecture

As we saw previously, our implementation of a dedicated *transformation* engine allowed us to choose our CPU architecture without consideration of special-purpose graphics computation requirements. Given this freedom, we have implemented a parallel-scalar multiprocessor subsystem. This general-purpose architecture performs extremely well across a wide variety of both scalar and vector applications. Figure 1 is a block diagram of the major components of this subsystem.

Key to both the performance and the efficiency of the processor system is a hierarchy of busses, each tailored to a particular function. The sync bus provides high-speed synchronization between the four application processors in support of fine-grained parallelism. Processor busses support full-speed data and instruction transfer between individual processors and their first-level instruction and data caches. The MPlink bus protocols support consistent data sharing between processors, as well as high-speed block data transfer between processors, the memory and I/O subsystem, the graphics subsystem. Processors communicate to the MPlink bus through individual read/write buffers and second-level data caches, both increasing the efficiency of MPlink bus utilization and supporting the data consistency operations.

Sync Bus

The sync bus was designed to meet the synchronization needs of a multiprocessor supporting efficient fine-grained parallelism. Single application programs can make efficient use of parallel processors at the individual loop level.

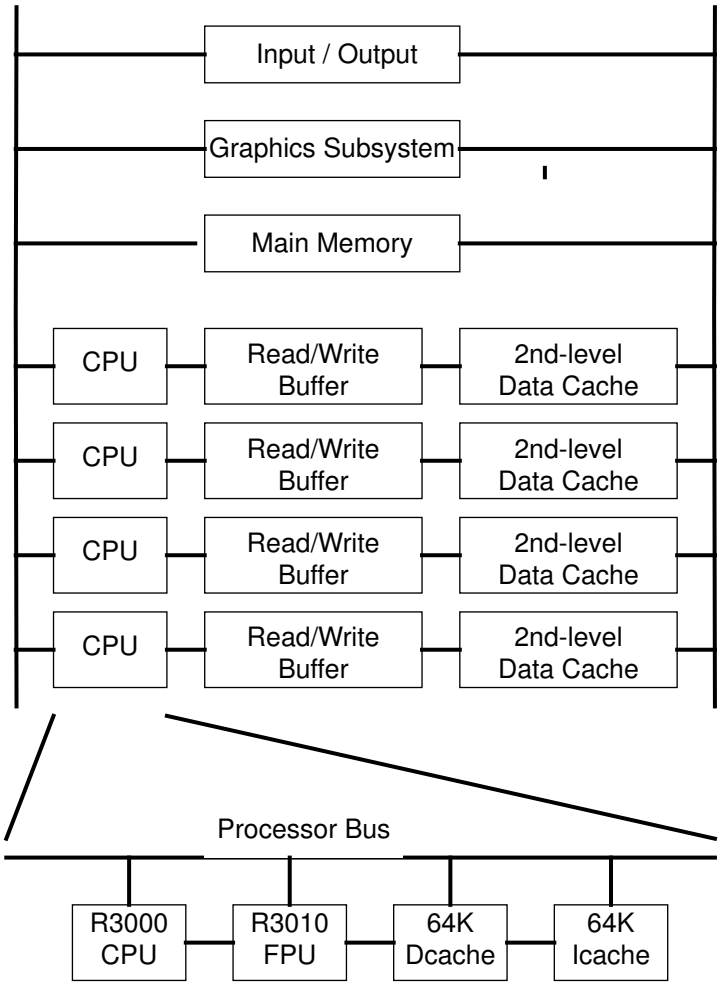


FIGURE 1. System block diagram.

Large-grain parallelism, such as consistent access of shared data structures between *generation* and *traversal* tasks, is supported as well.

The sync bus provides 64K individual test-and-set variables, which are accessed in the physical address space of each processor. These variables can be mapped directly into the virtual address space of a user process, supporting application synchronization with overhead measured only in fractions of microseconds. As a result, many programming and compiler techniques developed for vector processors are also suitable for the scalar multiprocessor system. For example, strip mining, the technique of taking a long vector and breaking it into a number of strips for a vector register, is implemented by providing data strips to each of the 4 processors.

IRIX™, the Silicon Graphics implementation of UNIX™ System V.3., is itself a parallel, fully-symmetric multiprocessing application. Several thousand test-and-set variables are used by the system to provide fine-grained locks for its control variables. IRIX also benefits from the sync bus interrupt facility, which supports both directed and broadcast interrupts. The flexible interrupt distribution scheme allows process scheduling to support, rather than disrupt, the private first-level caches.

Processor Bus

The RISC CPU, floating-point coprocessor (FPU), and first-level instruction and data caches of each processor are connected by the processor bus. Each 32-bit processor bus operates at twice the 25 MHz CPU clock rate, sustaining instruction and data transfer at up to 200 Mbytes/sec. Because each CPU includes its own processor bus, aggregate processor bus bandwidth scales directly with the number of CPUs in a system. Processor-to-cache transfer is sustained at up to 800 Mbytes/sec in the 4-processor 4D/240GTX, 400 Mbytes/sec in the 2-processor 4D/220GTX.

First-level instruction and data caches each store 64K bytes, resulting in a 4-processor aggregate first-level cache size of 0.5 Mbytes. A read buffer and a 4-word deep write buffer isolate the first-level caches from the MPlink bus and its second-level data caches. In addition to improving performance, these buffers provide an asynchronous interface between the 25 MHz processors and the 16 MHz MPlink bus. Processor clock speed can be increased to improved performance, or decreased to reduce cost, independent of the system bus frequency.

MPlink Bus

The MPlink bus supports the system's cache consistency protocol and provides block data transfer between the four processors, the memory and I/O subsystem, and the graphics subsystem. Second-level 256 Kbyte data caches, one associated with each processor, are best thought of as components of the MPlink bus. Operating as a single, synchronous unit, the four second-level data caches and the MPlink bus monitor each block data transfer. Together, using a convention known as the Illinois Protocol[7], they ensure that only valid data are made available to the four processors.

Because bus monitoring is a single, synchronous operation, the 64-bit data path of the MPlink bus can be efficiently utilized. The peak data rate for this bus is 128 Mbytes/sec (8 bytes/clock x 16 clocks/us). A data rate of 64 Mbytes/sec is sustained, implying an overhead of only 50 percent for both bus monitoring and cache-line updating operations.

Graphics Support

Each processor includes hardware to support efficient transfer of graphics data to the *transformation* engine. The transfer is referred to as a 3-way transfer because:

1. The application processor initiates the transfer, after it completes the virtual-to-physical address translation of the first word of a short (2 through 4) word data vector,
2. The memory subsystem provides the specified data, and
3. The graphics system accepts the data.

While 3-way transfer can be thought of as a DMA operation, its atomic nature (the 3 steps cannot be interrupted) and integral address translation make it far more efficient than a typical DMA implementation. We prefer to think of the short data vectors as additional data types for which the hardware has been tuned, just as double-precision floating point numbers are constructed of two 32-bit words on many machines. A 3D coordinate, for example, is a vector of 3 floating point values, representing x, y, and *z* in object space. Data types include 2, 3, and 4 dimension coordinates, vertex normals *(nx,ny,nz)*, and 3 and 4 component colors *(r,g,b,a)*.

The 3-way transfer hardware, which is added without modification to the RISC integrated circuits, allows the Silicon Graphics immediate mode graphics library to transfer randomly located vertex data at the rate of over 400,000 position/normal pairs per second. Graphics data can be stored in an application-defined structure, and traversed with application code, at the full transformation rate. Because graphics library subroutines are called indirectly through a shared library interface, programs compiled for the 4D/240GTX can be run on any 4D product without recompilation or relinking, and without performance loss.

Performance

MIPS R3000 CPU and R3010 FPU RISC components are used in each of the 4 processors. Running at 25 MHz, each of the 4 processors executes approximately 20 million instructions per second (MIPS), and achieves 4 double-precision LINPACK MFLOPS, Their performance operating as a system, however, cannot be so easily reduced to two numbers.

One useful performance measure is throughput, the rate that *n* equivalent processes execute on an *n*-processor system. Both Dhrystone and LINPACK[8]

benchmark results document that the 4D/240GTX delivers near-linear throughput performance improvement.

	1 processor/ 1 program	4 processors/ 4 programs	Ratio
Dhrystone	37,400	149,085	3.98
(VAX MIPS)	21.3	84.8	
LINPACK	4.0	15.3	3.83

LINPACK numbers are for 100 x 100 double-precision data, measured with coded BLAS. Dhrystone VAX MIPS are related to the base numbers by a factor of 1760. Large, write-back second-level data caches, ample I/O capability, and careful attention to reduction of system overhead all contribute to near-linear throughput performance. The fully semaphored operating system, which executes as a distributed process rather than in master/slave mode, allows the 4 processors to make balanced throughput contributions. (Semaphores block less than 0.1 percent of the times they are encountered.)

Near-linear throughput performance, though certainly not achieved by all multiprocessor systems, is more easily accomplished than near-linear performance increase of a single process as it is distributed across the *n*-processor system. Unfortunately, the performance of a single application distributed across all available processors, a very significant measure of a multiprocessor system, is not well represented by benchmarks in use today. An indication of the performance delivered by the 4D/240GTX is provided by the table below:

	1 processor/ 1 program	4 processors/ 4 programs	Ratio
LINPACK	6.6	16.6	2.56
BRL RTFM	1360	5034	3.7

LINPACK numbers are for 1000 x 1000 double-precision data, measured with coded BLAS. BRL RTFM is the Ballistic Research Lab's ray tracing figure of merit, an application designed to test parallelization capability. Hardware cache consistency, distributed interrupts, synchronization supported by a dedicated bus, and an automatic parallelizing compiler all contribute to the support of 1->*n* parallel execution.

5 Graphics Architecture

The graphics system of the 4D/240GTX is itself partitioned into four subsystems, corresponding almost exactly to the task partitions described earlier in this paper. Figure 2 is a block diagram of the entire graphics system.

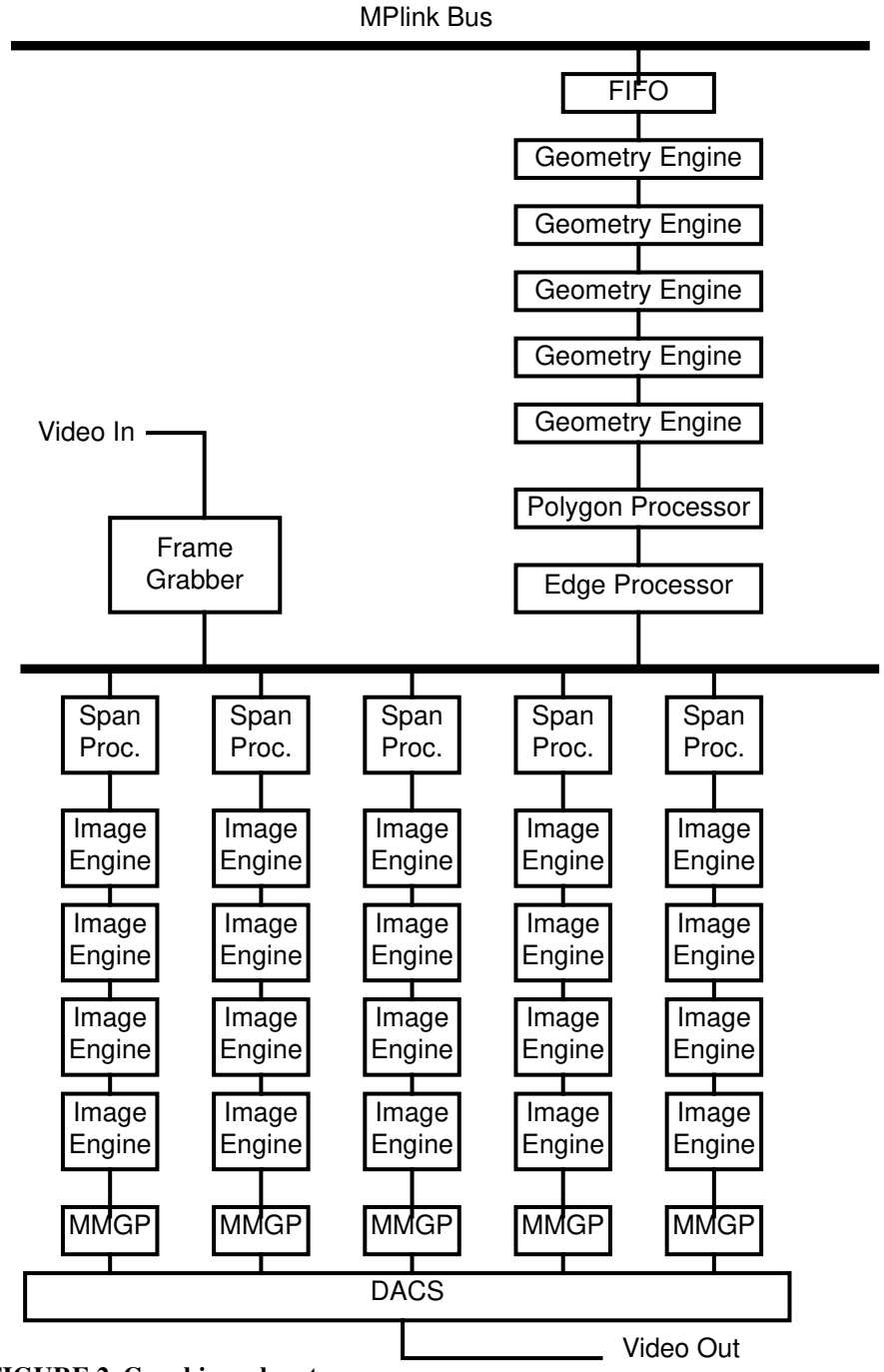


FIGURE 2. Graphics subsystem.

Transformation Subsystem

Vertex *transformation* is a very demanding task in an interactive 3D environment. Roughly 100 single-precision floating-point operations are required to

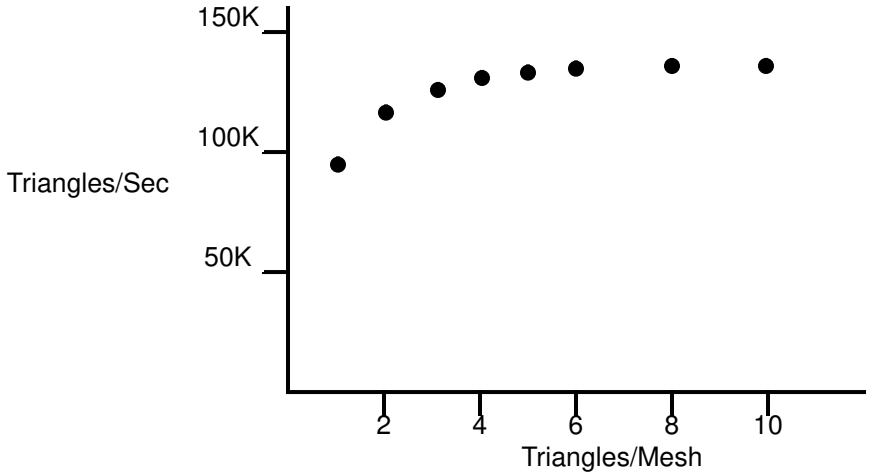


FIGURE 3. Triangle mesh performance.

transform, light, clip test, project, and map an object-space vertex to screen-space[9].

Operation	FLOPS
Vertex transformation	28
Normal transformation	15
Single-source lighting	28
Clip testing	6
Projection	11
Map to screen-space	9
Total	97

The 4D/240GTX dedicated transformation subsystem, called the Geometry Engine™, processes vertexes at the rate of 400,000 per second, sustaining a computation rate of 40 MFLOPS.

Careful attention to the flow of both data and control in the design of the Geometry Engine ensure that polygonal and mesh boundaries do not substantially affect performance. Thus full transformation performance is available not only to large blocks of contiguous vertexes, but also to the vertexes of independent polygons and of short triangle meshes. Both independent triangles and independent quadrilaterals, for example, are drawn at the full 400,000 vertex per second rate (100K quadrilaterals/second, 135K triangles/sec). Meshes of only 5 vertexes (3 triangles), draw at over 90 percent of the peak triangle mesh rate (also 135K, see Figure 3).

A workstation that is to support immediate mode graphics (with the desirable properties described in previous sections) must achieve full transformation performance on small data sets, as well as support both *generation* and *traversal* on its application-programmable processor. The *transformation* proces-

sor of the 4D/240GTX, in conjunction with the 3-way transfer hardware of the processor system, meets this requirement. Special-purpose application processors in GTX-SD machines sometimes do not.

The Geometry Engine comprises a single conversion and FIFO module, followed by 5 identical microcoded floating-point processors. The 6 processors are organized as a single pipeline. Each executes a specific subset of the standard *transformation* algorithm.

Coordinate and color data are accepted by the conversion module in single and double-precision floating-point, integer, and packed integer formats. All data are converted to single-precision IEEE floating-point format before being transferred to the first floating-point processor. A 512-word deep FIFO precedes the conversion module. The processor that is currently executing the *traversal* process is interrupted whenever the FIFO becomes more than half full. It stalls until the FIFO empties past a low-water mark, then returns control to the application graphics program. Thus graphics flow control adversely affects *traversal* performance only when the allowable transfer rate is exceeded.

The floating-point processors are identical modules, each consisting of a 20 MFLOPS peak data path component (FPU) and a RAM-based microcode sequencer. Data and control transfer between the processors is arranged to avoid lost FPU cycles. Transformation tasks are distributed among the 5 processors as follows:

1. Matrix and normal transformation. Coordinate matrix and normal (inverse-transpose) matrix stack maintenance. Normal normalization.
2. Lighting calculations.
3. Clip testing.
4. Perspective division. Clipping (when required).
5. Viewport transformation. Color clamping to a maximum value. Depth-cue calculations.

Scan Conversion Subsystem

Output data of the Transformation Subsystem specify the vertexes of points, lines, and polygons in screen coordinates. The Scan Conversion Subsystem performs the calculations required to reduce these vertex data to individual pixels. Each pixel is assigned an x, y, and *z* coordinate and an r, g, b, and a color value. Color and *z* data are interpolated linearly between vertexes and between the edges of polygons.

Polygon scan conversion is partitioned across 3 separate processors within the Scan Conversion subsystem. The Polygon Processor decomposes polygons into screen-aligned trapezoids whose left and right edges are vertical. (Figure

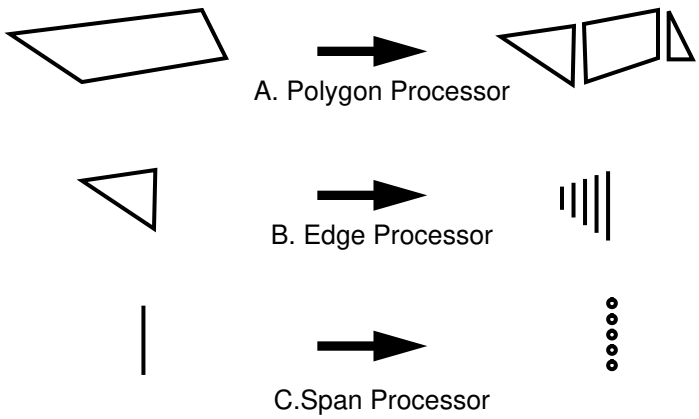


FIGURE 4. Polygon decomposition.

4A.) These trapezoids are themselves decomposed into vertical single-width spans by the Edge Processor. (Figure 4B.) Finally, the individual spans are reduced to pixels by an array of 5 Span Processors, each of which operates on a single span. (Figure 4C.)

Color and depth slopes are computed at each decomposition step, resulting in bilinear interpolation across the interior of the polygon[10]. The scan conversion is point-sampled, meaning that:

1. Only pixels whose exact centers are within the precise polygon boundary are filled, and
2. Color and depth parameters are iterated with subpixel precision, then corrected to correspond exactly to the (pixel centered) sample points. (Figure 5.)

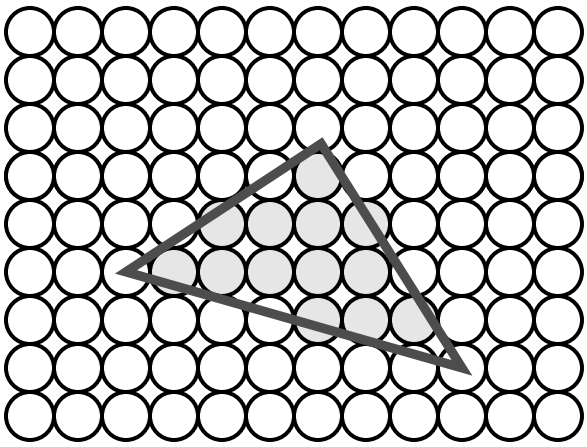


FIGURE 5. Point-sampled polygon.

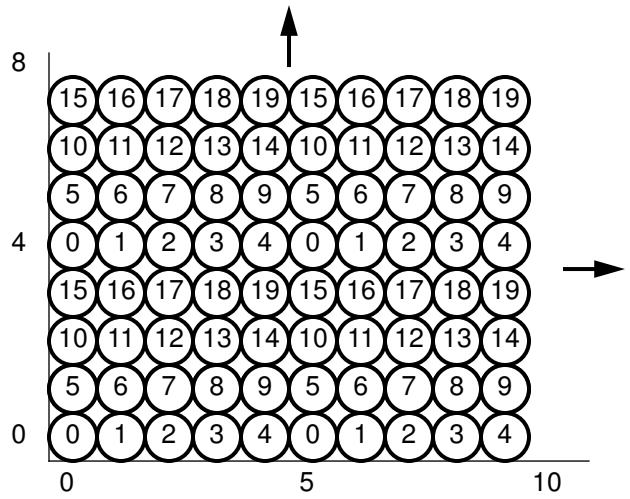


FIGURE 6. Image engine tiling pattern.

As a result, iterated color and depth values remain planar across polygonal surfaces, avoiding unnecessary color banding and yielding clean Z-buffer intersections. Also, adjacent polygons neither share nor omit pixels, properly supporting transparent renderings.

Polygon spans are generated at the rate of 2 million per second. Span processors reduce spans to pixels at the rate of 16 million pixels per second, resulting in an aggregate (5 Span Processor) fill rate of 80 Mpixels/sec. Line pixels are generated by the Edge Processor at the rates of 16 Mpixels/sec.

Raster Subsystem

The Raster Subsystem contains 20 Image Engines™, each of which is an independent state machine that controls 1/20th of the frame buffer memory. Groups of 4 Image Engines are driven by each Span Processor. The array of Image Engines tiles the frame buffer in a 5-wide, 4-high, pattern. (Figure 6.) This 2-dimensional interleaved tiling pattern is arranged such that any screen-aligned 5 x 4 region is supported by all 20 Image Engines. As a result, the Image Engines operate in parallel even when very small regions are being filled.

Bitplane memory is organized into 5 banks, with a total of 96 bits per pixel. The banks are arranged as follows (Figure 7):

- *Image banks.* Two banks of 32 bits each provide direct support of double-buffered images. Organized as 8 bits each of red, green, blue, and alpha data when used for RGB (true color) imaging, and as 12-bits each when used for pseudo color imaging.

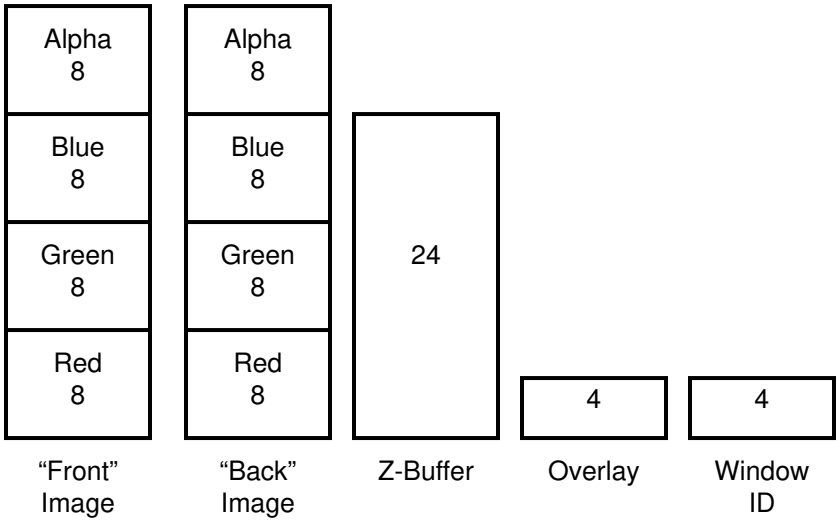


FIGURE 7. Bitplane assignments.

- *Depth bank.* One bank of 24 bits. Stores 24-bit integer depth information when used in conjunction with the Image Engine Z-buffer pixel algorithm. It is also available for image data.
- *Overlay bank.* One bank of 4 bits. Two bits are reserved for the window manager to display popup menus. The two remaining banks are available to application programs for both overlay and underlay images.
- *Window ID bank.* One bank of 4 bits, used by the window manager to tag pixels based on the drawing process to which they belong.

The 4-bit window ID field identifies each pixel as belonging to one of (up to) 16 currently active windows. All pixel algorithms except high-speed clear compare the contents of this ID field to the ID of the currently drawing window process. If the IDs do not match, no change is made to the current pixel values. Thus both partially obscured and non-rectangular windows are supported with no performance penalty.

Image Engines operate as specialized memory controllers, supporting video RAM refresh, display refresh, and a handful of pixel access algorithms. These algorithms include:

- *Replace.* Replace the destination color with the source color.
- *Z-buffer.* Compare the source and destination *z* values. If the test passes, replace the destination color and *z* with the source color and *z*, Any combination of greater than, equal to, and less than can be specified as the test condition.

- *Alpha blend.* Replace the destination color with a linear combination of the source and destination colors[11]. Each of the four color components is operated on with the following equation:

$$C_{dst} = \alpha F_{src} C_{src} + F_{src} C_{dst}$$

- Applications choose the component multipliers from the following lists:

$$F_{src} \in 0, 1, \alpha_{src}, 1 - \alpha_{src}, \alpha_{dst}, 1 - \alpha_{dst}, C_{dst}, 1 - C_{dst}$$

$$F_{dst} \in 0, 1, \alpha_{src}, 1 - \alpha_{src}, \alpha_{dst}, 1 - \alpha_{dst}, C_{src}, 1 - C_{src}$$

  The subscripts *src* and *dst* refer to the incoming and current color values respectively.
- *High-speed clear.* Simple replace available only for large, screen-aligned rectangles.

Although the Image Engines are simple machines, their parallel operation and multiple algorithms result in extremely powerful pixel fill operation. Aggregate performance for the various pixel algorithms is:

Pixel Algorithm	Fill Rate Mpixel/sec
Replace	80
Z-buffer	40
Alpha blend	16
High-speed clear	160

Display Subsystem

The Display Subsystem receives pixel data from the frame buffer, interprets it, and routes the resulting red, green, blue, and alpha data to the Digital-to-Analog converters for display. Five Multimode Graphics Processors (MMGPs) operate in parallel, one assigned to the pixels controlled by each Span Processor. These MMGPs receive all 64 image bank bits, the 4 auxiliary bank bits, and the 4 window ID bits for each pixel. They interpret the image and auxiliary bits as a function of the window ID bits, using an internal 16-entry table. Thus windows can independently select single or double buffer operation, and double buffer windows can swap buffers independently. Color index (pseudo color) and RGB (true color) operation are also selected independently on a per-window basis.

Window ID interpretation allows complete window support in a system that scan converts into and displays from the same raster memory (i.e. frame buffer). Eliminating the need for copy operations between raster memory and display memory reduces system bandwidth requirements, allows double buffer

windows to be easily synchronized with monitor refresh, and supports 30 and even 60 Hz frame refreshes. Rapid frame rates are critical in real time simulation environments, and were an important consideration in the design of the 4D/240GTX.

Context Switching

The graphics subsystem is designed to support context switching with minimal overhead. Because significant quantities of state are accumulated in the 5 floating-point processors of the Geometry Engine, each maintains complete context for 16 independent processes in its local data memory. The floating-point processors are also able to dump and restore context to and from a host processor, allowing more than 16 processes to share the hardware. Thus a working set of up to 16 processes is supported, with essentially no limit to the total number of processes.

Because the Edge, Span, and Image Processors are unable to return state information, the few states stored in these processors are shadowed by the Polygon Processor. The Polygon Processor state, including shadow state, is minimal, and is therefore maintained by a host CPU.

Graphics System Features

Pan and zoom are implemented as high-speed pixel copy operations. Copy performance varies from 4 to 16 million pixels per second as a function of pixel zoom factor. A 1/4-screen zoom by a factor of 2, for example, runs at 7 frames per second. Smaller areas, common in window-capable systems, easily zoom at 30 frames per second. Because the effects of pan and zoom are limited to a single window, or to multiple windows with independent factors if desired, the full screen remains a useful resource.

An optional frame-grab board captures both NTSC and PAL video in real time. Images are transferred to an arbitrary window at the rate of 16 million pixels per second. Once in raster memory, these images are operated on and displayed just like geometry-based images. Multiple buffers in the frame-grab hardware ensure that synchronization is correct and that neither images nor performance are lost.

The Image Engines**TM** support alpha blending, allowing pixel color components to be replaced with a linear combination of their previous components and the incoming color components. Application programmers select a specific blending equation from a list of options (Section 5.3). Perhaps the most commonly used blending function is:

$$C_{dst} = \alpha_{src} C_{src} + (1 - \alpha_{src})\, C_{dst}$$

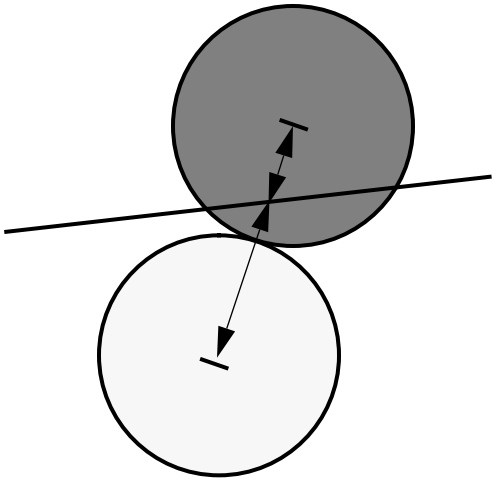


FIGURE 8. Antialiased line generation.

When α_{src} represents opacity of the incoming pixel value, the pixel is rendered transparently. Alternatively, when α_{dst} represents pixel coverage (perhaps computed by the antialiased line drawing hardware) the edges of the object being drawn are smoothed.

Antialiased lines, critical to molecular modeling applications, are drawn at up to 1/2 the rate of standard lines. Pixel coverage is computed from the fractional line position in the direction perpendicular to its primary direction. (Figure 8.) The computed coverage, in the range 0..1, is used to scale the a of each pixel in the line.

$$\alpha_{src} = \alpha_{coverage}\alpha_{src}$$

A blending function such as the one described above is used to merge the line into the frame buffer.

6 Summary

The 4D/240GTX represents state-of-the-art in superworkstation architecture. Its parallel-scalar CPU architecture supports an attractive performance balance of 80 MIPS and 16 MFLOPS. Graphics performance of over 100,000 lighted, Z-buffered quadrilaterals per second is provided by its tightly-integrated graphics subsystem. A complete set of development software, including a parallelizing FORTRAN compiler, user-level parallelizing directives, and the immediate-mode graphics library, allows application programs to easily

achieve optimal performance. Its GT-XSD architecture insures that full CPU and graphics performance are available simultaneously to application programmers.

7 References

1. R. Swanson and L. Thayer, "A Fast Shaded-Polygon Renderer," *Computer Graphics* 20(4), August 1986, pp. 95-101.
2. J. Torborg, "A Parallel Processor Architecture for Graphics Arithmetic Operations," *Computer Graphics* 21(4), July 1987, pp. 197-204.
3. B. Apgar, B. Bersack and A. Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000," *Computer Graphics* 22(4), August 1988, pp. 255-262.
4. C. Thacker, E. McCreight, B. Mapson, R. Sproull, and D. Boggs, "Alto: A Personal Computer," in D. Siewiorek, G. Bell, and A. Newell, *Computer Structures: Readings and Examples*, 2nd ed., McGraw Hill 1981.
5. T. Myer and I. Sutherland, "On the Design of Display Processors," *CACM* 11(6), June 1968.
6. W. Tiller, "Rational B-Splines for Curve and Surface Representation," *Computer Graphics and Applications*, September 1983, pp. 61-69. NURBS.
7. M. Papamarcos and J. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of 11th Int. Symposium on Computer Architecture*, IEEE, 1984, pp. 348-354.
8. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations in a Fortran Environment," Argonne National Laboratory, February 16, 1988.
9. K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics* 22(4), August 1988, pp. 239-246.
10. H. Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers* Vol. 20, No. 6, June 1971, pp. 623-629.
11. T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics* 18(3), July 1984, pp. 253-259.