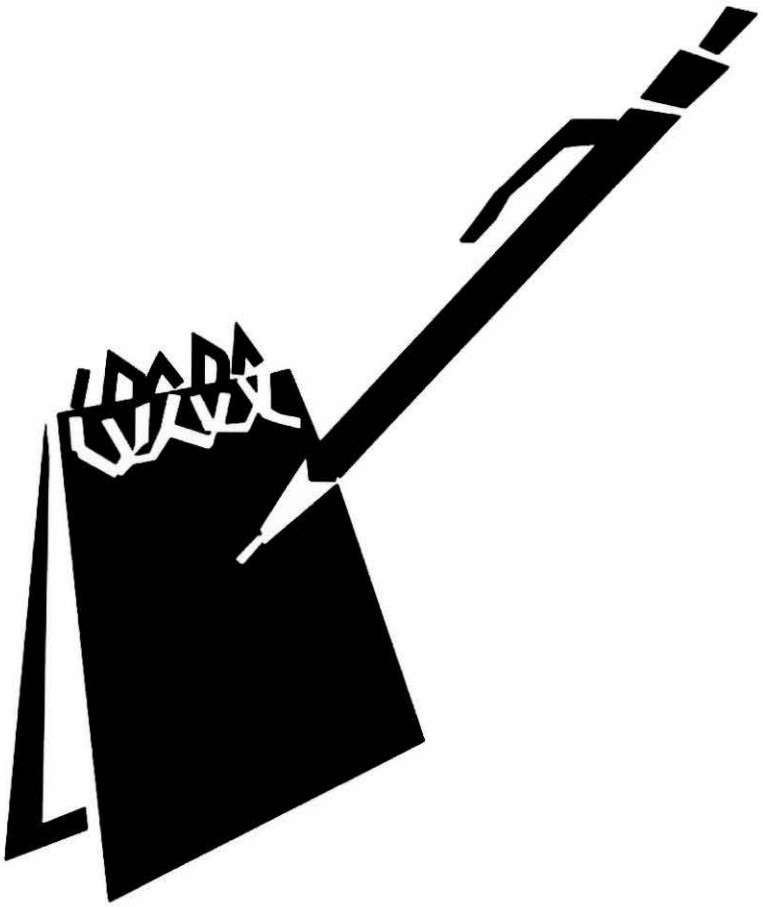
SIGGRAPH 98

# 29

# Developing High-Performance Graphics Applications for the PC Platform



**25th** International Conference on Computer Graphics and Interactive Techniques

Exhibition **21-23 July** 1998 Conference **19-24 July** 1998

**Orlando,** Florida USA

## course notes

# 29

# Developing High-Performance Graphics Applications for the PC Platform

*Organizers*
**Michael Cox**
MRJ Technology Solutions/NASA Ames Research Center

**David Sprague**

*Lecturers*
**John Danskin**
Dynamic Pictures

**Rich Ehlers**
Evans & Sutherland Computer Corporation

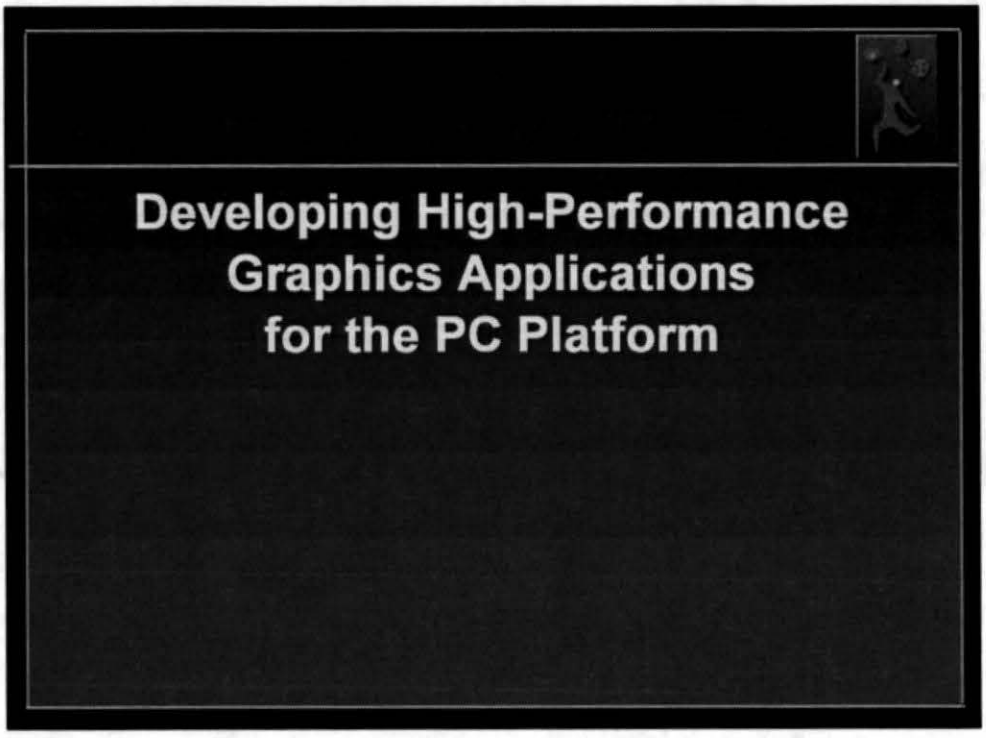
**Brian Hook**
id Software

**Bill Lorensen**
General Electric Corporate R&D Center

**Gary Tarolli**
3Dfx Interactive

**25th** International Conference on Computer Graphics and Interactive Techniques
Exhibition **21-23 July** 1998 Conference **19-24 July** 1998
**Orlando,** Florida USA

**course notes**


Developing High-Performance
Graphics Applications
for the PC Platform

# Course Overview

*Michael Cox, MRJ/NASA Ames*

*Dave Sprague, Intel Corp.*

## Motivation

***PC graphics capability has increased dramatically over the past 3 years***

- Hardware acceleration
- CPU Floating Point Performance

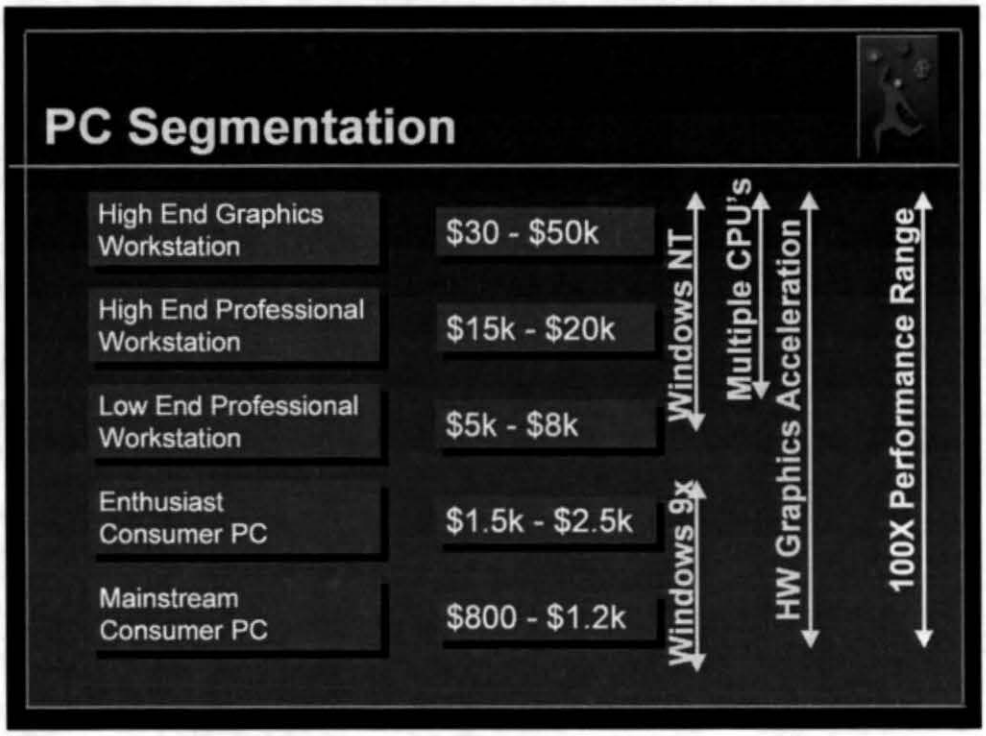
***PC's are now an important target for graphics intensive applications***

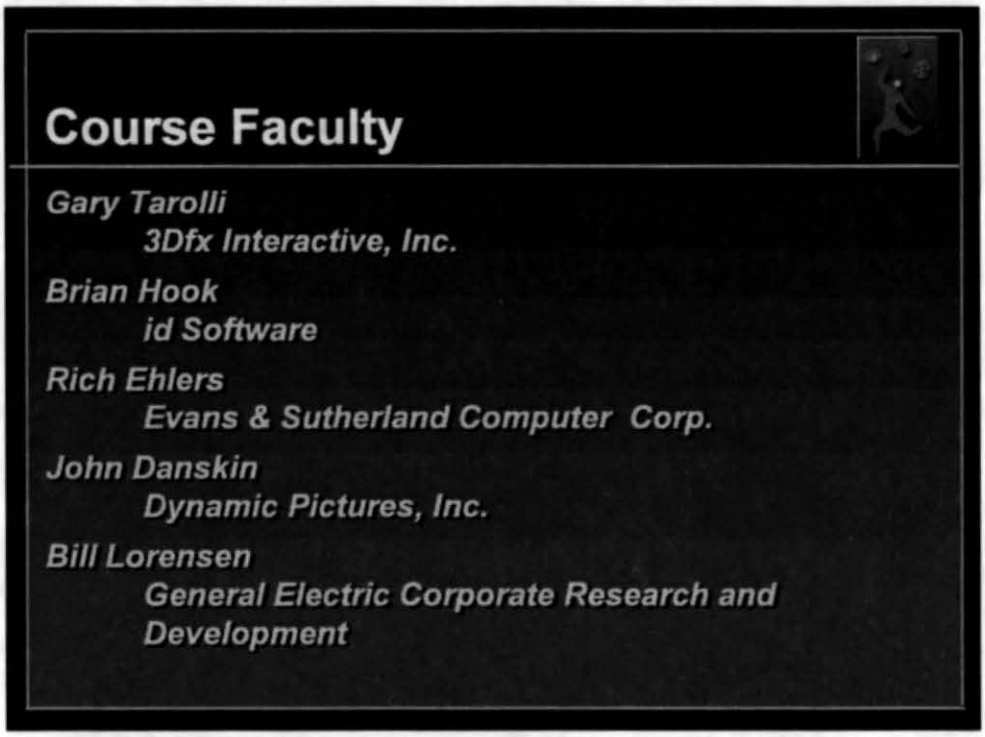
- Above threshold performance
- Compelling Price-Performance Points

## Course Objectives

***Identify Issues and Approaches for Developing High Performance Graphics Applications on PC Platforms***

- Architecture & Applications POV's
- HW Performance History and Future Trends
- How Real-World Apps are Developed
- Optimization Techniques and Rules of Thumb
- Scalability Issues


PC Segmentation
High End Graphics Workstation
$30 - $50k
High End Professional Workstation
$15k - $20k
Low End Professional Workstation
$5k - $8k
Enthusiast Consumer PC
$1.5k - $2.5k
Mainstream Consumer PC
$800 - $1.2k
Windows NT
Windows 9x
Multiple CPU's
HW Graphics Acceleration
100X Performance Range



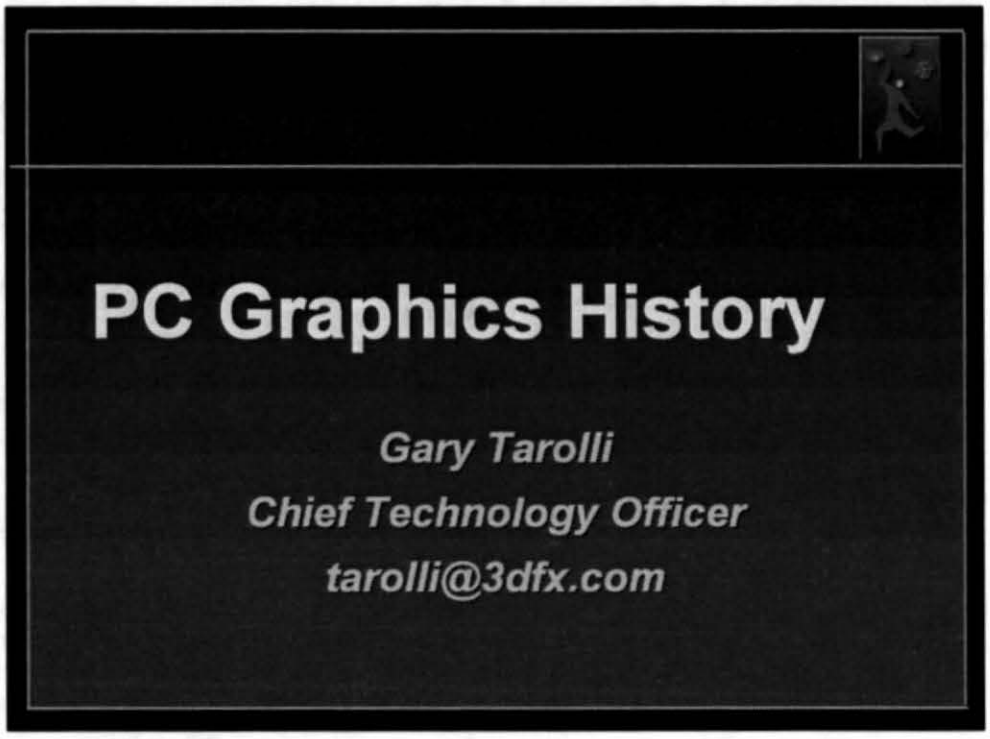

**Gary Tarolli** was a founder and is currently Senior Vice President and Chief Scientist at 3Dfx Interactive. He was one of the architects of the Voodoo Graphics chipset, a current high-end graphics accelerator popular among game and content developers. After receiving a B.S. from Rensselaer Polytechnic Institute and an M.S. from the California Institute of Technology, Gary has worked in the field of VLSI design and graphics for the last 16 years at DEC, SGI, Pellucid, Kubota, and now 3Dfx. In 1992 he was the second recipient of the Electronic Engineering Times "Celebrating the Engineer" award.

**Brian Hook** is a software engineer specializing in the field of 3D real-time graphics. He got his start working at 3Dfx Interactive, where he was responsible for designing and implementing the first version of Glide, a proprietary API designed to interface to 3Dfx's various chipsets. After leaving 3Dfx, he worked as a consultant with such companies as Trident Microsystems, nVidia, and Silicon Graphics. At nVidia he reviewed the design of the Riva-128 before its production, and at SGI he was part of their SGI OpenGL for Windows team. In mid-1997 he accepted a job with id Software where he worked on the completion of Quake II. He is currently working on the new technology that will be used ini their next-generation graphics engine code-named Trinity. Brian has also written on the subject of computer graphics and computer programming most recently as a columnist for Game Developer Magazine. Prior to joining id Software, Brian was a student at the University of Florida.

**Rich Ehlers** is Director of Graphics Software, Desktop Graphics at Evans & Sutherland. He has been a technical contributor and technical lead at E&S since 1982. He lead the investigation beginning in 1993 into the feasibility of putting graphics technology on the PC, and has since directed NT software driver development. Prior to work on the PC, he managed software development for the E&S Freedom series on Sun workstations, represented E&S on the PHIGS committee and was one of the original PHIGS architects, and worked on the software, microcode, and performance enhancements of numerous E&S graphics products. He holds a B.A. in CS and Physics from Luther College and an M.S. in Computer Science from Purdue University. He taught at a SIGGRAPH course in 1995, has served on the SIGGRAPH Course Committee (1988, '90, '92, '94) and has served as SIGGRAPH Courses Chair (1991, '93).

**John Danskin** designs graphics accelerators at Dynamic Pictures, where one of his major areas of focus is accelerator integration with the PC architecture. John has worked on software, hardware, and microcode for graphics acceleration since 1987 for such companies as CadTrak, Daisy Systems, DEC, and Dynamic Pictures. John is also on the faculty at Dartmouth College, where he is on leave of absence. At Dartmouth he has taught compression, computer graphics, and computer systems, and has a little over a dozen technical papers in several fields. John holds a B.A. in Computer Science from UC Santa Cruz, and a Ph.D. in Computer Science from Princeton University.

**Bill Lorensen** is a Graphics Engineer in the Electronic Systems Laboratory at GE's Corporate Research and Development Center in Schenectady, NY. He has over 25 years of experience in computer graphics and software engineering. Bill is currently working on algorithms for 3D medical graphics and scientific visualization. He is a co-developer of the marching cubes and dividing cubes surface extraction algorithms, two popular isosurface extraction algorithms. His other interests include computer animation, color graphics systems for data presentation, and object-oriented software tools. Bill is the author or co-author of over 60 technical articles on topics ranging from finite element pre/postprocessing, 3D medical imaging, computer animation and object-oriented design. He is a co-author of "Object-Oriented Modeling and Design" published by Prentice Hall, 1991. He is also co-author with Will Schroeder and Ken Martin of the book "The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics" published by Prentice Hall in February 1996. Bill holds nineteen US Patents on medical and visualization algorithms. In 1991, he was named a Coolidge Fellow, the highest scientific honor at GE's




Over the last several years improvements in CPU and platform architecture and the evolution of hardware-accelerated rendering have enabled high-performance graphics on the PC. The history of the CPU and the PC platform itself, as well as the history of PC hardware acceleration will be explored in this section, leading to a picture of the current PC architecture. Brief projections of the future will be painted, to be further explored in the panel at the end of the course.

## Who am I

***I spent many years at SGI***

- workstation graphics SW and HW
- SGI flight simulator
- Indigo Entry graphics

***co-founded 3Dfx Interactive in 1994***

- co-architect of Voodoo Graphics
- best selling 3D game accelerator chip

## Consumer vs. Workstation

***consumer graphics < $300***

***consumer PC < $3000***

***consumer graphics is typically***

- single chip (or single chip cost)
- limited memory and expandability
- limited windowing capabilities

The big difference between consumer graphics and professional/workstation graphics is cost , or budget. Consumer graphics work within a fixed budget and try to do as much of workstation graphics as can be done within that budget. Skimp on memory, number of rendering chips, etc.

## Timeline

| Date | CPU | GFX | Fab | Gates | Comments |
|---|---|---|---|---|---|
| <1992 | 486 | | .8 | 50k | not interesting |
| 1993 | P5-90 | | .6 | 75k | barely interesting |
| 1995 | P5-166 | NV1 | .5 | 100k | capable |
| 1996 | P6-200 | VG1 | .5 | 200k | very capable |
| 1997 | P2-300 | NV3 | .35 | 500k | beyond capable |
| 1998 | P2-450 | VG2 | .25 | 1000k | wow! |
| 1999 | P?-700 | | .18 | 2000k | look out! |

Note, approx evolution, only discussing consumer 3D graphics, not professional or workstation.

The Fab column shows the CPU volume fab and also the graphics fab.

Before 1992 things just weren't interesting. CPUs were just too slow and the available gate count was too low. In 1993 PC's starting getting capable with the P5 processors. The higher speed P5 processor, e.g. 166 were what I call capable processors, and now with P2-450 available, we have transitioned into the wow! stage. And things aren't slowing down. There's a predictable 1.6x increase every year, and a doubling in the number of gates appearing in graphics controllers.

The key to all the progress: semiconductor technology!

Now let's look a bit closer at the graphics for each year.

## History of consumer PC 3d Graphics

***Before 1994 - pretty bleak***

***1995 - emergence of HW texture mapping***

- 3Dlabs, ATI, nVidia, Rendition, S3
- 100K tris/sec
- < 10 Mpixels/second z-buffered
- barely perspective correct
- little or no alpha-blending

Before 1994 there was only some primitive RGB shading hardware, possibly z-buffered. Perfomance was pretty abysmal

During 1995 there were a LOT of texture mapping engines in design for consumer entertainment (aka games!), the first showing up late in 1995

Smaller lithographies allowed a critical mass of gates on a chip to actually do something reasonable.

## History of consumer PC 3d Graphics

***1996 - we have landed!***

- Voodoo Graphics (3D only)
  - *1.5M tris/sec peak*
  - *50 Mpixels/sec z-buffered, alpha-blended, bi/trilinear, per-pixel fog, per-pixel LOD mipmapping, etc, etc*
  - *demonstrated at SIGGRAPH '96*
  - *advanced configurations up to 100 Mpixels and two or three simultaneous textures*

In 1996 my company, 3Dfx, shipped Voodoo Graphics which was the first game accelerator to reach "critical mass" in features and performance.

## History of consumer PC 3d Graphics

***1997 - bigger, better, faster***

- nVidia Riva 128
  - *1-5 M tris/sec, 50-100 Mpixels/sec*
- Oak Warp 5
  - *beginning of full-scene AA*

1997 was characterized by Voodoo Graphics.

Many competing chips appeared, although many had much lower fill rate and were not full features

## History of consumer PC 3d Graphics

### *1998 - bigger, better, faster, cont.*

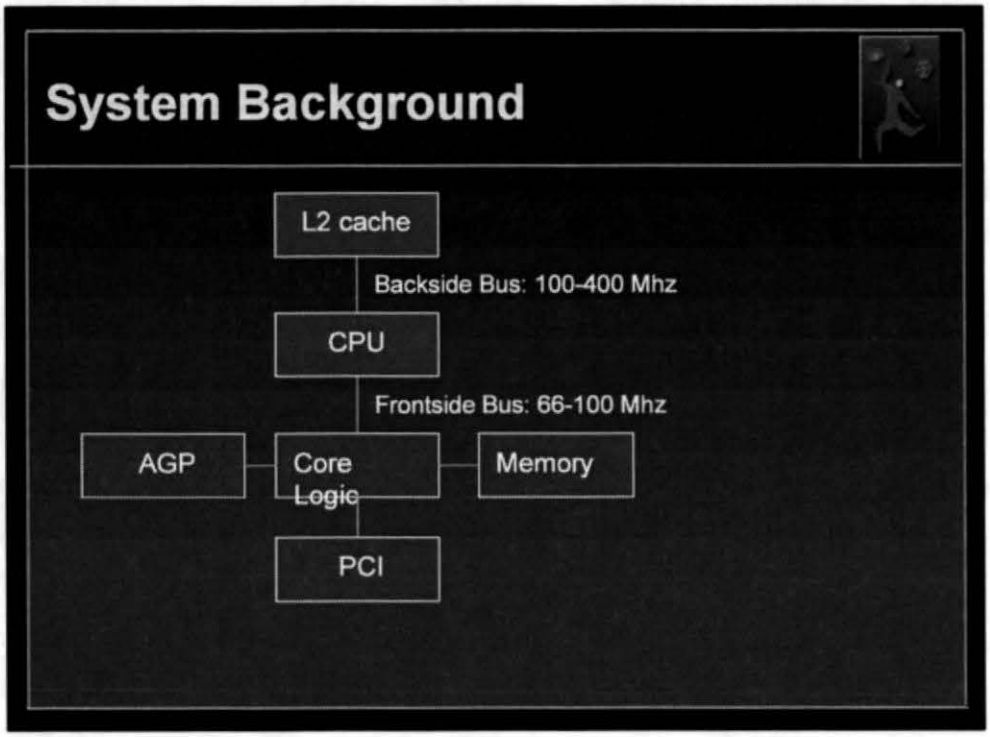
- Voodoo$^2$
  - *3 M tris/sec, 90-180 Mpixels/sec*
  - *2 simultaneous textures*
- multi-texturing
- texture compression: 4 bits per texel

In 1998 things keep getting bigger, better and faster.

You will see many chips like the Voodoo Graphics and Riva-128

Although Voodoo Graphics has supported multi-texturing since 1996, you will see multi-texturing going mainstream in 1998.

Texture compression will become to become prevalent as well, which helps the hardware reach higher pixel rates with a limited memory bandwidth budget..




This is the generic system architecture for today's PC. The speeds vary all over the place. Processors range from 200 to 450 Mhz, frontside bus speeds from 66 to 100 Mhz and maybe even 200 Mhz soon. Backside busses to the L2 cache typically run at between 100 and 400 Mhz. Memory systems range from 500 MB/sec to 800 MB/sec. So there's at least a 2x range in performance level at every component in the system.

Graphics controllers sit on either the AGP or PCI bus. Most were on the PCI bus (33 Mhz) until this year, when the rapid switch to AGP is occuring.

There are two models for talking (transferring commands and data) between the cpu and graphics systems: push and pull.

Voodoo1 & 2 use push model - all commands and data are written from the CPU to the graphics using memory-mapped IO.

Many other designs use the pull model (bus-mastering) where the commands and data are written by the CPU into system memory and the graphics controller reads (pulls) them as needed.

Both require buffering so as to not stall CPU.

## Where are we headed?

*1999 - look out!*

- 5-10M tris/sec, bus bandwidth permitting
- 300+ Mpixels/sec
- full-scene anti-aliasing more prevalent
- texture compression below 4 bits

Consumer PC graphics will be rapidly catching up to, and perhaps exceeding that of workstation PC graphics. There is a lot more money in the consumer markets, and thus more research and development can be justified and amortized over a larger volume. The volumes in the consumerspace are orders of magnitude larger than in the professional space.

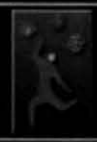
## Where are we headed?

*2000 - ouch!*

- 10+ M tris/sec
- 900+ Mpixels/sec

Blistering speed - approaching and probably exceeding 1 Gigapixel/sec.

## Where are we headed?



*higher resolutions*

*higher framerates thru Mtris+Mpixels*

*higher quality*

- better quality per pixel
- more bits of RGBA
- more textures/effects
- full scene anti-aliasing

Through the years you will see higher resolutions, higher framerates thru increases in triangle rates and fill rates, higher quality thru new features such as full-scene anti-aliasing and deeper framebuffers.

Resolution (both size and depth) and quality are areas where consumer PC graphics have traditionally trailed professional PC graphics, mostly due to cost. As memory gets cheaper, and technologies like AGP open up the system memory to graphics, these differences will diminish. As an example - many consumers are purchasing 24 Mbyte versions of Voodoo2! - something I would not have predicted a few years ago.

## Where are we headed?

***in many areas, PCs copy Workstations***

- z-buffering, alpha-blending, texturing, fog

***but in some areas, PCs will innovate***

- multiple textures
- bump mapping
- low-cost full-scene AA

In the past, PCs have lagged workstations by a number of years. Now that major UNIX vendors have moved their graphics over to PCs, this lag is gone. The only lag that remains is the lag between workstation-level graphics on a PC and consumer-level graphics on a PC.

PC consumer graphics have been "catching" up with workstation graphics for a number of years. But in some areas, consumer graphics have been innovating - mostly due to cost constaints and the real-time game focus.

## Prediction for the day

*I've ignored professional 3D*

*Bold prediction:*

- in the same manner that current PCs blur the line between UNIX workstations and PCs, the advances in consumer 3D I've shown will blur the line between consumer 3D and professional 3D

Here is my bold prediction for the day.

My prediction is that professional 3D on the PC will quickly experience a TON of competition from consumer 3D and will threaten professional 3D cards the same way NT threatens UNIX workstation vendors. Note that no one is predicting the demise of UNIX workstations (at least I'm not), nor am I predicting the demise of professional 3D on the PC. My prediction is that consumer 3D cards will begin eating away at large sections of the professional 3D market, just like NT is doing to the UNIX market.

There will always be applications that require special features or configurations or performance that are not practical in a consumer 3D product. However, these applications and markets will shrink rather than grow in the future.


Questions

# Deploying a High Performance 3D Application for the Consumer PC

Brian Hook
id Software

## Who Am I?

*first engineer hired by 3Dfx Interactive, architected and implemented first version of Glide, the 3Dfx proprietary hardware API*

*worked on SGI's Cosmo OpenGL implementation for Win32*

*now work for id software working on future graphics engines*

## Overview

*A History of id Software's games*

*The Recent Past: Quake 2*

*The Near Future: Quake 3*

## A Brief History of id Software's Games

*Wolfenstein 3D*

*DOOM*

*Quake*

*GLQuake*

*Quake 2*

*Quake 3*

## Wolfenstein 3D

*256-color palette (VGA)*

*World organized on tiles*

*Single elevation*

*Raycasting used for rendering, overdraw management, and hidden surface removal*

*Hardcoded lighting*

*Objects represented as sprites*

## DOOM

*256-color palette (VGA)*

*World represented as a 2D BSP tree*

*Allowed for multiple, non-overlapping elevations*

*Front to back rendering minimized overdraw and handled hidden surface removal*

## DOOM (cont.)

*Arbitrary angles for the walls*

*Based on a constant Z texture mapper (perfectly horizontal or vertical spans)*

*Crude lighting*

*Not a true 3D engine*

*Objects represented as sprites*

## Quake

*256-color palette (VGA)*

*World represented as a 3D BSP tree*

*Overdraw minimized by using PVS computed for each BSP node*

## Quake (cont.)

*Software Z-buffer used to manage hidden surface removal*

*Fully arbitrary 3D polygonal world*

*Preprocessed geometric information forced statically placed architecture that could not be modified dynamically*

## Quake (cont.)

*Precomputed lighting via radiosity-like lighting utility*

*Surface caching*

*Simple dynamic lighting capability (sphere of light rendered into the surface cache)*

## VQuake

*Quake ported to the Rendition Verite V1000 3D accelerator*

*Essentially accepted Quake's span lists directly*

*Free upgrade for registered Quake owners*

## GLQuake

- *Brute force port of Quake to OpenGL*
- *RGB rendering, albeit using 8-bit source art*
- *Surface cache architecture replaced with multipass rendering*
- *Major learning experience*
- *Unsupported free upgrade for registered Quake owners*

## Quake 2

*Selectable rendering subsystems (ref_soft and ref_gl)*

*Software rendering engine slightly upgraded to support transparent surfaces*

*Still primarily 8-bit artwork due to expectation that software renderer would be heavily used by target audience*

## Quake 2 (cont.)

*OpenGL subsystem based on upgraded and refined GLQuake rendering technology, including support for colored lighting and translucent surfaces*

*OpenGL subsystem shipped as part of the commercial product*

## A Visual Comparison (Wolf3D)


## A Visual Comparison (DOOM)

## A Visual Comparison (Quake)

## A Visual Comparison (GLQuake)

## A Visual Comparison (Q II)

## A Visual Comparison (Q III)

Image not available at time of printing

## The Recent Past: Quake 2

### *General Information*

- Christmas '97 release date
- Supported under Win32, however unsupported versions available for Irix, Win32 (AXP), and Linux

## The Recent Past: Quake 2 (cont.)

*Target Platform*

- Pentium/133 minimum
- Pentium/166 recommended
- 16MB recommended for ref_soft
- 32MB recommended for ref_gl

## The Recent Past: Quake 2 (cont.)

***Hardware accelerator requirements***

- Texture mapping, preferably bilinear filtered
- Z-buffer
- Smooth shading
- Alpha blending
  - *transparency: src*a + dst*(1-a)*
  - *colored lightmaps: src*dst + dst*0*

## The Recent Past: Quake 2 (cont.)

### *The REF architecture*

- Allowed us to abstract the rendering at a very high level
- Both refs utilized the same incoming data describing the world, view, entities, effects, etc.
- Each ref responsible for taking the given information and rendering a correct image

## The Recent Past: Quake 2 (cont.)

*ref_soft*

- Surface cache architecture
- Depth complexity of one
- Z-buffered

## The Recent Past: Quake 2 (cont.)

*ref_gl*

- Why not Direct3D?
  - *Horribly broken in terms of functionality, documentation, ease of use, and driver support*
  - *not portable to non-Windows platforms*
  - *hardware acceleration non-existent under Windows NT*

## The Recent Past: Quake 2 (cont.)

***ref_gl (cont.)***

- Why not proprietary APIs?
  - *Design, implementation, and support issues too complex*
  - *Didn't cover a large enough customer base*
  - *Not portable to other accelerators or platforms supported by the API provider*

## The Recent Past: Quake 2 (cont.)

*ref_gl (cont.)*

- Why OpenGL?
  - *Easy to use, well documented, highly functional*
  - *Arbitrary extensions*
  - *Hardware acceleration support under WinNT*
  - *Portable to non Wintel platforms*
  - *GL_RENDERER_STRING*

## The Recent Past: Quake 2 (cont.)

### *How we used OpenGL*

- Only used to access hardware acceleration, software rendering implementations specifically not supported
- Strip/fan oriented, did not use display lists nor vertex arrays
- Did not use OGL's native lighting and fogging capabilities

## The Recent Past: Quake 2 (cont.)

*How we used OpenGL (cont.)*

- Extensions
  - *GL_EXT_point_parameters*
  - *WGL_EXT_swap_control*
  - *GL_SGIS_multitexture*
  - *GL_EXT_paletted_texture*
  - *GL_EXT_shared_texture_palette*
- Non-extended OpenGL 1.1 always supported

## The Recent Past: Quake 2 (cont.)

### *How we used OpenGL (cont.)*

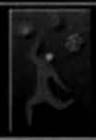
- OpenGL's full transform pipeline was utilized
- Did not use glPrioritizeTextures
- Significant overdraw in some cases, relied on fill rate to make up for a lack of cleverness

## The Recent Past: Quake 2 (cont.)

*How we used OpenGL (cont.)*

- *Multipass rendering*
- *first pass: base textures*
- *second pass: lightmaps*
- *third pass: screen flashes*

## The Recent Past: Quake 2 (cont.)



### *ref_gl development notes*

- Developed on Intergraph TDZ410 workstations with Realizm graphics
- 3Dfx Voodoo features and performance were used as a target for the consumer market
- 3Dfx developed the Voodoo "miniGL" driver with very little assistance from id

## The Recent Past: Quake 2 (cont.)

***ref_gl development notes (cont.)***

- The QGL interface
  - *Necessary because of the existence of minidrivers*
  - *Necessary because Microsoft's WGL interface did not have a device enumeration capability*
  - *Implemented using LoadLibrary and GetProcAddress on Win32*

## The Recent Past: Quake 2 (cont.)

### *Scalability*

- Voodoo was the one and only target for most of Quake 2's development
- Scalability was not considered in the design, a minimum feature set and performance level was expected, and no design was taken into account for taking advantage of more features or performance

## The Future: Quake 3

### *General Information*

- Christmas '98 release target
- Supported under Win95 and WinNT, however we still expect to support other platforms and CPUs

## The Future: Quake 3 (cont.)

***Target Platform***

- Pentium/200MMX w/ 32MB of RAM
- Hardware accelerator required
  - *Superset of Quake 2's rendering requirements*
  - *Needs a complete blending unit*
  - *Will leverage stencil*

## The Future: Quake 3 (cont.)

### *Choice of API*

- Direct3D?
  - *DX5 great improvement, but still not as good as OpenGL*
  - *Still no hardware acceleration under NT*
  - *Still not portable*
  - *Already committed significant investment in OpenGL technology*

## The Future: Quake 3 (cont.)

- Proprietary APIs?
  - *Even less appealing that during Quake 2's development*
- OpenGL!
  - *Same reasons*
  - *Driver support has improved significantly now that OGL has become more mainstream*

## The Future: Quake 3 (cont.)

***Basic OpenGL Features Used***

- Lighting and fogging performed by application code
- OpenGL used for full transformation pipeline
- Do not rely on GLU

## The Future: Quake 3 (cont.)

***Multipass Rendering***

- (Pass 1-4: accumulate bump map)
- Pass 5: diffuse lighting
- Pass 6: base texture (with specular component)
- (Pass 7: specular lighting)

## The Future: Quake 3 (cont.)

### *Multipass Rendering (cont.)*

- (Pass 8: emissive lighting)
- (Pass 9: volumetric/atmospheric effects)
- (Pass 10: screen flashes)

### *Texture Management*

- Handled by the driver
- Still don't use glPrioritizeTextures

## The Future: Quake 3 (cont.)

### *Rendering Primitives*

- Strips/fans still supported, however vertex arrays and compiled vertex arrays are heavily used and favored for performance reasons related to multipass rendering

## The Future: Quake 3 (cont.)

### *OpenGL Extensions*

- Additional extensions supported over Quake 2's extensions
- WGL_EXT_gamma_control
- GL_EXT_multitexture
  - *updated version of GL_SGIS_multitexture*
- GL_EXT_compiled_vertex_array

## The Future: Quake 3 (cont.)

*Overdraw*

- Still significant, but as hardware becomes faster the time spent making overdraw approach perfection reaches a point of diminishing returns

## The Future: Quake 3 (cont.)

### *Coordinating with Hardware Vendors*

- Significant effort spent coordinating with IHVs on new OpenGL extensions and future hardware designs

### *The QGL Interface*

- Still supported because of legacy minidrivers

## The Future: Quake 3 (cont.)

### *Development Notes*

- Developed on Intergraph TDZ workstations
- A lot more good 3D accelerators meant no obvious consumer target accelerator
- Driver redistribution and tech support will become a source of concern

## The Future: Quake 3 (cont.)

*Surface Shaders*

## The Future: Quake 3 (cont.)

*Curved Surfaces*

## The Future: Quake 3 (cont.)

### *Feature Scalability*

- Limitations we have to deal with
  - *lack of multitexture*
  - *lack of blending modes*
  - *limited texture memory*
  - *varying multitexture hardware designs*

## The Future: Quake 3 (cont.)

***Feature Scalability (Cont.)***

- Limitations the driver has to deal with
  - *lack of MIP mapping*
  - *limited texture memory*
  - *maximum aspect ratios*
  - *maximum texture sizes*

## The Future: Quake 3 (cont.)

### *Feature Scalability (Cont.)*

- Limitations that can't be dealt with
  - *Lack of framebuffer precision*
- Shader description language
  - *Given an understanding of a particular piece of hardware, we can do a text description of how to render effectively on that platform*

## The Future: Quake 3 (cont.)

### *Performance Scalability*

- Fill rate: adjust window size and adjust rendering quality
  - *low quality rendering: fewer passes, more reliance on iterators*
  - *high quality rendering: more passes, more realism*

## The Future: Quake 3 (cont.)

### *Performance Scalability (cont.)*

- Triangle throughput: adjust tessellation
- Texture memory: adjust texture quality by using lower color depths, lower resolutions, and enabling compressed textures

## Predicting the Marketplace

*Gut instinct*

*Statistics are either highly misleading or generally inaccurate*

*Plan a realistic road map then execute it*

*Avoid constantly revising your technology during a product's development unless you have no choice*

## Dealing with Driver Bugs

*GL_RENDERER_STRING*

*Runtime bug-checking overrides*

*Work closely with IHVs, including spending lots of time on the phone and in e-mail*

*Provide assistance to driver writers, including providing source code*

## Features/Performance Forecast for Next-Generation PC Apps

*It will become increasingly difficult for applications to lever the higher performance coming out with innovative technology instead of using it as a crutch*

*Trade off between performance and generality will become even more drastic*

*Scalability over a vast range of performance/features will be vital*

## Features/Perf. Forecast for Next-Generation PC Apps (cont.)

*Dynamic tessellation to control scene complexity will be required*

*Optional features to control fill rate requirements*

*Bus bandwidth limitations are going to change the way we think sooner than expected*


Questions

# Optimizing a Graphics Application for Consumer PC Graphics

*Gary Tarolli*

*Chief Technology Officer*

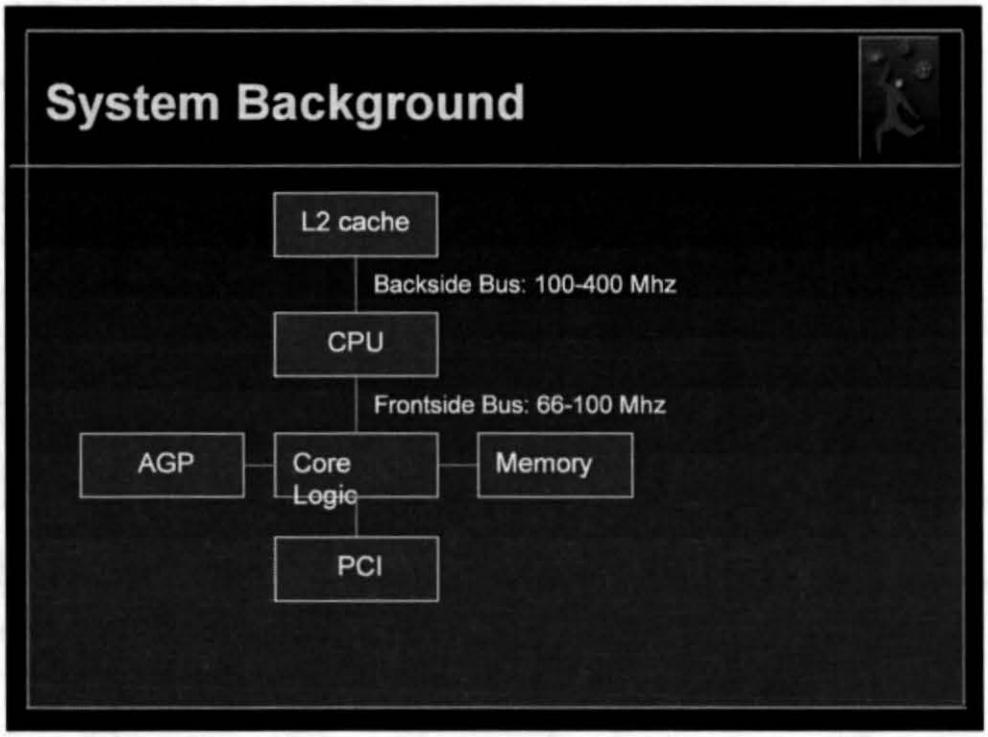
*tarolli@3dfx.com*

## Case Study

*1 million sustained triangles per second, or approx. 20,000 @ 60 Hz*

A good way to present the issues involved in optimizing graphics is to look at a simple test case: 1 million triangles per second.

Let's review the typical PC system.




I prefer to analyze the system from the monitor back to the CPU. The monitor certainly isn't an issue, nor is the DAC. Next up is the graphics controller, which is either hanging off the PCI or AGP bus. So let's start with the graphics controller.

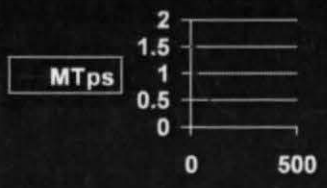
## Start at Graphics

***Is the graphics chip fast enough?***

- small triangles are no problem if within spec
- but they won't all be small, guaranteed

***Typical performance curve:***

| | |
|---|---|
| *1.5M* | *12 pixel* |
| *1.0M* | *25 pixel* |
| *.6M* | *50 pixel* |
| *.2M* | *200 pixel* |




Obviously, if your goal exceeds the graphics chip maximum rate, you are in big trouble. In this case here it is a perfect match - or so it seems.

## Triangle Mix

*goal: 20,000 25-pixel triangles/sec*

| # | tri-size | usecs | msecs |
|---|---|---|---|
| 100 | 1000 | 20 | 2 |
| 1,000 | 100 | 3 | 3 |
| 19,000 | 25 | 1 | 19 |
| | | | 24 |

With 20,000 25 pixel triangles, each taking 1 usec, we end up with a budget of 20 msecs/frame or 50 fps. However, this mix will never happen in real life (it's too ideal), and it only renders 500,000 pixels.

If we mix in 100 large triangles and 1000 medium size triangles, the frame time goes up. And as you see we are 20% over budget.

## Triangle Mix, cont.

*goal: 20,000 25-pixel triangles/sec*

| # | tri-size | usecs | msecs |
|---|---|---|---|
| 500 | 1000 | 20 | 10 |
| 500 | 100 | 3 | 1 |
| 19,000 | 25 | 1 | 19 |
| | | | 30 |

If we assume even more large triangles then we are 50% over budget. In order to get back on budget, the small triangles have to run twice as fast.

## My Rule: peak triangle rate must be 2-3x the sustained rate

*goal: 20,000 25-pixel triangles/sec*

| # | tri-size | usecs | msecs |
|---|---|---|---|
| 500 | 1000 | 20 | 10 |
| 500 | 100 | 3 | 1 |
| 19,000 | 25 | 0.5 | 9 |
| | | | 20 |

Here we see that even with the same fill rate, if the small triangles are capable of running at 2 million/sec we can sustain 1 million/sec. This is because half the frame time is "hogged" by large triangles which fill many pixels but do NOT add significantly to the total triangle count. Therefore to sustain N triangles/sec you need a peak of 2N to 3N in the graphics chip. It will fall behind on large triangles, but catch up on small triangles.

## State Changes

***Changing state can be expensive***

- e.g. texture, shading, and lighting modes
- larger machines are usually worse

***Changing state can be cheap***

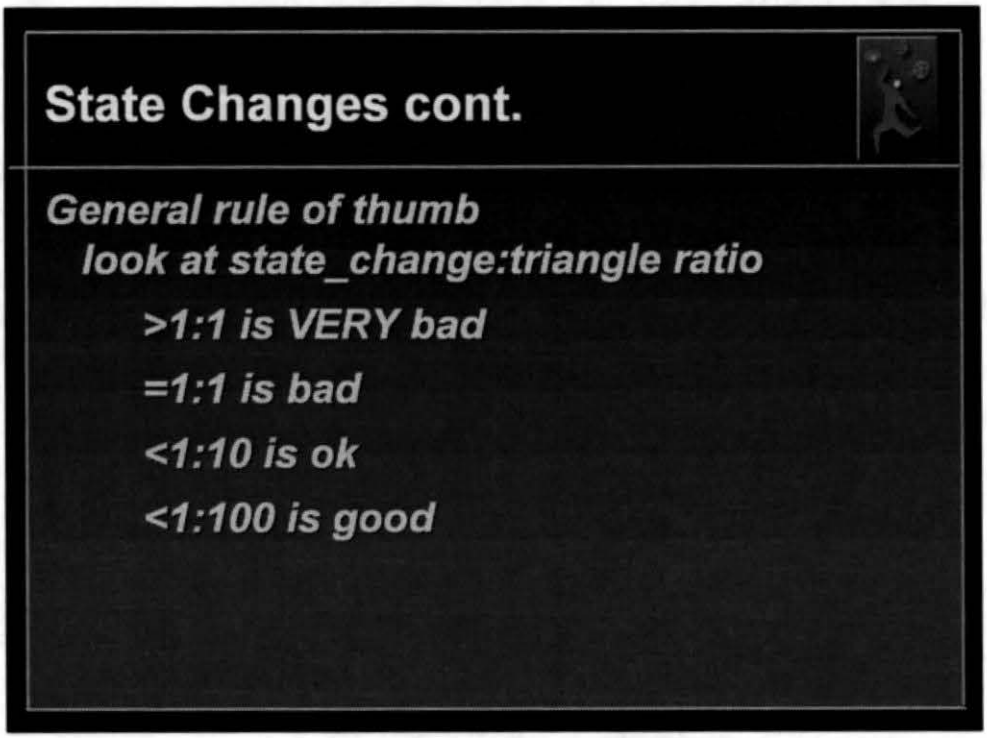
- Voodoo is typically fully pipelined

***How do you know, what do you do?***

Just because a graphics chip is rated at 2 million triangles/second, does not mean you are going to get that performance in your application. There are many things that can slow it down. The most common culprit is state changing.

There is no set rule for how long a state change can take, it could take 1 clock, it could take 100's of clocks. Different state changes could take different amounts of time. The only way to know is the either read the manual or benchmark (which never lie). I cannot speak for other hardware, but I will say that almost all state changes on Voodoo1 and 2 are extremely fast - much faster than rendering a very small triangle.

A good rule is to try to minimize state changes no matter what machine you are on. Here is one area where a hardware Zbuffer comes to the rescue - BSP sorting results in frequent state changes, Zbuffers allow you to render polygons in any order - prefereably sorted by state (material).




All this is based on the rough assumption that a state change (or a few simple changes) are approx. the cost of a small triangle. Thus if the ratio is 1:100 you only lose about 1% performance due to state changes.

Actually to be more precise, it's the ratio of state changes to clocks it takes to render a triangle. The goal is to make state changes a small percentage of triangle rendering time, so this is the most accurate measure.

If triangles are large, e.g. 1,000 pixels and take 1,000 clocks, a state change per triangle is acceptable (barely).

## The Magic Bus

*Is the bus fast enough?*

- vertex = XYZW, RGBA, ST etc
- figure 40 bytes
- independent tri = 144 MB/sec (66 Mhz)
- strip = 50 MB/sec (33 Mhz)

After the graphics chip we move upwards through the system and cross the bus - the wires that transfer data and commands from the CPU to the graphics system.

If we assume floating point data for all parameters then a vertex is around 40 bytes. Sometimes data can easily be compressed, e.g. packed 32-bit RGBA, but in the future we will also have multiple textures, so 40 bytes is a good illustrative number.

With independent triangles, which require 3 vertices per triangle to be sent over the bus, a 66 Mhz bus is required as the bandwidth requirements exceed the peak bandwidth of a 33 Mhz bus (assuming 32-bits wide).

Triangle strips and fans reduce the vertex count by 3x, and thus only require around 50 Mbytes/sec which is well within the limits of a 33 Mhz bus.

## Push vs. Pull

*Pull model*

+ graphics can pull data during entire frame
- CPU has to write data to memory first resulting in both a write and read of memory and less bandwidth left for application

## Push vs. Pull

### *Push model*

- \+ write data once (fire and forget), more system memory bandwidth avail. for appl.
- \- requires large fifos on graphics card, instantaneous bandwidth may exceed bus, although good CPU buffering helps

Neither the push model or the pull model is "right". The push model saves system memory and memory bandwidth in exchange for graphics memory and memory bandwidth. Depending on the "balance of power", this may or may not be the best thing to do.

## Oops

***What if graphics card does Geometry?***

***Pull:***

- might be able to pull data without CPU touching the data

***Push:***

- CPU has to move more data unless cached on the graphics card somehow

This slide shows you how nothing is absolute: what is best for one design may not be best for another. The assumptions for the arguments I just showed you can be wrong if the design changes. If a graphics card performs geometry computations, it might be able to pull data without the CPU touching the data. Thus the PULL model may not require any CPU memory bandwidth at all!

In both cases, this might result in more data being transferred over the bus if backface culling is performed in the hardware instead of on the CPU.

## CPU

*The final bottleneck is the CPU*

*Good transform+light code is essential*

*Optimal use of memory bandwidth*

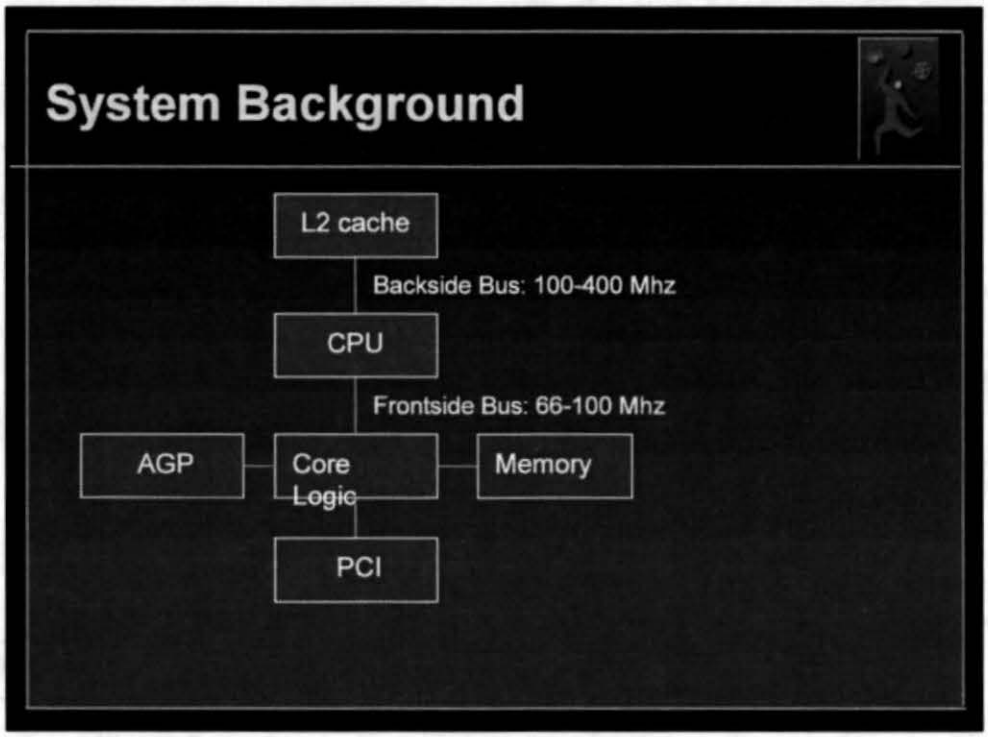
- no wasted reads/writes
- prefetching if available
- L1 and L2 cache friendly

As fast as Intel can make CPUs, code will consume them. Bad code can slow down the fastest computer. In 1996 we showed 1 million triangles/sec. being rendered on a P5-166 on Voodoo Graphics. Today, I read about people not being about to exceed 5000/sec. on the same machine. Now, while I never expected to see a real application sustain 1 million like our highly tuned demo, also didn't expect to see numbers as low as 5000 either!

The first step is optimized computation, e.g. transform and lighting if you are doing it yourself. Next up is memory bandwidth. This is true on any machine, even workstations.

Wasted reads/writes is my hot button. When I was working on the Indigo Entry project at SGI we wrote the entire fast path to IrisGL in assembly language. Every single load and store was scrutinized. Optimizing math was the easy part, minimizing loads/stores was the hard part. In my eyes, all loads and stores are wasteful, only arithemetic operators are useful. Performance is an attitude!

If you are touching data multiple times, try to work within L1 and L2 caches




If we make the assumption that the 20,000 triangles/frame come from main memory (no hierarchy) and require transformation in the CPU, then 144 Mbytes/sec of vertex data has to travel from memory to the CPU, perhaps back to memory a few times (at least once for the pull model), and then finally to the graphics controller.

One sanity check is to see what percentage of the system bandwidth these transfers are. If they exceed or even come close, you are in trouble - your application needs some bandwidth too!

If your models have hierarchy, or are dynamically generating data (e.g. nurbs) then this can dramatically reduce the memory bandwidth requirements.

## Dilemma

*Does the application care about all this?*

*Yes, when trying to optimize!*

So this brings us to the following dilemna. Should applications care about any of these topics I just presented? The answer is ABSOLUTELY if you are trying to optimize performance. You must first undertand these issues, and then monitor and address them.

Now, these performance issues are not the ONLY issues you have to deal with....there's more

## Different Configurations

*I think 1 and 2 Mbyte boards are history*

*Now you have to deal with 4-16 Mbytes*

*Variable screen resolution*

*Variable texture resolution*

## Different Performance

***Performance may vary by 10x***

- triangle rates (vary tesselation or LOD)
- pixel fill rate (vary screen resolution)
  - *may differ base on modes!*
- bus bandwidth (10x range)
  - *33 Mhz PCI : 100 Mbytes*
  - *AGP-Pro (4x) : 1000 Mbytes*

## Missing Features

*Capability bits*

- tell you if the feature is present
- but there are too many to deal with

*Extensions*

It's easy to deal with 1 capability bit, e.g. z-buffering. But now add on top of that:

chromakey

floating point z-buffering

alpha-blending

texture blend modes, multi-texture

anti-aliasing

NOTE: OpenGL mandatory requirement is NOT a solution for real-time applications. Slow is the same as not implemented!!!!!

## Summary

*No absolute answers*

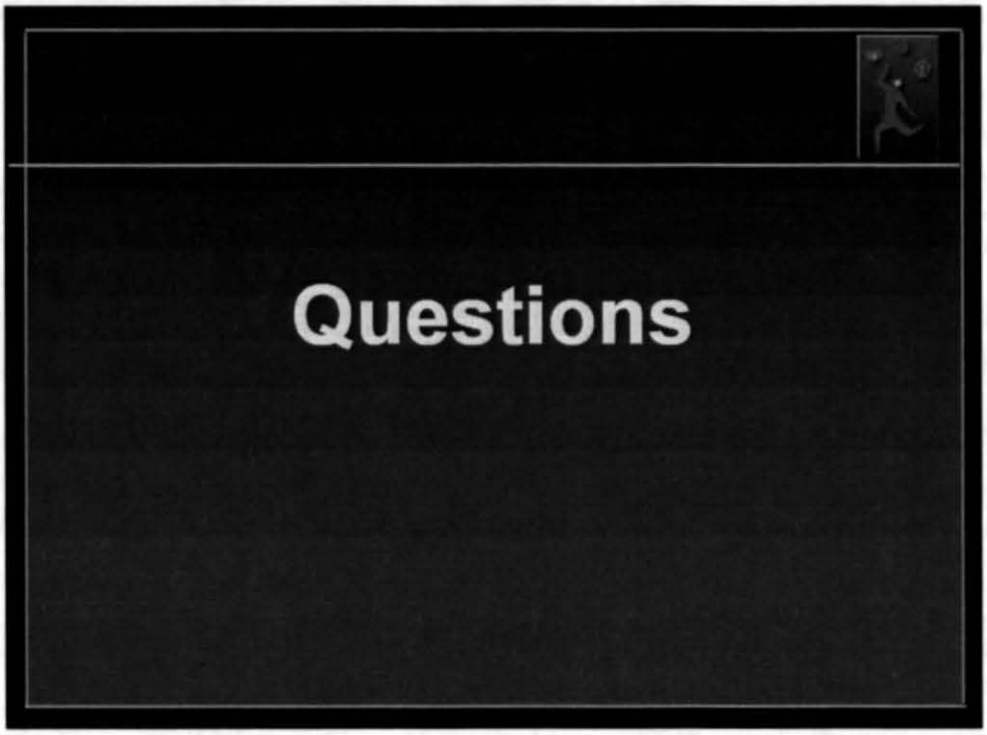
*Answer is often application dependent*

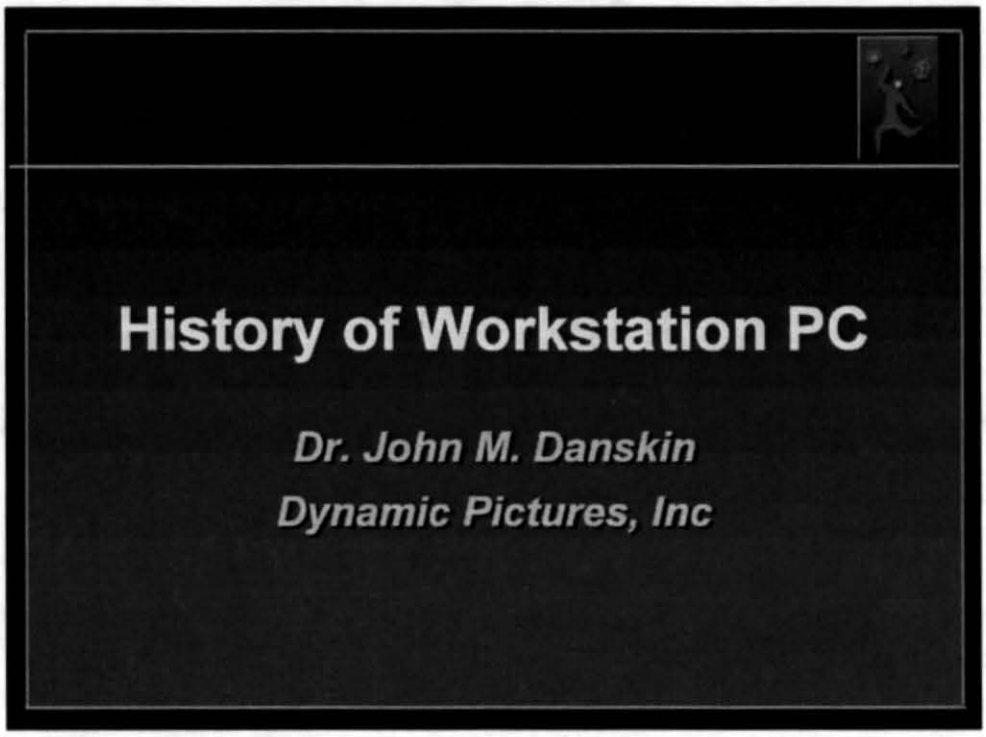
*Answer is often system dependent*

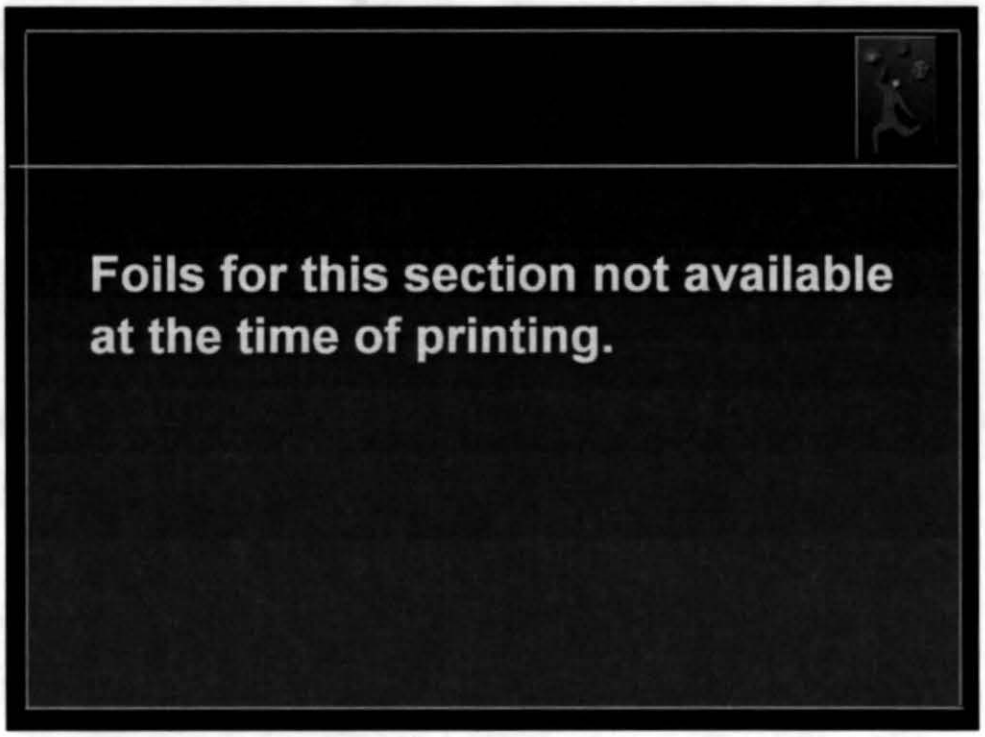
*Understanding the system is key*

While I haven't given you any real answers, I hope I have given you insights into how things work. I truly believe knowing the system and how it works is critical info. I can't give you any one generic answer - it's very appl. and system dependent. My goal is to give you the tools so that you can find the answers.

The other thing to note is that optimizing for a particular platform yields amazing results, but at the cost of portability across other hardware. Witness the amazing games that you see on some of the older game console system - even those without 3D hardware. The same thing holds true for PC graphics. This is one of the reasons why 3Dfx has a large developer relations and support staff: tuning to a particular platform makes a world of difference; targetting the least-common-denominator results in mediocrity.


Questions


History of Workstation PC
Dr. John M. Danskin
Dynamic Pictures, Inc

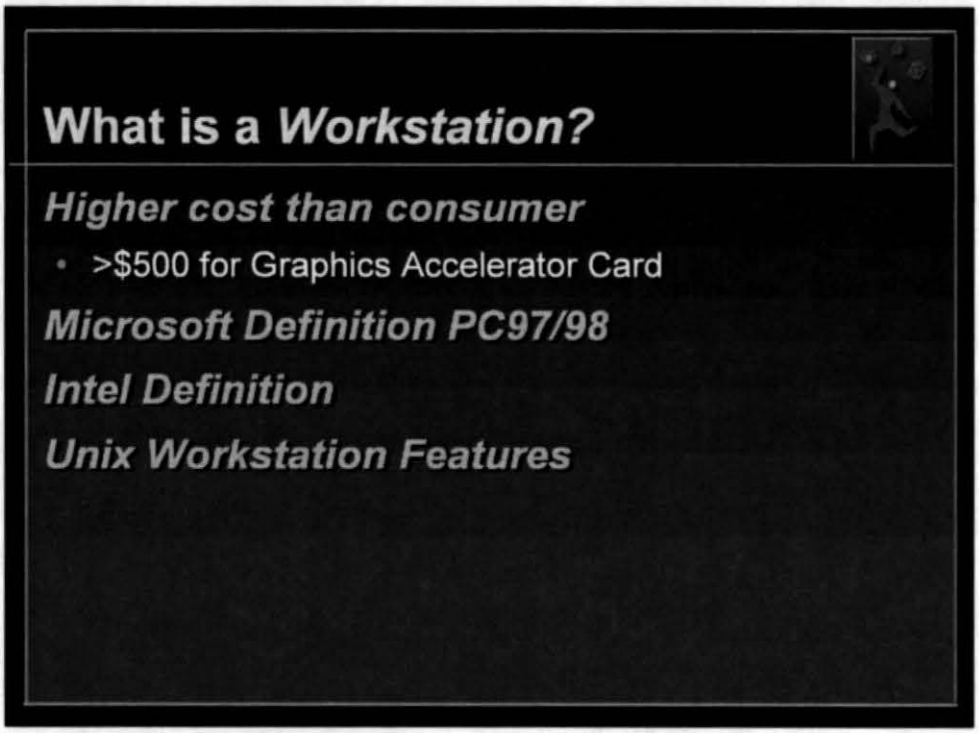

Foils for this section not available
at the time of printing.

# Architecture and Acceleration on the Workstation

*Rich Ehlers*

*Evans & Sutherland*

*Desktop Graphics (the REALimage Technology Group)*




What is a workstation? There are a number of different ways of defining it.

The term workstation generally implies that it is intended for the technical engineering office, not for the home or administrative assistant. This can include:

•mechanical engineers

•artists and designers

•mathematicians

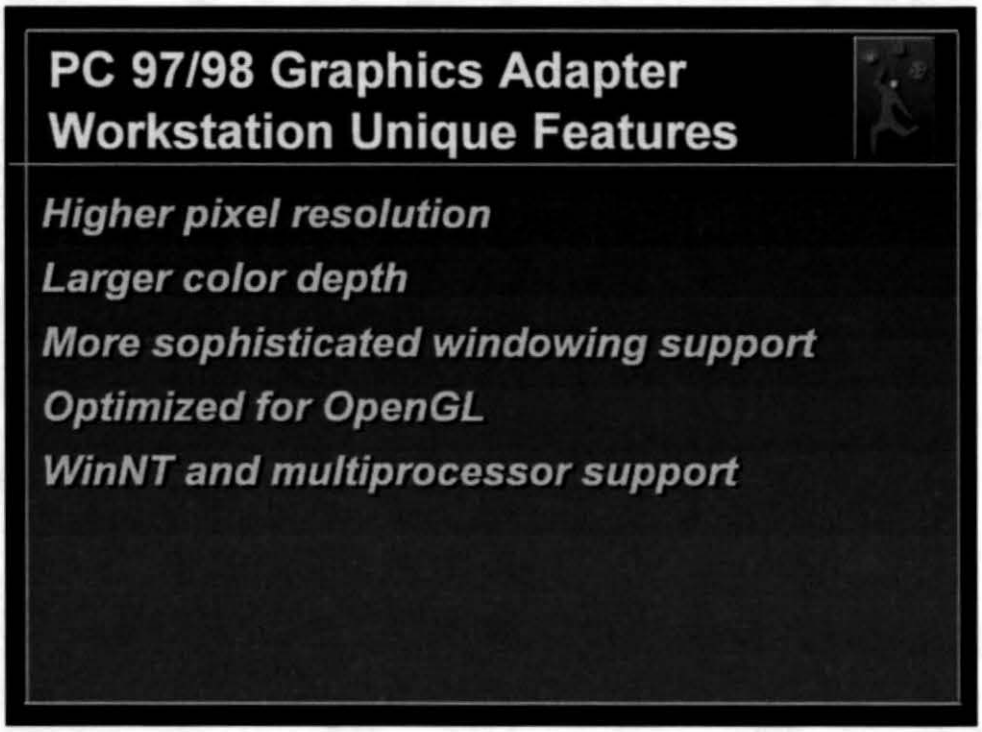
The first definition is a function of cost. A workstation PC is a higher cost box and the graphics is also higher. A home graphics card is under $200. A workstation graphics card (or subsystem) is over $500 but may also come down to around $300.

Microsoft has a definition of different PCs and publishes what it expects in each of the various platform types. These falls into the three classes of PCs:

•general

•workstation

•entertainment




There are many common requirements but this list is the unique requirements for the workstation compared to the general PC.

Higher pixel resolution - workstations are standard at 1280x1024xtrue color. They are going up to HDTV resolutions and have multiple monitor capabilities to get 'side-by-side' HDTV resolutions from a single graphics card.

Larger color depth is a requirement. True color (8 bits of each red, green, blue)

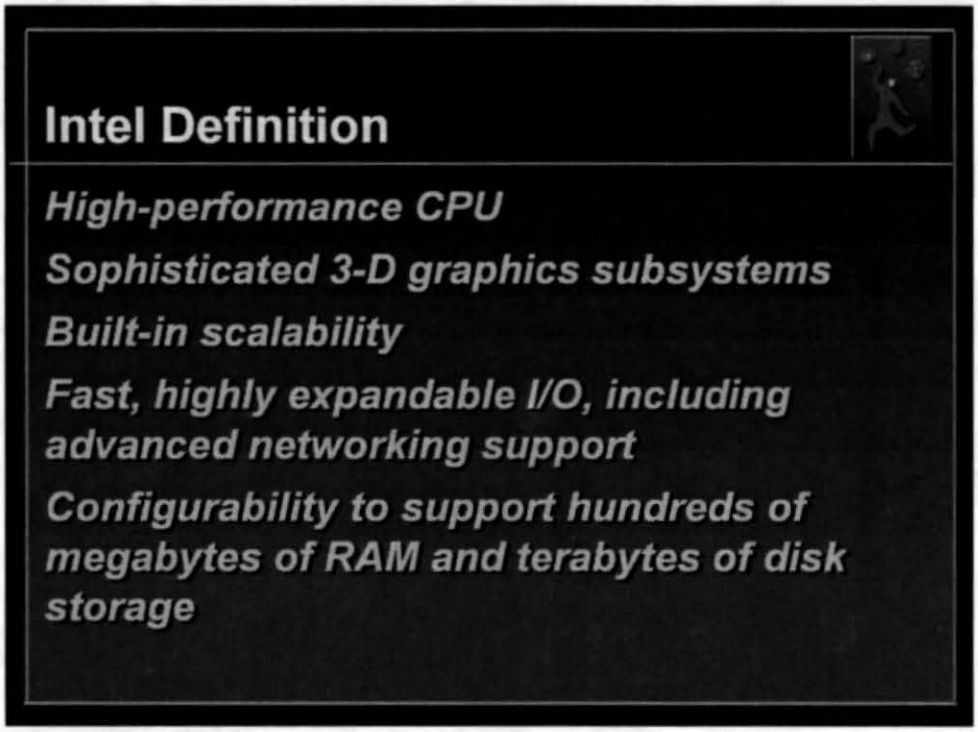
A workstation must do rasterization in true color.

The graphics must be double buffered.

Since the graphics must be double buffered, when the picture changes there is a requirement that there must be no visual artifacts. The artifact that is seen is called 'tearing'. To eliminate tearing, the picture must be able to swapped during a vertical retrace. This is sometimes called swapping at vertical sync or swap syncing for short.

Hidden surface removal is the elimination of parts of solid objects that are obscured by other in the same scene.

Performance of the graphics in a workstation is to be higher than that of the general PC. To accomplish this Microsoft requires that there is some parallelism between the host CPU and the graphics. While the graphics is going on the CPU must be able to do other work. Hence they recommend using DMA or large fifos to contain the graphics commands.




Intel has its definition of a workstation. This is reflected in many offerings that you can find from Intel based computers. The CPU is central. It can be single but many workstations offer dual processors.

While there is variability in the end product that can be purchased, most offer sophisticated 3-D graphics subsystems. Graphics is the focus of SIGGRAPH and this class, but with sophistication comes complexity and differences. There is no such thing as a *VGA* standard for 3D graphics.

The system should have built-in scalability, especially in the ability to add the second processor.

As the phrase from the past says: *The Network is the Computer* workstations do not work alone, but are networked.

Workstations can have huge amounts of storage to meet the requirements of many applications.

## Unix Workstation Features

### *Already Available*

- Window IDs
- Independent swap-able windows
- MIPMAP texturing
- Geometry engine subsystems
- Local geometry storage

### *Future Enhancements*

- Non-homogeneous color depth visuals

Graphics vendors have looked at what has made graphics for Unix workstations successful and most of these features have been incorporated into graphics cards.

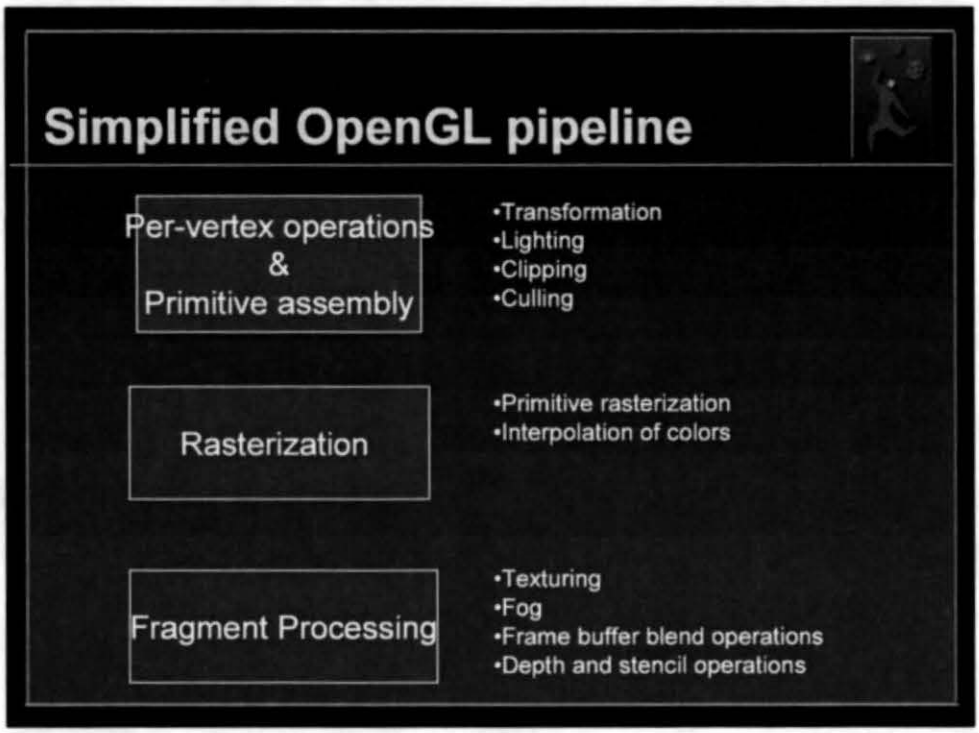
Features that are required on Unix workstations exist already in graphic cards available for NT systems today.

Window IDs and Independent swapable windows will be addressed in detail later. Window IDs are the hardware feature name that provides functionality which includes the independent swapable windows.

MIPMAP texturing is selecting proper texture resolution and includes proper perspective correction.

Geometry engine subsystems are more functionality that offloads the work from the host computer. If it includes local geometry storage it can possibly offload bus bandwidth requirements.

There is one feature that is not present on graphics subsystems but is on Unix workstations. That is non-homogeneous color depth visuals on the same screen simultaneously. This is having an 8-bit indexed color window visible at the same time a true-color window is also visible. The reason for this is that in Windows NT 3D graphics is not fully integrated into the OS.

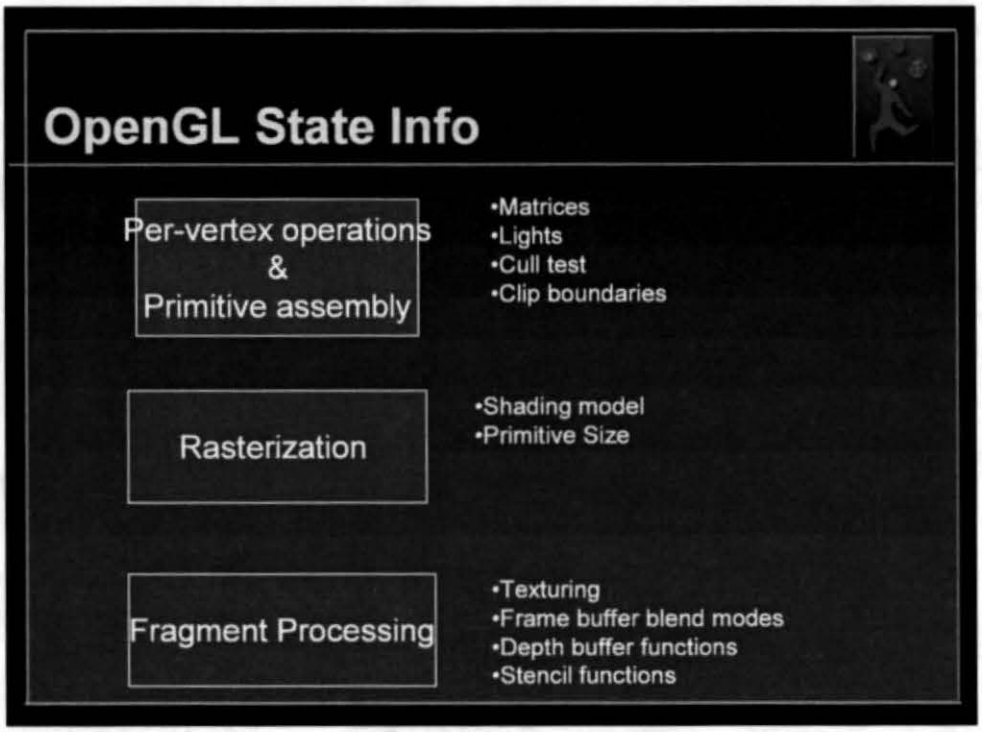



The OpenGL pipeline can be generalized into three major sections.

Per-vertex operations and primitive assembly is the section that does the geometry processing: transformation of vertices and lighting of vertices. The primitive assembly takes the transformed vertices and assembles them into the primitives whether the primitive be line loops, triangles, polygons, etc. After the primitives have been assembled the section can then do culling and clipping on the primitives.

Rasterization is the conversion of points, lines, or polygons to fragments, each corresponding to a pixel in the frame buffer.

Fragment processing is the application of texturing and fog to get the pixel color. It concludes with doing the depth and stencil operations followed by the frame buffer blend operation if the pixel passes all tests to be written to the frame buffer.



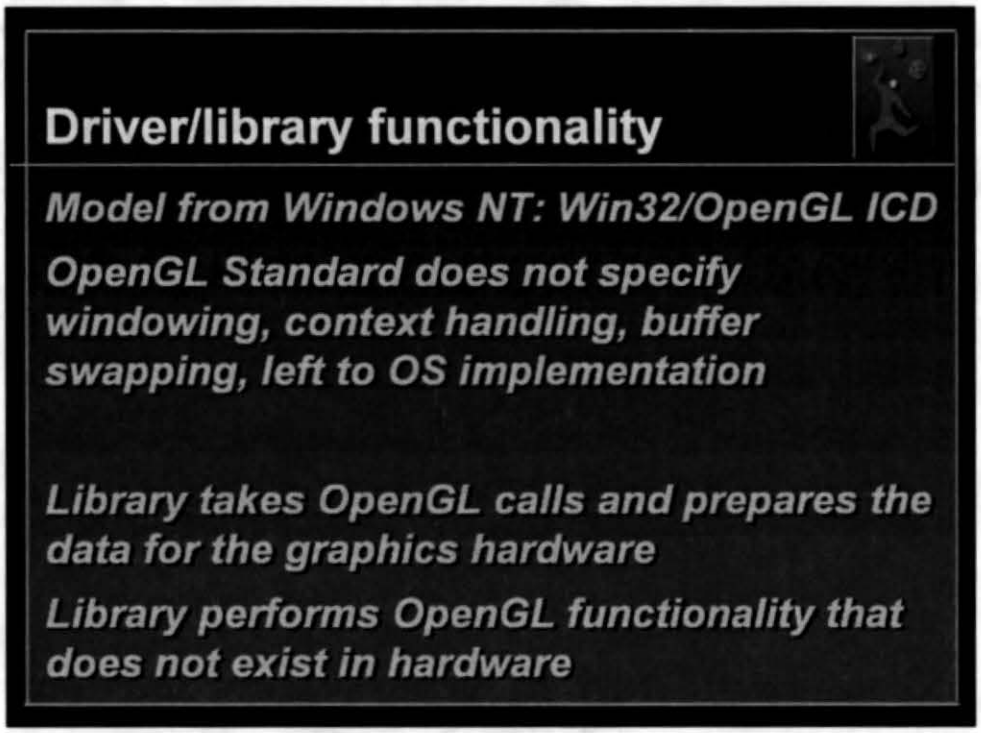

Associated with each section of the OpenGL pipeline is information that is necessary to complete the operations, but does not come with each vertex data.

The per-vertex operations and primitive assembly section requires matrices for transformation, lights for lighting, cull test mode and clip boundaries.

The rasterization section requires the shading model and primitive size.

The fragment processing section requires texture maps and texture modes, depth test and stencil functions, and frame buffer blend modes.

This information is referred to as state or information in the graphics context. The state needs to be available to the proper section and as is described in detail later with regards to hardware implementation may require special handling to get to the proper section of the hardware pipeline.




The model for the remainder of the notes is based on Microsoft's OpenGL Installable Client Driver for Windows NT for 3D graphics and Win32 for 2D graphics.

The OpenGL standard only specifies the 3D graphics operations independent of operating system and windowing system. The major areas of functionality that can vary include windowing, graphics context handling and buffer swapping. These are the areas where the two areas meet. All further discussions will concentrate on Win32/OpenGL ICD.

Library refers to the OpenGL functionality that executes on the host CPU. Generically this functionality provides the function calls specified by the OpenGL standard and gets the information to the graphics adapter in whatever form it requires. The library also is required to do whatever OpenGL functionality does not exist in hardware so that the entire system is OpenGL compliant and applications will have the same (possibly not exact) results on all systems.

The driver is the interface into NT to be able to communicate with the hardware. In the Win32 model the 2D is generally considered to be part of the driver while the ICD model can either have OpenGL implemented in the kernel or in user space. It is the 2D driver that handles window events and has the appropriate information available to the OpenGL to determine window position, size and exposed areas.

## Optimizing Applications for *High-End* Graphics Subsystems

***Just because there is a workstation 3D subsystem does not guarantee high performance***

***Never ASSUME functionality based on a single implementation of OpenGL***

***Knowing the basics about hardware features will greatly enhance application performance and portability***

Applications that were developed on software-only OpenGL libraries may not have had the opportunity to experience the subtleties of hardware graphics systems. Hardware does have its bottlenecks where software systems that are single-threaded don't have specific bottlenecks. Not knowing and coding for hardware bottlenecks may result in when the application is run on a high-end card it does not get significant performance improvements.

All graphics standards, including OpenGL, have areas where the actual functionality is *implementation dependent*. This leads to rule two: do not implement to a particular implementation of OpenGL for other systems may have made different decisions.

The rest of this part of the course is going to go into hardware implementation of graphics subsystems, with an emphasis on rasterization and the frame buffer. By having knowledge of the hardware, applications that are hardware *aware* will be better able to take advantage of the graphics hardware.

## Graphics Functional Areas that can exist on Graphics Cards

*Rasterization*

*Set-up*

*Geometry Transform/Light/Clipping*

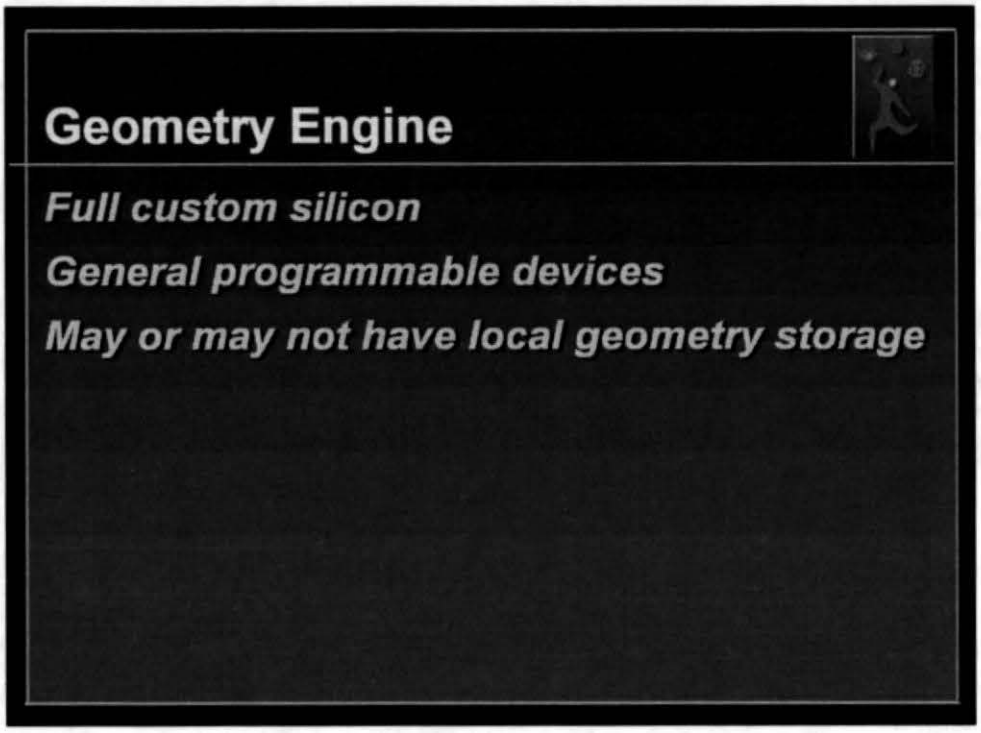
*Display list storage*

Graphics systems come in various shapes and sizes. These vary by price, performance and functionality. The varying functionality that are on the cards fall into just a few categories.

Rasterization is required and all systems have it. Set-up is a required feature and if a graphics card does not include this functionality avoid it.

Geometry engines are additional functionality that you will find in some graphics systems. The geometry engine does the per-vertex operations and primitive assembly. There are different approaches and features that can be included. One item that can be included is local display list storage.

When writing an application for maximum performance, write the application as though the maximum graphics subsystem is the target. Putting in optimizations for a geometry engine with display list storage. On systems without a geometry engine the application will have higher performance than if the application had not optimized for a geometry engine.




Geometry engines come in various shapes and sizes. One implementation is a full custom silicon pipeline. Probably best cost and performance per gate.

Other approaches are to use general programmable devices, whether they be DSPs or general purpose CPUs.

The geometry engine can be further enhanced by adding more memory so that it has local display list or geometry storage.

The geometry engine reduces the workload on the host CPU.

The local display list storage reduces the CPU's bus bandwidth.

## Geometry Engine Task Comparison

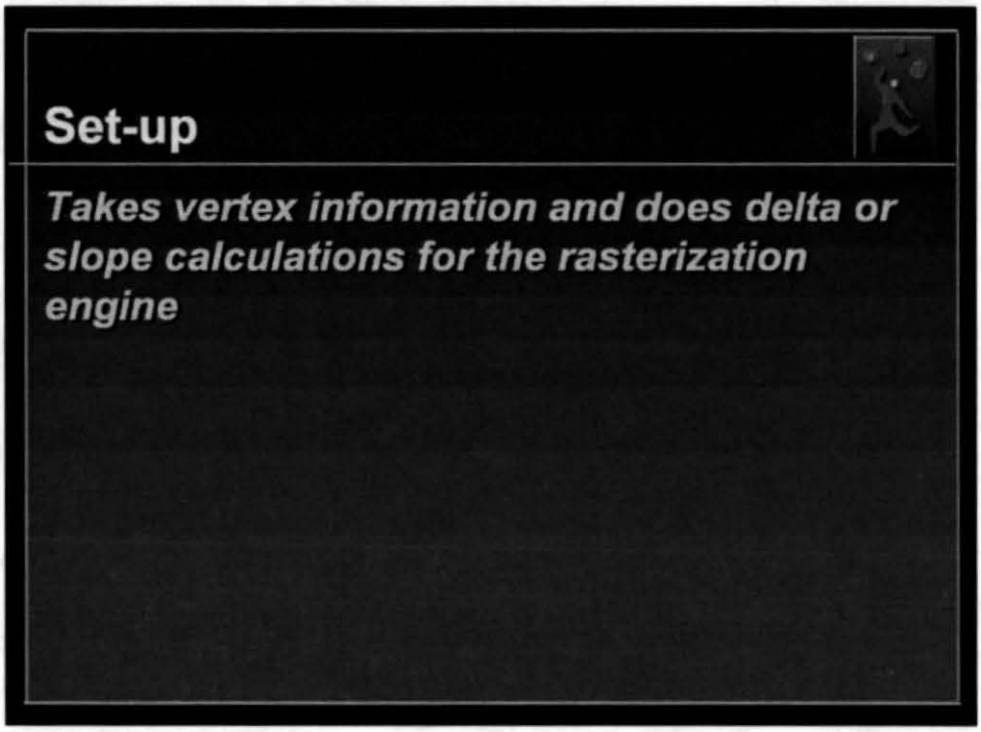
| | No Geometry Engine | Geometry Engine | Geometry Engine with Geometry Storage |
|---|---|---|---|
| **Host CPU** | Full Geometry: xform, light, clip | *Package* untransformed geometry | *Package* storage updates, and list execution commands |
| **Bus Traffic** | Transformed geometry data | All non-transformed geometry data | Storage updates and list execution commands |

This table points out the differences in the host CPU tasks and bus traffic given three possible graphics subsystems. The example is for the case where the application has put all geometry into display lists.

*Package* means putting the data into the appropriate form to be communicated to the graphics subsystem.

One will note for the Host CPU, the amount of processing required is reduced going from left to right.

For bus traffic, geometry storage requires the minimum amount of data to be sent across the bus, provided the geometry data is contained in local display lists and does not need to be sent to the graphics subsystem every update. Generally bus traffic for the no geometry engine case will be less than the case of just a geometry engine. This is due to the fact that clipping occurs in the geometry engine and not on the host, hence all data for the update must be sent to the graphics subsystem.




All workstation graphics subsystems must contain *set-up*. This refers to the calculations that take the vertices for the primitive and then calculate all the slopes and deltas to perform the rasterization operation. Experience has shown that this is a compute intensive task and if not included in the graphics subsystem it can greatly increase the bus bandwidth required.

Set-up influences the vertices per second performance rating of the hardware.

## Rasterization & Fragment Processing

*Turns dots/lines/triangles/polygons into fragments (pixels & associated data)*

*Interpolates associated data*

- colors
- texture coordinates

*Applies texture*

*Depth and stencil testing*

*Alpha test*

*Pixel ownership test*

*Frame buffer blend operations*

Rasterization together with fragment processing determine the graphics system pixel fill rate.

Actual hardware chips usually include some of the fragment operations during rasterization. The end picture needs to be the same but the order and exactly what is done in rasterization and fragment processing is totally hardware implementation dependent. Some of the choices are done for performance. Some systems are *no compromise* systems. Those are the ones that as you turn on features such as texturing and depth testing the pixel fill rate does not decrease. On other systems enabling more features has a performance impact. In these cases faster operations, such as depth test, may be done first to see if other more expensive operations, such as texturing, are done at all.

## Pixel Ownership

*Window Identification (WID) functionality*

*Arbitrary boundary for hardware clipping*

*Display buffer selection*

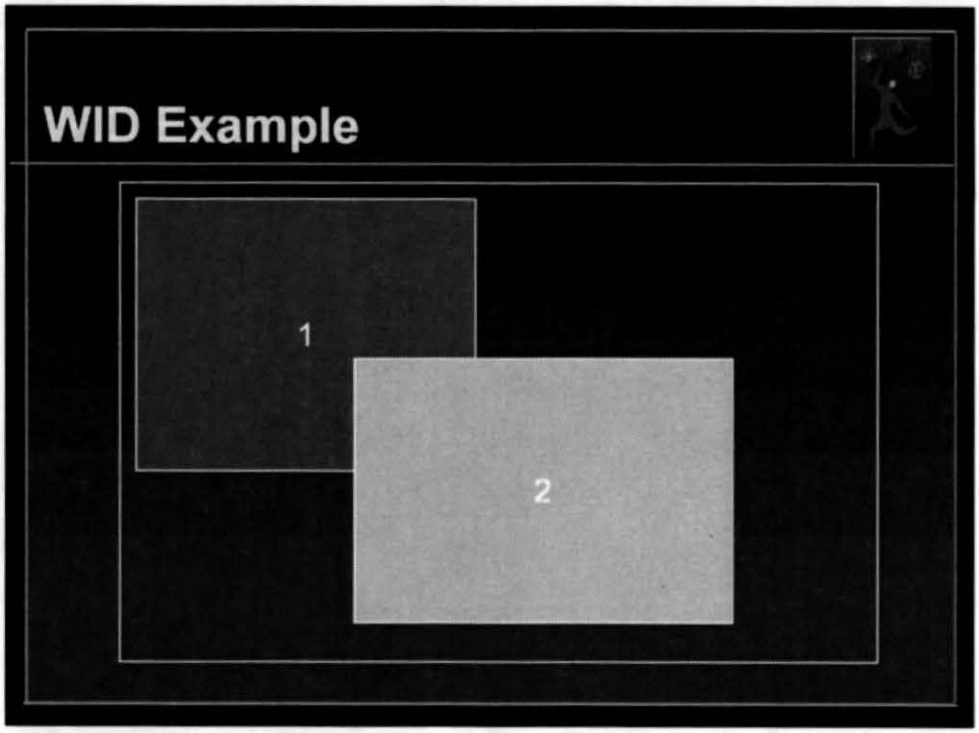
Pixel ownership is where OpenGL meets the windowing system. This feature is very often called Window Identification (WID) as this is the name given to the bits in the frame buffer that enable this feature. It requires more memory but gives more functionality and can increase performance of the geometry engine.

Allocation of a unique number for each 3D window allows for the hardware to do unique things for each. This generally includes:

•hardware clipping enabling arbitrary clipping boundaries

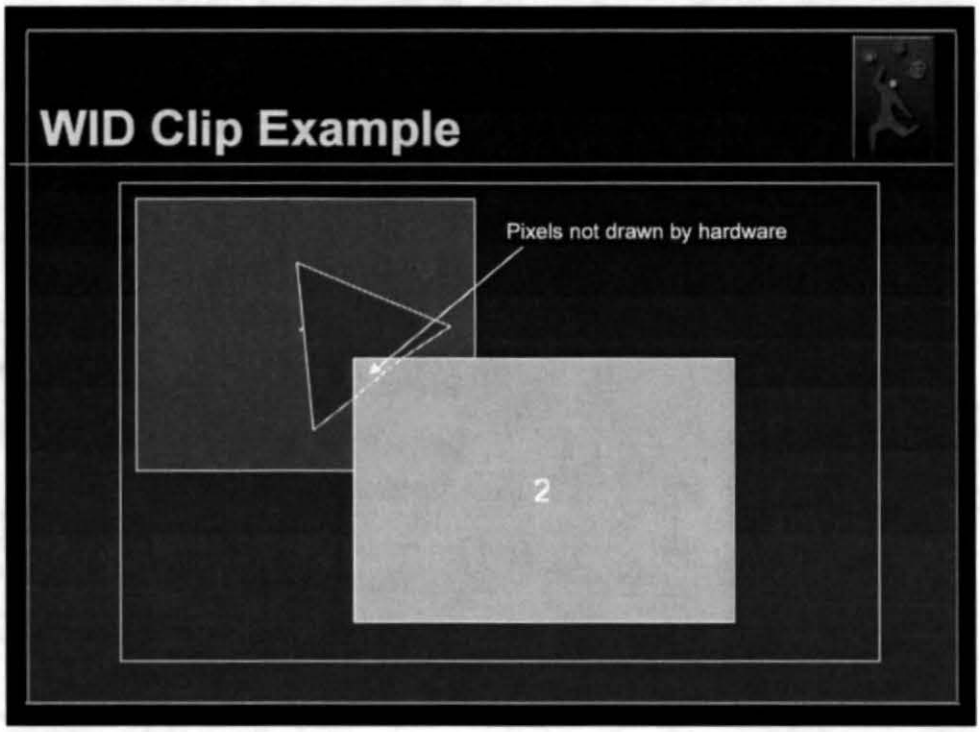
•display buffer selection

•CLUT table selection




This example is to show how WID can be used for arbitrary boundary clipping. There are two 3D windows with WID 1 and 2.

Window 2 is on top of window 1. The application is going to draw a triangle into window 1.




The blue triangle is all within window 1 but window 2 overlaps part of the triangle. With WID clipping, those pixels that are covered by window 2 are automatically not drawn by the hardware since the test to see if pixel is written to the frame buffer will only pass if the WID associated with the pixel is equal to 1 and in this case it is equal to 2.

The geometry processing performance gain is that if there was no hardware support for this type of clipping, the geometry engine would have to know all the boundaries of window 2 and clip all primitives being drawn into window 1 to those boundaries. If you go and add more windows, especially 2D menus you can see how complicated it can get.

## Buffer Swapping Techniques

*Number of different techniques*

*Video display hardware specific*

*Difficult to impossible to know which is used; information not inquirable through OpenGL since this is part of the windowing system*

*Must not assume one particular implementation*

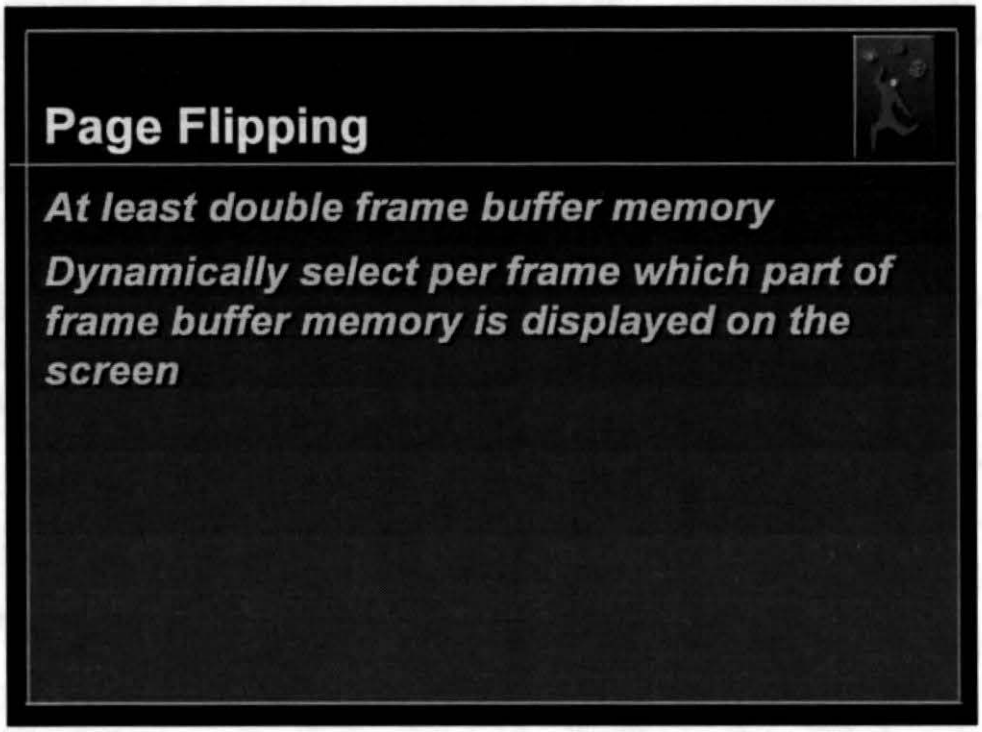
*After swap, back buffer data effectively unknown*

Three dimensional graphics is usual drawn double buffered. This is where the picture as it is updated is not visible and then when it is complete, the buffers are swapped and the new picture is now visible.

This switching of pictures can be accomplished in a number of ways. It is very dependent on the capabilities of the display hardware. OpenGL does not have any inquiry mechanisms that allow an application to inquire what type of double buffering exists on the system.

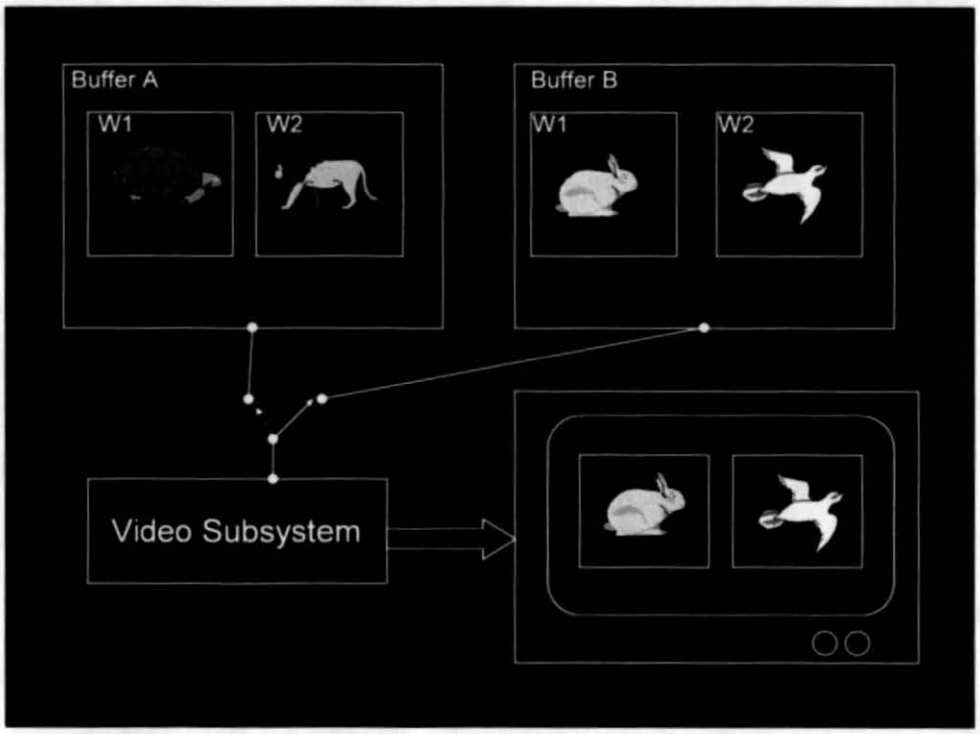
An application must not assume one particular implementation of buffer swap. If it does then it will not be very portable from system to system.

The currently displayed buffer is commonly referred to as the *front buffer* and the buffer that is being updated is referred to as the *back buffer*. After a buffer swap the names reverse. After a swap, the new back buffer's contents may be the updated picture or the previous picture or totally unknown. An application should assume that the data in the buffer after a swap is unknown and should clear before drawing unless it knows it is going to write every pixel in the buffer.

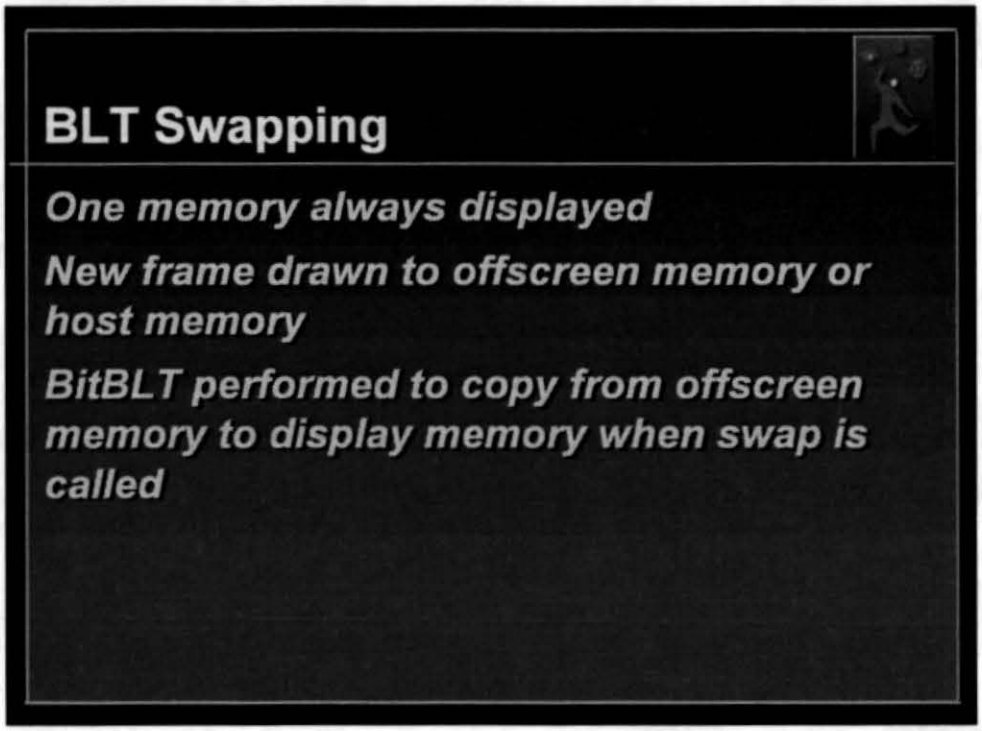



One common buffer swap technique is called page flipping. This exists on many 2D cards. This is the method where there is a register in the chip that identifies to the video display hardware what memory location is the pixel at (0,0). By writing this register, the picture can be panned on the screen. That is the reason why it was included in 2D systems. If the system has double the memory, then the buffers can be swapped by writing the register to display whatever half of the memory is to be the front buffer.

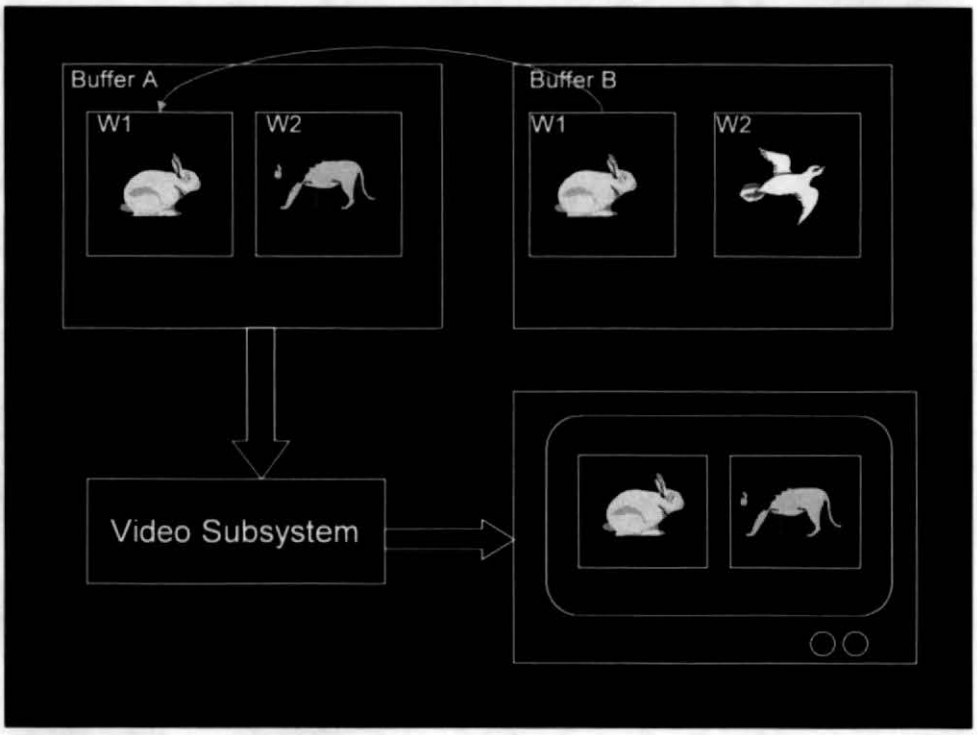
This method has the advantage of a buffer swap effectively taking no time at all. The disadvantage of this method is that if the 3D window does not cover the entire screen then all the rest of the information has to be copied to the back buffer prior to the page flip. This gets very complicated if there are multiple 3D double buffered windows all updating at different rates.




This example shows the basic concept of page flipping. There are two buffers in memory A and B. There is then a switch that can select either Buffer A or Buffer B to be displayed.




Another buffer swapping technique is BLT swapping. This technique is where the hardware BitBLT engine is used to copy the back buffer to the front buffer. The back buffer can be in the video memory or can reside in any memory that the BLT engine has access to, including host memory. After the picture is drawn and swap is called the BLT engine will copy the back buffer to the front buffer. This has the advantage that the back buffer can reside multiple places. There is no problem with synchronizing between multiple 3D double buffered windows. It has the disadvantage that is can be extremely difficult to time the swap so that there is no visual effect called *tearing*. Also the BLT does take time since it is a memory to memory copy.




This example shows the technique of BLT swapping. Buffer A is always the front buffer. Buffer B is always the back buffer. When the update to Window 1 is complete, the picture is then BLTed from the back buffer B to the front buffer A and that is what becomes visible on the screen.

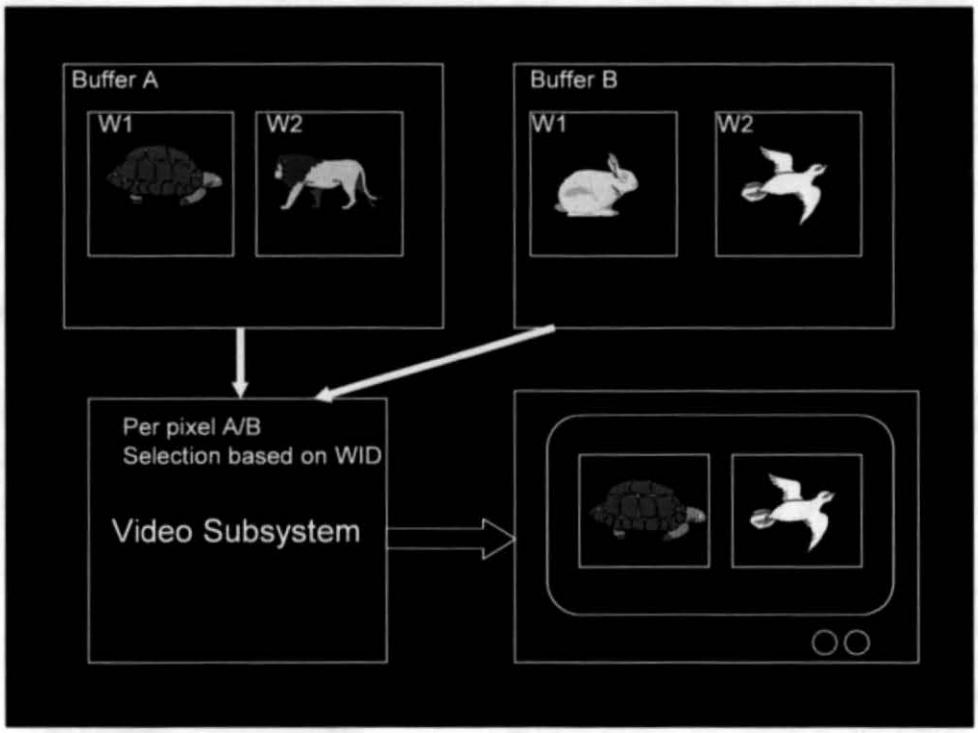
## Independent Window Buffer Swaps

*For all pixels, both buffers available to video subsystem*

*At video time, for each pixel select either A or B buffer for display (based on WID)*

*Buffer swap happens* instantaneously

A different technique of buffer swapping was developed for high-end UNIX workstations and is now available on graphic cards for WinTel architectures. This technique is called *Independent Buffer Swaping*. As the name implies, each window that is double buffered can be swapped independently. This is accomplished through the added feature of the Window IDs. Both buffers go to the video subsystem along with the Window ID. The Window ID is used as an index into a table which contains whether that buffer should be displaying from Buffer A or Buffer B. The video subsystem can then select on a pixel-by-pixel basis which buffer to display. Buffer swaps are accomplished by updating the table and hence windows, like page flipping, can be swapped *instantaneously*.




This example shows independent buffer swapping. Data from both buffers goes into the video subsystem. Included are the WIDs associated with each pixel. The video subsystem checks its table as to which buffer is to be displayed for which window. In this example the table is such that Window 1 is displayed form Buffer 1 which Window 2 is displayed from Buffer B.

## Depth Buffer

***Data stored not specified by OpenGL standard***

***Expect***

- Zp (transformed Z / W)
- W

Another area that varies in OpenGL implementations is the data that is stored in the Depth Buffer. Normally this is called the Z-buffer, but that implies what is actually stored in there. The data that is stored in the depth buffer is not explicitly defined by OpenGL. Only the characteristics of how things act is specified.

There are two types of data that can be stored in the depth buffer. The most common is the Z perspective value. This is the transformed Z divided by W when you take the vertex and multiply it by the 4x4 composite transformation matrix. This has the characteristics of have more precision to distinguish depth for objects close to the eyepoint and decreasing precision as things are farther from the eyepoint.

The other type of data stored in the depth buffer is W. This is the homogenous transformation W, provided that it is a perspective transformation. This data has the characteristic that the precision does not vary but is constant moving away from the eyepoint. This has been the traditional data stored by simulation systems as there must be a great depth range when doing flight simulators.

Direct3D recognizes the need for both types of data and allows the application to specify which is used in the depth buffer.

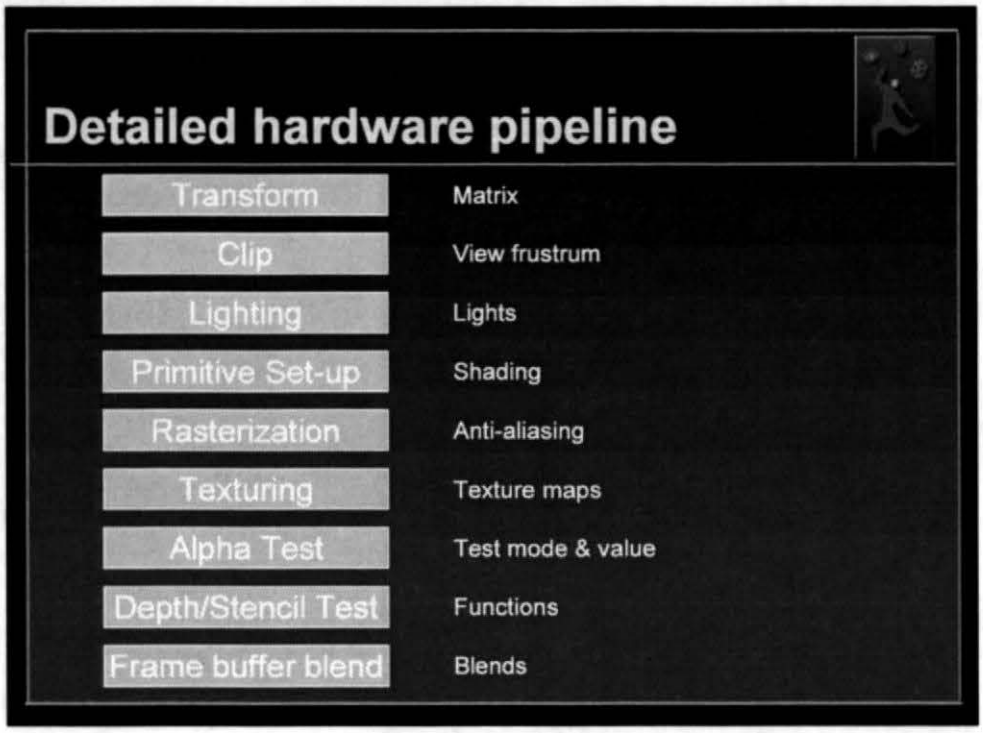
## Interaction between OpenGL calls and the pipeline

***State changes can be costly***

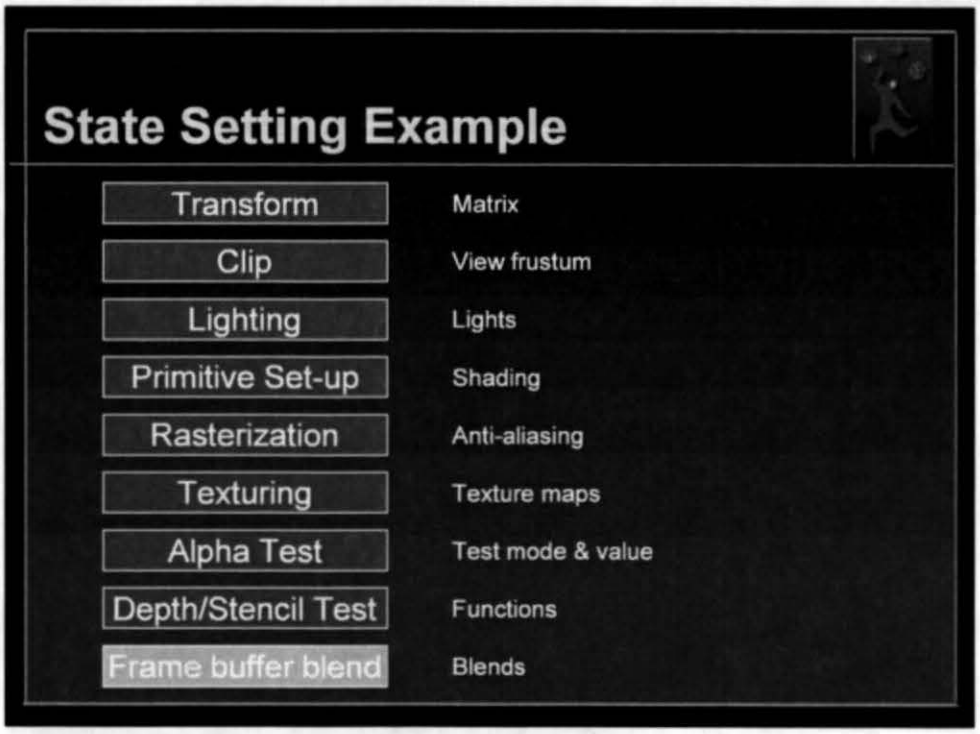
- many stages working simultaneously
- may have to have all stages down to state to be changed idle before able to change the information
- each stage must start up again (i.e. long latency)

The focus has been on possibly implementation differences which can have an effect on high performance graphics applications. This next item points to an area of hardware implementation which can have a dramatic impact on application performance if the graphics calls are made in the improper order.

Hardware can be thought of as *pipelined* which is where there are many different computing units or stages that can work in parallel. As long as all the stages are busy they graphics hardware is being used to peak efficiency and performance is achieved. When all the stages cannot be working simultaneously then all the parallelism is lost and so is the performance.




Here is a example diagram of many stages of a graphics hardware pipeline. It consists of nine different stages. As long as all are busy great performance is achieved. Each stage has different state information that it uses to do its work. What flows down the pipeline is the graphics vertex information. To change any of the state information all the previous graphics information must be past that stage prior to changing the information. This is sometimes referred to as *flushing the pipeline*.




This diagram shows the case where the depth function needs to be changed. The pipeline has to be flushed all the way down through the depth/stencil test. The function value can then be changed. After the value has been changed new graphics information can start flowing in the pipeline. The flushing and filling of the pipeline takes time and reduces the performance of the hardware.

## Projections for the workstation

### *Graphics*

- faster
- cheaper
- feature rich
- indistinguishable from Unix graphics subsystems

### *Windows NT*

- with help (pressure from you) will become truly *3D-aware*

Where is graphics going on the workstation? As all things, it is going to get high performance, lower cost and have many more features added to it. Essentially you can no longer tell the difference between graphics on Windows NT platforms and Unix platforms.

The thing that needs to happen is that Windows NT needs to become truly *3D-aware* and with your requests to Microsoft it will happen.

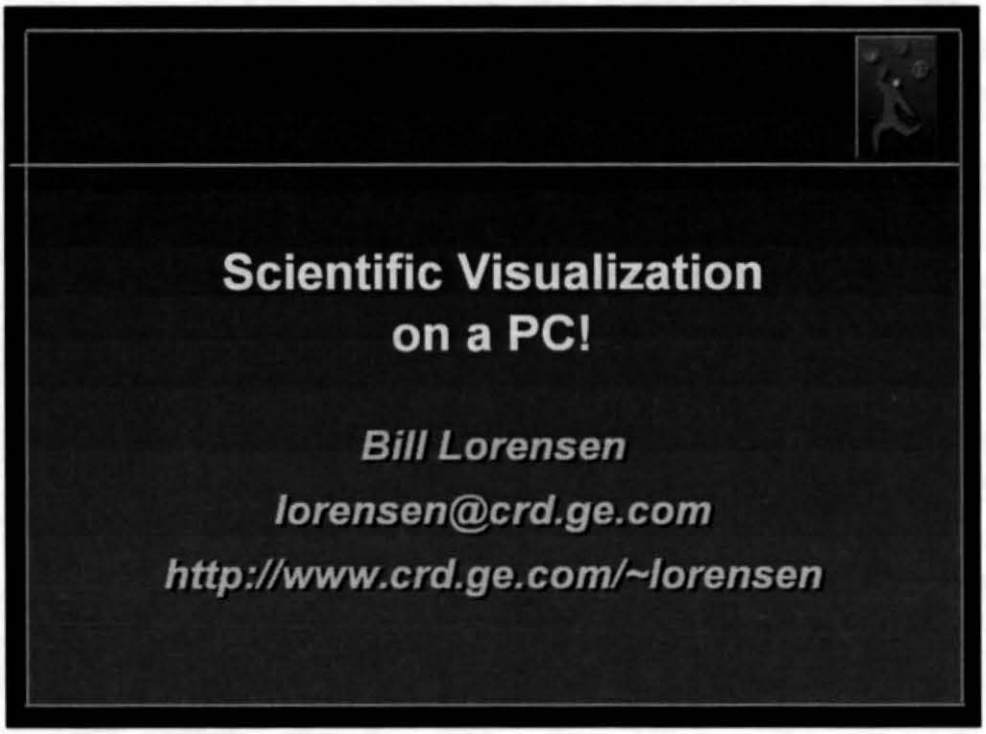
## Summary

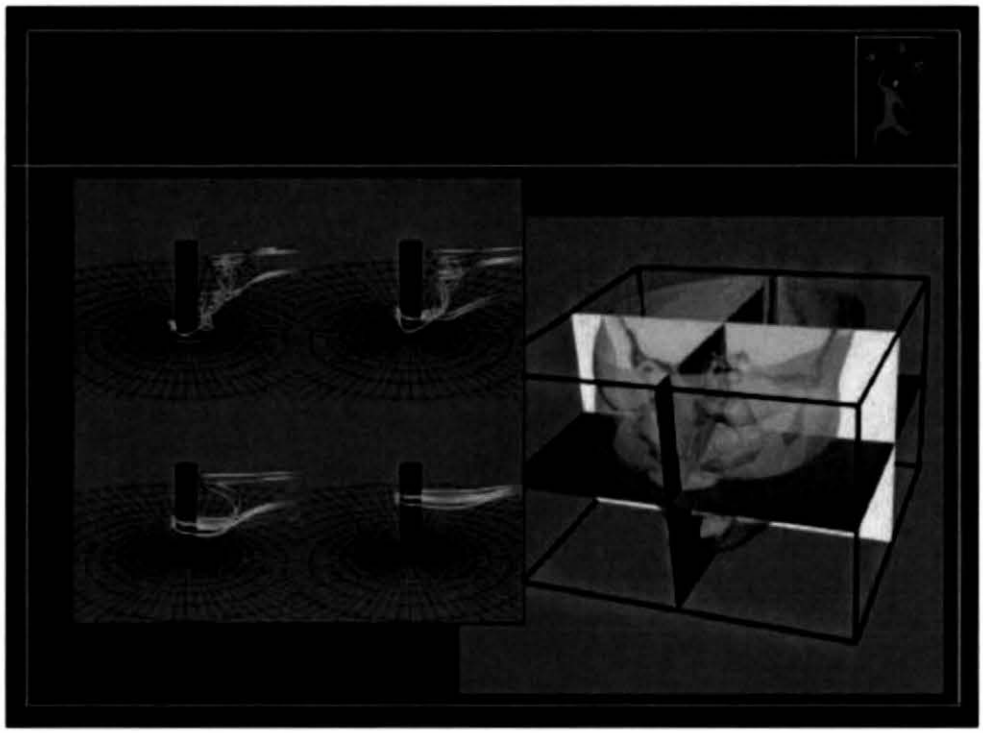
*Just because there is a workstation 3D subsystem does not guarantee high performance*

*Never ASSUME functionality based on a single implementation of OpenGL*

*Knowing the basics about hardware features will greatly enhance application performance and portability*


Questions


Scientific Visualization
on a PC!
Bill Lorensen
lorensen@crd.ge.com
http://www.crd.ge.com/~lorensen



## Outline

- *Motivation*
- *Visualization Vs Computer Graphics*
- *Visualization Requirements*
- *Visualization Software*

## Outline

- *The Visualization Toolkit*
- *Case Study*
- *Benchmarks*
- *Conclusions*

## Motivation

***The Visualization Community is no longer in control***

- Technology drivers have changed
- Customers expectations are high

***but...we do have lots of software experience***

- OO is a proven technology
- We have a large installed base
- We know our application domain

## External Forces

*Internet*

*Wintel*

*Standards*

*Language Wars*

## Internet

***The right information, to the right person, at the right time…***

***In the future,***

- Most applications will be Internet-ready
- Finance market will solve security problem

***But,***

- Performance remains an issue

## Wintel

***Microsoft OS's and API's dominate***

***Intel processors dominate***

***Scientific Visualization is a small player compared to***

- Entertainment
- Word Processing

***What can we leverage???***

## Standards

*OpenGL*

- Available on Unix and PC's
- Cheap accelerator boards
- Impacts graphics and volume rendering

*Java*

- Write once, run anywhere (???)
- Performance

## Language Wars

***Java Wars***

- Sun vs Microsoft

***Java vs C++***

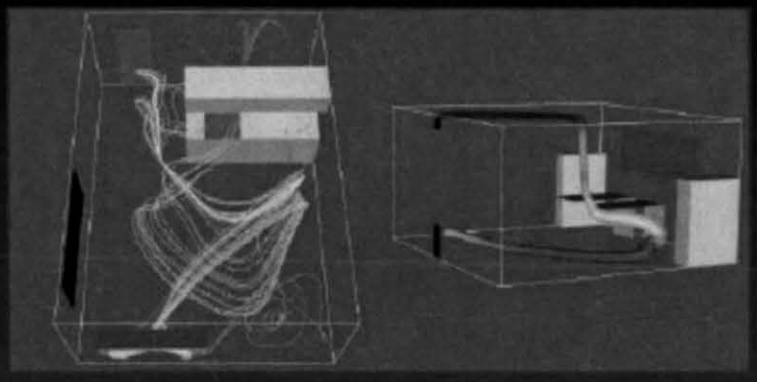
- Portability vs Performance

***Java3D***

- Sun vs the world

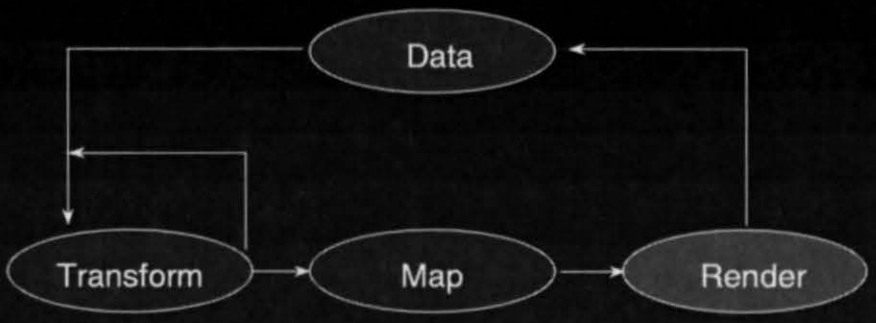
***We need strategies to protect our software investment***

## Visualization

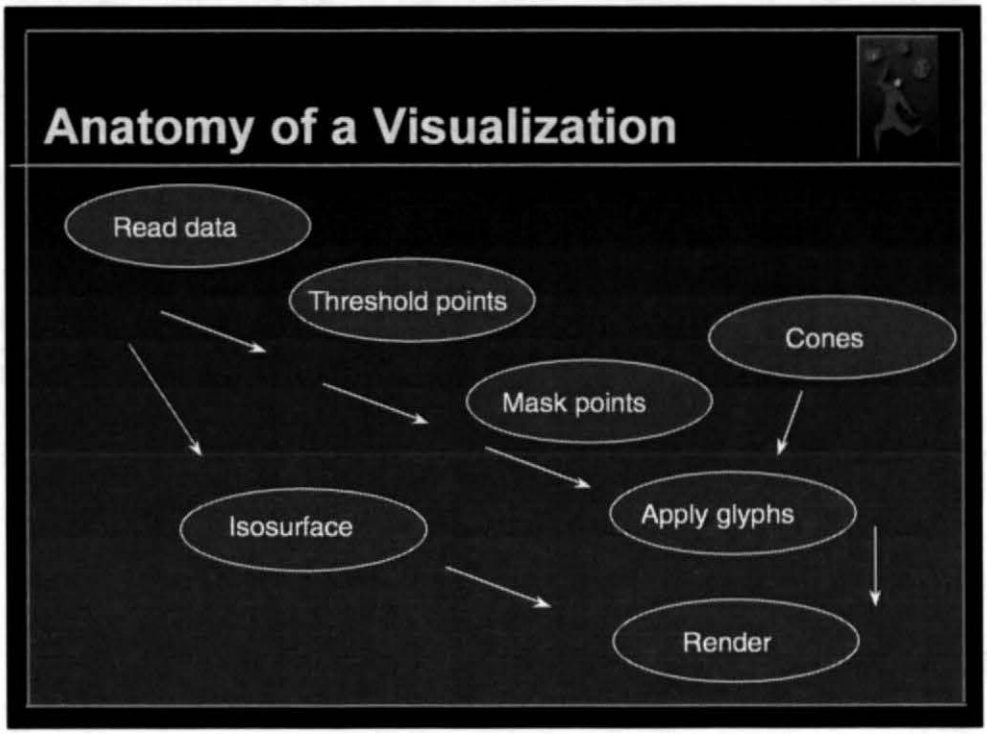


*The transformation of data into images*

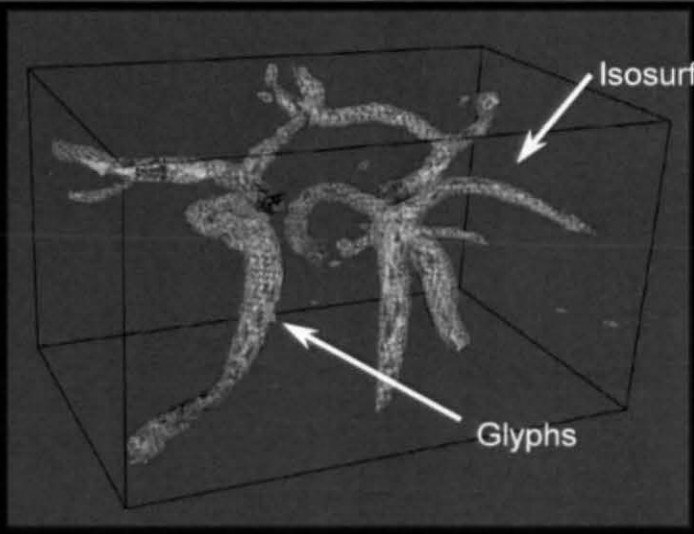
## Visualization vs Graphics



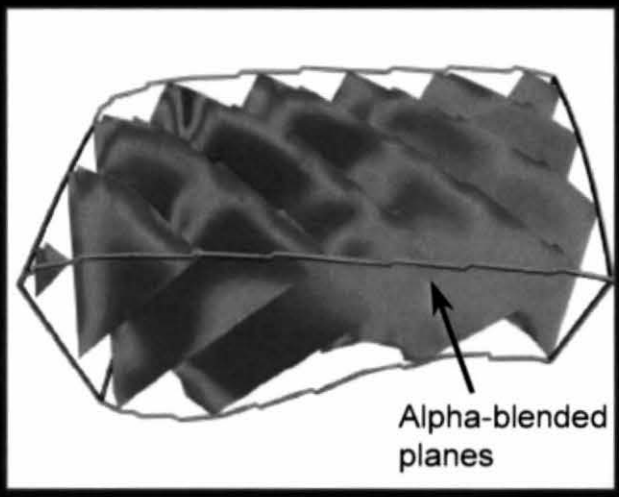

***The transformation of data into images***


Anatomy of a Visualization
Read data
Threshold points
Cones
Mask points
Isosurface
Apply glyphs
Render

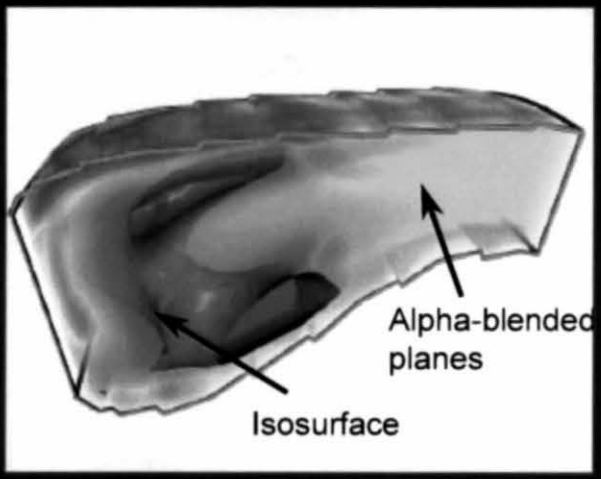
## Glyphs and Isosurfaces



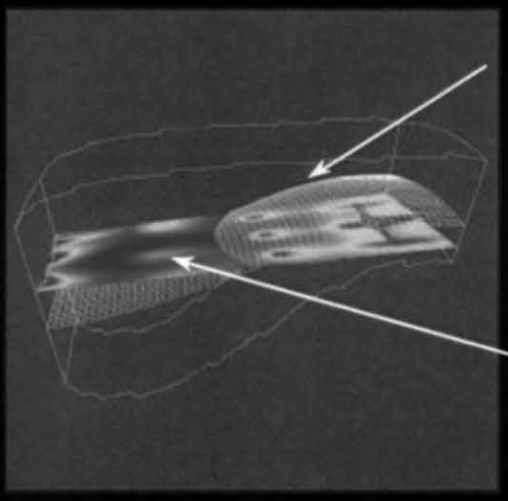

## Slicing



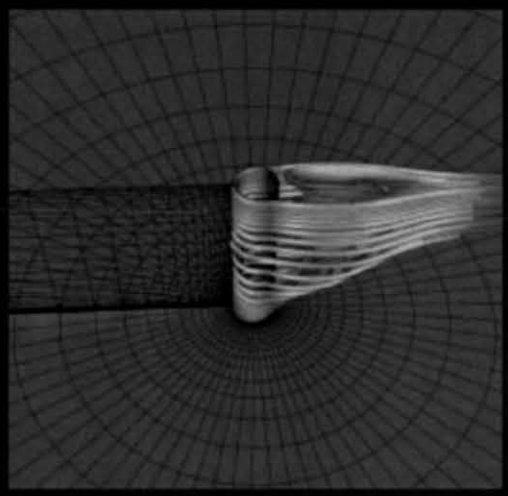
## Volume Rendering using Alpha Planes



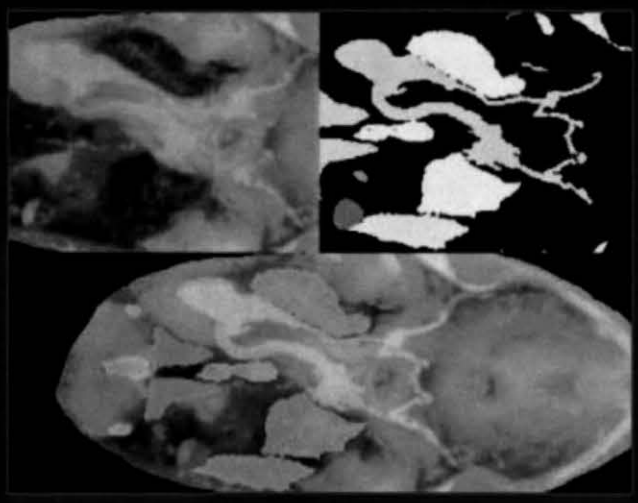

## Data Probing



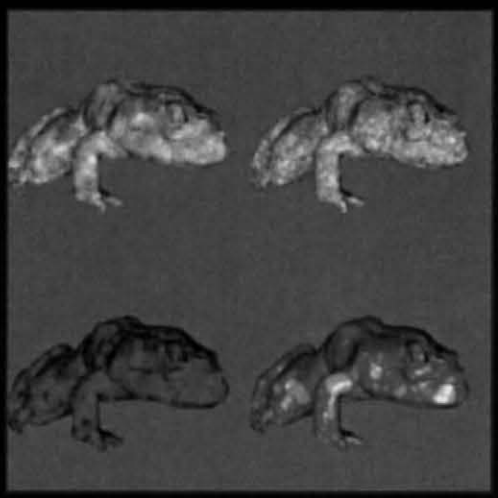
## Stream Polygons



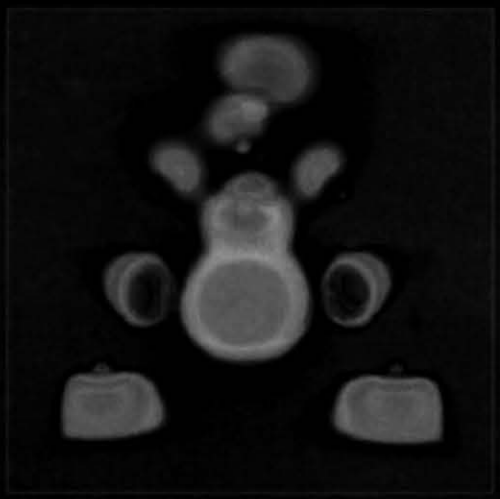
## Texture Mapping



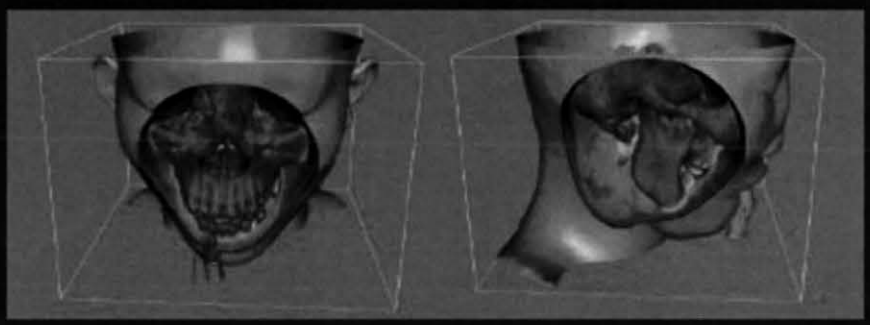
## High Quality Software Rendering



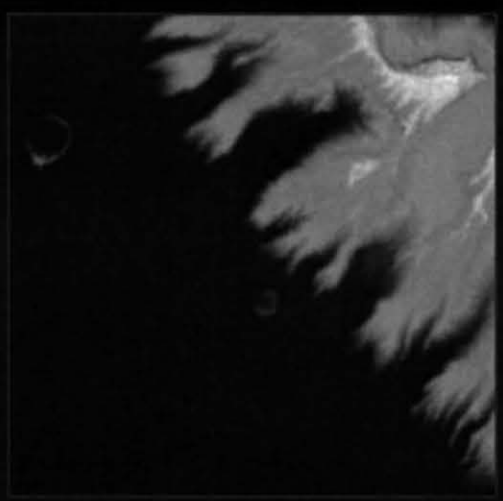
## Volume Rendering



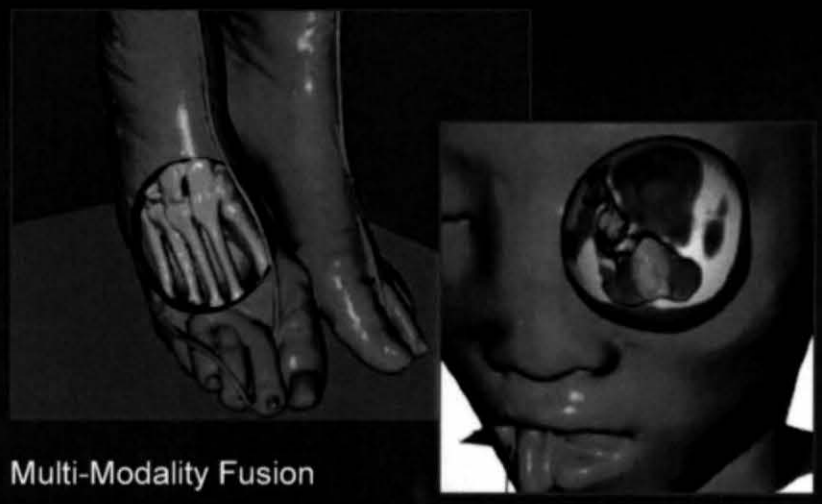
## Surface and Volume Rendering



## Triangle Decimation



## Tissue Lens



Multi-Modality Fusion

## Processor Requirements

***Memory***

***Multi Processing***

- Imaging
- Volume Rendering

***CPU Speed***

## Graphics Requirements

***Graphics Acceleration***

- Polygons, mostly triangles
- Texture Mapping

***Triangle Stripping***

- Lots of them!!

***Per vertex Colors and Alpha***

## Graphics Requirements

### *2D Texture Mapping*

- Images
- Volume Rendering
- Clipping

### *3D Texture Mapping*

- Volume Rendering

## Graphics Requirements

*Image buffer read*

*Z Buffer read*

- Polygon assisted ray tracing

## Systems versus Toolkits

*Systems*

- Self-contained
- Often turn-key
- Great reuse
- Integrated user interface
- All or nothing

## Systems versus Toolkits

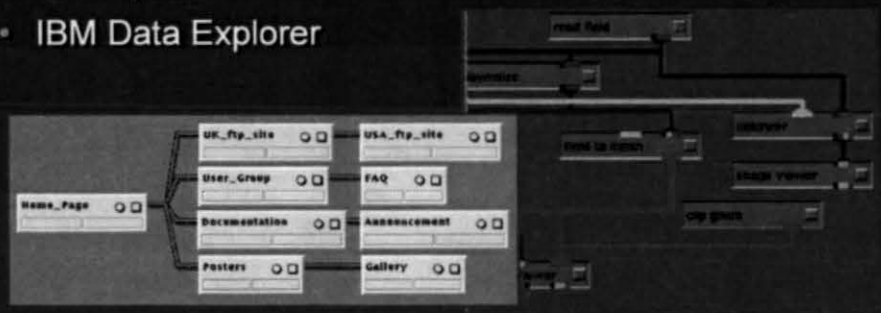
*Toolkits*

- More than a library
- Includes an architecture
- Use only what you need
- Independent of user interface

## Systems

### *Examples*

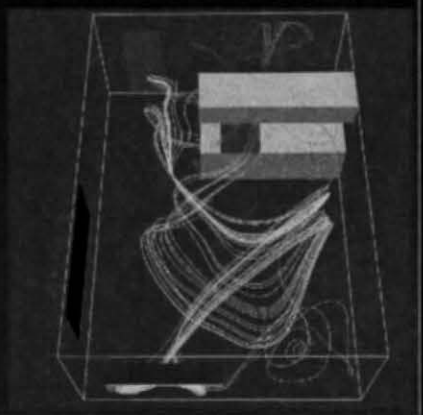
- AVS
- Iris Explorer
- IBM Data Explorer



## Toolkits

***Examples***

- The Visualization Toolkit
- Inventor
- ISG's IAP



## The Visualization Toolkit

*Started as an example implementation for a text book*

*Implemented in C++*

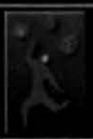
- runs on Unix workstations and PC's

*Graphics abstractions*

*Visualization pipeline*

*No interpreter (initially)*

## The Visualization Toolkit



***Interpreters added through automatically generated wrappers***

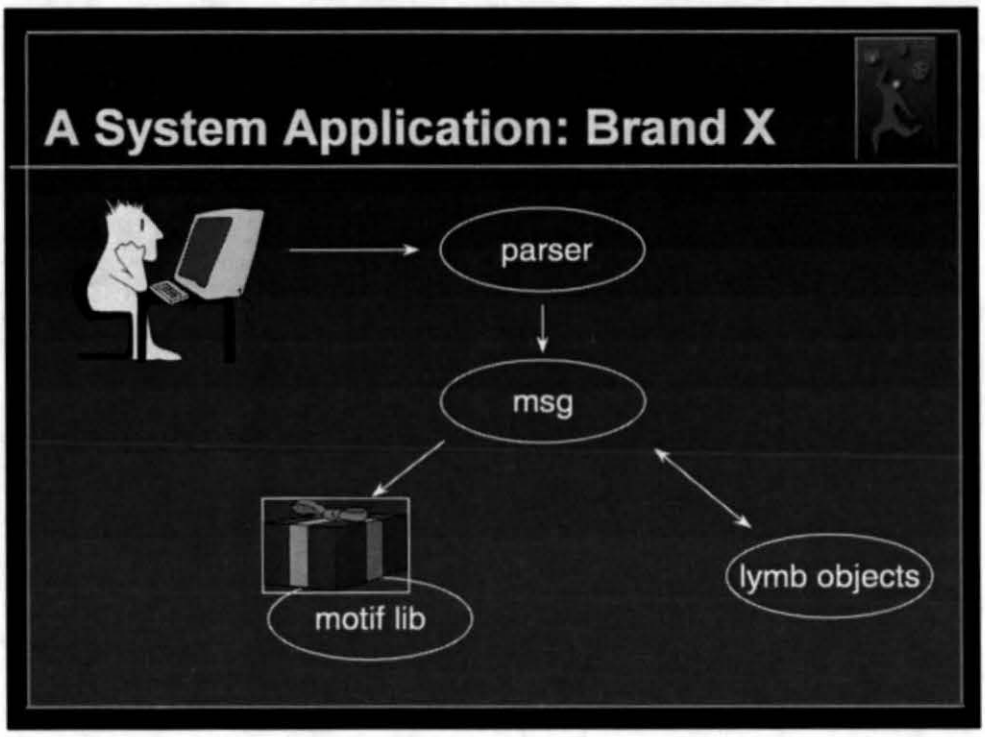
- tcl / java / python...

***All documentation contained within code***

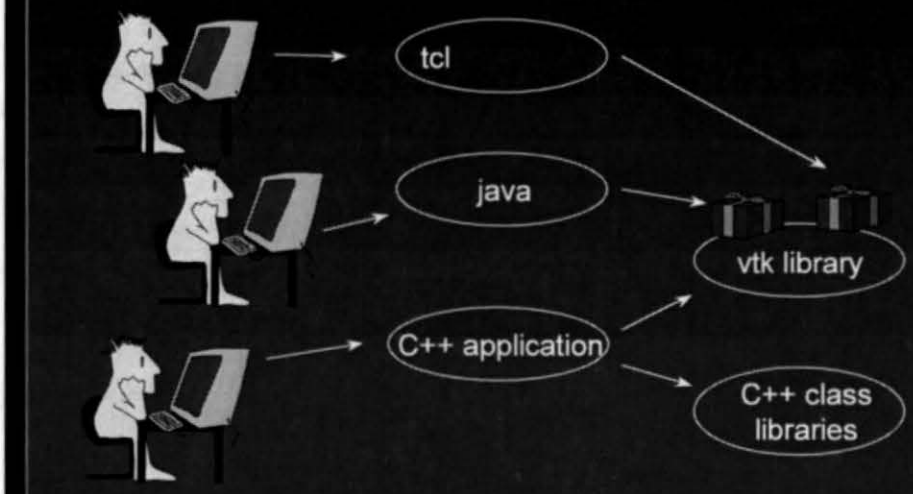
- makes for easy man page, html, etc.... generation

***Source code available via Internet***

***http://www.kitware.com/vtk.html***


A System Application: Brand X
parser
msg
motif lib
lymb objects

# A Toolkit Application: VTK



## Lessons Learned

***Object-oriented is good***

- if you enforce a methodology

***Interpreters are good***

- but don't invent your own

***Abstractions are good***

- they protect against future changes beyond your control

## Lessons Learned

*C++ is mature*

*Isolate the user interface*

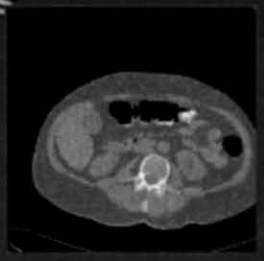
*Keep it simple!*

*Watch those features!*

*Proprietary is bad!!!*

## Case Study: Virtual Endoscopy

*A technique that provides diagnostic information of internal passages using 3-D computer models generated from Computed Tomography (CT) or MRI studies*



## Motivation

*Why develop a VE software application on a PC*

- Familiar computing environment
- Reduce computer-phobia
- Evaluate feasibility of VE
- Patient education

## VE on a PC !

- *CPU speeds (driven by mass market)*
- *Graphic accelerators (driven by games)*
- *Affordability (CPU & memory)*
- *Software portability (OpenGL, vtk, tcl/tk, C++)*
- *Combine application specific components with standard components*

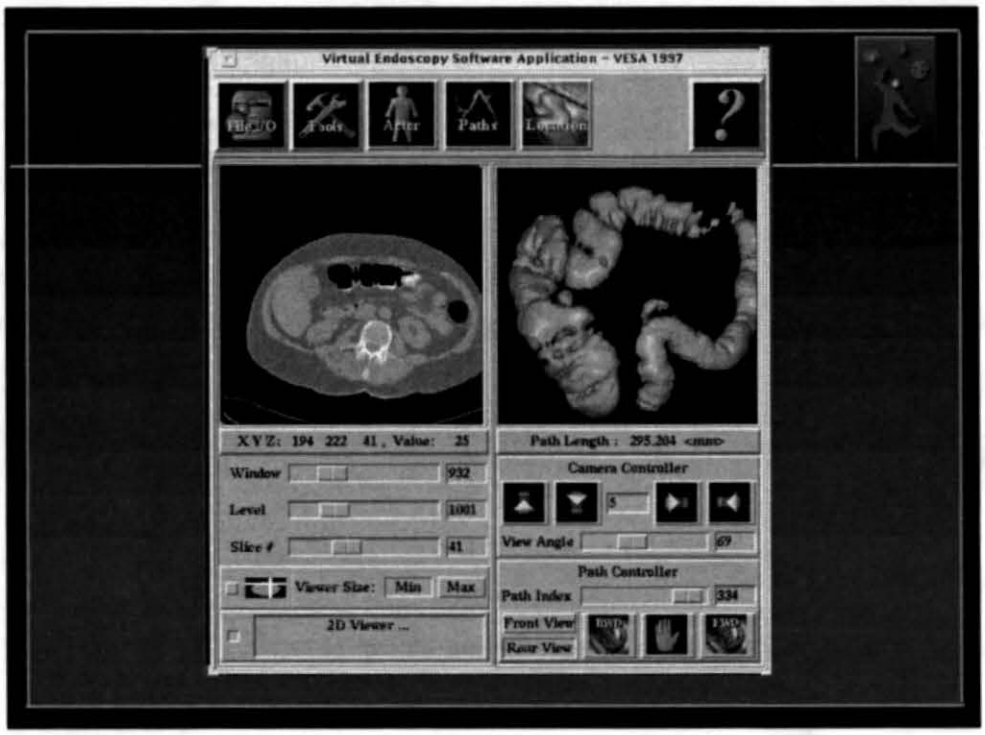
## Portable Software Components

*Visualization Toolkit (vtk) for 2D & 3D visualization*

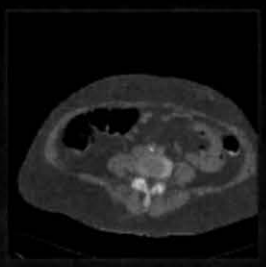
*Tcl/Tk for user interface*

*OpenGL for graphics*

*C++*


Virtual Endoscopy Software Application - VESA 1997
?
XYZ: 194 222 41 , Value: 25
Path Length : 295.204 <mm>
Window
932
Level
1001
Slice #
41
Viewer Size:
Min
Max
2D Viewer ...
Camera Controller
5
View Angle
69
Path Controller
Path Index
334
Front View
Rear View

## Segmentation



CT Image

Threshold

Connectivity

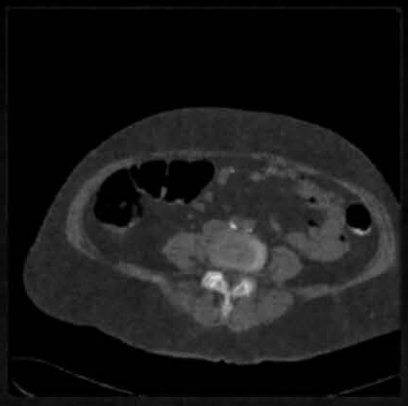
## Surface Generation

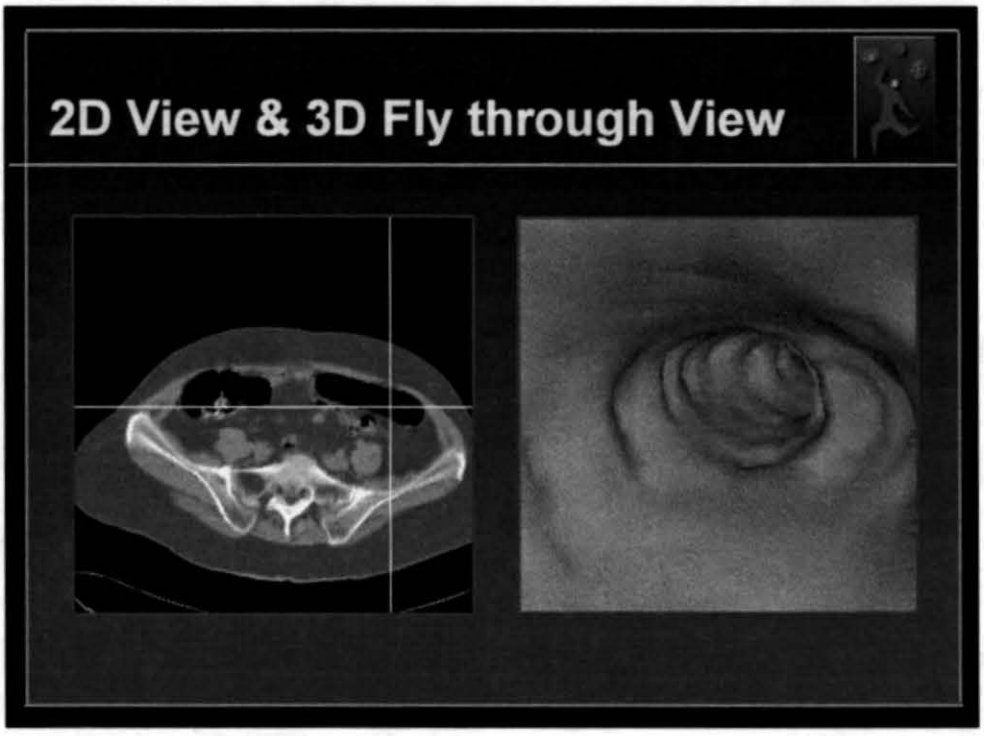
Marching Cubes

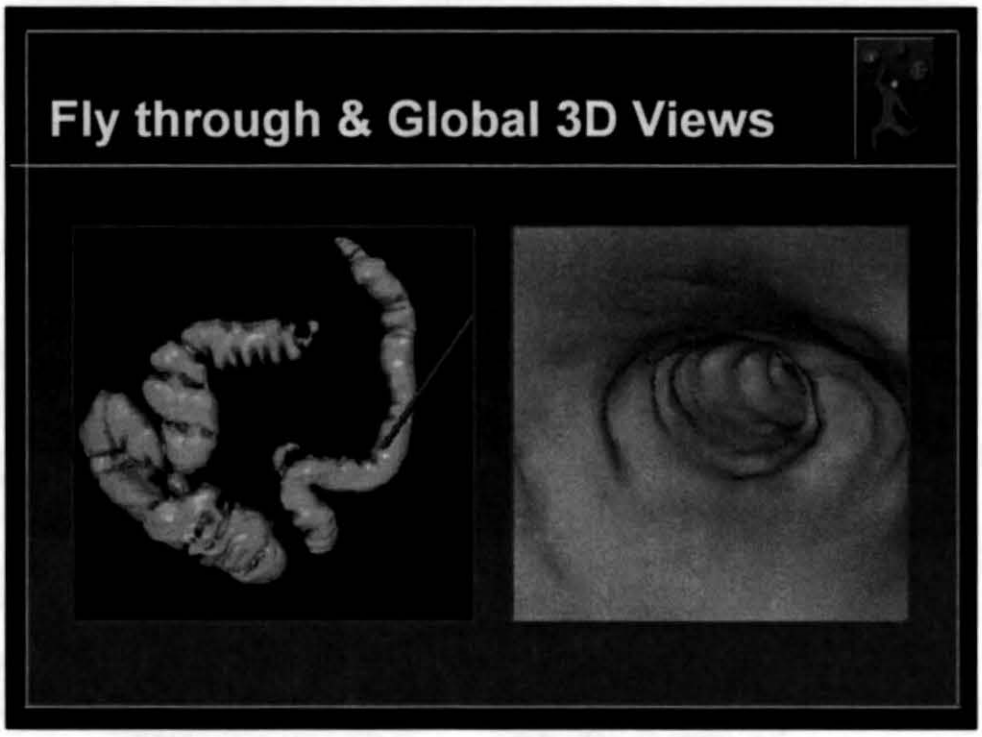
Decimate

Smooth

## Path Planning




2D View & 3D Fly through View




The is an added value to VE compared to CE

## Performance

Data Size: 256x256x114

| | 133 MHz PC | 200 MHz PC | Onyx 10000 |
|---|---|---|---|
| Surface Generation | 6:40 min | 3:26 min | 57sec |
| Path Planning | 45 sec | 21 sec | 9 sec |
| Navigation | 1.9 fr/sec | 4 fr/sec | 20 fr/sec |

This is elapsed time not CPU time ...

## Conclusion

*VE runs on UNIX and on a PC*

*VE offers a non-invasive alternative to conventional endoscopy.*

*Clinical validation of VE is still needed*

Does Virtual Endoscopy on a PC help in any way?

## Acknowledgments

*Brigham & Women's Hospital*
*Bowman Gray School of Medicine*

*DARPA TRP #F41624-96-2-0001*

## Visualization Benchmarks

*Go to:*

*http://www.kitware.com/vtk/vtkmarks.html*

## Conclusions

*PC's can do visualization now*

*Graphics performance will increase dramatically*

*Sit back and enjoy...*

## Conclusions

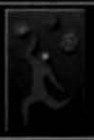
*More information at:*

*http://www.kitware.com/vtk.html*

*and*

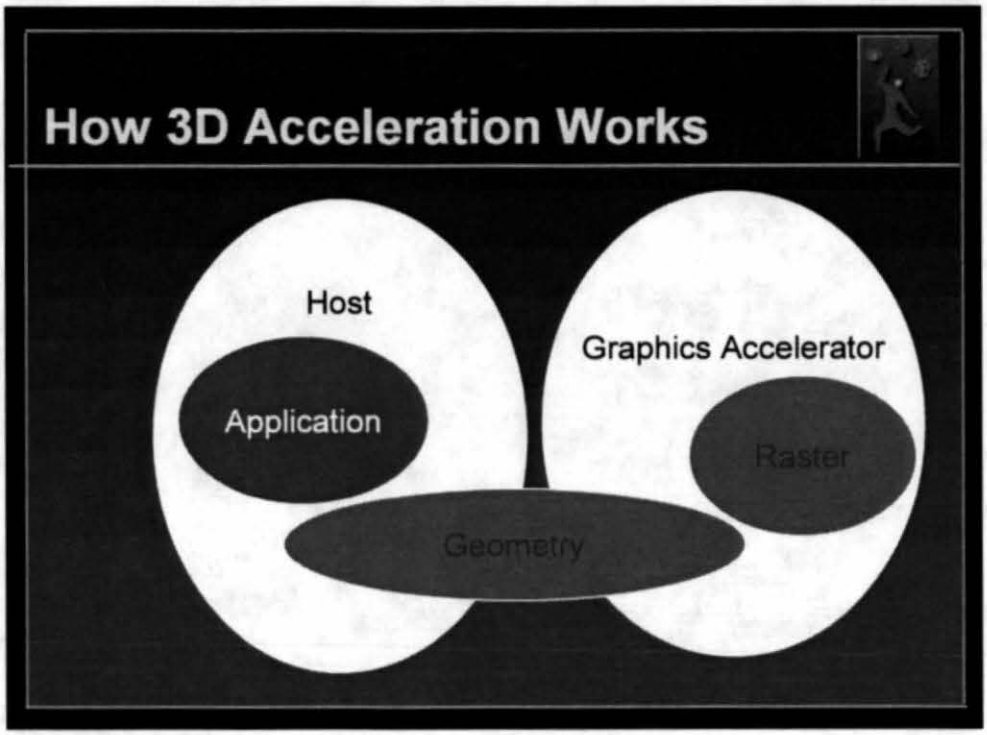
*http://www.crd.ge.com/~lorensen*


Questions



# Factors in Graphics Performance

*Dr. John M. Danskin*
*Dynamic Pictures, Inc*




Applications run on the host.

Raster processing runs on the graphics accelerator.

Geometry runs on either or both.

The point is that this is a multiprocessing system. The problems in multiprocessing systems are well known:

- Communications
- Load balancing

We are concerned with issues in the efficient, cost-effective implementation of graphics acceleration hardware-software subsystems.

## How 3D Acceleration Works

***Application Runs on Host***

***Raster Runs on Accelerator***

***Geometry happens on either or both***

This is a multiprocessor system

- Issues are:
  - *Managing Data Flow / Communications Resources*
  - *Maximizing Parallelism*

## Factors in Graphics Performance

*Host Performance*

*Memory Subsystem Performance*

*I/O Bus Performance*

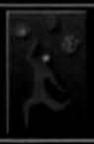
*Multi-Processor Host Issues*

*Geometry Accelerators*

*Host and Accelerator Parallelism Issues*

- Host performance:
  - issues are work distribution, parallelism with accelerator, and communications overhead (memory bandwidth and I/O bus bandwidth).
- Memory Subsystem Performance
  - What kind of memory performance is necessary for fast graphics?
- I/O bus performance
  - Same
- Multi-Processor Host Issues
  - When do you want more than one host processor?
  - What do multiple host processors buy you?
- Geometry Accelerators
  - What are the windows of opportunity for geometry accelerators?
- Multiple Graphics Accelerators
  - What architectures support multiple graphics accelerators?
  - How can we build scaling into accelerator architectures?
- Host and accelerator parallelism issues:
  - How can we ensure that hosts and accelerators operate in parallel?

## Host Performance - Limiting Factors



*Memory contention*

*I/O Bus contention*

*Synchronization with Accelerator*

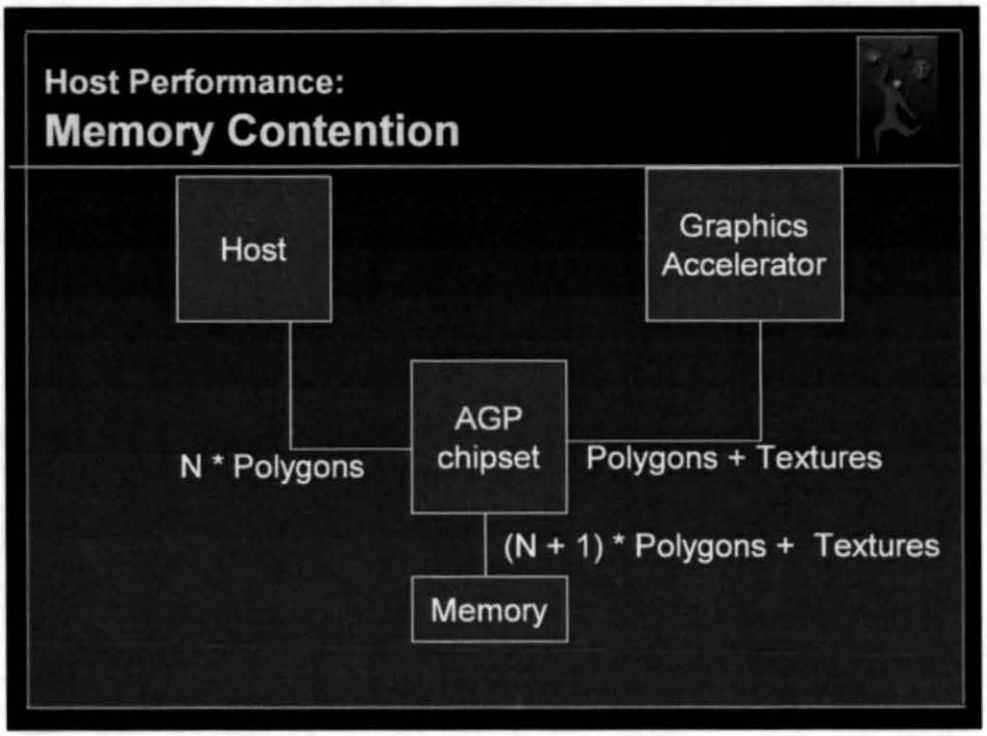
*Application*

Graphics performance is often host limited.

Sometimes this is because most of the graphics processing occurs on the host and the host is slower than the graphics accelerator.

Sometimes this is because the application running on the host is itself doing a non-trivial amount of computation.

Often the host is apparently slower than it could be because it can't get the memory bandwidth it needs, or because it is fighting with the graphics processor for the I/O bus, or because it spends too much time synchronizing with it's accelerator. This synchronization is especially egregious because it removes parallelism, and because it is an easily removed interface issue.



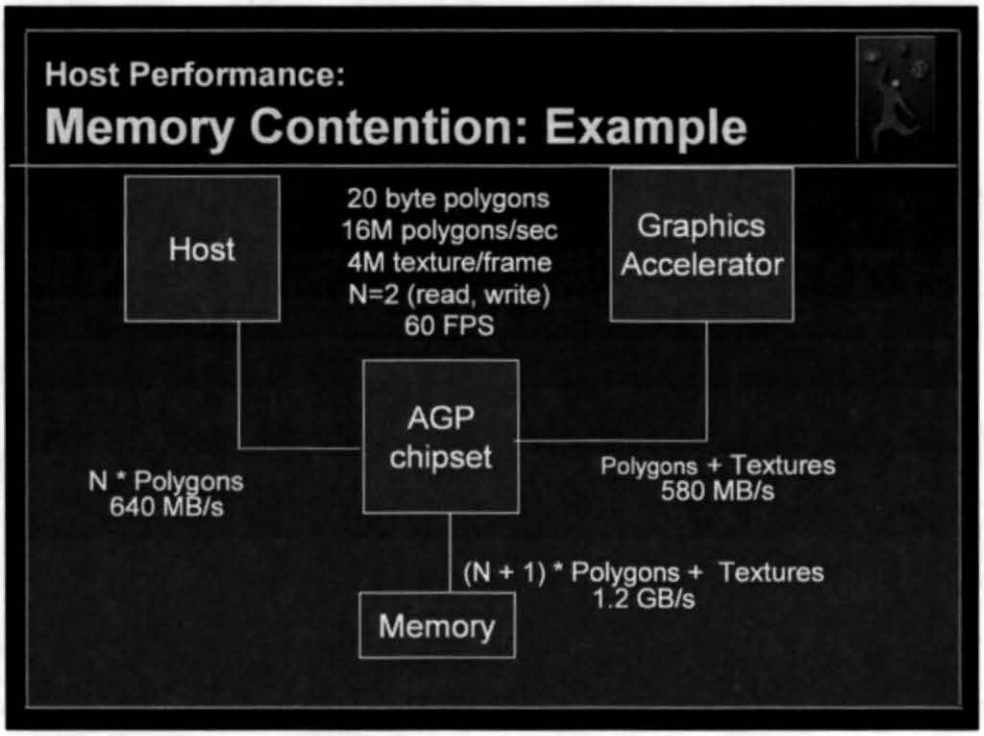

**Host:**

In this example, the host reads and writes polygons to main memory. The number N is the number of times the polygons are read or written to or from the main memory. In the case where the polygon dataset together with data necessary to render it, and the data the application uses to generate a frame of data is larger than the lowest level cache, the smallest possible value of N is 2. A competent 3D implementation will block intermediate calculations so that they fall comfortably inside the first or second level cache.

**Graphics Accelerator:**

This graphics accelerator is reading polygons and textures over it's graphics bus. This is a currently popular architecture. The accelerator will cache textures in a small on-chip cache, to prevent loading individual texels multiple times in a frame, but this small onchip cache cannot be large enough to prevent loading all of the textures referenced in every frame.

Alternatively, at the cost of a few more RAM chips, the graphics accelerator could store textures in a local memory, and avoid some of the bandwidth across the graphics bus and to the system memory. This is what we'd expect to see in a high-end architecture.




Example:

Polygons per sec = 16M

Bytes per polygon = 20 ( xy rgb Z U V )

N = 2

Texture bandwidth = 4MB / frame

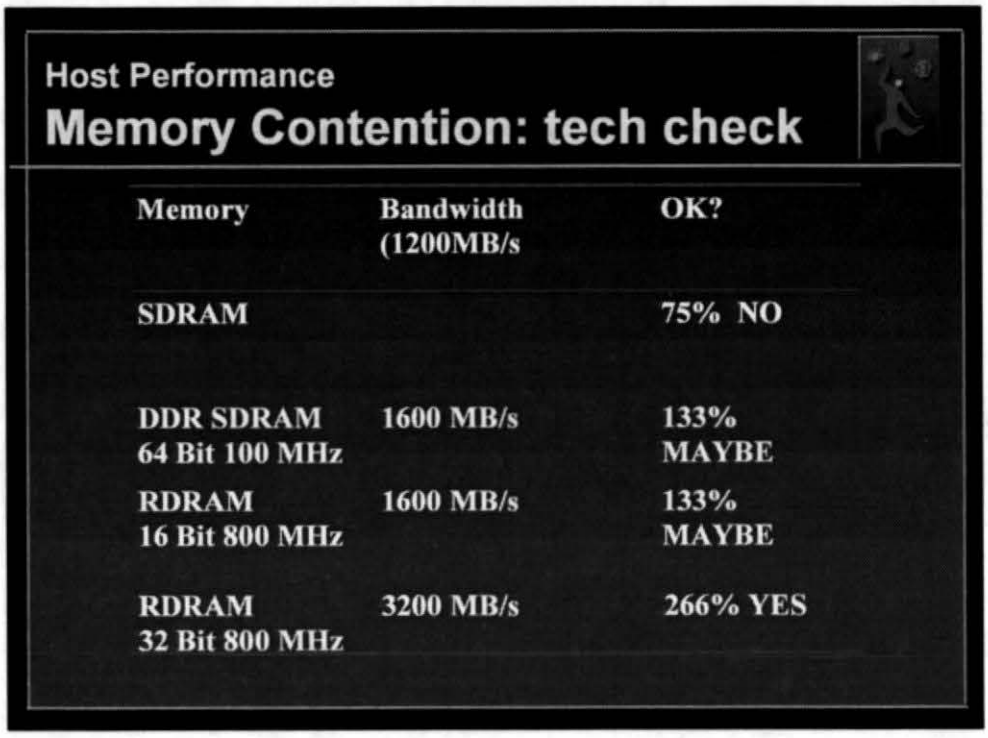
Frame Rate = 60 FPS

Required:

Host Bus bandwidth = 580 MB/sec

Graphics Bus Bandwidth = 640 MB/sec

Memory bandwidth = 1200 MB/sec

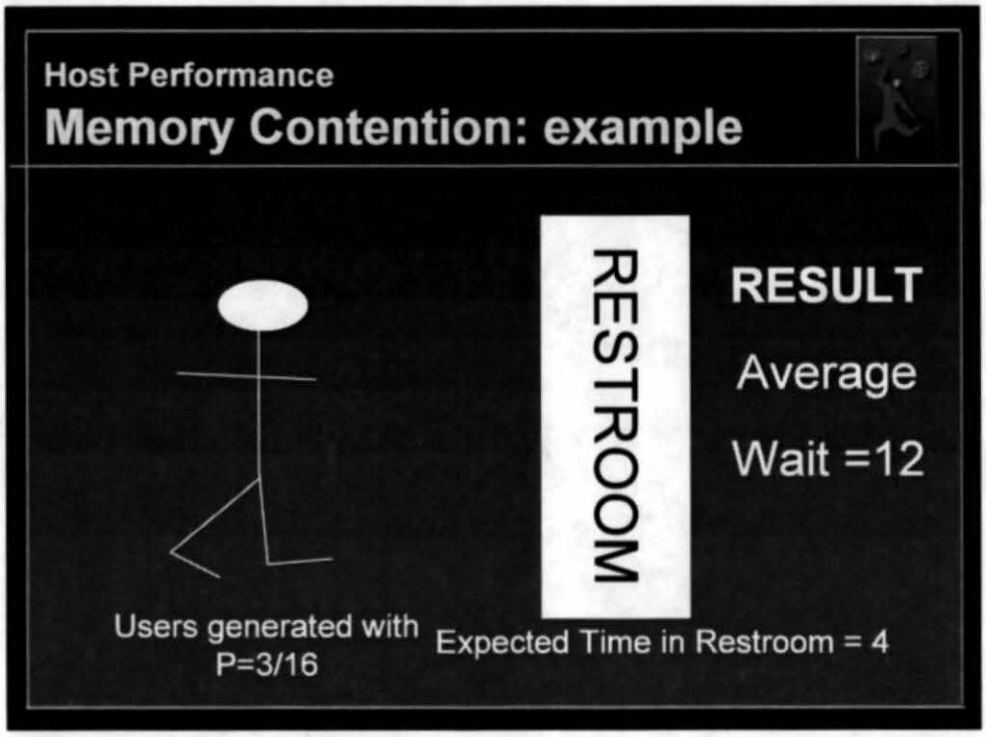



Obviously the current 100 Mhz SDRAMS are too slow because their peak bandwidth is less than the required bandwidth.

Why are the DDR SDRAMS and single bank RDRAMS marginal at 30% overcapacity? Without delving into queueing theory:

> If a facility is busy 75% of the time, then when you need to use the facility, you have a 75% chance of having to wait. In fact, there is a large chance that there are several people in line in front of you and you'll have to wait a long time. This degrades your performance considerably.

There are ways to offload the system memory. One is to move the textures back into the graphics accelerator. Another is to store the accelerator's input queues in accelerator local memory. Both of these options increase the cost of the accelerator



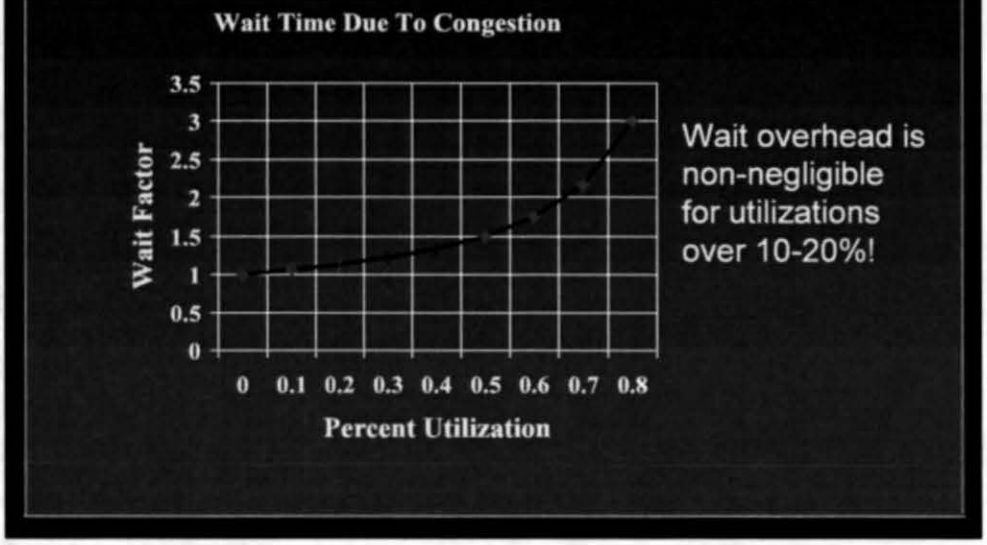

Using a result from queueing theory, if we have a restroom user generated at each time interval with probability = 3/16 in a uniform random distribution, and the expected time in the restroom of each user is a Gaussian distribution centered around 4, then the average wait for a customer is 12.

To the extent that restroom use is an important part of our agent's activities, our agents have lost 75% of their performance. This doesn't mean that we cannot achieve 75% utilization of this resource, it just means that we have to accept a significant degradation in the performance of our agents if we wish to do so. It might be cheaper to put in another bathroom.

To make the analogy explicit, We may be able to achieve 75% utilization of a shared memory bus, but we will have to accept significant degradation of our host processor performance in order to do so.

Why aren't our busses currently appropriately oversized? Because the current generation of PCs is tuned to run Microsoft office. 400 MHz PIIs are designed to run MS Office really fast.

## Memory Congestion:
### Example Continued




Wait overhead is non-negligible for utilizations over 10-20%!

This graph plots the wait factor, which is a multiplier for how long it takes to access memory, under the same assumptions as the restroom example from the previous slide.

With a finite number of agents arriving at random times, it is not possible to achieve 100% utilization of this resource. In fact, at 65% utilization, our agents are already degraded by 50%.

## Memory Congestion:

Impact on the Host

Performance degradation factor is:

$$f = (o + m) / (o + mw)$$

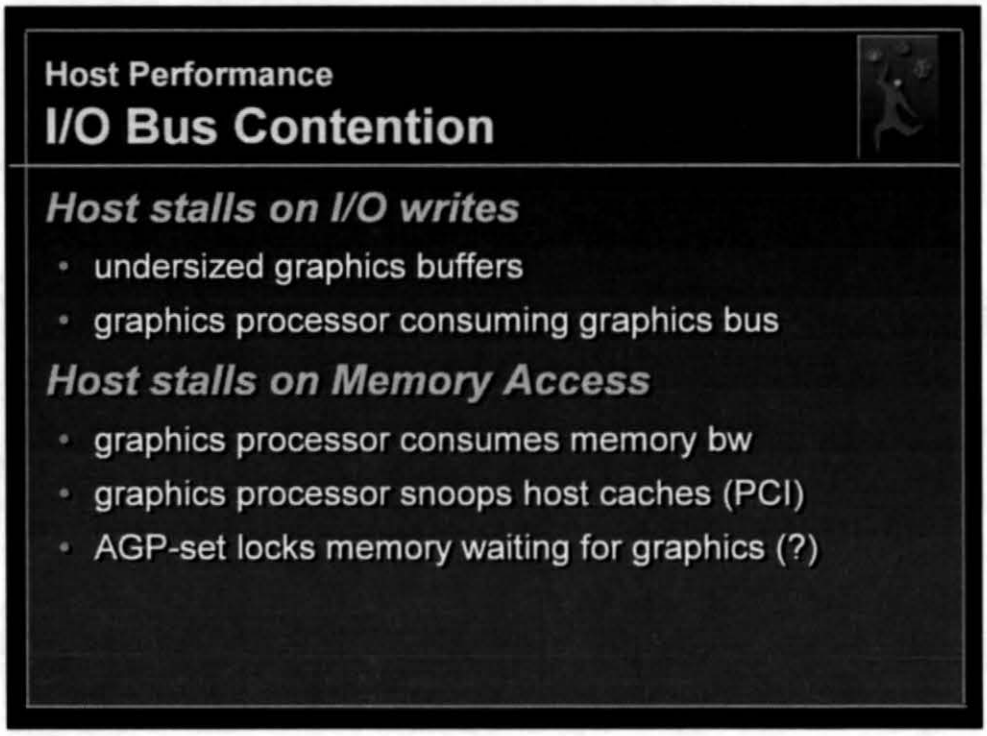
Where:

- ***f*** is the performance degradation factor
- ***m*** is the percentage of time spent doing accessing memory without contention
- ***o*** is the percentage of time spent doing other things, (without contention)
- ***w*** is the memory contention degradation factor

For games, Microsoft Office, and other traditional PC applications, ***o*** is near 1, ***m*** and ***w*** are near 0, and therefore ***f*** is near 1. Degradation due to memory contention is very small.

Memory contention may become more of an issue for games and game chips with AGP textures. Textures stored in system memory will increase memory contention noticeably.

For some workstation graphics applications, I'm estimating ***f*** < .5. This is based on comparing elapsed time with hand-counted-cycle time. Memory contention is a serious issue when PCs are used as workstations.




**I/O Writes:**

Hosts are generally bursty. The host tends to to send polygons considerably faster than the graphics accelerator can render them for a while, then it stops sending polygons and thinks about the next frame. If the queues (on the accelerator or in system memory) are a lot smaller than one frame's worth of data, the host will end up waiting for the graphics accelerator, wasting time. This is especially bad when there is a hardware stall, because ALL of the busses and hosts in the system will lock up behind the stall. If the host can wait in software, there is the possibility of letting other processes try to make progress. The best thing is to just have big queues.

Another problem is that if the graphics processor can be using the graphics bus for texture accesses, which can delay the host, if the host is trying to dump a lot of polygons over the I/O bus.

**Memory Access:**

In the (old, outmoded, past) PCI case, every graphics processor generated cache snoops. If these snoops hit in your cache, they could lock the host out of its cache for the duration of the PCI transaction. (Microseconds.) Depending on the AGP-set implementation, graphics could reduce access to memory a lot more than the expected fraction. This is to be seen.

Host Performance

## I/O Bus Contention - Solutions

***Compact Primitives:***

- 16 Bit X/Y. Packed RGBA.
- Triangle Strips & Fans and so on: 3:1 reduction!

***Store Textures on graphics board***

***Graphics chips use on-chip input buffers***

- Buffer removes wait penalty

***Host avoids I/O Bus I/O.***

There are two strategies for dealing with I/O bus contention.

- Reducing I/O bus Utilization
- Reducing the penalty due to I/O bus contention

In the first strategy, we try to make our primitives smaller, and remove avoidable traffic.

- Packing X/Y into a single word saves 33% of the bandwidth of an XYZ triple stored in floats.
- Triangle strips use 1/3 of the bandwidth used by triangles.
- Storing textures on the graphics board reduces I/O bus contention, memory contention, and removes a space-consuming pipelining requirement.

In the second strategy, we hide contention with buffers, or by not using unbuffered processors.

- The graphics chip can hide the effects of contention on its input stream by reading large chunks, and by reading ahead before input is needed.
- The host doesn't have large input or output buffers, so it really stalls when an I/O takes a long time. The host should avoid I/O over the I/O bus.

Host Performance

## I/O Bus Contention - Conclusions

***Memory Bus requires over-capacity***

- because Host is essentially unbuffered.

***I/O Bus requires reduced over-capacity***

- because Graphics Chips can be buffered

***System Memory Textures Hurt Host & Graphics Performance***

- because they contribute to memory contention
- because they contribute to I/O bus contention
- because they need huge latency hiding pipelines

•On input, the host will block if it misses the cache even if it doesn't have to wait behind memory contention. There just isn't a way for the host to specify enough things to do to cover the latency of a cache miss.

•On output, the host has a two entry write buffer for uncached write-combining (USWC) writes. On a 400 MHz Host with a 66 MHz bus, it takes 4 host cycles to fill a cache line, and at least 24 host cycles to write it to memory. If the host is writing graphics primitives it will be gated by access to the system bus after the first 64 bytes. These buffers are too small to make it worthwhile to rearrange the Host's geometry code so that write bursts fit in the USWC buffers.

•Graphics chips reading geometric data over the I/O bus can take advantage of the streaming nature of their input and their specialized function, and read ahead so that several primitives are always waiting in their input queues. In this way, graphics chip's queues may have to wait for input, but the graphics chip's processor never has to wait.

•System memory textures are an appropriate way to reach a graphics price point, and the ability to store some textures outside of graphics memory is certainly useful. But, in a performance oriented system, reading textures over the AGP bus frame after frame is a waste of bandwidth.

Host Performance

## Synchronization with Accelerator

***Host synchronizes to update state***

- ***ENSURES*** inefficiency.

***Host synchronizes at screen refresh***

- ***ENSURES*** non-realtime frame-display

***Host should NEVER synchronize with accelerator.***

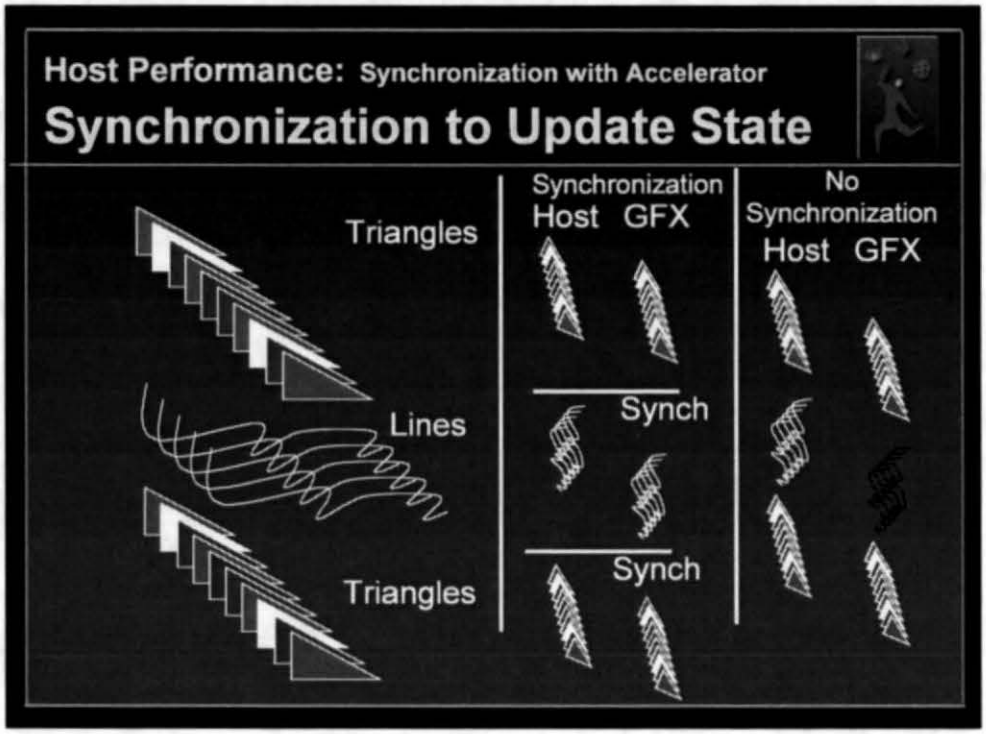
Graphics accelerators without input queues of some kind restrict parallelism to a very fine grain, which is inefficient.

Many graphics accelerators require input queues to be drained before changing state such as: primitive type, dash pattern, frame-buffer operation, texture mode, and so on. This can require the host to synchronize dozens or even thousands of times per frame.

Even synchronizing with the graphics accelerator at end-of-frame can rob the host of a good deal of parallelism. A better time to synchronize with the graphics processor is at start-of-frame, after the host has done all of the computation to start the next redraw.

Even so, synchronizing at all means that there is a window where the host could take an interrupt with nothing in the accelerator's queue, which means that the display can't be real-time. In a simulator, or in video processing applications, or in certain other kinds of digital content creation, real time is either a goal, or required.

In the real-time case, what the host needs is the ability to stay a certain number of frames behind the graphics accelerator.



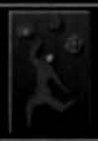

Here we see an example where the host has to draw a set of triangles, some lines and some more triangles. This could be any case where the graphics chip and the host might have to synchronize to update state that can't be updated in the input stream, for instance, switching from textured to untextured, or changing dash styles on lines.

In the middle of this slide, we see the case where the graphics processor has to synchronize. Time flows down. The Host starts processing triangles. Shortly after it begins, the Graphics Accelerator also starts processing triangles. When the Host finds lines on its input, it has to **wait** for the Graphics accelerator to finish with the triangles before it can update registers on the accelerator switching it over to lines. The accelerator also ends up spending some idle time waiting for the host to get ready with the lines. At the end, we don't finish up in time and the last set of triangles falls off the slide.

To the right, we see the same work to be done, but this time the Host can insert state changing commands into the accelerator's input queue. The processors spend NO time waiting for each other, and the work happens much more quickly.

Host Performance: Synchronization with Accelerator

## Real-Time Issues



***Real-time means meeting frame deadlines***

- Soft real-time: Occasional Missed Frame OK
  - *Games*
  - *DCC Previewing*
- Hard real-time: Missed Frames Not OK.
  - *Video Output*
  - *DCC Previewing*

In a real-time system, deadlines have to be met. For graphics, this means displaying frames at a regular, specified rate. There are two categories of real-time: soft and hard.

•In a soft real-time system, the deadlines should usually be met, but if one is missed now and then it isn't a big deal.

> •Games are an example of systems that would usually be classified as soft real-time. It's nice when your game maintains a steady 40 fps, but if it misses a single frame now and then, you probably won't even notice.
>
> •Digital Content Creation (DCC) previewing is another example of a system where it may not be necessary for every deadline to be met. This is certainly the case at the low end, for instance in game development.

•In a hard real-time system, deadlines must be met, or you'll get product return.

> •A classic example of a hard real-time system is fuel mixture adjustments that have to happen in some aircraft every 40ms. or an explosion occurs.
>
> •Video output is an example that is closer at hand. If a video field is not formatted and correctly placed in time, the video hardware will place a bad field on tape, where it will be very hard to get rid of.
>
> •Some Digital Content Creation activities have essentially hard real-time constraints. If an animator is previewing motion for a

Host Performance: Synchronization with Accelerator

## Achieving Real-Time on a PC

***NT isn't real time.***

- Anything can happen at any time.
- Buffering ahead is essential.
- How far? 100ms.

***Buffers have to be there ALWAYS***

- Can't ever synchronize.
- Need a way to control degree of buffering.
  - *Better glFinish();*

•NT isn't a real time system.

•This means that NT make no guarantees about how long it will take for your interrupts to be serviced, or how much of the CPU any given piece of code will get in any given period of time. But, anecdotally, most operating systems have some real-time characteristics at some time-scale. I can usually count on my mailer booting up in a few seconds, and so on.

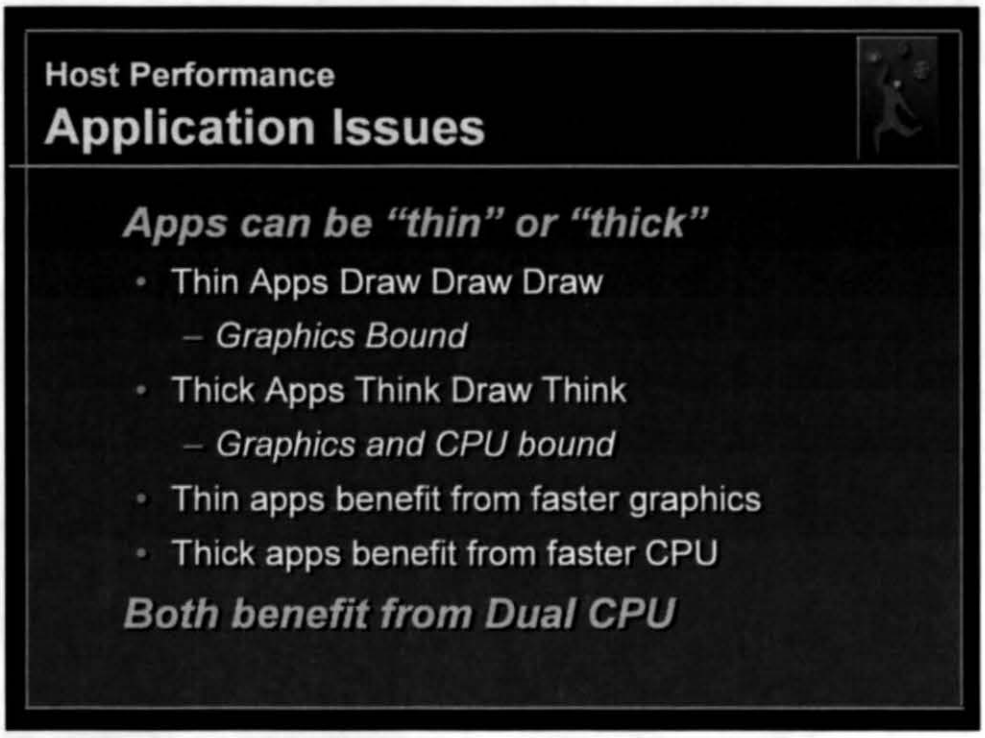
•The trick is to get the CPU to do enough useful work while you have it, to tide you over the periods while you don't have it.

•The question is, how far ahead to get? If we get too far ahead, the system will be difficult to use. We might be feeding input into the system now, while watching output from seconds ago. The Human Factors answer is "100 ms." or 1/10 seconds. For most people, a delay of 100ms in their visual feedback has little or no impact on their performance. If NT can meet real-time deadlines within a 100ms time-frame, we can build an interactive real-time system.

•The Buffers always have to be there

•If we ever let our output buffers get empty, or almost empty, that might be when NT decides to go spend 20ms talking to a disk drive. The graphics input stream can synchronize with the refresh, but the Host can't.

•We need a way for the app to control how far ahead of the graphics accelerator it gets, so that it can trade off



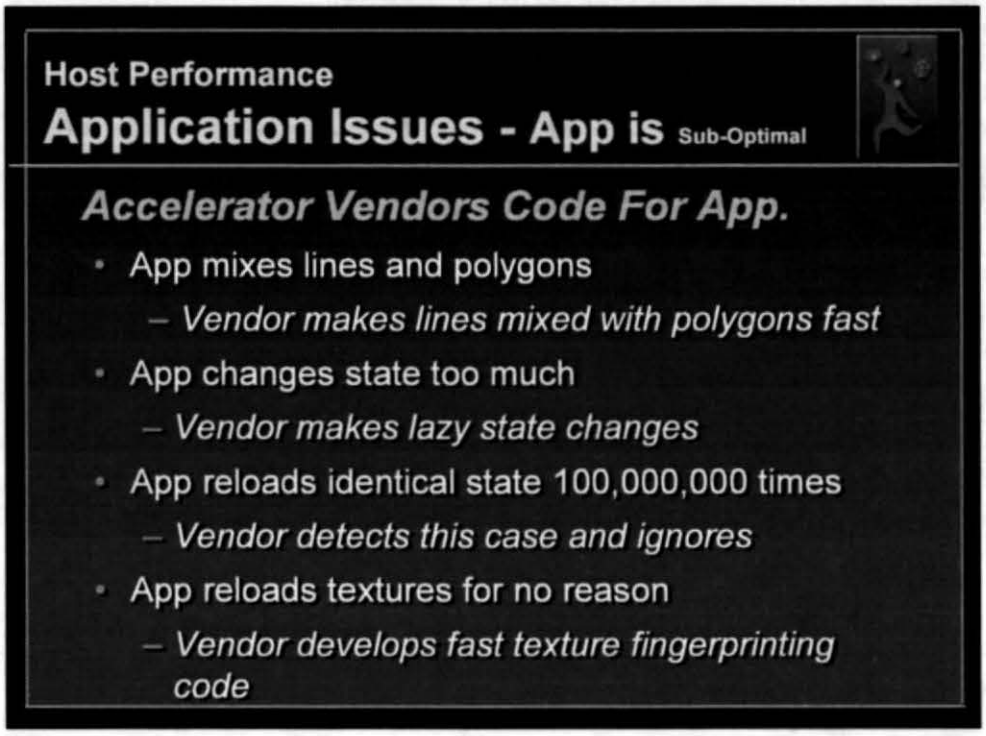

Thin apps (like the CDRS benchmark) do nothing but draw. Inevitably these programs are graphics bound on any platform.

Thick apps, like Pro Engineer, think and draw. These apps require balanced host and graphics performance.

Thin apps benefit from any improvement in graphics performance. If, for instance, a Geometry Accelerator is the most cost effective way to generate FLOPS for geometry calculations, then Geometry Accelerators are more cost effective for thin apps.

Thick apps benefit from faster graphics and from a faster CPU. The benefit to a thick app from a Geometry Accelerator is limited, because the app needs more CPU as well.

The addition of an extra CPU can help both the thick app and its graphics subsystem, if they are coded to take advantage of it. Many applications are now multithreaded, and at least one Graphics Accelerator company has multi-threaded drivers now.

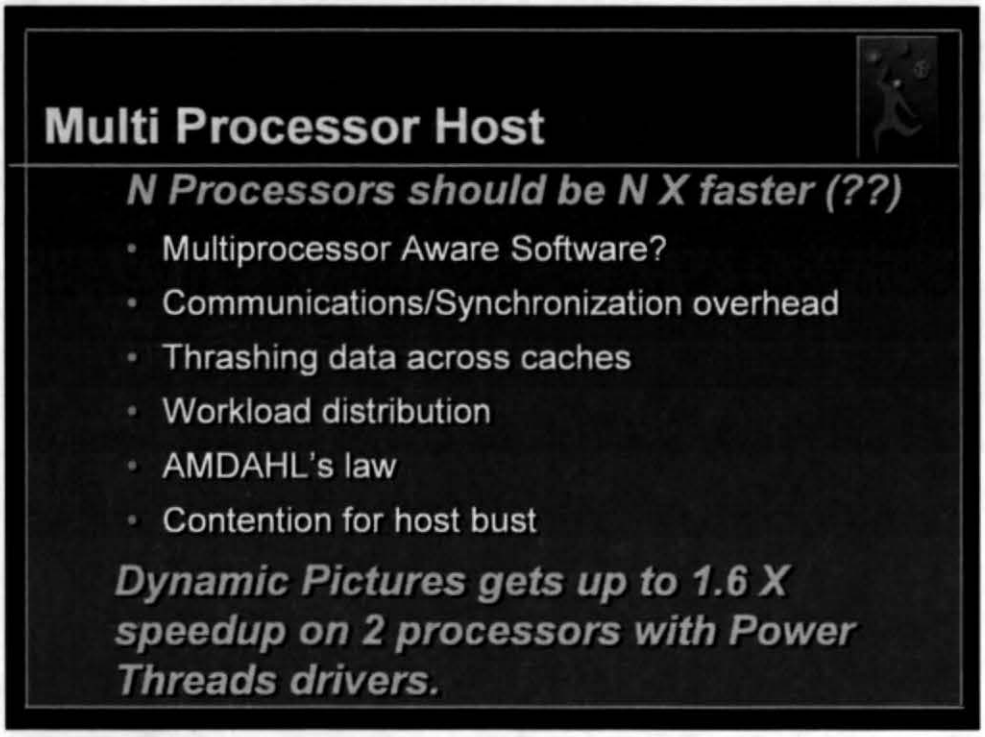



Modern applications are hard to write. Information hiding practices required to complete large software projects mean that the application code often can make no assumptions about the current graphics state. This can lead to massively redundant state changes. We have observed applications reloading the same state thousands of times per frame. Sometimes, this state is quite expensive, for example, textures.

The common technique for dealing with state changes which may not be used when geometry is executed on the host is to avoid propagating their effects through the system until the application tries to draw something. This is lazy execution, and it saves work. In a geometry accelerator, the tradeoff is reversed. The geometry accelerator has nothing to do but graphics, so it usually processes each state change immediately, or eagerly. Lazy algorithms are appropriate when you are using all of your silicon efficiently. Eager algorithms are appropriate when you have cycles that would otherwise go unused.

Another thing that can hurt graphics performance is finely interleaving primitives of mixed types. Even if the accelerator has a fully streaming interface, changing primitive types has a cost: instruction caches aren't used as efficiently, there has to be some overhead in the instruction stream associated with switching primitive types, and each time the primitive is switched, some state validation may occur.

The point here is that graphics vendors need to react to the application vendors and not the other way around. Applications are hard enough to write, and the graphics vendors should try to make things easier to the




People always hope to go N times faster with N processors. This never (rarely) happens. Why?

•Not all software is capable of using all of the available processors. Single threaded software running on N processors runs 1 time faster.

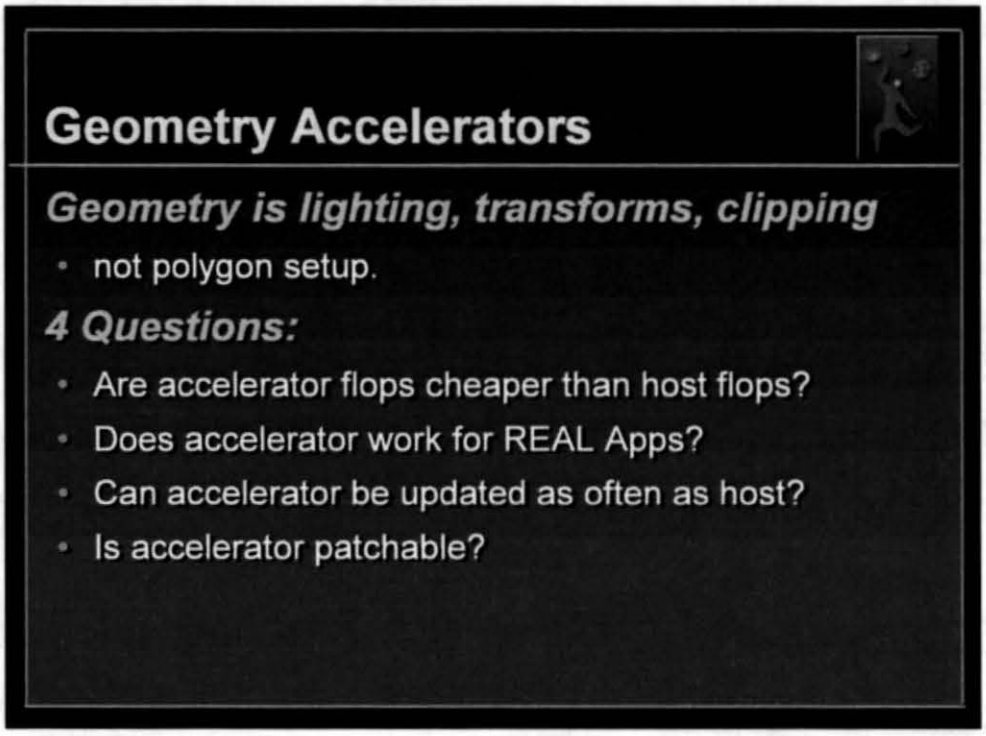
•If you aren't very careful, you spend all of your time communicating and synchronizing. It is worth while to try very hard to remove as much synchronization from your system as possible, because synchronization primitives are signs of explicit serialism. In addition, the operating system provided synchronization primitives are often surprisingly expensive.

•In a system where one processor generates data, and another reads it, the data has to move from one cache to another. This is usually a lot slower and harder on both processors than simple cache misses to system memory. This can sometimes be avoided for some graphics operations.

•If all of the processors are to be busy all of the time, work has to be divided fairly. When the unit of parallelism is only a frame, this can be challenging. To make it more difficult, work has to be distributed coarsely, because the task of distribution is itself an overhead.

•AMDAHLS law says that if fraction X of the problem is intrinsically serial, you can't go more than 1/X times faster using parallelism. This hurts.

•Finally, adding more processors makes the host bus busier, which slows down all of the processors.




Some companies have advertised raster engines than accept polygons as input rather than edges with slopes as geometry accelerators. While this certainly speeds things up, we think of this functionality as an integral part of the minimal raster engine.

Instead of buying extra processors, and multi-threading our drivers, we could buy or build a special purpose geometry accelerator and implement our geometry pipeline on the unit. This is a good idea iff:

• Accelerator FLOPS are cheaper than host FLOPS. This was almost always true in 1985, and it used to be clearly true for Intel processors. The analysis is trickier now. It is very hard to do geometry acceleration for truly fast processors like Alpha, and it will probably be very hard for Katmai New Instruction set and Merced.

•The accelerator works for real applications: often accelerators have limitations spurring from the challenging coding environment they are implemented in that limit their performance on real applications. For instance, they may have a limited number of graphics contexts, and require time consuming context switching when the number of active contexts becomes too large.

•The accelerator can be updated as often as the host: If the host gets 50% faster every 12 months, and the accelerator takes 2 years, then the accelerator is going to be unbalanced at least half of the time.

•The accelerator is FIELD-PATCHABLE. If the accelerator is hard-coded then errors are too. Nobodies software driver is bug-free. Why should we expect the much more difficult hardware implementation to

## Summary

*Graphics acceleration is multiprocessing*

*Coarse grained parallelism is where the success has been in multiprocessing*

*Bandwidth is key. Eliminating contention for system memory is key.*

*Geometry processors are risky and expensive.*


Questions

## Open Q&A for Course Faculty

*Gary Tarolli*
*3Dfx Interactive, Inc.*

*Brian Hook*
*id Software*

*Rich Ehlers*
*Evans & Sutherland Computer Corp.*

*John Danskin*
*Dynamic Pictures, Inc.*

*Bill Lorensen*
*General Electric Corporate Research and Development*