Performance OpenGL

Platform Independent Techniques

or

"A bunch of good habits for every OpenGL programmer"

Dave Shreiner

Alan Commike

Brad Grantham

Bob Kuehne

Introduction

Schedule

1:45 Introduction	Dave Shreiner
2:00 Bottlenecks	Dave Shreiner
2:45 VP/FP Operation	Alan Commike
3:15 Break	
3:30 VP/FP Performance	Bob Kuehne
4:00 Validation	Dave Shreiner
4:30 Geometry Storage	Brad Grantham
5:20 Conclusion / Q&A	All

What You'll See Today…

- An in-depth look at the OpenGL pipeline from a performance perspective
- Techniques for determining where OpenGL performance bottlenecks are limiting your application's performance
- A look at the latest OpenGL mechanisms and how you might use them to increase your application's performance
- A bunch of simple, good habits for OpenGL applications

Performance Tuning Assumptions

- You're trying to tune an interactive OpenGL application
- There's an established metric for estimating the application's performance
  - Consistent frames/second
  - Number of pixels or primitives to be rendered per frame
- You can change the application's source code

Quick Review of Benchmarking

- Use glFinish() when timing rendering
  - remember to take it out of production code!

```
Time  start = getTime();
drawScene();
glFinish();
Time  elasped = getTime() – start;
```

A Brief OpenGL Update

- OpenGL 1.5 was announced at SIGGRAPH!
- Major Features
  - vertex buffer objects
  - occlusion query
  - shadow functions
  - point sprites
- OpenGL Shading language approved as an extension

A Brief OpenGL Update (cont.)

- OpenGL 1.4 added:
  - multi-draw vertex arrays
  - window-coordinate raster position
  - additional blending functionality
  - secondary color
  - level-of-detail control for mipmapped textures
  - mirrored-repeat wrap modes for textures

A Brief OpenGL Update (cont.)

- Vertex and Fragment Program extensions approved as extensions to 1.4
- OpenGL 1.3 added:
  - multiple texturing
  - multi-sampled visuals
  - transposed matrix entry points
  - cube mapping

Some Thoughts on OpenGL's Evolution

- OpenGL 1.3's multi-texture capability marked a paradigm shift
  - de-emphasized multi-pass algorithms
  - began a sequence of rapid evolution
    - register combiners
    - vertex programs
    - fragment programs
    - The OpenGL Shading Language

Some Thoughts on OpenGL's Evolution (cont.)

- Advanced data placement extensions
  - compiled vertex arrays
  - vertex array objects (extensions)
  - vertex buffer objects (extension ➡ core)

Conclusion:

  - OpenGL's evolution make understanding the pipeline and state management even more important

Performance Bottleneck Determination

Eliminating OpenGL Errors

- Asynchronous Error Reporting
  - OpenGL doesn't tell you when something goes wrong
    - Calls will silently mark an error and glColor3fv
    - Need to use **`glGetError()`** to determine if something went wrong

"We should forget about small efficiencies, about 97% of the time. Premature optimization is the root of all evil" – Donald Knuth

Checking for Errors

- Check Early and often in Application Development
  - Only first error* is retained
    - Additional errors are discarded until error flag is cleared by calling **`glGetError()`**
  - Erroneous OpenGL function skipped

Checking a single command

- Simple Macro

```
#define CHECK_OPENGL_ERROR( cmd ) \
     cmd; \
     { GLenum  error; \
       while ( (error = glGetError()) != GL_NO_ERROR) { \
         printf( "[%s:%d] '%s' failed with error %s\n", \
                 __FILE__, __LINE__, #cmd, \
                 gluErrorString(error) ); \
     }
```

  - Some limitations on where the macro can be used
    - can't use inside of **glBegin()** / **glEnd()** pair

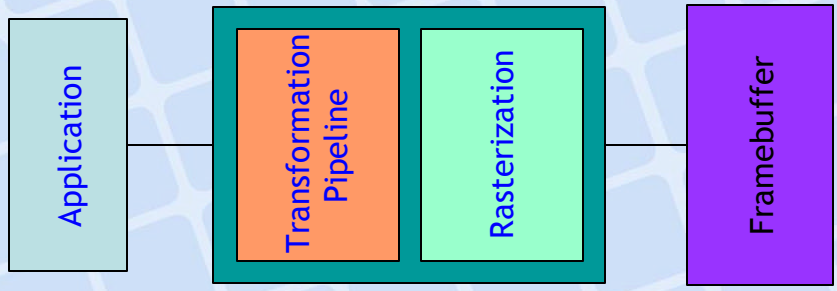
Checking More Thoroughly

- Modified **gl.h** checks almost every situation

```
#define glBegin( mode ) \
  if ( __glDebug_InBegin ) { \
    printf( "[%s:%d] glBegin( %s ) called between" \
            "glBegin()/glEnd() pair\n", \
            __FILE__, __LINE__, #mode ); \
  } else { \
    __glDebug_InBegin = GL_TRUE; \
    glBegin( mode ); \
  }
```

  - Script for re-writing **gl.h** available from web site

The OpenGL Pipeline
(The Macroscopic View)



Performance Bottlenecks

- *Bottlenecks* are the performance limiting part of the application
  - *Application* bottleneck
    - Application may not pass data fast enough to the OpenGL pipeline
  - *Transform-limited* bottleneck
    - OpenGL may not be able to process vertex transformations fast enough

Performance Bottlenecks (cont.)

- *Fill-limited* bottleneck
  - OpenGL may not be able to rasterize primitives fast enough

There Will Always Be A Bottleneck

- Some portion of the application will always be the limiting factor to performance
  - If the application performs to expectations, then the bottleneck isn't a problem
  - Otherwise, need to be able to identify which part of the application is the bottleneck
  - We'll work backwards through the OpenGL pipeline in resolving bottlenecks
  - Sometimes you can take advantage of bottlenecks in a positive way
    - enhance quality of another pipeline stage that isn't the bottleneck

Fill-limited Bottlenecks

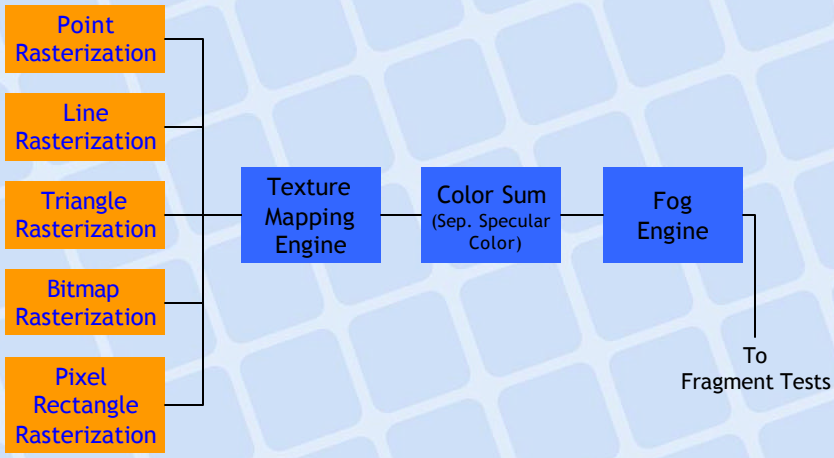
- System cannot fill all the pixels required in the allotted time
  - Easiest bottleneck to test
  - Reduce number of pixels application must fill
    - Make the viewport smaller

Reducing Fill-limited Bottlenecks

- The Easy Fixes
  - Make the viewport smaller
    - This may not be an acceptable solution, but it's easy
  - Reduce the frame-rate

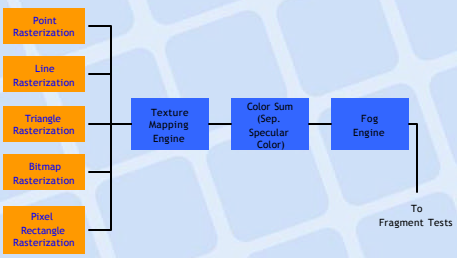
$$\frac{800\ \mathrm{M}\ \frac{\text{pixels}}{\text{second}}}{75\ \frac{\text{frames}}{\text{second}}} \approx 10.7\ \mathrm{M}\ \frac{\text{pixels}}{\text{frame}} \qquad \frac{800\ \mathrm{M}\ \frac{\text{pixels}}{\text{second}}}{60\ \frac{\text{frames}}{\text{second}}} \approx 13.3\ \mathrm{M}\ \frac{\text{pixels}}{\text{frame}}$$

A Closer Look at OpenGL's Rasterization Pipeline



Reducing Fill-limited Bottlenecks (cont.)

- Rasterization Pipeline
  - Cull back facing polygons
    - Does require all primitives have same facediness
  - Use a simpler texture filter
    - Particularly on objects that occupy small screen area
      - far from the viewer
  - Use per-vertex fog, as compared to per-pixel



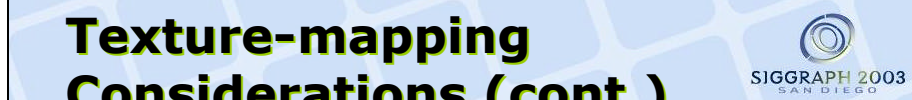
Texture-mapping Considerations

- *Use Texture Objects*
  - Allows OpenGL to do texture memory management
    - Loads texture into texture memory when appropriate
    - Only convert data once
  - Provides queries for checking if a texture is resident
    - Load all textures, and verify they all fit simultaneously

Texture-mapping Considerations (cont.)

- Texture Objects (cont.)
  - Assign priorities to textures
    - Provides hints to texture-memory manager on which textures are most important
  - Can be shared between OpenGL contexts
    - Allows one thread to load textures; other thread to render using them
  - Requires OpenGL 1.1

Texture-mapping Considerations (cont.)



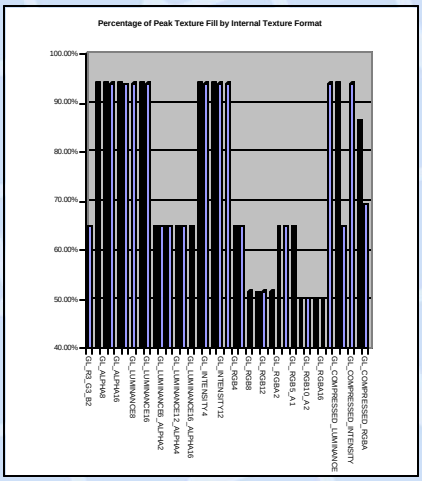
- Sub-loading Textures
  - Only update a portion of a texture
    - Reduces bandwidth for downloading textures
    - Usually requires modifying texture-coordinate matrix

Texture-mapping Considerations (cont.)

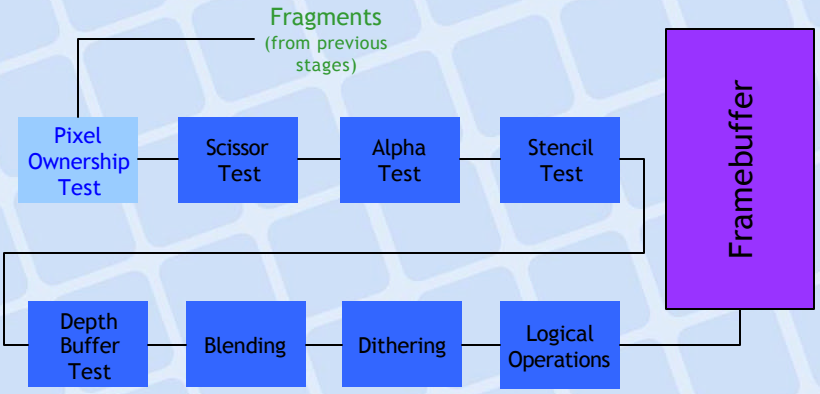
- Know what sizes your textures need to be
  - What sizes of mipmaps will you need?
  - OpenGL 1.2 introduces texture *level-of-detail*
    - Ability to have fine grained control over mipmap stack
      - Only load a subset of mipmaps
      - Control which mipmaps are used

Internal Texture Formats

- OpenGL provides numerous formats for storing textures
  - tradeoff visual quality vs. speed vs. residency
- Compressed textures look like a good option
  - make sure they meet you quality needs

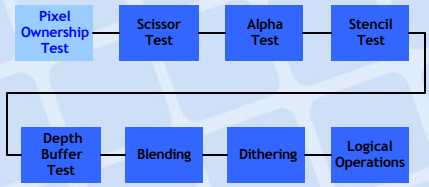


A Closer Look at OpenGL's Rasterization Pipeline (cont.)



Reducing Fill-limited Bottlenecks (cont.)

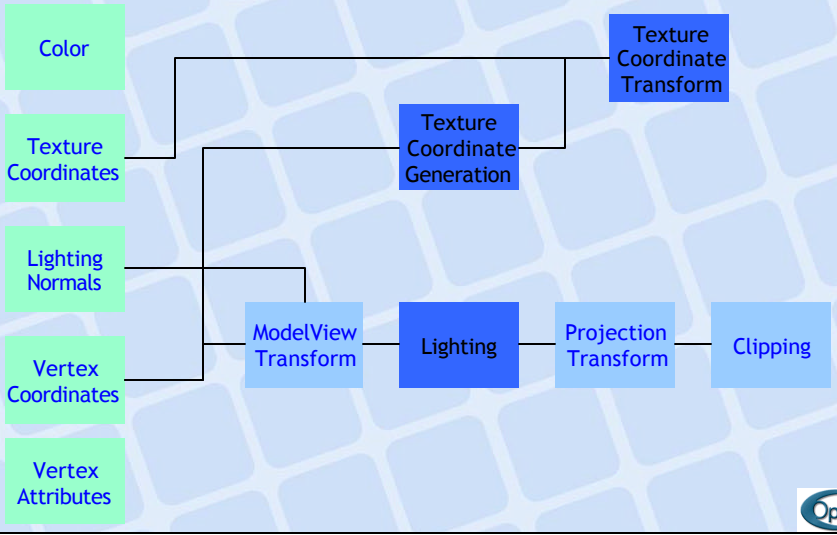
- Fragment Pipeline
  - Do less work per pixel
    - Disable dithering
    - Depth-sort primitives to reduce depth testing
    - Use alpha test to reject transparent fragments
      - saves doing a pixel read-back from the framebuffer in the blending phase



Transform-limited Bottlenecks

- System cannot process all the vertices required in the allotted time
  - If application doesn't speed up in fill-limited test, it's most likely transform-limited
  - Additional tests include
    - Disable lighting
    - Disable texture coordinate generation

A Closer Look at OpenGL's Transform Pipeline



Reducing Transform-limited Bottlenecks

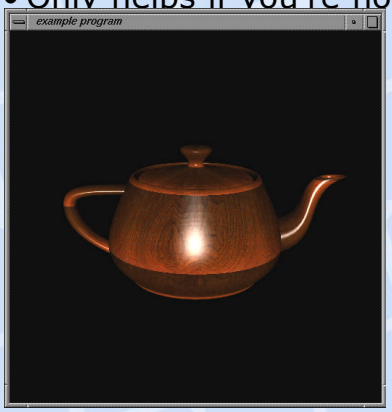
- Do less work per-vertex
  - Tune lighting
  - Use "typed" OpenGL matrices
  - Use explicit texture coordinates
  - Simulate features in texturing
    - lighting
  - Use triangle strips
  - Use indexed primitives on newest hardware

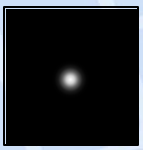
Lighting Considerations

- Use infinite (directional) lights
  - Less computation compared to local (point) lights
  - Don't use `GL_LIGHTMODEL_LOCAL_VIEWER`
- Use fewer lights
  - Not all lights may be hardware accelerated

Lighting Considerations (cont.)

- Use a texture-based lighting scheme
  - Only helps if you're not fill-limited





Reducing Transform-limited Bottlenecks (cont.)

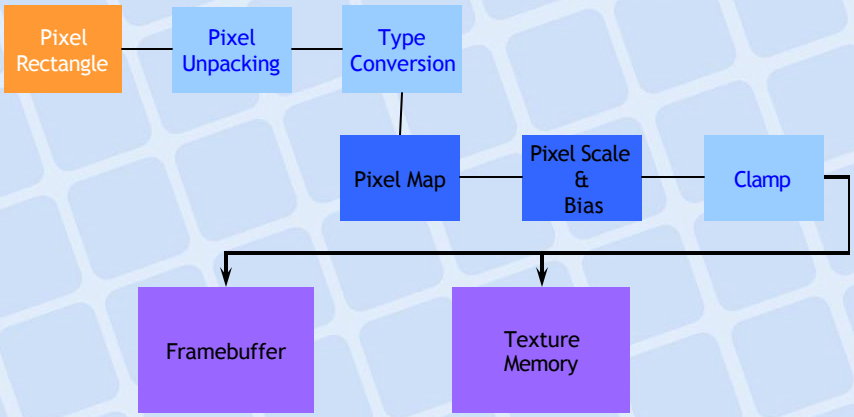
- Matrix Adjustments
  - Use "typed" OpenGL matrix calls

"Typed"	"Untyped"

`glRotate*()`
`glScale*()`
`glTranslate*()`
`glLoadIdentity()`
	`glLoadMatrix*()`
	`glMultMatrix*()`
	`glTransposeLoadMarix*()`
	`glTransposeMultMatrix*()`

  - Some implementations track matrix type to reduce matrix-vector multiplication operations

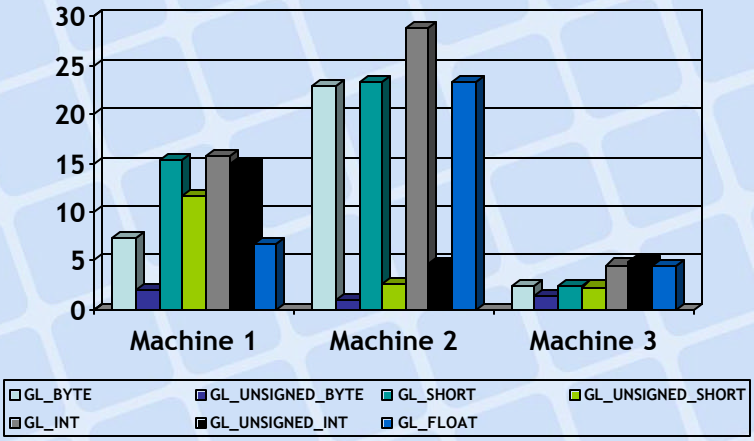
A Closer Look at OpenGL's Pixel Pipeline



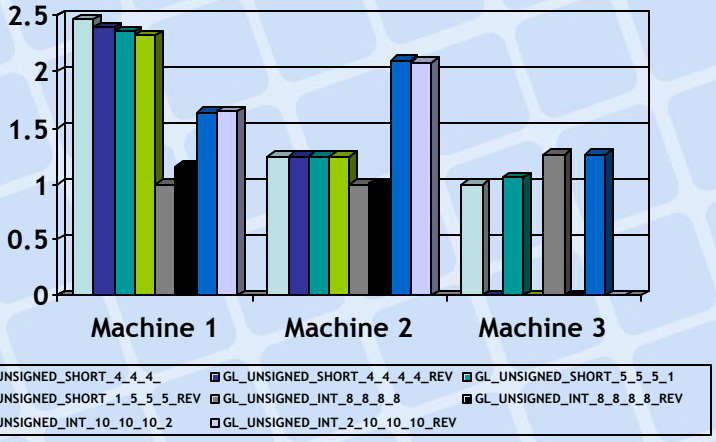
Working with Pixel Rectangles

- Texture downloads and Blts
  - OpenGL supports many formats for storing pixel data
    - Signed and unsigned types, floating point
  - Type conversions from storage type to framebuffer / texture memory format occur automatically

Pixel Data Conversions



Pixel Data Conversions (cont.)



Pixel Data Conversions (cont.)

- Observations
  - Signed data types probably aren't optimized
    - OpenGL clamps colors to [0, 1]
  - Match pixel format to window's pixel format for blts
    - Usually involves using *packed pixel formats*
    - No significant difference for rendering speed for texture's internal format

What If Those Options Aren't Viable?

- Use more or faster hardware
- Utilize the "extra time" in other parts of the application
  - Transform pipeline
    - tessellate objects for smoother appearance
    - use better lighting
  - Application
    - more accurate simulation
    - better physics

OpenGL Vertex and Fragment Program Operation

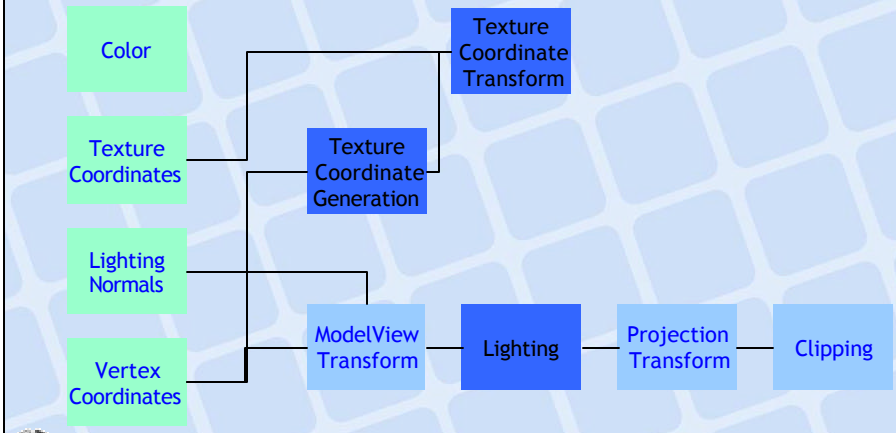
Introduction

- Fixed-function OpenGL hardware still exists but most new & future hardware exposes programmability.
- The old model was a box with lots of knobs.
- The new model is a box with many fewer knobs and two programs.

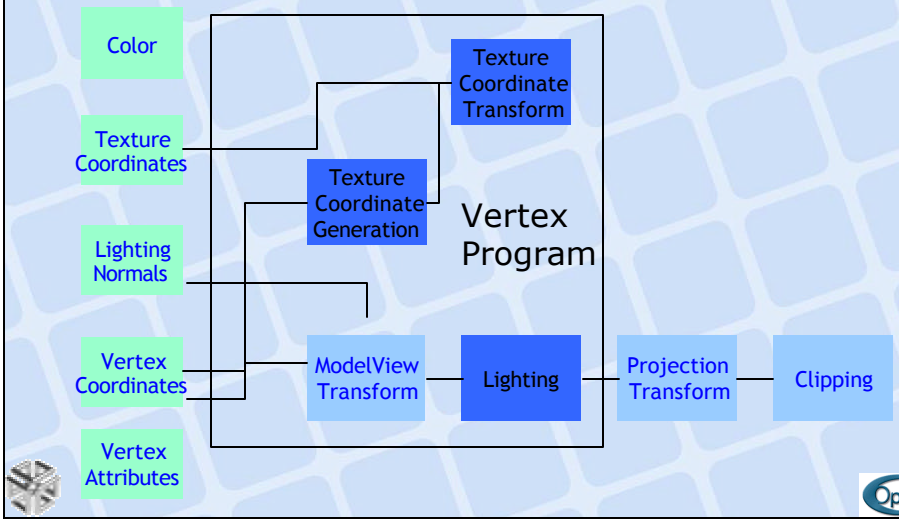
Programmable Graphics

- Two types of operations, per-vertex, per-fragment
- ARB extensions to core OpenGL
- Low-level assembly-style code
- Common set of base instructions, some specific fragment instructions

Review: Fixed-Function Transform Pipeline



Programmable Transformation Pipeline



Programmable Vertex Processing

- Hardware has moved from *fixed function* pipeline *programmable pipeline*
- ARB_vertex_program provides a mechanism to replace OpenGL Transform, Lighting, Texture Coordinate Generation with user defined mechanism

What Can You Do?

- Complete control of xform and lighting
- Complete control of tex coord generation
- Operations all hardware accelerated
- Custom Lighting Equations
- Custom Transformations
- Offload compute to the Vertex Processor
- Lots of new cool effects...

You Win Some, You Lose Some

Win:

A programmable Vertex Pipeline gives you control! No need to wait for new OpenGL extensions to try out that *Cool New Algorithm™!*

Lose:

You need to implement *all* fixed function xform, light, tex coord generation

Functionality You Need To Implement

`glEnable(GL_VERTEX_PROGRAM_ARB)` turns off:

- Modelview and projection vertex transformations
- Vertex weighting/blending
- Normal transformation, rescaling, normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation and texture matrix transformations
- Per-vertex point size and fog coordinate computations
- User-clip planes

Functionality That's Not Replaced

- Evaluators
- Clipping to the view frustum
- Perspective divide
- Viewport transformation
- Depth range transformation
- Front and back color selection (for two-sided)
- Clamping of primary and secondary colors to [0,1]
- Primitive assembly, setup, rasterization

Short intro to ARB_vertex_program

- A Vertex Program is an assembly-language like set of 27 instructions
- Stored in an external file or internally as character array
- Parsed and compiled for the target hardware
- Managed like texture objects - Vertex Programs are bound and hardware ensures residency

Structure of a Vertex Program


```
!!ARBvp1.0
Define Attributes  # Vertex State
Define Parameters  # OGL State,
                   # Constants
Define Temporaries
Define Address Registers
Instructions
Set Results
END
```

```
!!ARBvp1.0
ATTRIB pos = vertex.position;
PARAM l0Dir = {state.light[0].position};
TEMP  eyeNormal;

.
.
.
DP3 eyeNormal.x, mvinv[0], normal;
.
.
.
MOV result.color, color;
END
```

Vertex Program Parameters

- OpenGL State (lighting, materials, matrices, texture coords)
- Need to enable state in OpenGL to ensure vertex data is set
- Used to declare program constants

```
PARAM ambient_l0 = state.light[0].ambient;
PARAM highLightColor = { 0.2, 0.5, 0.2, 1.0 };
```

Vertex Program Attributes

- State associated with a Vertex
- Read-only

```
vertex.position
vertex.weight
vertex.weight[n]
vertex.color
vertex.color.primary
vertex.color.secondary
vertex.fogcoord
vertex.texcoord
vertex.texcoord[n]
vertex.matrixindex
vertex.matrixindex[n]
vertex.attrib[n]
```

Vertex Program Attributes

- Attributes are per vertex data: position, normal, color, etc.
- Generic attributes can be used for application defined use: pressure, velocity, softness, etc.
- Either traditional or generic, not both

```
ATTRIB pressure = vertex.attrib[12];
```

Vertex Program Temporaries

- Temporaries must be declared
- Some programs may compile without declaring temporaries, but the results will not be portable!

```
TEMP tmp;
```

Vertex Program Instructions

- Wide assortment of instructions from simple add/multiple, to specialized lighting instructions
- No branching or looping
- Loops can be unrolled, but watch out for exceeding the max number of instructions

Vertex Program Instructions


```
ABS - absolute value
ADD - add
ARL - address register load
DP3 - 3-component dot product
DP4 - 4-component dot product
DPH - homogeneous dot product
DST - distance vector
EX2 - exponential base 2
EXP - exponential base 2 (est)
FLR - floor
FRC - fraction
LG2 - logarithm base 2
LIT - compute light
   coefficients
LOG - logorithm base 2
MAD - multiply and add
```

```
MAX - maximum
MIN - minimum
MOV - move
MUL - multiply
POW - exponentiate
MUL - multiply
POW - exponentiate
RCP - reciprocal
RSQ - reciprocal square root
SGE - set on greater than or
   equal
SIN - sine with reduction to
SLT - set on less than
SUB - subtract
SWZ - extended swizzle
XPD - cross product
```

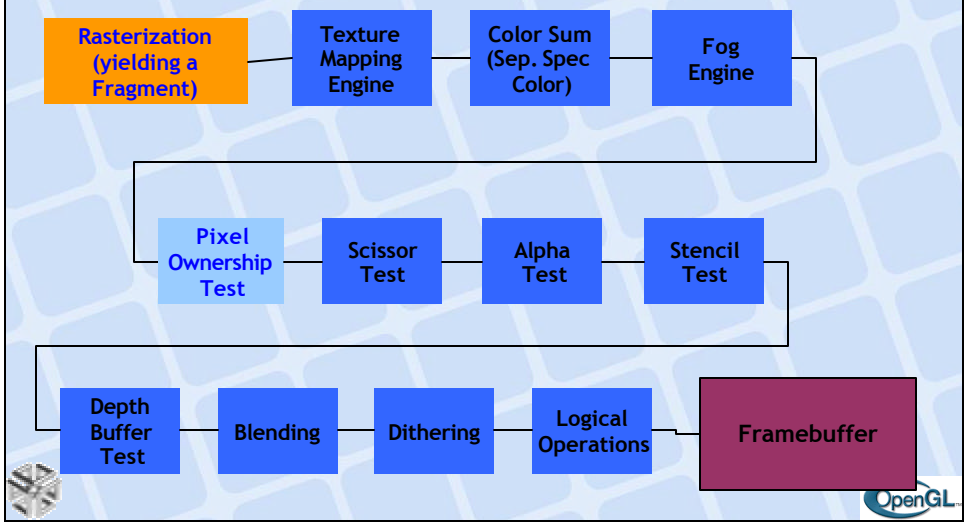
Vertex Program Results

- Results must be explicitly written
- Values written are interpolated across a primitive and accessible as a fragment later
- Results set in:

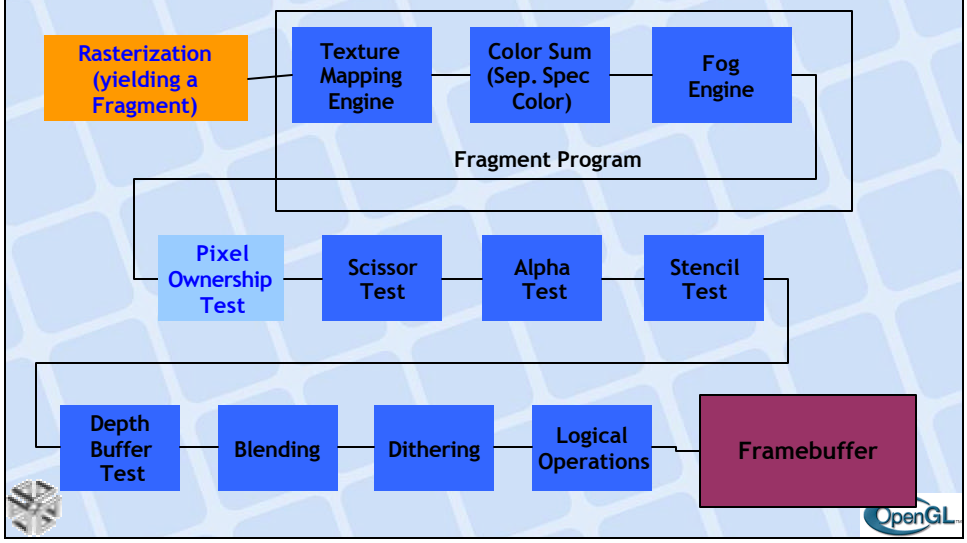
```
result.position
result.color
result.color.primary
result.color.secondary
result.color.front.primary
result.color.front.secondary
result.color.back
```

```
result.color.back.primary
result.color.back.secondary
result.fogcoord
result.pointsize
result.texcoord
result.texcoord[n]
```

Review: Fixed-Function Rasterization Pipeline



Programmable Rasterization Pipeline



Fragment Programs

- Similar to Vertex Programs, assembly-like instructions that let you control fragment generation
- Replaces fixed function Texture Blend, Color Sum, and Fog on a per fragment basis
- Per pixel tests are still part of the Fixed Function pipeline
- Same bind/compilation semantics as vertex program

Structure of a Fragment Program

```
!!ARBfp1.0
Define Attributes  # Fragment State
Define Parameters  # OGL State,
                   # Constants
Define Temporaries
Instructions
Set Results
END
```

```
!!ARBfp1.0
ATTRIB vnormal = fragment.texcoord[0];
PARAM l0Dir = {state.light[0].position};
TEMP  eyeNormal, fragcolor;

.
.
.
DP3 eyeNormal.x, mvinv[0], vnormal;
.
.
.
MOV result.color, fragcolor;
END
```

Fragment Program Attributes

- State associated with a fragment
- Read-only

```
fragment.color
fragment.color.primary
fragment.color.secondary
fragment.texcoord
fragment.texcoord[n]
fragment.fogcoord
fragment.position
```

Fragment Program Attributes

- Attributes are aliases for fragment program state.
- You need to know what each attribute represents, there is no compiler to yell at you

```
ATTRIB grouchyness = fragment.texcoord[3];
```

Fragment Program Parameters

- OpenGL State (lighting, materials, matrices, texture)
- Need to enable state in OpenGL to ensure fragment state is updated
- Used to declare program constants

```
PARAM ambient_l0 = state.light[0].ambient;
PARAM highLightColor = { 0.2, 0.5, 0.2, 1.0 };
```

Fragment Program Texturing

- Texture coordinates are part of the fragment state
- Coordinates are interpolated from per vertex values to per fragment values
- Coordinates are only updated if OpenGL texturing is enabled

Fragment Program Texturing

- Texture instructions fetch texels based on filtering mode set in OGL texture parameters.
- Fragment program has control over texture environment (aka blending): modulate, decal, add, etc.

```
# Decal texturing environment
TEX result.color, tex_coord, texture, 2D;
```

Fragment Program Operations

- 29 ALU type operations (**`mul, dp3, lrp, cos, min`**, etc.)
- 3 Texturing operations (**`tex, txp, txb`**)
- 1 operation not like the others (**`kil`**)
- two outputs: fragment color and fragment depth (not req)
- no branching or looping (can be simulated)

Fragment Program Instructions

- Branches can be simulated with conditional instructions
- SGE and SLT compare two vectors
- CMP will do conditional moves

```
if ( diffuse_color > 0.5 )
  tmp = dark_color;
else
  tmp = light_color;
```

```
SGE tmp, diffuse_color, point_five;
CMP tmp, dark_color, light_color;
```

Fragment Program Instructions

ABS - absolute value
ADD - add
CMP - compare
COS - cosine with reduction
DP3 - 3-component dot product
DP4 - 4-component dot product
DPH - homogeneous dot product
DST - distance vector
EX2 - exponential base 2
FLR - floor
FRC - fraction
KIL - kill fragment
LG2 - logarithm base 2
LIT - compute light coefficients
LRP - linear interpolation
MAD - multiply and add
MAX - maximum
MIN - minimum
MOV - move
MUL - multiply
POW - exponentiate
MUL - multiply
POW - exponentiate
RCP - reciprocal
RSQ - reciprocal square root
SCS - sine/cosine without reduction
SGE - set on greater than or equal
SIN - sine with reduction
SLT - set on less than
SUB - subtract
SWZ - extended swizzle
TEX - texture sample
TXB - texture sample with bias
TXP - texture sample with projection
XPD - cross product

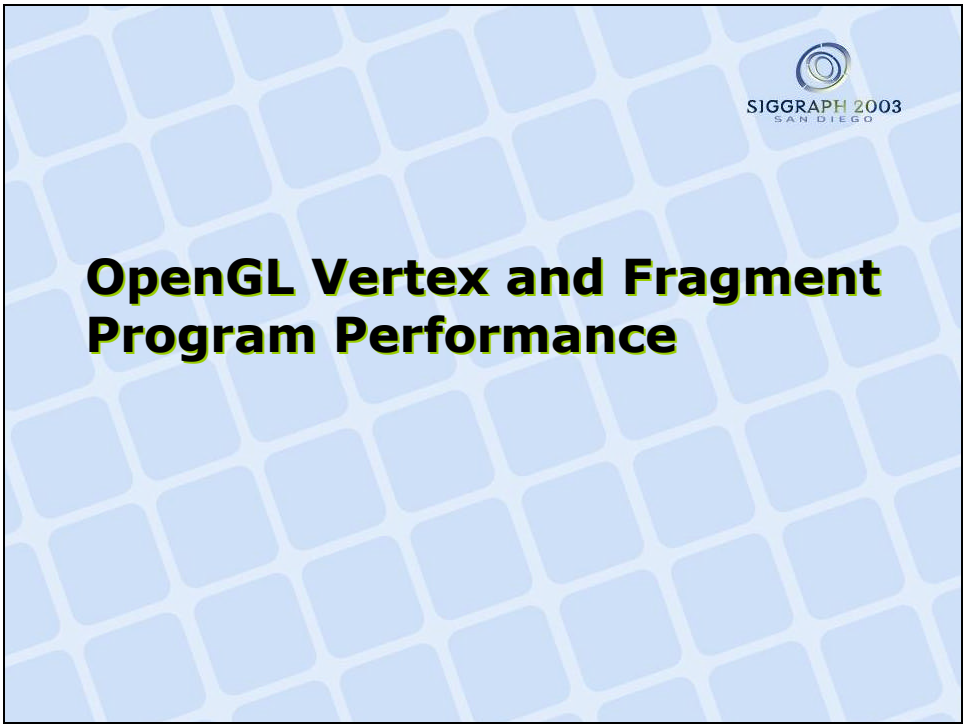
The KIL instruction

- The **KIL** instruction will kill a fragment causing it to not continue to the rest of the pipe.
- Killing a fragment does not stop fragment program execution.
- Useful for scissoring and other effects, but not free!

Fragment Program Results

- Results must be explicitly stored for the fragment to flow down the rest of the pipe
- Results set in:

```
result.color;
result.depth;
```



Program Performance

SIGGRAPH 2003
SAN DIEGO

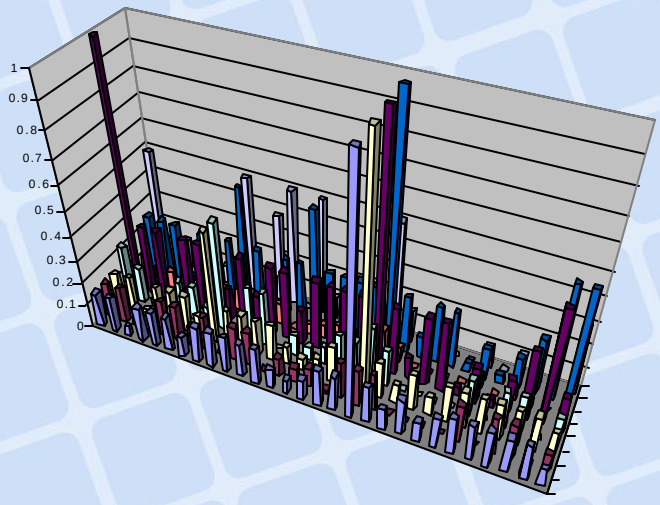
- Lots of variables (literally)
- Length, number of active programs, instruction types, instruction ordering, temporaries, etc.
- Note of interest: fixed-function OpenGL vs programmable evolution.

OpenGL

Instruction performance

- Each instruction has potentially different performance.
  - Some execute in a single clock, others multiple.
  - Some do 'math', others do interpolation (tex lookup) which will have different perf based on texture state too.
  - Good vendor docs can help.

All Instructions aren't Created Equal



Instruction Performance

- Some instructions are macros and take up more resources than you would think.

Example:

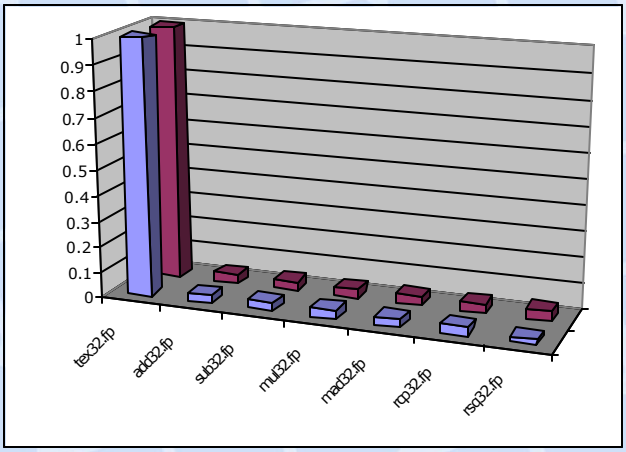
One vendor can fit 32 add instructions in their instruction buffer, but only 7 **`sin/cos`** instructions

Instruction Tips

- Code, then optimize
- Aggregate instructions together (**`DP3, XPD`**)
- Break out common pieces, pass in per-object as ATTRIBS (**`glVertexAttrib*ARB()`**)
- Consider texture coords as general interpolate engines

Instruction Tips

- Texture is significantly faster



Instruction Tips: Texture

- Texture an order of magnitude faster on architectures we tested.
  - ~4x number ALU instructions vs **TEX** instructions in a program. Cool! A win, for now!
- Tip: Embed 'complex' math in textures.
  - sin, cos, acos, tan, tables, airspeeds of the English swallow, whatever.
- Caution: Interpolation & Continuity…

Textures as General Arrays

- Textures can be used as general 1D, 2D, and 3D arrays
- In some cases a table lookup via a texture can replace a calculation
- Can do dependent lookups

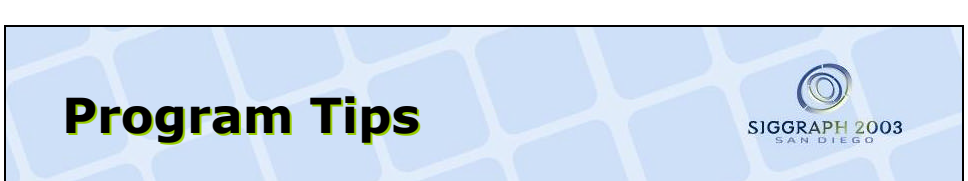
```
a = x[ y[i] ];
```

Texture Coordinates As General Interpolants

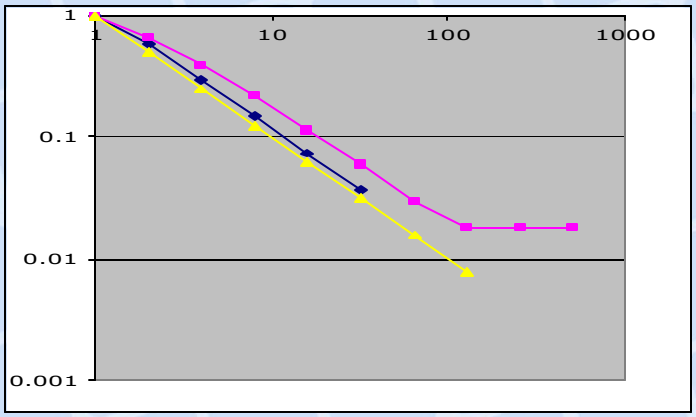
- General program data can be stored in Texture Coordinates per vertex and interpolated into per fragment data
- Useful for advanced rendering algorithms to attach attributes to fragments

```
ATTRIB normal = fragment.texcoord[3];
ATTRIB pressure = fragment.texcoord[2];
```

Program Tips



- Longer programs run slower.



Program Tips

- Not all instructions are created equal.
  - Some require more internal hardware resources
  - Some are dramatically faster
  - Swizzling/Masking may affect performance
  - Texture performance depends on native texture format support.
    - `GL_LINEAR_MIPMAP_LINEAR` is going to require more interpolates than `GL_NEAREST_MIPMAP_NEAREST`
  - No program break.
    - `KIL` doesn't improve performance
  - No branching.
    - Conditional emulation computes both branches.

Program Performance

- Hard to evaluate performance based on program contents
  - Looks like assembly, but it's compiled.
  - Native instruction set different
  - Dead code, instruction reorder, etc.

Other Painful Bits

- You get to do a lot of work, if you want to use this stuff.
- All-or-nothing (lighting, attrib alias, etc)
- Programs can pass stuff down-stream only - for now.
  - Recirculation is coming, via 'über-buffers'.
  - Short version: Render pixel to buffer, then re-use as vertex array in another pass.

Future Programmability

- High-level languages are here
  - **`ARB_(fragment|vertex)_shader`** approved!
  - C-like language vs today's assembly-like
  - Not widely implemented yet. Wait for it …
- Problem domain is same, level of code is different.
  - Should be easier to write code
  - Puts your code farther from the metal

Trends and Observations

- Everything will change.
- Beginning of lifecycle for hardware programmability.
- Firm 'do this' results hard to provide because few platforms support full programmability now.

Trends and Observations

- Knowing your data is more important than ever.
  - Lots of geometry? Lots of fill? Lots of App?
- Can then write Vertex & Frag programs to do the right thing.

Application Performance

Review: Bottlenecks

Balance Workload

Bottleneck	Data
App	Object
Transform	Vertex
Fill	Fragment

Programmable Pipeline Bottlenecks

Bottleneck	Data	Hardware
App	Object	CPU
Transform	Vertex	GPU - VP
Fill	Fragment	GPU - FP

Test Fill/Transform Limited

- Never easier
  - Vertex Limited?
    - Old technique: stub glVertex calls
    - New technique: Install stub vertex program
  - Fill Limited?
    - Old technique: Shrink window
    - New technique: Install stub fragment program
  - Else: App/Download Limited

Programmable Pipeline Performance

Bottleneck	Data	Hardware	Ops
App	Object (1)	CPU	variable
Transform	Vertex (100s)	GPU - VP	1x
Fill	Fragment (Nx)	GPU - FP	10x

Balance Workload - App Limited

Data	Execution Frequency
Object	1x
Vertex	100x
Fragment	10000x

- Move color computation to frag.
- Any required generated per-vertex data.
- Bandwidth limited – could store vertices on card and generate other thingies.

- Most modern CPUs have a SIMD computation engine too (AltiVec, MMX)

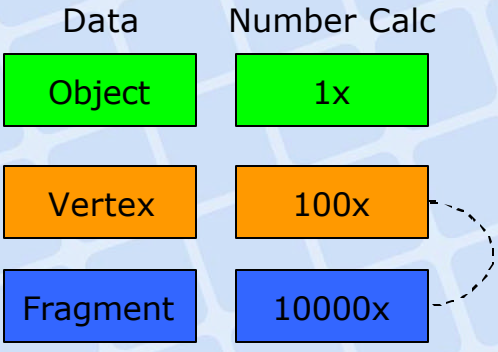
Balance Workload - Fill Limited

Data	Number Calc
Object	1x
Vertex	100x
Fragment	10000x

- Pre-light static objects using CPU, and apply as color. Maybe do light on host.
- Move Per-Frag Lighting to Per-Vertex

- Replace math with texture lookups.
- Interpolation from vertex program engine

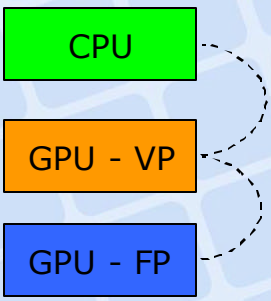
Balance Workload - Transform Limited



- Move transformations to host. Pre-transform static objects, etc.
- Move lighting to fragment program.

What to Keep in Mind

- You have full programmatic control over 3 individual processors.
- You can keep them fully utilized by moving work among them.

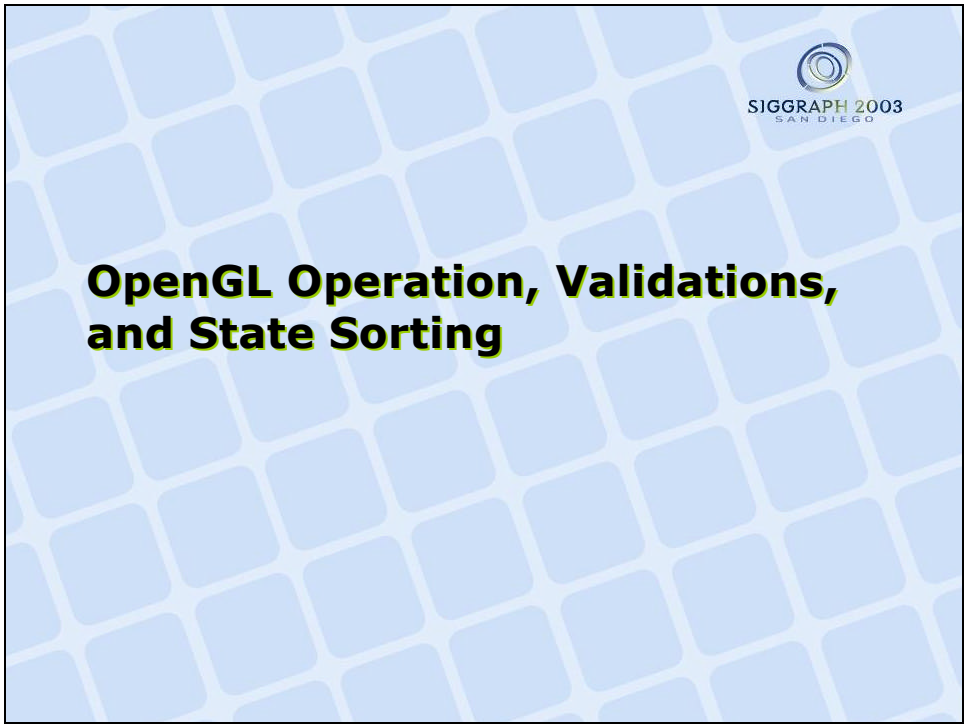


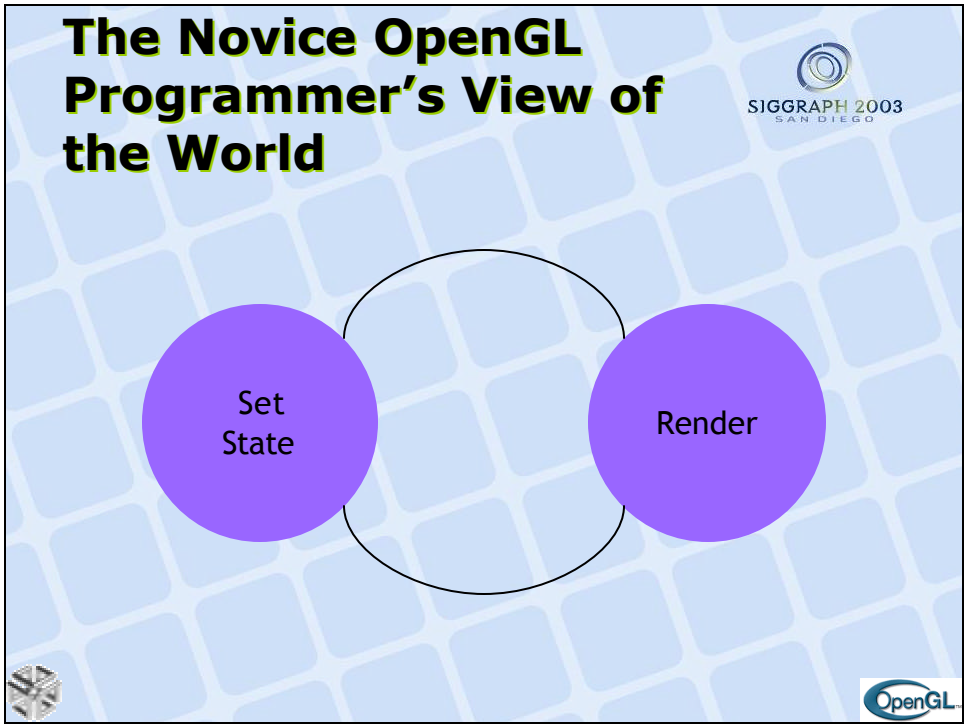
State Sorting

- State sorting in the way previously described is still valid, but different
- Use of any individual vertex or fragment program implies explicit application binding of:
  - Textures
  - Complementary program (vert/frag)
  - Vertex Attributes
  - VBOs, Matrices, Colors, etc.

Example: Lighting

- Lighting is easy in fixed-function.
- More complex in programmable.
  - Must implement each light, no simple 'enable'
  - If program doesn't do calculation, lighting isn't applied.
  - Can get complex: multiple lights, local, non-local, tex coords, etc.
- If program only needs a subset of ogl lighting, a vertex program can execute faster than fixed-function pipeline. Generally applicable.


SIGGRAPH 2003
SAN DIEGO
OpenGL Operation, Validations, and State Sorting


The Novice OpenGL Programmer's View of the World
SIGGRAPH 2003
SAN DIEGO
Set State
Render
OpenGL

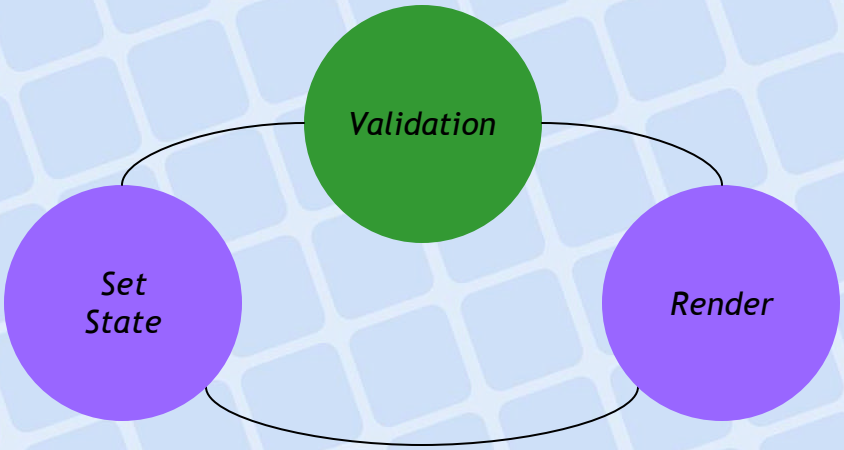
What Happens When You Set OpenGL State

- The amount of work varies by operation

Turning on or off a feature (`glEnable()`)	Set the feature's enable flag
Set a "typed" set of data (`glMaterialfv()`)	Set values in OpenGL's context
Transfer "untyped" data (`glTexImage2D()`)	Transfer and convert data from host format into internal representation

- But all request a validation at next rendering operation

A (Somewhat) More Accurate Representation



Validation

- OpenGL's synchronization process
  - *Validation* occurs in the transition from state setting to rendering

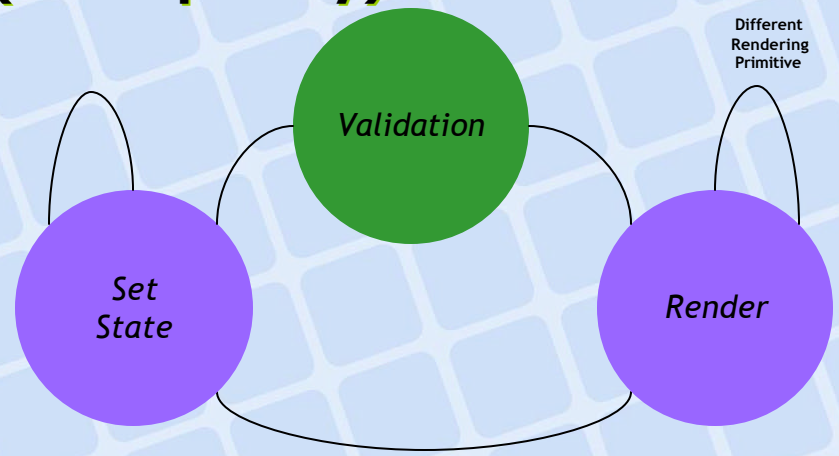
```
glMaterial( GL_FRONT, GL_DIFFUSE, blue );
glEnable( GL_LIGHT0 );
glBegin( GL_TRIANGLES );
```

  - Not all state changes trigger a validation
    - Vertex data (e.g. color, normal, texture coordinates)
    - Changing rendering primitive

What Happens in a Validation

- Changing state may do more than just set values in the OpenGL context
  - May require reconfiguring the OpenGL pipeline
    - selecting a different rasterization routine
    - enabling the lighting machine
  - Internal caches may be recomputed
    - vertex / viewpoint independent data

The Way it Really Is (Conceptually)



Why Be Concerned About Validations?

- Validations can rob performance from an application
  - "Redundant" state and primitive changes
  - Validation is a two-step process
    - Determine what data needs to be updated
    - Select appropriate rendering routines based on enabled features

How Can Validations Be Minimized?

- Be Lazy
  - Change state as little as possible
  - Try to group primitives by type
  - Beware of "under the covers" state changes
    - `GL_COLOR_MATERIAL`
      - may force an update to the lighting cache ever call to `glColor*()`

How Can Validations Be Minimized? (cont.)

- Beware of `glPushAttrib()` / `glPopAttrib()`
  - Very convenient for writing libraries
  - Saves lots of state when called
    - All elements of an *attribute groups* are copied for later
  - Almost guaranteed to do a validation when calling `glPopAttrib()`

State Sorting

- Simple technique … Big payoff
  - Arrange rendering sequence to minimize state changes
  - Group primitives based on their state attributes
  - Organize rendering based on the expense of the operation

State Sorting (cont.)

Texture Download

Most Expensive

Modifying Lighting Parameters

Matrix Operations

Vertex Data

Least Expensive

State Sorting – Additional Considerations

- Rendering passes may use more than one texture
  - multi-texturing
- Can't (arbitrarily) sort on Vertex and Fragment progams
  - Fragment program may use generated output of a specific vertex program

A Comment on Encapsulation

- An Extremely Handy Design Mechanism, however …
  - Encapsulation may affect performance
    - Tendency to want to complete all operations for an object before continuing to next object
      - limits state sorting potential
      - may cause unnecessary validations

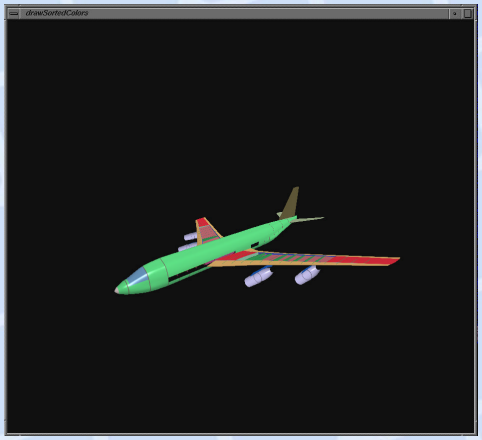
A Comment on Encapsulation (cont.)

- Using a "visitor" type pattern can reduce state changes and validations
  - Usually a two-pass operation
    - ① Traverse objects, building a list of rendering primitives by state and type
    - ② Render by processing lists
  - Popular method employed by many scene-graph packages

Case Studies

Case Study: Application Description

- 1.02M Triangles
- 507K Vertices
- Vertex Arrays
  - Colors
  - Normals
  - Coordinates
- Color Material



Case Study: What's the Problem?

- Low frame rate
  - On a machine capable of 13M polygons/second application was getting less than 1 frame/second

$$\frac{13.1\,\mathrm{M}\,\frac{\text{polygons}}{\text{second}}}{1.02\,\mathrm{M}\,\frac{\text{triangles}}{\text{frame}}} \approx 12\,\frac{\text{frames}}{\text{second}}$$

  - Application wasn't fill limited

Case Study: The Rendering Loop

- Vertex Arrays

```
glVertexPointer( GL_VERTEX_POINTER );
glNormalPointer( GL_NORMAL_POINTER );
glColorPointer( GL_COLOR_POINTER );
```

- **`glDrawElements()`** – index based rendering
- Color Material

```
glColorMaterial( GL_FRONT,
  GL_AMBIENT_AND_DIFFUSE );
```

Case Study: What To Notice

- Color Material changes two lighting material components per **`glColor*()`** call
- Not that many colors used in the model
  - 18 unique colors, to be exact
  - (3 * 1020472 – 18) = 3061398 "redundant" color calls per frame

Case Study: Conclusions

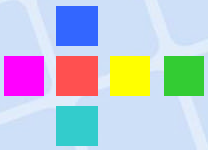
- A little state sorting goes a long way
  - Sort triangles based on color
  - Rewriting the rendering loop slightly

```
for ( i = 0; i < numColors; ++i ) {
   glColor3fv( color[i] );
   glDrawElements( …, trisForColor[i] );
}
```

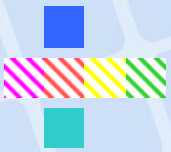
  - Frame rate increased to six frames/second
    - 500% performance increase

Case Study: Rendering A Cube

- More than one way to render a cube
  - Render 100000 cubes



Render six separate quads



Render two quads, and one quad-strip

Case Study: Method 1

- Once for each cube …

```
glColor3fv( color );
for ( i = 0; i < NUM_CUBE_FACES; ++i ) {
  glBegin( GL_QUADS );
  glVertex3fv( cube[cubeFace[i][0]] );
  glVertex3fv( cube[cubeFace[i][1]] );
  glVertex3fv( cube[cubeFace[i][2]] );
  glVertex3fv( cube[cubeFace[i][3]] );
  glEnd();
}
```

Case Study: Method 2

- Once for each cube …

```
glColor3fv( color );
glBegin( GL_QUADS );
for ( i = 0; i < NUM_CUBE_FACES; ++i ) {
 glVertex3fv( cube[cubeFace[i][0]] );
 glVertex3fv( cube[cubeFace[i][1]] );
 glVertex3fv( cube[cubeFace[i][2]] );
 glVertex3fv( cube[cubeFace[i][3]] );
}
glEnd();
```

Case Study: Method 3


```
glBegin( GL_QUADS );
for ( i = 0; i < numCubes; ++i ) {
  for ( i = 0; i < NUM_CUBE_FACES; ++i ) {
    glVertex3fv( cube[cubeFace[i][0]] );
    glVertex3fv( cube[cubeFace[i][1]] );
    glVertex3fv( cube[cubeFace[i][2]] );
    glVertex3fv( cube[cubeFace[i][3]] );
  }
}
 glEnd();
```

Case Study: Method 4

Once for each cube …

```
glColor3fv( color );

glBegin( GL_QUADS );
glVertex3fv( cube[cubeFace[0][0]] );
glVertex3fv( cube[cubeFace[0][1]] );
glVertex3fv( cube[cubeFace[0][2]] );
glVertex3fv( cube[cubeFace[0][3]] );

glVertex3fv( cube[cubeFace[1][0]] );
glVertex3fv( cube[cubeFace[1][1]] );
glVertex3fv( cube[cubeFace[1][2]] );
glVertex3fv( cube[cubeFace[1][3]] );
glEnd();
```

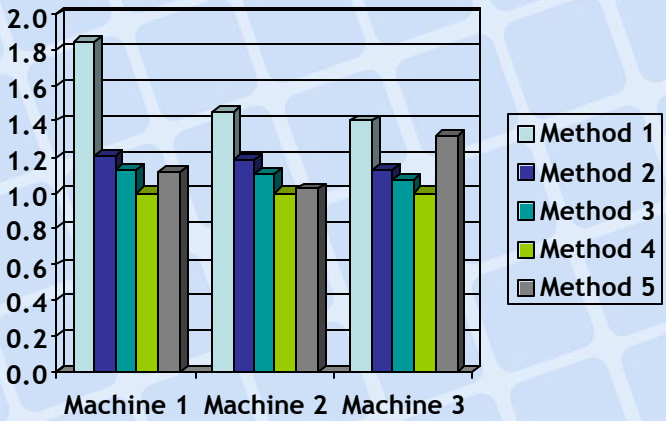
```
glBegin( GL_QUAD_STRIP );
for ( i = 2; i < NUM_CUBE_FACES; ++i ) {
    glVertex3fv( cube[cubeFace[i][0]] );
    glVertex3fv( cube[cubeFace[i][1]] );
}
glVertex3fv( cube[cubeFace[2][0]] );
glVertex3fv( cube[cubeFace[2][1]] );
glEnd();
```

Case Study: Method 5


```
glBegin( GL_QUADS );
for ( i = 0; i < numCubes; ++i ) {
  Cube& cube = cubes[i];
  glColor3fv( color[i] );

  glVertex3fv( cube[cubeFace[0][0]] );
  glVertex3fv( cube[cubeFace[0][1]] );
  glVertex3fv( cube[cubeFace[0][2]] );
  glVertex3fv( cube[cubeFace[0][3]] );

  glVertex3fv( cube[cubeFace[1][0]] );
  glVertex3fv( cube[cubeFace[1][1]] );
  glVertex3fv( cube[cubeFace[1][2]] );
  glVertex3fv( cube[cubeFace[1][3]] );
}
glEnd();
```

```
for ( i = 0; i < numCubes; ++i ) {
  Cube& cube = cubes[i];
  glColor3fv( color[i] );

  glBegin( GL_QUAD_STRIP );
  for ( i = 2; i < NUM_CUBE_FACES; ++i ){
    glVertex3fv( cube[cubeFace[i][0]] );
    glVertex3fv( cube[cubeFace[i][1]] );
  }
  glVertex3fv( cube[cubeFace[2][0]] );
  glVertex3fv( cube[cubeFace[2][1]] );
  glEnd();
}
```

Case Study: Results

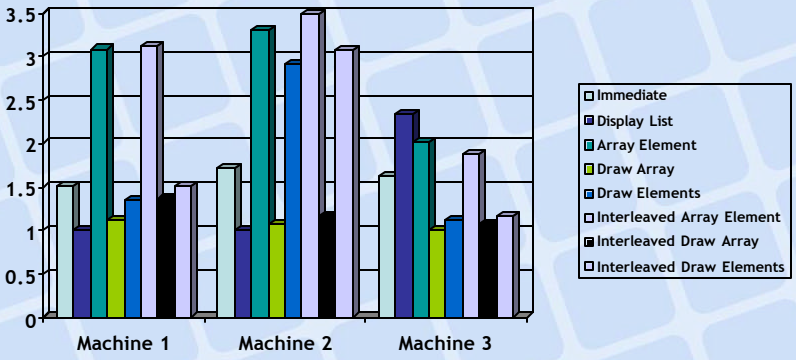


Rendering Geometry

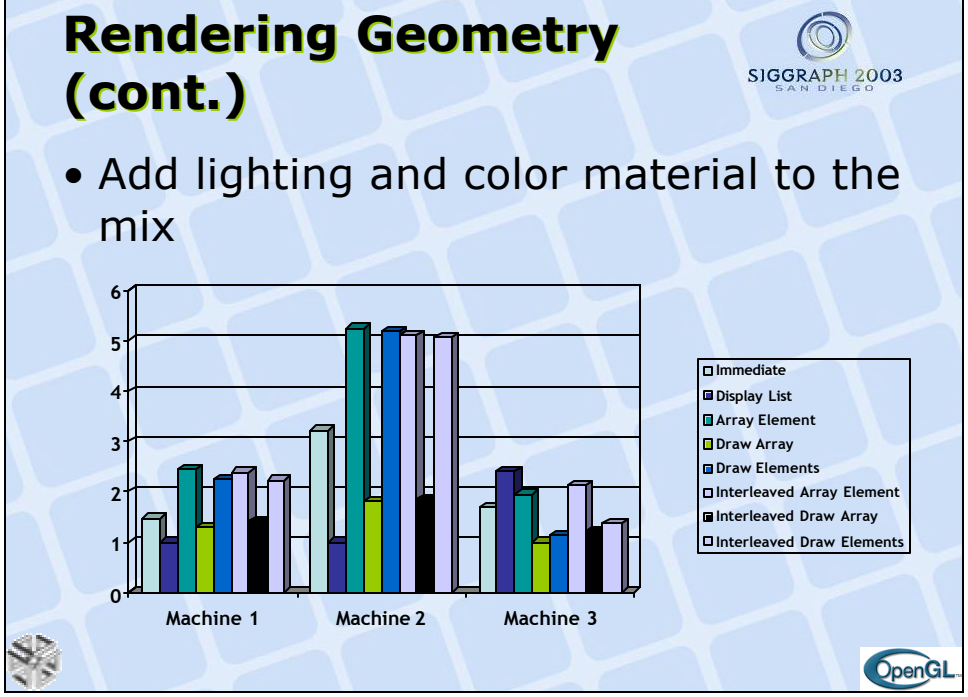
- OpenGL has four ways to specify vertex-based geometry
  - Immediate mode
  - Display lists
  - Vertex arrays
  - Interleaved vertex arrays

Rendering Geometry (cont.)

- Not all ways are created equal



Rendering Geometry (cont.)

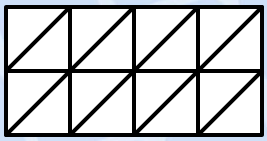


- Add lighting and color material to the mix

OpenGL's Advanced Geometry Storage

Triangle and Quad Strips

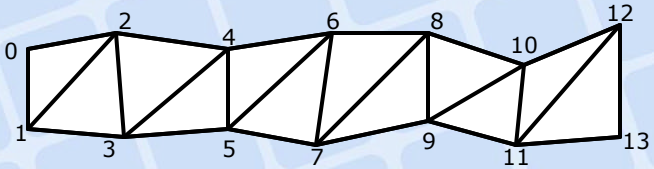
- Often many triangles or quads share vertices
  - E.g. tessellation of a grid



  - When vertices are shared, use **GL_TRIANGLE_STRIP** or **GL_QUAD_STRIP** if possible

Triangle and Quad Strips

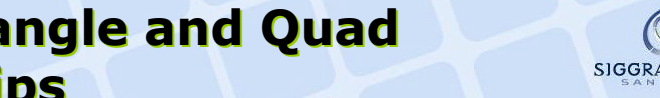
- **GL_TRIANGLES** draws n triangles for 3 * n vertices
- **GL_TRIANGLE_STRIP** draws n triangles for n + 2 vertices
  - Can improve performance substantially



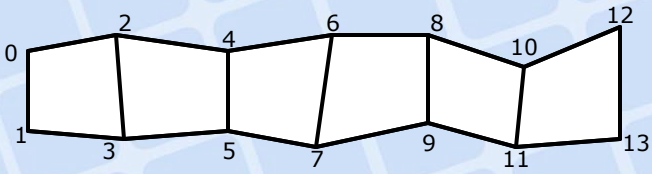
12 triangles drawn, only 14 vertices transformed

0

Triangle and Quad Strips



- `GL_QUAD_STRIP` draws n quads for 2*n + 2 vertices
  - (Often turned into triangle strip anyway)

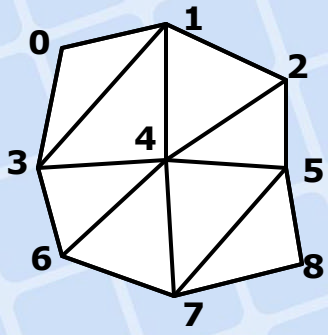


6 quads drawn, only 14 vertices transformed

Vertex Caching - Indexed Primitives

- Many current devices have a "vertex" or "geometry" cache
  - Most recently transformed vertices are kept in a small (10-16 element) cache
- Use indexed primitives to take advantage of the cache
  - `glDrawElements()`
- Use indexed triangle strips for hardware with or without vertex cache

Vertex Caching - Indexed Primitives

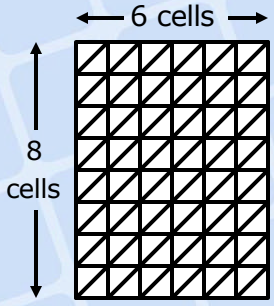


Strips:
0 3 1 4 2 5
3 6 4 7 5 8

Actually transformed:
0 3 1 4 2 5 6 7 8

Vertices transformed per triangle:
1.125
(vs. 1.5 in a strip)

Vertex Caching - Indexed Primitives



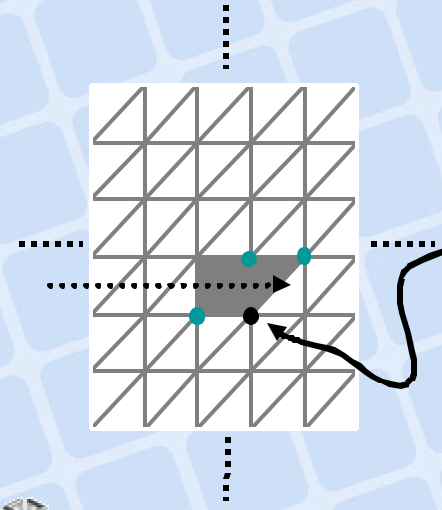
Triangles:
96

Vertices transformed:
63

Vertices transformed per Triangle:
.657
(vs. 1.16 for strip)

Assuming one row can fit in cache

Vertex Caching - Indexed Primitives



Inside the mesh:

Each new vertex enables *two* new triangles!

Cyan vertices already in cache

Black vertex enables both green triangles

Vertex Buffer Objects

- Bus limitation is more and more an issue ...
- Put vertex and index data in graphics memory for best performance!
- New `GL_ARB_vertex_buffer_object` extension
- Uses existing vertex array API
  - But a current *buffer* is accessed and app provides offsets instead of pointers

Vertex Buffer Objects

- Vertex buffer API is similar to textures:
  - `glGenBuffersARB(bufCount, buffers);`
  - `glBindBufferARB(target, buffer);`
  - `glDeleteBuffersARB(bufCount, buffers);`
- Target for binding is either:
  - `GL_ARRAY_BUFFER_ARB` for vertex data
  - `GL_ELEMENT_ARRAY_BUFFER_ARB` for index data

Vertex Buffer Objects

- Can load new data or replace range of data:
  - `glBufferDataARB(target, byteCount, srcData, usage);`
  - `glBufferSubDataARB(target, byteOffset, byteCount, srcData);`
- Whole series of "usage" enumerants
  - `GL_STATIC_DRAW_ARB` written once, never read
  - `GL_DYNAMIC_DRAW_ARB` written repeatedly
  - Among others...

Vertex Buffer Objects

- Can map buffer data into memory for access
  - `glMapBufferARB(target, accessPattern);`
  - `glUnmapBufferARB(target);`
- Access pattern indicates intention
  - `GL_READ_ONLY_ARB` App won't change
  - `GL_WRITE_ONLY_ARB` App won't read
  - `GL_READ_WRITE_ARB` …
  - OpenGL doesn't enforce with errors, but deviating from intention may be slow

Vertex Buffer Objects

- Example vertex buffer setup:

```
#define OFFSET(a) ((char *)NULL + a)
glGenBuffersARB(1, &vbuffer);
glBindBufferARB(GL_ARRAY_BUFFER_ARB,
   vbuffer);
glBufferDataARB(GL_ARRAY_BUFFER_ARB,
   sizeof(vert) * numverts, vertData,
   GL_STATIC_DRAW_ARB);
glInterleavedArrays(GL_N3F_V3F,
   sizeof(vert), OFFSET(offset));
```

- Then call glDrawArrays in draw func

Vertex Buffer Objects

- Can also set up element array:

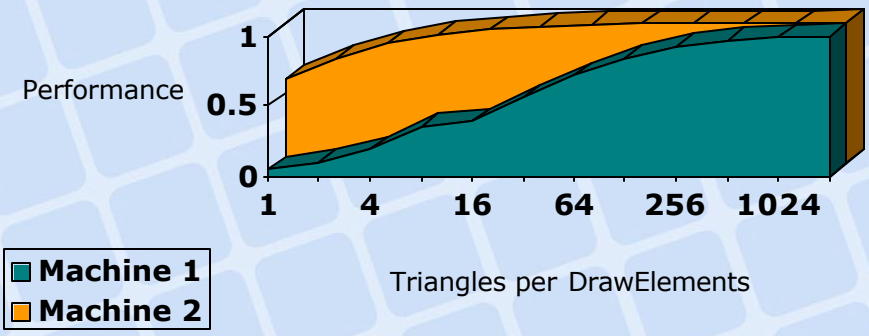
```
/* set up vertices… */
glGenBuffersARB(1, &ebuffer);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB,
   ebuffer);
glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB,
   sizeof(unsigned int) * numIndices,
   indexData, GL_STATIC_DRAW_ARB);
```

- Then call glDrawElements in draw:

```
glDrawElements(GL_TRIANGLES, indexCount,
  GL_UNSIGNED_INT, OFFSET(offset));
```

Vertex Buffer Objects

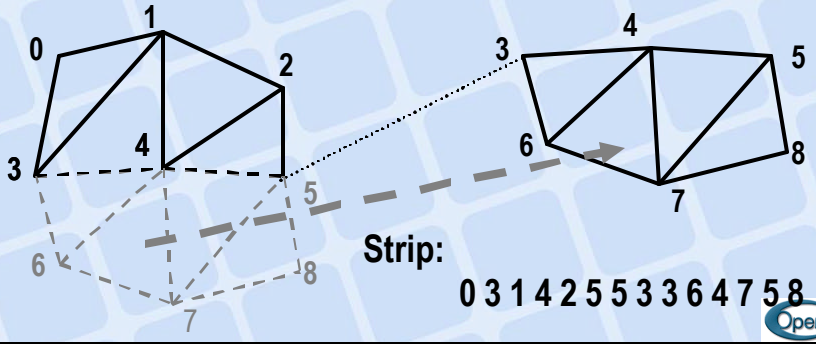
- DrawElements is somewhat expensive



- So draw more tris per DrawElements

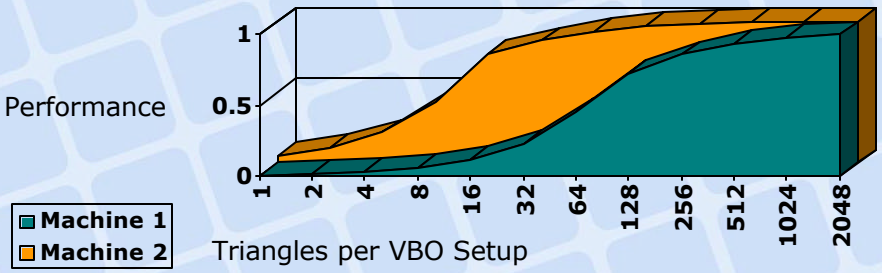
Vertex Buffer Objects

- Connect strips with empty triangles to reduce overhead (func call turnaround much higher than 0-pixel triangle overhead)



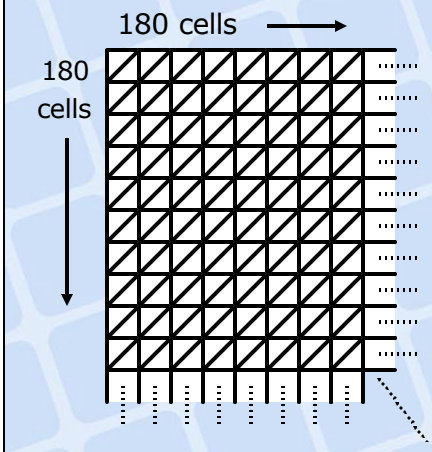
Vertex Buffer Objects

- Setting up VBO and array is more expensive



- So `glBindBufferARB()` only when necessary

Indexed Primitives and Vertex Buffers

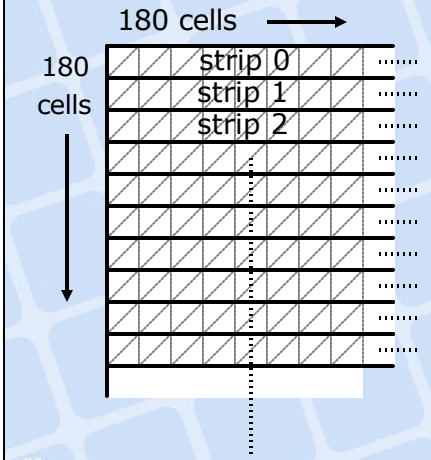



```
ARB_vertex_buffer_object

glDrawArrays(GL_TRIANGLES,
             ...);
```

194400 Vertices:
4.66MB (V3F_N3F)

20 Million Triangles per Second

Indexed Primitives and Vertex Buffers



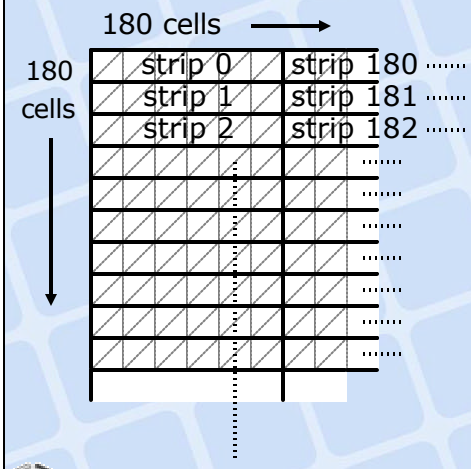

```
ARB_vertex_buffer_object

glDrawArrays(
     GL_TRIANGLE_STRIP,
     ...)
```

65160 vertices:
1.56MB (V3F_N3F)

55 Million Triangles per Second

Performance Increase :
2.25x

Indexed Primitives and Vertex Buffers




```
ARB_vertex_buffer_object

glDrawElements(
      GL_TRIANGLE_STRIP,
      ...
```

32761 vertices:
786 KB (V3F_N3F)
75600 indices:
302K
Total 1.09MB

98 Million Tris per Second

Performance increase :
4.9x

Another Possibility: Occlusion Query

- `NV_occlusion_query` now, `ARB_occlusion_query` soon, core OpenGL 1.5 after that
- Ask OpenGL how many pixels covered by sequence of commands
  - Draw a simple representation (“proxy”) without color and depth update
    - Disable lighting, texturing, etc
  - If the proxy drew zero pixels, don’t bother drawing the real thing

One More Possibility: Occlusion Query

- Proxy geometry
  - Reasonably tight bounding volume
  - No texturing, no shading
  - No lighting, no `glTexGen()`
  - No colors, tex coords, normals
- Trading off a little fill and transform for potential reduction of a lot of fill and transform

Occlusion Query

- Occlusion Query
  - First, generate queries at initialization time
  - Like Textures or Display Lists
  - `glGenOcclusionQueriesNV(int count, unsigned int queryIDs[]);`

Occlusion Query

- Occlusion Query
  - Render a bunch of stuff (big occluders)
  - Then start queries for a bunch more stuff

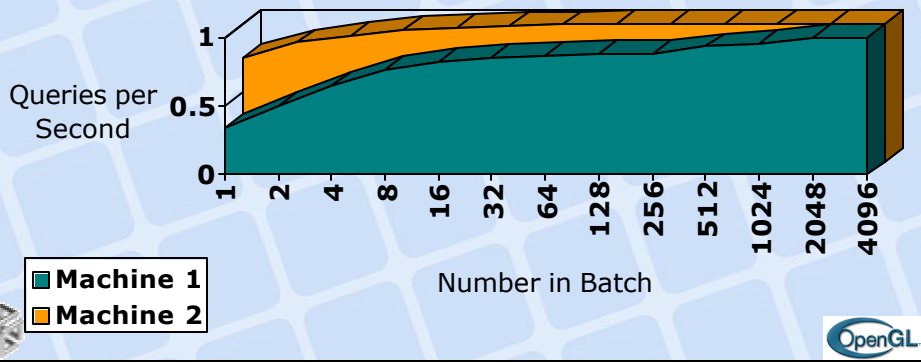
```
glBeginOcclusionQueryNV(int querynum);
/* draw proxy geometry... */
glEndOcclusionQueryNV();
```

  - Then read results of queries
    - `glGetOcclusionQuery{iv,uiv}NV(`

```
int querynum, GL_PIXEL_COUNT_NV,
int *samplesCounted);
```

Occlusion Query

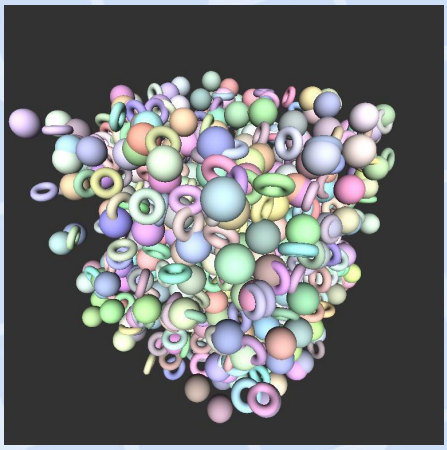
- But getting query result is synchronous, so issue a batch of queries, *then* read results for batch



Occlusion Query


```
Sort objects front-back into batches
For each batch
   For each object in batch
        BeginQuery
        Draw proxy
        EndQuery
   For each object in batch
        GetQuery
        if(samples > 0)
             Draw object
```

Occlusion Query

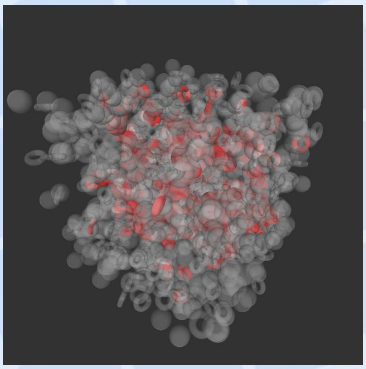


Without occlusion query: 15 fps

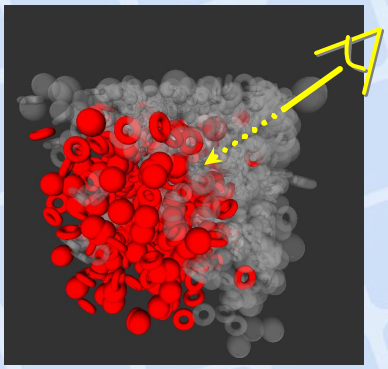
With occlusion query: 25 fps

10% batching

Occlusion Query



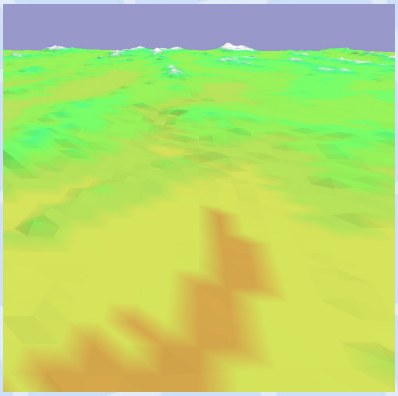
Occluded objects drawn in red



Scene from the side

Case Study: Terrain

- 4001x4801 Height Field
- Drawn at 1/256th resolution
  - 75,000 GL_TRIANGLES
- All data generated *per frame*
  - Vertices
  - Colors
  - Normals
- Color Material
- *10 frames / sec*



Case Study: Terrain

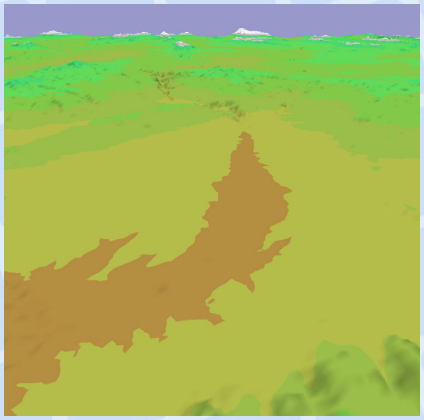
```
const GLfloat color0[3]  = { 0.65, 0.40, 0.10 };
const GLfloat color1[3]  = { 0.60, 0.50, 0.15 };
const GLfloat color2[3]  = { 0.65, 0.55, 0.25 };
const GLfloat color3[3]  = { 0.70, 0.55, 0.25 };
const GLfloat color4[3]  = { 0.70, 0.75, 0.30 };
const GLfloat color5[3]  = { 0.60, 0.75, 0.30 };
const GLfloat color6[3]  = { 0.50, 0.80, 0.30 };
const GLfloat color7[3]  = { 0.40, 0.85, 0.35 };
const GLfloat color8[3]  = { 0.30, 0.85, 0.45 };
const GLfloat color9[3]  = { 0.80, 0.80, 0.80 };
const GLfloat color10[3] = { 1.00, 1.00, 1.00 };
```

```
if (elev < 0.0)          glColor3fv (color0);
else if (elev < 304.8)   glColor3fv (color1);
else if (elev < 609.6)   glColor3fv (color2);
else if (elev < 914.4)   glColor3fv (color3);
else if (elev < 1219.2)  glColor3fv (color4);
else if (elev < 1524.0)  glColor3fv (color5);
else if (elev < 1828.8)  glColor3fv (color6);
else if (elev < 2133.6)  glColor3fv (color7);
else if (elev < 2438.4)  glColor3fv (color8);
else if (elev < 2743.2)  glColor3fv (color9);
else                     glColor3fv (color10);
```

Case Study: Terrain

- Techniques Used:
  - Pre-compute values
    - (downside is memory footprint)
  - Break terrain data into tiles
  - Use Vertex Buffer Object – let system load geometry on demand
  - Use `glTexGen()` for color
  - Use indexed `GL_TRIANGLE_STRIPs` for vertex-cache friendliness
  - Use occlusion query for easy frustum cull

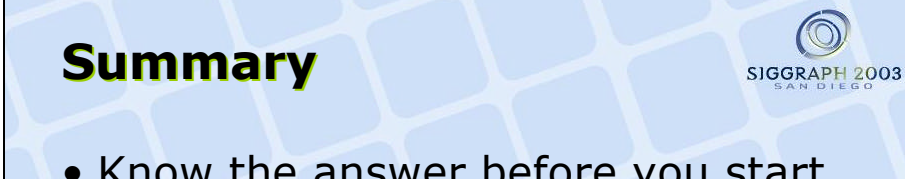
Case Study: Terrain



- Results:
  - 1/5 resolution
    - 2000 x 2000
    - 50x improvement
  - Typically > 10 Hz
    - 3 Hz - 60 Hz depending on viewpoint
  - Per-pixel height colormap

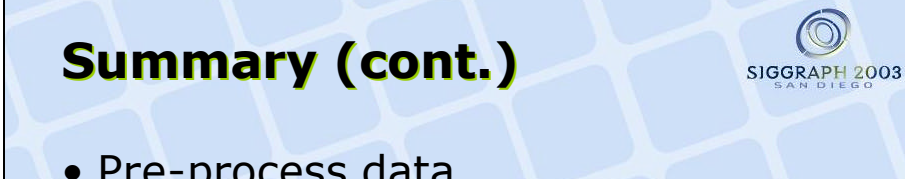
Conclusions and Future Work

Summary



- Know the answer before you start
  - Understand rendering requirements of your applications
    - Have a performance goal
  - Utilize applicable benchmarks
    - Estimate what the hardware's capable of
  - Organize rendering to minimize OpenGL validations and other work

Summary (cont.)



- Pre-process data
  - Convert images and textures into formats which don't require pixel conversions
  - Pre-size textures
    - Simultaneously fit into texture memory
    - Mipmaps
  - Determine what's the best format for sending data to the pipe

Questions & Answers

- Thanks for coming
  - Updates to notes and slides will be available at
    `http://www.PerformanceOpenGL.com/`
  - Feel free to email if you have questions

Dave Shreiner
`shreiner@sgi.com`

Alan Commike
`commike@sgi.com`

Brad Grantham
`grantham@sgi.com`

Bob Kuehne
`rpk@sgi.com`

References

- *OpenGL Programming Guide*, 3rd Edition
  Woo, Mason et. al., Addison Wesley
- *OpenGL Reference Manual*, 3rd Edition
  OpenGL Architecture Review Board, Addison Wesley
- *OpenGL Specification*, Version 1.2.1
  OpenGL Architecture Review Board

Acknowledgements

- A Big Thank You to …
  - SGI
  - Peter Shaheen for a number of the benchmark programs
  - David Shirley and Frank Merica for the Case Study applications
  - 3DLabs for lending us hardware

Questions?

Thanks for Coming!