OpenGL Essential Design Practices

Best practices for effective OpenGL programming

Dan Omachi
OpenGL Development Engineer

What Is OpenGL?

"OpenGL is a software interface to graphics hardware"

- OpenGL Specification

What Does the GPU Do?

- GPU accelerates rendering
- Faster rendering = better image quality
  - Drawing efficiently allows drawing more
  - More models, more vertices, more pixels, longer shaders

What You'll Learn

- Using OpenGL efficiently
- OpenGL under the hood
  - State Validation:
    - Translates GL calls into GPU commands
    - A CPU intensive operation
- Fundamental OpenGL techniques
  - Avoiding CPU bottlenecks
  - Efficient GPU access
- Apply equally to iOS 4 and Mac OS X

OpenGL's Design

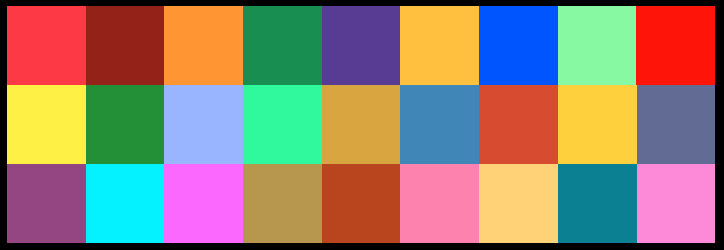
- A low level API
  - Direct access to GPU capability
- High level enough to drive many GPUs
  - Lots of work to translate to GPU

OpenGL's Design

Designed to be a low level API

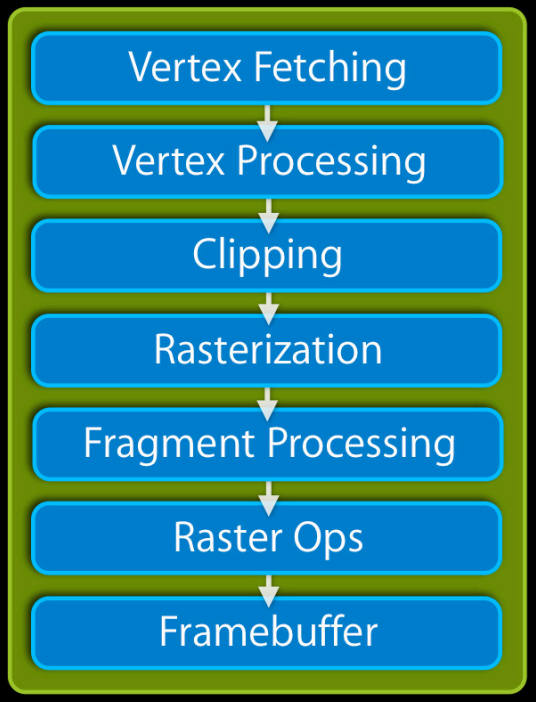
- A "state machine"
  - Maps to the GPU pipeline

Context State



≈

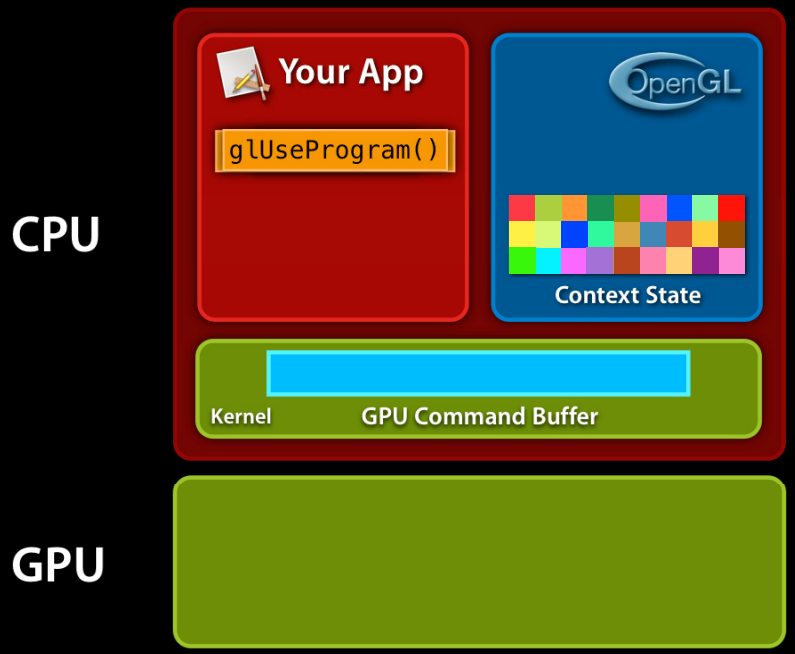
GPU Pipeline



OpenGL: Under the Hood

The OpenGL software stack

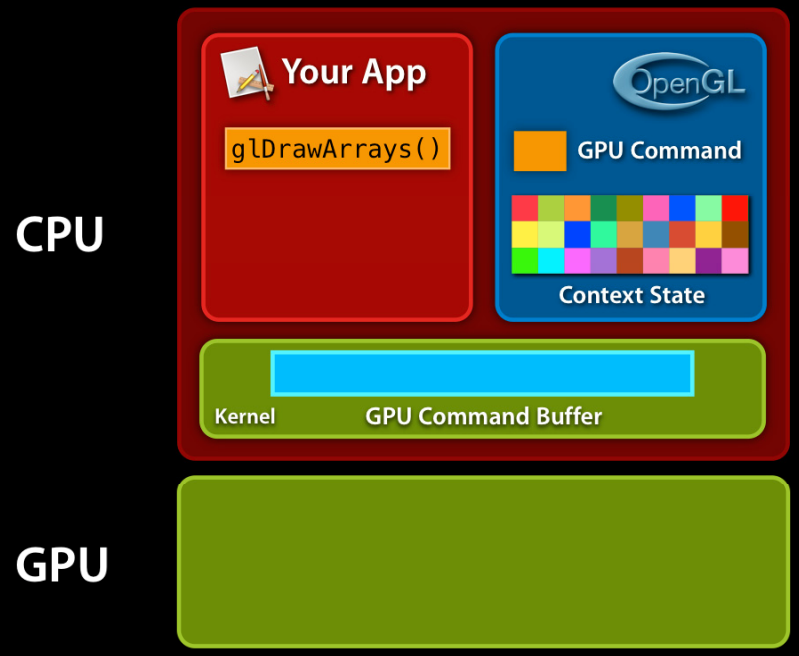
- When a state change call is made
  - Context state updated



OpenGL: Under the Hood

The OpenGL software stack

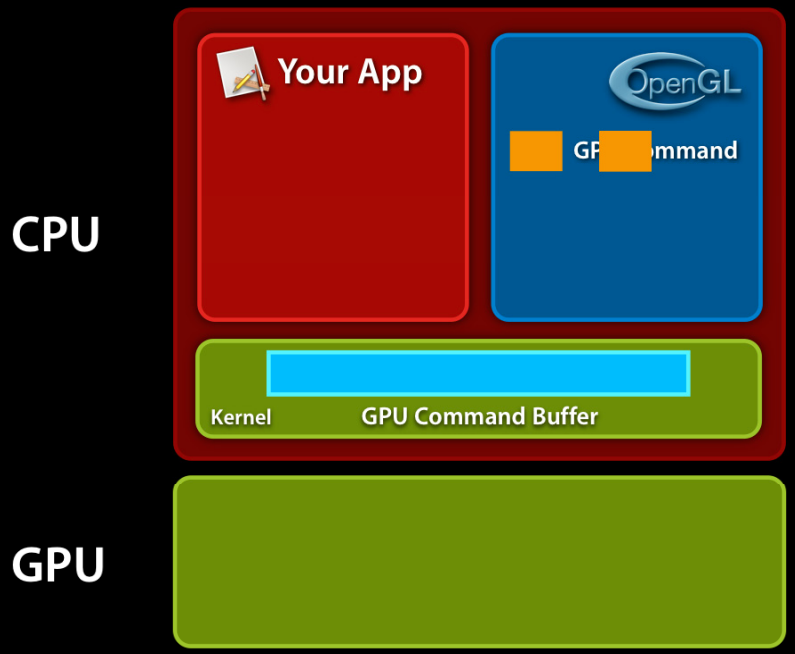
- API calls translated into GPU commands
  - Work deferred until draw call
  - Translation stage:
    CPU intensive operation



OpenGL: Under the Hood

The OpenGL software stack

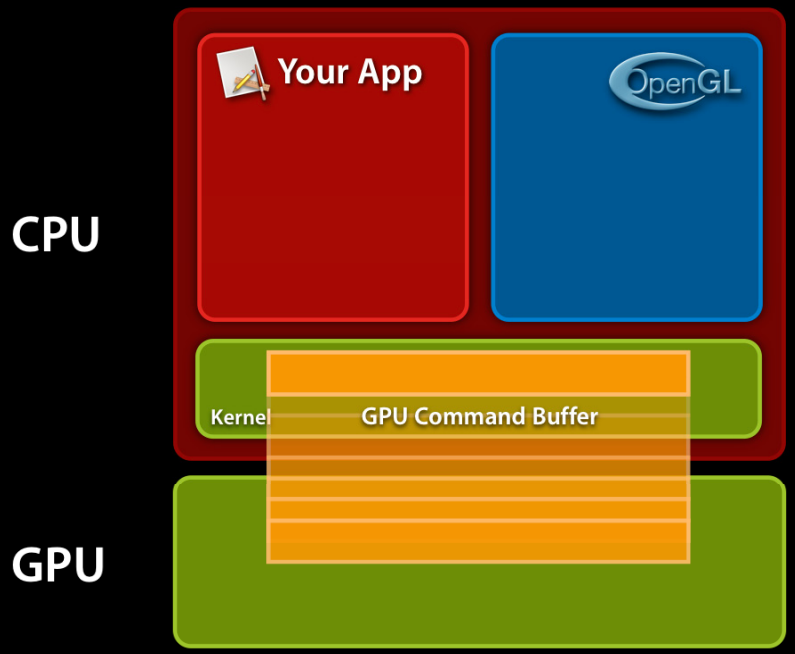
- GPU commands are inserted into a command buffer
- If the buffer fills up OR `glFlush` called…



OpenGL: Under the Hood

The OpenGL software stack

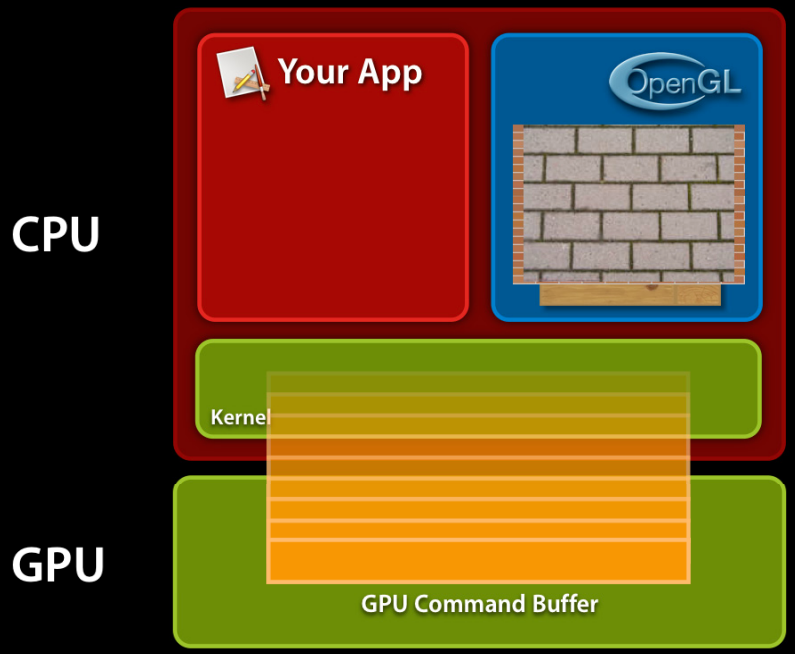
- Command buffer is flushed to GPU



OpenGL: Under the Hood

The OpenGL software stack

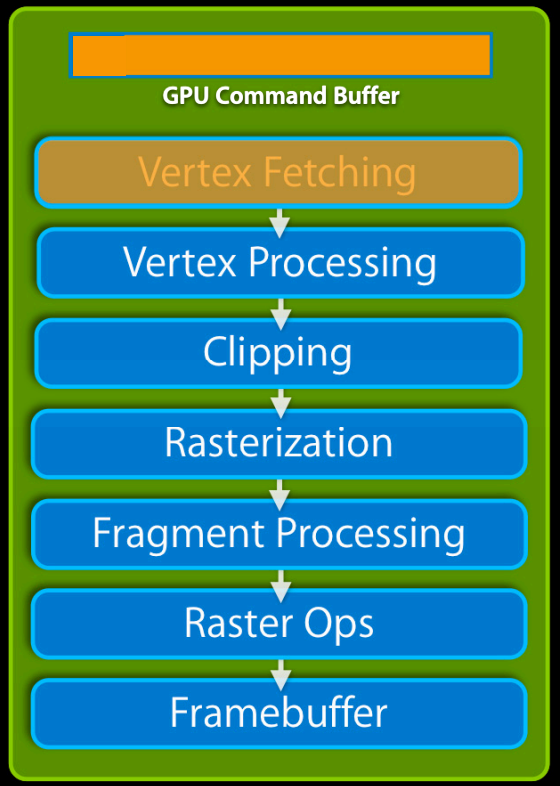
- Textures and VBOs needed are loaded into GPU
- Flushing also a CPU intensive process



OpenGL: Under the Hood

The OpenGL software stack

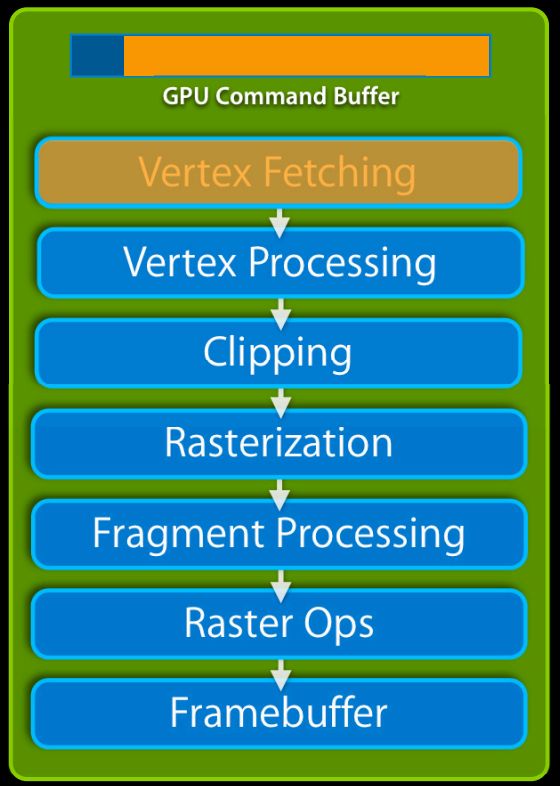
- GPU processes the command buffer
  - Starts pipeline by fetching vertices



OpenGL: Under the Hood

The GPU pipeline execution

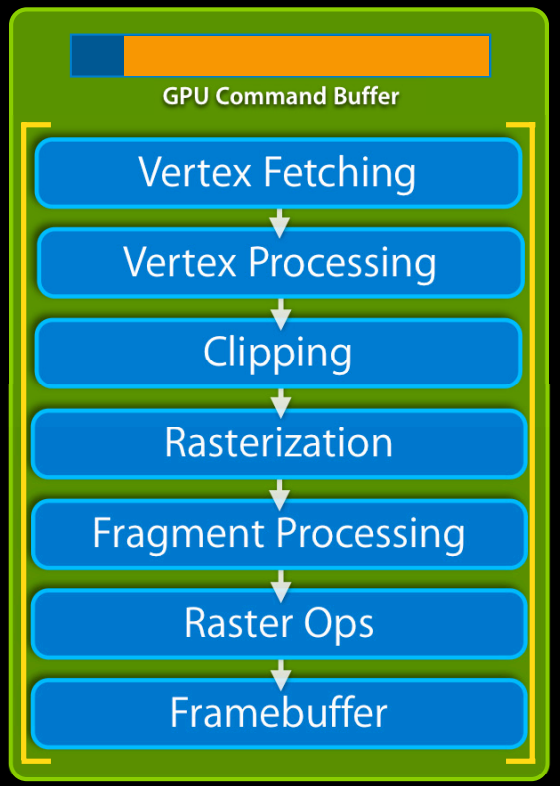
- Sends the data down the GPU Pipeline



OpenGL: Under the Hood

The GPU pipeline execution

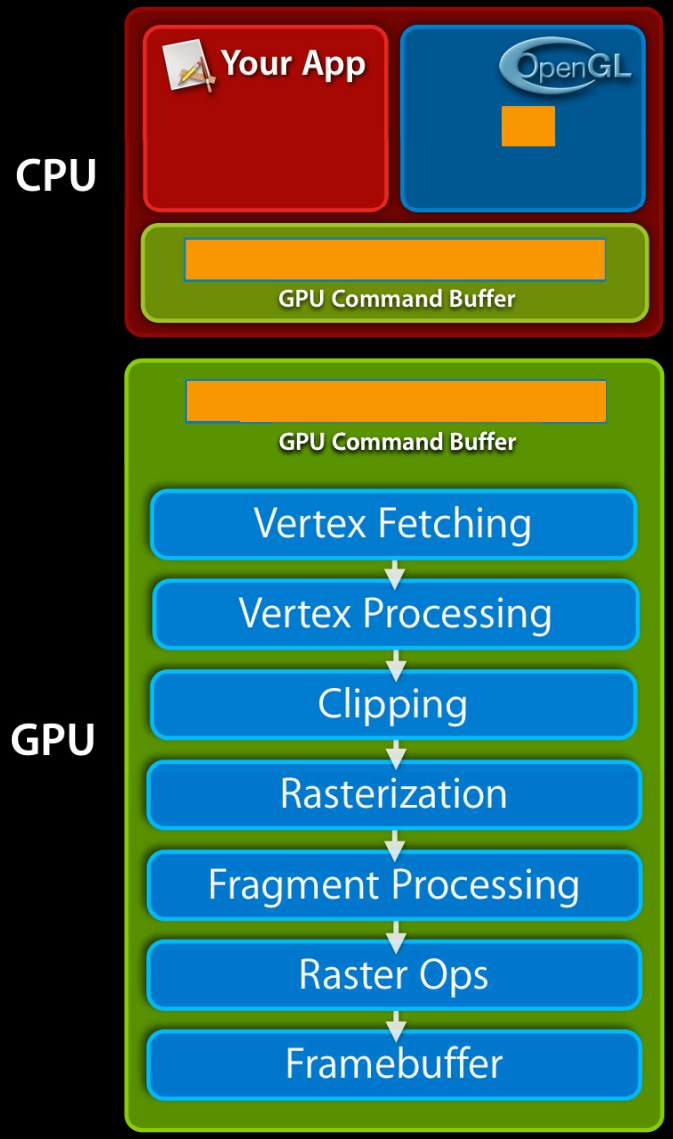
- Many potential bottlenecks on the GPU
- Common bottleneck is the CPU



OpenGL: Under the Hood

CPU/GPU parallelism

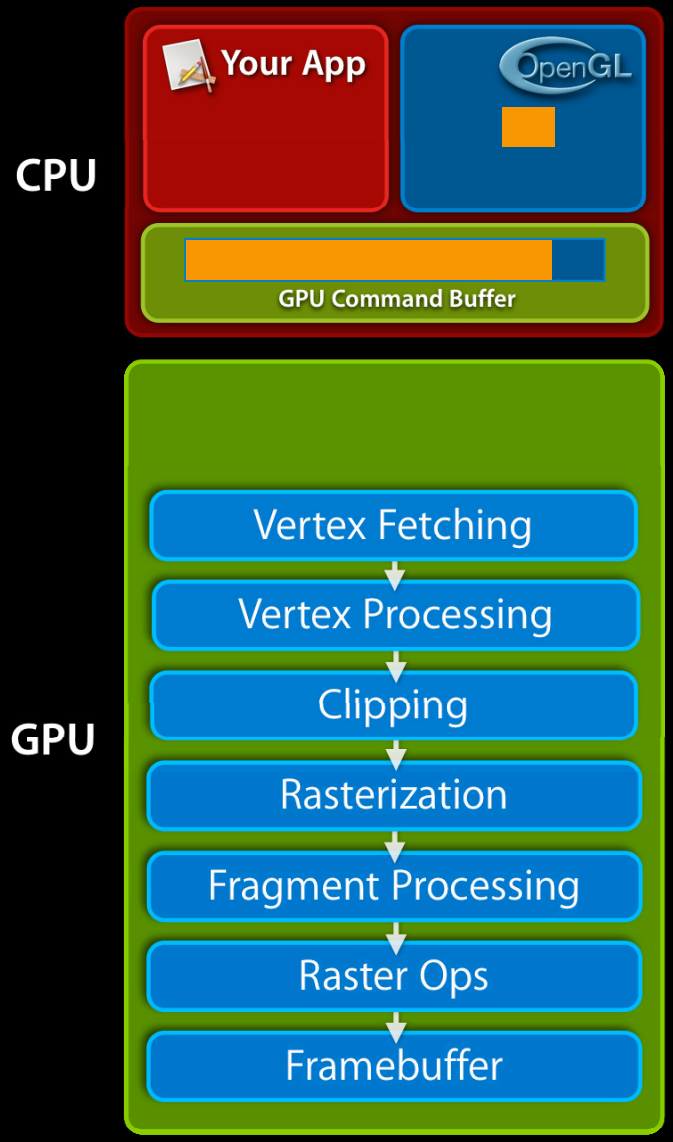
- Key point: GPU another processor
  - Parallel to the CPU



OpenGL: Under the Hood

CPU/GPU parallelism

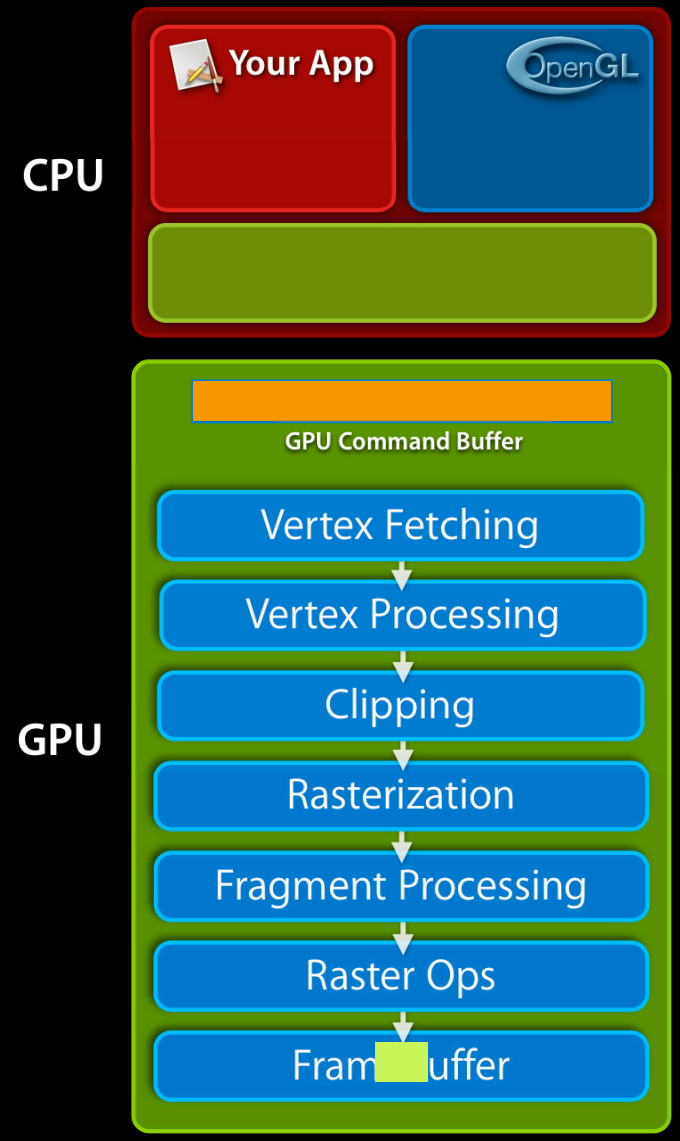
- Don't let GPU wait for CPU
  - Don't waste CPU for graphics
  - Offload the graphics work to the GPU



OpenGL: Under the Hood

CPU/GPU parallelism

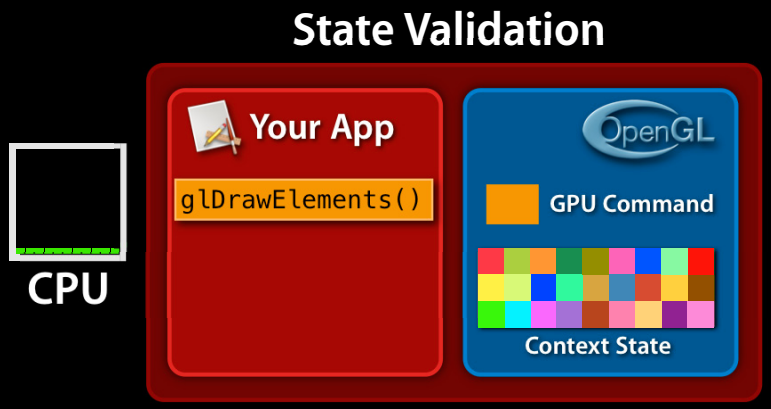
- Don't let the CPU wait for the GPU
  - Anytime your app needs data from the GPU, it can stall the CPU
    - Readpixels, queries, fences



OpenGL: Under the Hood

GPU command translation (state validation)

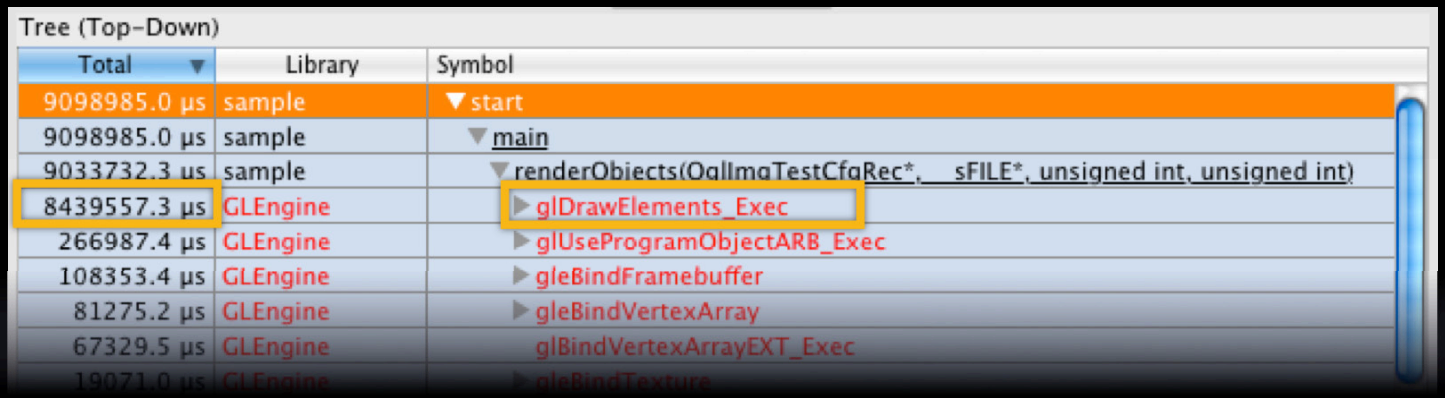
- For each draw call
  - Context state and Draw call translated to GPU Commands
    - Called "state validation"
    - CPU intensive operation



OpenGL: Under the Hood

State validation cost

- Draw calls look CPU intensive in a Shark profile
  - Most of cost result of state setting
- i.e. State setting cost won't show up in the state set call
  - Shows up in subsequent draw call



Reducing Validation Overhead

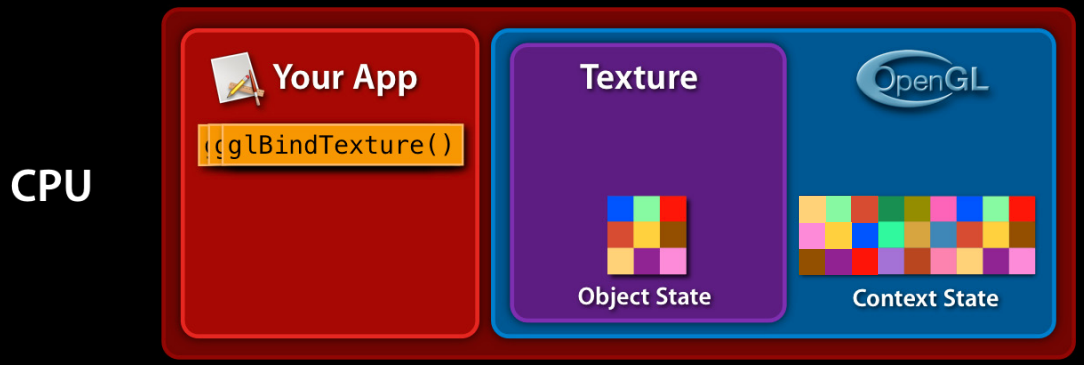
Summary of techniques

- Use OpenGL's objects
- Manage rendering state
  - Sort state
- Batching state
  - Reduce setting state by combining object

OpenGL Objects

Objects and state validation

- Pre-validated state cached in objects
  - Not validated at draw time
  - When bound, easily translated to GPU commands
- Set up objects when app/level/document loads
  - Don't wait until it's in the middle of your real-time run loop

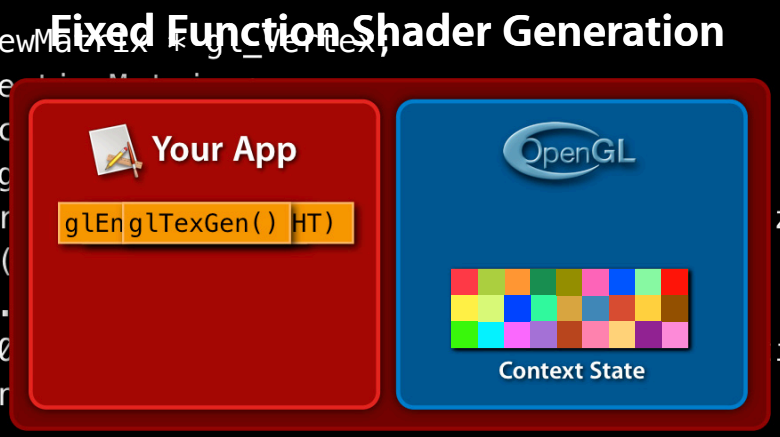


OpenGL Objects

Fixed function: Shader in disguise

- Vertex and fragment pipe is entirely programmable
  - No fixed function vertex or fragment pipe on GPU
  - Shader objects created internally to emulate fixed function

```
void main (void)
{
    vec4 eye = gl_ModelViewMatrix * gl_Vertex;
    gl_Position = gl_Proje
    float u = nomalize(vec
    float r = reflect(u, g
    float m = 2.0 * sqrt(r                                  z + 1.0));
    gl_TexCoord[0] = vec2(
    gl_FogFragCoord = eye.
    float nDotVP = max(0.0                                  ion));
    gl_Color = inColor * r
}
```



OpenGL Objects

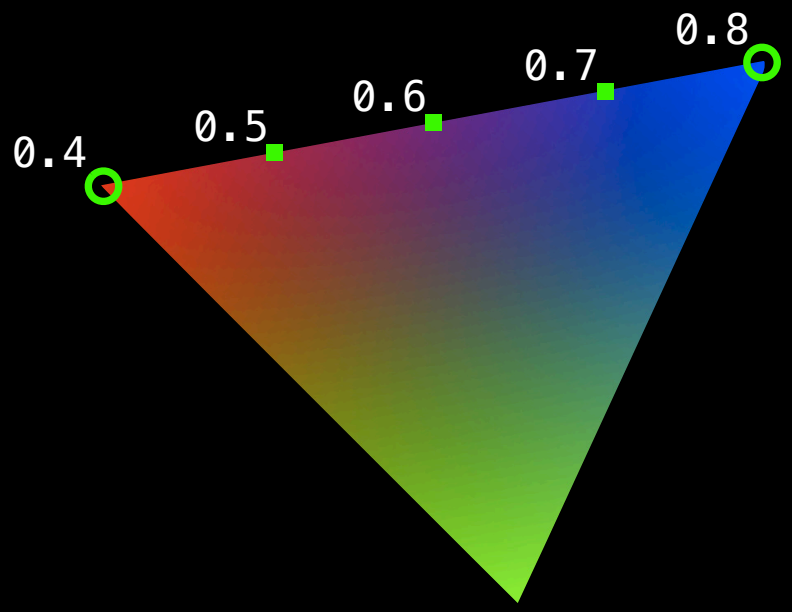
Program objects and pipeline setup

- Program objects—most efficient way to setup pipeline
- Specify shader code, compile, link into a program object

OpenGL Objects

Vertex shader

- Shader executed for each vertex
- Inputs are per-vertex attributes
  - Specified outside the shader
- Two types of outputs
  - One position in clip space
    - `gl_Position`
  - One or many varyings
    - Color, normals, texture coordinates
    - Values interpolated across rasterized triangles



OpenGL Objects

Vertex shader example

```
varying vec2 varTexCoord;
void main (void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    varTexCoord = gl_MultiTexCoord0.st;
}
```

OpenGL Objects

Fixed function built-in shader variables

- Avoid fixed function built-in uniforms, attributes, or varyings in shaders
  - OpenGL needs to perform mapping
  - Not forward compatible
    - Don't exist on OpenGL ES 2.0
    - Coding without them is the future of OpenGL

```
gl_ModelViewProjectionMatrix
gl_ModelViewMatrix
gl_ProjectionMatrix
gl_ModelViewProjectionMatrixInverse
gl_Point
gl_LightSource[]
gl_LightModelParameters
gl_Fog
```

```
gl_Vertex
gl_Color
gl_SecondaryColor
gl_Normal
gl_FogCoord
gl_MultiTexCoord0–7
```

OpenGL Objects

Coding shaders with generics

```
attribute vec4 inPosition;
attribute vec2 inTexCoord;
uniform mat4 modelViewProjectionMatrix;
varying vec2 varTexCoord;
void main (void)
{
    gl_Position = modelViewProjectionMatrix * inPosition;
    varTexCoord = inTexCoord;
}
```

OpenGL Objects

Fragment shader

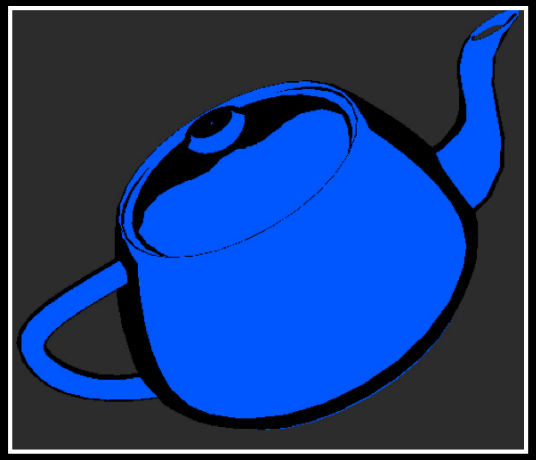
- Runs once per pixel produced by each polygon
- Can render effects not possible with the fixed function pipeline

```
varying vec3 normal;

void main (void)
{
    float edgeness = dot(vec3(0,0,1), normal);

    vec4 color = vec4(0, 0, 1, 1); // Blue
    if (abs(edgeness) < 0.45)
        color = vec4(0, 0, 0, 1);  // Black

    gl_FragColor = color;
}
```



OpenGL Objects

Client side vertex arrays

- `glVertexAttribPointer` points to data to feed vertex shader

```
GLfloat* positionData = (GLfloat*) malloc(positionDataSize);

// Load vertex position data into positionData memory
...
```

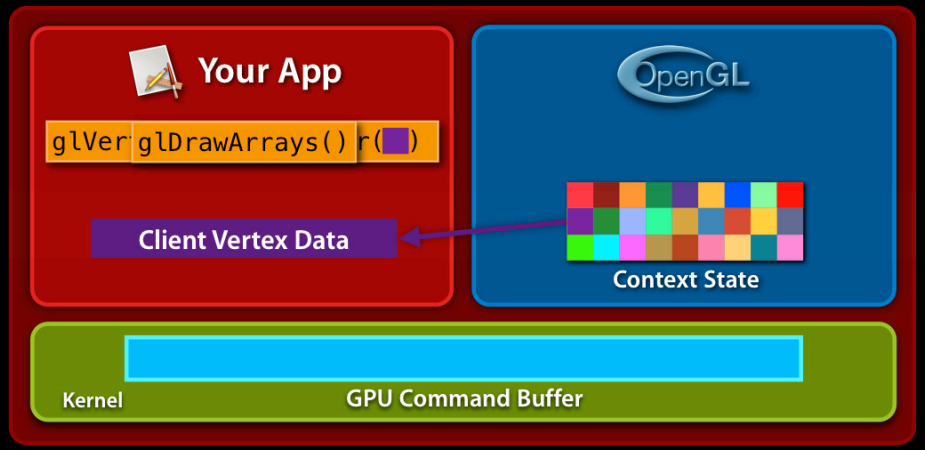
```
glEnableVertexAttrib(positionAttributeIndex);
glVertexAttribPointer(positionAttributeIndex,
                      3,
                      GL_FLOAT,
                      GL_FALSE,
                      12,
                      positionData);
```

OpenGL Objects

Overhead of client-side vertex arrays

- CPU cycles required to copy vertex data
- OpenGL may copy the data to the command buffer
  - Command buffer fills quicker
    - Expensive flushes more frequent

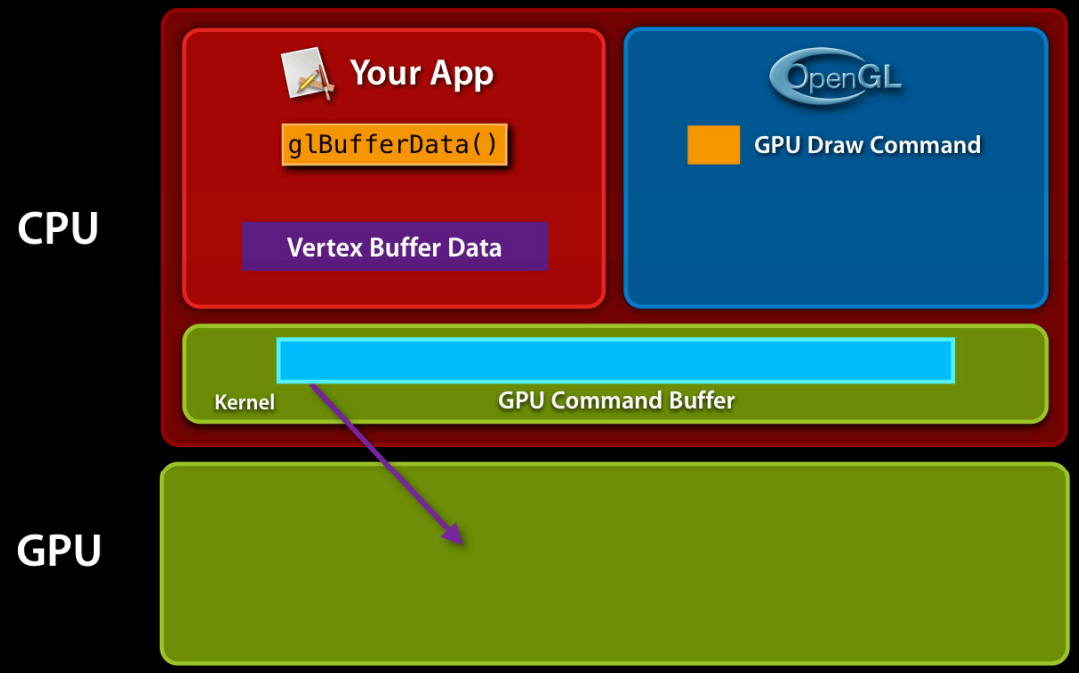
CPU



OpenGL Objects

Vertex buffer objects

- Store vertex data in VBOs
  - OpenGL caches this in the GPU's VRAM
- Draw command references data on GPU



OpenGL Objects

Vertex buffer objects

- Allocate VBO store with `glBufferData`

```
glGenBuffers(1, &vboName);
glBindBuffer(GL_ARRAY_BUFFER, vboName);
```

```
glBufferData(GL_ARRAY_BUFFER,
             positionDataSize,
             positionData,
             GL_STATIC_DRAW);
```

```
glEnableVertexAttrib(positionAttributeIndex);
glVertexAttribPointer(positionAttributeIndex,
                      3,
                      GL_FLOAT,
                      GL_FALSE,
                      12,
                      0); // <- data offset in VBO
```

OpenGL Objects

Dynamic vertex data

- You may want to modify vertex data for animations
  - If data small enough, use multiple VBOs
  - If data generated on-the-fly use `glBufferSubData` or `glMapBuffer`

```
glBufferData(GL_ARRAY_BUFFER,
             positionDataSize,
             positionData,
             GL_DYNAMIC_DRAW);

...
// Modify vertex data to update
...

glBufferSubData(GL_ARRAY_BUFFER,
                updateOffset,
                updateSize,
                updateData);
```

OpenGL Objects

Dynamic vertex data and syncing

- Changing vertex data of a VBO can force GPU to sync with CPU
  - CPU waits for draw to finish before update
  - Can happen with `glBufferSubData` and `glMapBuffer`
- Don't load vertex buffer while it's read by GPU
  - Use double buffering technique

OpenGL Objects

Double buffering VBOs

```
// On odd frames...

glBindBuffer(GL_ARRAY_BUFFER, oddBuffer);
glBufferSubData(GL_ARRAY_BUFFER, offset, size, data);

glDrawArrays(GL_TRIANGLES, 0, numberOfVerticies);

...

// On even frames...

glBindBuffer(GL_ARRAY_BUFFER, evenBuffer);
glBufferSubData(GL_ARRAY_BUFFER, offset, size, data);

glDrawArrays(GL_TRIANGLES, 0, numberOfVerticies);
```

OpenGL Objects

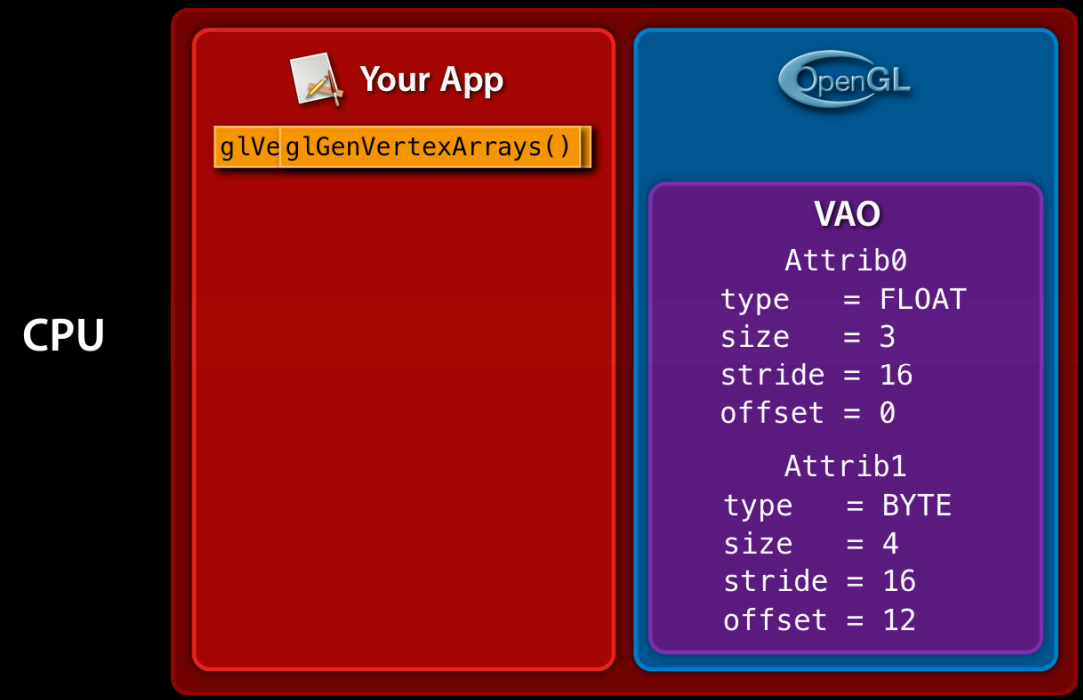
Vertex array layout

- Not only does `glVertexAttribPointer` indicate where vertices live
  - Also specifies vertex layout

OpenGL Objects

Vertex array objects

- Allows OpenGL to cache vertex layout



OpenGL Objects

Coding with vertex array objects

```
glGenVertexArrays(1, &vaoName);
glBindVertexArray(vaoName);
```

```
glEnableVertexAttrib(positionAttributeIndex);
glVertexAttribPointer(positionAttributeIndex, 3, GL_FLOAT,
                      GL_FALSE, 16, 0);
```

```
glEnableVertexAttrib(colorAttributeIndex);
glVertexAttribPointer(colorAttributeIndex, 4, GL_BYTE,
                      GL_TRUE, 16, 12);
```

```
...
```

```
glBindVertexArray(vaoName);
glDrawArrays(GL_TRIANGLES, 0, numberOfVerticies);
```

OpenGL Objects

Framebuffer objects and renderable textures

- With EAGL and OpenGL ES you must always use an FBO
- Attaching textures to FBOs enables some interesting effects
  - Reflections, refractions, and shadows

OpenGL Objects

Framebuffer objects and renderable textures

```
glGenTextures(1, &texName);
glBindTexture(GL_TEXTURE_2D, texName);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 512, 512, 0,
    GL_RGBA, GL_UNSIGNED_BYTE, imageData);
```

```
glGenFramebuffers(1, &fboName);
glBindFramebuffer(GL_FRAMEBUFFER, fboName);
```

```
// Later, when texture is no longer bound...
```

```
glFramebufferTexture2D(GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texName, 0);
```

```
glDrawArrays(GL_TRIANGLES, 0, numVerticies);
```

OpenGL Objects

Object mutability

- An object's state can be modified after its initial setup
- Avoid this
  - Forces OpenGL to re-validate object next time it's used

```
glGenTextures(1, &texName);
glBindTexture(GL_TEXTURE_2D, texName);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 512, 512,
    0, GL_RGBA, GL_UNSIGNED_BYTE, imageData);
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
```

```
...
// After the run-loop has begun
glBindTexture(GL_TEXTURE_2D, texName);
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR)
```

OpenGL Objects

Lazy validation

- Objects aren't actually pre-validated when created
- Objects validated when first used to draw
  - Shader objects can't be compiled until they are used to draw
    - Compiler needs to know other context state set before it can compile
    - May need FBO, VAO, textures bound, and the blend state used
  - Textures and VBOs won't get cached in VRAM until they are first drawn

OpenGL Objects

Pre-warming objects

- Lazy validation may cause hiccups during run-loop
- Avoid validation during run-loop by "pre-warming" objects
  - Bind the object and draw it
    - Use state and other objects it's used with
  - Do this before app's run-loop begins
- Only consider using this if your app experiences hiccups

```
for every program in our scene
    bind the program
    for every VAO used with that program
        bind the VAO
        for every texture used with that VAO and program
            bind the texture
            for every blend state used with this program
                Set the blend state
                Draw
```

OpenGL Objects

Object size

- Know how much memory your objects take
  - All current graphics resources need to fit in memory
  - Some devices have limited VRAM
- Use compressed textures
- Fit texture to size of model rendered
- Fit entire frame's resources in to VRAM
  - If possible, fit entire level's/scene's textures into VRAM

OpenGL Objects

Cost of object binding

- Some objects more expensive
- Determine cost through profiling
- Batch draw calls to reduce binding more expensive objects

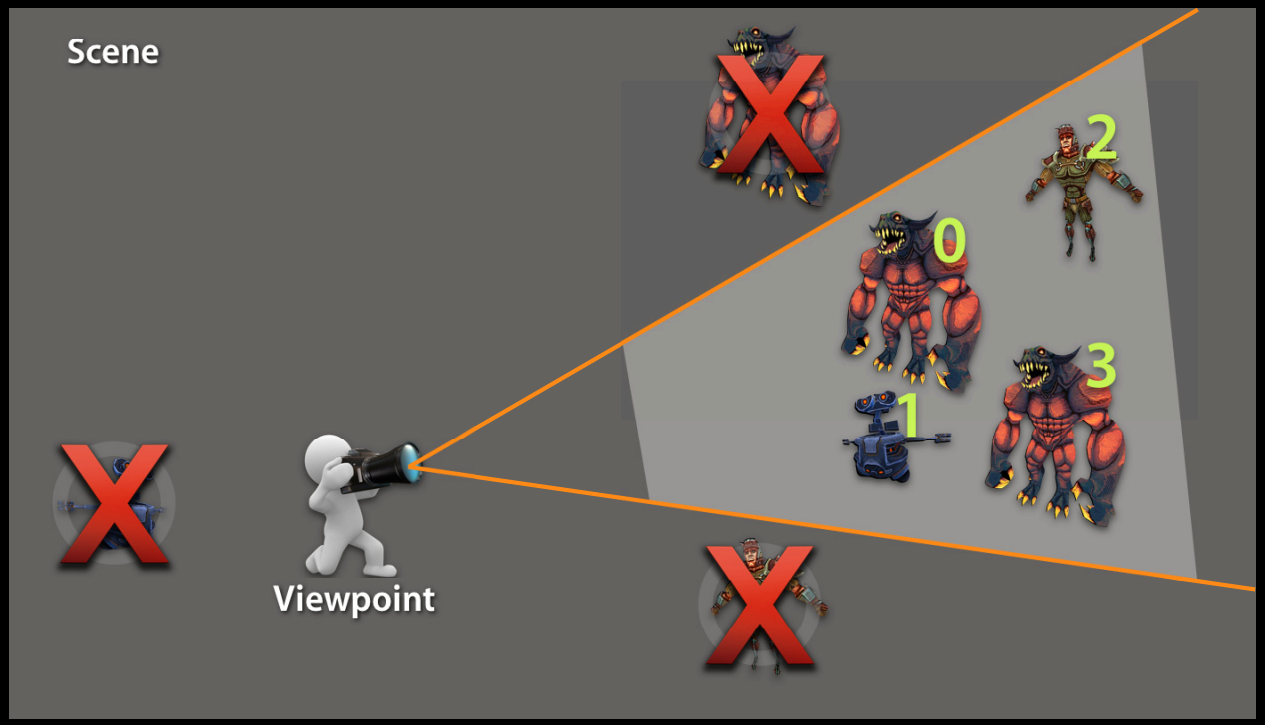
Object Culling

Visibility

- OpenGL processes everything sent to it
  - Even if ultimately not visible
- Cull non-visible objects and don't send to OpenGL

Object Culling

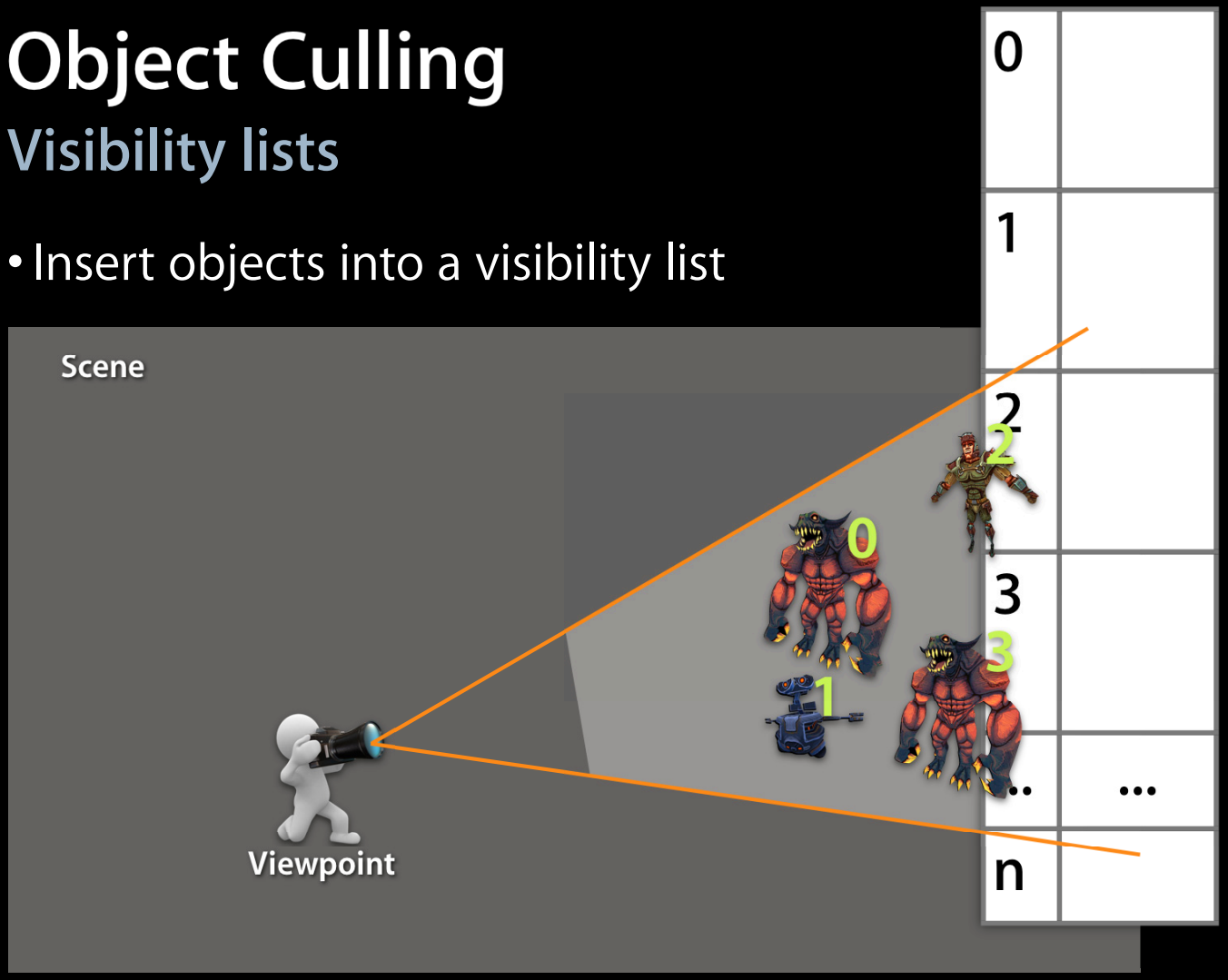
Visibility



Object Culling

Visibility lists

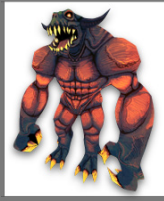
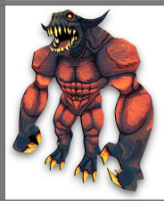
- Insert objects into a visibility list



State Sorting

Render state and visibility

- Don’t draw objects in same order they were determined visible
- Sort by render state
  - Textures, programs, vertex arrays

0	
1	
2	
3	
...	...
n	

State Sorting

State trees

State Sorting

State trees

More Expensive

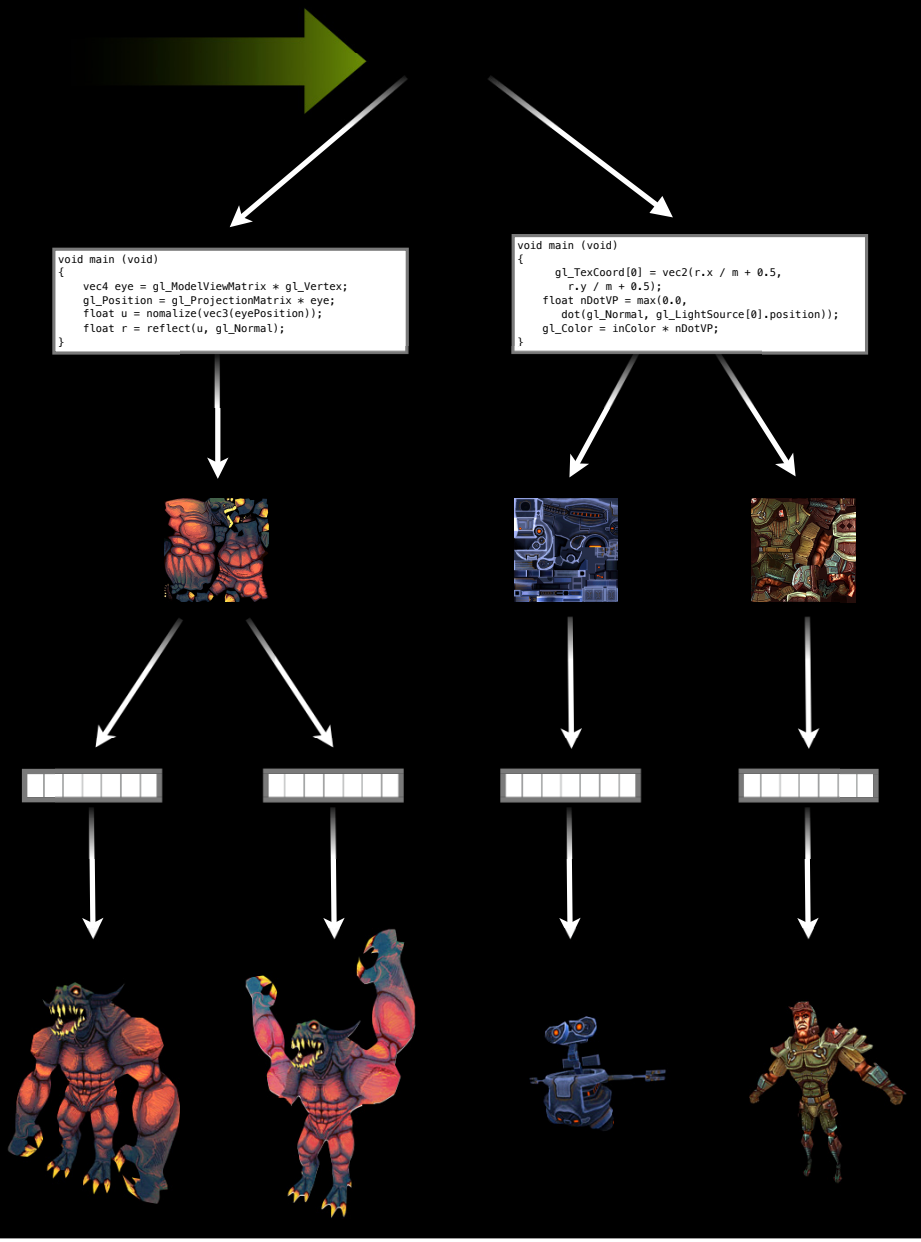
GLSL Programs

Textures

Less Expensive

Vertex Arrays

Renderable Objects



State Batching

Combined draw calls

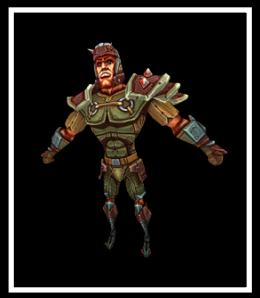
- Reduce CPU overhead made by draw calls by making less draw calls
  - Combine many renderable objects into a single draw call
- Texture atlas
  - Combine multiple textures into a single texture image
- Instancing
  - Combine vertex array data into a single VBO

State Batching

Bind overhead

State Batching

Texture atlases



State Batching

Quest texture atlas

Multithreading OpenGL

Reasons to multithread

- Makes sense to move some of the work to another thread
  - Second thread amortizes CPU cost across other cores
- Makes sense on iOS 4 devices too
  - CPU intensive calls can block
    - Won't be able to handle UI event, audio, etc.
    - Watchdog kills apps blocking main thread too long

Multithreading OpenGL

Object creation on secondary threads

- Simple multithreading technique
- Second thread loads vertex and texture data and compiles shaders
  - Kill second thread and context when loading done
    - OpenGL will take locks if shared context live
      - Increases CPU overhead
      - Can block threads

Multithreading OpenGL

Rendering on secondary thread

- Main thread produces data
  - Produces data dependent on app logic
    - Position, animation frame, visibility
  - No OpenGL context
- When producer done with data
  - Signal render thread to begin
- Render thread only consumes data
  - Has only OpenGL context
  - Calls OpenGL with produced data
- Main thread can process audio, input, other app logic in parallel
  - To balance some app logic can move to render thread

The GPU and the Display

Synchronizing with the display

- You can only render as fast as the display can refresh
- Running as fast as possible wastes power

The GPU and the Display

CADisplayLink with iOS 4

- On iOS 4, use CADisplayLink to initiate per-frame rendering
  - NSTimer arbitrary when fired with respect to display
    - Causes more latency in rendering
    - Can reduce frame rate

```
[CADisplayLink displayLinkWithTarget:self
                            selector:@selector(renderFrame:)];
```

The GPU and the Display

CVDisplayLink on Mac OS X

- On Mac, use CVDisplayLink
  - Can control app loop
    - Don’t need to render faster than display can refresh
    - Looping more than needed wastes power

```
CVDisplayLinkSetOutputCallback(displayLink,
                               &MyRenderFrameCallback,
                               self);
```

Mac OS X and iOS 4 Portability

Coding for both platforms

- OpenGL ES a subset of OpenGL
  - Coding with OpenGL ES 2.0 allows porting to Mac easier
- Things to be aware of
  - Memory and performance constraints on iOS devices
  - Different compressed formats
  - Slightly different function names
- Sample code cross compiles for both

Summary

- Minimize OpenGL's CPU overhead
  - Efficiently access the GPU
- Validation during draw where CPU overhead occurs
  - OpenGL objects cache validation
  - Minimize state changes and draw calls to reduce validation

More Information

Allan Schaffer
Graphics and Game Technologies Evangelist
aschaffer@apple.com

Documentation
OpenGL Dev Center
http://developer.apple.com/opengl

Apple Developer Forums
http://devforums.apple.com

Session Specific Information
http://developer.apple.com/wwdc/sessions/details/?id=414

Related Sessions

OpenGL ES Overview for the iPhone	Presidio Wednesday 2:00PM
OpenGL ES Shading and Advanced Rendering	Presidio Wednesday 3:15PM
OpenGL ES Tuning and Optimization	Presidio Wednesday 4:30PM
OpenGL for Mac OS X	Nob Hill Thursday 9:00AM
Taking advantage of Multiple GPUs	Nob Hill Thursday 10:15AM

Labs

OpenGL ES Lab	Graphics and Media Lab A Thursday 9:00AM
OpenGL for Mac OS X Lab	Graphics and Media Lab C Thursday 2:00PM


WWDC10