OpenGL ES Tuning and Optimization

More frames per second

Alex Kan and Jean-François Roy
GPU Software

Session Overview

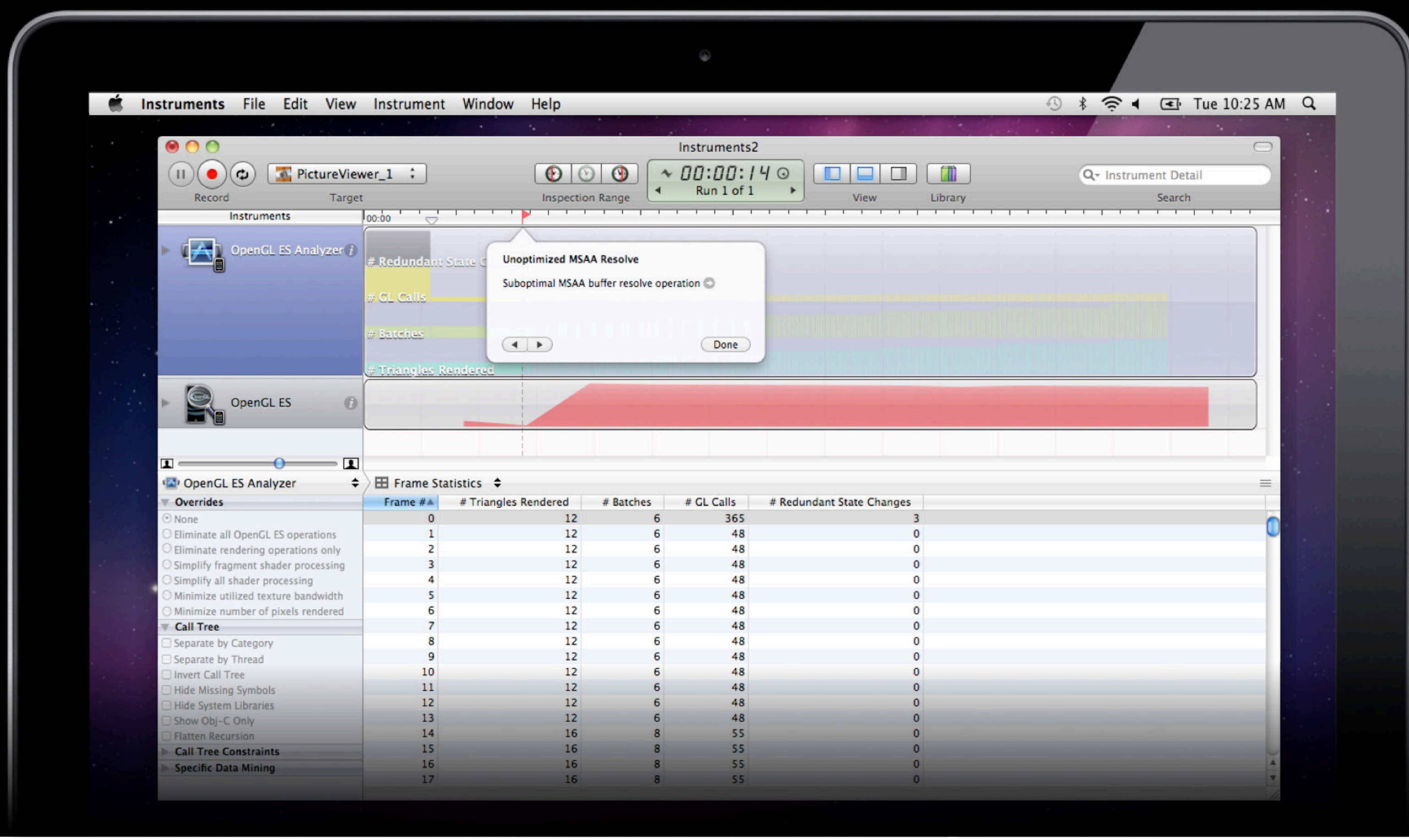
- OpenGL ES Analyzer
- Tuning the graphics pipeline
- Analyzer demo

OpenGL ES Analyzer Instrument

Developer preview

Jean-François Roy
GPU Software Developer Technologies

OpenGL ES Analyzer Instrument



Available on the
Attendees Site

OpenGL ES Analyzer Instrument

Developer preview

iOS 4

- Available as a developer preview with Xcode 4
  - Preview only supports PowerVR SGX devices with iOS 4
  - Preview cannot attach to a running application
- We want your feedback!

Why an Instrument?

OpenGL ES Analyzer Instrument

The power of many

- Correlate data from multiple instruments
- Powerful data mining
- Well-understood tool and interface

OpenGL ES Analyzer Instrument

Activity Monitor

- Traces all OpenGL ES activity
- Provides key statistics

Overrides

- Disable specific parts of the graphics pipeline
- Helpful for finding bottlenecks

?

OpenGL ES Analyzer Instrument

Activity Monitor

Overrides

OpenGL ES Analyzer Instrument

Activity Monitor

Measuring OpenGL ES Activity

Measuring OpenGL ES Activity

Understanding the problem

- Your application may be doing…
  - A lot more work than you thought
  - Work in an unexpected order
  - The wrong kind of work

Measuring OpenGL ES Activity

Activity monitor

- Records a trace of all OpenGL ES activity
- Four main hubs of information
  - Frame statistics
  - API statistics
  - Command trace
  - Call tree

Measuring OpenGL ES Activity

Frame statistics

- Get an idea of your per-frame workload
- Navigate the trace by frame
- Select range of frames in the timeline

Frame Statistics

Frame #	# Triangles Rendered	# Batches	# GL Calls	# Redundant State Changes	# Render Passes
0	25296	6	1220	32	1
1	41732	8	239	65	1
2	41732	8	218	65	1
3	41732	8	218	65	1
4	41732	8	218	69	1
5	41732	8	218	69	1
6	41732	8	218	69	1
7	41732	8	218	69	1
8	41732	8	218	69	1
9	41732	8	218	69	1
10	41732	8	218	69	1
11	41732	8	218	69	1
12	41732	8	218	69	1
13	41732	8	218	69	1
14	41732	8	218	69	1
15	41732	8	218	69	1
16	41732	8	218	69	1
17	41732	8	218	69	1
18	41732	8	218	69	1
19	41732	8	218	69	1

Frame Statistics

Primitives and batches

- Maximize ratio of primitives to batches (draw commands)
- Minimize batches
- Find your most costly frame w/r to geometry

Frame Statistics

Frame #	# Triangles Rendered	# Batches	# GL Calls	# Redundant State Changes	# Render Passes
0	25296	6	1220	32	1
1	41732	8	239	65	1
2	41732	8	218	65	1
3	41732	8	218	65	1
4	41732	8	218	69	1
5	41732	8	218	69	1
6	41732	8	218	69	1
7	41732	8	218	69	1
8	41732	8	218	69	1
9	41732	8	218	69	1
10	41732	8	218	69	1
11	41732	8	218	69	1
12	41732	8	218	69	1
13	41732	8	218	69	1
14	41732	8	218	69	1
15	41732	8	218	69	1
16	41732	8	218	69	1
17	41732	8	218	69	1
18	41732	8	218	69	1
19	41732	8	218	69	1

Frame Statistics

OpenGL commands

- Minimize how many commands you issue per frame
- Ratio of batches to GL commands should approach 1
  - If small, try sorting the geometry by state

Frame Statistics

Frame #	# Triangles Rendered	# Batches	# GL Calls	# Redundant State Changes	# Render Passes
0	25296	6	1220	32	1
1	41732	8	239	65	1
2	41732	8	218	65	1
3	41732	8	218	65	1
4	41732	8	218	69	1
5	41732	8	218	69	1
6	41732	8	218	69	1
7	41732	8	218	69	1
8	41732	8	218	69	1
9	41732	8	218	69	1
10	41732	8	218	69	1
11	41732	8	218	69	1
12	41732	8	218	69	1
13	41732	8	218	69	1
14	41732	8	218	69	1
15	41732	8	218	69	1
16	41732	8	218	69	1
17	41732	8	218	69	1
18	41732	8	218	69	1
19	41732	8	218	69	1

Frame Statistics

Redundant state changes

- How many commands were issued to change state to the same value
  - Trigger work in the driver nonetheless
- Don't do them!

Frame Statistics

Frame #	# Triangles Rendered	# Batches	# GL Calls	# Redundant State Changes	# Render Passes
0	25296	6	1220	32	1
1	41732	8	239	65	1
2	41732	8	218	65	1
3	41732	8	218	65	1
4	41732	8	218	69	1
5	41732	8	218	69	1
6	41732	8	218	69	1
7	41732	8	218	69	1
8	41732	8	218	69	1
9	41732	8	218	69	1
10	41732	8	218	69	1
11	41732	8	218	69	1
12	41732	8	218	69	1
13	41732	8	218	69	1
14	41732	8	218	69	1
15	41732	8	218	69	1
16	41732	8	218	69	1
17	41732	8	218	69	1
18	41732	8	218	69	1
19	41732	8	218	69	1

Frame Statistics

Render passes

- The graphics hardware operates in render passes
- Aim for only one render pass per frame
- Some commands can force the hardware to end the current pass

Frame Statistics

Frame #	# Triangles Rendered	# Batches	# GL Calls	# Redundant State Changes	# Render Passes
0	25296	6	1220	32	1
1	41732	8	239	65	1
2	41732	8	218	65	1
3	41732	8	218	65	1
4	41732	8	218	69	1
5	41732	8	218	69	1
6	41732	8	218	69	1
7	41732	8	218	69	1
8	41732	8	218	69	1
9	41732	8	218	69	1
10	41732	8	218	69	1
11	41732	8	218	69	1
12	41732	8	218	69	1
13	41732	8	218	69	1
14	41732	8	218	69	1
15	41732	8	218	69	1
16	41732	8	218	69	1
17	41732	8	218	69	1
18	41732	8	218	69	1
19	41732	8	218	69	1

Measuring OpenGL ES Activity
API statistics

- API statistics allow you to see what commands
  - Are used most frequently
  - Cost the most

API Statistics

OpenGL ES Function	Count	Total Time (µs)	Average Time (µs)
glClear	954	5025619	487160
glDrawElements	7622	1683719	24750
EAGLPresentRenderBuffer	953	1092747	104377
glUseProgram	10488	203210	1720
glBindBuffer	40132	188810	370
glUniformMatrix4fv	17150	155419	768
glBufferData	113	146701	1298
glCompileShader	8	134188	16773
glVertexAttribPointer	23820	115560	414
glBindTexture	23843	113148	399
glDiscardFramebufferEXT	953	100226	9458
glEnableVertexAttribArray	23820	68953	223
glUniform4f	5716	65583	1000
glActiveTexture	23821	65155	208
glUniform4fv	954	39133	3691

API Statistics

Cost in time

- Ideally, draw commands should dominate after loading
- Simple commands with can be costly when called frequently
- Minimize usage of expensive commands during gameplay or do them on another thread

API Statistics

OpenGL ES Function	Count	Total Time (µs)	Average Time (µs)
glClear	954	5025619	487160
glDrawElements	7622	1683719	24750
EAGLPresentRenderBuffer	953	1092747	104377
glUseProgram	10488	203210	1720
glBindBuffer	40132	188810	370
glUniformMatrix4fv	17150	155419	768
glBufferData	113	146701	1298
glCompileShader	8	134188	16773
glVertexAttribPointer	23820	115560	414
glBindTexture	23843	113148	399
glDiscardFramebufferEXT	953	100226	9458
glEnableVertexAttribArray	23820	68953	223
glUniform4f	5716	65583	1000
glActiveTexture	23821	65155	208

API Statistics

Cost in time

- Ideally, draw commands should dominate after loading
- Simple commands can be costly when called frequently
- Minimize usage of expensive commands during gameplay or do them on another thread

API Statistics

OpenGL ES Function	Count	Total Time (µs)	Average Time (µs)
glClear	1101	5693504	551033
EAGLPresentRenderBuffer	1100	1290681	123059
EAGLInitWithAPIProperties	1	32043	32043
glDrawElements	8800	1848703	26701
glCompileShader	8	134188	16773
glDiscardFramebufferEXT	1100	115167	10876
EAGLRenderbufferStorageFromDrawable	2	11360	5680
glLinkProgram	5	22830	4566
glUniform4fv	1101	45596	4302
glBindFramebuffer	1105	28533	2636
glCheckFramebufferStatus	2	5160	2580
glUseProgram	12107	244640	2071
glBufferData	113	146701	1298
glUniform4f	6600	80893	1237
glUniformMatrix4fv	19803	179685	886

API Statistics

Frequency

- Make sure you're not issuing commands more than you need to
  - Likely that many of them are redundant
- Use performance extensions to reduce command count

API Statistics

OpenGL ES Function	Count	Total Time (µs)	Average Time (µs)
glBindBuffer	42342	202249	397
glBindTexture	25158	125878	446
glVertexAttribPointer	25138	119847	429
glEnableVertexAttribArray	25138	74719	242
glActiveTexture	25136	67135	213
glUniformMatrix4fv	18099	164720	813
glUseProgram	11066	216278	1831
glDrawElements	8043	1742220	25440
glGetFloatv	6034	21129	284
glUniform4f	6033	68349	1040
glGetError	4050	13859	259
glEnable	4023	14420	265
glDepthMask	3016	24012	679
glBlendColor	3016	17587	551

API Statistics

Time range filtering

- API statistics are re-computed to match the selected time range
  - Use the frame statistics table to select a range of frames
  - Region of interest highlighted by other instruments

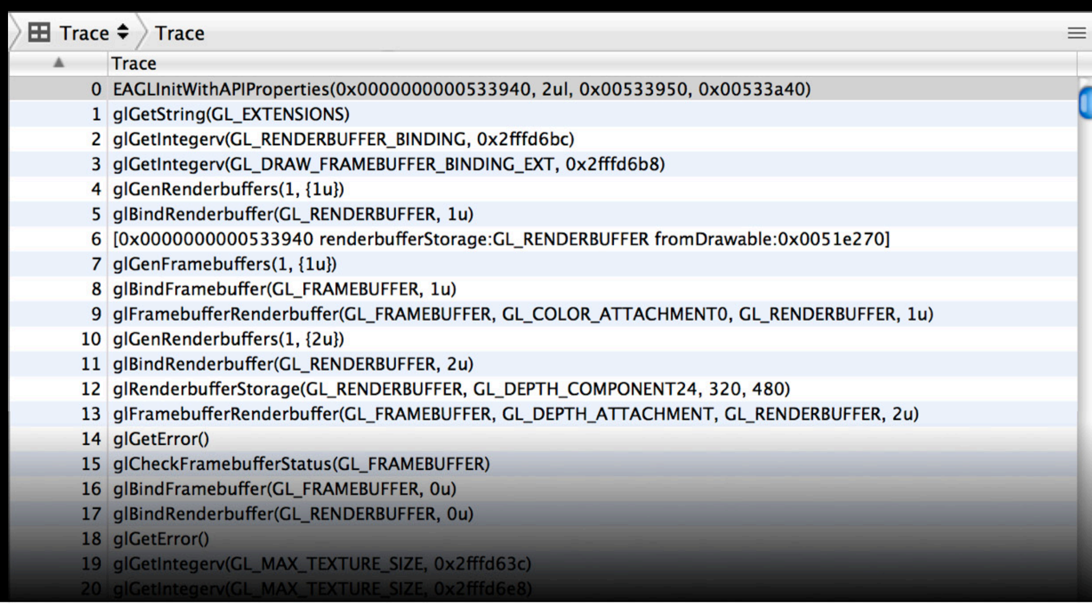
API Statistics

OpenGL ES Function	Count	Total Time (µs)	Average Time (µs)
glClear	954	5025619	487160
glDrawElements	7622	1683719	24750
EAGLPresentRenderBuffer	953	1092747	104377
glUseProgram	10488	203210	1720
glBindBuffer	40132	188810	370
glUniformMatrix4fv	17150	155419	768
glBufferData	113	146701	1298
glCompileShader	8	134188	16773
glVertexAttribPointer	23820	115560	414
glBindTexture	23843	113148	399
glDiscardFramebufferEXT	953	100226	9458
glEnableVertexAttribArray	23820	68953	223
glUniform4f	5716	65583	1000
glActiveTexture	23821	65155	208
glUniform4fv	954	39133	3691

Measuring OpenGL ES Activity

Command trace

- The complete list of every OpenGL command issued by your application
- Useful in conjunction with the other hubs



Measuring OpenGL ES Activity

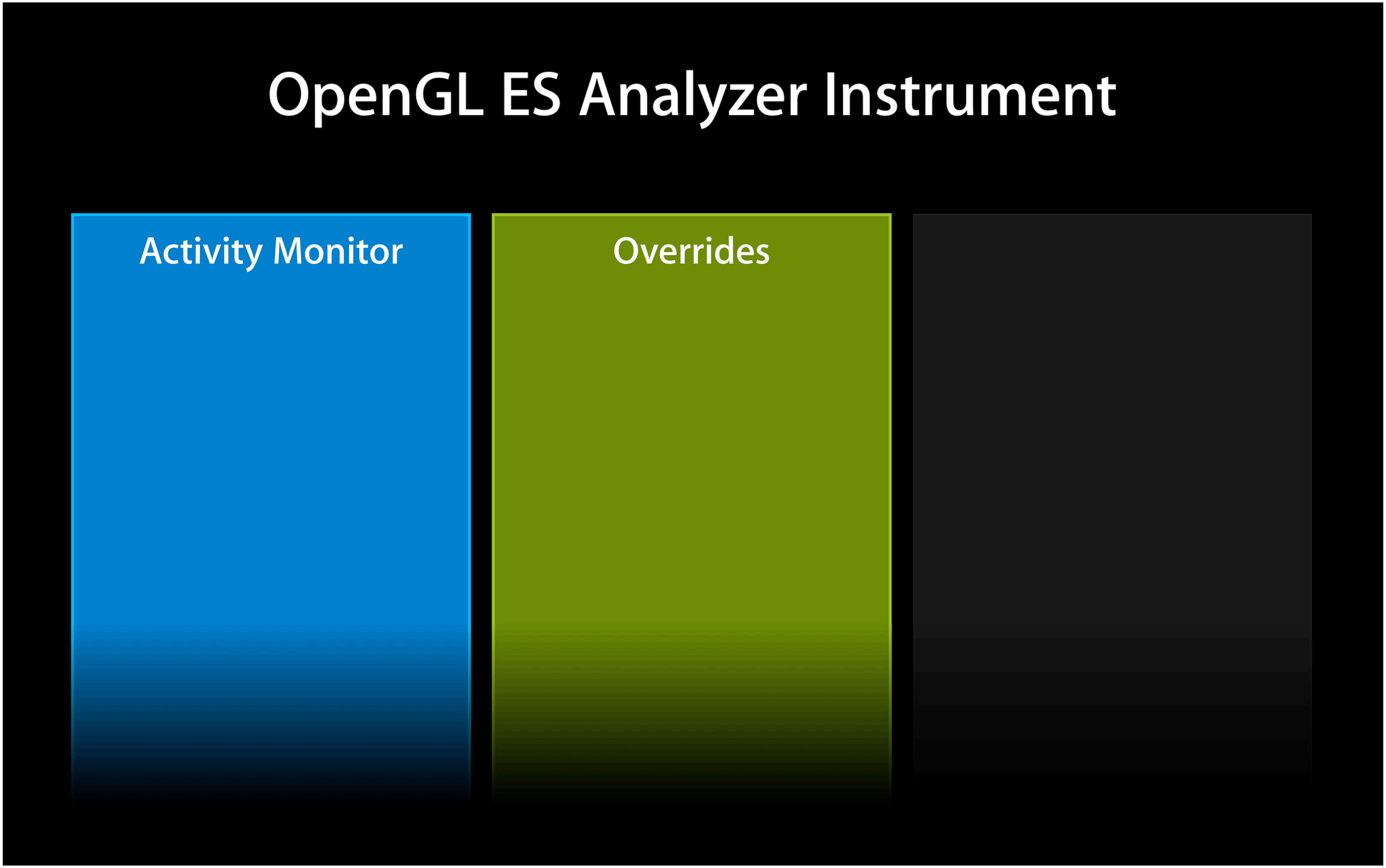
Call tree

- Standard Instruments call tree view but focused onOpenGL commands
- Allows you to see which OpenGL commands took the most amount of time and from where they were called

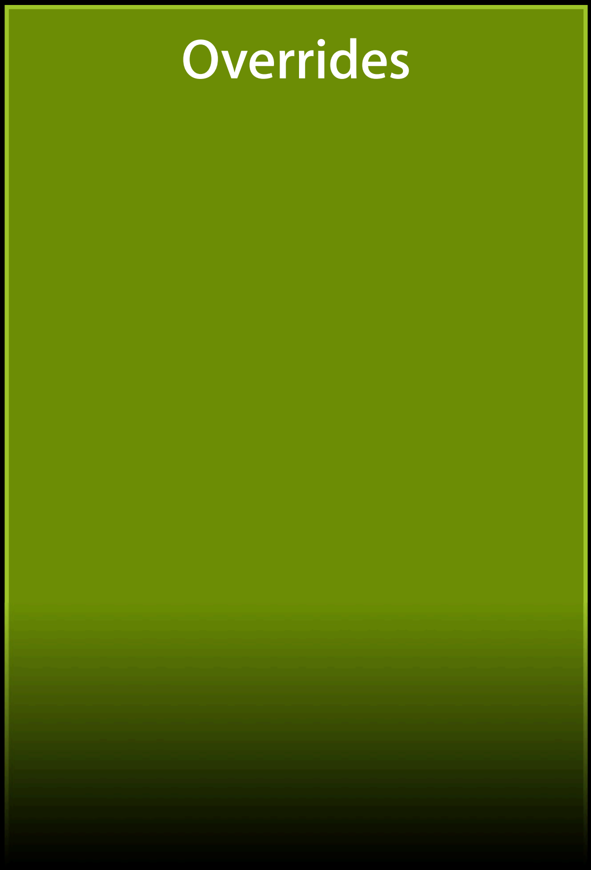
Call Trees › Call Tree

Running Time		Running Count		Symbol Name
4463.7ms	52.5%	845	0.4%	▶glClear OpenGLES
1554.9ms	18.3%	6756	3.6%	▶glDrawElements OpenGLES
962.9ms	11.3%	844	0.4%	▶CA::Display::DisplayLink::dispatch(unsigned long long, unsigned long long) QuartzCore
182.2ms	2.1%	9295	4.9%	▶glUseProgram OpenGLES
171.5ms	2.0%	35586	19.1%	▶glBindBuffer OpenGLES
146.7ms	1.7%	113	0.0%	▶glBufferData OpenGLES
140.1ms	1.6%	15203	8.1%	▶glUniformMatrix4fv OpenGLES
134.2ms	1.5%	8	0.0%	▶glCompileShader OpenGLES
105.6ms	1.2%	21116	11.3%	▶glVertexAttribPointer OpenGLES
104.2ms	1.2%	21136	11.3%	▶glBindTexture OpenGLES
91.6ms	1.0%	844	0.4%	▶glDiscardFramebufferEXT OpenGLES
63.9ms	0.7%	21116	11.3%	▶glEnableVertexAttribArray OpenGLES
60.6ms	0.7%	21114	11.3%	▶glActiveTexture OpenGLES
59.8ms	0.7%	5068	2.7%	▶glUniform4f OpenGLES
34.1ms	0.4%	845	0.4%	▶glUniform4fv OpenGLES
23.1ms	0.2%	2533	1.3%	▶glDepthMask OpenGLES
22.8ms	0.2%	5	0.0%	▶glLinkProgram OpenGLES
21.6ms	0.2%	106	0.0%	▶glCompressedTexImage2D OpenGLES

OpenGL ES Analyzer Instrument


OpenGL ES Analyzer Instrument
Activity Monitor
Overrides

OpenGL ES Analyzer Instrument



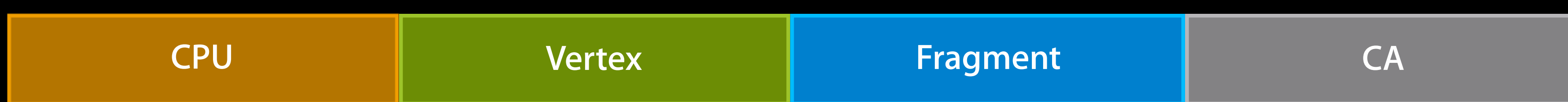
Tuning the Graphics Pipeline

Alex Kan
Embedded Graphics Acceleration

How a Frame Is Rendered

The basics

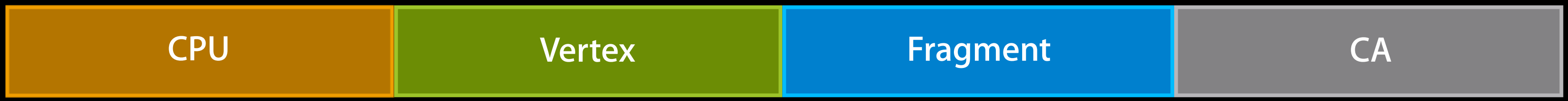
- CPU encodes rendering commands for GPU
- GPU reads and processes vertices
- GPU shades fragments for primitives
- Core Animation composites rendered results to framebuffer



How a Frame Is Rendered

Pipelining

- Not all stages take the same amount of time



How a Frame Is Rendered

Pipelining

- Not all stages take the same amount of time

How a Frame Is Rendered

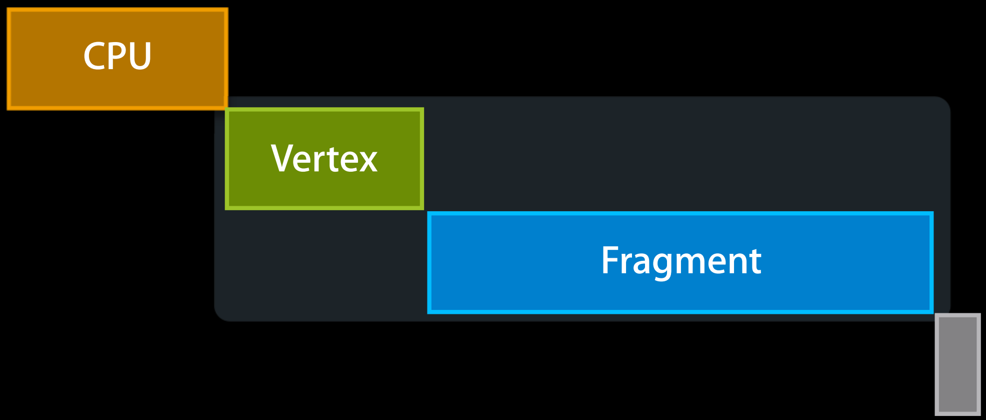
Pipelining

- Not all stages take the same amount of time
- Most stages can run in parallel with each other

How a Frame Is Rendered

Pipelining

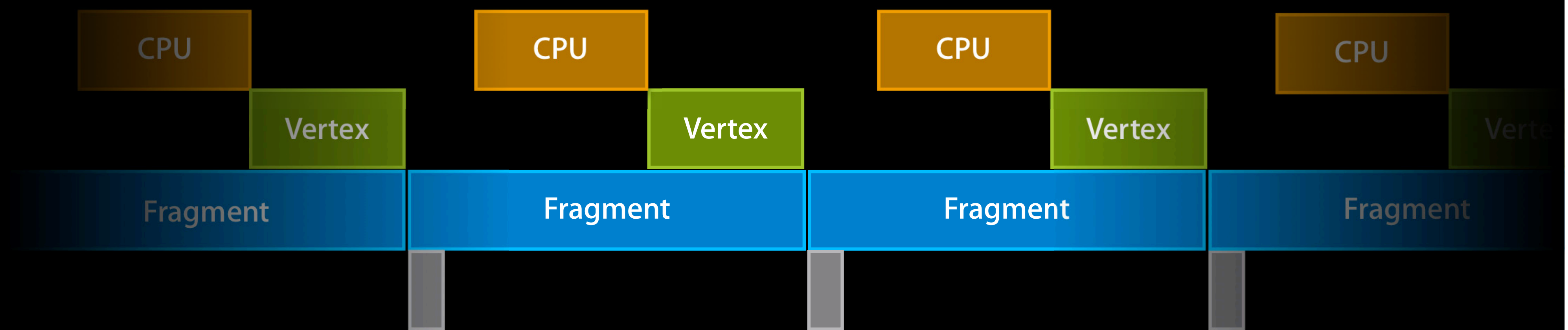
- Not all stages take the same amount of time
- Most stages can run in parallel with each other



How a Frame Is Rendered

Pipelining

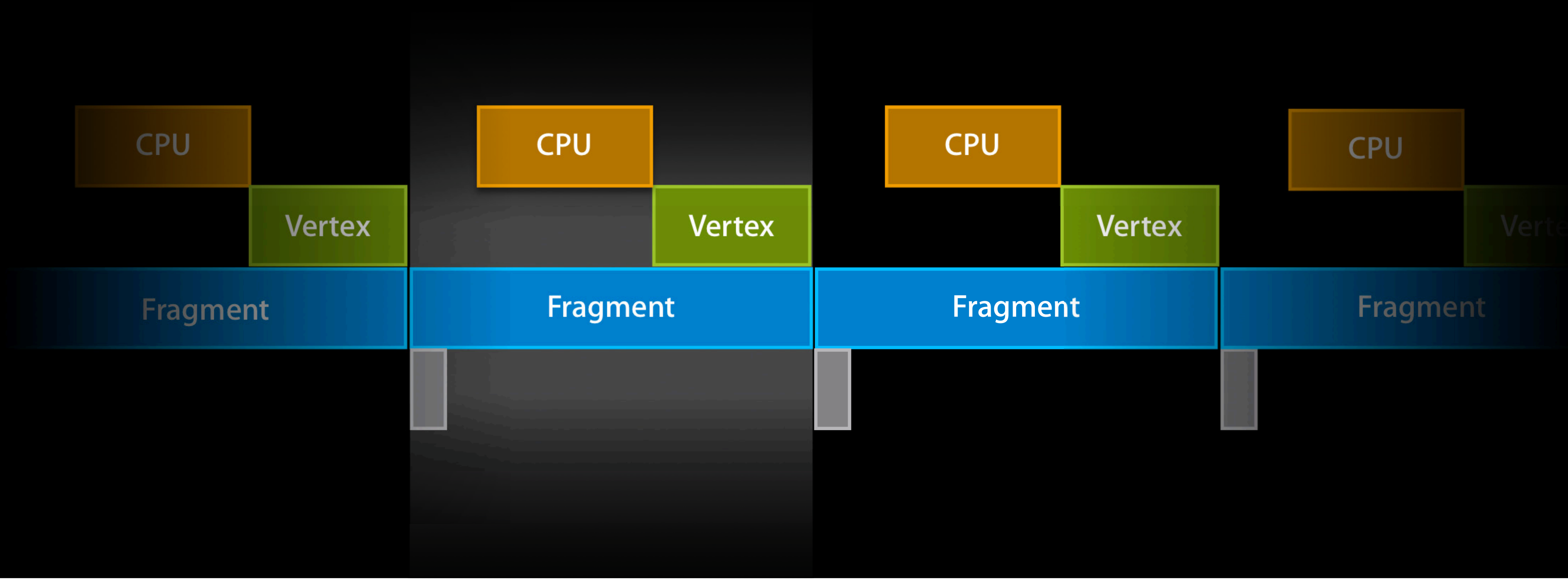
- Not all stages take the same amount of time
- Most stages can run in parallel with each other



How a Frame Is Rendered

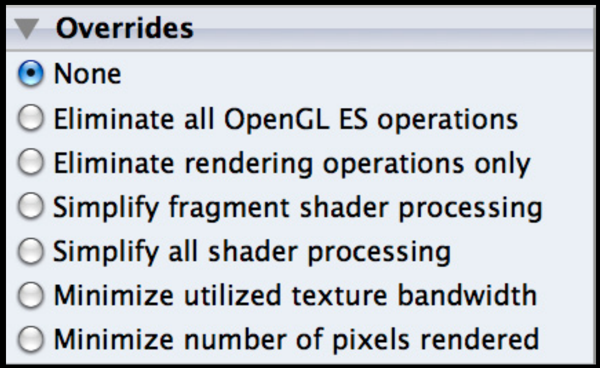
Bottlenecks

- The slowest stage determines how long your frame takes
- Optimizing the slowest stage will produce the largest gains



Finding Bottlenecks Using Overrides
Strategy

- Examine CPU/GPU utilization with CPU Sampler and OpenGL ES Driver instruments
- Apply various combinations of overrides with OpenGL ES Analyzer to see the effect on performance



Optimizing the Pipeline

Stage by stage

Optimizing the Pipeline

Topics

- CPU bottlenecks
- GPU bottlenecks
  - Bandwidth, computation, and workload size
  - Vertex bottlenecks
  - Fragment bottlenecks

CPU Bottlenecks

Identification and classification

- GPU utilization well below 100%
- Framerate unchanged when simplifying draw calls
- CPU mostly in GL framework, in either of two states:
  - Fully utilized
  - Frequently blocked

CPU Bottlenecks

High CPU utilization

- Usual culprit is handling state changes
  - Analyzer state statistics can tell you if state changes are redundant
- Some state changes are more expensive than others

Frame Statistics

Frame #	# Triangles Rendered	# Batches	# GL Calls	# Redundant State Changes	# Render Passes
0	25296	6	1220	32	1
1	41732	8	239	65	1
2	41732	8	218	65	1
3	41732	8	218	65	1
4	41732	8	218	69	1
5	41732	8	218	69	1
6	41732	8	218	69	1
7	41732	8	218	69	1
8	41732	8	218	69	1
9	41732	8	218	69	1
10	41732	8	218	69	1
11	41732	8	218	69	1
12	41732	8	218	69	1
13	41732	8	218	69	1
14	41732	8	218	69	1
15	41732	8	218	69	1
16	41732	8	218	69	1
17	41732	8	218	69	1

CPU Bottlenecks

Reducing CPU utilization

- Minimize state changes
- Take advantage of per-object state
  - Textures
  - Programs
  - Vertex arrays

CPU Bottlenecks

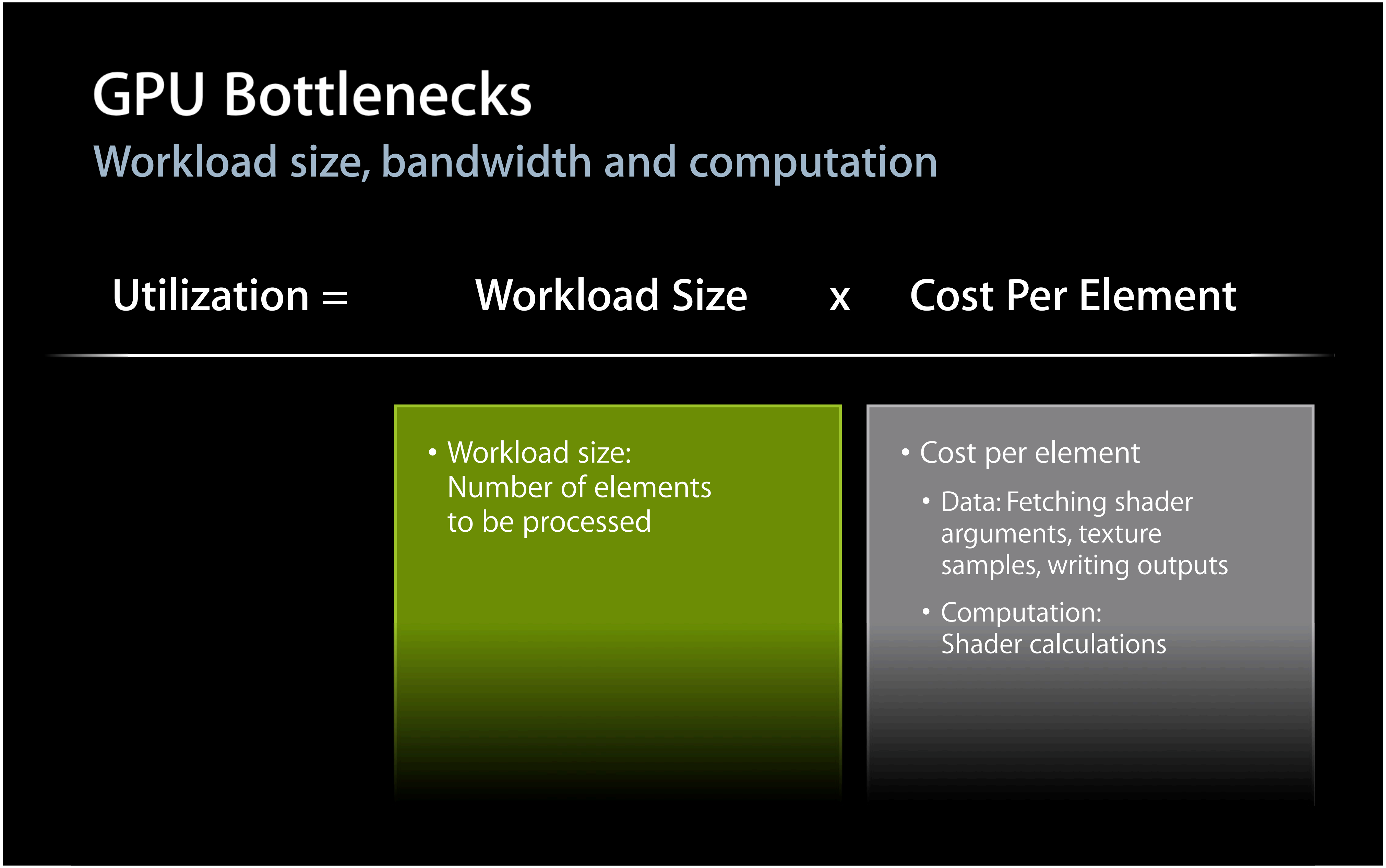
Pipeline stalls

- CPU may need to synchronize to access/modify an object in flight
  - Wait as long as possible to retrieve rendering results
  - Avoid modifying textures and VBOs after using them in a frame
  - Consider double- or triple-buffering objects if you need partial updates

CPU Bottlenecks

Summary

- Reduce state changes, particularly redundant ones
- Avoid making CPU wait for GPU
- Instruments can help you identify both these cases


GPU Bottlenecks
Workload size, bandwidth and computation
Utilization = Workload Size x Cost Per Element
• Workload size: Number of elements to be processed
• Cost per element
• Data: Fetching shader arguments, texture samples, writing outputs
• Computation: Shader calculations

Vertex Processing Bottlenecks

Identification and classification

Tiler utilization at or near 100%

- Workload size: Number of vertices
  - Frame rate increases when using simplified models

- Cost per vertex
  - Data: Fetching of vertex data
  - Computation: Vertex transformation/ lighting/shading
    - Frame rate increases when doing less calculation

Vertex Fetching Bottlenecks

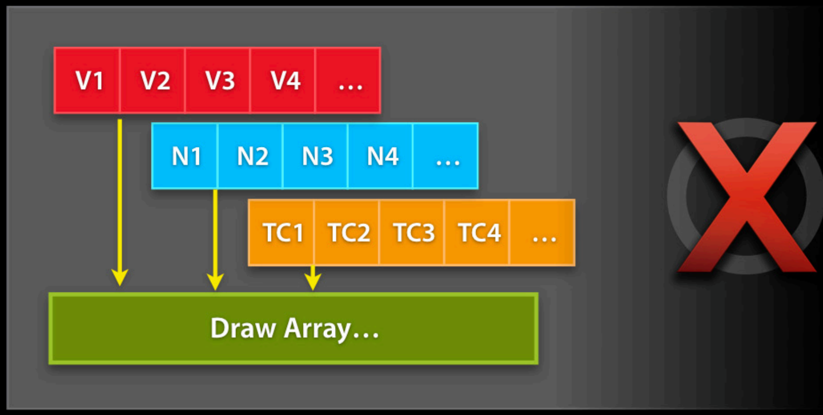
Getting vertex data to the GPU quickly

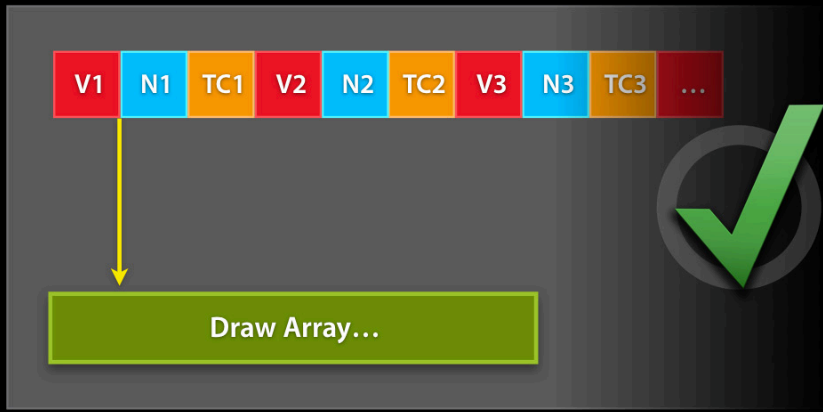
- Use vertex buffer objects (VBOs)
  - Indicate usage pattern with storage hints
    - GL_STATIC_DRAW: update once, draw repeatedly
    - GL_DYNAMIC_DRAW: update repeatedly, draw repeatedly
    - GL_STREAM_DRAW: update once, draw at most a few times
- Use indexed draw calls where possible
  - Improves performance if vertices are reused

Vertex Fetching Bottlenecks

Getting vertex data to the GPU quickly

- Interleave your vertex attributes
  - Align vertex attributes and strides to 4-byte boundaries





Vertex Processing Bottlenecks

Summary

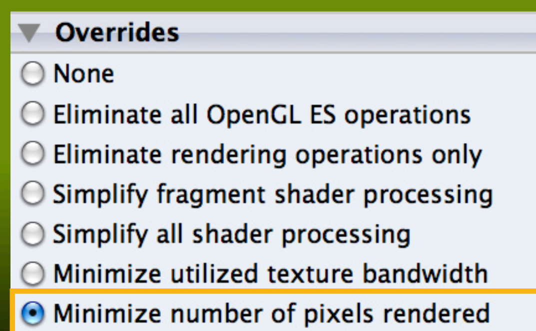
- Take advantage of VBOs and VAOs
- Size/structure your vertex data for efficient data transfer

Fragment Processing Bottlenecks

Identification and classification

Renderer utilization at or near 100%

- Workload size: Number of visible fragments



- Cost per fragment
  - Data: Framebuffer and texture bandwidth
  - Computation: Fragment shading

Processing Fewer Pixels

Dealing with overdraw

- Hidden surface removal operates on groups of opaque objects
  - Maximum efficiency by drawing opaque objects together first
- Sort order:
  - Draw all opaque objects
  - Draw any objects using “discard” keyword
  - Draw all alpha blended objects

Processing Fewer Pixels

Overdraw and blending/discard

- Blending affects every pixel in a quad
  - Even transparent ones
- Alpha-testing is generally expensive on embedded hardware
- Wasted pixels can add up as number of layers increase

Processing Fewer Pixels

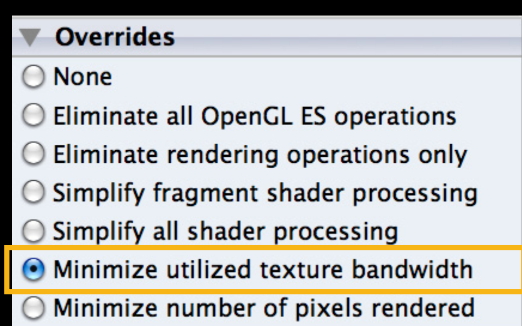
Sprite trimming

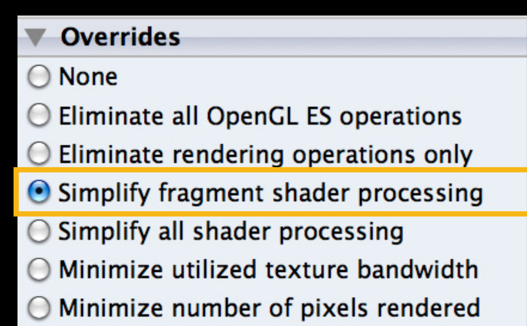
- Reduce area by using a shape that tightly encloses sprite
- Trades smaller shape for extra vertex processing

Processing Pixels Faster

What's my limit?

- Bandwidth
  - Performance increases with smaller textures/lower bit depth
- Computation
  - Performance increases with simplified fragment shader
- Analyzer has overrides for both of these situations



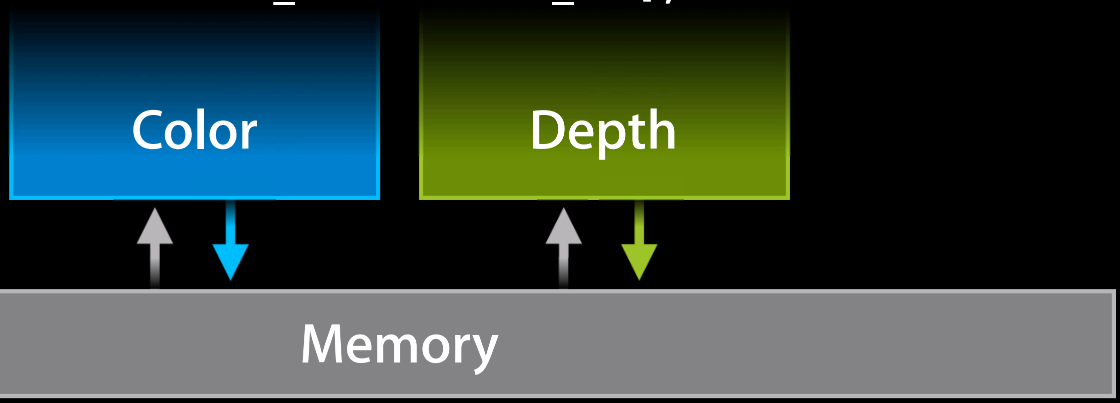


Bandwidth-Bound Fragment Processing

Reducing framebuffer bandwidth

- If you don't reuse your buffer contents from frame to frame:
  - Do a full-screen clear of all buffers at the start of the frame
  - Discard non-color buffers at the end of the frame

```
GLenum attachments[] = { GL_DEPTH_ATTACHMENT };
glDiscardFramebufferEXT(GL_READ_FRAMEBUFFER_APPLE, 1, attachments);

glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

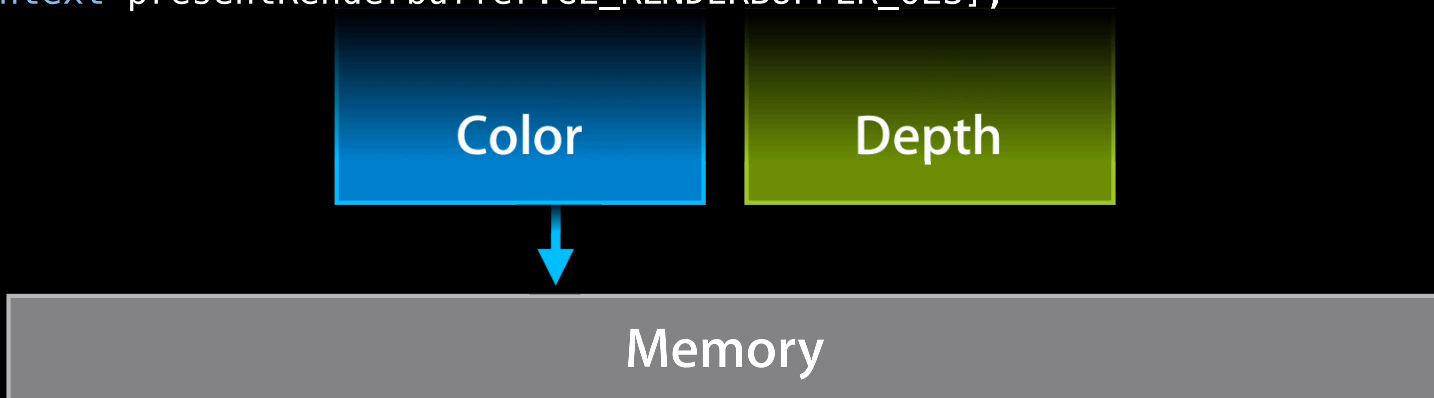


Bandwidth-Bound Fragment Processing

Reducing framebuffer bandwidth

- If you don't reuse your buffer contents from frame to frame:
  - Do a full-screen clear of all buffers at the start of the frame
  - Discard non-color buffers at the end of the frame

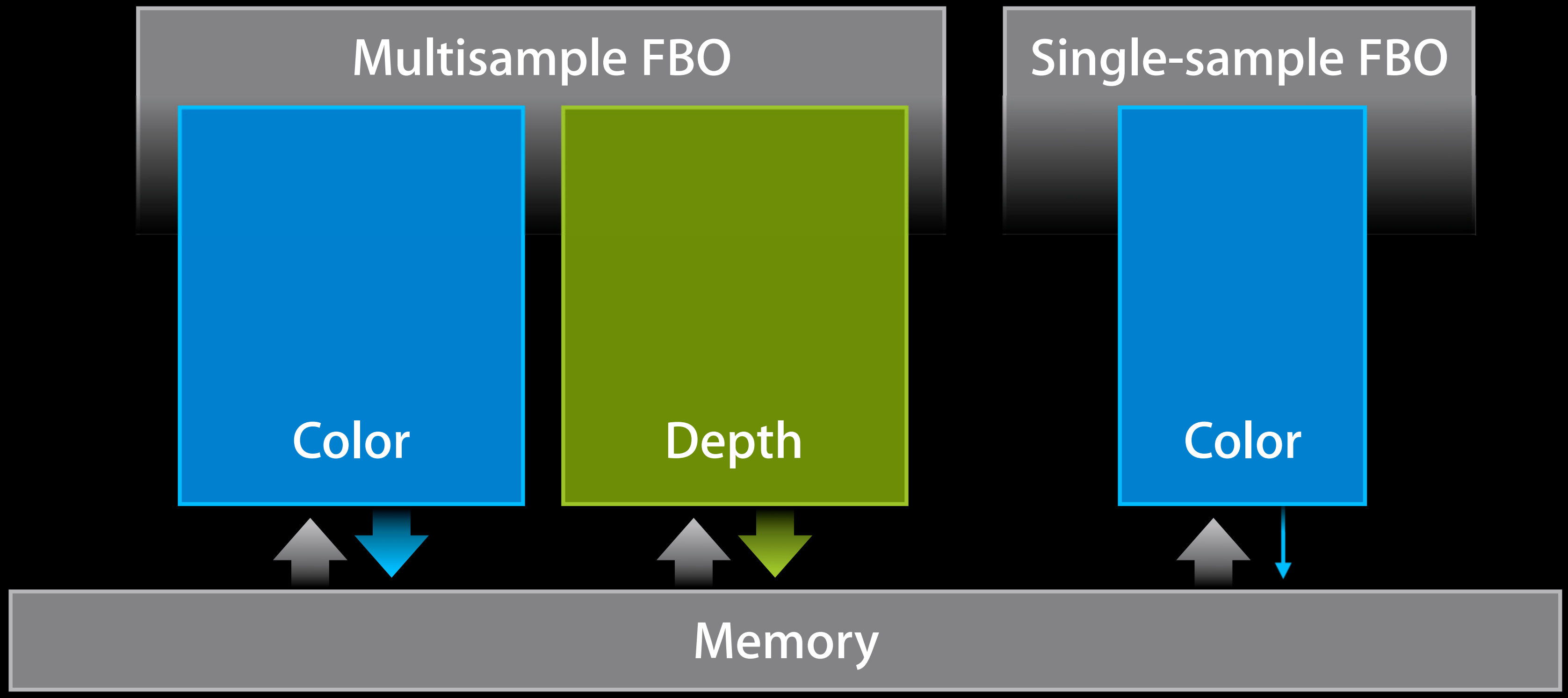
```
GLenum attachments[] = { GL_DEPTH_ATTACHMENT };
glDiscardFramebufferEXT(GL_READ_FRAMEBUFFER_APPLE, 1, attachments);

glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```



Bandwidth-Bound Fragment Processing

Reducing framebuffer bandwidth

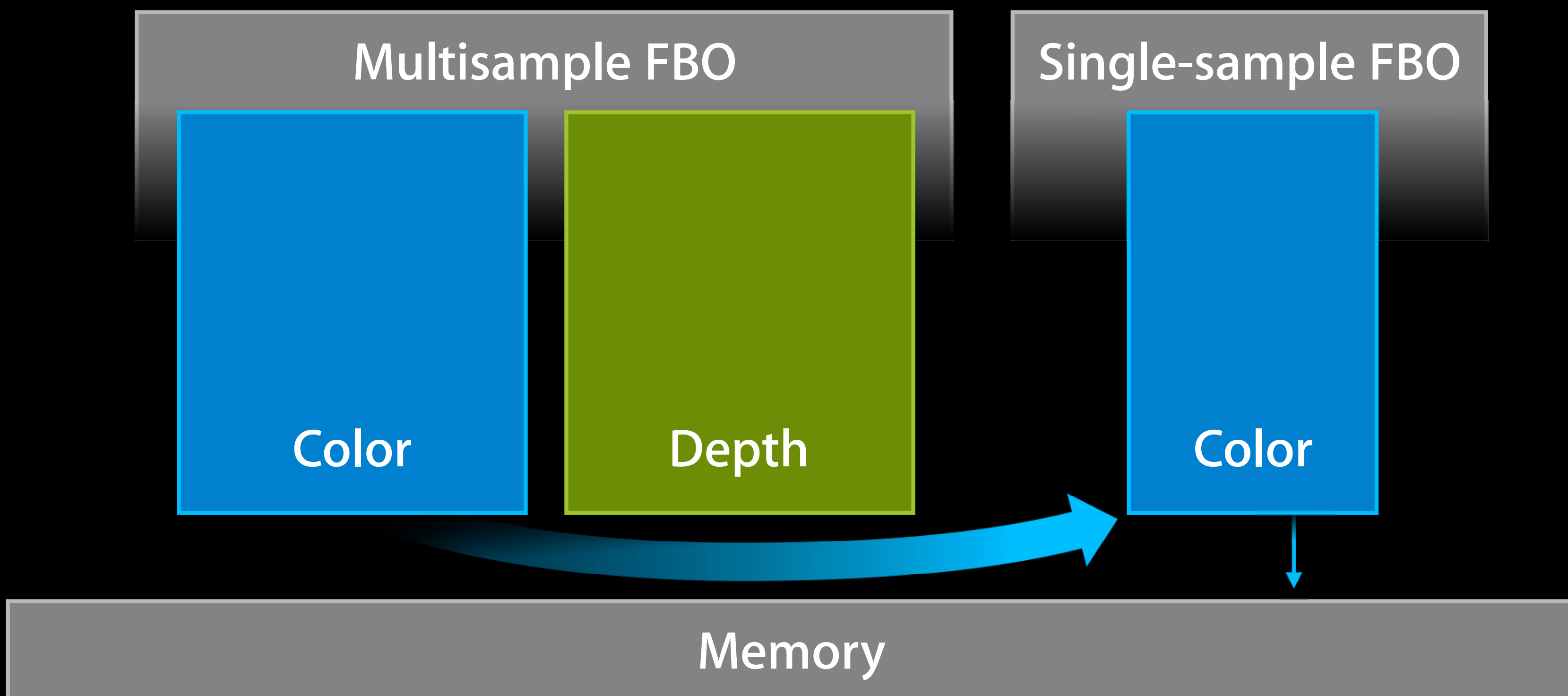
- Even more important when performing multisample rendering



Bandwidth-Bound Fragment Processing

Reducing framebuffer bandwidth

- Even more important when performing multisample rendering



Bandwidth-Bound Fragment Processing

Reducing texture bandwidth

- Use the smallest texture format and type suitable for your assets
  - PVRTC, if your content is suitable
  - Single-channel luminance, alpha
  - 16-bit formats: RGBA4444, RGBA5551, RGB565
- Size your textures appropriately for display
  - Use mipmaps if your textures will be drawn at many different scales

Fragment Processing Bottlenecks

Summary

- Give the GPU the smallest number of pixels to deal with
  - Take advantage of GPU's ability to cull hidden surfaces
- Minimize amount of external bandwidth consumed by GPU

Shader Tuning for GLSL ES

Shader Tuning for GLSL ES

Topics

- Precision qualifiers
- Minimizing computation
- Dependent texture reads

Care and Feeding of Shader Precisions

Introduction to shader precisions

- Specific to GLSL ES, not in desktop GLSL
- Varying precisions can differ between vertex and fragment stages
- Appropriate precision choices are important for good performance

Care and Feeding of Shader Precisions

highp

- Single-precision floating point
  - Use for vertex positions and texture coordinate calculations
  - Good for texture coordinates in general

```
uniform highp mat4 modelviewProjection;
uniform highp mat3 mapTexMatrix;
attribute highp vec3 position;
varying highp vec2 mapTexCoord;
```

```
mapTexCoord = (mapTexMatrix * position).st;
gl_Position = modelviewProjection * vec4(position, 1.0);
```

Care and Feeding of Shader Precisions

mediump

- Half-precision floating point
  - Potentially higher throughput
- Good for texture coordinate varyings if:
  - Texture size < 512 × 512
  - Minimal wrapping/perspective

```
mediump float ndotl = max(dot(normal, objectLightDirection), 0.0);
mediump float hdotn = sign(ndotl) * dot(normal, halfAngle);
mediump litColor = ambientColor + diffuseColor * ndotl;
litColor += specularColor * pow(hdotn, specularExponent);
```

Care and Feeding of Shader Precisions

lowp

- [-2, +2] range, 8-bit fractional precision
  - Good for color, normals, and color mixing factors
- Use 3- and 4- component vectors
- Don't swizzle components

```
uniform lowp sampler2D tex;
varying lowp vec3 litColor;

lowp vec3 modulatedColor = texture2D(tex, coord).rgb * litColor;
gl_FragColor = vec4(modulatedColor, 1.0);
```

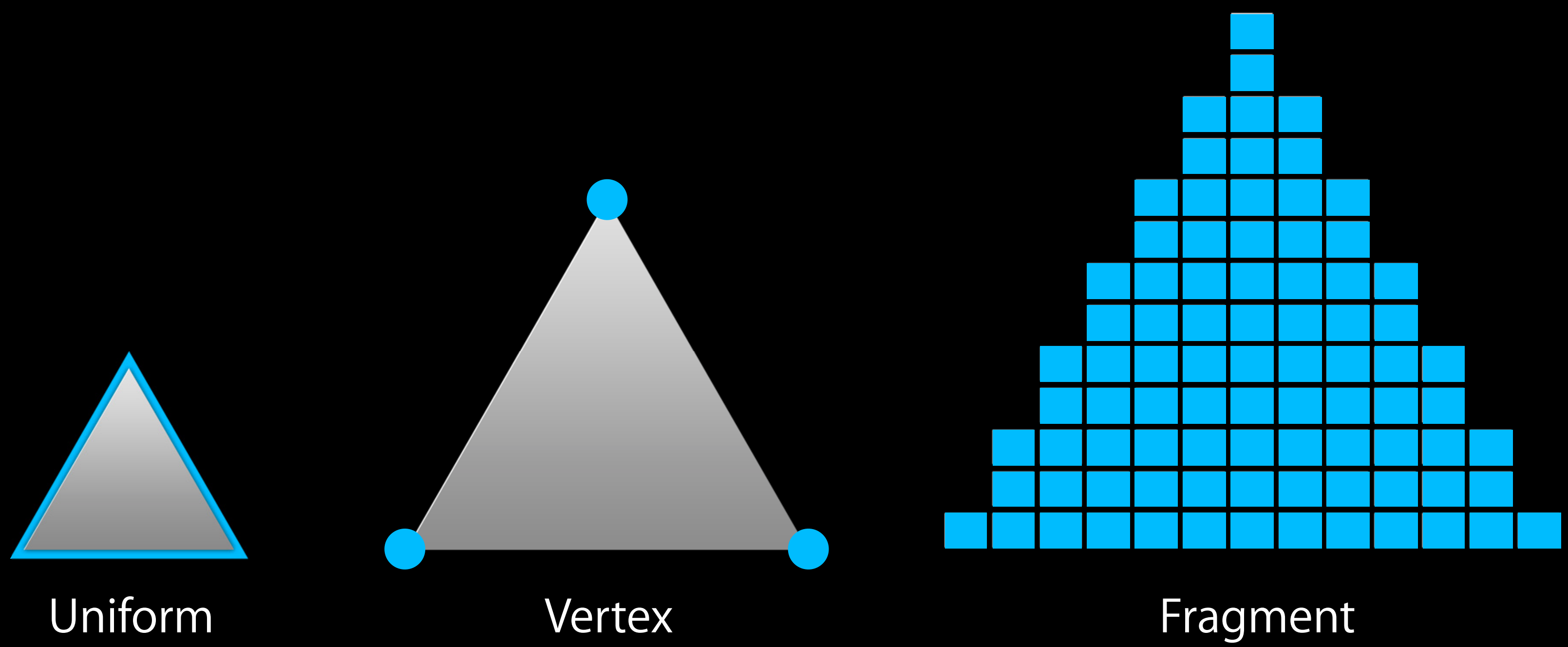
Care and Feeding of Shader Precisions

Choosing and using precisions

- Compiler is free to promote calculations to a higher precision
  - Keep in mind minimum required precision/range of type
  - Check implementation-specific precision and range
- Minimize conversions between precisions in calculations
  - Especially for conversions to/from lowp

Efficient Computation

Hoisting computation



Efficient Computation

Expressing operations

- Operate only on the elements that you need
  - Don't coerce expressions into being vectors
- When mixing scalars and vectors, keep scalars together

```
uniform vec3 attenFactor;

mediump float ndotl = max(dot(normal, objectLightDirection), 0.0);
mediump float attenuation = attenFactor.x + attenFactor.y * objectDistance
+ attenFactor.z * (objectDistance * objectDistance);
litColor = ambientColor + diffuseColor * (ndotl / attenuation);
```

Efficient Computation

Expressing operations

- Operate only on the elements that you need
  - Don't coerce expressions into being vectors
- When mixing scalars and vectors, keep scalars together

```
uniform vec3 attenFactor;

mediump float ndotl = max(dot(normal, objectLightDirection), 0.0);
mediump float attenuation = attenFactor.x + attenFactor.y * objectDistance
+ attenFactor.z * (objectDistance * objectDistance);
litColor = ambientColor + diffuseColor * (ndotl / attenuation);
```

Efficient Computation

Expressing operations

- Operate only on the elements that you need
  - Don't coerce expressions into being vectors
- When mixing scalars and vectors, keep scalars together

```
uniform vec3 attenFactor;

mediump float ndotl = max(dot(normal, objectLightDirection), 0.0);
mediump float attenuation = attenFactor.x + attenFactor.y * objectDistance
+ attenFactor.z * (objectDistance * objectDistance);
litColor = ambientColor + diffuseColor * (ndotl / attenuation);
```

Efficient Computation

GLSL built-ins

- Convenience functions for common functionality
- Convenient for hardware and shader compiler as well

```
lowp vec3 clean = texture2D(cleanTexture, coord).rgb;

lowp vec3 dirty = texture2D(dirtyTexture, coord).rgb;
lowp float dirtiness = texture2D(dirtMap, coord).a;
```

```
// BAD
lowp vec3 mixed = (1.0 - dirtiness) * clean + dirtiness * dirty;
lowp vec3 mixed = clean + (dirty - clean) * dirtiness;
```

```
// BETTER
lowp vec3 mixed = mix(clean, dirty, dirtiness);
```

Dependent Texture Reads

What are they?

- Dependent texture reads
  - Texture samples from coordinates modified in shader
- Non-dependent texture reads
  - Texture samples from coordinates passed directly from varyings

Dependent Texture Reads

What causes them?

- Explicit texture coordinate modification in fragment shader

```
uniform lowp sampler2D tex;

varying highp vec2 coord;
varying highp vec2 warpTime;

lowp vec4 nonHoistableSample = texture2D(tex, coord + sin(warpTime));
lowp vec4 hoistableSample = texture2D(tex, coord + vec2(0.5, -0.5));
```

Dependent Texture Reads

What causes them?

- Projective or biased texture samples (on PowerVR SGX)

```
uniform lowp sampler2D tex;
varying highp vec3 coord;

// projection
lowp vec4 proj = texture2DProj(tex, coord);

// LOD bias
lowp vec4 bias = texture2D(tex, coord.st, bias);

// LOD selection
lowp vec4 lod = texture2DLod(tex, coord.st, lod);
```

Dependent Texture Reads

Why does this matter?

- Non-dependent reads are typically faster
  - Fewer shader cycles
  - Better parallelism

Shader Tuning

Summary

- Choose your precisions carefully
- Make sure your calculations are:
  - Performed in the appropriate stage
    - Including texture coordinate calculations
  - Expressed efficiently

Summary

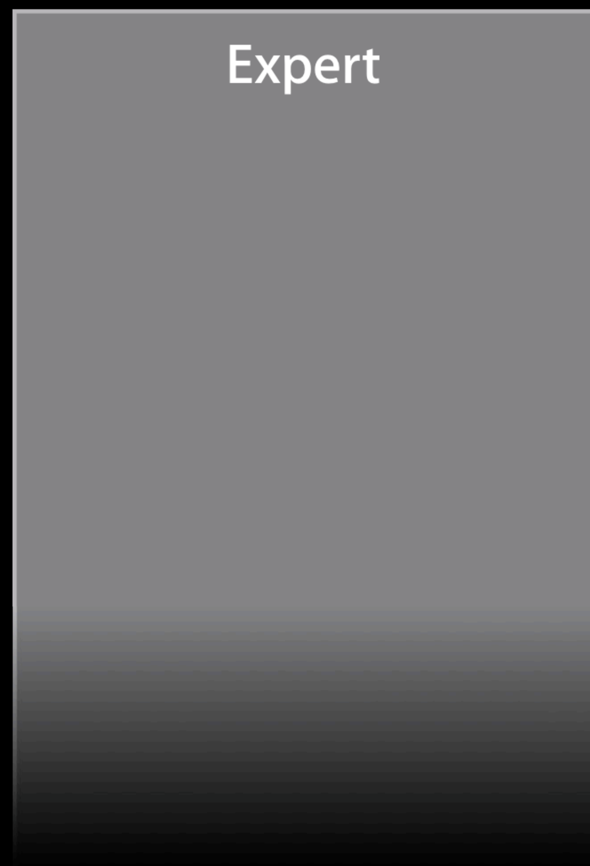
Rules of thumb

- Target the slowest pipeline stage
  - Let the tools tell you where to tune
  - Don't be afraid to do more work in other stages
- Do less
  - Take advantage of GPU hidden-surface removal
  - Use smaller data types
  - Minimize computation

OpenGL ES Analyzer Instrument

Part deux

OpenGL ES Analyzer Instrument



OpenGL ES Expert

It knows a thing or two

- Comprehensive expert system
- Knows the hardware and the software
- Finds problems in your app
  - …and provides recommendations on how to address each of them

OpenGL ES Expert

Categories of problems

- Redundant state changes
- Invalid framebuffer configurations
- Invalid texture configurations
- Invalid OpenGL operations
- Sub-optimal vertex data formats, layouts and storage
- Sub-optimal operation order
- Hardware-specific performance events

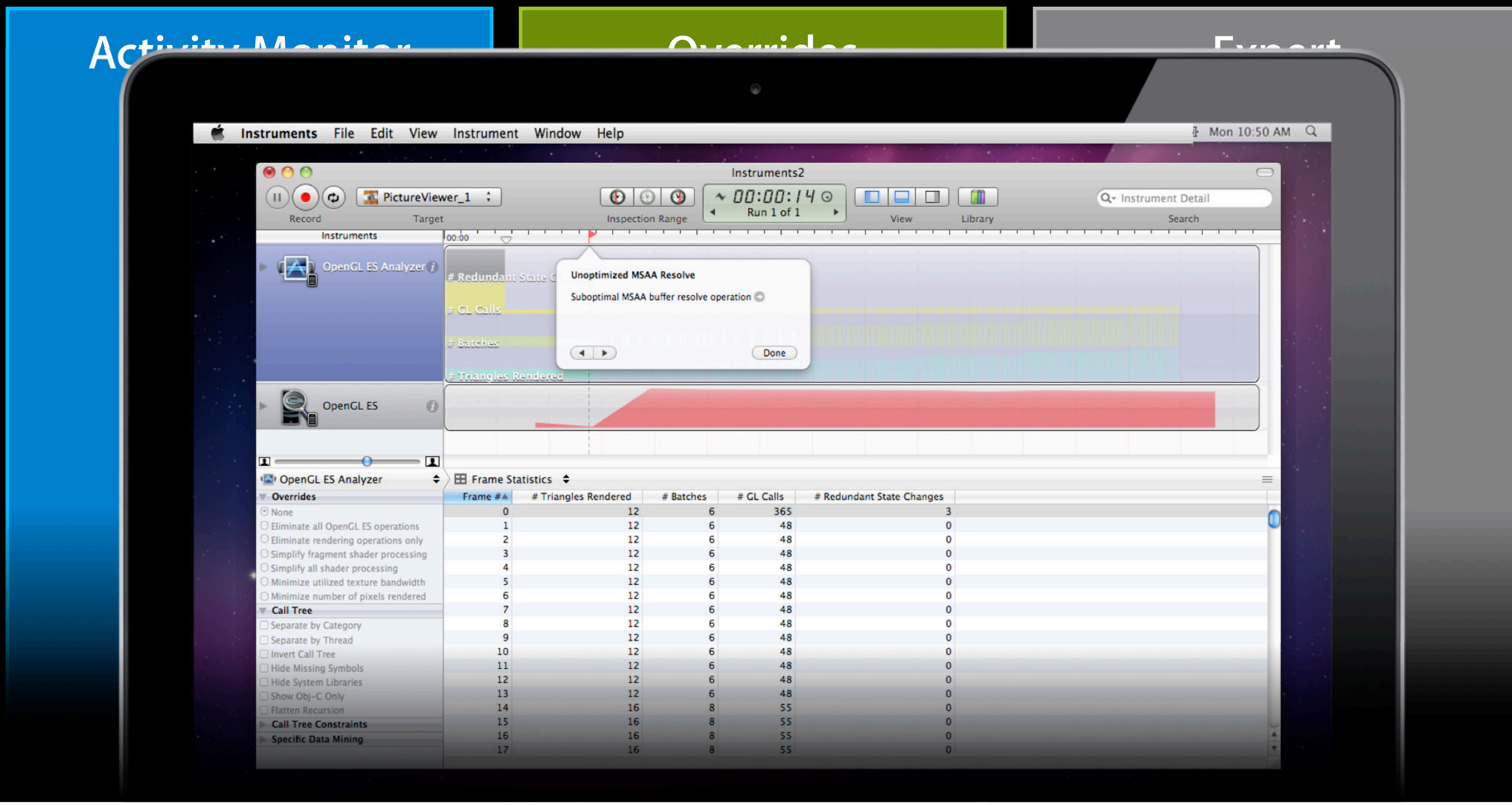
Demo

PictureViewer, analyzed

OpenGL ES Analyzer Instrument

Expert

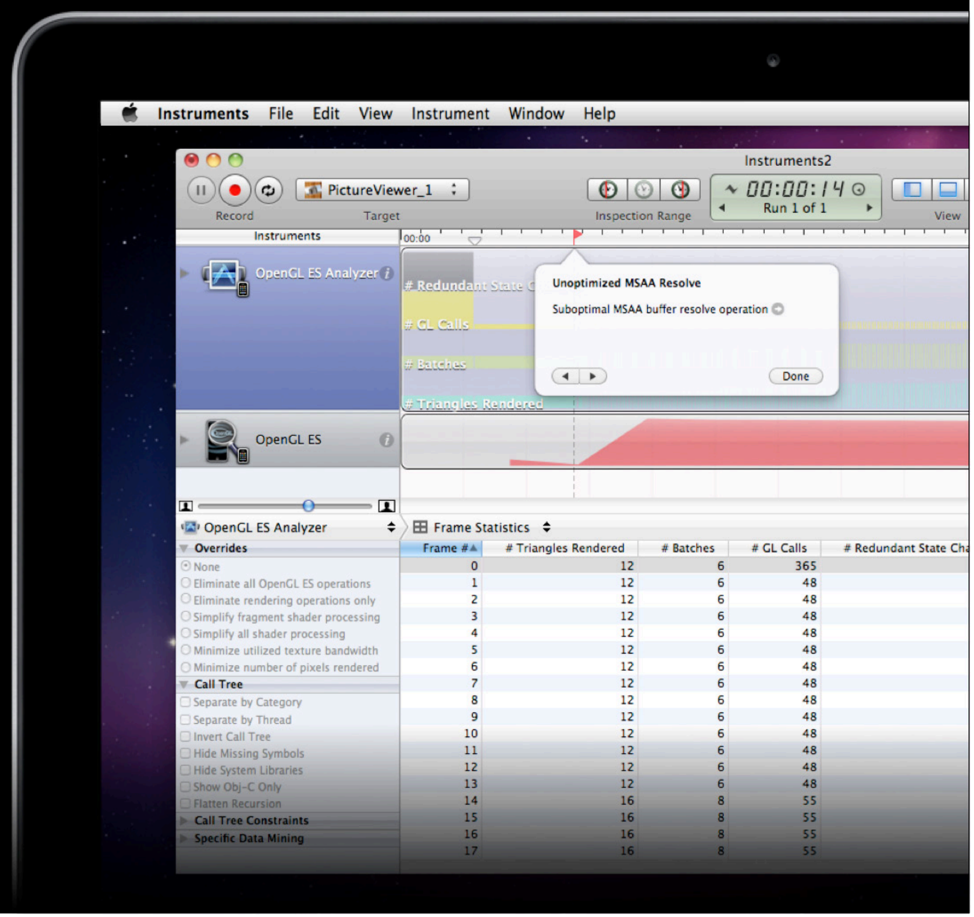
OpenGL ES Analyzer Instrument



OpenGL ES Analyzer Instrument

A call to arms

- Go get the tool
- Improve your performance
- Send us feedback
- Make an awesome app!



More Information

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

Mike Jurewitz
Developer Tools and Performance Evangelist
jurewitz@apple.com

Documentation
OpenGL ES Programming Guide for iPhone OS
http://developer.apple.com/iphone

Khronos
http://www.khronos.org

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

Related Sessions

Advanced Performance Analysis with Instruments	Mission Thursday 9:00AM
OpenGL Essential Design Practices	Pacific Heights Wednesday 11:30AM
OpenGL ES Overview for iPhone OS	Presidio Wednesday 2:00PM

Labs

OpenGL ES Lab	Graphics and Media Lab A Thursday 9:00AM to 4:15PM

WWDC10

[redacted]