



nVIDIA®

BRINGING UNREAL ENGINE 4 TO OPENGL

Nick Penwarden - Epic Games

Mathias Schott, Evan Hart - NVIDIA

| March 20, 2014

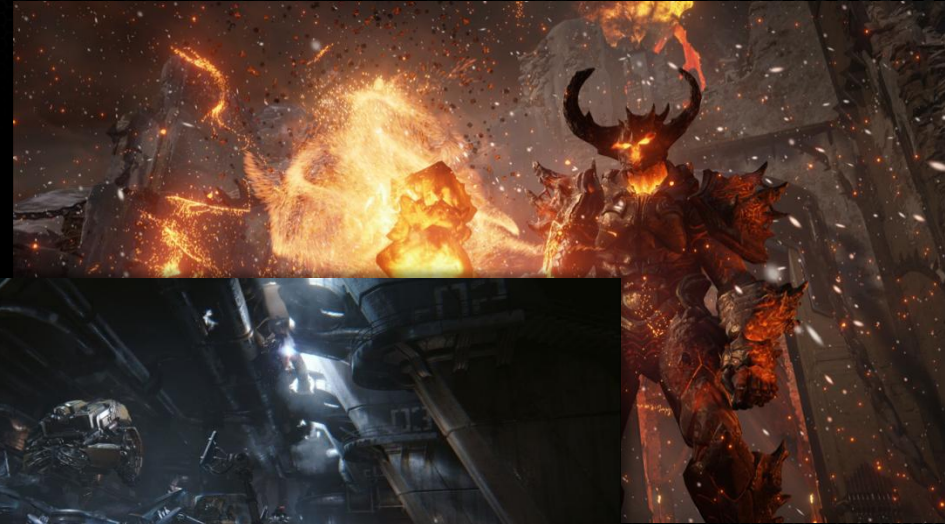


About Us

- Nick Penwarden
 - Lead Graphics Programmer - Epic Games
 - Twitter: @nickpwd
- Mathias Schott
 - Developer Technology Engineer - NVIDIA
- Evan Hart
 - Principal Engineer - NVIDIA

Unreal Engine 4

- Latest version of the highly successful UnrealEngine
 - Long, storied history
 - Dozens of Licensees
 - Hundreds of titles
- Built to scale
 - Mobile phones and tablets
 - Laptop computers
 - Next-gen Console
 - High-end PC



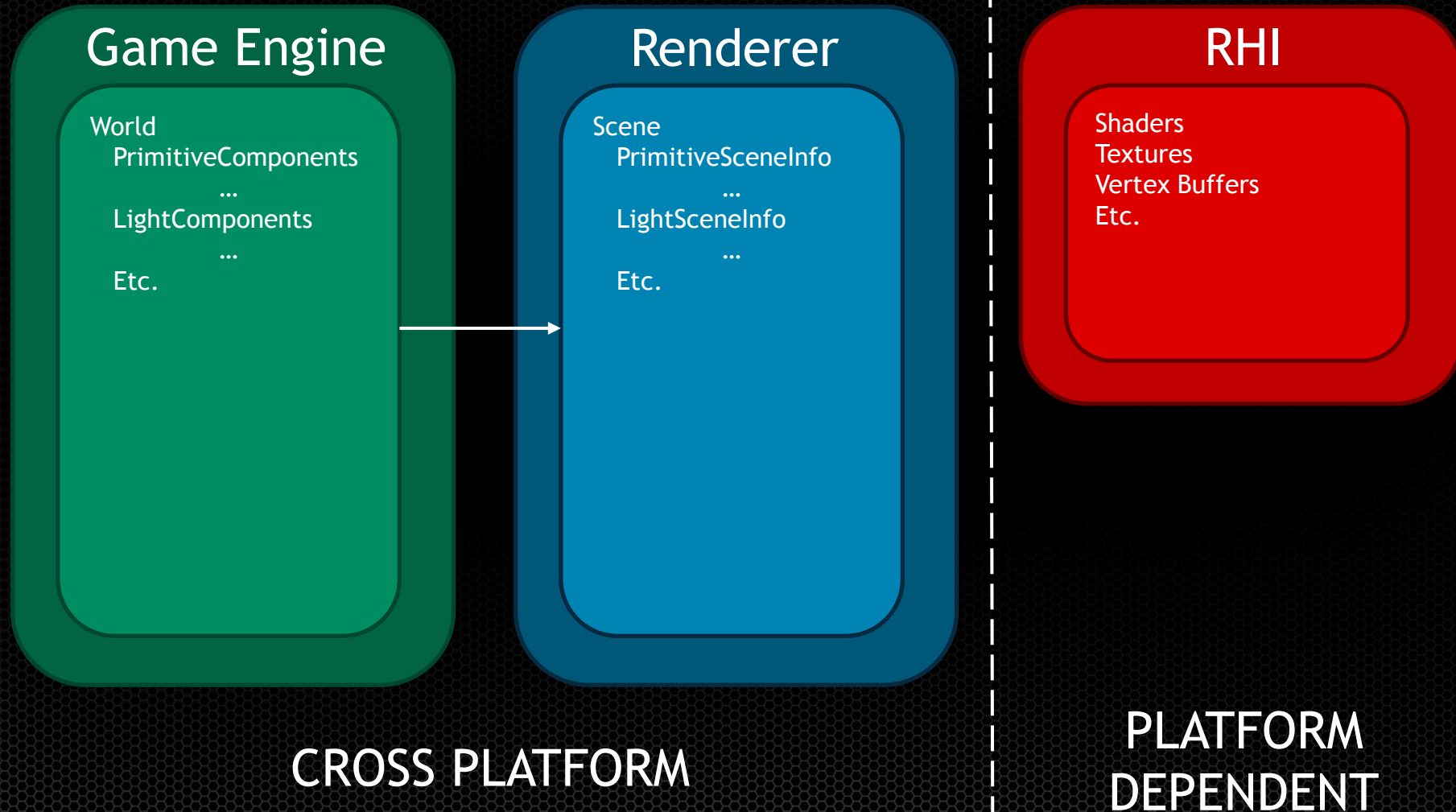
Why OpenGL?

- The only choice on some platforms:
 - Mac OS X
 - iOS
 - Android
 - Linux
 - HTML5/WebGL
- Access to new hardware features not tied to OS version.
 - Microsoft has been tying new versions of Direct3D to new versions of Windows.
 - OpenGL frees us of these restrictions: D3D 11.x features on Windows XP!

Porting UE4 to OpenGL

- UE4 is a cross platform game engine.
- We run on many platforms today (8 by my count!) with more to come.
- Extremely important that we can develop features once and have them work across all platforms!

(Very) High Level Look at UE4



Render Hardware Interface

- Largely based on the D3D11 API
- Resource management
 - Shaders, textures, vertex buffers, etc.
- Commands
 - DrawIndexedPrimitive, Clear, SetTexture, etc.

OpenGL RHI

- Much of the API maps in a very straight-forward manner.
 - RHICreateVertexBuffer -> glGenBuffers + glBufferData
 - RHIDrawIndexedPrimitive -> glDrawRangeElements
- What about versions? 4.x vs 3.x vs ES2 vs ES3
- What about extensions?
- A lot of code can be shared between versions, even ES2 and GL4.

OpenGL RHI

- Use a class hierarchy with static functions.
- Each platform ultimately defines its own, typedefs it as FOpenGL.
- Optional functions defined here, e.g. FOpenGL::TexStorage2D()
 - Depending on platform could be unimplemented, glTexStorage2DEXT, glTexStorage2D, ...
- Optional features have a corresponding SupportsXXX() function.
 - E.g. FOpenGL::SupportsTexStorage().
 - If support depends on an extension, we parse the GL_EXTENSIONS string at boot time and return that.
 - If support is always (or never) available for the platform we return a static true or false. This allows the compiler to remove unneeded branches and dead code.
- Allows us to share basically all of our OpenGL code across ES2, GL3, and GL4!

What about shaders?

- That other stuff was easy.
 - Well, not exactly, Evan and Mathias will explain why later.
- We do not want to write our shaders twice!
 - We have a large, existing HLSL shader code base.
 - Most of our platforms support an HLSL-like shader language.
- We want to be able to compile and validate our shader code offline.
- Need reflection to create metadata used by the renderer at runtime.
 - Which textures are bound to which indices?
 - Which uniforms are used and thus need to be computed by the CPU?

What are our options?

- Can we just compile our HLSL shaders as GLSL with some magical macros?
 - No. Maybe for really simple shaders but the languages really are different in subtle ways that will cause your shaders to break in OpenGL.
- So use FXC to compile the HLSL and translate the bytecode!
 - Can work really well!
 - But not for UE4: we support Mac OS X as a full development platform.
- Any good cross compilers out there?
 - None that support shader model 5 syntax, compute, etc.
 - At least none that we could find two years ago :)

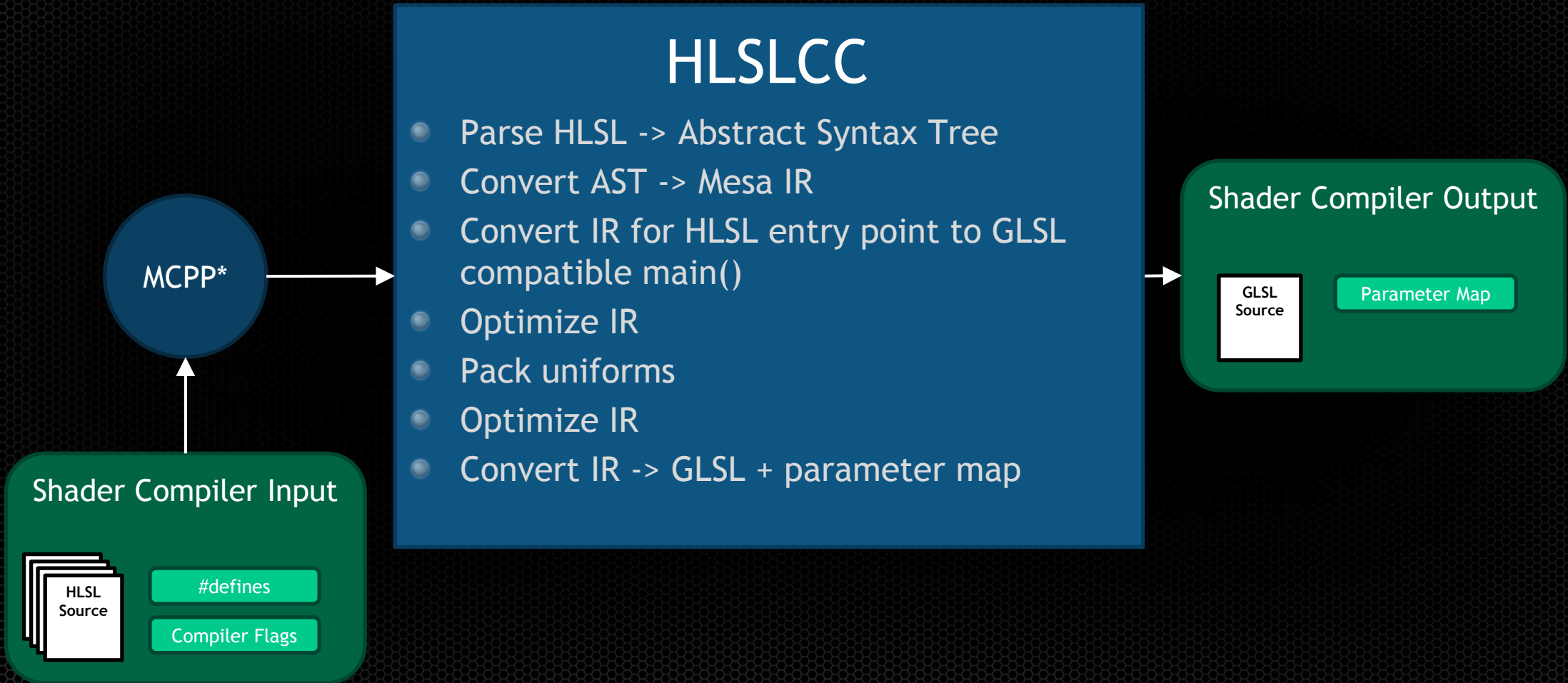
HLSLCC

- Developed our own internal HLSL cross compiler.
- Inspired by glsl-optimizer: <https://github.com/aras-p/glsl-optimizer>
- Started with the Mesa GLSL parser and IR: <http://www.mesa3d.org/>
- Modified the parser to parse shader model 5 HLSL instead of GLSL.
- Wrote a Mesa IR to GLSL converter, similar to that used in glsl-optimizer.
- Mesa includes IR optimization: this is an optimizing compiler!

Benefits of Cross Compilation

- Write shaders once, they work everywhere.
- Contain platform specific workarounds to the compiler itself.
- Offline validation and reflection of shaders for OpenGL.
 - Treat generated GLSL that doesn't work with a GL driver as a bug in the cross compiler.
- Simplifies the runtime, especially resource binding and uniforms.
- Makes mobile preview on PC easy!

GLSL Shader Pipeline



* <http://mcpp.sourceforge.net/>


Resource Binding

- When the OpenGL RHI creates the GLSL programs, setting up bind points is easy:

```
for (int32 i=0; i < NumSamplers; ++i)
{
    Location = glGetUniformLocation(Program, "_psi");
    glUniform1i(Location,i);
}
```

- Renderer binds textures to indices.
 - RHISetShaderTexture(PixelShader,LightmapParameter,LightmapTexture);
 - Same as every other platform.

(from parameter map)



Uniform Packing

- All global uniforms are packed in to arrays.
- Simplifies binding uniforms when creating programs:
 - We don't need the strings of the uniform's names, they are fixed!
 - `glGetUniformLocation("_pu0")`
- Simplifies shadowing of uniforms when rendering:
 - `RHISetShaderParameter(Shader,Parameter,Value)`
 - Copy value to shadow buffer, just like the D3D11 global constant buffer.
- Simplifies setting of uniforms in the RHI:
 - Track dirty ranges. For each dirty range: `glUniform4fv(...)`
 - Easy to trade off between the quantity of `glUniform` calls and the amount of bytes uploaded per draw call.

Uniform Buffer Emulation (ES2)

- Straight-forward extension of uniform packing.
- Rendering code gets to treat ES2 like D3D11: it packages uniforms in to buffers based on the frequency at which uniforms change.
- The cross compiler packs used uniforms in to the uniform arrays.
- Also provides a copy list: where to get uniforms from and where they go in the uniform array.

UV Space Differences



-1,-1

Normalized Device Coordinates

+1,+1

0,0



1,1

1,1



0,0

Direct3D UV Space

OpenGL UV Space

From the Shader's Point of View



+1,+1

-1,-1

Normalized Device Coordinates

0,0



1,1

Direct3D UV Space

0,0



1,1

OpenGL UV Space

Rendering Upside Down

- Solutions?
 - Could sample with $(U, 1 - V)$
 - Could render and flip in NDC $(X, -Y, Z)$
- We choose to render upside down.
 - Fewer choke points: lots of texture samples in shader code!
 - You have to transform your post-projection vertex positions anyway. D3D expects Z in $[0, 1]$, OpenGL expects Z in $[-1, 1]$.
 - Can wrap all of this in to a single patched projection matrix for no runtime cost.
 - You also have to reverse your polygon winding when setting rasterizer state!
 - BUT: Remember not to flip when rendering to the backbuffer!

After Patching the Projection Matrix



+1,+1

-1,-1

Normalized Device Coordinates



0,0

1,1

Direct3D UV Space



0,0

1,1

OpenGL UV Space

Next...

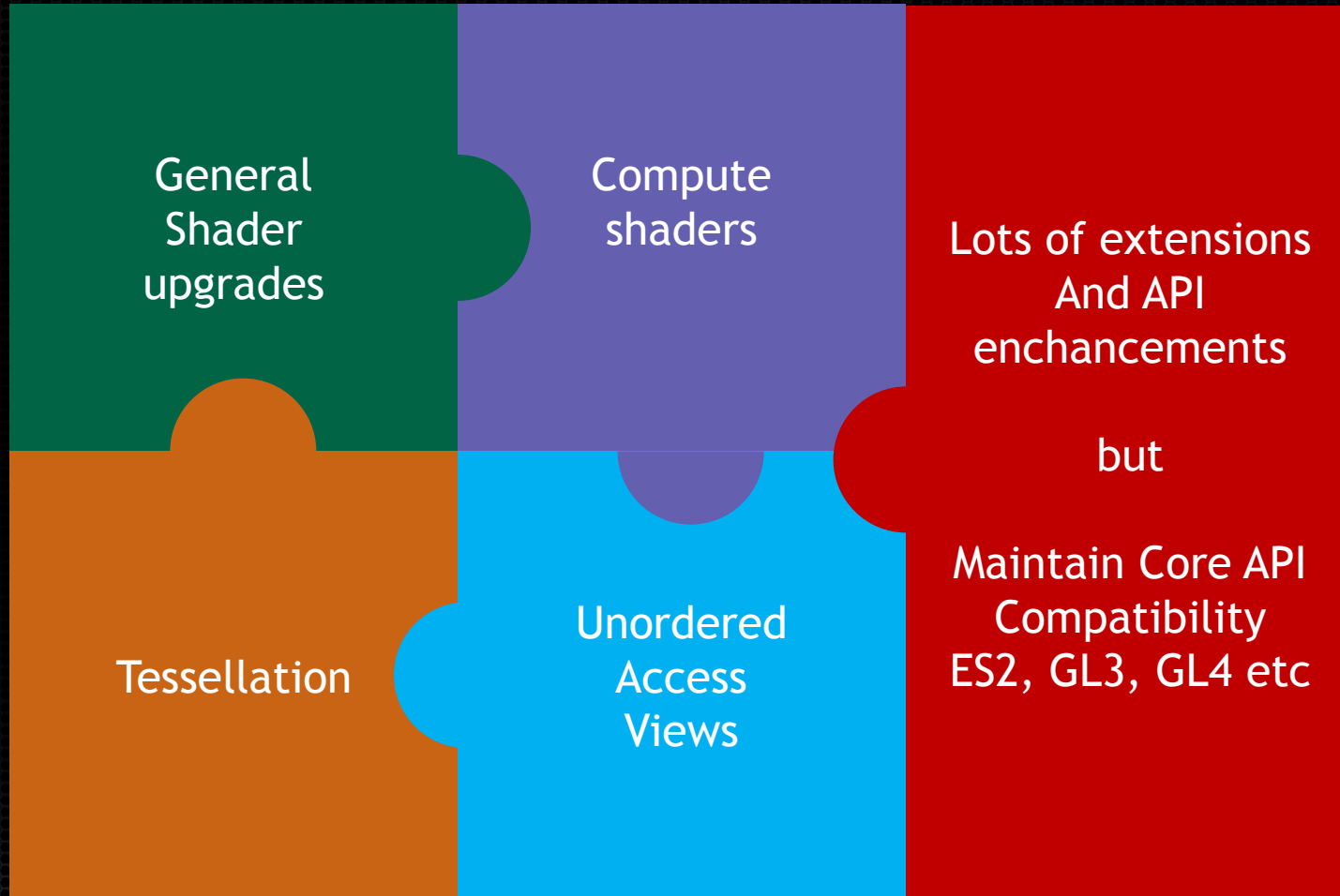
- Evan and Mathias will talk about extending the GL RHI to a full D3D11 feature set.
- Also, how to take all of this stuff and make it fast!

Achieving DX11 Parity

- Enable advanced effects
- Not a generic DX11 layer
- Only handle engine needs
- 60% shader / 40% engine
- Evolves over time
 - with engine
 - OpenGL API



From OpenGL 3.x to 4.x



General Shader Upgrades

- Too many to discuss
- Most straight forward
- Some ripple in HLSLCC
 - Type system
 - IR
 - Optimization

Shared Memory
RWTexture*
RWBuffer*
Memory Barriers
Shared Memory Atomics
UAV Atomic
Conversion and packing functions
Texture arrays
Integer Functions
Texture Queries
New Texture Functions
Early Depth Test

Compute Shader API

- Simple analogs for Dispatch
- Need to manage resources
 - No separate binding points
 - i.e. no CSetXXX
- Memory behavior is different
 - DirectX 11 hazard free
 - OpenGL exposes hazards
- Enforce full coherence
 - Guarantees correctness
 - Optimize later

```
//State Setup
glUseProgram( ComputeShader);

SetupTexturesCompute();
SetupImagesCompute();
SetupUniforms();

//Ensure reads are coherent
glMemoryBarrier(GL_ALL_BARRIER_BITS);

glDispatchCompute( X, Y, Z);

//Ensure writes are coherent
glMemoryBarrier(GL_ALL_BARRIER_BITS);
```


Unordered Access Views

- Direct3D 11 world view
 - Multiple shader object types
 - RWBuffer, RWTexture2D, RWStructuredBuffer, etc
 - Unified API for setup
 - CSetUnorderedAccessViews
- OpenGL world view
 - GL_image_load_store (Closer to Direct3D 11)
 - GL_shader_buffer_storage_object
- Solution
 - Use GL_image_load_store as much as possible
 - Indirection table for other types

Tessellation (WIP)

- Both have two shader stages
 - Direct3D11: Hull Shader & Domain Shader
 - OpenGL: Tessellation Control Shader & Tessellation Evaluation Shader

Tessellation (WIP)

Direct3D

Vertex Shader

Hull Shader

Tessellator

Domain Shader

Geometry Shader

OpenGL

Vertex Shader

Tessellation Control Shader

Tessellator

Tessellation Evaluation Shader

Geometry Shader

Tessellation (WIP)

- **Notable divergence:** HLSL Hull vs. GLSL Tessellation Control Shader
- HLSL has two Hull Shader functions
 - Hull shader (per control point)
 - PatchConstantFunction
- GLSL has a single TESSELLATION_CONTROL_SHADER
 - Explicitly handle patch constant computations

Tessellation Solution

- Execute TCS in phases
 - First hull shader
 - Next patch constants
- GLSL provides barrier
 - Ensures values are ready
- Patch constant function is scalar
 - Just execute on one thread
 - Possibly vectorize later

```
layout( vertices = 3) out;  
  
void main()  
{  
    //Execute Hull shader  
  
    barrier();  
  
    if (gl_InvocationID == 0)  
    {  
        //Execute Patch constant  
    }  
}
```


Optimization

- Feature complete is only half the battle
- No one wants pretty OpenGL 4.3 at half the performance
- Initial test scene < ¼ of Direct3D
 - 90 FPS versus ~13 FPS (IIRC)

HLSLCC Optimizations

- Initially heavily GPU bound
 - Internal engine tools were fantastic
 - Just needed to implement proper OpenGL query support
- Shadows were the biggest offender
 - Filtering massively slower
 - Projection somewhat slower
- Filter offsets and projection matrices turned into temp arrays
 - GPU compilers couldn't fold through the temp array
 - Needed to aggressively fold / propagate
- Reduced performance deficit to $< 2x$

Software Bottlenecks

- Remaining bottlenecks were mostly CPU
- Both application and driver issues
- Driver is very solid, but
 - Unreal Engine 4 is a big test
 - Heaviest use of new features
 - This effort made the driver better
- Applications issues were often unintuitive
- Most problems were synchronizations
- Overall, lots of onion peeling

Application & Driver Syncs

- Driver likes to perform most of its work on a separate thread
 - Often improves software performance by 50%
 - Control panel setting to turn it off for testing
 - Can be a hindrance with excessive synchronization
- Most optimization effort went into resolving these issues
- General process
 - Run code with sampling profiler
 - Look at inclusive timing results
 - Find large hotspot on function calling into the driver
 - Time reflects wait for other thread

Top Offenders

- glMapBuffer (and similar)
 - Needs to resolve and queued glBufferData calls
 - Replace with malloc + glBufferSubData
- glGetTexImage
 - Used with reallocating mipmap chains
 - Replace with glCopyImageSubData
- glGenXXX
 - Every time a new object is created
 - Replace with a cache of names quaried in a large block
- glGetError
 - Just say no, except in debug

Where are we?

- Infiltrator performance is at parity
 - (<5% difference)
- Primitive throughput test is faster
 - Roughly 1 ms
- Rendering test is slower
 - Roughly 1.5-2.0 ms

Where do we go?

- More GPU optimization
 - Driver differences
 - Loop unrolling improvements
- More software optimizations
 - Reduce object references
 - Already implemented for UBOs
 - Bindless Textures
 - More textures & Less overhead
 - BufferStorage
 - Allows persistent mappings
 - Reduces memory copying

Supporting Android

- Tegra K1 opened an interesting door
 - Mobile chip with full OpenGL 4.4 capability
 - Common driver code means all the same features
- Can we run “full” Unreal Engine 4 content?
 - Yes
 - Need to make some optimizations / compromises
 - No different than a low-end PC
- Work done in NVIDIA's branch
 - On-going work with Epic to feedback changes

What were the challenges?

- High resolution, modest GPU
 - Rely on scaling
 - Turn some features back (motion blur, bloom, etc)
 - Aggressively optimize where feasible
 - Do reflections always need to be at full resolution?
- Modest CPU
 - Often disable cpu-hungry rendering
 - Shadows
 - Optimize API at every turn
- Modest Memory
 - Remove any duplication

WE WOULD LIKE YOUR FEEDBACK

Please take a moment to fill out this 2 minute survey on your own device for this talk



We appreciate your input



We Are Hiring