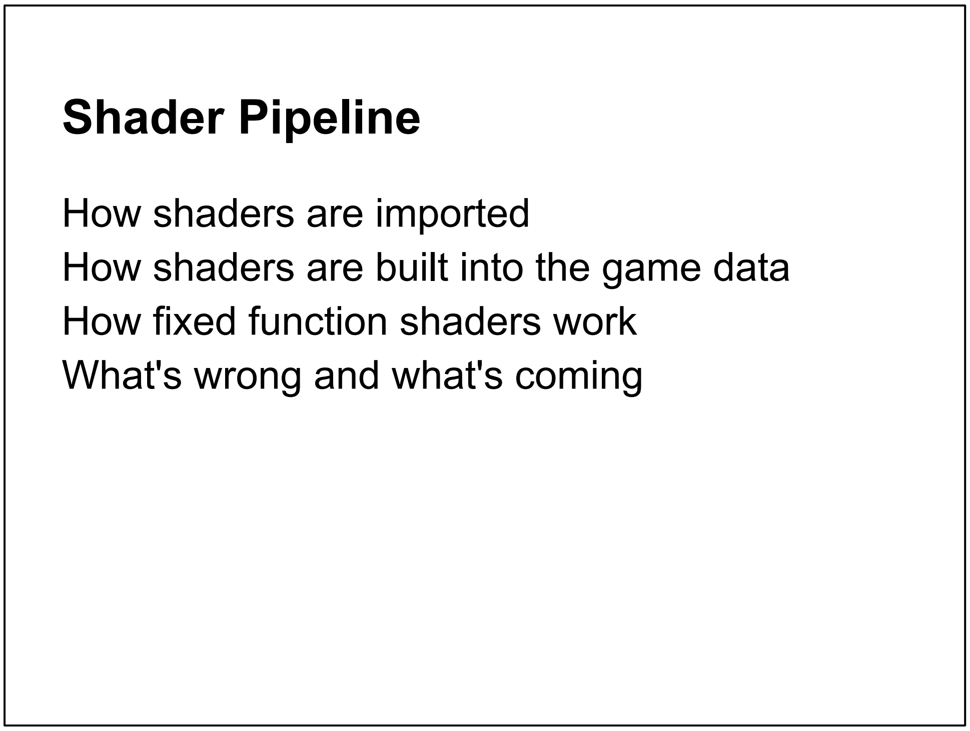
Unity's Shader Pipeline

2013.01, PirateCamp I
Aras Pranckevičius



Shader Pipeline

How shaders are imported
How shaders are built into the game data
How fixed function shaders work
What's wrong and what's coming

Everything in "what's coming" is subject to change!

Confusion:

ShaderLab
vs
"real actual shaders" aka Cg/HLSL
vs
"surface shaders"

ShaderLab

"a friend you can afford" (c) Nicholas ~2002?

Metadata: subshaders, properties, tags, LODs
Fixed function: blend, Z, colormask, culling, ...
Fixed function shaders: for old GPUs

Nested braces syntax

ShaderLab

```
Shader "Diffuse" {
Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
}
SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
CGPROGRAM
#pragma surface surf Lambert
sampler2D _MainTex;
fixed4 _Color;
struct Input {
        float2 uv_MainTex;
};
void surf (Input IN, inout SurfaceOutput o) {
        fixed4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
        o.Albedo = c.rgb;
        o.Alpha = c.a;
}
ENDCG
}
Fallback "VertexLit"
}
```

ShaderLab: code where?

Code: External/shaderlab/Library

stupid place, should move somewhere

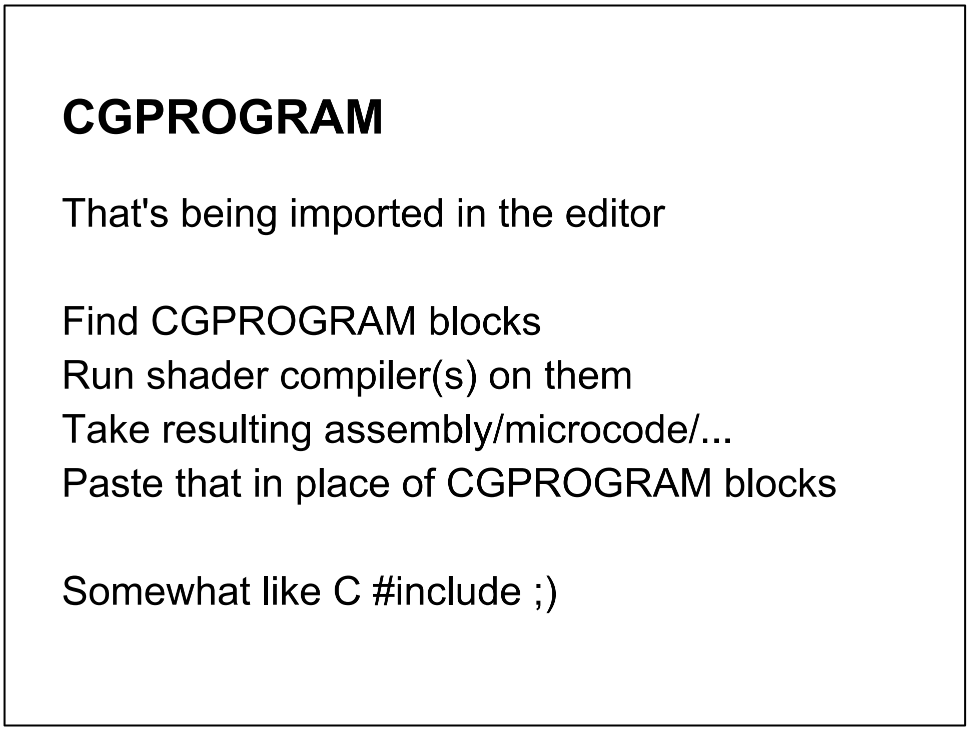
Code: Runtime/Shaders

Cg/HLSL code

```
Shader "Diffuse" {
Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
}
SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
CGPROGRAM
#pragma surface surf Lambert
sampler2D _MainTex;
fixed4 _Color;
struct Input {
        float2 uv_MainTex;
};
void surf (Input IN, inout SurfaceOutput o) {
        fixed4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
        o.Albedo = c.rgb;
        o.Alpha = c.a;
}
ENDCG
}
Fallback "VertexLit"
}
```

Cg/HLSL

Inside CGPROGRAM..ENDCG
Totally standard Cg/HLSL syntax *(almost)*
Completely separate world from ShaderLab!



CGPROGRAM

That's being imported in the editor

Find CGPROGRAM blocks
Run shader compiler(s) on them
Take resulting assembly/microcode/...
Paste that in place of CGPROGRAM blocks

Somewhat like C #include ;)

i.e. ShaderLab doesn't even know CGPROGRAM keyword exists. The blocks are extracted by shader importer, compiled into assembly, result pasted in place of original CGPROGRAM block. All "shaderlab" ever sees are compiled final result.

There's no Cg/HLSL compilation at runtime.

Cg importing

External executable "CgBatch", launched from editor

Code: External/shaderlab/CgBatch

Subject to change

ShaderLab importing

There's *no* importing going on for "just ShaderLab" syntax

All these Properties/SubShader/Pass/... blocks go straight into the game data, loaded & parsed at runtime

Stupid? Yes!

Will we fix it? *Maybe ;)*

Surface shaders

One more layer
For shaders that need lights/shadows
CGPROGRAM with *#pragma surface*
Parse that, generate *tons more* Cg core around it
Proceed as if the user had written that Cg code

Purely import time in CgBatch
Can just write all that by hand

Kuba&Tim's Unite 2012 talk is good: http://is.gd/UnityRenderingPipeline & http://video.unity3d.com/video/6957514/unite-2012-the-unity

Surface shader future?

Most likely a visual Shader Graph
Started at NinjaCamp VII
Who knows when will be finished ;)

Actual compiling

We compile into all backends at once

+++ no reimport on platform switch

+++ get error immediately if fails on some plat

--- longer import process

At game build time, strip out non-needed platform parts

Subject to change

Game build time: remove un-needed platform parts, and shader parts that "will never ever be used" on some platform (like deferred rendering passes when building to mobile)

This is all in ShaderWriter.cpp

Platforms

CGPROGRAM code is compiled to:

Direct3D 9 via Cg (*)

OpenGL via Cg (**) or hlsl2glsl

OpenGL ES 2.0 via hlsl2glsl + glsl optimizer

Direct3D 11 via MS' HLSL

Flash via Cg + our own d3d9-AGAL converter

Xbox360 via MS' 360 HLSL

PS3 via Sony's Cg

WiiU via *something*

(*) Want to change D3D9 to use Microsoft's HLSL compiler as well; less bugs and better codegen than Cg.
(**) Want to switch to hlsl2glsl for OpenGL soon-ish, i.e. always compile shaders into GLSL. Deprecate and eventually get rid of ARB_vertex_program/ARB_fragment_program support.

Parts we control

hlsl2glsl: open source, maintained by us

- https://github.com/aras-p/hlsl2glslfork
- fork of old ATI's project (dead since 2006). really bad code.

glsl optimizer: open source, maintained by us

- https://github.com/aras-p/glsl-optimizer
- fork of Mesa3D's GLSL compiler (active development, periodically synced)

d3d9-to-AGAL: in our codebase

Compile with Cg into shader model 2.0 D3D9 assembly

Parse that with Mojoshader

Convert to AGAL (added code to Mojoshader)

Some post-optimization passes to save registers & instructions

...all source files that have "AGAL" in their name ;)

Needs to happen soon(ish)

OpenGL ES 3.0

hlsl2glsl: parse bits of DX10-level HLSL, emit 3.0 syntax

glsl optimizer: emit 3.0 syntax (parsing *ShouldWork(tm)* already)

OpenGL 4.x

DX11-level HLSL, in hlsl2glsl and glsl optimizer

DX11 ComputeShaders -> GL 4.3

No one has *really* started doing any of the above yet.

Alternatively

Sounds like there's wide industry need for DX10/11 HLSL -> GLES3.0 / GL4.x toolchain

Collab with IHVs, other companies

Build on top of hlsl2glsl; add HLSL parser to glsl optimizer; or start from scratch

Back to compiling: problems

"One shader" in Unity: tons of GPU shaders
Light types, shadow types, lightmaps off/on, ...
"Shader keywords" pick which one
#pragma multi_compile
Preprocessor macros at compile time

Very soon (4.1): per-material keywords set by user; custom shader inspectors
Means: more variants in each shader

Want: even more shader variants

Imagine: custom shadowing system asset (e.g. VSMs), that defines shadow code snippet that needs to end up in all shaders.

Custom rendering pipeline (full deferred etc.) asset, likewise.

Old pipeline

Essentially unchanged since Unity 1.0

Shader source in, import all of it, spit out whole result

Importing in separate app (CgBatch), so process launch/exit for each shader

Back then you had **"// autolight 7"** instead of **#pragma surface**, but really the same

"autolight 7" - seven is a bitmask of which light type support to enable. of course ;)

Not webscale!

Create new shader in editor:

Compiles down to 245 GPU shaders

Takes 1-2 seconds

On Butterfly demo, we had just several shaders with many "keywords" for options

2-10 minutes to reimport one shader like that

Need to solve this

Some ideas

http://confluence.hq.unity3d.com/display/DEV/Shader+Compilation+Pipeline

Don't compile whole shader at once

Get Cg code + keywords + platform + version, hash it. Lookup if we have compiled result for that hash.

Very much like Cache server ;)

Compile variants that are needed *right now* first

Continue compiling the rest in background / on demand / at game build time

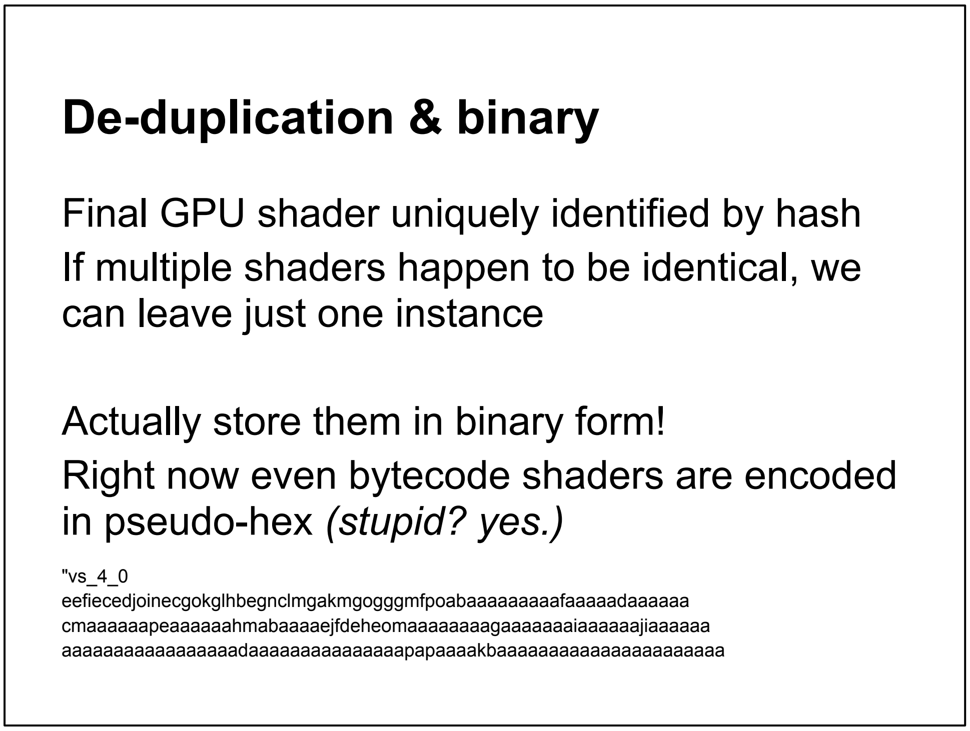
Don't ship non-needed shader variants

A built-in shader that we'd want (something like MIA shader from Butterfly demo) might have tens of thousands of variants inside

But a game might be using a hundred

At build time, collect which ones are used (what options materials use; what kind of lights we have; ...) and only leave these

Might need to expose some sort of "force include _this_" control to users. E.g. if there's no point lights in the actual scenes, but they are created dynamically from a script - then you still need point light shader variants to ship.



De-duplication & binary

Final GPU shader uniquely identified by hash

If multiple shaders happen to be identical, we can leave just one instance

Actually store them in binary form!

Right now even bytecode shaders are encoded in pseudo-hex *(stupid? yes.)*

```
"vs_4_0
eefiecedjoinecgokglhbegnclmgakmgogggmfpoabaaaaaaaaafaaaaadaaaaaa
cmaaaaaapeaaaaaahmabaaaaejfdeheomaaaaaaaagaaaaaaaiaaaaaajiaaaaaa
aaaaaaaaaaaaaaaadaaaaaaaaaaaaaaapapaaaakbaaaaaaaaaaaaaaaaaaaaa
```

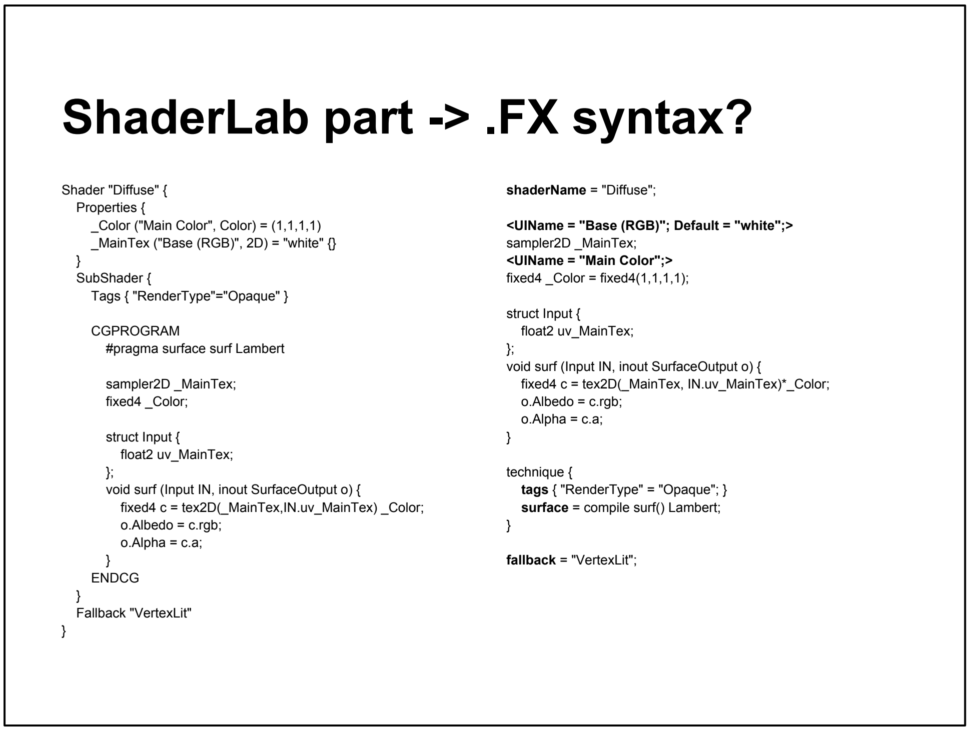
Pseudo-hex: 16 digits are just 'a'+digit. You can sing the shader bytecode!

ShaderLab part

Zero need to lex & parse string representation of ShaderLab part at runtime

Serialize it into actual binary data

Perhaps blobify Mécanim stylee



ShaderLab part -> .FX syntax?

```
Shader "Diffuse" {
   Properties {
      _Color ("Main Color", Color) = (1,1,1,1)
      _MainTex ("Base (RGB)", 2D) = "white" {}
   }
   SubShader {
      Tags { "RenderType"="Opaque" }

      CGPROGRAM
         #pragma surface surf Lambert

         sampler2D _MainTex;
         fixed4 _Color;

         struct Input {
            float2 uv_MainTex;
         };
         void surf (Input IN, inout SurfaceOutput o) {
            fixed4 c = tex2D(_MainTex,IN.uv_MainTex) _Color;
            o.Albedo = c.rgb;
            o.Alpha = c.a;
         }
      ENDCG
   }
   Fallback "VertexLit"
}
```

```
shaderName = "Diffuse";

<UIName = "Base (RGB)"; Default = "white";>
sampler2D _MainTex;
<UIName = "Main Color";>
fixed4 _Color = fixed4(1,1,1,1);

struct Input {
   float2 uv_MainTex;
};
void surf (Input IN, inout SurfaceOutput o) {
   fixed4 c = tex2D(_MainTex, IN.uv_MainTex)*_Color;
   o.Albedo = c.rgb;
   o.Alpha = c.a;
}

technique {
   tags { "RenderType" = "Opaque"; }
   surface = compile surf() Lambert;
}

fallback = "VertexLit";
```

In hindsight, inventing our own syntax for ShaderLab part wasn't probably the best idea. In FX-like syntax, both parts (Cg/HLSL code and fixed function/metadata) are much more naturally "integrated", plus there's less duplication of same information.

Here's an imaginary example of how Diffuse shader *could* look in FX-like syntax. Details may vary, but you get the idea.

Not sure if worth it, especially with shader graph potentially coming...

Fixed function shaders

For DX7 / OpenGL ES 1.1 GPUs

```
Material {
    Diffuse [_Color]
    Ambient [_Color]
    Emission [_Emission]
}
Lighting On
SetTexture [_MainTex] {
    Combine texture * primary DOUBLE, texture * primary
}
```

Fixed function shaders

Not a problem on actual DX7 / GLES1.1

Problem on platforms where fixed function **doesn't exist**

...which is almost everywhere except D3D9 & desktop OpenGL!

Fixed function emulation

Emulate FF shaders by generating *actual shaders*

- at runtime
- on demand

Can't pre-generate them offline since too many possible variants (hundreds of thousands)

Fixed function emulation problems

Possible & quite easy: PC (D3D9/GL), mobile, Flash

Impossible in theory: D3D11 (incl. WinRT/WP8)
- Didn't stop us, but not ideal ;)

Impossible: consoles (360/PS3/WiiU/...)

Very similar problem: how Fog is implemented. To avoid shipping 4x more shader variants (see previous slides), we patch shaders at runtime to insert fog calculations. Easy on D3D9/GL/mobile. Not so much on D3D11 (made it work for desktop, but not WinRT/WP7 DX11Level9) & consoles.

Want: kill fixed function

- Kill pre-DX9 GPU support on PC, pre-GLES1.1 support on mobile
- Remove fixed function traces from built-in shaders
  - Problem VertexLit ones, hard to make that fully work
- Shader Graph (eventually) should make it easy to write simple custom shaders

Question time!