TA 5

- The GPU
- Fragment & Vertex Shaders
- Shaders in Unity
- Demo
- EX2

Shaders

Computer Graphics 2020

The GPU

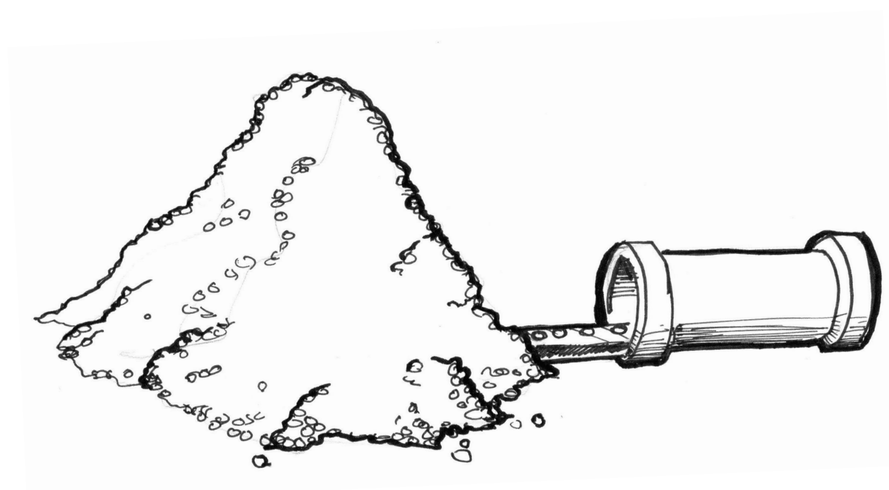
- The ***Graphics Processing Unit***, also known as a *Graphics Card*, is responsible for displaying images (i.e. everything!) on screen

The GPU

- The GPU Needs to calculate the color of each and every pixel, many times per second
- For a modern 2880×1800 display running at 60 frames per second this adds up to 311,040,000 calculations per second!
- The GPU solves this using ***Parallel Processing***

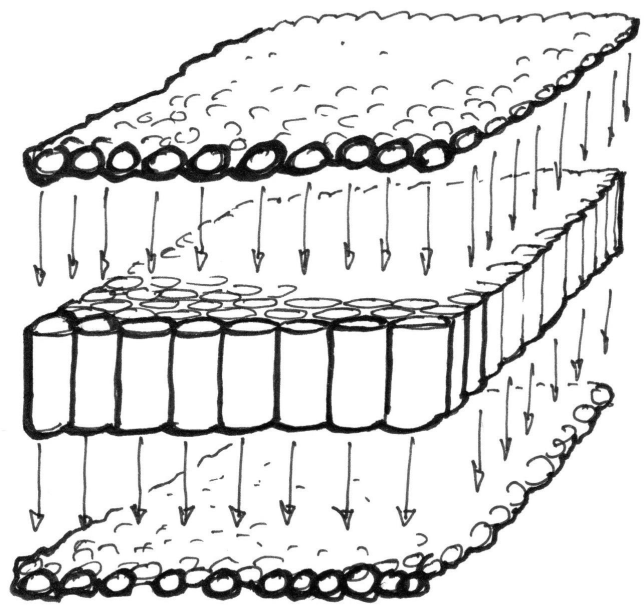
The GPU

- The CPU is generally composed of a few cores with lots of cache memory, that can handle a few software threads at a time



The GPU

- The GPU is composed of hundreds of cores that can handle thousands of threads simultaneously



The GPU

- Another “super power” of the GPU are special math functions accelerated via hardware
- This same architecture is also useful for processing hundreds and thousands of faces, edges and vertices that make up a 3D scene!

The GPU

- The GPU’s parallel architecture can be used for more than just graphics

- Most notably in deep learning, training neural networks requires many operations that can be run in parallel on the GPU - orders of magnitude faster than training on the CPU

GPU Programming

- The powerful architectural design of the GPU comes with constraints and restrictions
- To run in parallel threads have to be independent from each other, blind to what the rest of the threads are doing
- Data must flow in one direction only - impossible to pass the output of one thread into another thread

GPU Programming

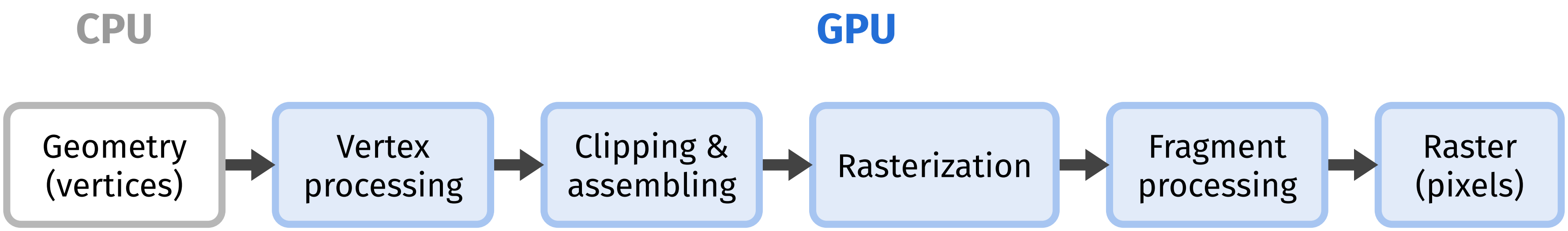
- A thread does not know what it was doing in the previous moment. It could be drawing a button from the UI of the operating system, then rendering a cube in a 3D scene, then displaying the text of an email
- Because of these constraints, a GPU cannot run "normal" programs
- No debugging and breakpoints for shaders

Shaders

- A ***Shader*** is a program that runs on the GPU rather than the CPU, to take advantage of parallel computation
- Written in shading languages - Cg, HLSL, GLSL
- Usually has a “C-like” syntax

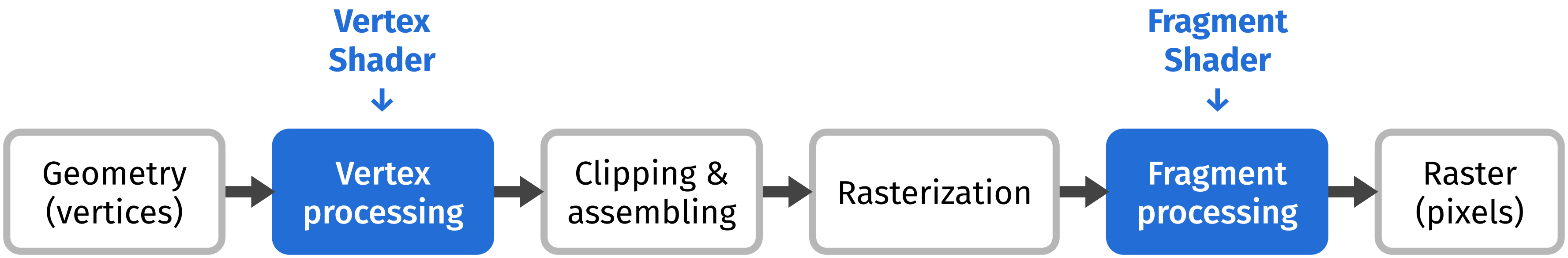
The Graphics Pipeline

- Most of the graphics pipeline runs on the GPU, to take advantage of parallel processing



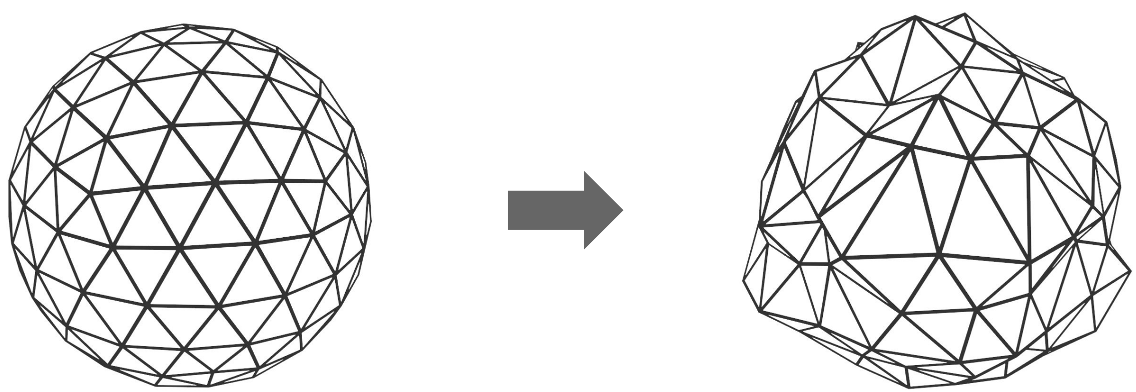
Shaders

- Shaders allow us to program and edit the graphics pipeline in 2 steps:



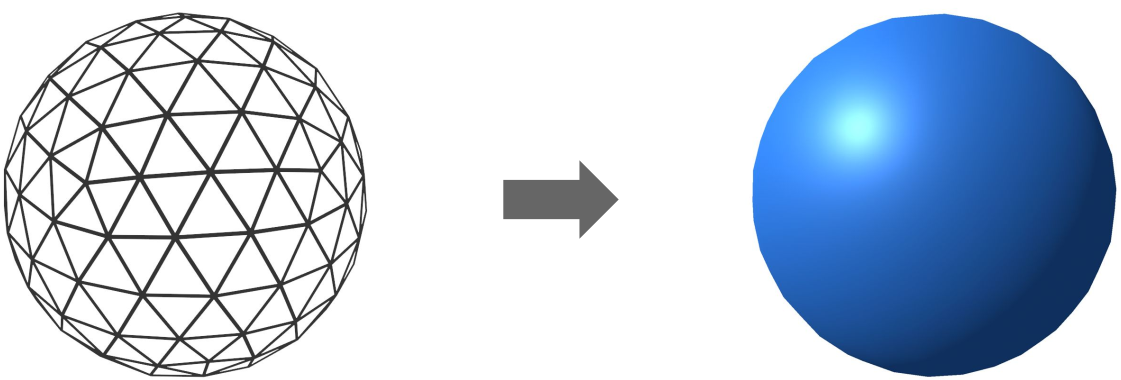
Shaders

- Vertex Shader
  - Converts world-space coordinates to clip-space coordinates (using the MVP matrix)
  - Can optionally edit the position of a vertex



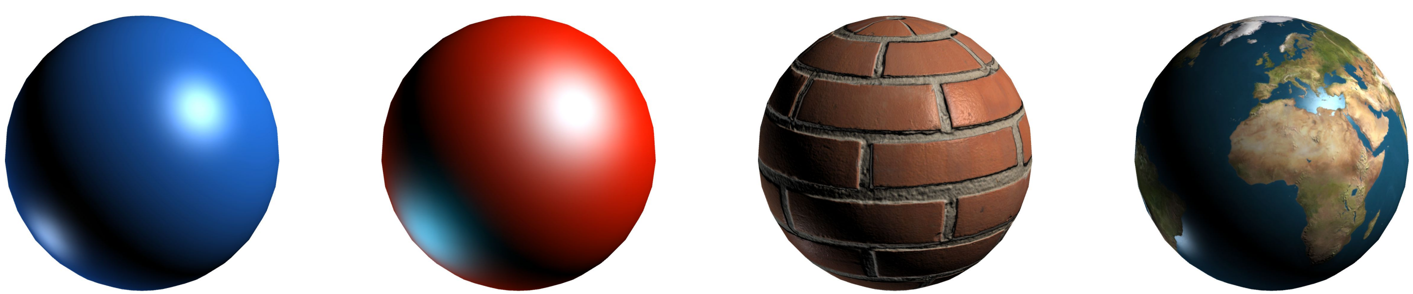
Shaders

- Fragment Shader
  - Uses vertex data, lighting information, materials, etc.
  - Determines the color of a fragment / pixel



Shaders in Unity

- Unity lets us write shaders for multiple platforms
- To use a shader, we must first create a ***Material***, to which we assign our shader
- Using the ***Material Inspector*** we can edit the properties of the shader to create multiple materials with the same shader:



Shaders in Unity

- Shader code is in Unity is wrapped in ***ShaderLab*** which is used to organize the Shader structure
- Files have a `.shader` extension
- General file structure:

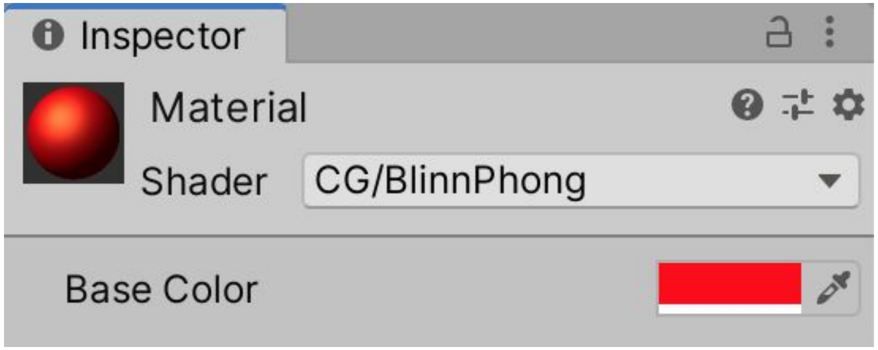
```
Shader "CG/MyShader" {
    Properties {
        // Shader parameter e.g. colors and textures
    }
    SubShader {
        // Shader code goes here
    }
}
```

Shaders Properties

- ***Properties*** are declared in at the beginning of the file. These are parameters for the shader that are exposed in the Unity Material Inspector:

```
_Color ("Base Color", Color) = (1, 1, 1, 1)
```

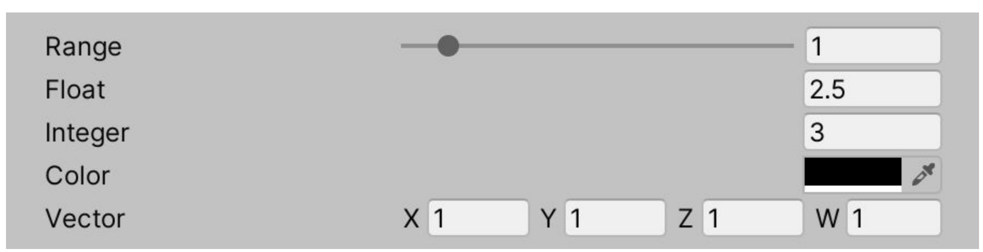
Internal name | Inspector title | Property type | Default value



Shaders Properties

- Each type has a different UI in the inspector:

```
_Range ("Range", Range(0, 10)) = 1
_Float ("Float", Float) = 2.5
_Integer ("Integer", Int) = 3
_Color ("Color", Color) = (0, 0, 0, 1)
_Vector ("Vector ", Vector) = (1, 1, 1, 1)
```



ShaderLab File

```
Shader "CG/MyShader"
{
    Properties
    {
        // Shader parameter e.g. colors and textures
    }
    SubShader
    {
        Pass
        {
            // Shader code goes here
        }

        // more passes if needed
    }
}
```

SubShaders & Passes

- Each ***SubShader*** is a collection of Passes.
- ***Passes*** contain the actual shader code. Each pass renders the object geometry once. Sometimes we will want to render multiple passes to achieve certain effects
- Multiple SubShaders can be defined to support different hardware. In this course we will always only use one SubShader

Cg Snippets

- The actual shader code is written in the Cg language (which is practically identical to HLSL) by embedding ***Cg Snippets*** in a pass block:

```
Pass {
    // usual pass setup...
    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        // the actual Cg/HLSL shader code
    ENDCG
    //  ...the rest of pass
}
```

Cg Snippets

- At the start of the Cg snippet, compilation directives can be given as `#pragma` statements
- `#pragma vertex vert` - Compile the function named `vert` as the vertex shader
- `#pragma fragment frag` - Compile the function named `frag` as the fragment shader
- Each snippet must contain at least one vertex program and one fragment program

Cg Data Types

- The majority of calculations in shaders are carried out on floating-point numbers
- Cg has a few primitive data types which differ in precision (and, consequently, efficiency)
- `float` - High precision (32 bit)
    - generally used for world space positions, texture coordinates, or scalar computations involving complex functions

Cg Data Types

- `half` - Medium precision (16 bit)
  - Useful for short vectors, directions, object space positions
- `fixed` - Low precision (11 bit)
  - Useful for colors
- `int` - Integer precision
  - Useful as loop counters or array indices

Cg Vector Types

- Cg has built-in vector and matrix types that are created from the basic types. For example:
- `float3` - High precision 3D vector with three components that can be accessed via `.x .y .z`
- `fixed4` - Low precision 4D vector with four components: `.x .y .z .w`
  - When working with colors, we can also access the components via: `.r .g .b .a`

Cg Vectors Usage

```
float2 a = float2(0.1, 0.2);
float3 b = float3(0.0, a); // = float3(0.0, 0.1, 0.2)
float4 c = float4(b, 1.0); // = float4(0.0, 0.1, 0.2, 1.0)
float3 d = 1.0; // = float3(1.0, 1.0, 1.0)

length(a); // = sqrt(0.1^2 + 0.2^2) = 0.223
c.z;  // = 0.2 ⇒ xyzw access
c.b;  // = 0.2 ⇒ rgba access
c[2]; // = 0.2 ⇒ integer indexing
float3 v = c.zyx; // = float3(0.2, 0.1, 0.0) - “swizzling”

b + d;     // = float3(1.0, 1.1, 1.2)
b * d;     // = float3(0.0, 0.1, 0.2) ⇒ element-wise
dot(b, d) // = 0.3 ⇒ dot product
```

Cg Matrix Usage

```
float3x3 m = float3x3(
    1.1, 1.2, 1.3, // first row
    2.1, 2.2, 2.3, // second row
    3.1, 3.2, 3.3  // third row
);
float3 row2 = m[2];  // = float3(3.1, 3.2, 3.3)
float m20 = m[2][0]; // = 3.1
float m21 = m[2].y;  // = 3.2

float2 v = float2(10., 20.);
float2x2 a = float2x2(1., 2., 3., 4.);
float2x2 b = transpose(a); // = float2x2(1., 3., 2., 4)
float2 w = mul(a, v); // = float2(1*10 + 2*20, 3*10 + 4*20)
```

UnityCG.cginc

- Unity has a number of built-in utility functions and variables designed to make writing shaders simpler and easier
- Import at the beginning of your Cg snippet:

```
#include "UnityCG.cginc"
```

- Full documentation:

docs.unity3d.com/Manual/SL-BuiltinIncludes.html

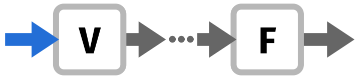
Semantics

- When writing shader programs, input and output variables need to have their “intent” indicated
- ***Semantics*** indicate to the hardware how certain variables connect to the rest of the graphics pipeline
- For example, when sending inputs to the vertex shader, color and position are indicated with the `COLOR` and `POSITION` semantics respectively

Semantics

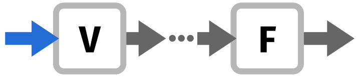
- Often inputs are declared in a struct, instead of listing them one by one:

```
// input to vertex shader
struct appdata
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
// vertex output to fragment input
struct v2f
{
    float4 pos : SV_POSITION;
    fixed4 color : COLOR0;
};
```



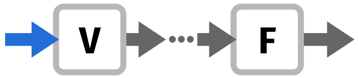
Vertex Shader Inputs

- The main vertex shader function (indicated by the `#pragma vertex` directive) needs to have semantics on all of the input parameters
- These correspond to individual Mesh data elements, like vertex position, surface normal and texture coordinates
- `UnityCG.cginc` provides commonly used vertex structs, for example: `appdata_base` includes position, normal and one texture coordinate



Vertex Shader Inputs

- Vertex input data, such as `POSITION` and `NORMAL`, are given in object-space (i.e. *model*) coordinates
- A vertex shader **must** output the clip-space position (*projection*) of the vertex
- In many cases, we would also like to know the world-space vertex positions and surface normal directions (i.e. to calculate shading)

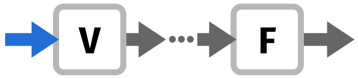


Vertex Shader Inputs

- `UnityCG.cginc` includes transformation matrices to transform between coordinate systems

- To transform from object-space to world-space we can use the *model matrix*, which is given in the variable `unity_ObjectToWorld`:

```
float3 worldPos = mul(unity_ObjectToWorld, input.vertex);
```



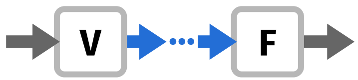
Vertex Shader Inputs

- To find the clip-space position of the vertex, we need to use the *MVP matrix*, given in the variable `UNITY_MATRIX_MVP`:

```
float3 clipPos = mul(UNITY_MATRIX_MVP, input.vertex);
```

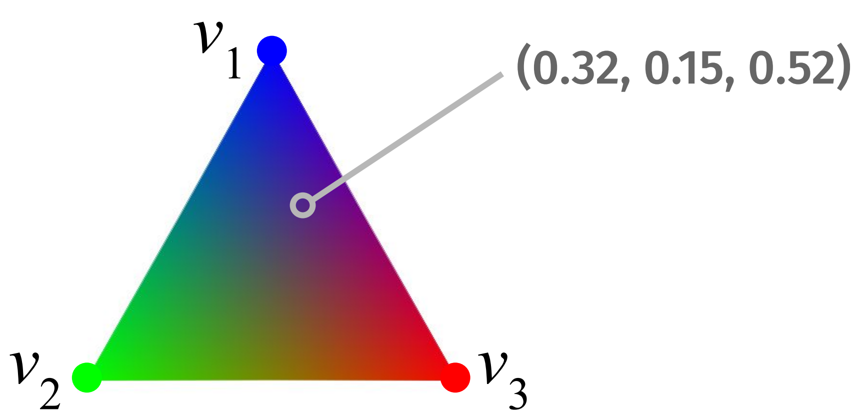
- Because this is a common and required operation, `UnityCG.cginc` defines a function to do this efficiently:

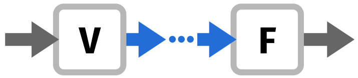
```
float3 clipPos = UnityObjectToClipPos(input.vertex);
```



Interpolators/Varyings

- Values output from the vertex shader are called ***interpolators*** or ***varyings***
- They will be interpolated across the face of the rendered triangles, and the values at each point will be passed as inputs to the fragment shader



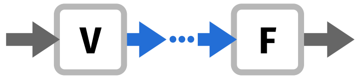


Interpolators/Varyings

- Like we said, a vertex shader must output the clip-space position of the vertex:

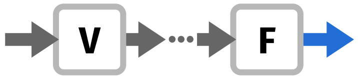
```
V2f output; // Initialize output struct
output.pos = UnityObjectToClipPos(input.vertex);
```

- This output needs to have the `SV_POSITION` semantic, and be of a `float4` type
- At each fragment we will get an interpolated clip-space position, which gives us the position of the particular fragment!



Interpolators/Varyings

- Other than SV_POSITION, Any other outputs produced by the vertex shader are whatever your particular shader needs
- General varyings are labeled with TEXCOORD0, TEXCOORD1, etc.
- Colors and other low-precision varyings should be labeled with COLOR0, COLOR1, etc.



Fragment Shader Output

- A fragment shader generally outputs a single `fixed4` color value, marked with the semantic `SV_Target`

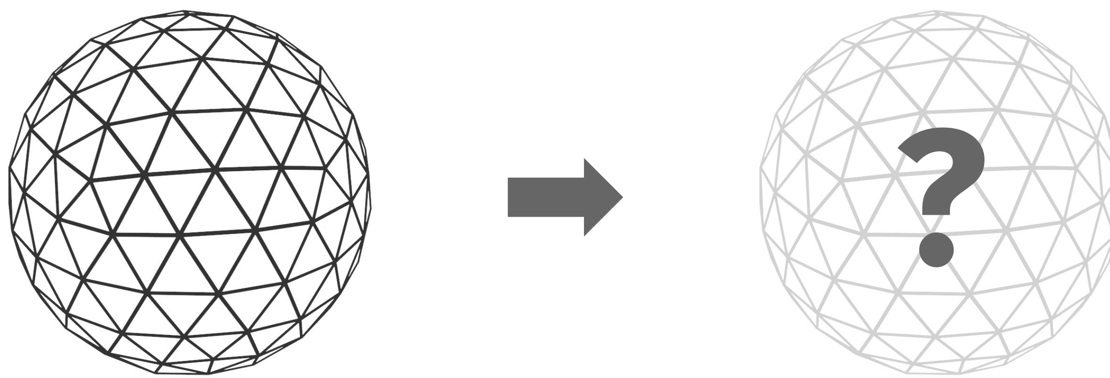
- Because it only returns a single value, the semantic can be indicated in the function declaration:

```
fixed4 frag (v2f i) : SV_Target
{ … }
```

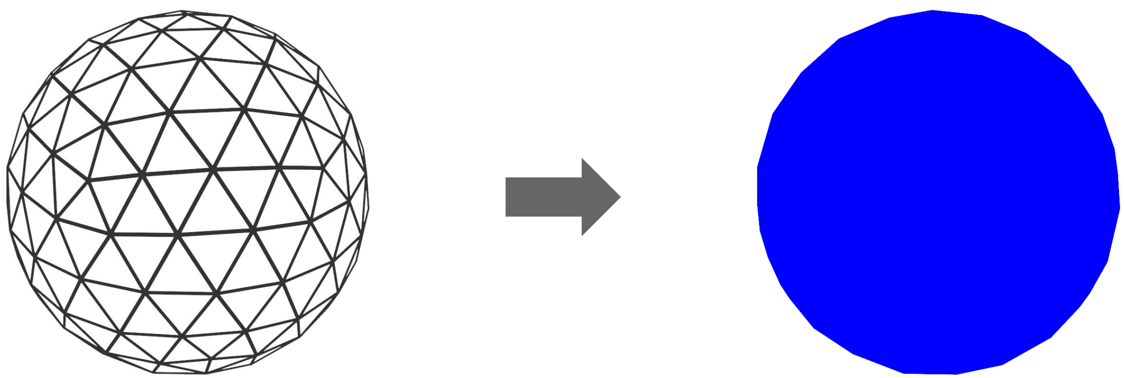
Shader Example

```
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"

struct appdata { float4 vertex : POSITION; };

struct v2f { float4 pos   : SV_POSITION;
             fixed4 color : COLOR0;      };

v2f vert (appdata input) {
    v2f output; // initialize output struct
    output.pos = UnityObjectToClipPos(input.vertex);
    output.color = fixed4(0.0, 0.0, 1.0, 1.0);
    return output;
}

fixed4 frag (v2f input) : SV_Target { return input.color; }
```

Shader Example



Shader Example



Using Properties

- To use ShaderLab properties we defined at the beginning of our shader file, we just need to declare them somewhere in the Cg Snippet using the chosen internal name
- In the properties block:

```
_Color ("Base Color", Color) = (1, 1, 1, 1)
```

- In our Cg Snippet:

```
uniform fixed4 _Color; // _Color is the internal name
```

The `uniform` Keyword

- The `uniform` Keyword marks a variable whose data is constant ("uniform") throughout the execution of a shader
- Uniforms are equal on all GPU threads and read-only
- Used to pass various information from the CPU to the GPU - lighting information, material properties and anything else you need

Using Lights

- If we want to use lighting information in a pass, we must add a tag before our Cg snippet so Unity can inject the data into our shader:

```
Pass {
    Tags { "LightMode" = "ForwardBase" }

    // rest of the pass, Cg snippet, etc.
}
```

- We can then access the position of the main light via the variable `float4 _WorldSpaceLightPos0`

Using Lights

- Generally: `(posX, posY, posZ, 1)`
- When using a directional light, the 4th coordinate will be “0”, and we get a normalized direction vector **towards** the light (position has no meaning!)
- To access the color of the light, we need to declare a uniform variable into which the value will be injected:

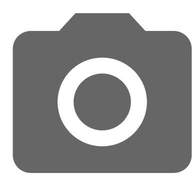
```
uniform fixed4 _LightColor0;
```

Camera

- To access the view position, i.e. position of our virtual camera, we can use the built-in variable:

```
float3 _WorldSpaceCameraPos
```

- Unlike light color, there is no need to declare a uniform, the variable is automatically included



Demo

Unity Shaders

EX2

- In this exercise you will use load a 3D mesh, process it and render it to the screen using shaders
- The goal of this exercise is to learn about 3D meshes, lighting, vertex shaders and fragment shaders
- You **must** submit this exercise in pairs