GAM250: Advanced Games Programming
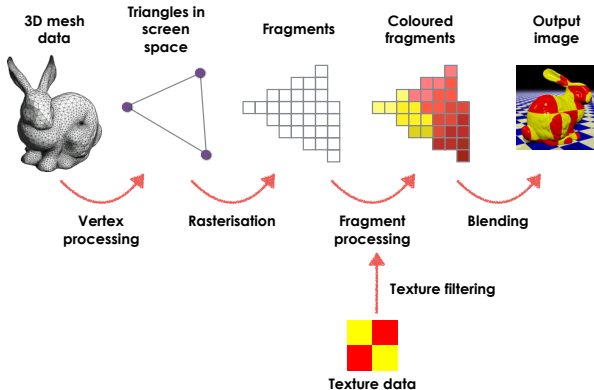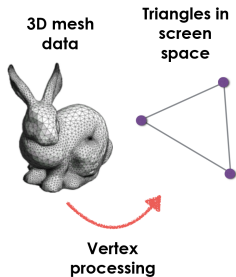# 3: Graphics Programming

Learning outcomes

- ► **Understand** the modern Programmable Graphics Pipeline
- ► **Understand** Unity's Material System
- ► **Write** shaders in Unity

The Graphics Pipeline

The 3D graphics pipeline

3D mesh data · **Triangles in screen space** · **Fragments** · **Coloured fragments** · **Output image**

Vertex processing · Rasterisation · Fragment processing · Blending

Texture filtering

Texture data

Vertex processing



3D mesh data

**Triangles in screen space**

Vertex processing

- ▶ Geometry is provided to the GPU as a **mesh** of **triangles**
- ▶ Each triangle has three **vertices** specified in 3D space (x, y, z)
- ▶ Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)
- ▶ May also apply particle simulations, skeletal animations or deformations, etc.

Rasterisation


Triangles in screen space

Fragments

Rasterisation
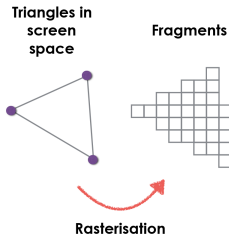
► Determine **which fragments** are covered by the triangle

► In practical terms, "fragment" = "pixel"

► Vertex processor can associate **data** with each vertex; this is **interpolated** across the fragments

Fragment processing

Fragments

Coloured fragments

Fragment processing

Texture filtering

Texture data
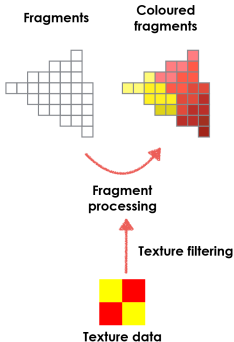
► Determine the **colour** of each fragment covered by the triangle

► **Textures** are 2D images that can be **wrapped** onto a 3D object

► Colour is calculated based on **texture**, **lighting** and other properties of the surface being rendered (e.g. shininess, roughness)

Blending

Coloured fragments

Output image
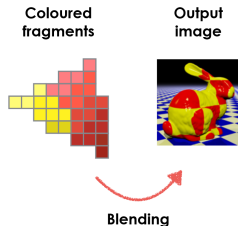
Blending

- ► Combine these fragments with the existing content of the image buffer
- ► **Depth testing**: if the new fragment is "in front" of the old one, replace it; if it is "behind", discard it
- ► **Alpha blending**: combine the old and new colours for a semi-transparent appearance

Shaders

- The vertex processor and fragment processor are **programmable**
- Programs for these units are called **shaders**
- **Vertex shader**: responsible for geometric transformations, deformations, and projection
- **Fragment shader**: responsible for the visual appearance of the surface
- Vertex shader and fragment shader are separate programs, but the vertex shader can pass arbitrary values through to the fragment shader

Subsurface Shaders

Shaders in Unity

- ► There are many approaches to writing shaders in Unity
    - ► Surface Shaders
    - ► Vertex and Fragment Shaders
    - ► Fixed Function Shaders
- ► The best method is to use Surface Shaders, this is the quickest way to get started
- ► This interacts with the standard lights and shadows in Unity
- ► Regardless of the shader type, your code will be wrapped in ShaderLab

ShaderLab

- ► ShaderLab is a simple scripting language for defining graphical effects
- ► It contains the following
  - ► Properties - These are shown in the inspector of the material and is a way to expose shader variables
  - ► SubShaders - Is a list of pass or the surface shader code itself

Shading Languages

High Level Shading Language (HLSL)

- ► Used for writing **shaders** for Direct3D and Unity3D
- ► C-like syntax
- ► But has data types that support mathematical operations

Programming in HLSL

- ► **if** statements, **for** loops, **while** loops, **do while** loops, **switch** statements, **break**, **continue**, **return** all work the same as C++
- ► *//Single-line comments* and */*Multi-line comments */* work the same too
- ► Function definitions and declarations are similar to C#, except that parameters must be declared as **in**, out or inout
- ► Recursion is **forbidden**
- ► No **class**

Data types in HLSL

- ► `bool`, `int`, `float`: just like in C++
- ► `float2`, `float3`, `float4`: **vectors** of `float`s
- ► `float2x2`, `float3x3`, `float4x4`: **square matrices** of `float`s
- ► **Arrays** of constant size e.g. `float` `myArray[10]`

Vectors

- An *n*-**dimensional vector** is formed of *n* numbers
- E.g. 2-dimensional vectors:

$$(1, 2) \qquad (-2.7, 0) \qquad (3.4, -12.7)$$

- E.g. 3-dimensional vectors:

$$(1, 2, 0) \qquad (-9, 6, 3.7) \qquad (2.1, 2.1, 2.1)$$

- Used to represent **points** or **directions** in *n* dimensions
- Also used to represent e.g. colours in RGB(A) space

Constructing vectors in GLSL

```
float3 a = float3(1.2, 3.4);
float3 b = float3(1); // same as float3(1, 1, 1)
float3 c = float3(a, 5.6); // same as float3(1.2,  ↩
    3.4, 5.6)
```

Vector maths

Most operations work **component-wise**:

```
float2 a = float2(1, 2);
float2 b = float2(3, 4);
float2 c = a + b; // c == float2(4, 6);
float2 d = a * b; // d == float2(3, 8);
```

Can also multiply a **vector** by a **scalar**:

```
float2 e = 3.1 * a; // e == float2(3.1, 6.2)
```

Accessing components

Can access the components of a vector as `.x, .y, .z, .w`:

```
float4 a = float4(1, 2, 3, 4);
float b = a.y;   // b == 2
float c = a.z;   // c == 3
a.x = 5;         // a == float4(5, 2, 3, 4)
a.w = a.y;       // a == float4(5, 2, 3, 2)
```

Can also use `r g b a` (for colours) and `t u v w` (for texture coordinates)

Swizzling

Can access multiple components in one go:

```
float4 a = float4(1, 2, 3, 4);
float2 b = a.xy;     // b == float2(1, 2)
float3 c = a.zyz;    // c == float3(3, 2, 3)
a.xw = float2(5,6);  // a == float4(5, 2, 3, 6)
a.xyzw = a.wzyx;     // a == float4(6, 3, 2, 5)
```

- ► **Can** use the same component twice in the **right-hand side** of an assignment
- ► **Cannot** use the same component twice in the **left-hand side** of an assignment
- ► Swizzling is generally **faster** than the equivalent code without swizzling
- ► Can also use `r g b a` or `t u v w`, but can't mix them (e.g. `.gbr` is valid but `.gzx` is not)

Texture Data Types

- Textures are stored in the **Sampler** data type
- There are different samplers for different types of texture
  - 1D Texture - **sampler1D**
  - 2D Texture - **sampler2D**
  - 3D Texture - **sampler3D**
  - Cube Map - **samplerCube**

Unity Types

- NB. When writing shaders you can used different precision data types rather than float (High precision)
    - Medium precision: **half** - directions, positions
    - Low precision: **fixed** - colours
- On Desktop PCs these are always converted to high precision
- These are important for optimisation for mobile

Surface Shader

Live Coding

Exercise 1 - Surface Shaders

- ► Map two textures onto an object
- ► Tint the object with a colour
- ► Animate the texture coordinates for one of the textures
- ► Implement a dissolve effect

Exercise 2 - Surface Shader (Vertex Shader)

- ► Add a vertex shader to the Surface Shader, ensure it carries out the standard transformation
- ► Extrude the mesh based on the Vertex Normals
- ► Animate this extrusion based

Further Reading

- ► Shaders Overview - `https://docs.unity3d.com/Manual/ShadersOverview.html`
- ► Gentle Introduction to Shaders - `http://www.alanzucconi.com/2015/06/10/a-gentle-introduction-to-shaders-in-unity3d/`
- ► Learning Shaders - `https://www.alanzucconi.com/2018/01/03/learning-shaders/`
- ► HLSL Language Syntax - `https://msdn.microsoft.com/en-us/library/windows/desktop/bb509615(v=vs.85).aspx`
- ► HLSL Intrinsic Functions - `https://msdn.microsoft.com/en-us/library/windows/desktop/ff471376(v=vs.85).aspx`