GAM250: Advanced Games Programming

4: Graphics Programming

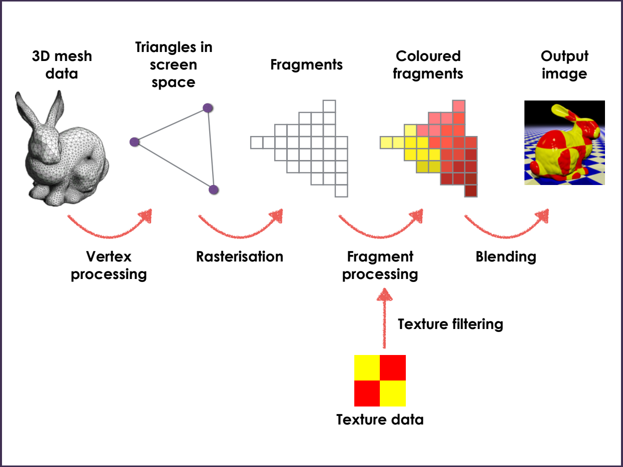
Learning outcomes

- **Understand** the modern Programmable Graphics Pipeline
- **Understand** Unity's Material System
- **Write** Surface and Image Effect Shaders in Unity

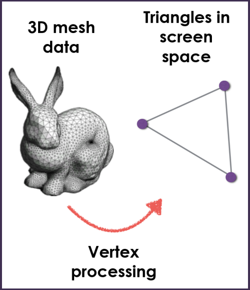
FALMOUTH
UNIVERSITY

The Graphics Pipeline

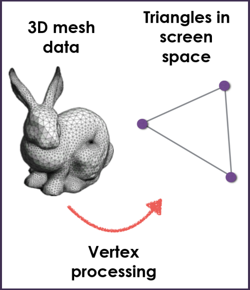
The 3D graphics pipeline



Vertex processing

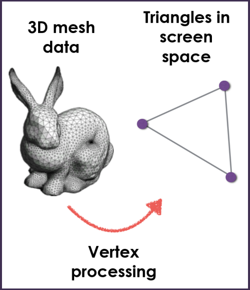


Vertex processing



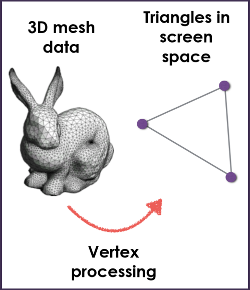
- Geometry is provided to the GPU as a **mesh** of **triangles**

Vertex processing



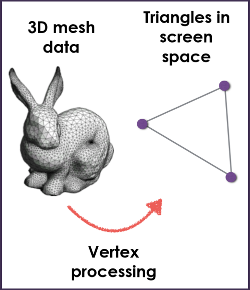
- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)

Vertex processing



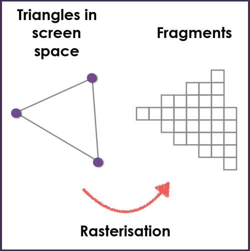
- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)

Vertex processing

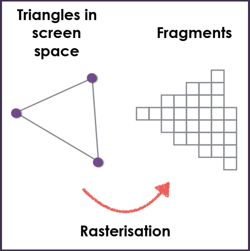


- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)
- May also apply particle simulations, skeletal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

Rasterisation

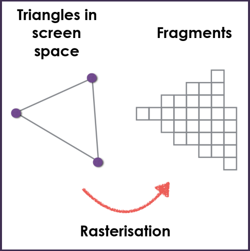


Rasterisation



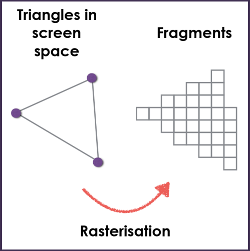
- Determine **which fragments** are covered by the triangle

Rasterisation



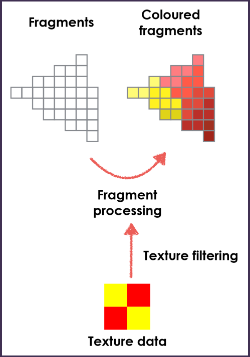
- Determine **which fragments** are covered by the triangle
- In practical terms, “fragment” = “pixel”

Rasterisation

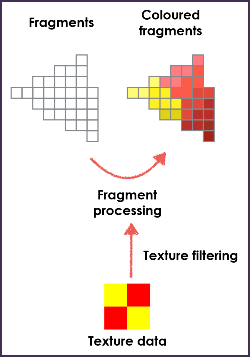


- Determine **which fragments** are covered by the triangle
- In practical terms, “fragment” = “pixel”
- Vertex processor can associate **data** with each vertex; this is **interpolated** across the fragments

Fragment processing

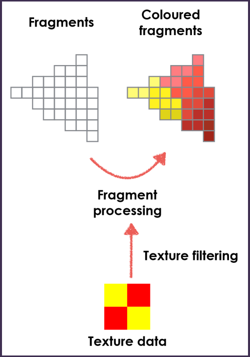


Fragment processing



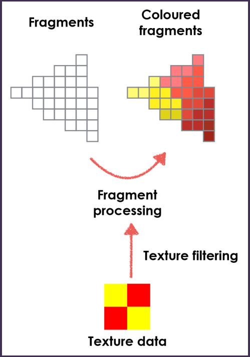
- Determine the **colour** of each fragment covered by the triangle

Fragment processing



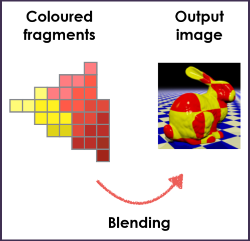
- Determine the **colour** of each fragment covered by the triangle
- **Textures** are 2D images that can be **wrapped** onto a 3D object

Fragment processing

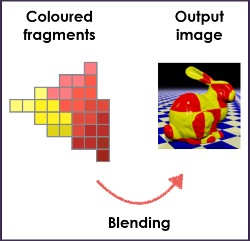


- Determine the **colour** of each fragment covered by the triangle
- **Textures** are 2D images that can be **wrapped** onto a 3D object
- Colour is calculated based on **texture**, **lighting** and other properties of the surface being rendered (e.g. shininess, roughness)

Blending

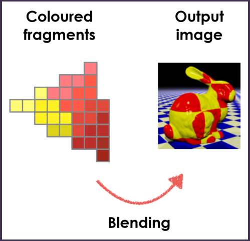


Blending



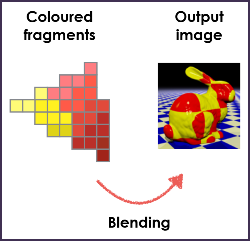
- Combine these fragments with the existing content of the image buffer

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

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**

Shaders

- The vertex processor and fragment processor are **programmable**
- Programs for these units are called **shaders**

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

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

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

Shaders in Unity

- There are many approaches to writing shaders in Unity
  - Surface Shaders

Shaders in Unity

- There are many approaches to writing shaders in Unity
  - Surface Shaders
  - Vertex and Fragment Shaders

Shaders in Unity

- There are many approaches to writing shaders in Unity
  - Surface Shaders
  - Vertex and Fragment Shaders
  - Fixed Function 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

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

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

ShaderLab

- ShaderLab is a simple scripting language for defining graphical effects
- It contains the following

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

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

High Level Shading Language (HLSL)

- Used for writing **shaders** for Direct3D and Unity3D
- C-like syntax

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++

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

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`

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**

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++

Data types in HLSL

- `bool`, `int`, `float`: just like in C++
- `float2`, `float3`, `float4`: **vectors** of `float`s

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

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

Vectors

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

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

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:

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

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

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

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);
```

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

Accessing components

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

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)
```

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

Swizzling

Can access multiple components in one go:

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)
```

FALMOUTH UNIVERSITY

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

FALMOUTH UNIVERSITY

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

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

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

Texture Data Types

- Textures are stored in the **Sampler** data type

Texture Data Types

- Textures are stored in the **Sampler** data type
- There are different samplers for different types of texture

Texture Data Types

- Textures are stored in the **Sampler** data type
- There are different samplers for different types of texture
  - 1D Texture - **sampler1D**

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**

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**

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)

Unity Types

- NB. When writing shaders you can used different precision data types rather than float (High precision)
  - Medium precision: **half** - directions, positions

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

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

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

FALMOUTH
UNIVERSITY

Post-Processing

What is Post-Processing

- Is processing that occurs after the scene is rendered

What is Post-Processing

- Is processing that occurs after the scene is rendered
- This allows us to implement effects such as static, distortions etc

What is Post-Processing

- Is processing that occurs after the scene is rendered
- This allows us to implement effects such as static, distortions etc
- Post-Processing effects can be stacked so that one feeds into the next

Writing Post-Processing Effects

- We first have to write a vertex and fragment shader and attach it to a material

Writing Post-Processing Effects

- We first have to write a vertex and fragment shader and attach it to a material
- Create a C# script and override the **OnRenderImage** function

Writing Post-Processing Effects

- We first have to write a vertex and fragment shader and attach it to a material
- Create a C# script and override the **OnRenderImage** function
- Inside the OnRenderImage function and call the **Graphics.Blit** function with the material as the last parameter

Writing Post-Processing Effects

- We first have to write a vertex and fragment shader and attach it to a material
- Create a C# script and override the **OnRenderImage** function
- Inside the OnRenderImage function and call the **Graphics.Blit** function with the material as the last parameter
- Attach the script to the camera

Post-Processing Live Coding

Further Reading

- Game Programming Patterns - `http://gameprogrammingpatterns.com/contents.html`
- Game Programming Patterns in Unity - `http://www.habrador.com/tutorials/programming-patterns/`
- Unity Design Patterns - `https://github.com/Naphier/unity-design-patterns`