COMP280: Specialisms in Creative Computing

# 9: Intro to Computer Graphics

# Learning outcomes

- **Recall** the key stages of the graphics pipeline
- **Explain** the differences between a CPU and a GPU
- **Understand** how the graphics pipeline is implemented in UE4

# Graphics and simulation hardware

# CPUs vs GPUs

- (CPU = central processing unit; GPU = graphics processing unit)
- GPUs are **highly parallelised**
  - Intel i7 6900K: **8** cores
  - Nvidia GTX 1080: **2560** shader processors
- GPUs are **highly specialised**
  - Optimised for floating-point calculations rather than logic
  - Optimised for performing the same calculation on several thousand vertices or pixels at once

# General purpose GPU (GPGPU)

- Early GPUs used a **fixed pipeline** – could only be used for rendering 3D graphics
- Modern GPUs use a **programmable pipeline** – can be programmed for other tasks
- Physics simulation (e.g. PhysX)
- Scientific computing (e.g. CUDA)
- Deep learning

# Graphics APIs

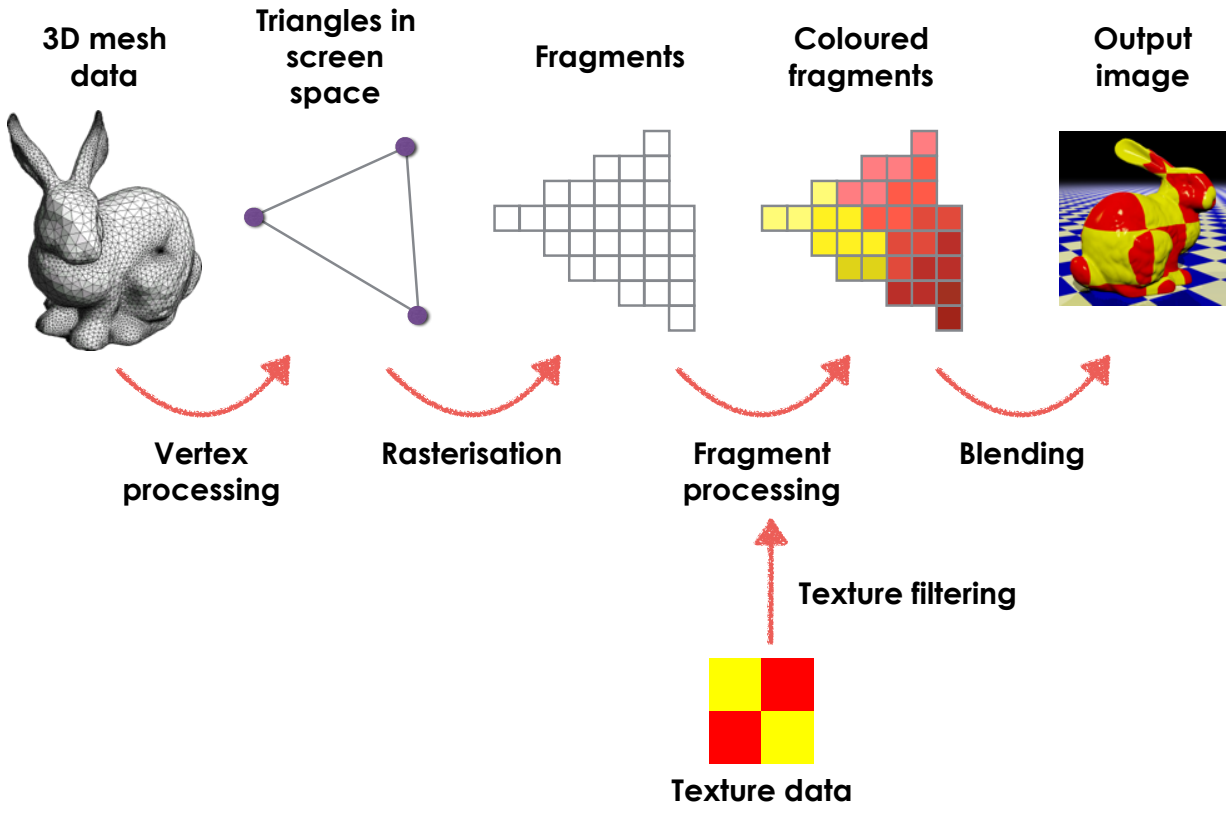
- Graphics APIs **abstract** away the differences between different manufacturers' GPUs
- There are several APIs in use today:
  - **OpenGL**: Open standard, very mature, very widely supported
  - **Vulkan**: Open standard, less mature, lots of control on rendering, lots of work to get a basic sample working
  - **Direct3D**: Microsoft only
  - **Metal**: Apple only
  - Sony and Nintendo consoles have their own APIs; Microsoft consoles use Direct3D

# Game Engines & Graphic APIs

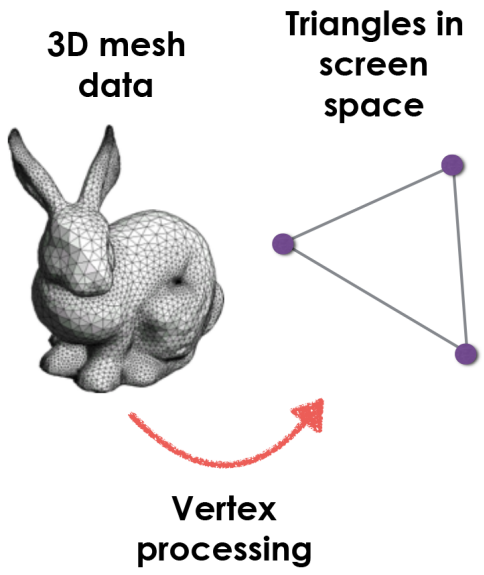
- Game Engines tend to support multiple Graphics API
- They using have an abstract rendering layer which has concrete implementations of D3D, OpenGL, Vulkan, Metal, Console APIs etc
- This allows the Engine to support multiple platforms
- In addition, this makes it easier to upgrade the engine to support new versions of APIs or newly released APIs

# The 3D graphics pipeline

# The 3D graphics pipeline

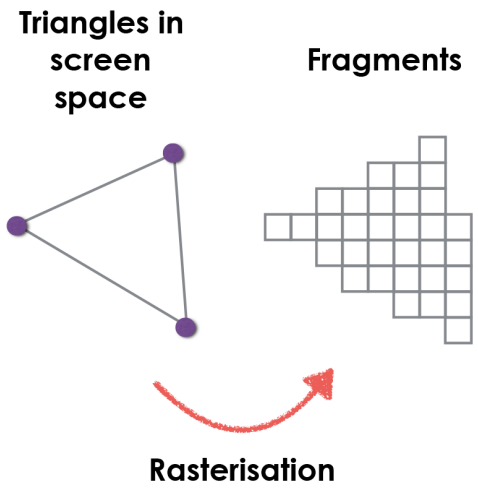


# Vertex processing



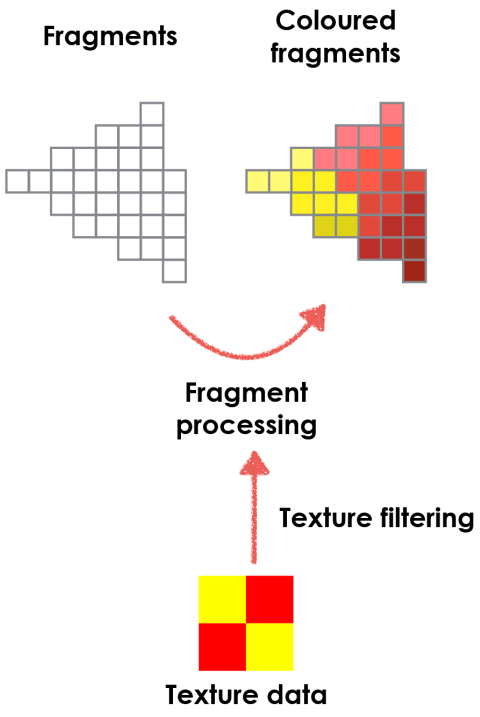

- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space $(x, y, z)$
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space $(x, y)$
- May also apply particle simulations, skeletal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

# Rasterisation



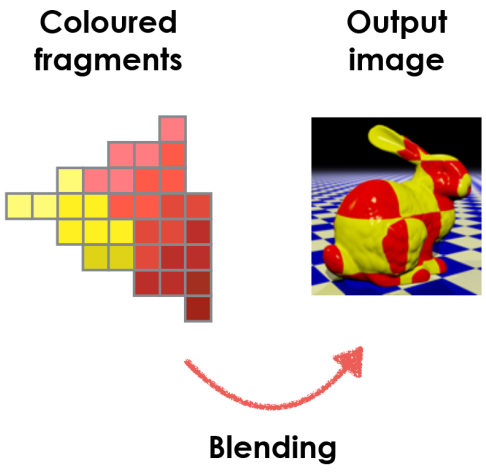

- Determine **which fragments** are covered by the triangle
- In practical terms, "fragment" = "pixel"
- Vertex processor can associate **data** with each vertex; this is **interpolated** across the fragments

# Fragment processing




- 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




- 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

# Shaders

- The Programmable units of the pipeline include
  - Vertex Processor
  - Tessellation Control
  - Tessellation Evaluation
  - Geometry Processor
  - Fragment Processor
- Programs for these units are called **shaders**

# Vertex & Fragment Shader

- These two units are a requirement for any Rendering to occur in D3D or OpenGL, the the other units are optional
- 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

- Takes in exactly one vertex as input
- Outputs one vertex
- Typical operations include Transformations and Animation

# Fragment Shader

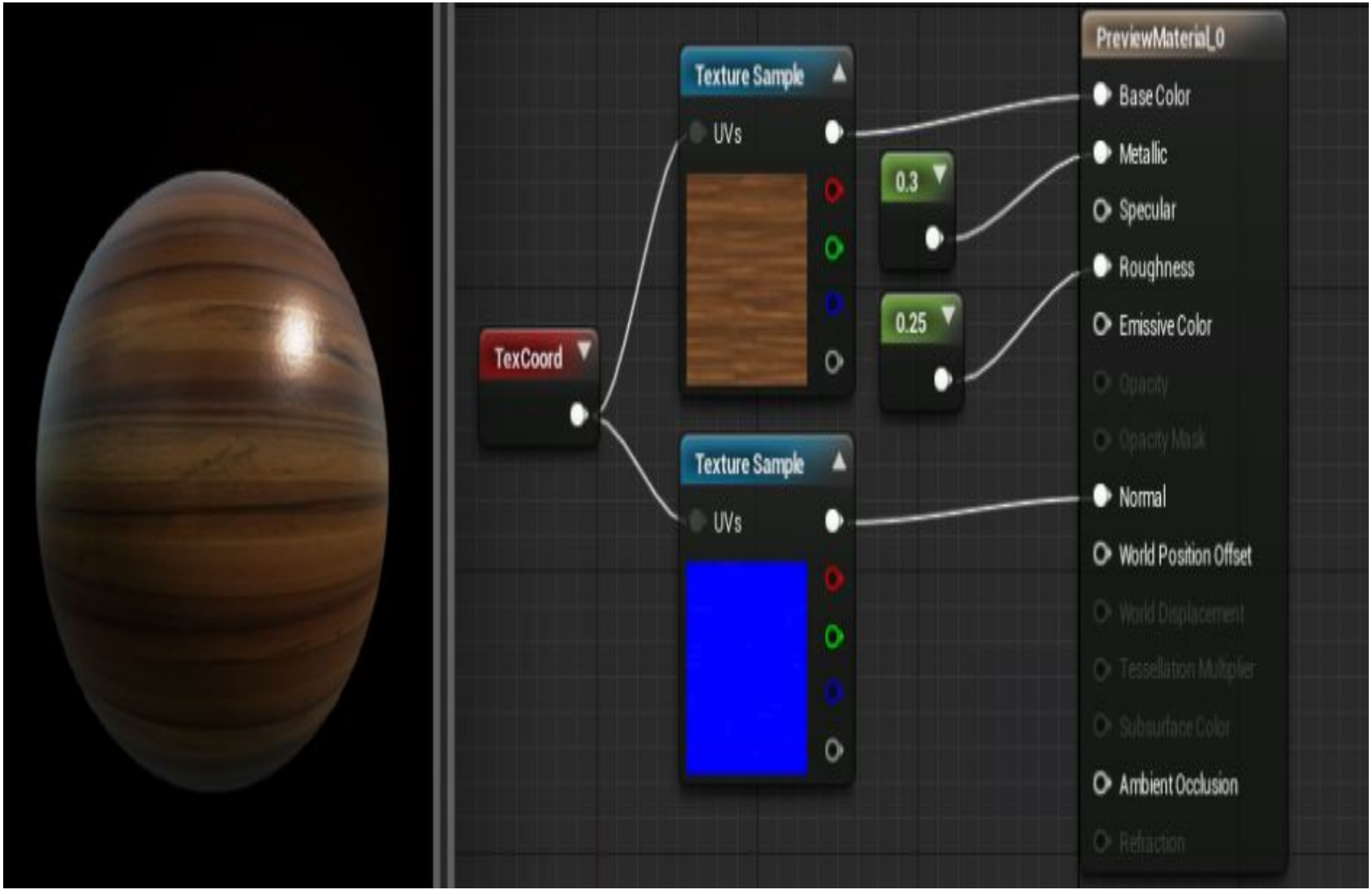
- Takes in a pixel fragment (see rasterization)
- Outputs colour and depth values
- Typically used for shading calculations and texturing

# UE4 Material System

# Shaders & Game Engines

- Most Game Engines abstract shaders into Materials
- These materials encapsulate a series of shaders and any other rendering states required to draw the effect
- These systems allow greater control for performance
- In addition, materials fit onto an Artists workflow

# UE4 Material System



# UE4 Material System

- This system uses a visual programming language to control the look of an object in the scene
- It consists of nodes called **Material Expressions**
- These nodes are simply bits of shader code designed to perform a single task
  - Multiplication
  - Texture Sample (also know as a look up)
  - Blends
- This allows you to build up a complex effect by chaining nodes together