COMP350: Algorithms & Optimisation

4: GPU Optimisation

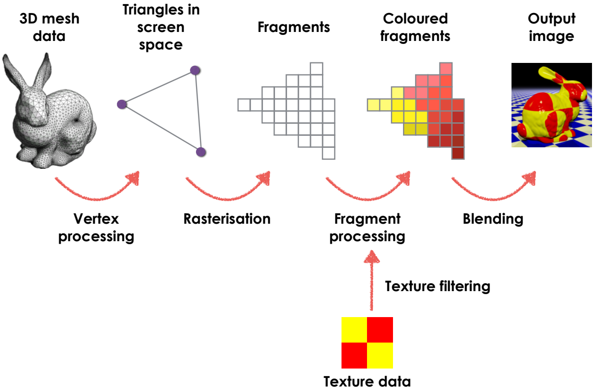
Learning outcomes

By the end of today's session, you will be able to:

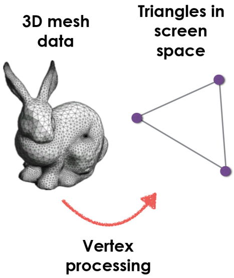
- **Recall** the key stages of the graphics pipeline
- **Understand** the GPU Debugger
- **Explain** some of the key areas for optimisation

The 3D graphics pipeline

The 3D graphics pipeline

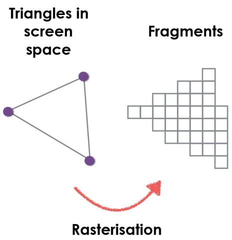


Vertex processing



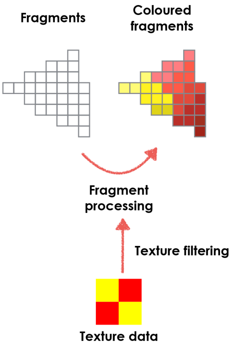
- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)
- May also apply particle simulations, skeletal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

Rasterisation



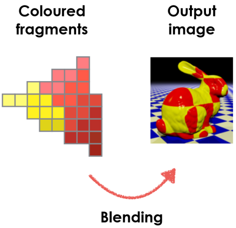
- Determine **which fragments** are covered by the triangle
- In practical terms, “fragment” = “pixel”
- Vertex processor can associate **data** with each vertex; this is **interpolated** across the fragments

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

Standard 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

Other Shaders

- **Geometry Shader**:
  - Operates on primitives
  - Can emit zero, one or more primitives
  - Usually used to expand geometry (i.e take in one point and produce a triangle)
  - Typically used for fur, hair, particle systems
- **Tessellation Control Shader**:
  - Receives input from vertex shader
  - Determines the amount of tessellation on a primitive
  - Perform any transformation on the patch data
  - Can change the size of the patch, add more vertices or fewer
  - Typically used for level of detail and stitching

Other Shaders

- **Tessellation Evaluation Shader**:
  - Relieves input from the Tessellator
  - Calculates the new vertices in the patch
  - Works in conjunction with the TC Shaders
- **Compute Shader**:
  - These types of shaders allow you to carry out general purpose computing on the GPU
  - Can access all the same data as normal shaders, exceptions are attributes
  - The shader has to write to an image or a shader storage object
  - This shader type can do simulations, AI or any other general purpose processing

GPU Profiling

Live Demo

- Render Doc
- Unity
- Unreal

GPU Optimisation

Visibility Culling

- You should always cull your scene based on the cameras view fulstrum
- This will allow to eliminate objects that are not visible
- This combined with a scene graph will allow us to cull large parts of the scene
- You should also sort all visible objects from back to front
- Caveat, transparent object should be sorted front to back

State Changes

- You should attempt to minimize state changes
- This includes
  - Changing Shaders/Materials
  - Changing Pipeline States
  - Changing Active Textures
- If you are working in an engine, try to minimum the amount of different materials
- This will allow the engine to sort the render queue based on material
- Attempt to use a texture atlas to manage your textures