COMP220: Graphics & Simulation

1: The graphics pipeline

Learning outcomes

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

- **Recall** the key stages of the graphics pipeline
- **Explain** the differences between a CPU and a GPU
- **Write** basic programs using SDL and OpenGL

Course introduction

From the module guide

This module will introduce you to the techniques of 3D graphics rendering and physics simulation used in modern computer games. Using the OpenGL library, you will develop an understanding of the 3D graphics pipeline, and how to program the GPU to produce advanced graphical effects.

Topic schedule

On LearningSpace...

Assignment 1: Portfolio task

First worksheet is due in week 4.

Assignment 2: Research journal/Mathematics Library

Work as a group on it **in parallel** to your portfolio task!
First component due in week 3.
Don't forget to update the wiki!

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

Physics processing unit

- Ageia PhysX PPU briefly appeared on the market in 2006-2008
- Similar architecture to GPU (many cores, optimised for floating point calculations)
- Ageia acquired in 2008 by Nvidia...
- Now PhysX is Nvidia's middleware for performing physics simulation on the GPU

General purpose GPU (GPGPU)

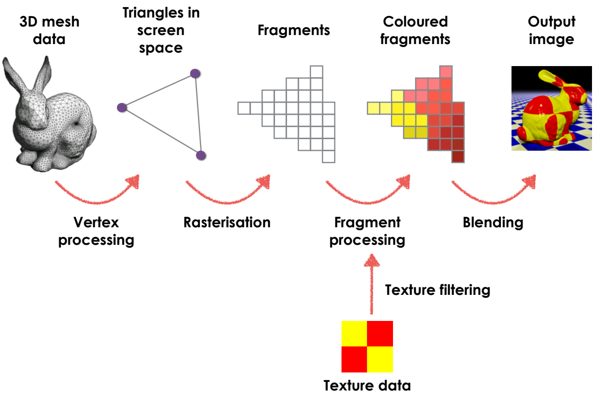
- Early GPUs used a **fixed pipeline** – could only be used for rendering 3D graphics
- Modern GPUs use a **programmable pipeline** – can be programmed for other tasks
- Physics simulation (e.g. PhysX)
- Scientific computing (e.g. CUDA)
- Deep learning

Graphics APIs

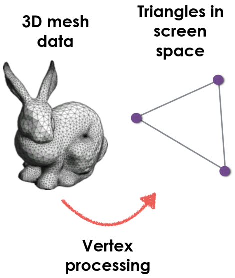
- Graphics APIs **abstract** away the differences between different manufacturers' GPUs
- There are several APIs in use today:
  - **OpenGL**: Open standard, very mature, very widely supported
  - **Vulkan**: Open standard, very new, support still growing
  - **Direct3D**: Microsoft only
  - **Metal**: Apple only
  - Sony and Nintendo consoles have their own APIs; Microsoft consoles use Direct3D
- Most general-purpose game engines (e.g. Unity, Unreal) support several graphics APIs
- On this module we will use **OpenGL** (but the principles are transferable)

The 3D graphics pipeline

The 3D graphics pipeline

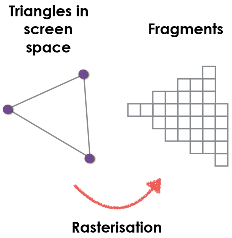


Vertex processing



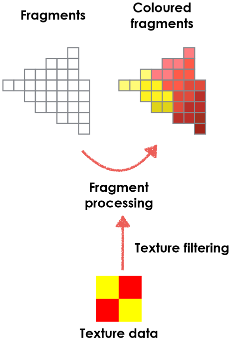
- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)
- May also apply particle simulations, skeletal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

Rasterisation



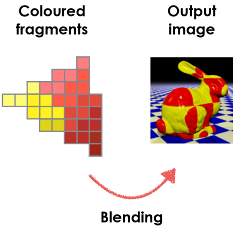
- Determine **which fragments** are covered by the triangle
- In practical terms, "fragment" = "pixel"
- Vertex processor can associate **data** with each vertex; this is **interpolated** across the fragments

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

- 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

Your first OpenGL program

SDL and OpenGL

- OpenGL only handles rendering of graphics
- We need something else to handle windows, events, audio etc
- We will use our old friend **SDL**

Live coding

`https://github.com/Falmouth-Games-Academy/bsc-live-coding`

Live coding

- Start with the basic SDL application
- Link with `opengl32.lib`
- `#include <gl/GL.h>`
- Pass `SDL_WINDOW_OPENGL` to `SDL_CreateWindow`
- Set some attributes:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, ↩
    SDL_GL_CONTEXT_PROFILE_CORE);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
```

- We don't need `SDL_CreateRenderer` any more...
- ... but we do need `SDL_GLContext`

Clearing the screen

With SDL:

```
SDL_SetRenderDrawColor(renderer, 255, 128, 0, 255);
SDL_RenderClear(renderer);
SDL_RenderPresent(renderer);
```

With OpenGL:

```
glClearColor(1.0f, 0.5f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
SDL_GL_SwapWindow(window);
```

Our first triangle

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/

Debrief

It's the end of today's session. You are now able to:

- **Recall** the key stages of the graphics pipeline
- **Explain** the differences between a CPU and a GPU
- **Write** basic programs using SDL and OpenGL

Don't forget! Portfolio task proposals due **this time next week!**