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

Proposal due **this time next week** (29th September)

Assignment 2: Research journal

Work on it **in parallel** to your portfolio task!

Assignment 2: Research journal

Work on it **in parallel** to your portfolio task!
Don't forget to update the wiki!

Graphics and simulation hardware

How computers work?

"A computer is like a small cardboard box. Inside this box lives an elf. The elf obeys instructions from my program, in the order they are given. Some instructions tell the elf to draw pictures onto the screen."

https://blogs.msdn.microsoft.com/shawnhar/2008/03/31/an-elf-in-a-box/

How computers really work

"A computer is like a cardboard box inhabited by a pair of elves, Charles Pitchwife Underhill plus his younger sibling George Pekkala Underhill (both more commonly known by their initials)."

https://blogs.msdn.microsoft.com/shawnhar/2008/03/31/an-elf-in-a-box/

How computers really work (cont.)

> *“Charles is smart, well educated, and fluent in dozens of languages (C, C++, C#, and Python, to name a few). George, on the other hand (...) finds it difficult to communicate with anyone other than his brother Charles, prefers to plan his day well in advance, and gets flustered if asked to change activities with insufficient warning. He has an amazing ability to multiply floating point numbers, especially enjoying computations that involve vectors and matrices.”*

https://blogs.msdn.microsoft.com/shawnhar/2008/03/31/an-elf-in-a-box/

How computers really work (cont.)

“When you run a program on this computer, Charles reads it and does whatever it says. Any time he encounters a graphics drawing instruction, he notes that down on a piece of paper. At some later point (when the paper fills up, or if he sees a Present instruction) he translates the entire paper from the original language into a secret code which only he and George can understand, then hands these translated instructions to his brother, who carries them out.”

https://blogs.msdn.microsoft.com/shawnhar/2008/03/31/an-elf-in-a-box/

CPUs vs GPUs

CPUs vs GPUs

- (CPU = central processing unit; GPU = graphics processing unit)

CPUs vs GPUs

- (CPU = central processing unit; GPU = graphics processing unit)
- GPUs are **highly parallelised**

CPUs vs GPUs

- (CPU = central processing unit; GPU = graphics processing unit)
- GPUs are **highly parallelised**
  - Intel i7 6900K: **8** cores

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

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**

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

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

ASUS
Rock Solid · Heart Touching
PhysX P1
PCI
PhysX by ageia
Accelerate the New Reality in Gaming Physics

Physics processing unit

ASUS
Rock Solid · Heart Touching
PhysX P1
PCI
PhysX by ageia
Accelerate the New Reality in Gaming Physics

- Ageia PhysX PPU briefly appeared on the market in 2006-2008

Physics processing unit

ASUS
Rock Solid · Heart Touching
PhysX P1
PCI
PhysX by ageia
Accelerate the New Reality in Gaming Physics

- Ageia PhysX PPU briefly appeared on the market in 2006-2008
- Similar architecture to GPU (many cores, optimised for floating point calculations)

Physics processing unit

ASUS
Rock Solid · Heart Touching
PhysX P1
PCI
PhysX
Accelerate the New Reality in Gaming Physics

- 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...

Physics processing unit

ASUS
Rock Solid · Heart Touching
PhysX P1
PCI
PhysX
Accelerate the New Reality in Gaming Physics

- 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)

General purpose GPU (GPGPU)

- Early GPUs used a **fixed pipeline** – could only be used for rendering 3D graphics

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

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)

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)

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

- Graphics APIs **abstract** away the differences between different manufacturers' GPUs

Graphics APIs

- Graphics APIs **abstract** away the differences between different manufacturers' GPUs
- There are several APIs in use today:

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

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

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

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

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

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

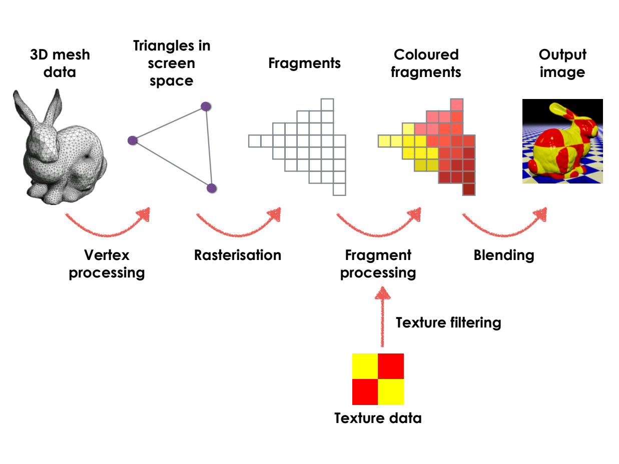
Graphics APIs

- 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)

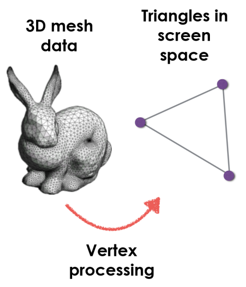
FALMOUTH
UNIVERSITY

The 3D graphics pipeline

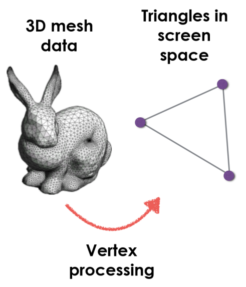
The 3D graphics pipeline



Vertex processing



Vertex processing



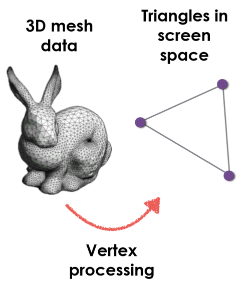
3D mesh data

Triangles in screen space

Vertex processing

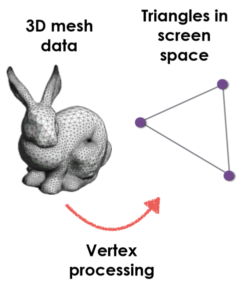
- Geometry is provided to the GPU as a **mesh** of **triangles**

Vertex processing



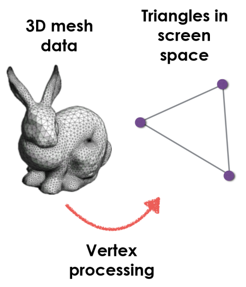
- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)

Vertex processing



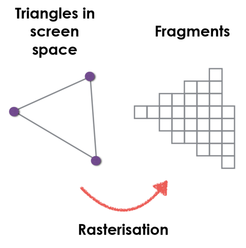
- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)

Vertex processing

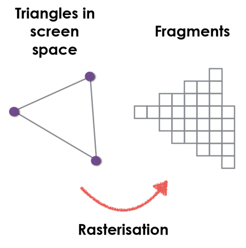


- Geometry is provided to the GPU as a **mesh** of **triangles**
- Each triangle has three **vertices** specified in 3D space (x, y, z)
- Vertex processor **transforms** (rotates, moves, scales) vertices and **projects** them into 2D screen space (x, y)
- May also apply particle simulations, skeletal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

Rasterisation

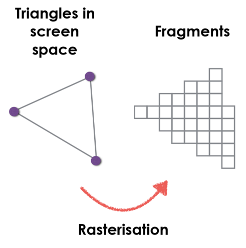


Rasterisation



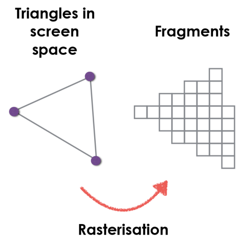
- Determine **which fragments** are covered by the triangle

Rasterisation



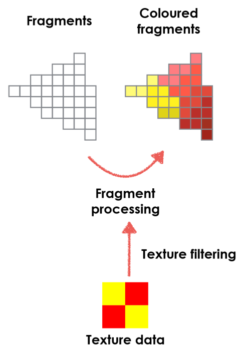
- Determine **which fragments** are covered by the triangle
- In practical terms, “fragment” = “pixel”

Rasterisation

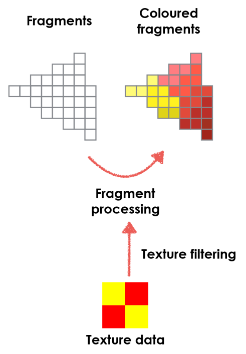


- Determine **which fragments** are covered by the triangle
- In practical terms, “fragment” = “pixel”
- Vertex processor can associate **data** with each vertex; this is **interpolated** across the fragments

Fragment processing

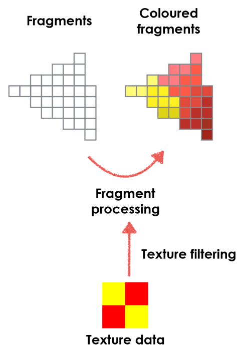


Fragment processing



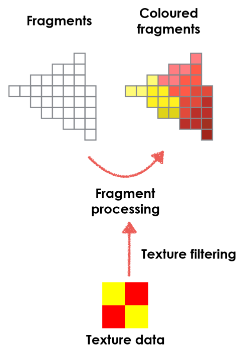
- Determine the **colour** of each fragment covered by the triangle

Fragment processing



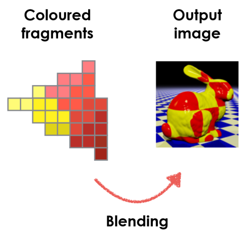
- Determine the **colour** of each fragment covered by the triangle
- **Textures** are 2D images that can be **wrapped** onto a 3D object

Fragment processing

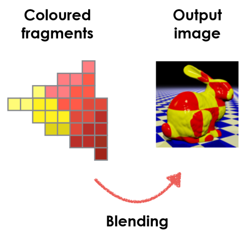


- Determine the **colour** of each fragment covered by the triangle
- **Textures** are 2D images that can be **wrapped** onto a 3D object
- Colour is calculated based on **texture**, **lighting** and other properties of the surface being rendered (e.g. shininess, roughness)

Blending

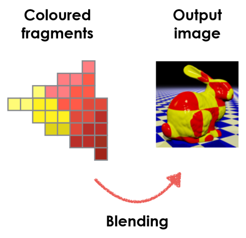


Blending



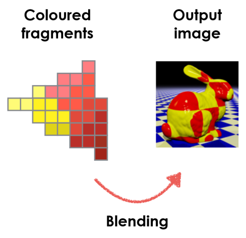
- Combine these fragments with the existing content of the image buffer

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

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

Your first OpenGL program

SDL and OpenGL

SDL and OpenGL

- OpenGL only handles rendering of graphics

SDL and OpenGL

- OpenGL only handles rendering of graphics
- We need something else to handle windows, events, audio etc

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

Live coding

- Start with the basic SDL application

Live coding

- Start with the basic SDL application
- Link with `opengl32.lib`

Live coding

- Start with the basic SDL application
- Link with `opengl32.lib`
- `#include <gl/GL.h>`

Live coding

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

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);
```

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...

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

Clearing the screen

With SDL:

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

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!**