# Computer Graphics

COMP3421/9415
2021 Term 3 Lecture 3

# What did we learn last week?

**Graphics in a Nutshell**

- History of Modern Computer Graphics
- What's in the Course
- Graphics Hardware (monitors and graphics cards)
- Polygon Rendering overview
- Course coding platform

# What are we covering today?

**2D Graphics**

- Continuing our learning about Polygon Rendering in 2D
- The OpenGL Pipeline
- Colouring shapes with shaders
- Textures

# The OpenGL Pipeline

# Going from Data to Pixels

**Last week, we looked at the Polygon Rendering Process . . .**

**Today, we go into more detail!**

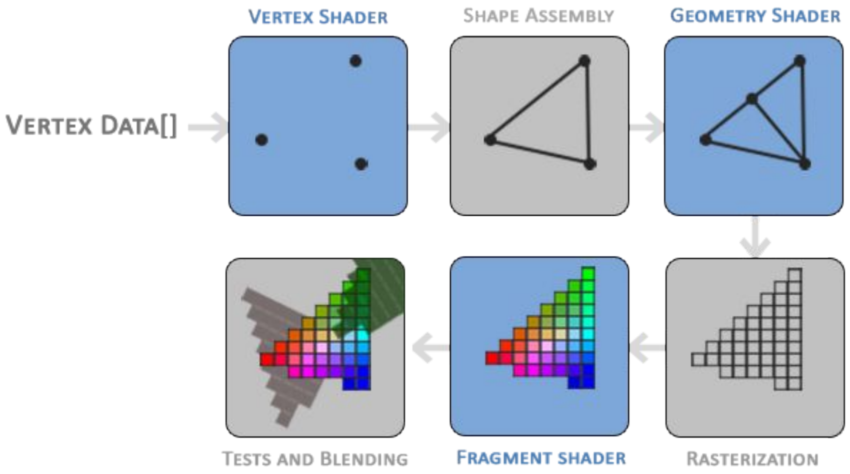



Image credit: learnopengl.com

# A step by step process

**A breakdown of the OpenGL Pipeline**

1. Vertex Data is passed to OpenGL
2. Vertex Shader
3. Shape Assembly
4. Geometry Shader (not covered in this course)
5. Rasterization
6. Fragment Shader
7. Tests and Blending (we'll look at this in later lectures)

# Before the OpenGL Pipeline

**What are our shapes?**

- In our CPU Code
- We will build up information first (like a vertex vector)
- . . . then pass it to OpenGL
- Each vertex can have a position vector (x,y,z coordinates)
- Also colours! (Red, Green, Blue)
- And more . . .

# How does OpenGL receive our data?

**Buffers and Arrays**

- We give information as a big collection of vertices
  - This is very similar to an array in memory
- But we tend to dump it all in at once!
- How do we organise it into separate vertices?
- How much data is in one vertex? It varies!

- **Vertex Buffer Object** - can store many vertices
- **Vertex Attributes** - split up a single vertex into different information

# Vertex Attributes

## Each Vertex takes up a certain amount of memory

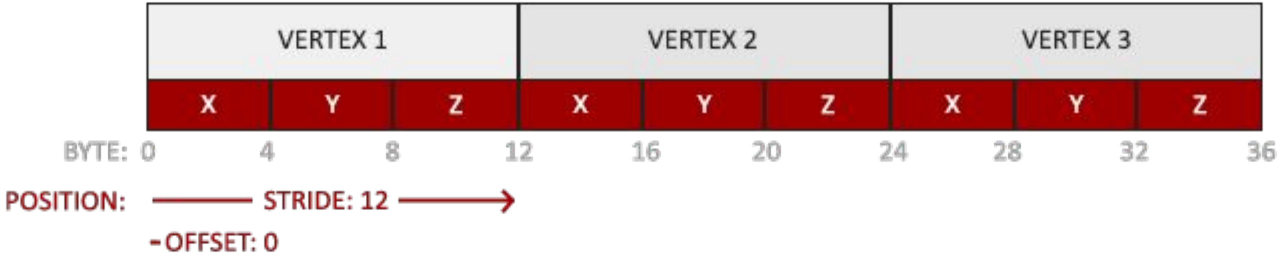



Image credit: learnopengl.com

- Attributes are things like coordinates, colours and other information
- Each attribute is somewhere in the vertex's memory
- We can tell OpenGL how big a vertex is and where in each vertex's memory each attribute is

# Vertex Array Object

**We end up with a group of Vertex Attribute Pointers**

- These allow us to reach each attribute in a vertex
- We're also going to want to treat all the vertices in a buffer the same
- We end up with a **Vertex Array Object** which can be applied to every vertex in a particular **Vertex Buffer Object**

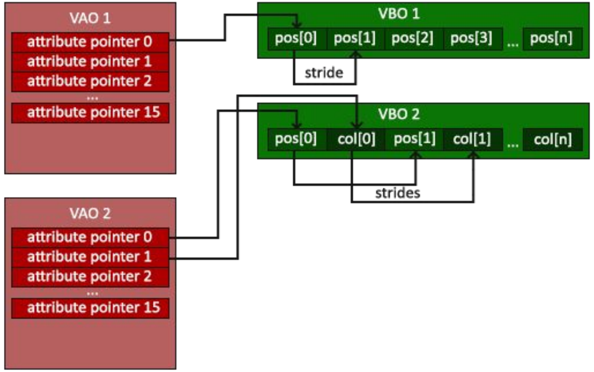



Image credit: learnopengl.com

# The Vertex Shader

## Giving Shape Information to the Graphics Card

- The Vertex shader works on one vertex at a time
- Each vertex will end up with a position (xyz coordinates)
- These might be different from what we provided (we'll learn more about this later)
- Some processing of colour information will happen

# Shape Assembly

**We never explicitly code edges between vertices**

- Edges don't exist, only vertices
- But how we connect them together is very important!
- OpenGL will take our list of vertices and convert it into triangles

# A Vector of Vertices

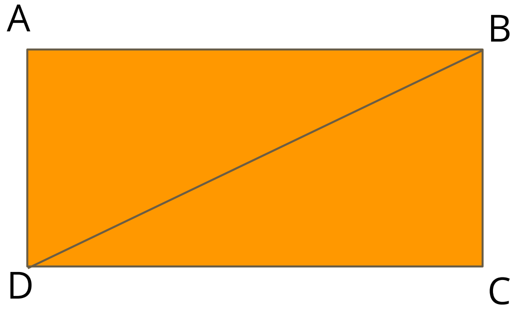
**Is it enough to give a big list of vertices?**

- Can you make shapes if all you have is a list of vertices?
- Technically yes?
- Is this a good idea?
- Let's look at a simple example . . .

# A Rectangle

**I want to make a simple object**

- Give a list of vertices to OpenGL so that it makes two triangles that form a rectangle
- {A,B,D,D,B,C}
- This works . . . we get two triangles
- But why do we have 6 vertices when there are obviously only four corners?
- This is wasting memory in our VBO



# Element Buffer Objects

**Let's reuse vertices instead of copying them**

- An array of vertices: {A,B,C,D}
- A triangle is an array of three indices into this array
- Our two triangles: {0,1,3,3,1,2}
- This array is an **Element Buffer Object**
- Significant reduction in the number of vertices needed
- Allows shared vertices to only exist once
- The element buffer of ints is much cheaper than an array of vertices

# Rasterization

**Conversion into grids of pixels**

- Taking shapes built from vertices
- Turning them into **fragments**, which correspond to pixels on the screen
- But they have more information like knowing which vertices make up their shape (nearly always a triangle)

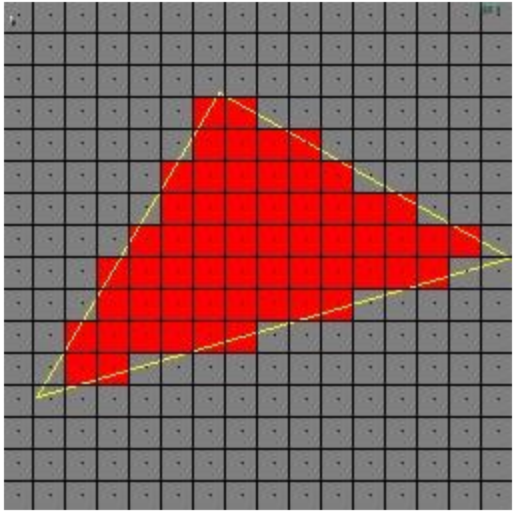


Image credit: Nvidia

# Fragment Shader

**A fragment is the information necessary to create a pixel**

- Calculates the final colour of a pixel
- Knows about vertex data in the shape
- But will also know things like lights in a scene (we'll be spending weeks on this later!)
- This information all gets written to the **Frame Buffer** containing colours
- The Frame Buffer is like a 1:1 mapping to the pixels in the monitor

# Break Time

**Assignment 1 has been released!**

- Yes, it's a test to see whether you've done all the tutorials :P
- Also a chance to stretch your creativity with the techniques we've taught
- Due on the 1st October at 5pm

# Colouring Shapes with Shaders

# How do we decide the colour of a pixel?

**We're using our Shaders!**

- Vertices can have a colour (a vector of floats using RGBA)
- Red, Green, Blue, Alpha(transparency, which we're not using yet)
- Vertex Shaders can specify a colour output
- Fragment Shaders can take that input and use it

# Colour Attributes in Vertices

## We're adding information to Vertices

- This means each vertex needs attribute pointers
- One to the 3 float vector of location
- Another to a 3 float set of RGB values for colour

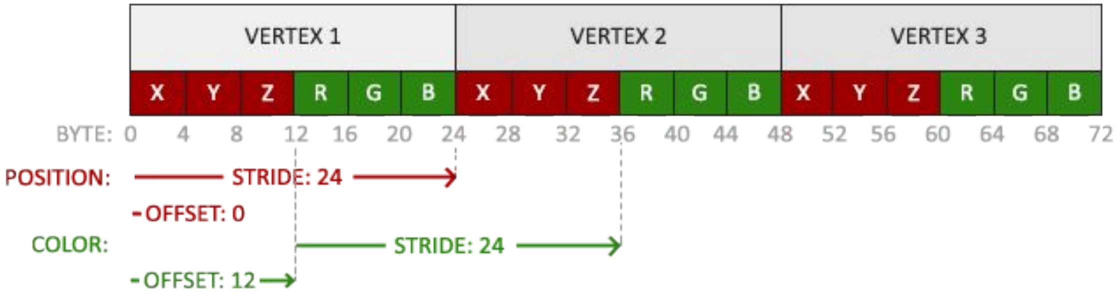



Image credit: learnopengl.com

# Fragment Interpolation

## Fragment Shaders and their tricks

- Each fragment exists somewhere between vertices
- Instead of just taking the colour from one of the vertices
- The fragment shader will interpolate values from all the vertices based on its position in the shape

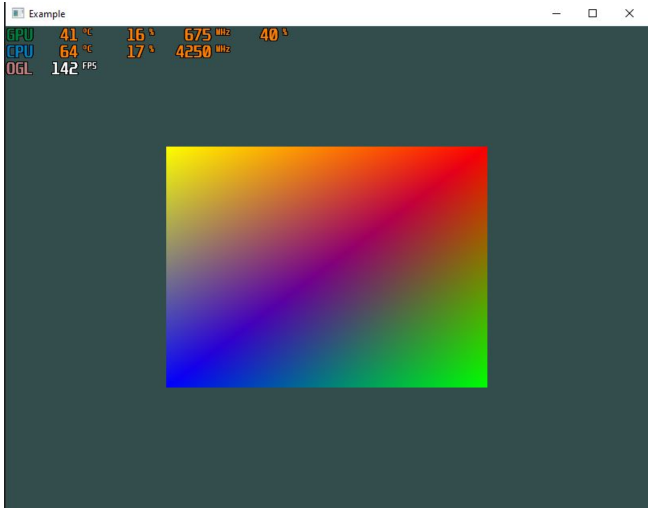



Image credit: Marc Chee (using course example code)

# Textures

# Textures are Images!

**Games before Polygon Rendering were often "sprite" based**

- Sprites are images that can be moved around the screen
- It's like putting an image on a rectangle in our rendering

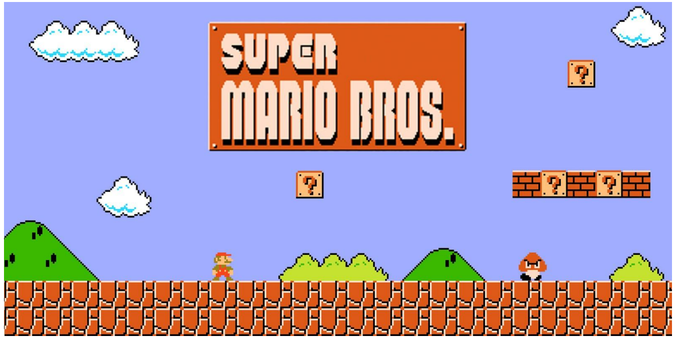



Image credit: Nintendo


Image credit: Capcom (edited by Marc)

# Textures on Surfaces in 3D

**3D Objects can have images wrapped around them**

- Shows surface detail that doesn't need extra vertices or triangles
- We can show details like faces, or surfaces like grass or brick walls
- Having lots of vertices and triangles is expensive (computationally)
- Textures can be included in the render pipeline!


Images credit: id Software

# Textures on Triangles

**Starting with the basics**

- We can provide OpenGL with a Texture (image file)
- We then "map" the vertices in our shape to coordinates in the image
- The fragment shader can interpolate each fragment's position
- The colour from the texture is "sampled" to give the pixel its colour
- More on this next lecture . . .

# What did we learn today?

**Details on Rendering**

- The OpenGL Pipeline (a first look)
- Some details on code constructs
  - Vertex Buffer (VBO), Vertex Attributes (VAO), Element Buffer (EBO)
- Shaders in the pipeline
- An intro to Textures