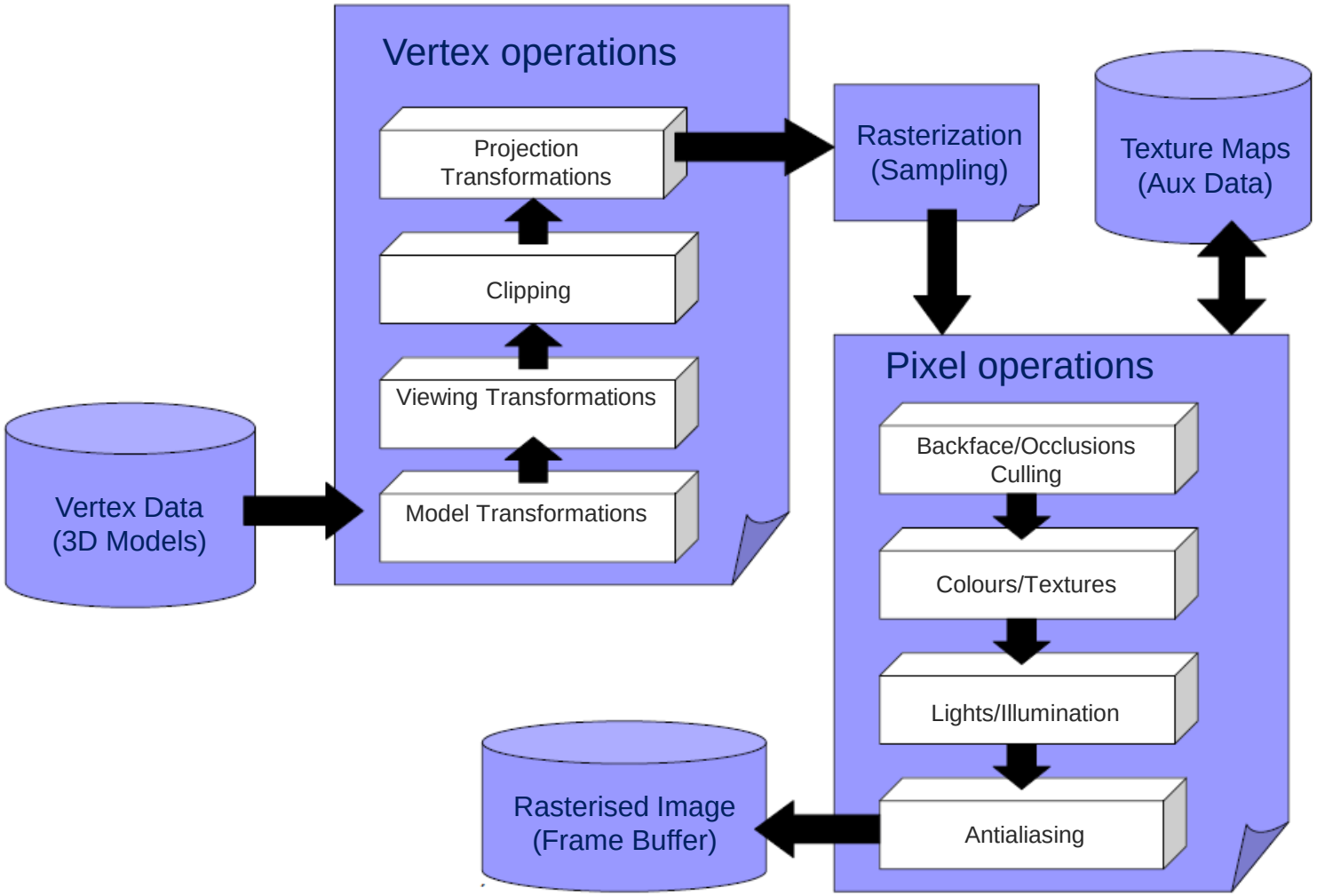
Introduction to DirectX

Wayne Rippin (UoD)
Dr Panagiotis Perakis (MC)

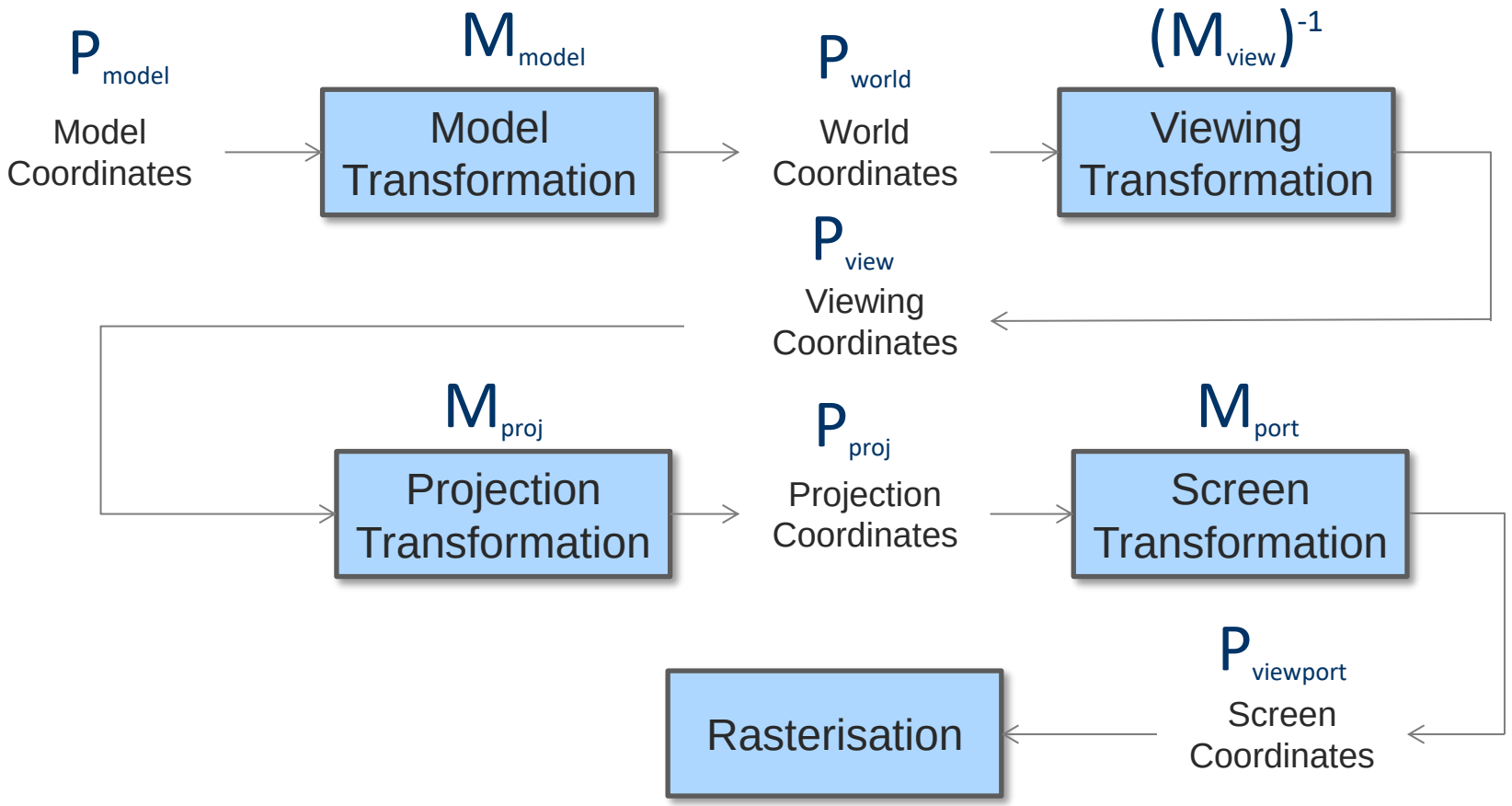
Review of Graphics Part I

- Wrote your own graphics renderer. This included:
  - Vectors & Matrices
  - Vector and matrix operations
  - Vertex and normal buffers
  - Models and model data structures
  - Transformations
  - Different rendering routines – wireframe, flat and smooth shading, and texturing
  - Lighting models

The Typical Graphics Pipeline

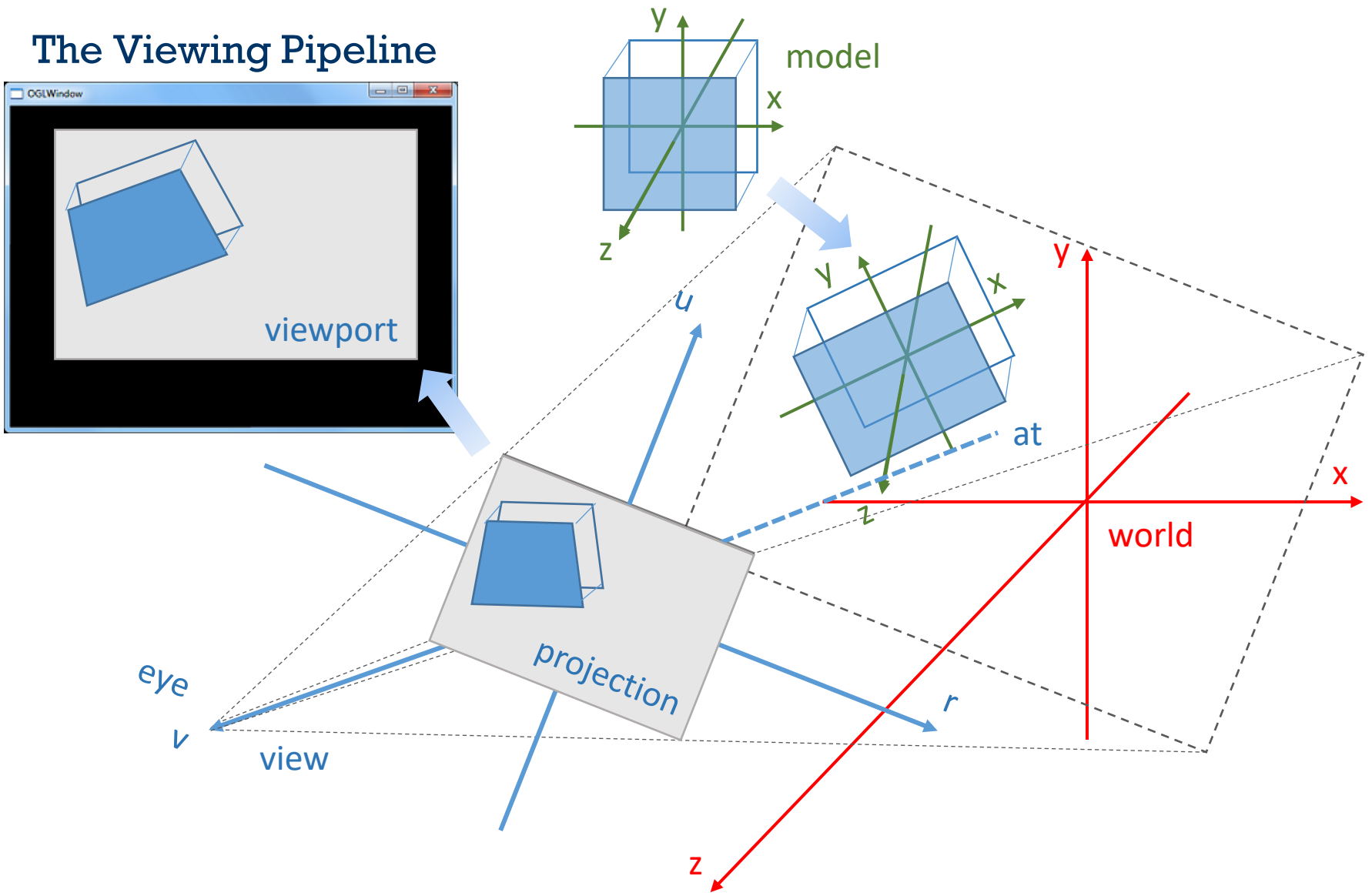


The Viewing Pipeline



$$P_{viewport} = M_{port} * M_{proj} * (M_{view})^{-1} * M_{model} * P_{model}$$

The Viewing Pipeline



Graphics Languages & Tools

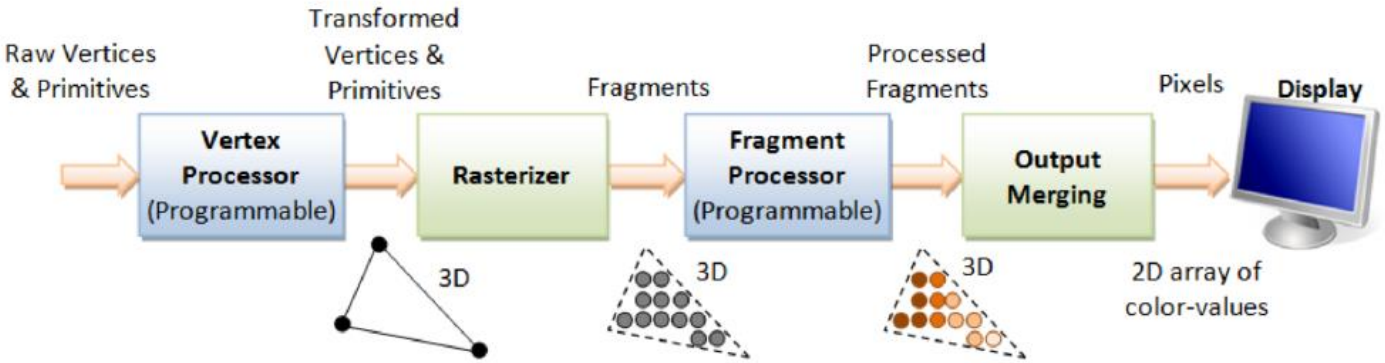
- OpenGL/GLSL
  - C/C++-like implementation of the Graphics pipeline
    - Cross-platform (Windows/Unix/Linux/Mac)
- Direct3D/DirectX
  - Windows GDI
    - Windows compatible.
- CUDA
- Unity
  - Game Engine
    - Cross-platform/Supports OpenGL/Direct3D.

Let Hardware take the Strain

- The key benefit of graphics APIs such as DirectX, OpenGL, etc is that they allow the programmer to offload work to processor(s) on the video card –the Graphics Processing Unit (GPU)
- Can be multiple cores allowing many operations to be done in parallel
- If the GPU was programmed directly, the code would be very complex and would require different code for each type of graphics processor.
- A graphics API provides the core functionality for you via a relatively high-level API
- Drivers are provided for different video cards enabling your applications to take advantage of hardware acceleration where it is available

The Modern Graphics Pipeline

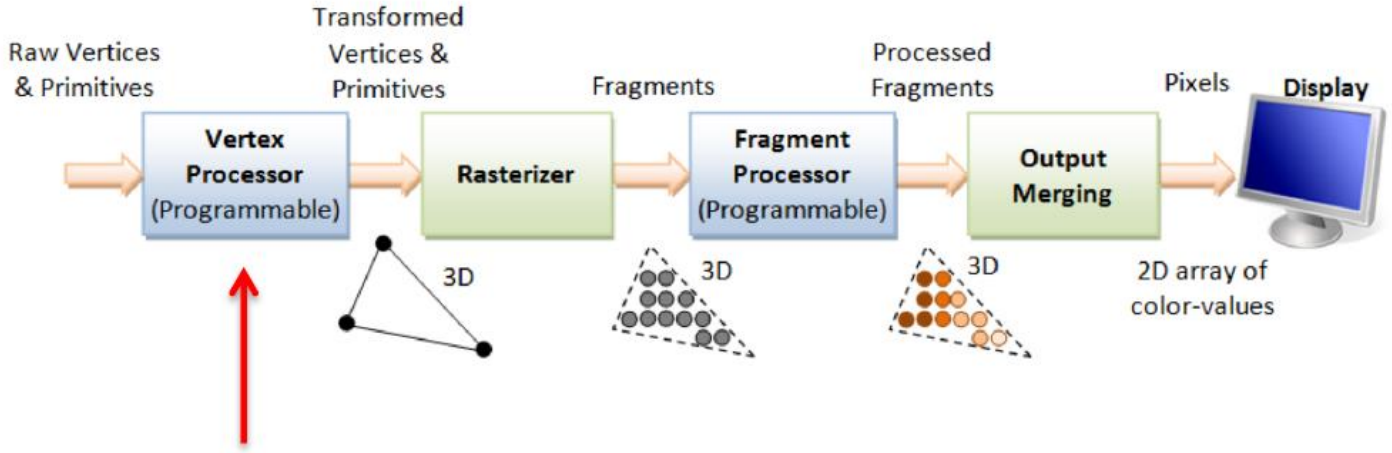
Programmable Rendering Pipeline (GPUs @ late 90s)



1. Vertex Processing: Process and transform individual vertices & normals.
2. Rasterization: Convert each primitive (connected vertices) into a set of fragments. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.
3. Fragment Processing: Process individual fragments.
4. Output Merging: Combine the fragments of all primitives (in 3D space) into 2D color-pixel for the display.

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

The Modern Graphics Pipeline

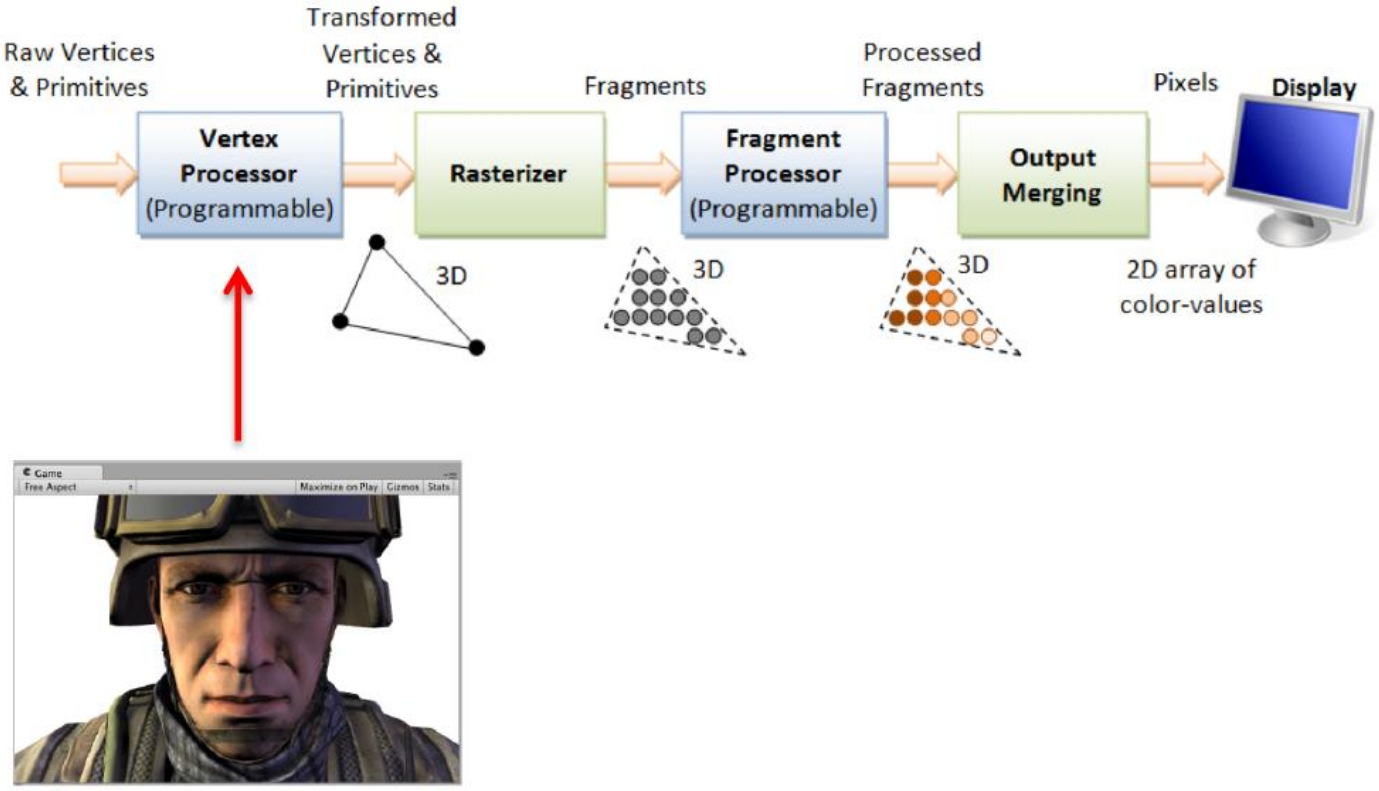


vertex shader

- transforms & (per-vertex) lighting

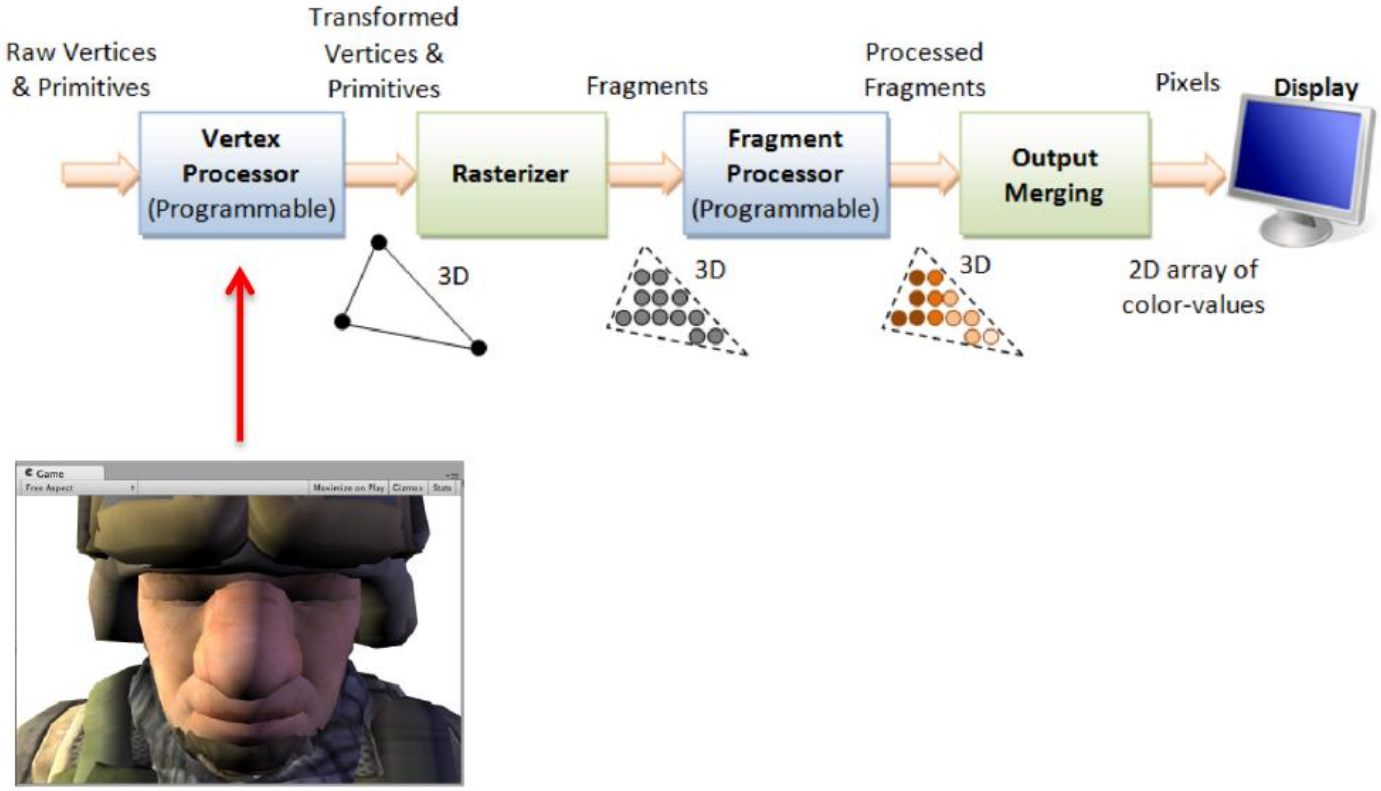
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

The Modern Graphics Pipeline



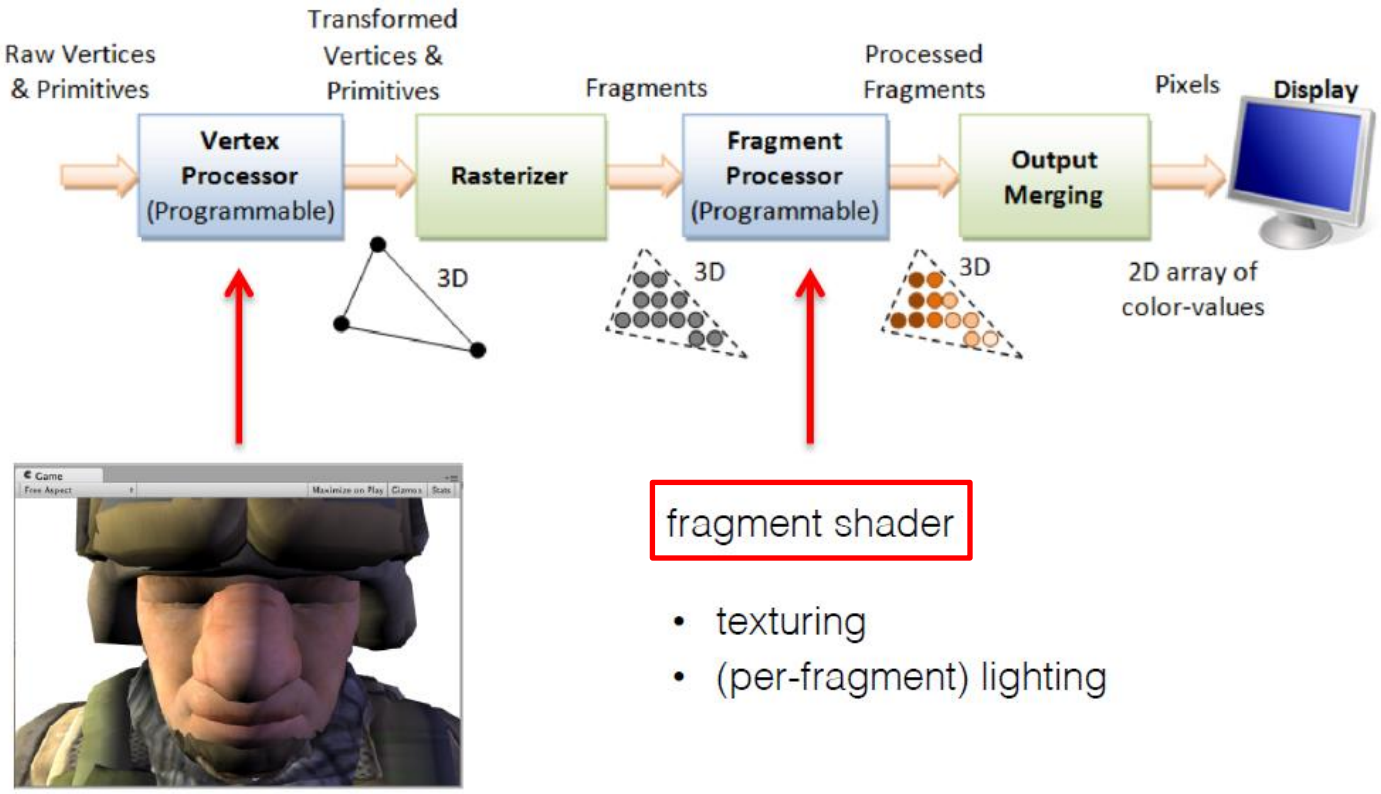
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

The Modern Graphics Pipeline



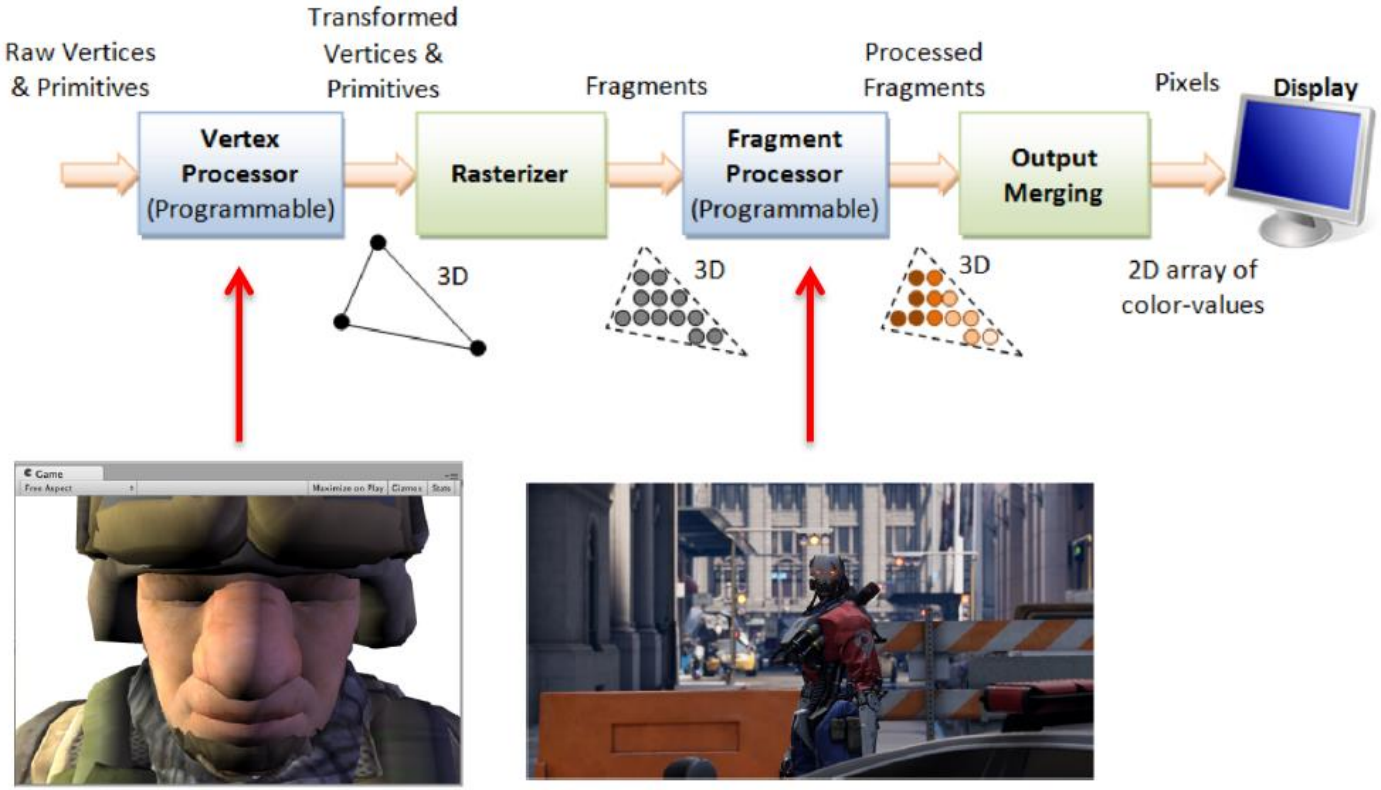
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

The Modern Graphics Pipeline



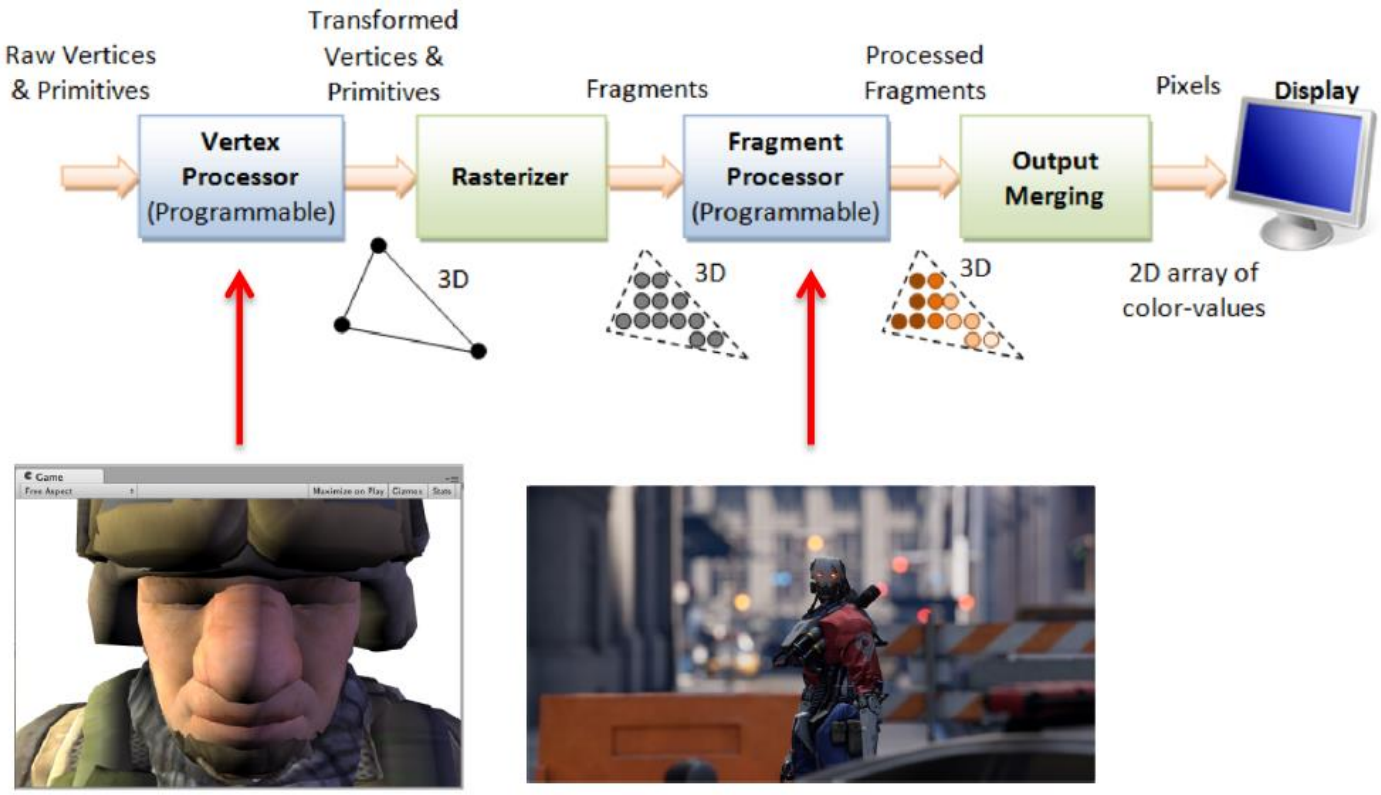
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

The Modern Graphics Pipeline



https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

The Modern Graphics Pipeline



https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

What is DirectX?

- DirectX is an umbrella term for a series of APIs that include the following that we will look at:
  - Direct3D
    – A 2D and 3D graphics API
  - XAudio
    – Sound API
  - XInput
    – For handling input devices
- Each of these are at different levels and over time, different APIs are discontinued and/or replaced.

Which Version of Direct3D?

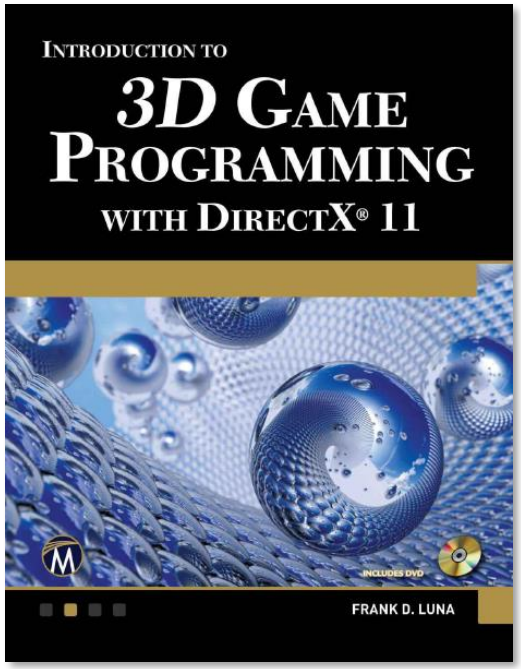
- We are going to use Direct3D 11
- Why?
  - Isn't Direct3D 12 the latest API?
  - Why are we not using the latest API?

Be Aware - The search engines are not necessarily your friend (and neither is ChatGPT, CoPilot, etc)

- Be aware:
  - Although Direct3D 11 has been out for over 10 years, some of the supporting APIs have changed:
    - You will find references to the xnamathlibrary. This has now changed to DirectXMath
    - You will find references to D3DX functions. These have now been deprecated and removed
    - You will find references to DirectX 9, 10 and 12.

Reference Books

- The same problem exists with books about Direct3D 11. Although there is good information in them, we have not found one yet without errors. So you can get ideas from them, but don't follow the code examples faithfully.
- The best book we have found is:
  - '3D Game Programming with DirectX 11' by Frank D. Luna
  - As mentioned above, the code examples are now out of date, but we will give you up-to-date versions of many of the examples during this module. However, the book does explain a lot of the DirectX concepts very well.
- Use the examples we give you as a starting point for current practice for Direct3D 11 and you will find things a lot easier.



Getting Started with Direct3D 11
Basic Terminology

- Some basic terms:
  - The Direct3D rendering pipeline
  - The swap chain
  - Driver types
  - Feature levels

The Direct3D 11 Rendering Pipeline

Many of these will not be used in this module

The Stages of the Rendering Pipeline

The Input Assembler stage (IA) reads primitive data (points, lines and/or triangles) from buffers supplied by the programmer and assembles the data into primitives that will be used by the other pipeline stages…

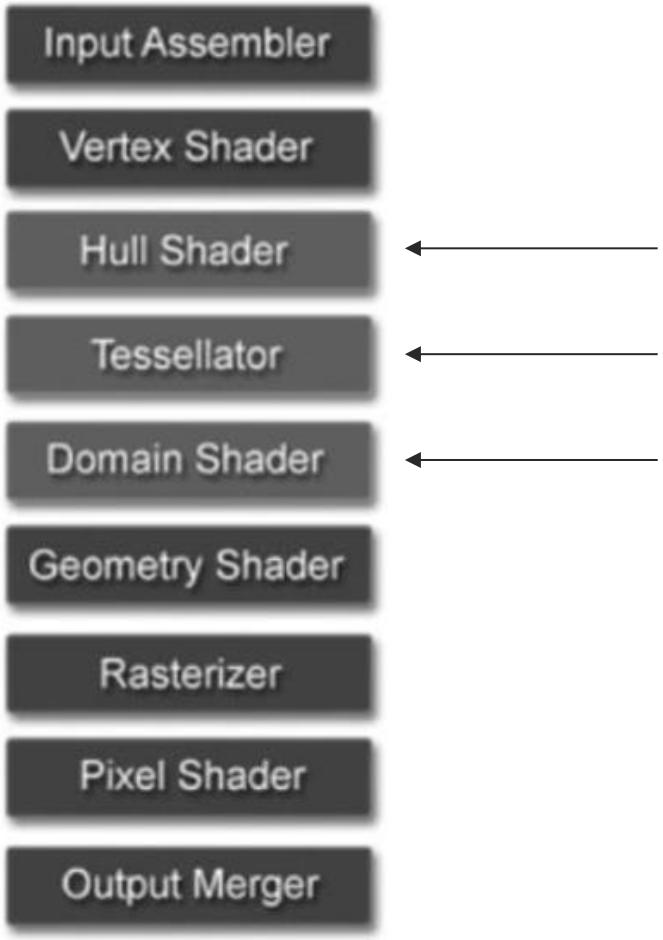
The Stages of the Rendering Pipeline

The Vertex Shader (VS) stage processes vertices from the input assembler, performing per-vertex operations (such as transformations and per-vertex lighting/shading).

This stage uses a shader that is written in HLSL (High Level Shader Language)

Vertex shaders always operate on a single input vertex and produce a single output vertex.

The Stages of the Rendering Pipeline



The Hull Shader, Tessellator and Domain Shader stages are optional stages that deal with tessellation.

Basically, hardware tessellation is the process of taking input geometry and increasing or decreasing its level of detail.

This allows for very high polygonal models to be rendered in real time with polygon counts in the hundreds of thousands or even the millions.

By having the hardware create the detail of the geometry, the application only has to submit a small amount of data that defines the low-level model.

We do not plan to cover these stages in this module.

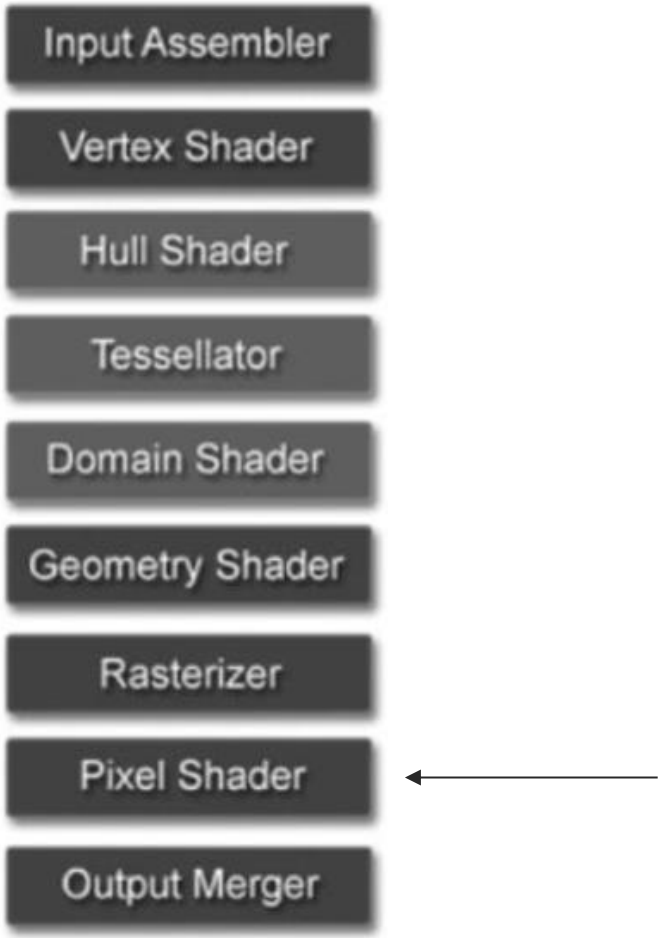
The Stages of the Rendering Pipeline

Geometry Shader (GS) stage: This is also an optional shader stage. If there is no tessellation being performed, the geometry shader stage occurs after the vertex shader stage.
Geometry shaders operate on entire shapes such as triangles, whereas the vertex shader operates on a single point of a shape.
The geometry shader has the ability to create or destroy geometry as needed, which depends largely on the effect you are trying to create.
One example is the generation of particles used to create particle effects such as rain or explosions by taking a list of points that act as the center of the particles and generating polygons around them.

The Stages of the Rendering Pipeline

Rasterizer Stage (RS): This has the job of determining what pixels are visible through clipping and culling geometry, setting up the pixel shaders and determining how the pixel shaders will be invoked.
The Rasterizer Stage is responsible for determining whether pixels should be clipped because they are not visible and the default colour value for each pixel.

The Stages of the Rendering Pipeline



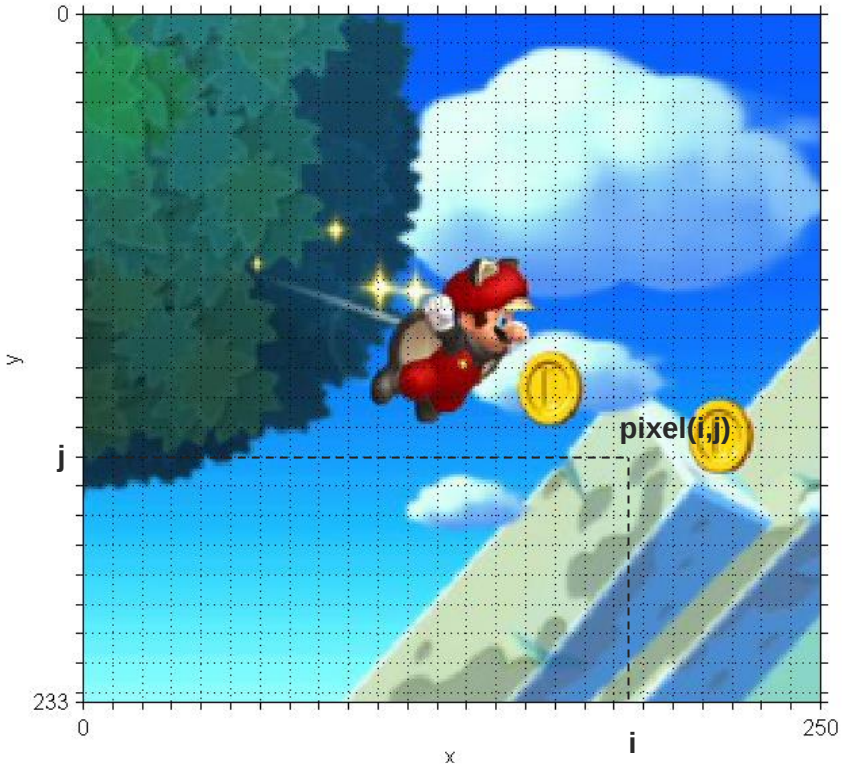
Pixel Shader (PS) stage: Uses a pixel shader written in HLSL.
In the pixel shader stage, the shader receives the geometric data from all previous stages and is used to shade the pixels that comprise that shape.
The output of the pixel shader is a single colour value that will be used by the final stage to build the final image displayed to the screen.
If there are no tessellation or geometry shaders, the pixel shader receives its input from the vertex shader directly.
By default, the input to the pixel shader is the interpolated value between the vertices of the shape.

The Stages of the Rendering Pipeline

Output Merger (OM) stage: The OM takes all of the output pieces from the other stages of the pipeline and builds up the final image to send to the screen.

The Raster Image

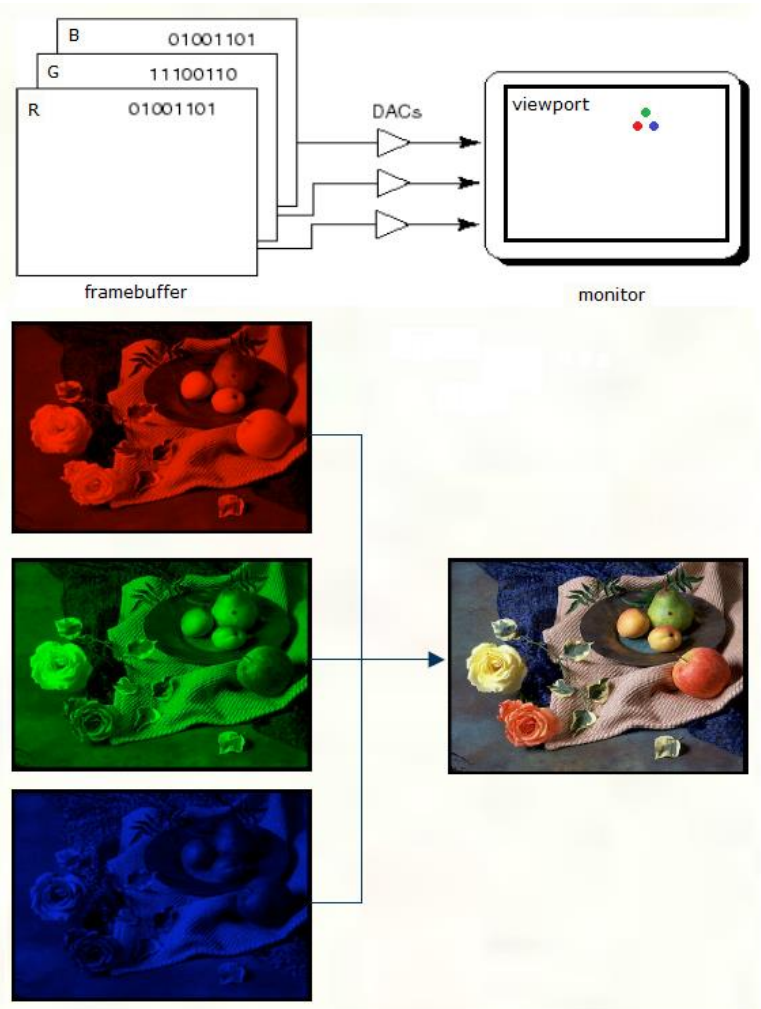
- A computer-generated image is also referred to as Raster Image and the process is called Rasterisation.
- Structurally, a raster image is a 2D array of cells known as picture elements or pixels.
- Each pixel is uniquely identified using a pair of indices (i,j).
- In an RGB colour image, each pixel stores the red, green, blue component.
- For an image of size w x h pixels we need w x h x bpp storage space, where bpp = bits per pixel, i.e. the Colour Depth.



Source: A screenshot from New Super Mario Bros. (www.nintendo.co.uk)

The Framebuffer

- A Framebuffer is a region of memory reserved for storing computer generated images.
- This is the final destination of a rendering pipeline before images are displayed.
- Modern graphics hardware has multiple framebuffers, e.g. colour, depth, stencil, accumulation etc.
- A colour buffer now supports colour depth 128bit per pixel, i.e. 32bit for each component RGBA (red, green, blue, alpha).



The Swap Chain

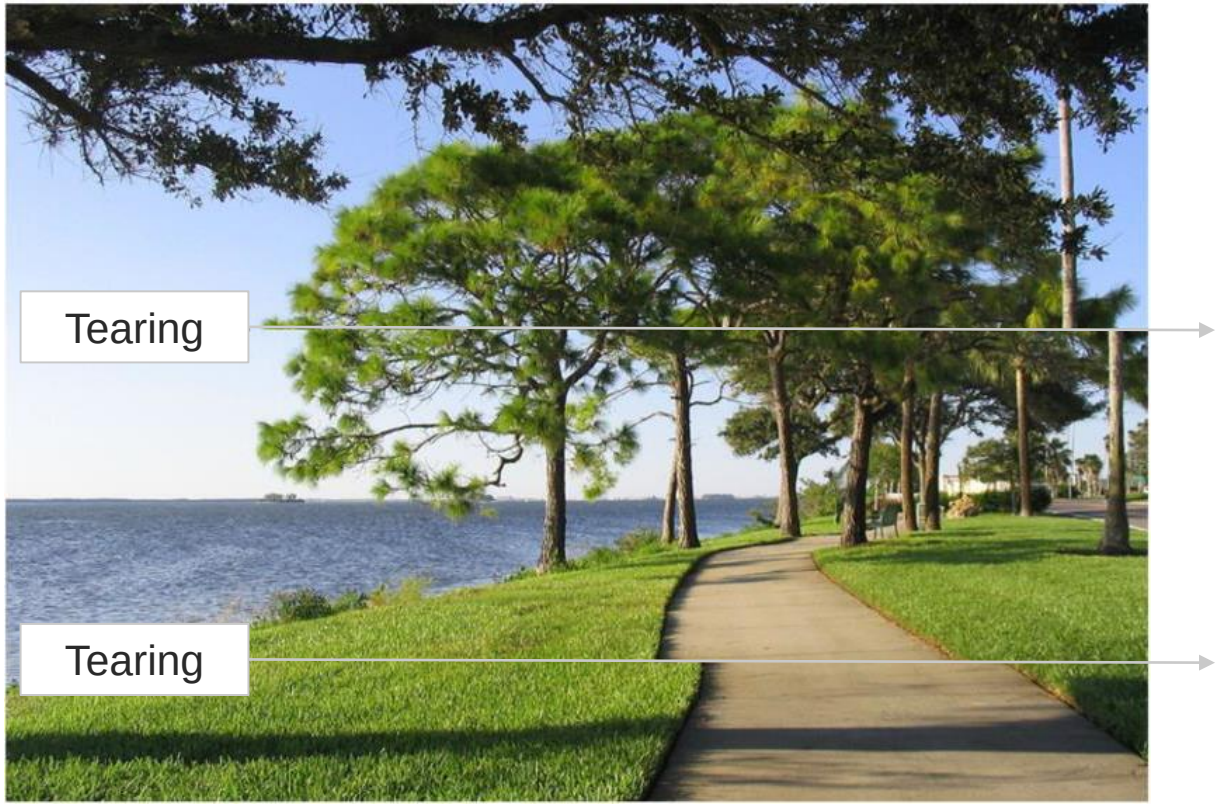
- A Graphics Processing Unit (GPU) contains a pointer in its memory to a buffer of pixels that contains the image currently being displayed on the screen. When you need to render something, such as a 3D model or image, the GPU updates this buffer and sends the information to the monitor to display. The monitor then redraws the screen replacing the old image with the new.
- However, there is a problem with this in that the monitor does not refresh as fast as needed for real-time rendering. Most refresh rates range from 60 Hz (60 frames per second) to about 100 Hz. If another model were rendered by the GPU while the monitor was refreshing, the image displayed would be cut in two, the top portion containing the new image and the bottom portion containing the old. This effect is called tearing.

Frame Rates

- Eye perceives smooth motion at >= 30 fps
- Monitor needs refreshing at constant rate (typically 60 fps)
- Application produces frames as fast as it can
  - No point going beyond monitor refresh rate
- Display controller (typically on GPU)
  - Reads frame buffer and feeds monitor at 60 fps

Tearing

- Tearing: when application writes to frame buffer at the same time the display controller is reading

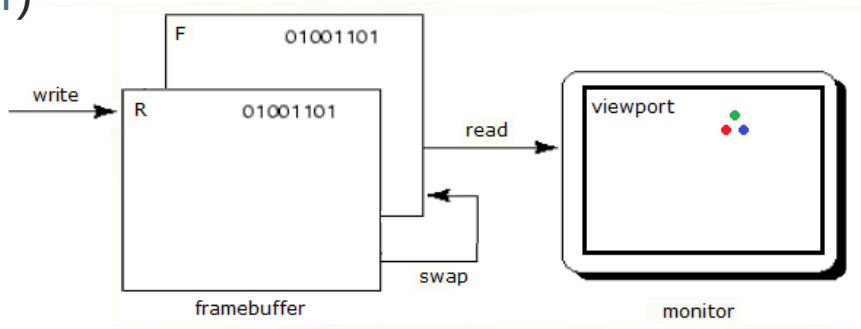


The Swap Chain

- To avoid this, Graphics APIs implement a feature called swapping. Instead of rendering new images directly to the monitor, images are drawn onto a secondary buffer of pixels, called the back buffer. The front buffer would be the buffer currently being displayed. You draw all your images onto the back buffer and when you are done, you update the front buffer with the contents of the back buffer, discarding the old image.
- If needed for a particular application, you can have multiple back buffers. This setup is called the swap chain, as it is a chain of buffers, swapping positions each time a new frame is rendered. We will be using a depth back buffer, along with a front depth buffer (also known as the Z buffer).

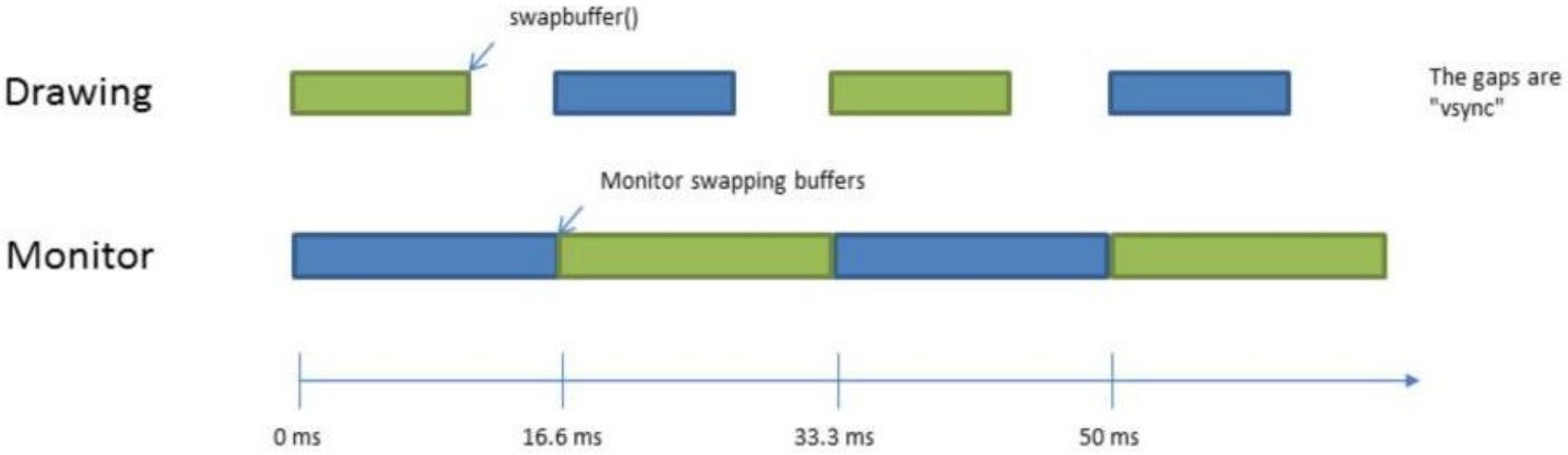
Double-Buffering

- Framebuffers are double (Double-Buffering).
- One buffer for writing (rear or back buffer) and one for reading (front buffer).
- Rasterisation is done on the "rear buffer"
- When rasterization is completed image data are transferred to the "front buffer" and are displayed on the screen.
- This process is referred as buffer-swapping.



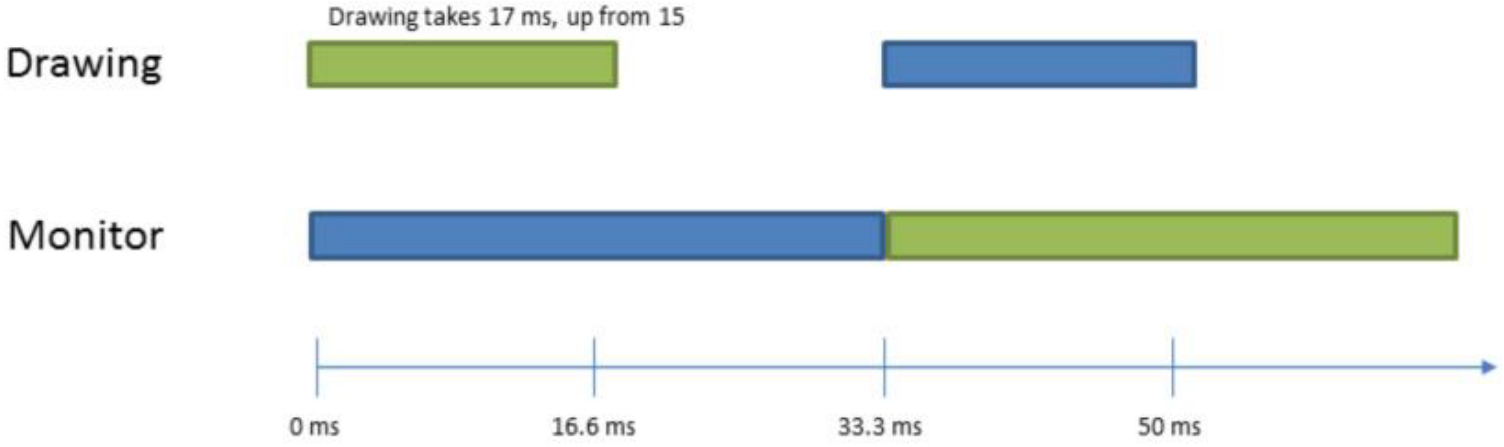
Double Buffering and Buffer swapping

- Avoids tearing
  - Front buffer is read by display controller (every 16.6ms for 60fps)
  - Back buffer is written by application



Double Buffering and Buffer swapping

- Buffer swap rate is an integer multiple of monitor's refresh rate, i.e. of 16.6ms
  - If drawing takes 17ms, buffer must swap every 33.3ms



Aliasing


© 2010-2013 SQUARE ENIX CO., LTD. All Rights Reserved. FINAL FANTASY XIV

Multi-Sampling and Antialiasing

Source: https://mynameismjp.wordpress.com/2012/10/24/msaa-overview/

Main Driver Types

Direct3D 11 support 3 graphics driver implementations:

- D3D_DRIVER_TYPE_HARDWARE:
  - The GPU hardware supports the required feature levels.
- D3D_DRIVER_TYPE_WARP:
  - A highly optimised software driver that supports all Direct3D 11 features. It makes use of whatever hardware support is available.
- D3D_DRIVER_TYPE_SOFTWARE:
  - The driver is implemented completely in software. We don't want this one since the performance is just too slow.

Feature Levels

- Indicates which levels of Direct3D are required by the application
  - We only specify D3D_FEATURE_LEVEL_11_0., because we only want Direct3D 11.0 features for now.

DirectX COM (Component Object Model)

- DirectX is provided as a series of COM components
- Enables DirectX to be language-independent and provides for backwards compatibility
- Rather than using the C++ *new* keyword, we obtain pointers to COM interfaces via calls to specific functions

COM (Component Object Model)

- All COM components are accessed via interfaces that inherit from the IUnknown interface,
- The IUnknown interface has the following methods:

Method	Description
AddRef	Increments the reference count for an interface on an object.
QueryInterface	Retrieves pointers to the supported interfaces on an object.
Release	Decrements the reference count for an interface on an object.

- When using DirectX components, methods are provided that retrieve the interface for us and do an AddRef.
  - See the DeviceDetectionWithoutComptr example.

COM (Component Object Model)

- Once we a have pointer to a COM interface, we can use that interface to call other functions/methods
- Once we are done with an interface, we **must** remember to call its Release method (inherited from IUnknown)
  - If you don't Release the object, you will find your application does not terminate!
- All COM components handle their own memory management
  - If an interface is not released, there will be significant memory leaks
- The problem is that it is really easy to forget to Release an object.
  - We are going to be using a lot of COM interfaces and remembering to release each one is a chore.

Smart Pointers in C++

- Recent standard updates to C++ have introduced smart pointers to the C++ standard library. These are intended to make the use of pointers in C++ easier since they remove the need to remember to delete objects that have been created using new.
- These new smart pointers include the types unique_ptr and shared_ptr.

ComPtr

- ComPtr is a *smart pointer* type that represents the interface specified by the template parameter
- ComPtr automatically maintains a reference count for the underlying interface pointer and releases the interface when the variable goes out of scope.
- There are two key methods for a ComPtr:
  - Get() Returns the pointer to the underlying interface
  - GetAddressOf() Returns the address of the underlying pointer
- One of the reasons why many books do not reference ComPtr is that it did not appear in the Windows SDK until Windows 8. However, because it is implemented as a template class, you can use it on code that needs to run on Windows 7 or later.
  - See the DeviceDetectionWithComPtr example.

Starting with DirectX

- Rather than going through a lot of slides that describe this in detail, it is better to provide code that shows it. The slides will just cover the basic terminology.
- Slides should be read in conjunction with reading the sample code provided to you.
- The basic code that is going to be used for all DirectX applications in this module is provided in the solution DirectX_Base.
- This builds on the basic framework introduced in the pre-session tasks and includes the code needed to initialise Direct3D 11.
- At the moment, it just clears the window to a black background

A Simplified Rendering Pipeline

- If you look through DirectX books or documentation, you will see discussion about various stages of the rendering pipeline which are really outside the scope of this module (such as the Tesselator, Hull Shader, Domain Shader and Geometry Shader). We will focus on the core pieces needed.
- In order to render a scene, you will need two core components:
  - Your C++ code
  - Shaders written in High Level Shader Language (HLSL)

Shaders

- Shaders are the name given to functions that run on the GPU
- In DirectX, shaders are written in High Level Shader Language (HLSL)
- There are two shaders we need to create:
  - A vertex shader that is run for each vertex in the vertex list. It applies any transformations to the vertex, as well as doing any other calculations that occur at the vertex level (such as per-vertex lighting calculations, etc)
  - A pixel shader that is run for each pixel that is going to going to be rendered to the target buffer (such as per-pixel lighting calculations, etc).
- Both shaders can be in the same file (usually with an extension of .HLSL)
- Each shader is compiled at run time and passed to the appropriate rendering stage.

Setup for DirectX

- The steps needed for an application to setup to use DirectX are as follows (all of these can be seen in the DirectX_Base solution):
  - Check that we have a video card and driver capable of supporting the features we want
  - Create a swap chain of buffers
  - Create a drawing surface the same size and format as the window (the render target)
  - Create a depth buffer the same size as the drawing surface
  - Define the viewport that we will be rendering to

Rendering Objects

- In order to render an object, we always need the following components:
  - A list of vertices
  - A list of indices onto the vertex list that form the polygons for our object
  - A constant buffer
  - A vertex shader
  - A pixel shader
- As we go further in the module, we will see additional things that are required for rendering more complex objects such as textures, materials, lights, etc.

Rendering Objects – Required Steps

- Setup:
  - Create vertex buffer and initialise with contents of vertices
  - Create index buffer and initialise with indices
  - Notify the Input Assembler stage of the format of each vertex (the input layout)
  - Create the constant buffer
  - Compile vertex shader
  - Compile pixel shader
  - Setup the rasteriser stage (not always needed).

Rendering Objects – Required Steps

- Rendering:
  - Initialise constant buffer and send through to Direct3D
  - Notify Direct3D of:
    - which vertex buffer to use
    - which index buffer to use
    - which vertex shader to use
    - which pixel shader to use
    - what primitive type the lists contain
  - Call DrawIndexed() to render the object
- The DirectX_Cube_Wireframe solution shows these steps added to the base solution seen previously.

Example

- The DirectX_Cube_Wireframe draws a 3D cube in wireframe
  - Wireframe is used so that you can see the different polygons
- It does not look 3D at the moment since the camera view is setup so that you are looking directly at the front face of the cube and cannot see the sides.

Practical Exercises

- Some tutorial exercises have been provided for you this week to enable you to work with applying basic transformations to the cube to see it being animated.
- You have been given a lot to look at and we know this is new, but you have all of the code you need – it is a case of studying it and applying it.
- If in doubt, ASK for help.

Next Week

- We will start to look at applying colours to the cube and looking more at HLSL.