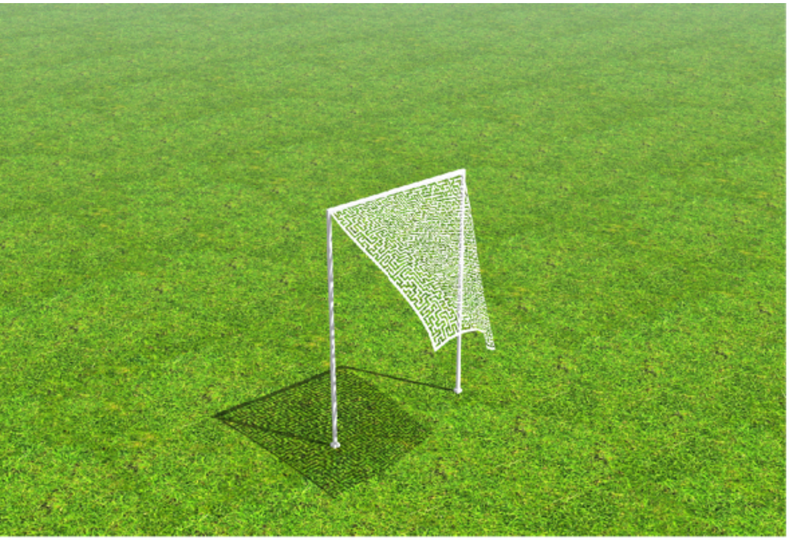
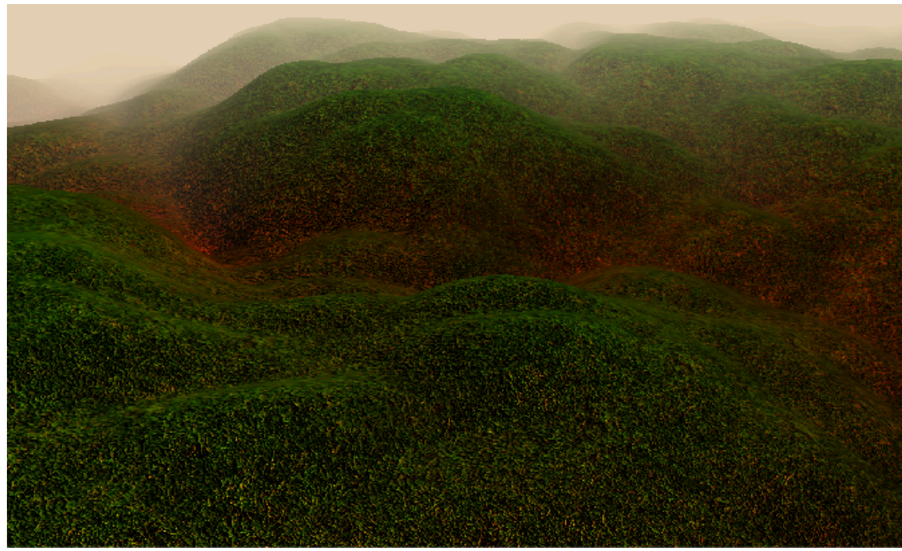
COSC342: Computer Graphics

2017





Lecture 12

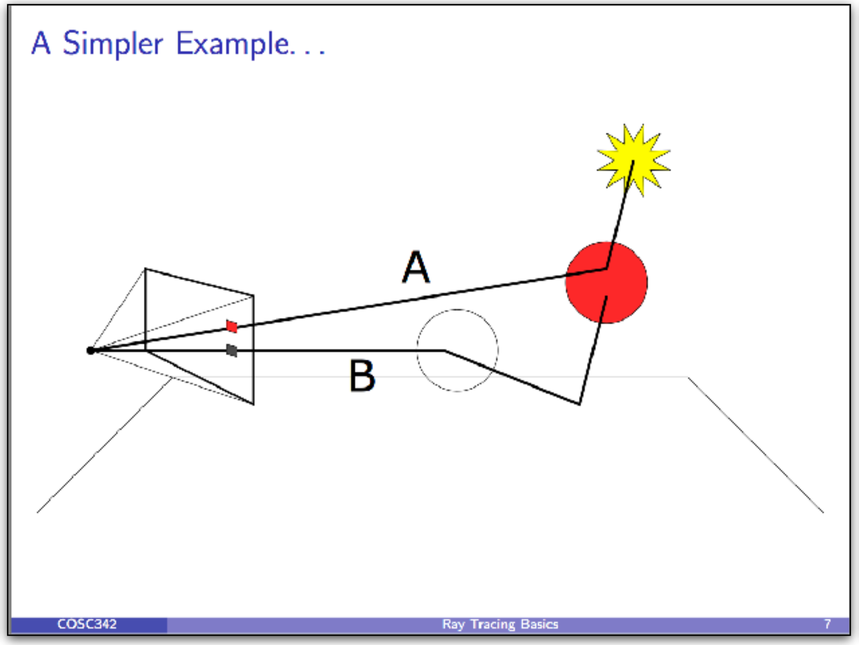
INTRODUCTION OPENGL

Stefanie Zollmann

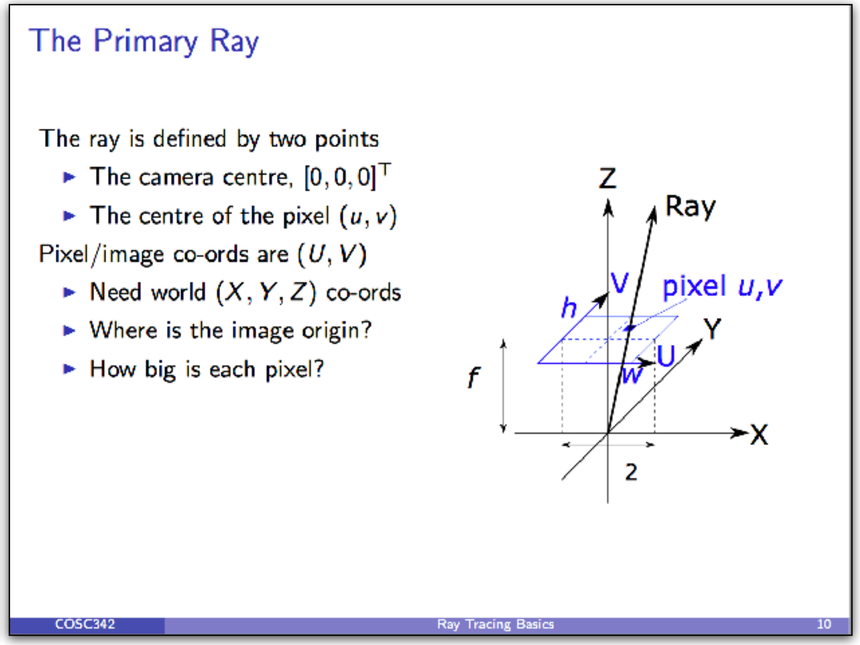
LAST LECTURE

Introduction into Raytracing

Ray/Path Tracing Intro

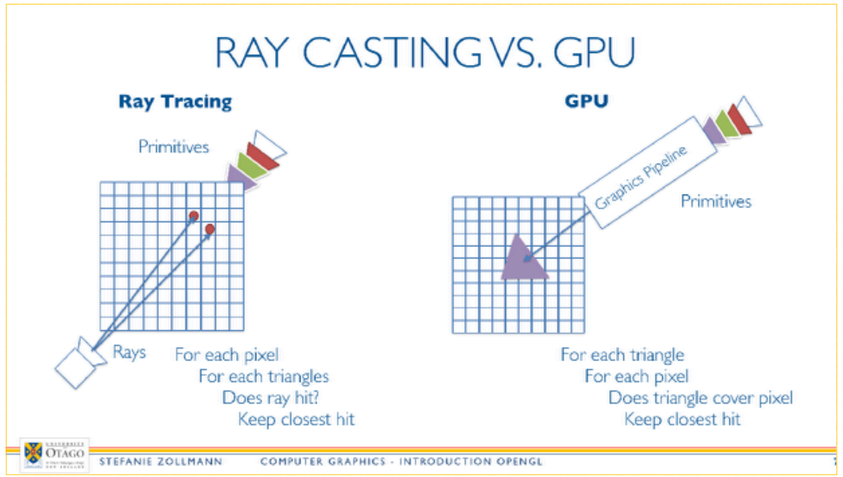


Principle

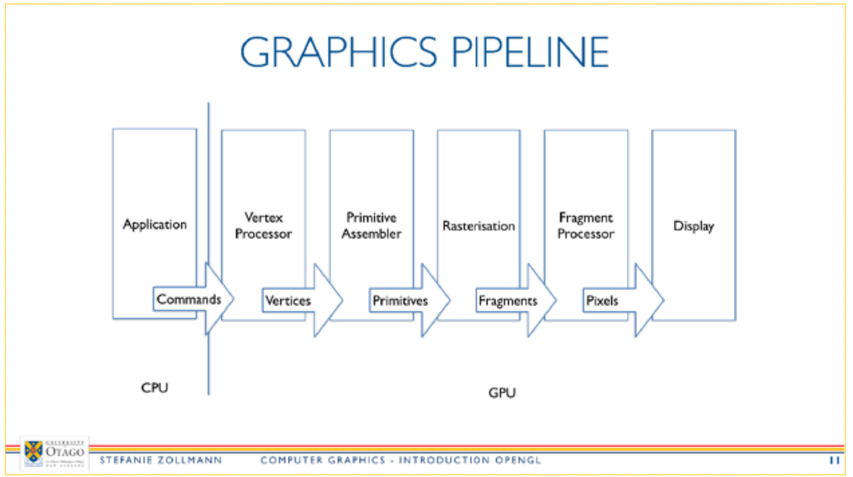


Primary Ray

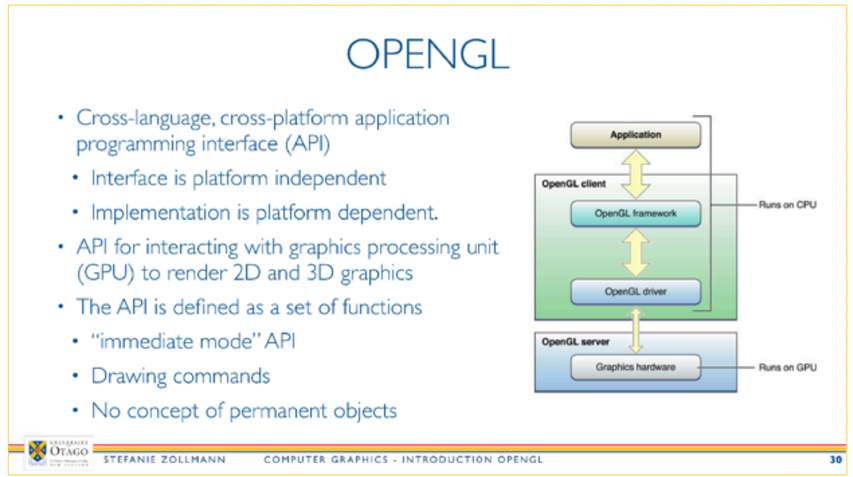
TODAY



Rasterisation vs. Raytracing



The Graphics Pipeline

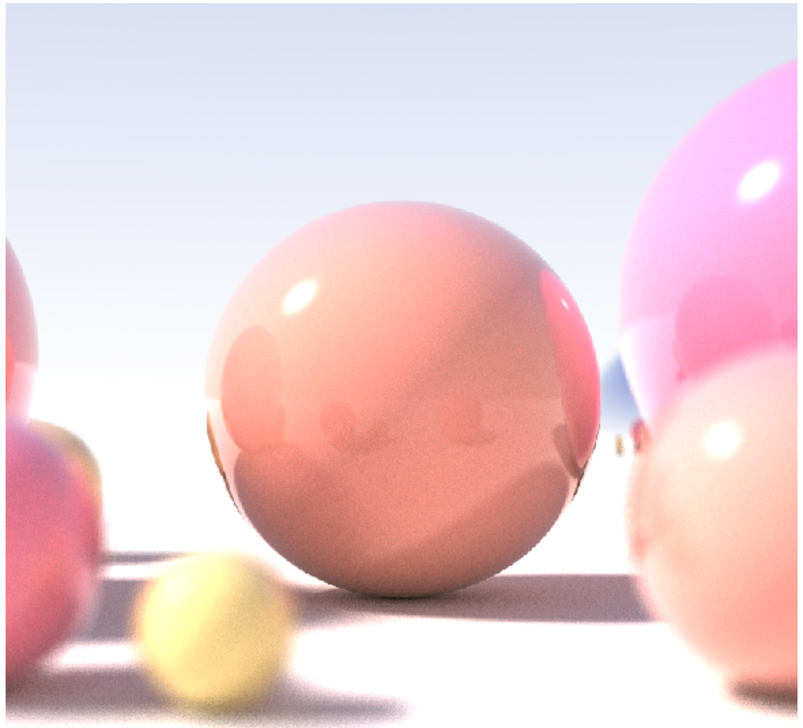


OPENGL

ANOTHER RENDERING APPROACH?

Raytracing:

- Advantages
  - Generality: Render anything that can be intersected with a ray
  - Realism: Shadows, reflection
- Disadvantages
  - Performance: Often too slow for interactive applications
  - Each pixel touched many times



By Tim Babb - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3856908

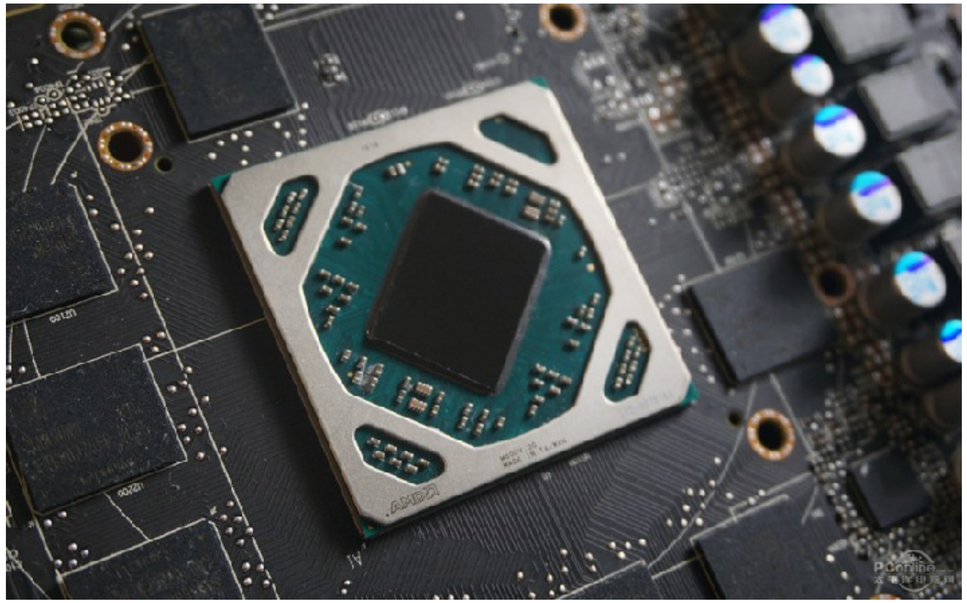
REAL-TIME GRAPHICS

Demands:

- Full HD at 60 Hz: 1920 × 1080 × 60 Hz = 124 Mpx/s
- Stereo x2 (computing two images)
- In real-time
- Capacities for additional computations

How?

- Hardware implementation: graphics processing unit GPU



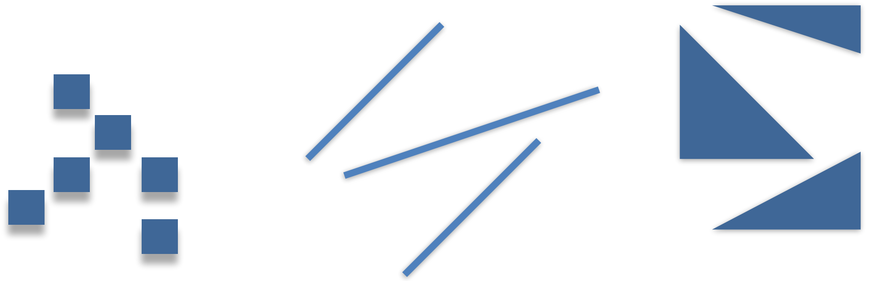
By Shawn Knight - http://www.techspot.com/news/65328-amd-radeon-rx-480-benchmarks-bare-pcb-photos.html, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=54979727

REAL-TIME GRAPHICS

Based on rasterisation of graphic primitives:

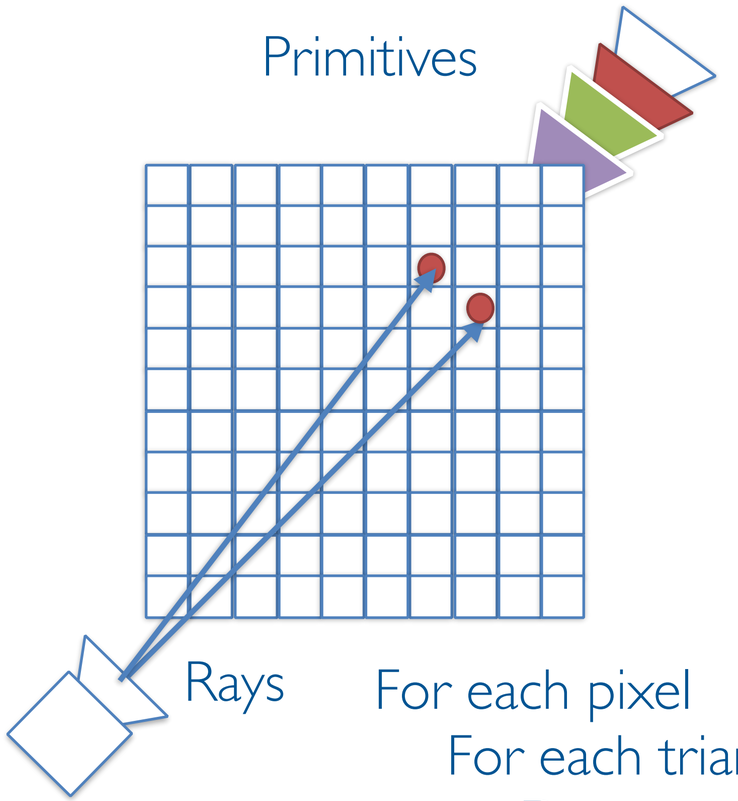
- Points
- Lines
- Triangles

- **Implemented in hardware (GPU)**



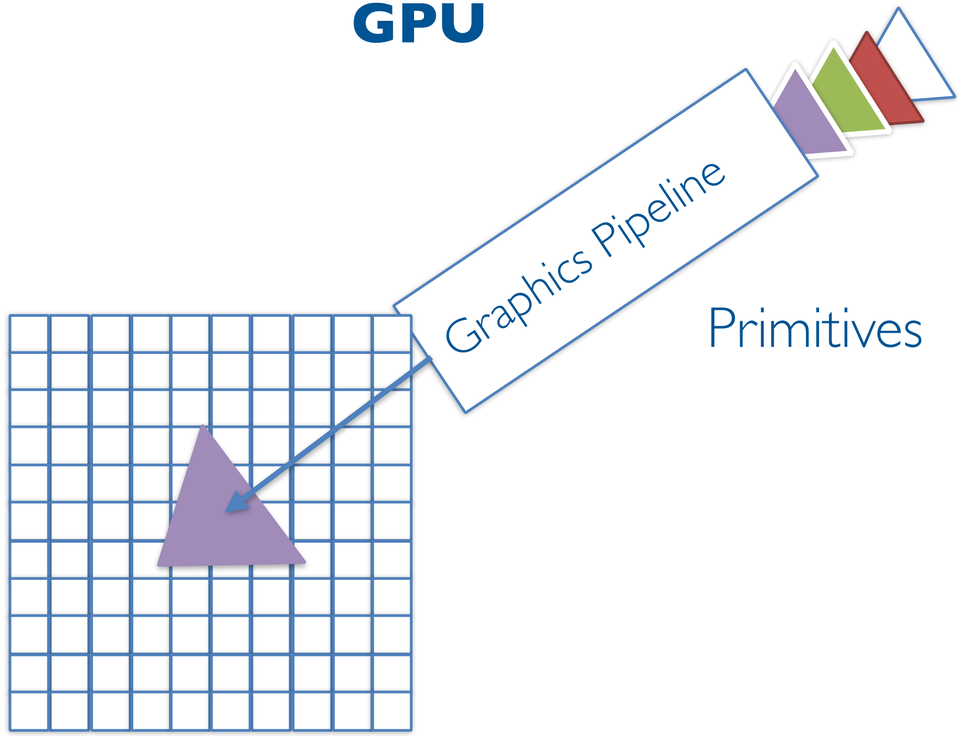
RAY CASTING VS. GPU

Ray Tracing



For each pixel
For each triangles
Does ray hit?
Keep closest hit

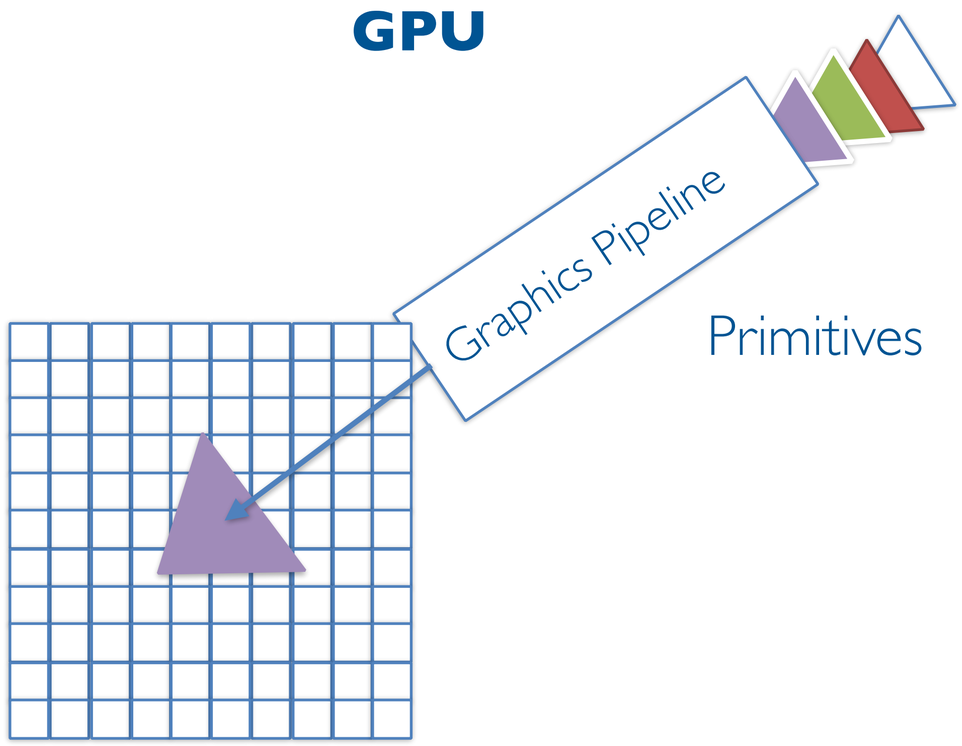
GPU



For each triangle
For each pixel
Does triangle cover pixel
Keep closest hit

GPUS DO RASTERISATION

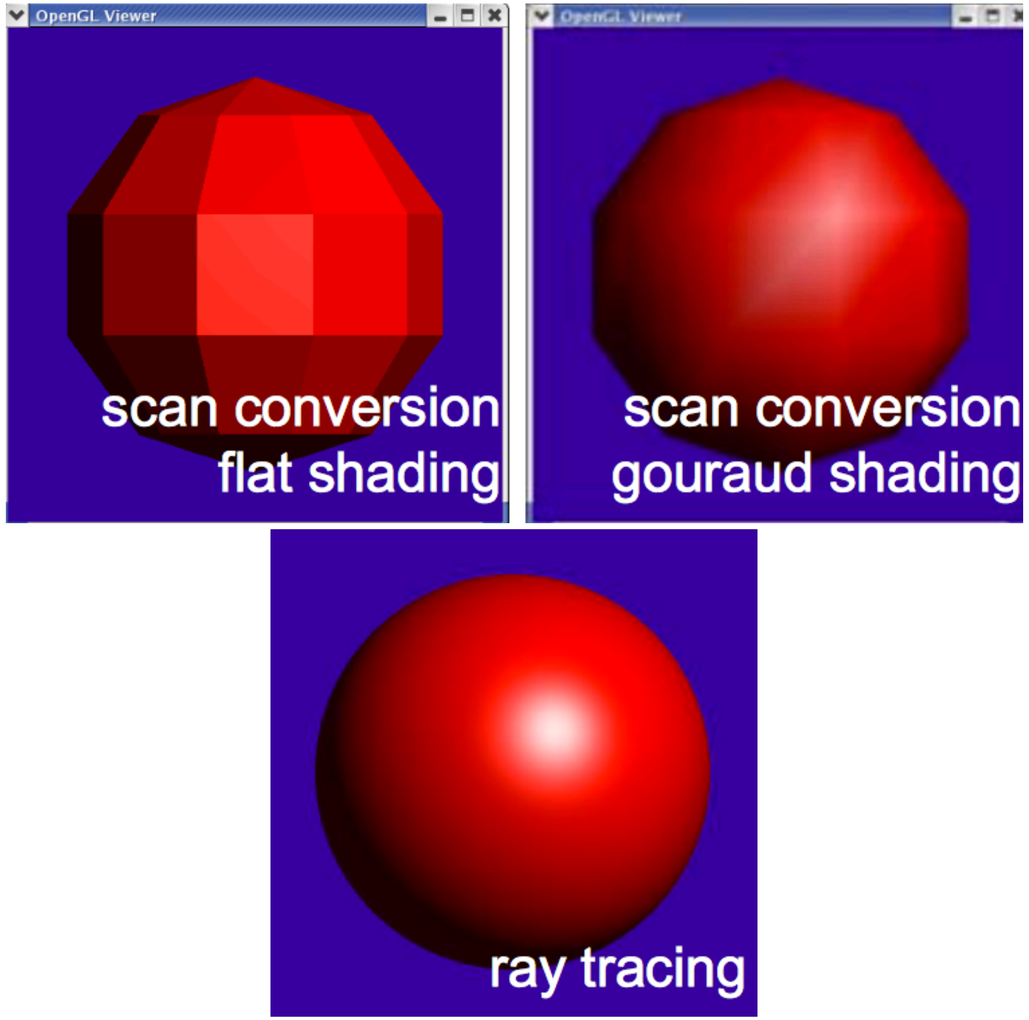
- **Rasterisation**
  - Computing for each triangle which pixel it covers
  - Triangle centric approach
  - Rasteriser only needs one triangle at a time



For each triangle
For each pixel
Does triangle cover pixel
Keep closest hit

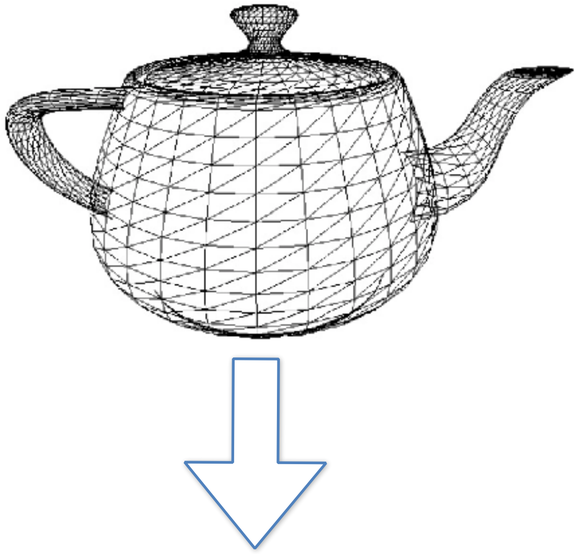
DISADVANTAGES RASTERISATION

- Restricted to scan-convertible primitives
- Requires conversion of complex objects into polygons
- Faceting, shading artefacts
  - Addressed by programmable per-pixel shading
- No automatic handling of shadows, reflection, transparency
- More triangles than pixels?



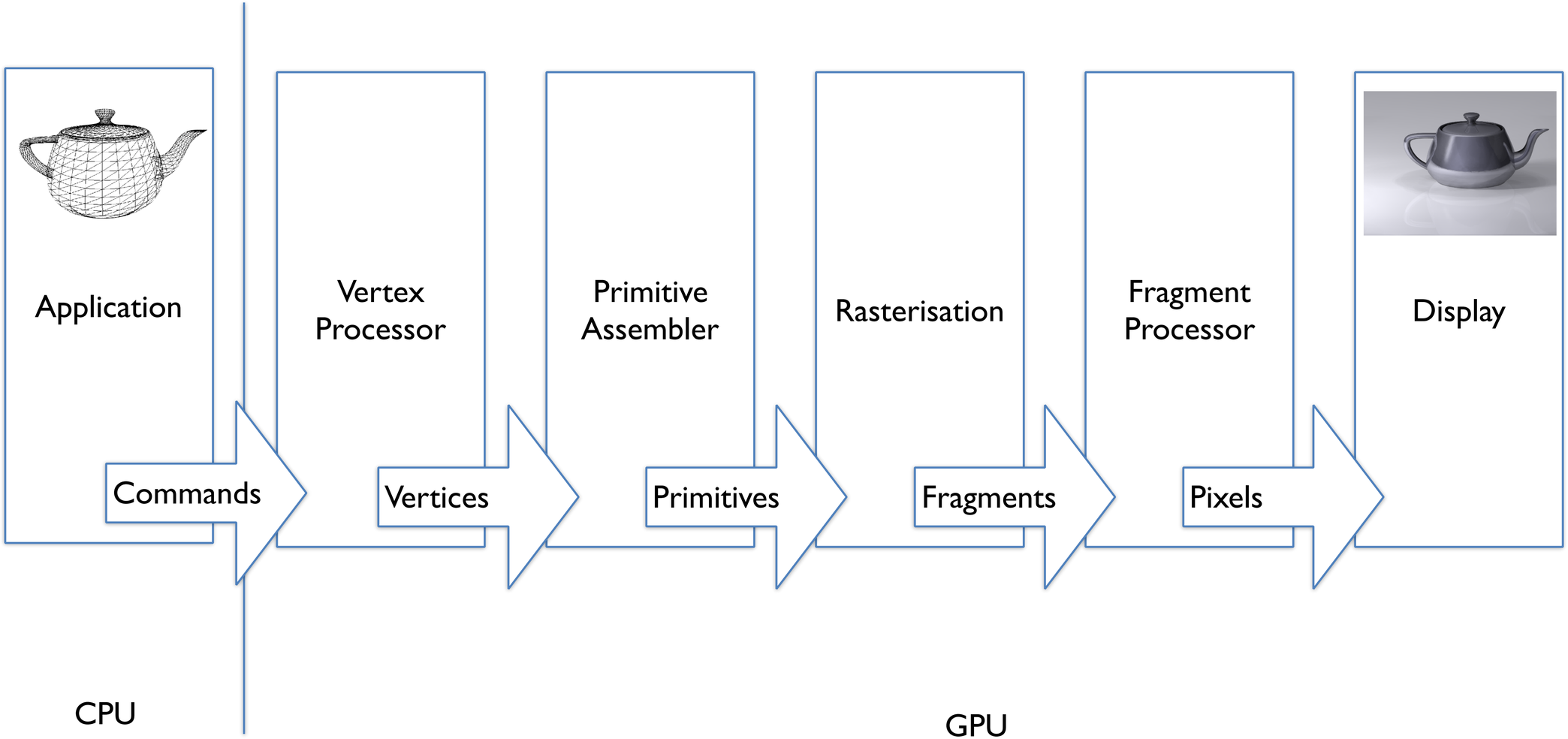
GRAPHICS PIPELINE

- **Input**
  - Geometric model
    - Vertices, normals, texture coordinates
  - Lighting/shading model
    - Light positions
    - View point and virtual camera configuration
- **Output**
  - Colour (and depth) per pixel on a screen



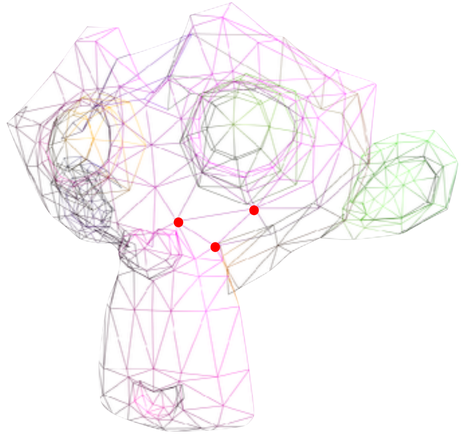
https://commons.wikimedia.org/wiki/File:Utah_teapot_simple.png

GRAPHICS PIPELINE



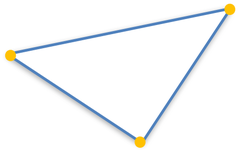
GRAPHICS PIPELINE

3D mesh

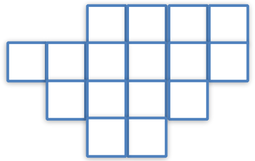


Vertices in screen space

Triangles in screen space



Fragments



Fragments with colours

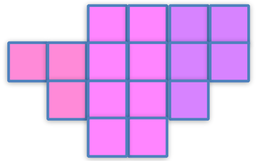
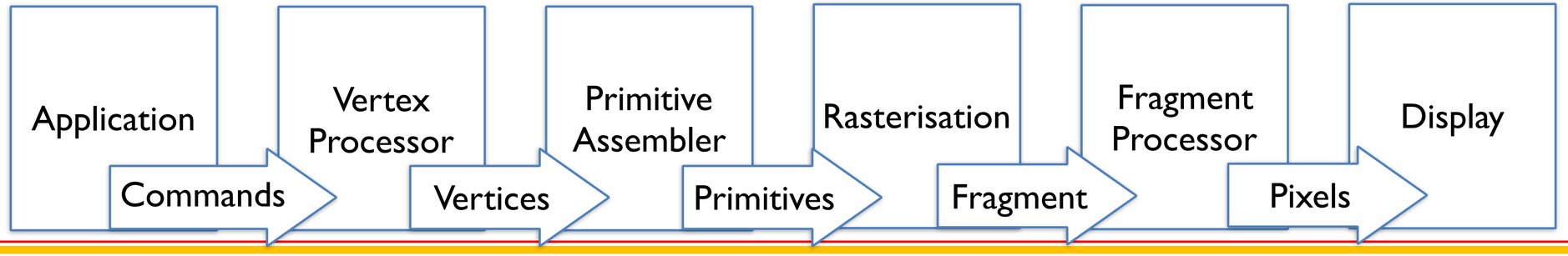
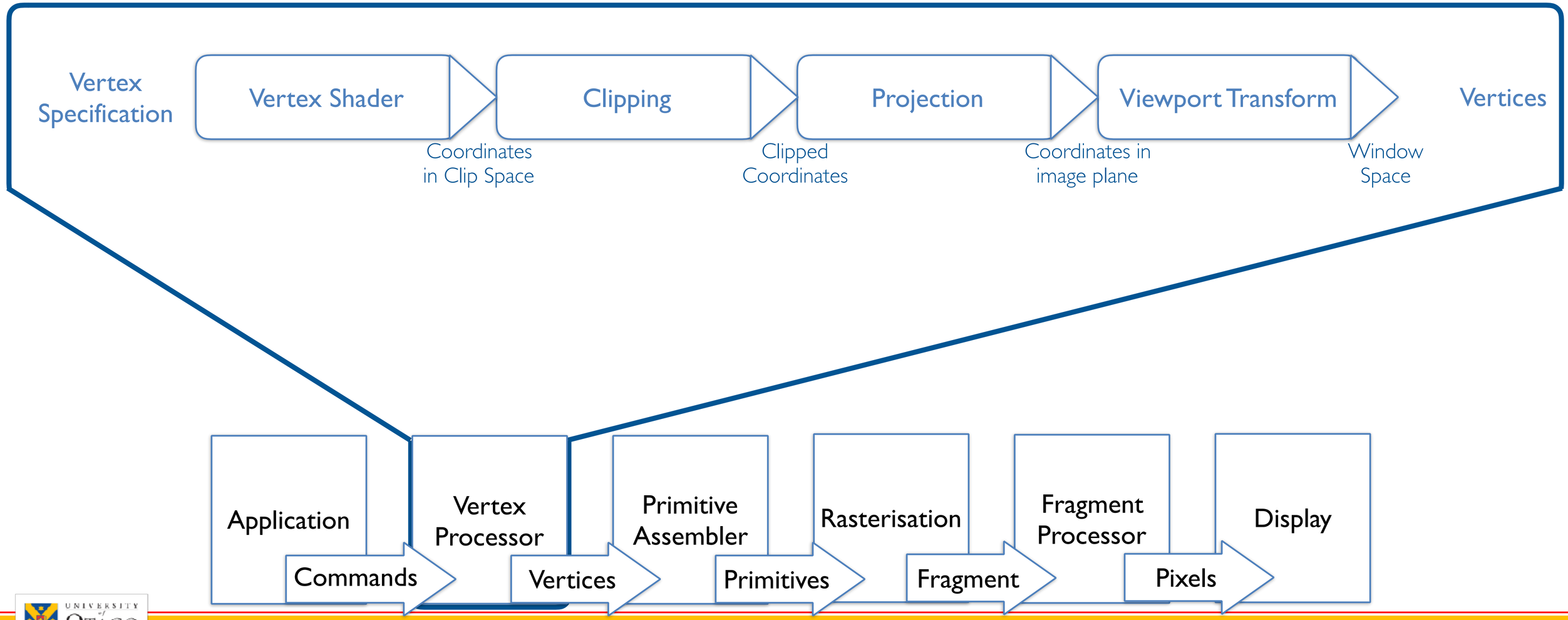


Image Output

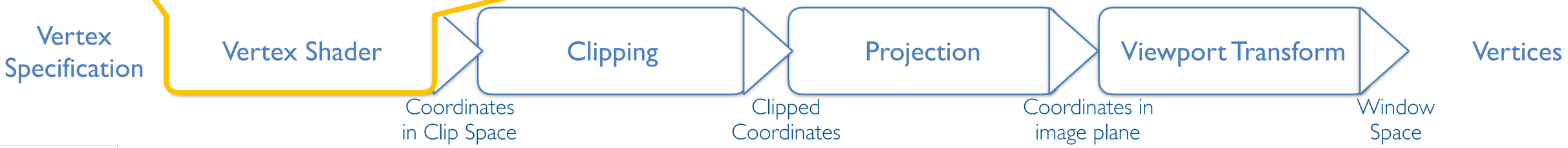


VERTEX PROCESSING

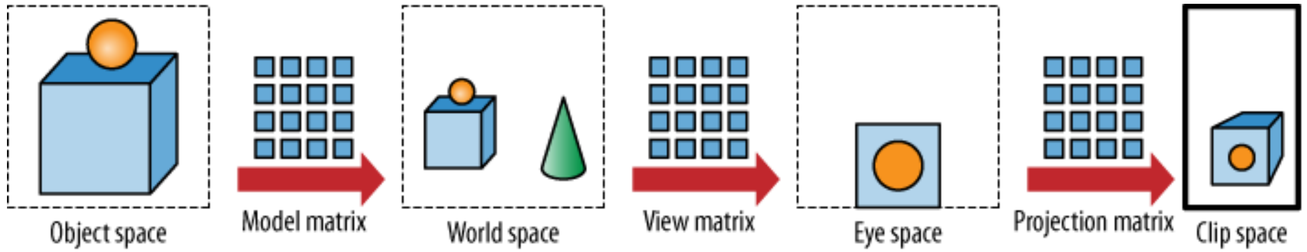


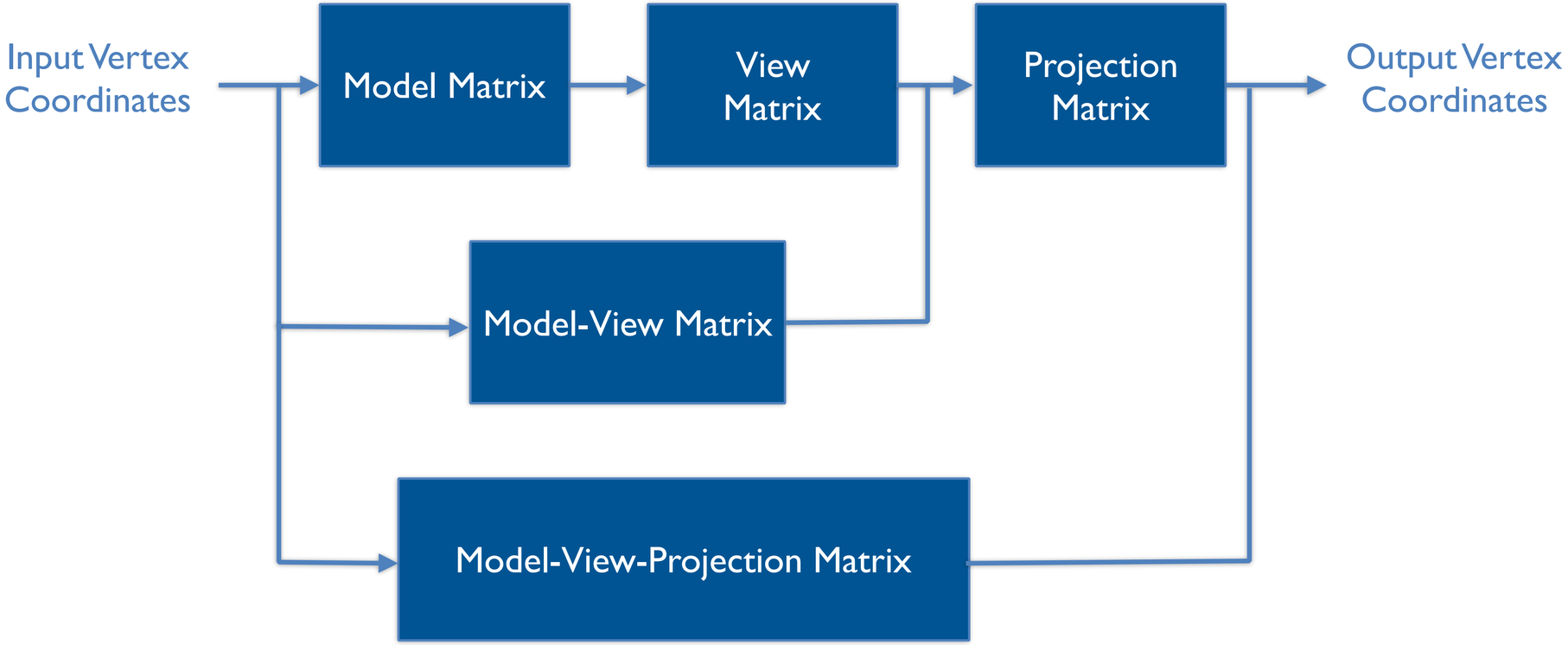
VERTEX SHADER

- Programmable shader stage (GPU program)
- "Shaders" were small programs performing lighting calculations
- Transforms input vertex stream into stream of vertices mapped onto the screen (clip space coordinates: homogeneous coordinates)
- Uses model, view and projection matrices to transform from model to world to view and to clip space

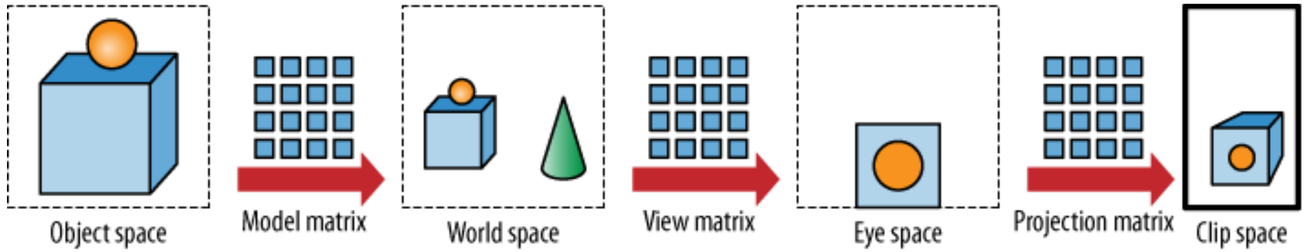


VERTEX SHADER: TRANSFORMATIONS





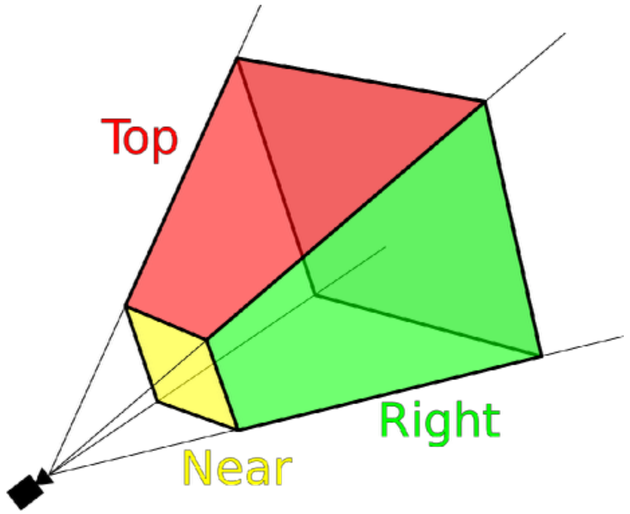
VERTEX SHADER: EXAMPLE




```
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosModelspace;

// Output data ; will be interpolated for each fragment.
out vec3 posWorldspace;

// Values that stay constant for the whole mesh.
// Model-view-projection matrix
uniform mat4 MVP;

void main(){
  // Output position of the vertex, in clip space : MVP * position
  gl_Position =  MVP * vec4(vertexPosModelspace,1);
}
```

CLIPPING

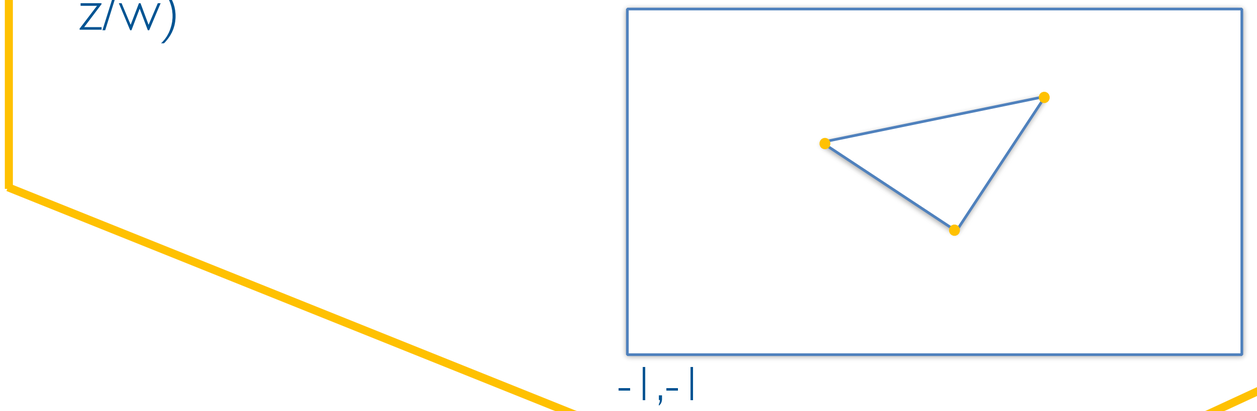
- Coordinates not entirely in view are clipped to the view volume



By MithrandirMage [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

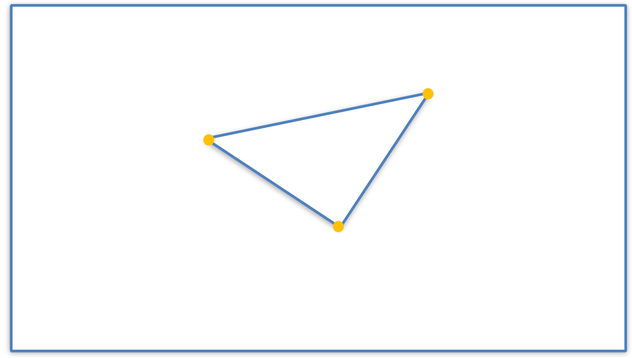
PROJECTION

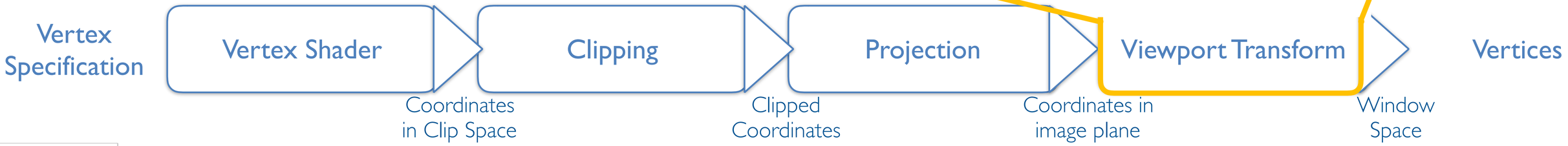
- We need normalised device coordinates ranging from -1 to +1 on each axis regardless of the shape or size of the actual screen
- Transforms clip space coordinates into normalised device coordinates (NDC)
- Divide clip space coordinates (x,y,z) by clip-space factor w (x/w, y/w, z/w)



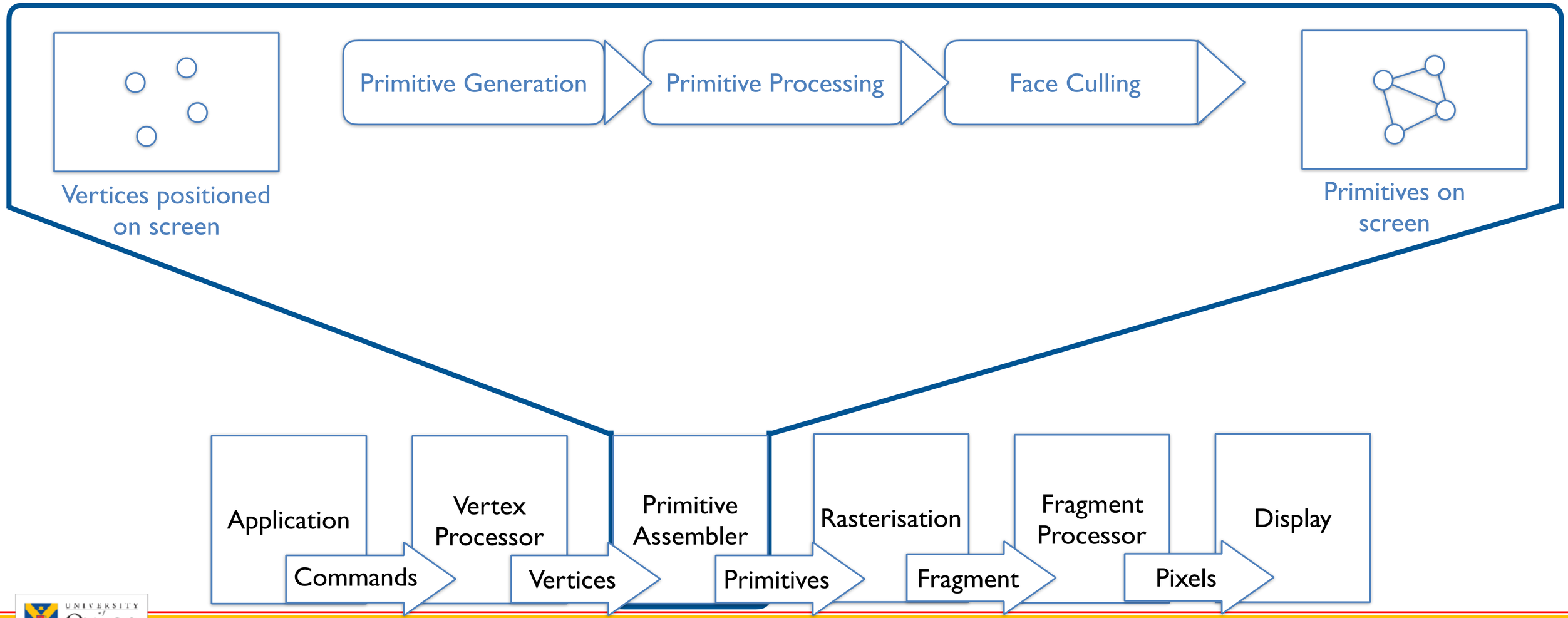
VIEWPORT TRANSFORM

- Map resolution-independent normalised device coordinates to a rectangular window in the frame buffer, the viewport
- Is defined by the viewport definition
- Output in window (pixel) coordinates





PRIMITIVES ASSEMBLY



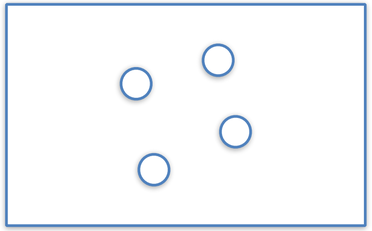
PRIMITIVES ASSEMBLY

Vertices positioned on screen

Primitives on screen

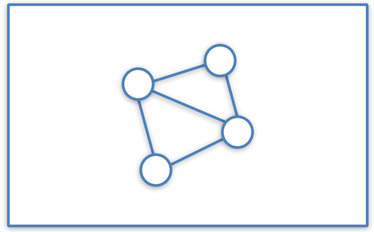
- Primitive assembly receives processed vertices, and the vertex connectivity information as input
- Divides them into a sequence of individual base primitives
- Primitives now have a certain facing
- Face culling discards faces based on their facing

PRIMITIVES ASSEMBLY: PRIMITIVES

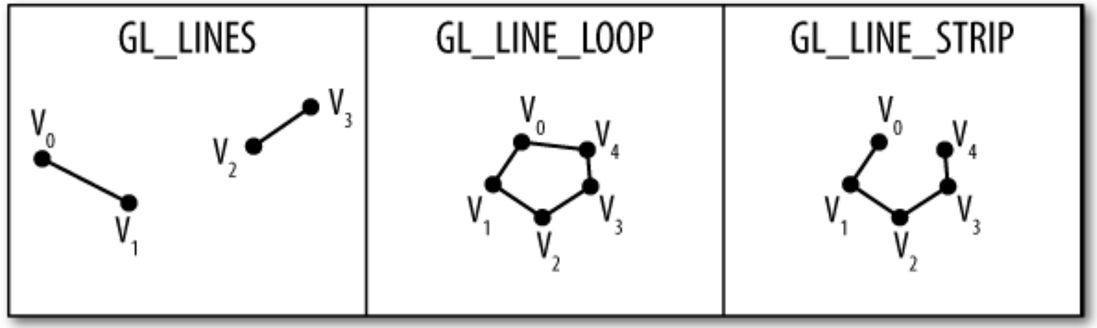


Vertices positioned on screen

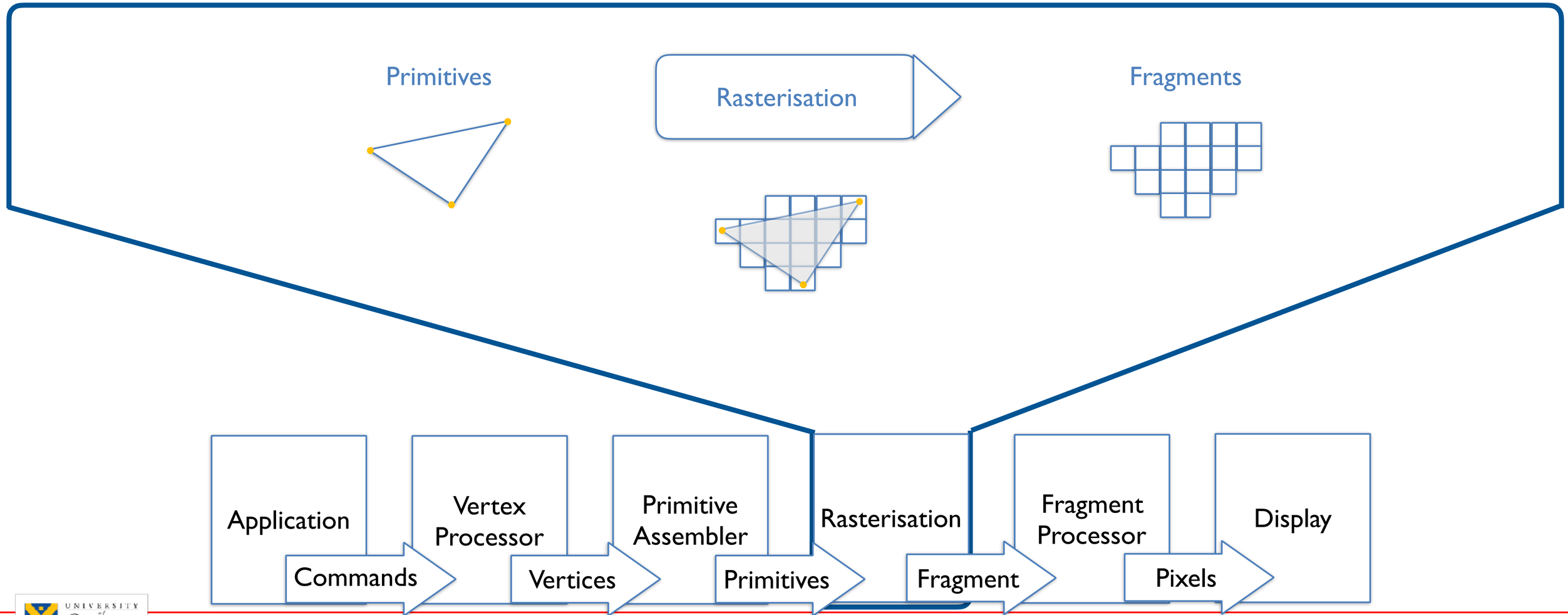
Primitive Generation → Primitive Processing → Face Culling



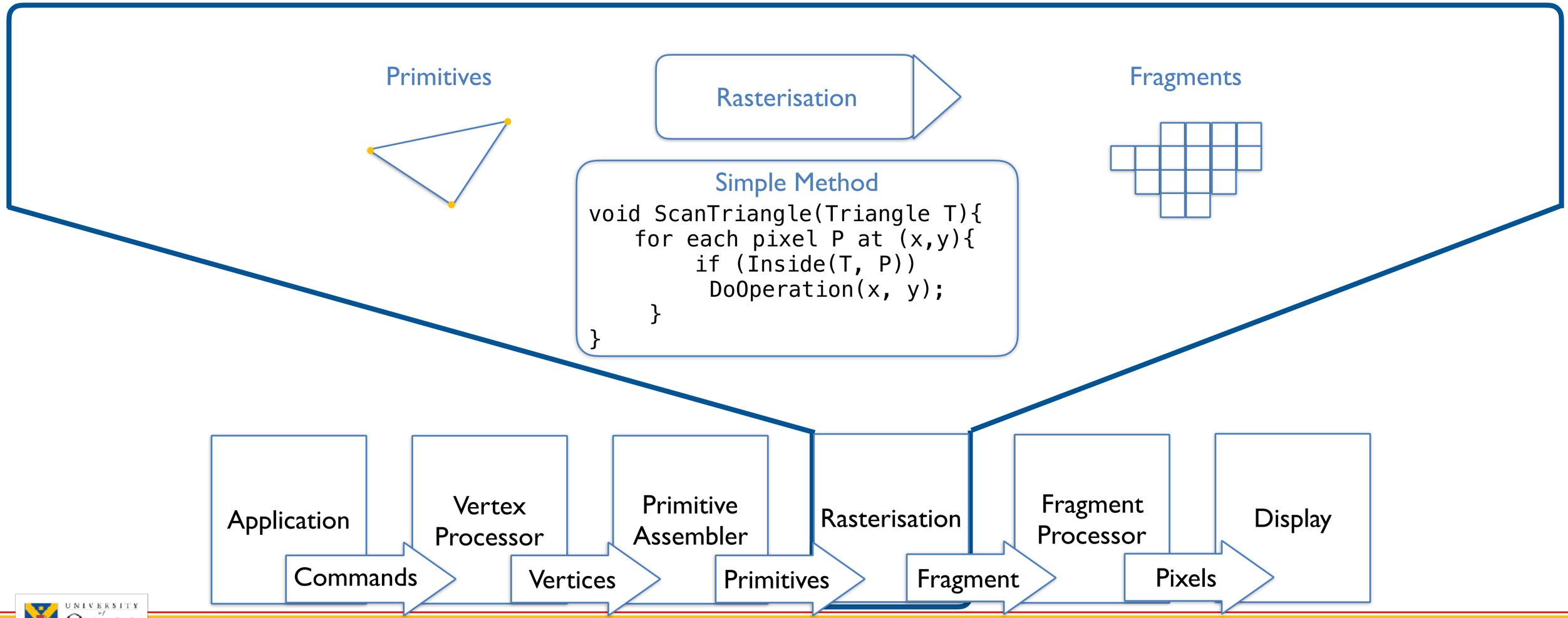
Primitives on screen



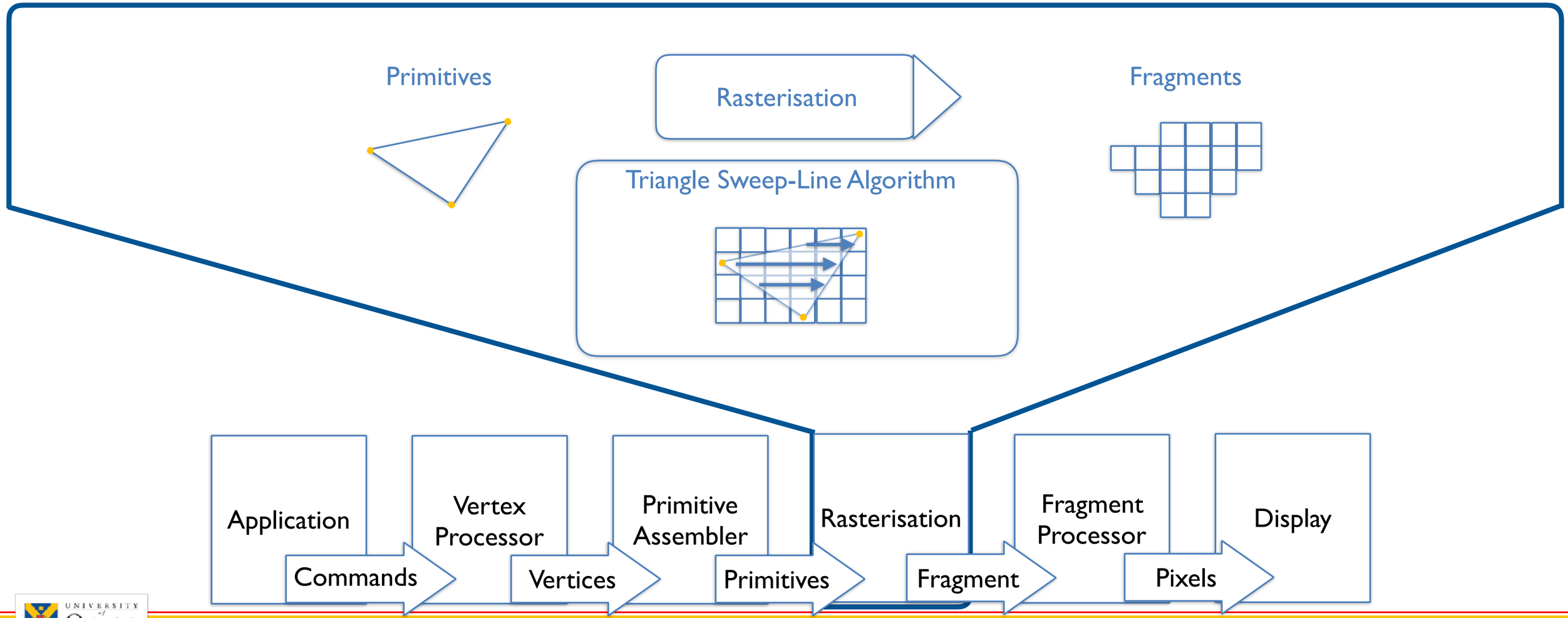
RASTERISATION



RASTERISATION

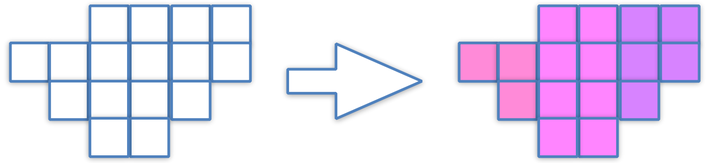


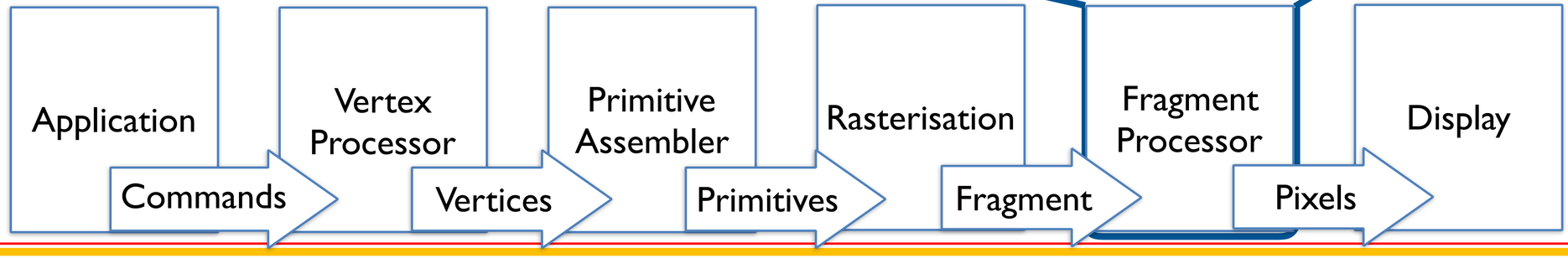
RASTERISATION



FRAGMENT PROCESSING

- Output of the rasterisation stage are fragments
- Fragments are processed by a fragment shader
- Fragment shader have control over the color and depth values





FRAGMENT SHADER: EXAMPLE

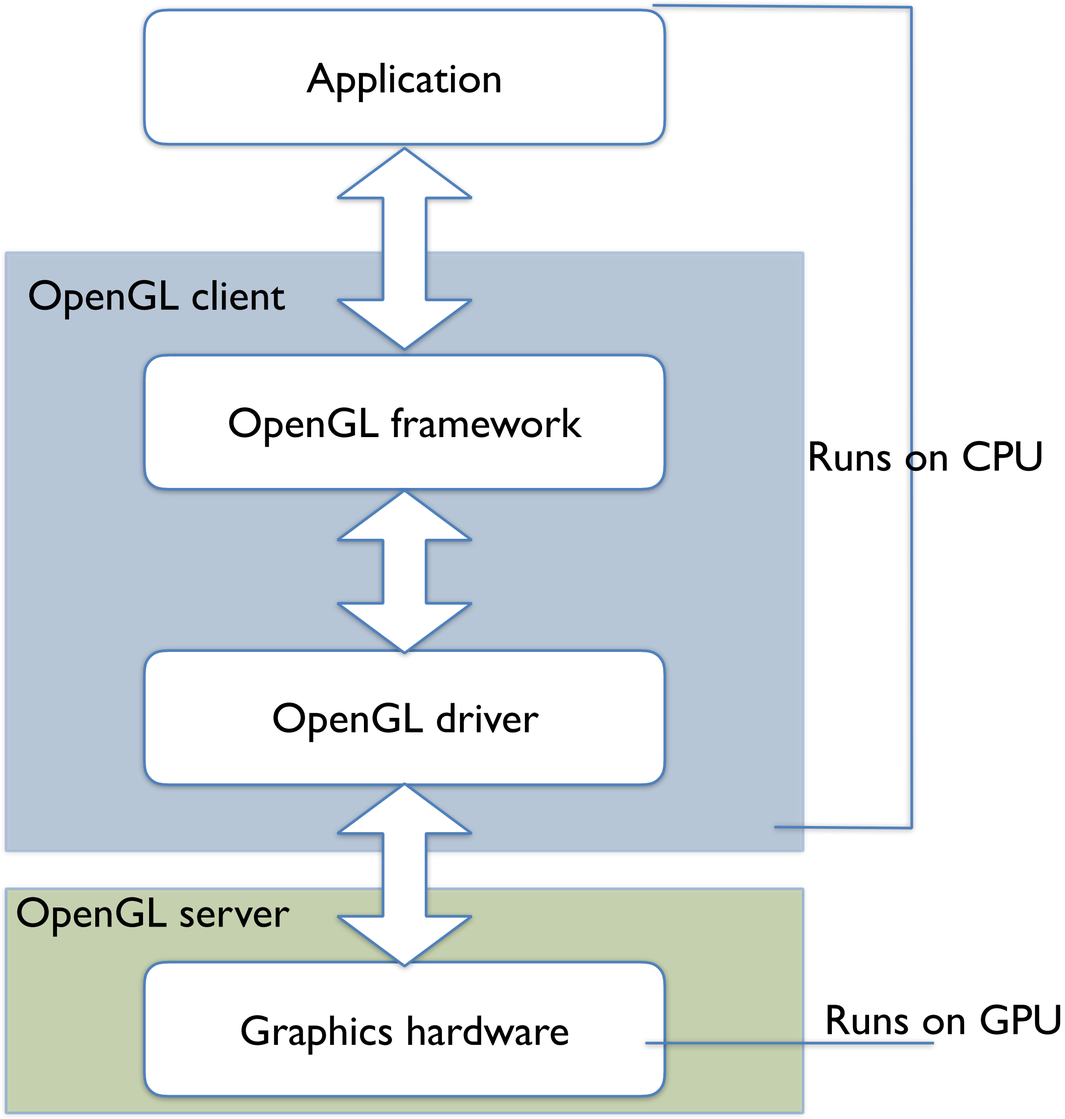
```
// Ouput data
out vec3 color;

uniform vec4 colorValue;
void main()
{

  // Output color
  color = colorValue.rgb;

}
```

LET'S GET PRACTICAL

How to talk to the Graphics Hardware?

OPENGL

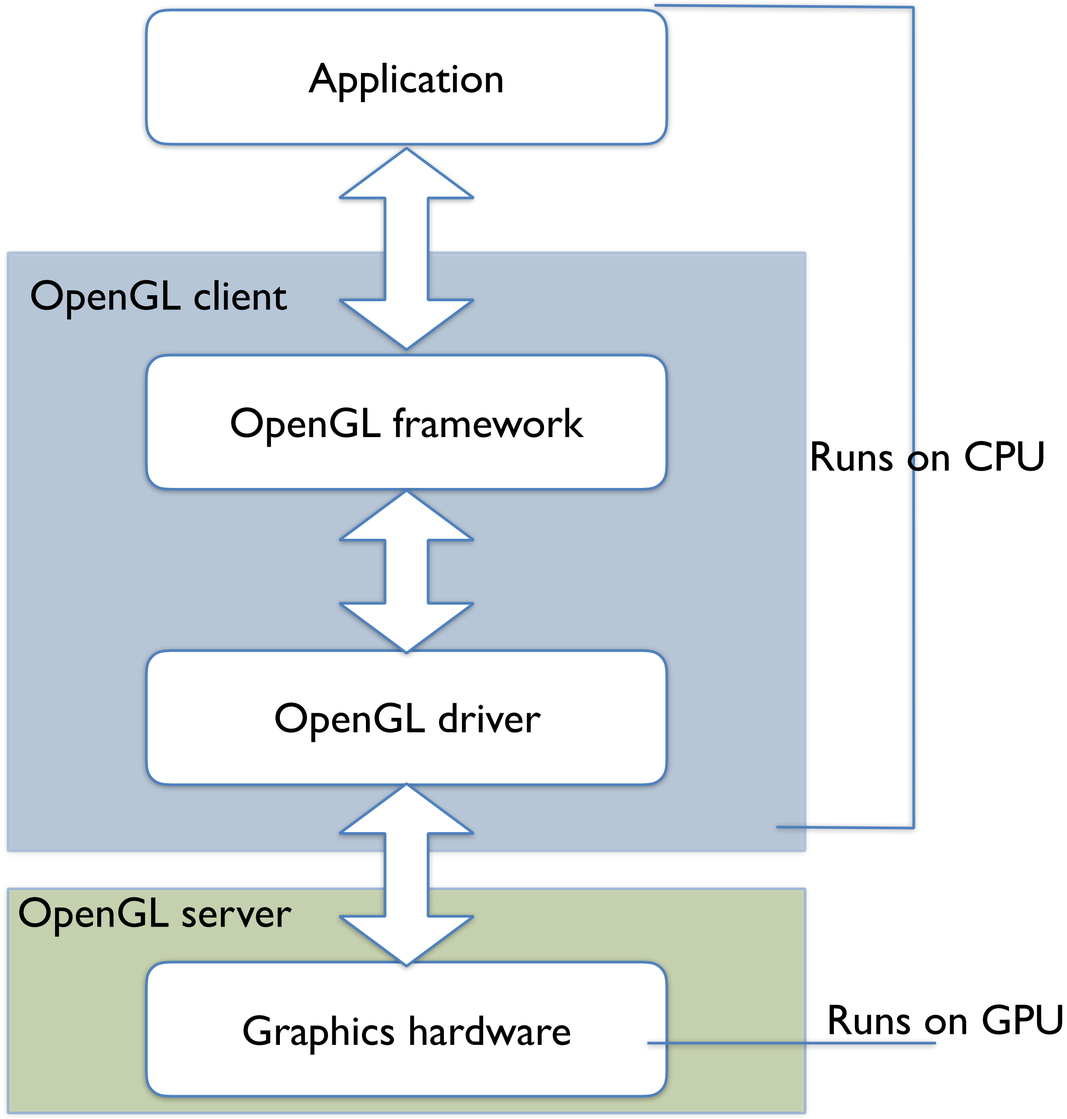
- Cross-language, cross-platform application programming interface (API)
  - Interface is platform independent
  - Implementation is platform dependent.
- API for interacting with graphics processing unit (GPU) to render 2D and 3D graphics
- Works using a client-server model
  - Client (application) creates commands
  - Server processes commands
  - May, in fact, be the same machine



OPENGL

Important to note:

- The API is defined as a set of functions
  - "immediate mode" API
  - Drawing commands
  - No concept of permanent objects



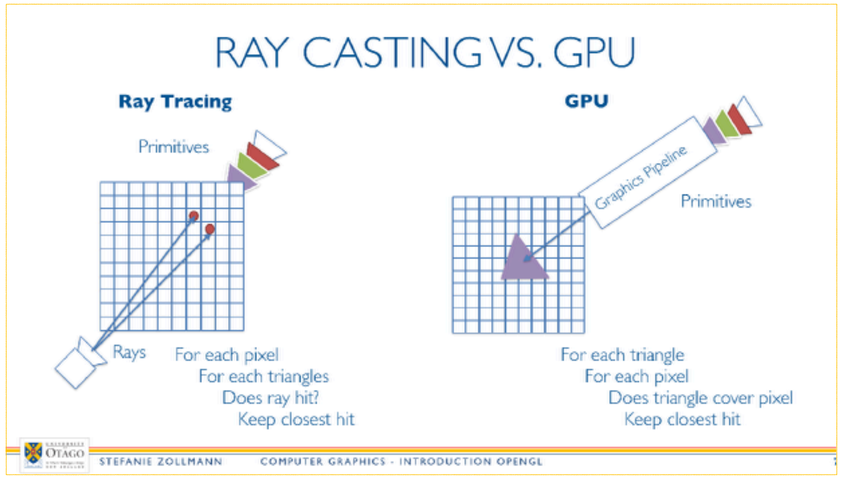
OPENGL - HISTORY

- Originally released by Silicon Graphics Inc. (SGI) in 1992
- Now managed by non-profit technology consortium Khronos Group
- OpenGL has been through a number of revisions
- Significant changes:
  - **OpenGL 2.0** incorporates the significant addition of the OpenGL Shading Language (also called GLSL)
  - **OpenGL 3.0** first major API revision deprecated fixed-function vertex and fragment processing and direct-mode rendering, using glBegin and glEnd

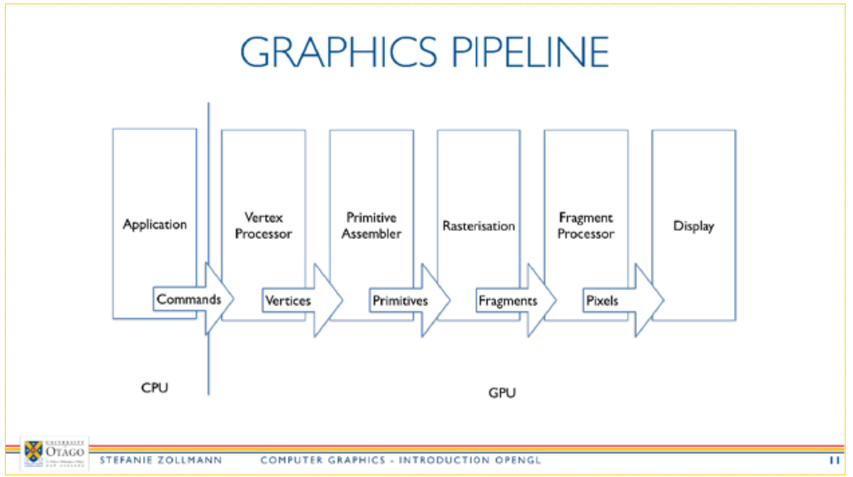
Important:

- This lecture uses "modern" OpenGL (version 3.x and higher, latest is 4.5)

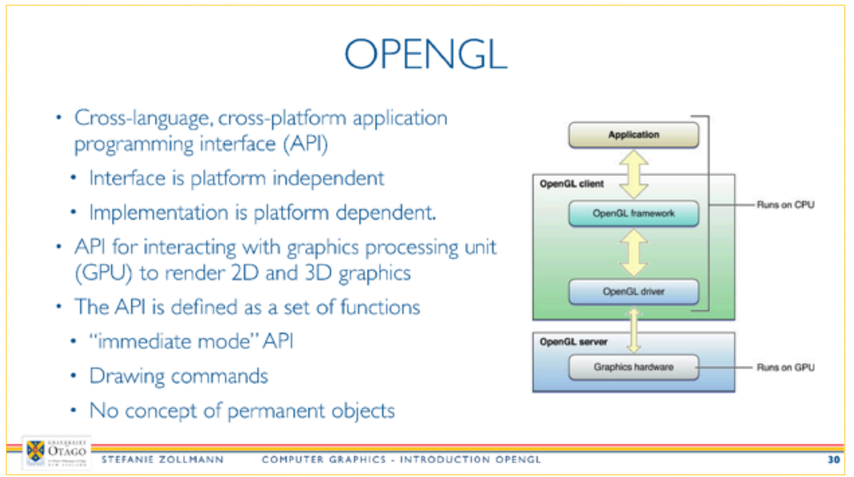
SUMMARY



Rasterisation vs. Raytracing



The Graphics Pipeline

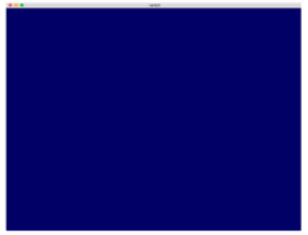


OPENGL

WHAT'S NEXT

OPENGL CONTEXT

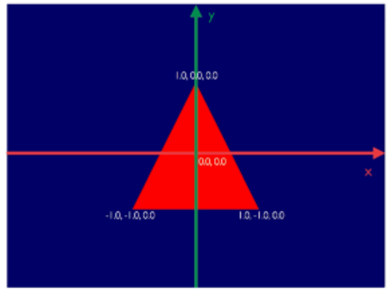
Context creation with GLFW (library for creation and management of windows with OpenGL contexts)



OpenGL Context

EXAMPLE: VERTEX BUFFER OBJECT

- Let's create our first triangle using a vertex buffer object




```
// Representation of the 3 vertices of
our triangle
//An array of 3 vectors each consisting
of x,y,z
    static const GLfloat data[] = {
        -1.0f, -1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        0.0f,  1.0f, 0.0f,
    };
```

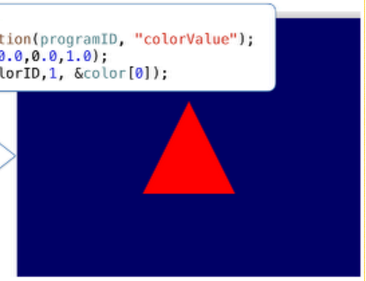

OpenGL Objects

FRAGMENT SHADER

Simple Fragment Shader outputting specified colour:

```
#version 330 core

// Ouput data
out vec3 color;

uniform vec4 colorValue;
void main()
{
  // Output color = red
  color = colorValue.rgb;

}
```

```
// add color parameter to shader
GLint colorID = glGetUniformLocation(programID, "colorValue");
glm::vec4 color = glm::vec4(1.0,0.0,0.0,1.0);
glProgramUniform4fv(programID,colorID,1, &color[0]);
```



Shaders

Thank You!

For more material visit

http://www.cs.otago.ac.nz/cosc342/