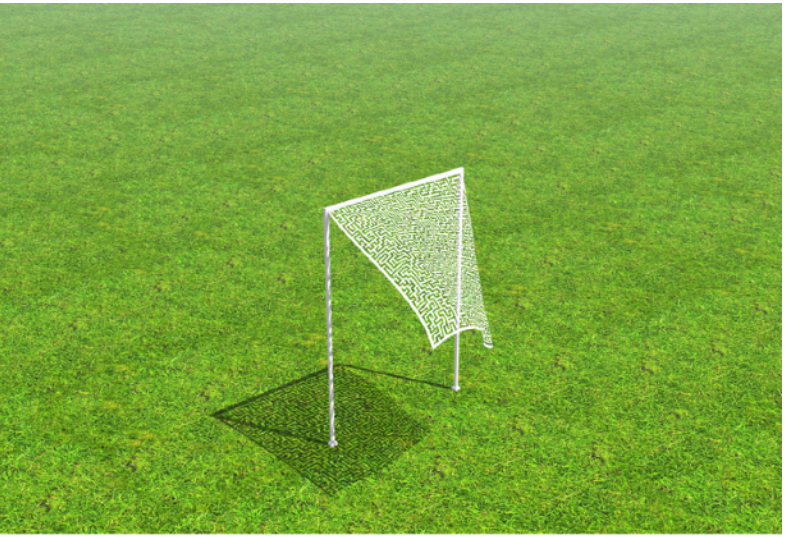
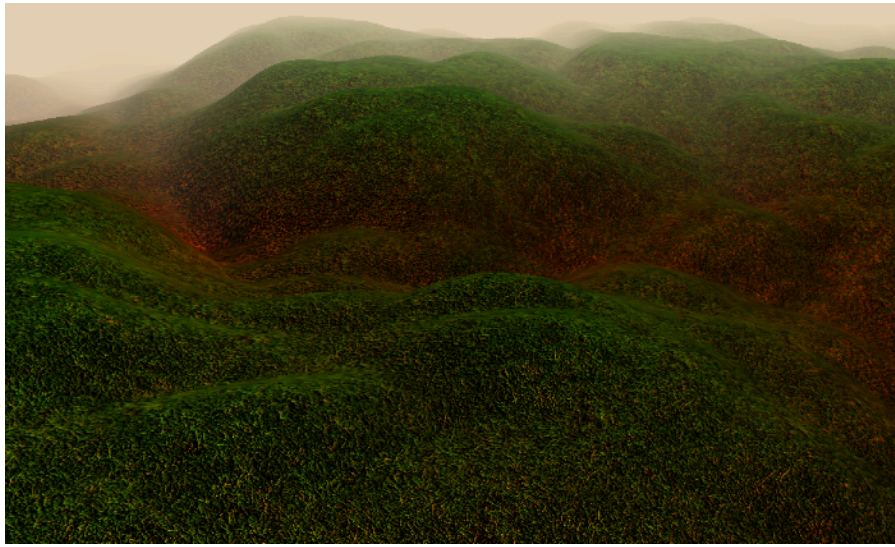
Visual Computing I:

Interactive Computer Graphics and Vision



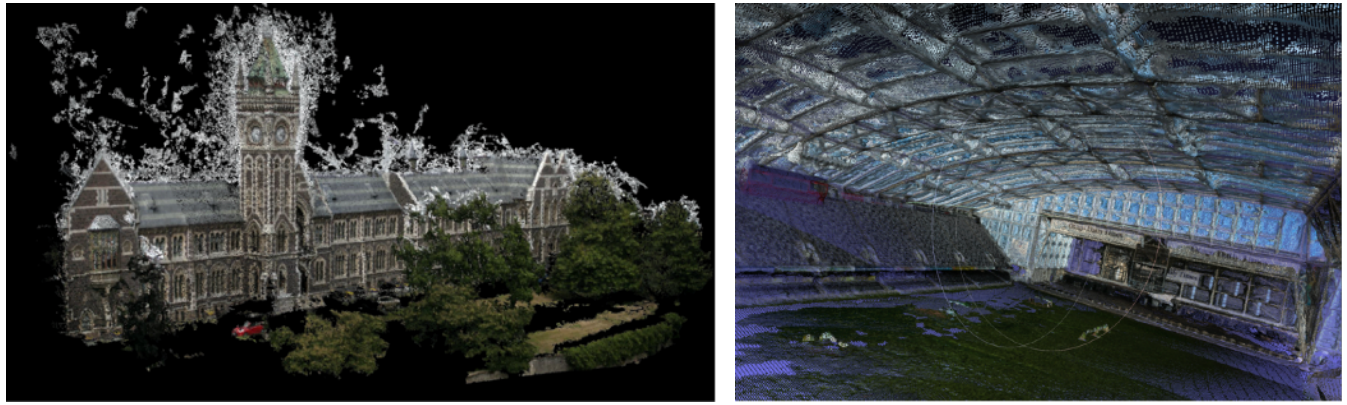


Computer Graphics Introduction

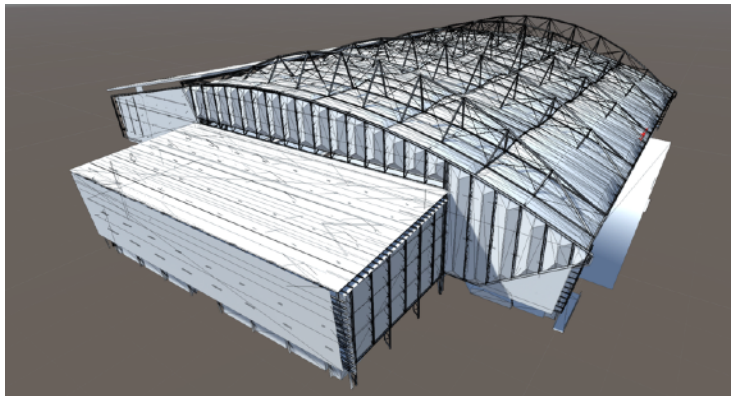
Stefanie Zollmann and Tobias Langlotz

Quiz

What to do with all this data?



Point Clouds



3D Mesh data

Why do we need real-time graphics?

Games

Augmented Reality

Virtual Reality

Visualization

Real-Time Graphics

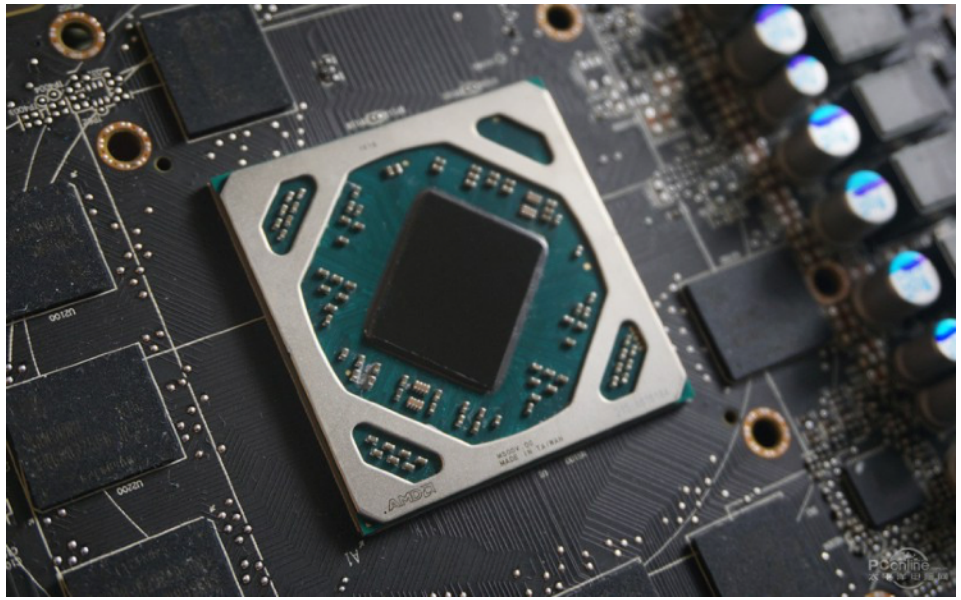
Real-Time Graphics

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



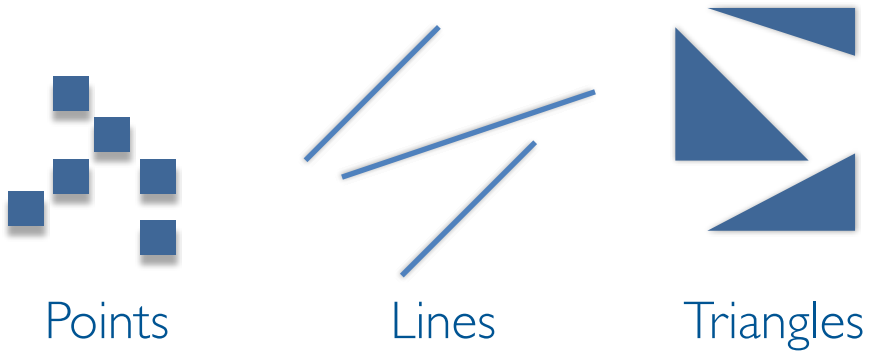
By Shawn Knight - http://www.techspot.com/news/65328-amd-radeon-rx-480-benchmarks-bare-pcb-photos.html, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=54979727

Real-Time Graphics

Based on rasterisation of graphic primitives:

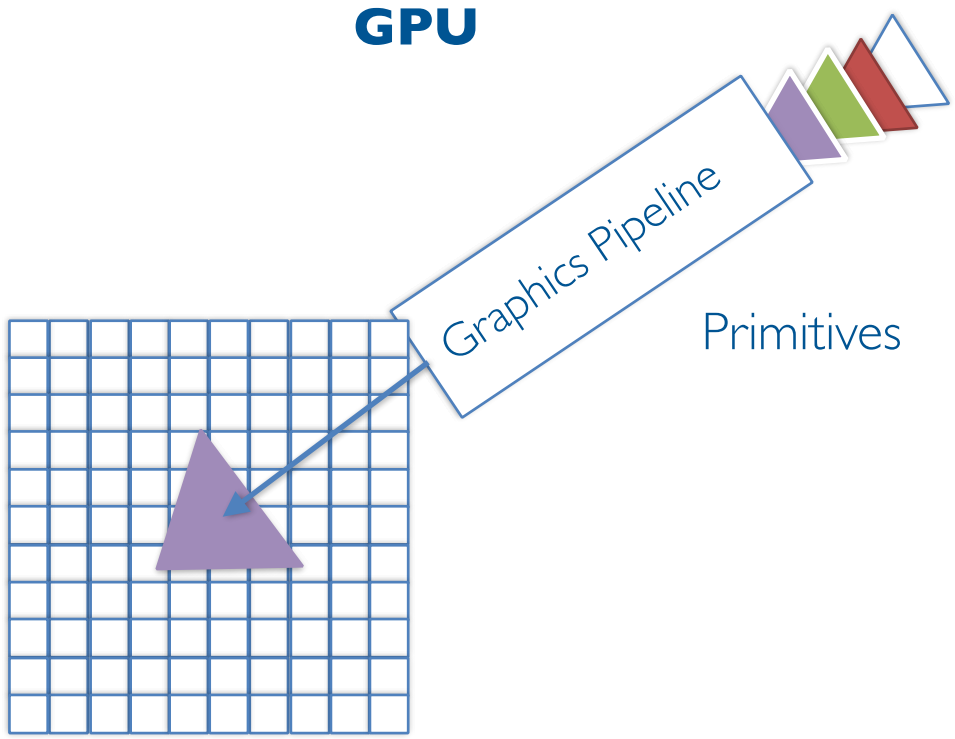
- Points
- Lines
- Triangles

- Implemented in hardware (GPU)



GPUs do Rasterisation

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



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

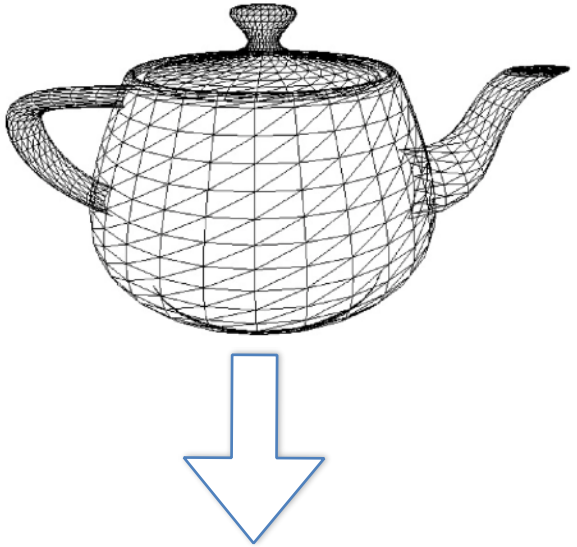
Mapping Vision Concepts to Graphics

Vision	Graphics
[R|t]	View matrix
K	Projection matrix
Triangulated points	Vertex buffer
Reprojection	Rasterization
Image	Framebuffer

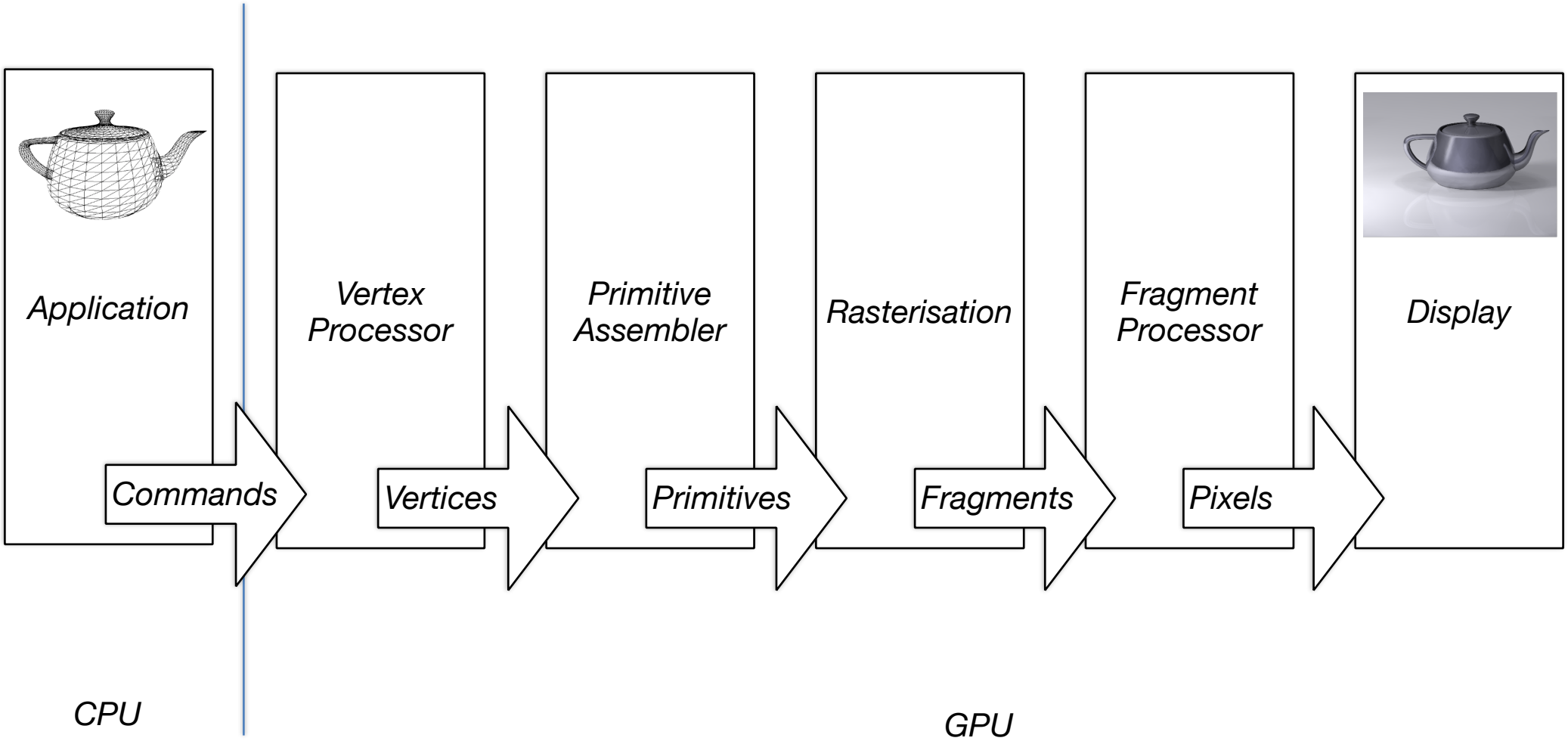
The math and data structures are equivalent -> only the implementation changes.

Graphics Pipeline

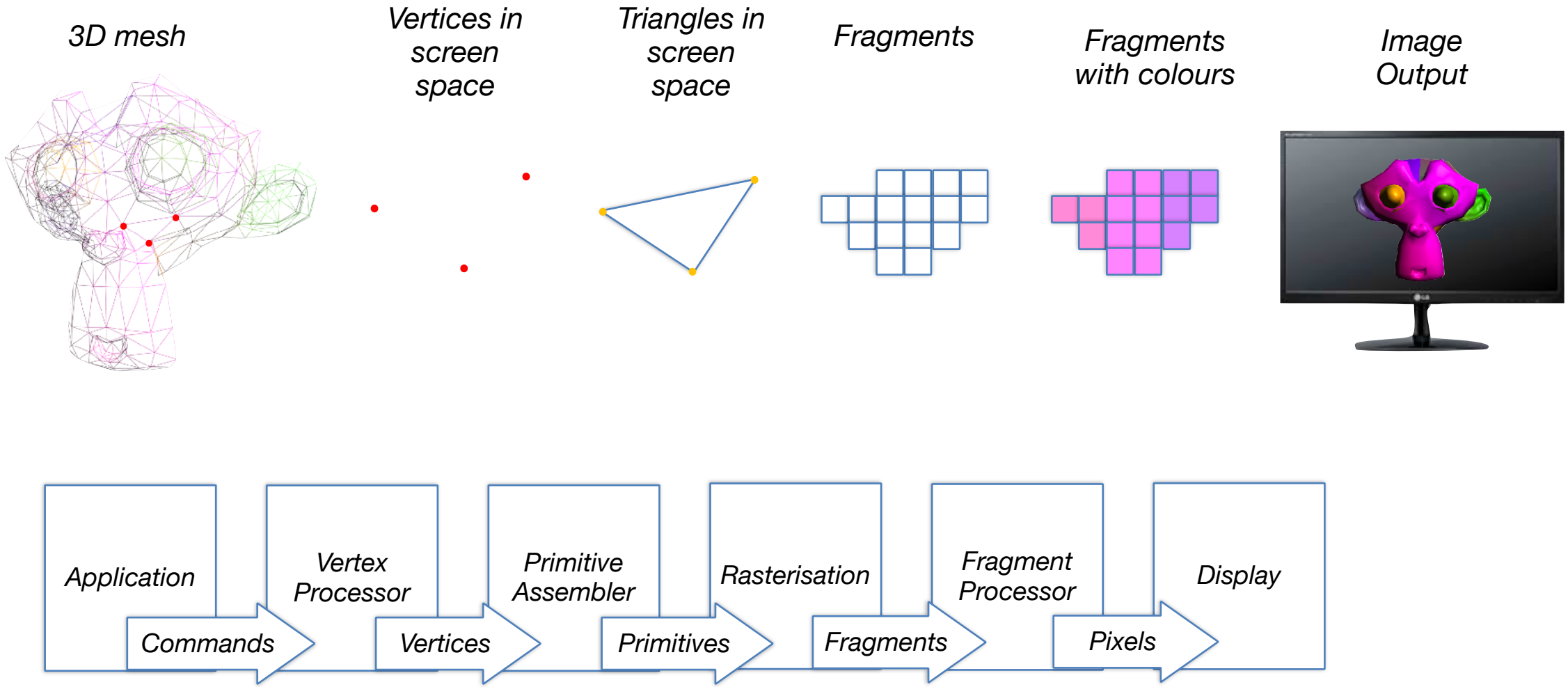
- Input
  - Geometric model
  - Vertices, normals, texture coordinates
  - Lighting/shading model
  - Light positions
  - View point and virtual camera configuration
- Output
- Colour (and depth) per pixel on a screen



Graphics Pipeline



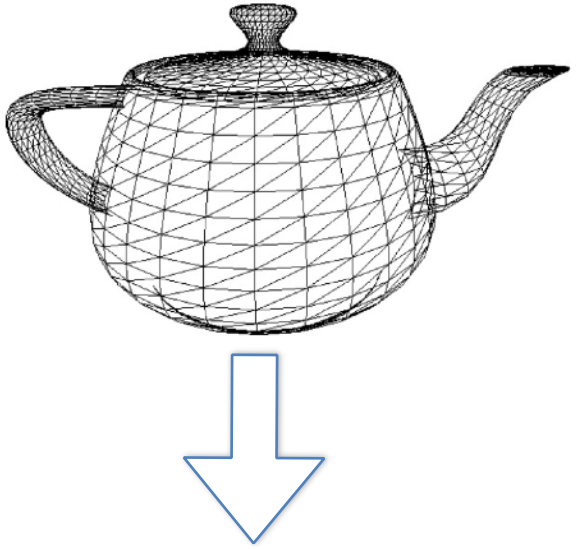
Graphics Pipeline



The Common Core: Projective Geometry

- $x = K[R \mid t]X$
  - Same equation defines the **vertex transformation** in Computer Graphics.
  - In computer vision we **estimate** these parameters; in graphics we **apply** them.

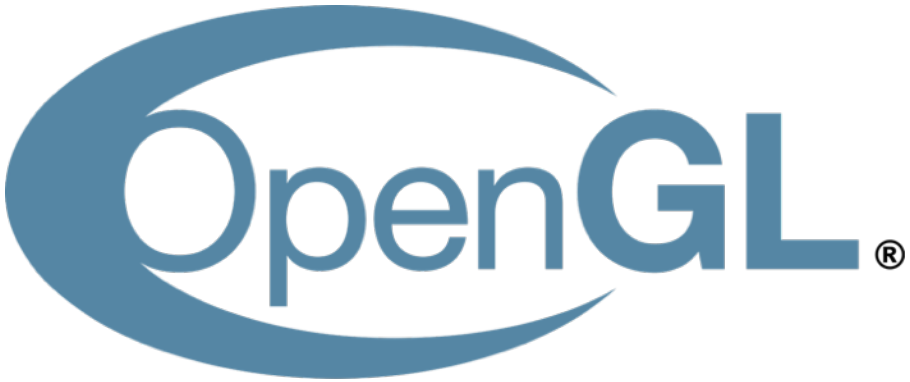
- Note:
  - Same intrinsic and extrinsic parameters.
  - Only the direction of computation changes.

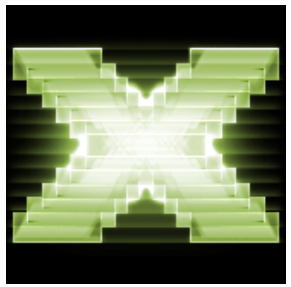


Let’s Get Practical

How to talk to the Graphics Hardware?

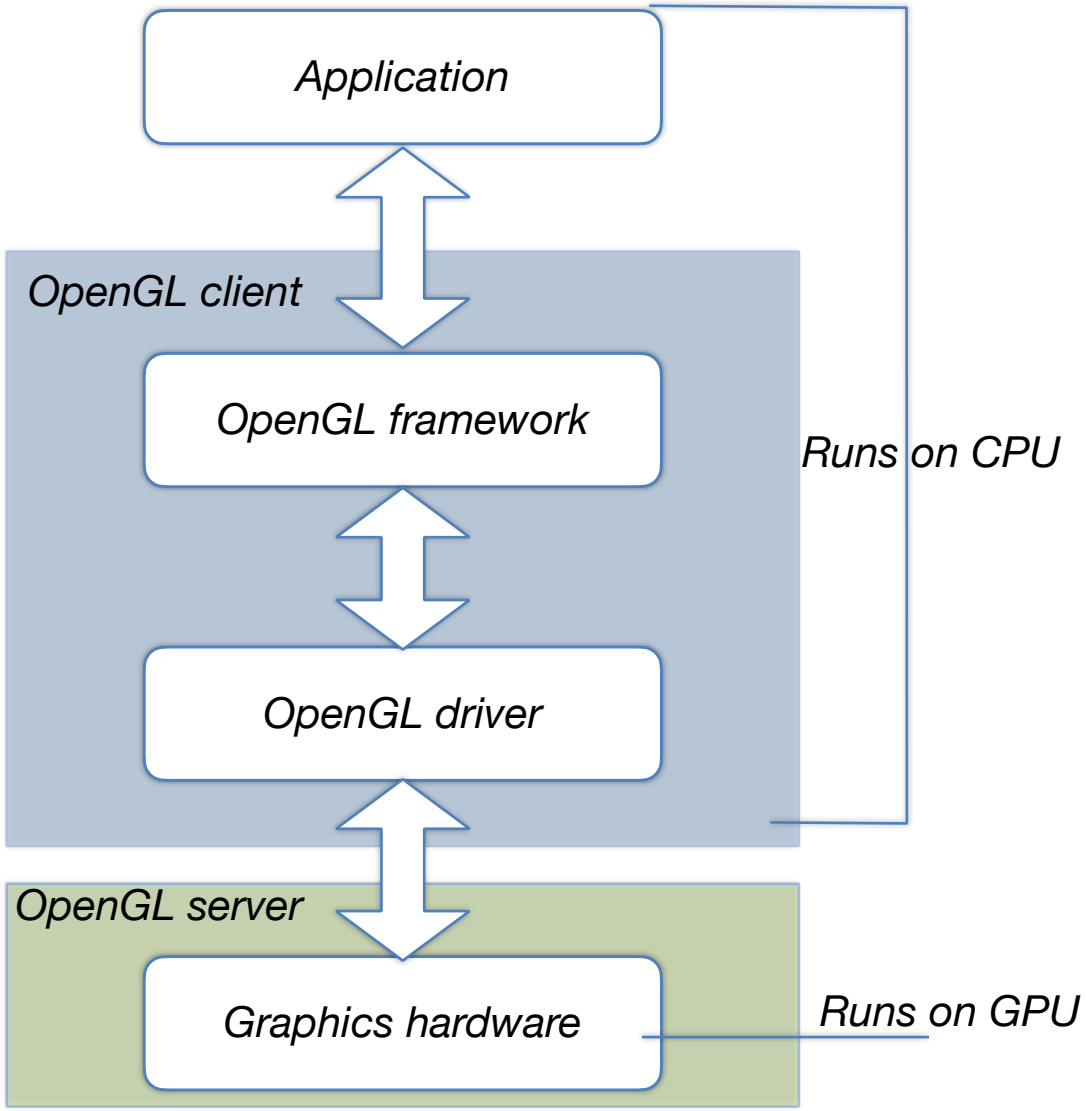
Graphics APIs





Real-Time Graphics

- 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



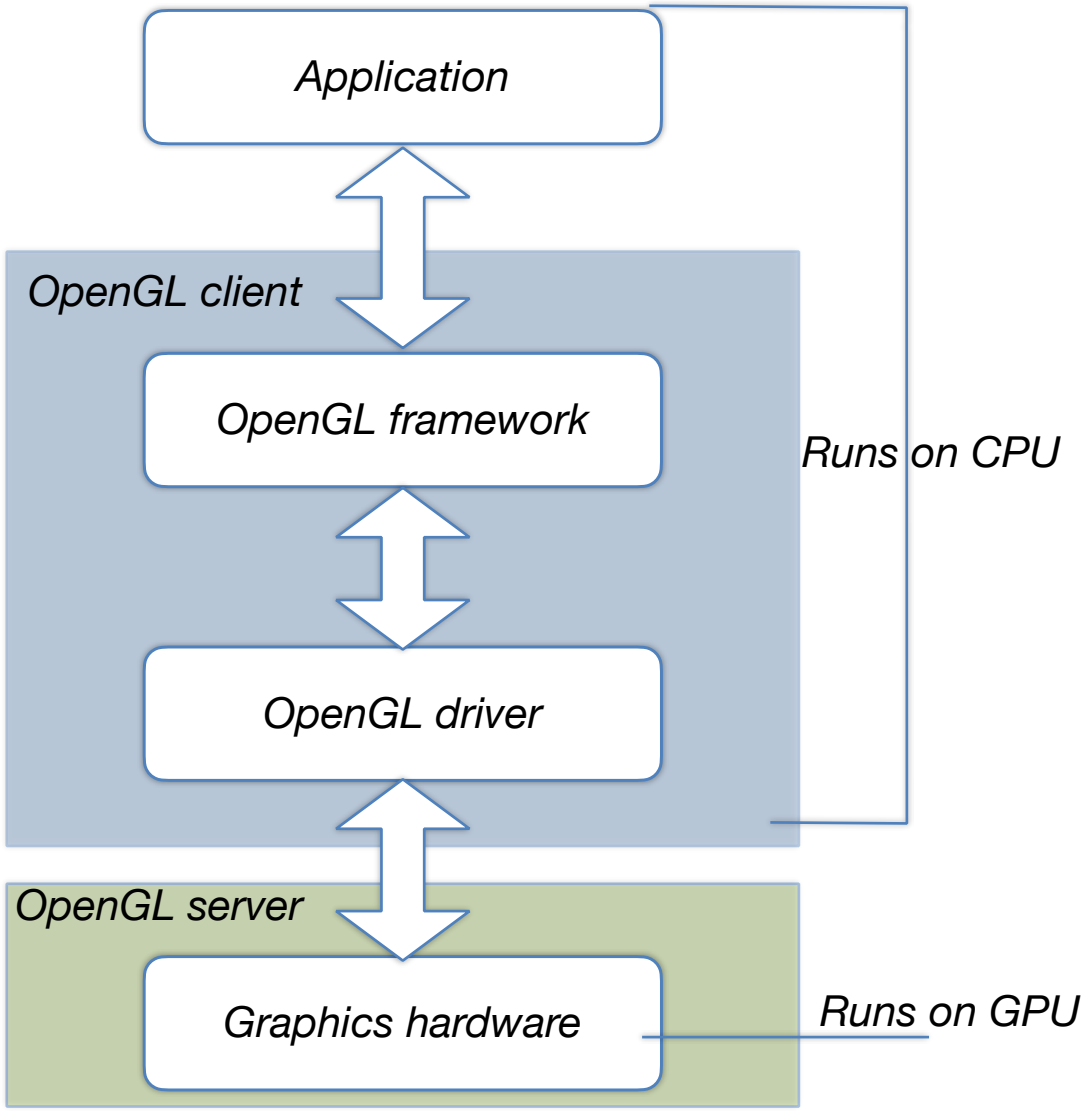
Real-Time Graphics

Important to note:

- The API is defined as a set of functions
  - Drawing commands

```
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableVertexAttribArray(0);
```

  - Working with identifier: 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
- 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, current is 4.6)

OpenGL Concepts

- OpenGL Context
  - Represents an instance of OpenGL
  - Context stores all of the state associated with this instance of OpenGL
  - A process can have multiple contexts
    - Each represent separate viewable surface (e.g. a window)
    - Each has own OpenGL Objects
    - Multiple contexts can share resources
- State
- OpenGL Object Model

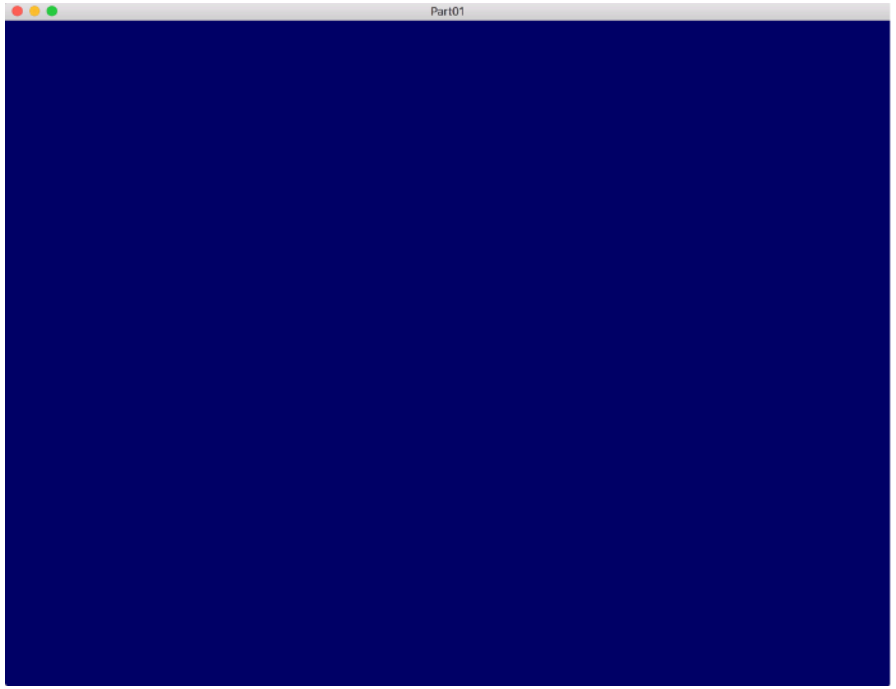
OpenGL Context

Context creation with GLFW (Graphics Library Framework: library for creation and management of windows with OpenGL contexts)

```
 // Open a window and create its OpenGL context
window = glfwCreateWindow( 1024, 768, windowName.c_str(), NULL, NULL);
if( window == NULL ){
          fprintf( stderr, "Failed to open GLFW window. \n” );
      getchar();
      glfwTerminate();
      return false;
}
// set the context as current
glfwMakeContextCurrent(window);
```

OpenGL Context

Context creation with GLFW (Graphics Library Framework: library for creation and management of windows with OpenGL contexts)



OpenGL State

- Information that the context contains and that is used by the rendering system
- A piece of state is simply some value stored in the OpenGL context
- OpenGL as "state machine"
- When a context is created, state is initialised to default values

```
// Enable blending
glEnable(GL_BLEND);

// Disable blending
glDisable(GL_BLEND);
```

Examples

Object Model

- OpenGL is “object oriented”
- Object instances are identified by a name
    - Unsigned integer handle (GLuint)
    - References that identify an object (no pointers)
- Commands work on targets
    - Each target has an object currently bound to the target
    - To modify objects, you must first bind them to the OpenGL context, then execute command

```
GLuint m_textureID;
// Create texture
glGenTextures(1, &m_textureID);

// "Bind" texture
glBindTexture(GL_TEXTURE_2D,
m_textureID);
```

GLuint

Object Model

- OpenGL is “object oriented”
- Object instances are identified by a name
    - Unsigned integer handle (GLuint)
    - References that identify an object (no pointers)
- Commands work on targets
    - Each target has an object currently bound to the target
    - To modify objects, you must first bind them to the OpenGL context, then execute command

```
GLuint m_textureID;
// Create texture
glGenTextures(1, &m_textureID);

// "Bind" texture
glBindTexture(GL_TEXTURE_2D,
m_textureID);
```

GLuint

```
//specifies textures
glTexImage2D(GL_TEXTURE_2D, 0,
GL_RGB, width, height, 0, GL_RGB,
        GL_UNSIGNED_BYTE, image);
```

Object Model

- OpenGL is “object oriented”
- Object instances are identified by a name
  - Unsigned integer handle (GLuint)
  - References that identify an object (no pointers)
- Commands work on targets
  - Each target has an object currently bound to the target
  - To modify objects, you must first bind them to the OpenGL context, then execute command

```
GLuint m_textureID;
// Create texture
glGenTextures(1, &m_textureID);

// "Bind" texture
glBindTexture(GL_TEXTURE_2D,
m_textureID);
```

GLuint

Object oriented?

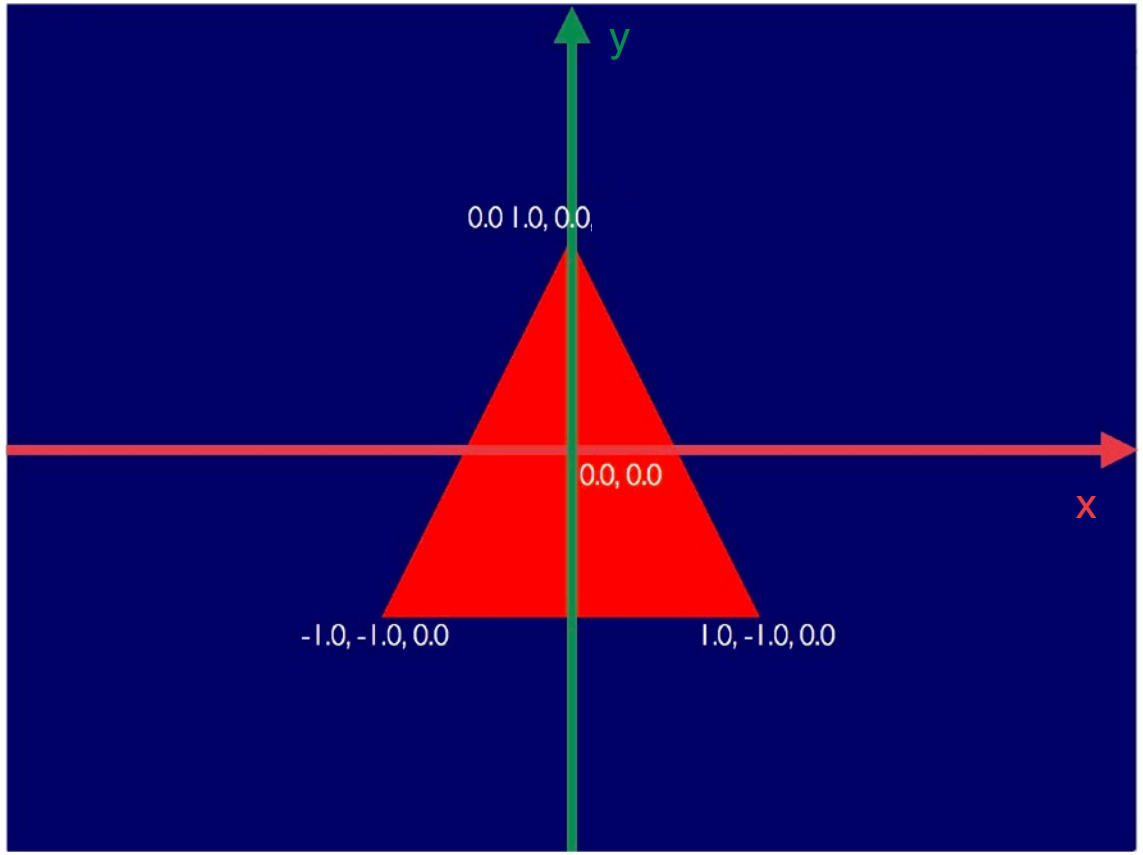
- target⇔type
- commands⇔methods

OpenGL Objects

- Act as
  - Sources of input
  - Sinks for output
- Examples:
  - Buffer objects
    - Unformatted chunks of memory
    - Can store vertex data (VBO) or pixel data, etc.
  - Textures
    - 1D, 2D, or 3D arrays of texels
    - Can be used as input for texture sampling
  - Vertex Array Objects
    - Stores all of the state needed to supply vertex data (vertex data + format)
  - Framebuffer Objects
    - User-defined framebuffers that can be rendered to

Example: Vertex Buffer Object

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



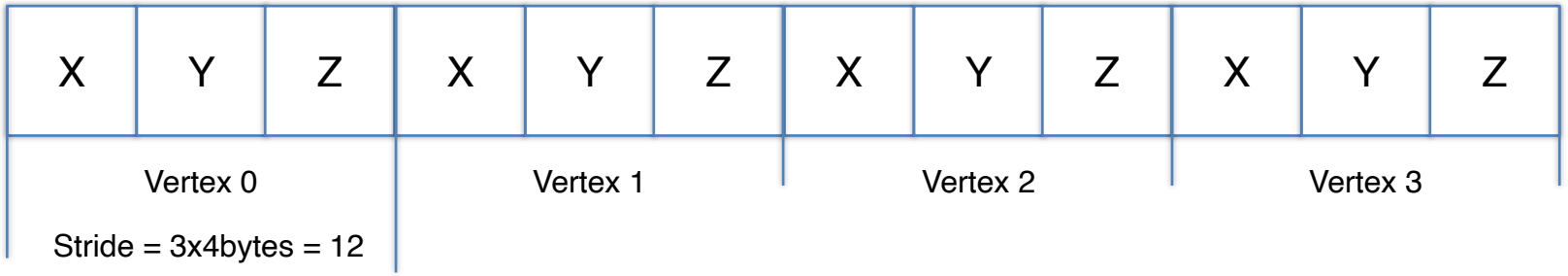

```
// Representation of the 3 vertices of
our triangle
// An array of 3 vectors each consisting
of x,y,z
    static const GLfloat data[] = {
        -1.0f, -1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        0.0f,  1.0f, 0.0f,
    };
```

Example: Vertex Buffer Object

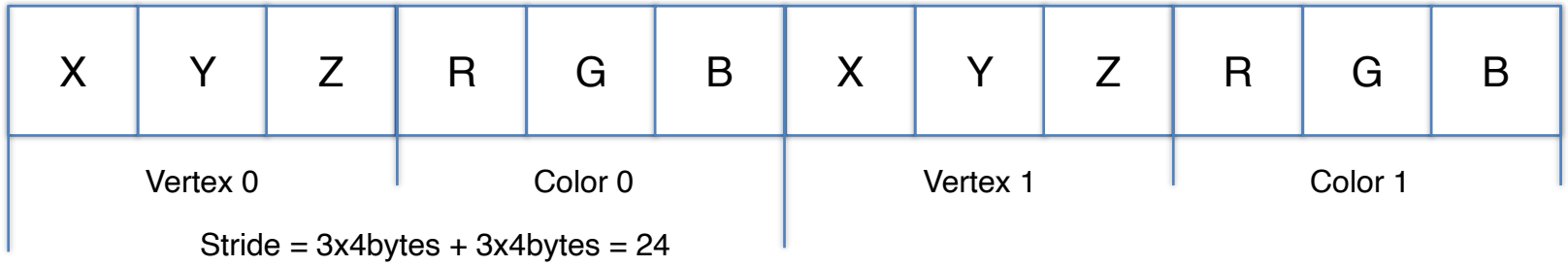
```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                          0,                  // attribute
                          3,                  // size
                          GL_FLOAT,           // type
                          GL_FALSE,           // normalized?
                          0,                  // stride - 0= tightly packed
                          (void*)0            // array buffer offset
                          );
//  - - - - - - - - - - - - - Actual drawing call - - - - - - - -
  glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

Stride

- Specifies the byte offset between consecutive generic vertex attributes (if stride equals 0 -> means tightly packed)
  - Tightly packed:



  - Interleaved:



Example: Vertex Buffer Object

```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                          0,                  // attribute
                          3,                  // size
                          GL_FLOAT,           // type
                          GL_FALSE,           // normalized?
                          0,                  // stride - 0= tightly packed
                          (void*)0            // array buffer offset
                          );
//  - - - - - - - - - - - - - Actual drawing call - - - - - - - -
  glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

Draw Call

- After creating and loading data:
  - We use the draw call to actually draw something

```
glDrawArrays(GL_TRIANGLES, 0, m_vertices.size());
```

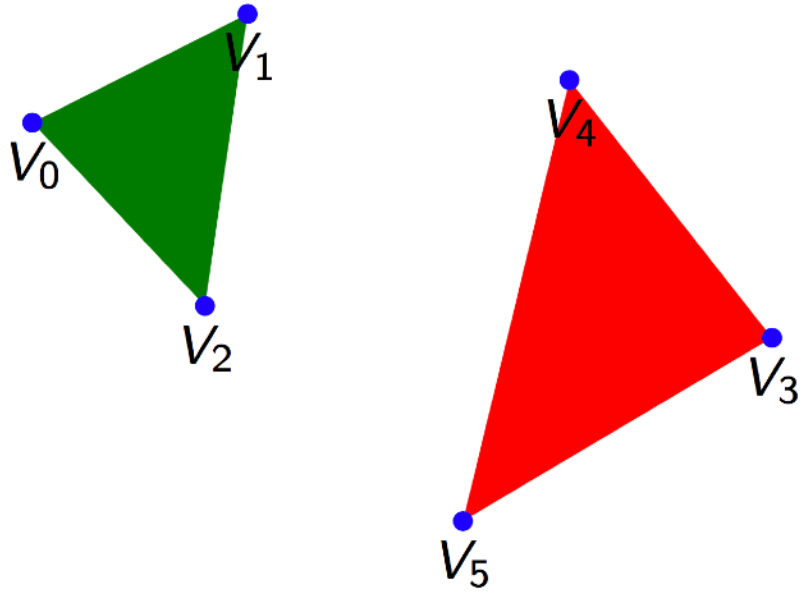
mode specifies what kind of primitives to render. e.g. GL_TRIANGLES

specifies the starting index in the enabled arrays.

count specifies the number of indices to be rendered.

Primitive Types

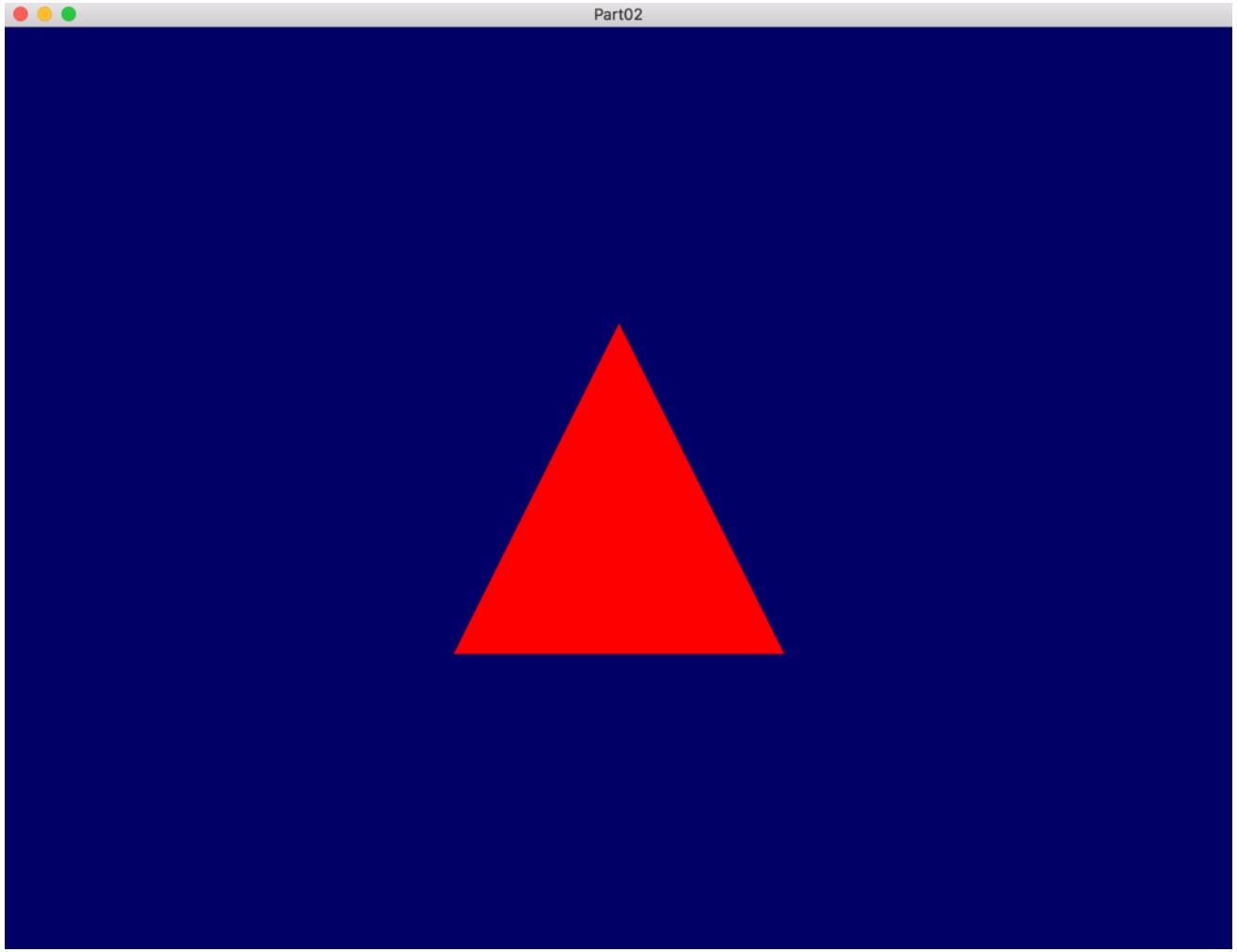
- Points
- Lines
- Triangles
- Line strips
- Lineloops
- Triangle strips
- Triangle fans



Example: Vertex Buffer Object

```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here
 // Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                          0,                  // attribute
                          3,                  // size
                          GL_FLOAT,           // type
                          GL_FALSE,           // normalized?
                          0,                  // stride
                          (void*)0            // array buffer offset
                          );
//  - - - - - - - - - - - - - Actual drawing call - - - - - - - -
  glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

RESULT: Vertex Buffer Object

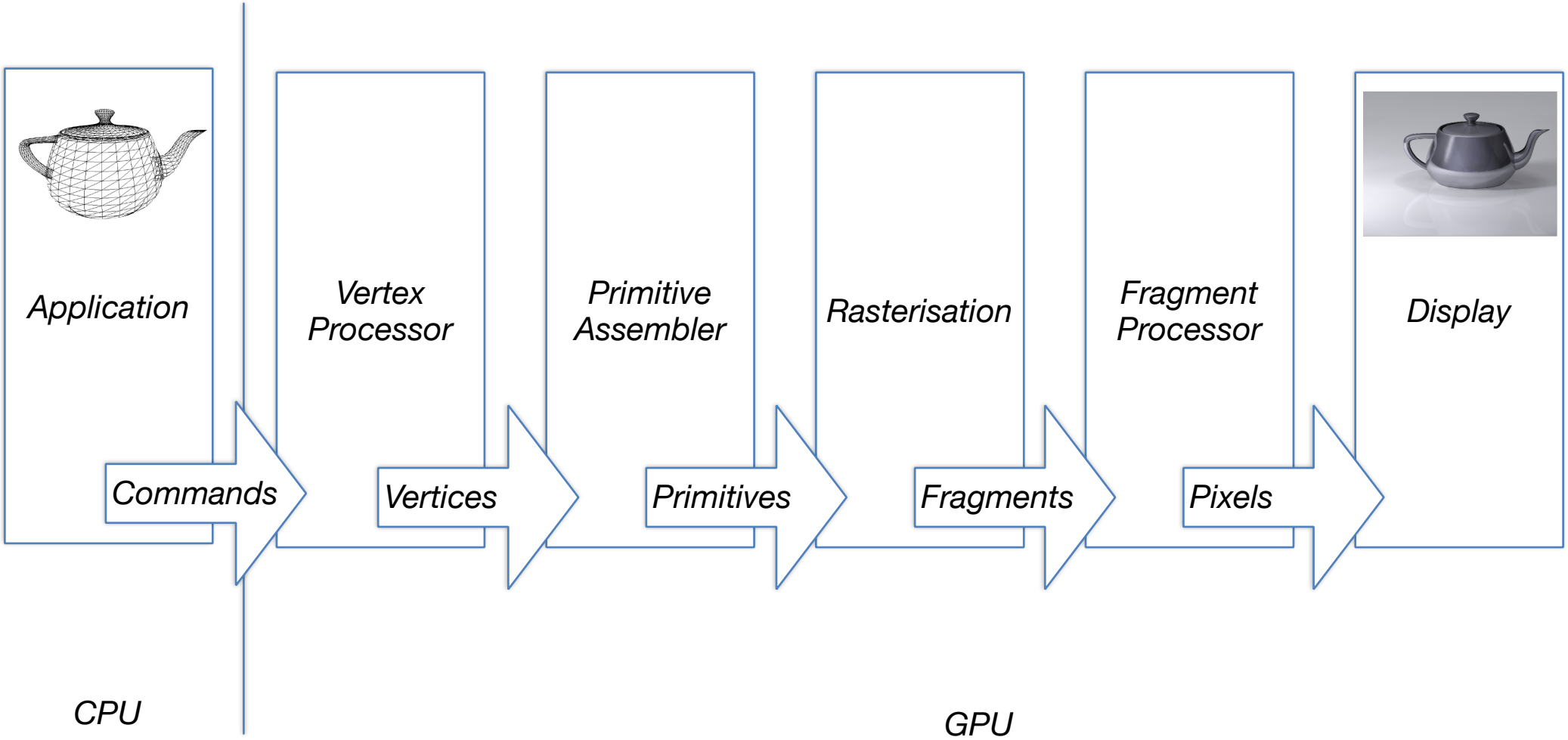


First Triangle

```
//  - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                          0,                  // attribute
                          3,                  // size
                          GL_FLOAT,           // type
                          GL_FALSE,           // normalized?
                          0,                  // stride - 0= tightly packed
                          (void*)0            // array buffer offset
                          );
//  - - - - - - - - - - - - Actual drawing call - - - - - - - -
  glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

Example in Triangle.cpp

So Far - Where are we?



Shader

- Programs implementing the programmable parts of the pipeline
- Parts of a pipeline
  - Vertex Shader
  - Fragment Shader
  - Others (e.g. Geometry Shader, Tessellation Shader)
- Name originates from small programs used to calculate the

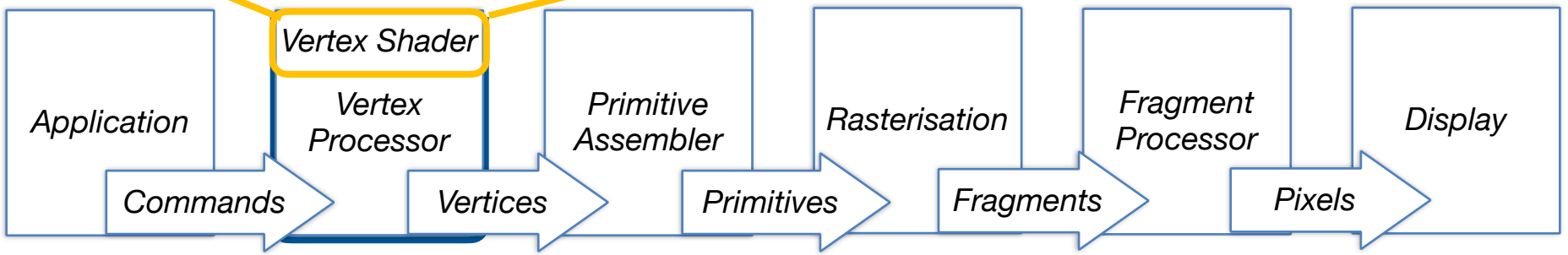
Shader

Shading languages:

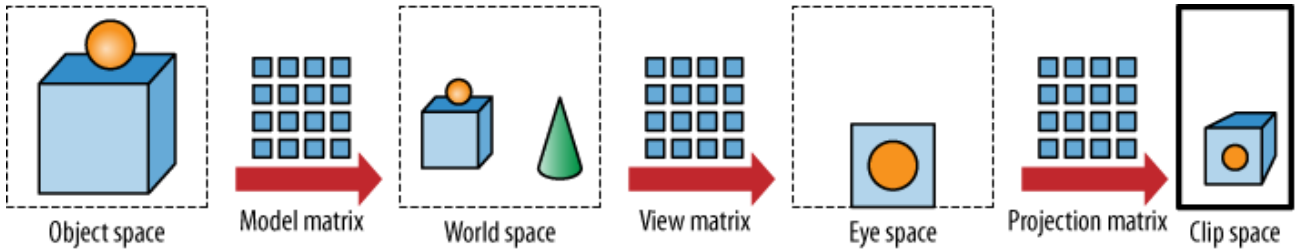
- GLSL
  - OpenGL Shading Language
  - C-like syntax
- HLSL
  - High-Level Shader Language
  - Developed by Microsoft for Direct3D
- CG
  - C for graphics

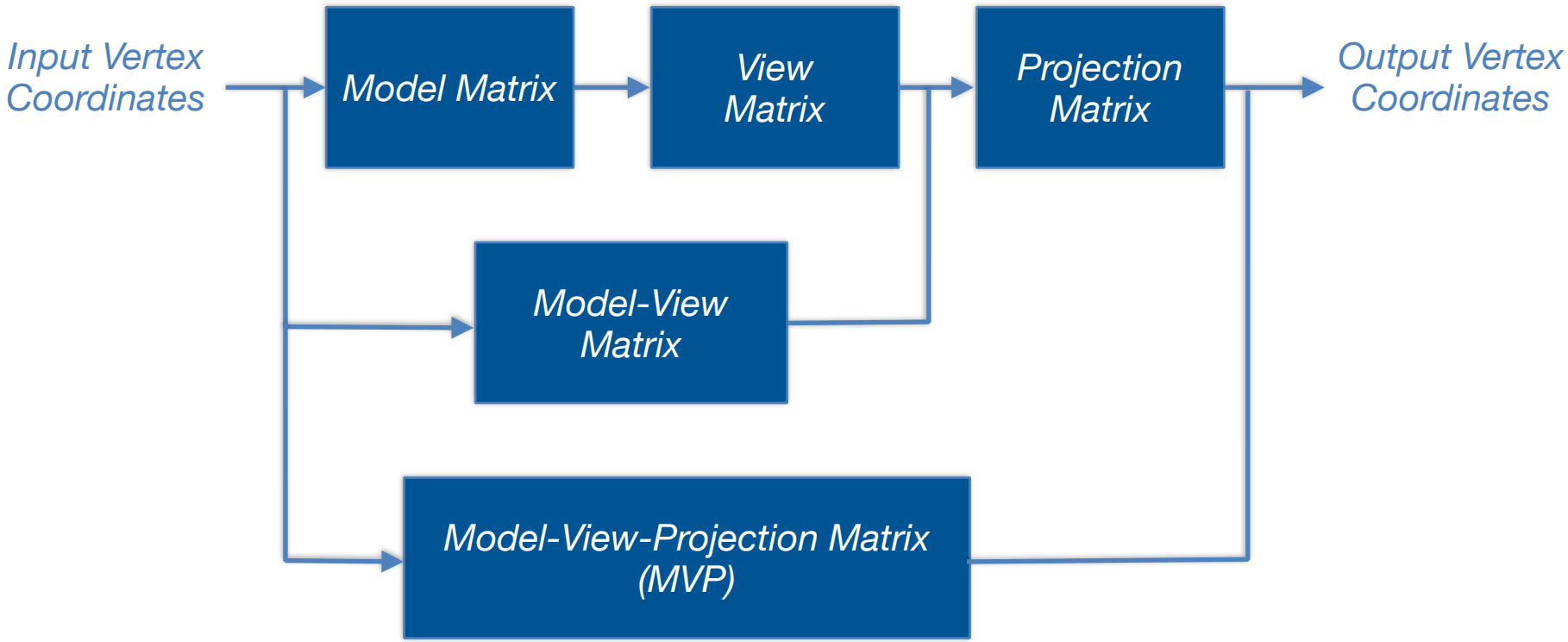
Vertex Processing

- 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

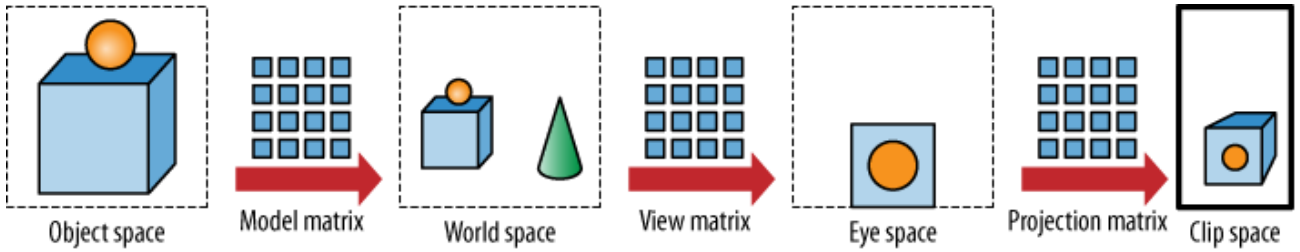


Vertex Shader: Transformations



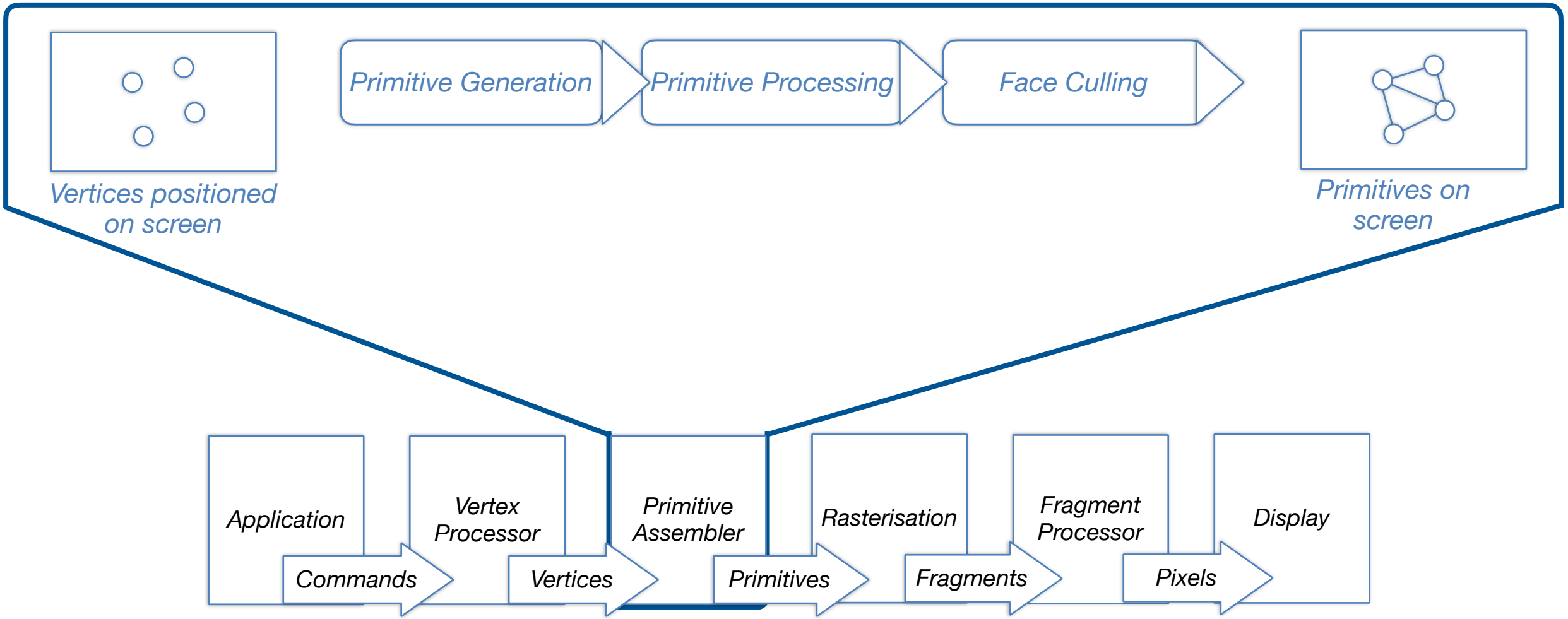


Vertex Shader: Example




```
// 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);
}
```

Primitives Assembly



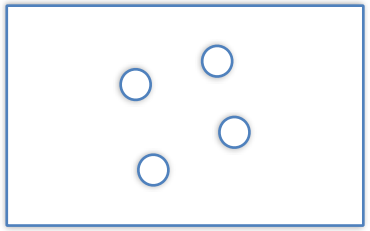
Primitives Assembly

Vertices positioned on screen

Primitives on screen

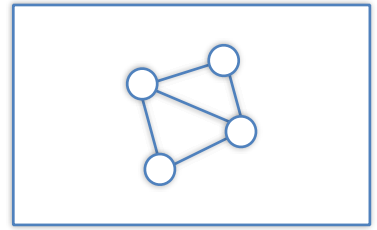
- Primitive assembly receives processed vertices, and the vertex connectivity information as input
- Divides them into a sequence of individual base primitives
- Primitives now have a certain facing
- Face culling discards faces based on their facing

Primitives Assembly: Primitives

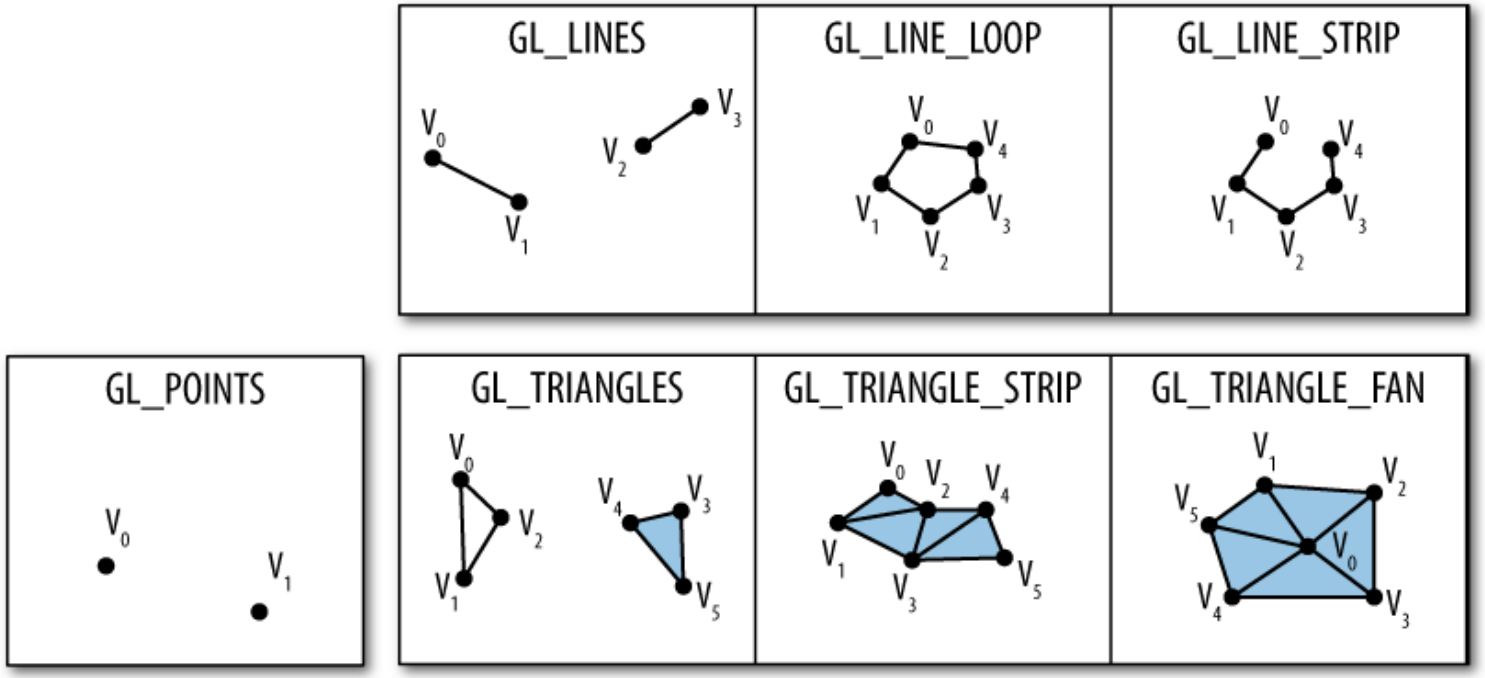


Vertices positioned on screen

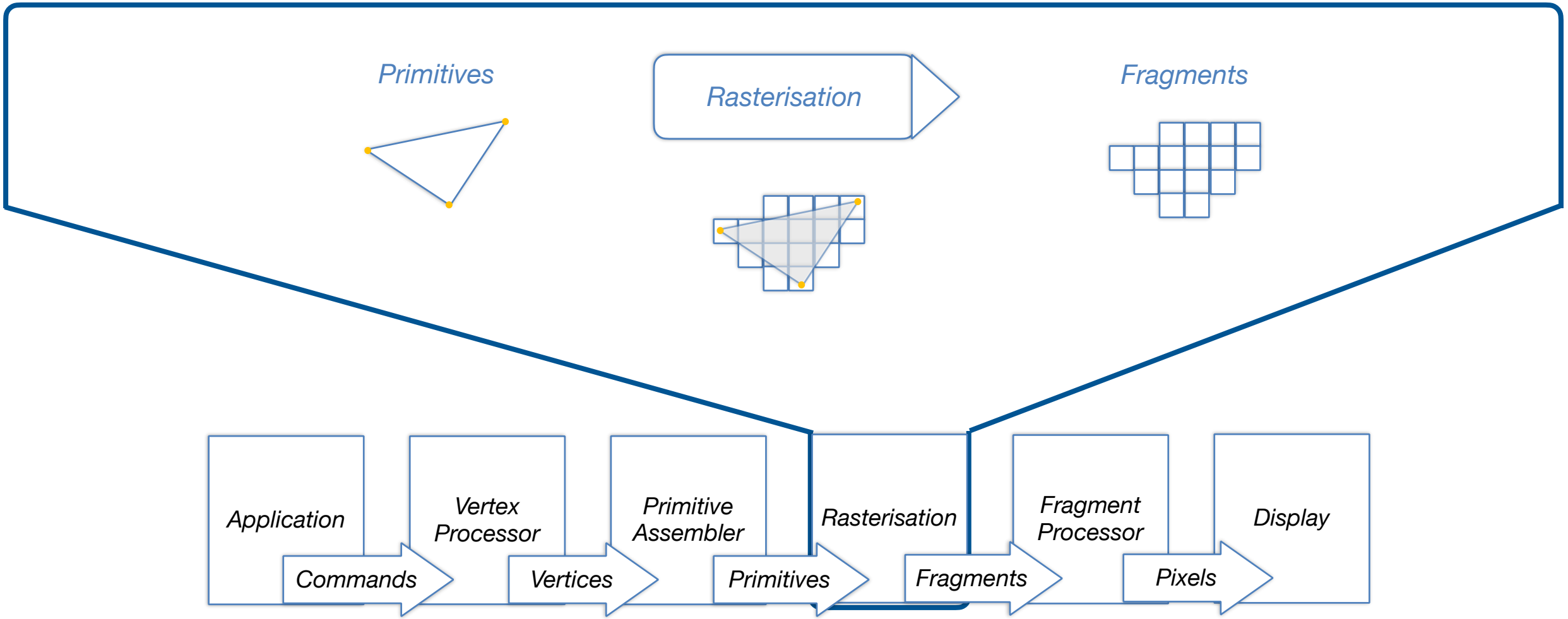
Primitive Generation → *Primitive Processing* → *Face Culling*



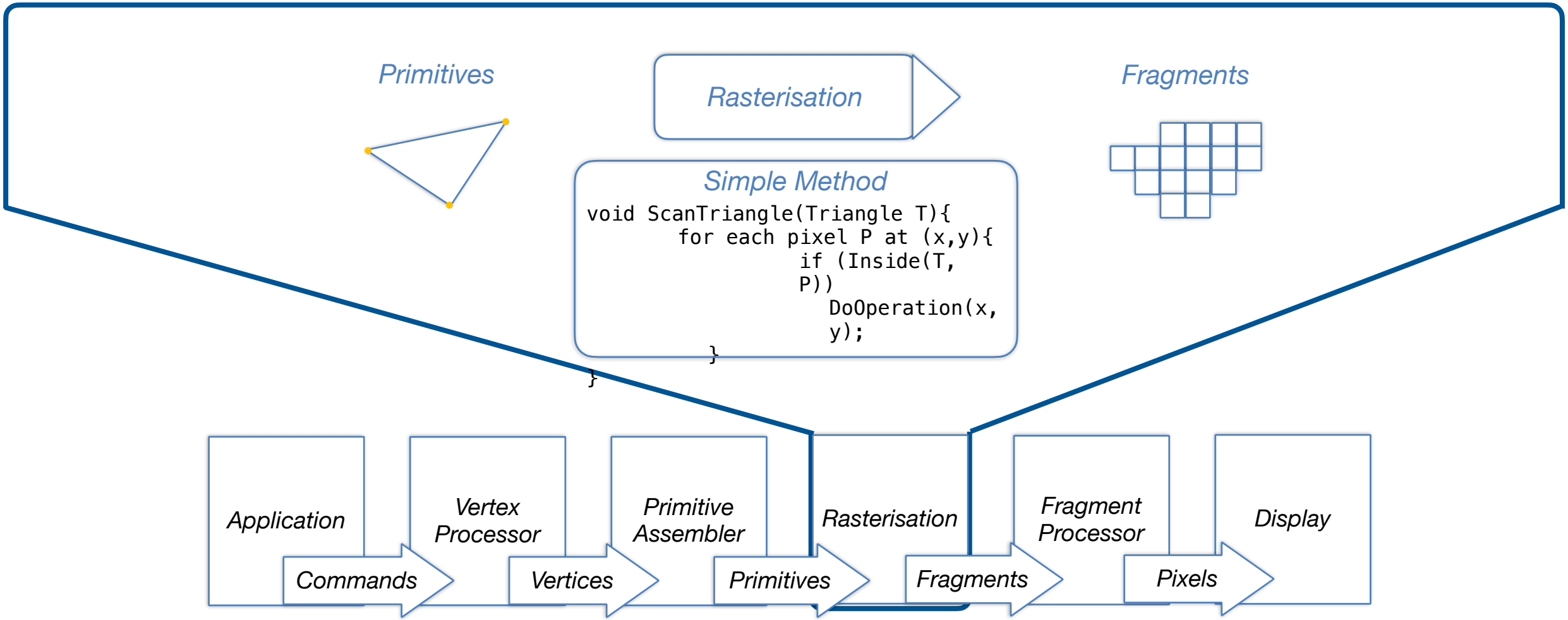
Primitives on screen



Rasterisation

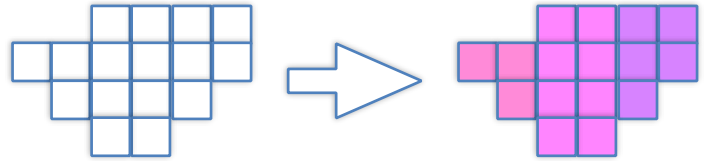


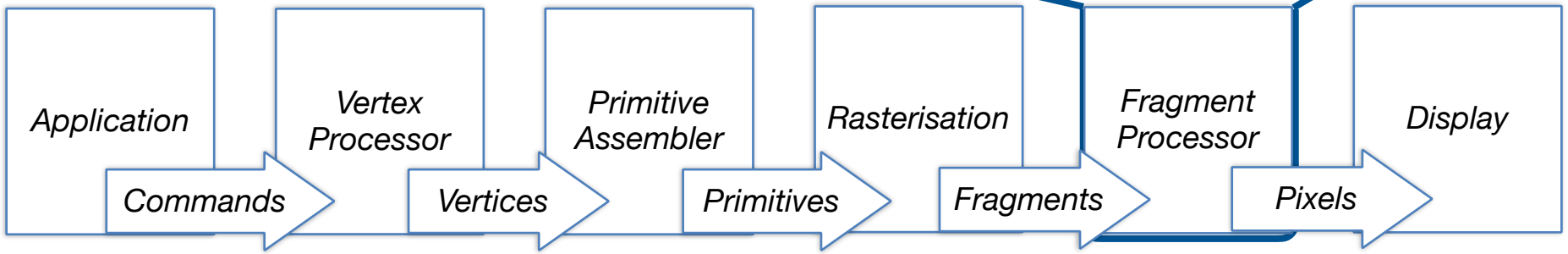
Rasterisation



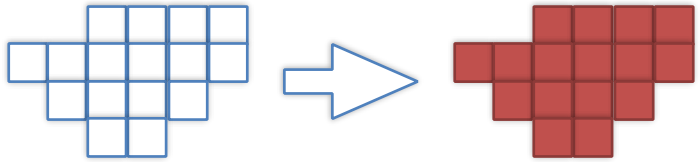
Fragment Processing

- 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;

}
```

The end!