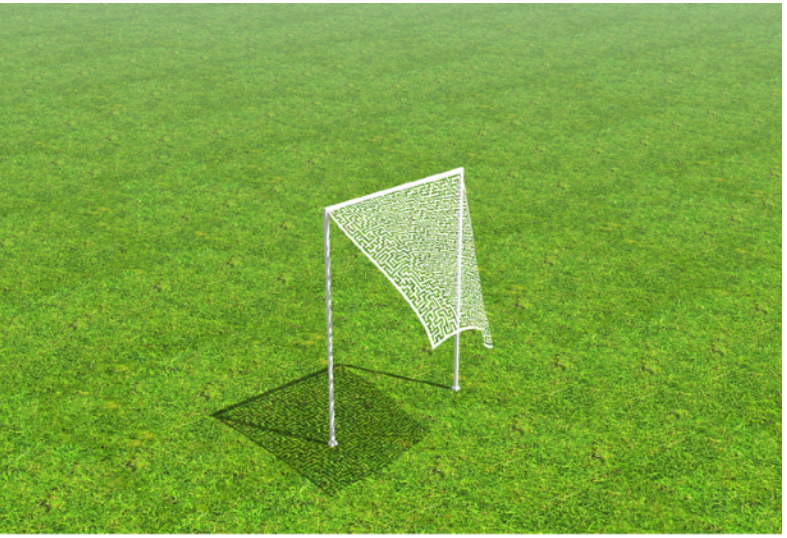
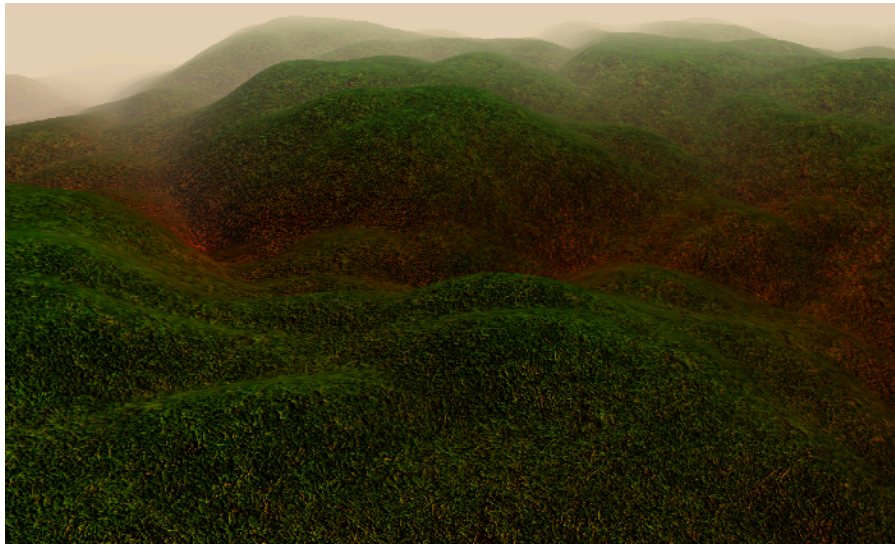
Visual Computing I:

Interactive Computer Graphics and Vision





Computer Graphics Introduction

Stefanie Zollmann and Tobias Langlotz

3D Geometry

Translation and Scaling in 3D

- Translation by $(\Delta x,\ \Delta y,\ \Delta z)$
- Scaling by Scaling by a factor s

$$\mathbf{x}' = \mathbf{T}\mathbf{x} \qquad \mathbf{x}' = \mathbf{S}\mathbf{x}$$

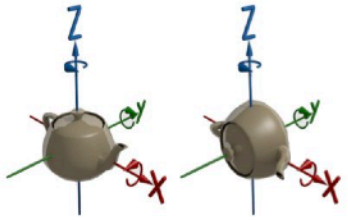
$$\mathrm{T} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathrm{S} = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

13

Rotation about the X Axis

- From Y to Z
  - X co-ordinate is unchanged

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

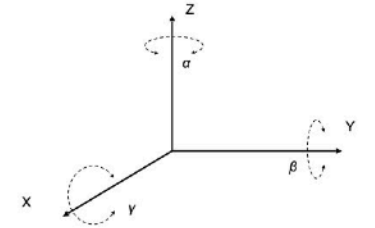


Rotation by 90° about the Z axis – from X to Y

18

Euler Angles

- Rotation defined by angles of rotation around the X-, Y-, and Z- axes
- Different conventions
  - Example Blender Rotation Mode:



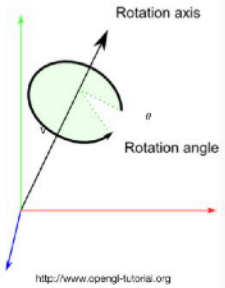
$$\mathbf{R} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_Z\mathbf{R}_Y\mathbf{R}_X$$

21

Quaternion

- Another way of presenting rotations (avoiding Gimbal Lock)
  - Can be expressed as a scalar and a 3- vector: (a, v)
  - A rotation about the unit vector v by an angle θ can be represented by the unit quaternion



angle in radians, unit-length axis

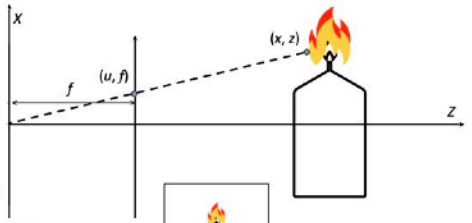
$$q(\theta, v) = \left(\cos\frac{\theta}{2}, v_x\sin\frac{\theta}{2}, v_y\sin\frac{\theta}{2}, v_z\sin\frac{\theta}{2}\right)$$

scalar = angle; vector = axis

30

The Pinhole Camera Model and Projective Geometry

- Removing the sign change
- We can put the image plane in front of the pinhole
  - Removes the sign change
  - Not practical for real cameras
  - The maths works out just fine



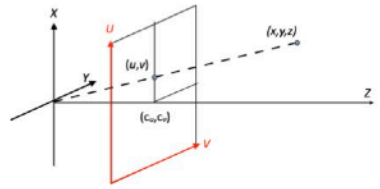
$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

61

Camera Parameters - Intrinsics and Extrinsics

- Often break this down into

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Intrinsic}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{Extrinsic}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



- Most simple case: $\mathbf{u} = \mathbf{K}[\mathbf{I} \mid \mathbf{0}]\mathbf{x}$
- $\mathbf{K}$: camera calibration or **Intrinsic parameters**
- $[\mathbf{I} \mid \mathbf{0}]$: camera pose or **Extrinsic parameters**

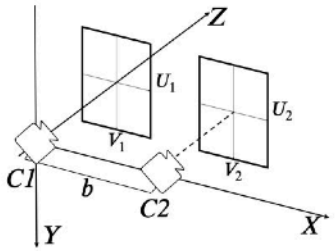
68

Stereo Vision/ Epipolar Geometry

A Simple Stereo System

- Assume a set of 2 pinhole cameras
  $\mathbf{u} = \mathrm{K}[\mathrm{R} \quad \mathbf{t}]\mathbf{x}$
  - Simplify for now: $\mathrm{K} = \mathrm{I}$
- Camera 1 (C1): $\mathrm{R} = \mathrm{I}, \mathbf{t} = \mathbf{0}$
  - No rotation, looks along Z
- Camera 2 (C2): shifted along X

$$\mathrm{R} = \mathrm{I}, \; t = \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}$$



33

Project a Point into the Cameras

- Camera 1:

$$\begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ z \end{bmatrix} \equiv \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}$$

- Camera 2:

$$\begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 & -b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x - b \\ y \\ z \end{bmatrix} \equiv \begin{bmatrix} (x-b)/z \\ y/z \\ 1 \end{bmatrix}$$

35

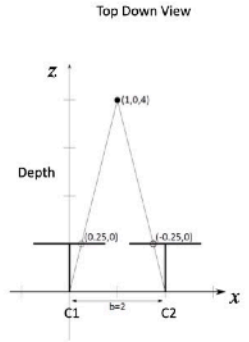
Depth and Disparity

- Difference between the views is the disparity
  - In this case it is in u:

$$u_1 - u_2 = \frac{x}{z} - \frac{x-b}{z} = \frac{b}{z}$$

  - The v values are the same
- If we know b, can find z:

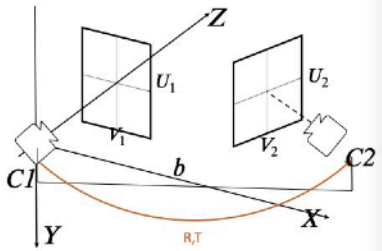
$$z = \frac{b}{u_1 - u_2}$$



36

More General Stereo

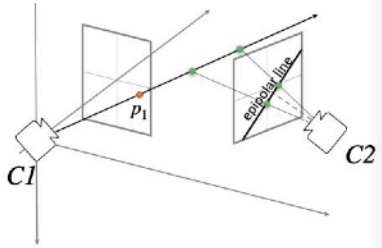
- In the more general case:
  - The cameras have different calibration parameters
  - There is a rotation between the cameras
  - There is translation along all three axes
- We can still align the XYZ coordinate system with camera C1



41

Epipolar Geometry

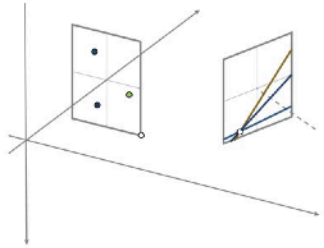
- Consider a point p_1 in the first camera
  - This gives a 3D line, along which the point must lie
  - The 3D line projects to a 2D line in the second image – this is called an **epipolar line**



44

Epipoles and Epipolar line

- Each point in one image makes an epipolar line
  - These all meet at the epipole in the other image
- For simple (parallel) case
  - Line through camera centres is the X axis
  - Epipoles are at infinity
  - Epipolar lines are the rows

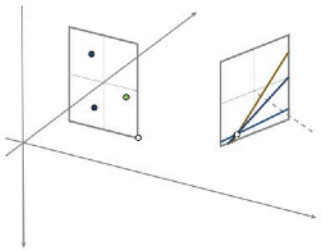


49

Stereo Vision: Fundamental Matrix

The Fundamental Matrix

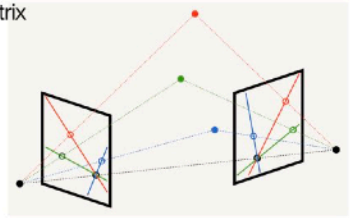
- In general: a point, $\mathbf{p}$, lies on a line, $\boldsymbol{l}$, if $p^T l = 0$
- In our case: $p_2^T l_2 = 0$
- If we now use our epipolar constraint:
  - $l_2 = \mathrm{F} p_1$
- Then $\mathbf{p}_2^\top \mathbf{F} \mathbf{p}_1 = 0$



15

The Essential Matrix

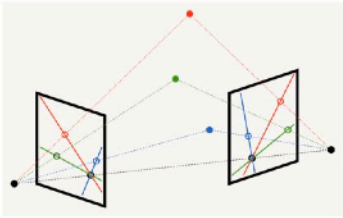
- For calibrated cameras
  - We can factor out K_1 and K_2 in $\mathrm{F} = \mathrm{K}_2^{-T}[\mathbf{t}]_\times \mathrm{R}\mathrm{K}_1^{-1}$
    - This gives us the essential matrix $\mathrm{E} = \mathrm{K}_2^T \mathrm{F} \mathrm{K}_1 = [\mathbf{t}]_\times \mathrm{R}$
  - E has only five DoF:
    - 3 for 3D rotation
    - 3 for 3D translation
    - Less 1 for unknown scale



18

The 8-Point Algorithm

- Estimate F from matches
  - F is a 3×3 matrix – 9 values
  - Defined up to a scale – 8 DoF
  - Need 8 constraints
- F turns out to be rank 2
  - Only 7 degrees of freedom
  - There is a 7-point algorithm



23

The Homogeneous Linear System

$$\begin{bmatrix} u_1u_1' & v_1u_1' & u_1' & u_1v_1' & v_1v_1' & v_1' & u_1 & v_1 & 1 \\ u_2u_2' & v_2u_2' & u_2' & u_2v_2' & v_2v_2' & v_2' & u_2 & v_2 & 1 \\ u_3u_3' & v_3u_3' & u_3' & u_3v_3' & v_3v_3' & v_3' & u_3 & v_3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_nu_n' & v_nu_n' & u_n' & u_nv_n' & v_nv_n' & v_n' & u_n & v_n & 1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

A (Known) $\quad f$ (Unknown)

$$Af = 0$$

26

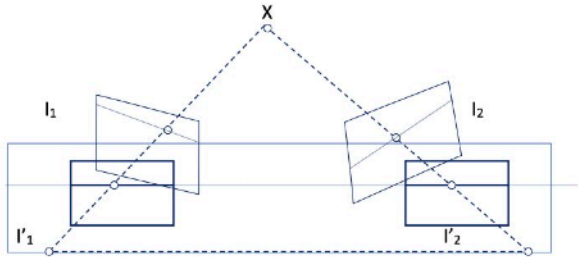
The 7-Point Algorithm

- Exploiting the geometric constraint that the determinant of F must be zero
- The "Two-Solution" Problem: With only 7 points, the system is underdetermined, resulting in two initial 'basis' solutions, let's call them F_1 and F_2
- Finding the Right Mix: The true solution F is a specific blend of these two basis solutions: $F = \alpha F_1 + (1 - \alpha) F_2$
- The Cubic Equation: By enforcing the constraint that det(F)=0, we get a cubic polynomial for the mixing value α

- **Pro**: Requires fewer points
- **Con**: More complex due to solving a cubic polynomial; can have multiple solutions; less robust to noise than the 8-point algorithm with normalisation and RANSAC

30

Stereo Rectification

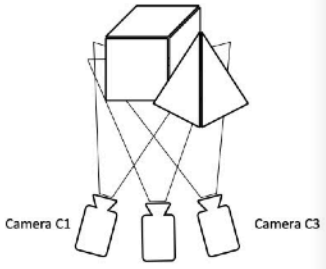


39

Depth Sensing

Passive Range Scanning

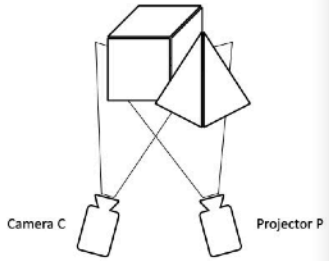
- Two or more cameras
  - Stereo camera
  - Two cameras
  - One camera takes several photos from different perspectives
  - Multiple cameras
- Which one is the easiest and why?
- What is the main challenge?



15

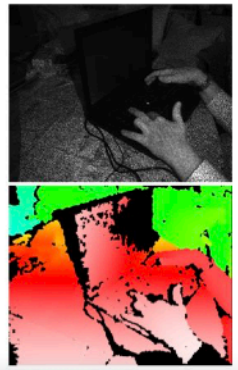
Structured Light Projection

- Many structured light techniques exist that follow the two objectives:
  - Determine projector-camera correspondences with as few images as possible
  - Be robust against surface modulation and noise
- Codes include:
  - Binary codes, such as the Gray code (most common): <100 images, quite robust
  - Intensity codes that apply cosine patterns and phase shifts: 6 images, but are not robust against strong absorptions
  - Color coding: <3 images, but very fragile (does not work on coloured surfaces)



25

Structured Light Projection



MS Kinect 1 (PrimeSense sensor, not MS Kinect 2 uses ToF discussed later)

42

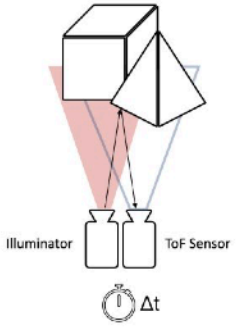
Structured Light Projection

Raskat et al. "Shaderlamps"

33

Optical Time-of-Flight (ToF)

- General idea:
  - Compute depth by round-trip estimation of a light wave emitted and its reflection back to the sensor
  - $D = C \cdot \Delta t/2$
  - $\Delta t = 2D/C$
  - D=1m $\Delta t = 6.6*10^{-09}$ s
  - D=0.01m $\Delta t = 6.6*10^{-11}$ s
  - D=0.001m $\Delta t = 6.6*10^{-12}$ s (pico seconds)
  - 3.3 picoseconds for light to travel 1 millimeter



55

Depth from Mono / Depth Estimation using ML

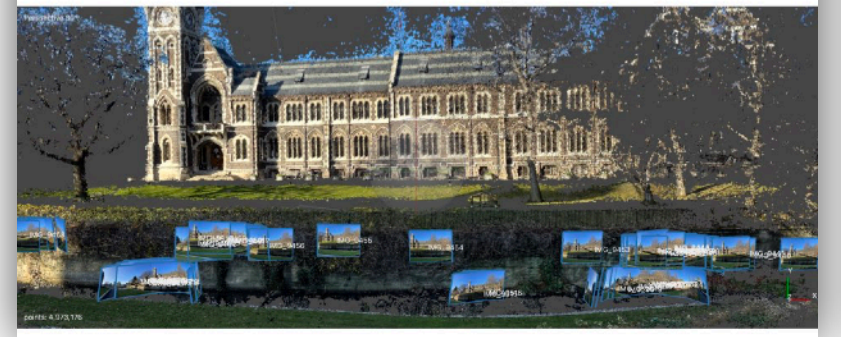
- Convolutional neural networks (CNNs) have been applied to monocular depth estimation
- Common architectures include U-Net, ResNet, and DenseNet
- During training:
  - Network is given pairs of images and their corresponding ground-truth depth maps
  - Minimize the difference between predicted depth and ground truth depth

https://github.com/yuhsuanyeh/BiFuse

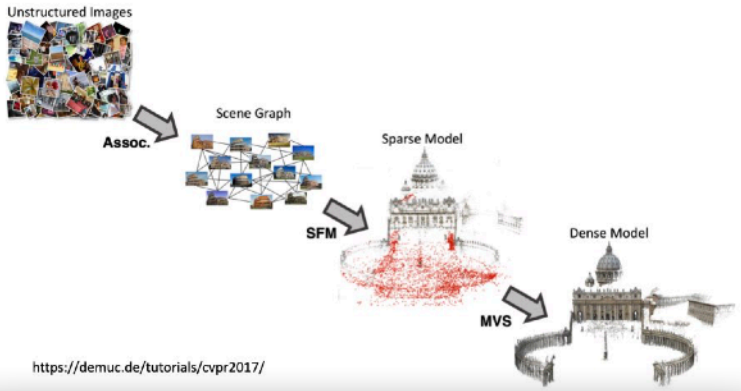
71

Multiview Geometry

Goal

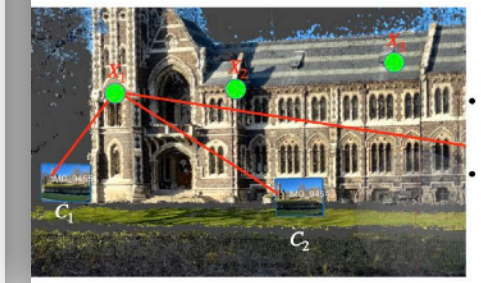


Workflow



https://demuc.de/tutorials/cvpr2017/

Structure-from-Motion

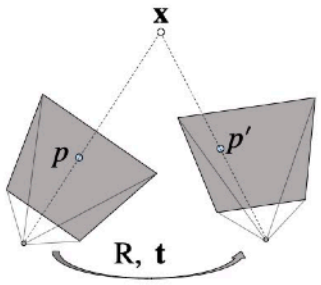


- Estimation of 3D points X_i and camera poses C_j
- Based on motion (camera views from different viewpoints)

43

Decomposing the Essential Matrix

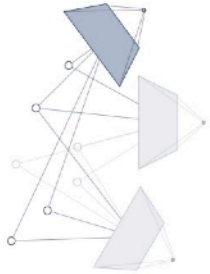
- To recover R and t
  - Compute SVD of E
  - SVD: gives us three matrices U S and V transpose: $\mathbf{E} = \mathbf{USV}^T$
  - From this decomposition, we get:
    - Two possible rotations (R)
    - Two possible translations (t)
  - That means there are four candidate solutions in total.
  - To find the correct one: check which solution places the 3D points in front of both cameras



48

Multi-View Stereo Pipeline

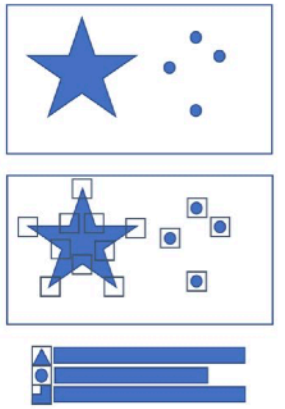
- Identify best matching pair
  - Two-view stereo for relative pose
  - Reconstruct 3D points
  - Minimise reprojection error
- While still images to add
  - Select next best view
  - Determine absolute pose
  - Reconstruct more 3D points
  - Minimise reprojection error



63

Bags of Words (BoW)

- Need equivalent of words
- Cluster features – can pre-train
  - Choose some number, k, of 'words'
  - Make k clusters – e.g. k-means
  - Cluster centres become words
- Bag of words for images
  - Detect and describe feature
  - Count features closest to each word

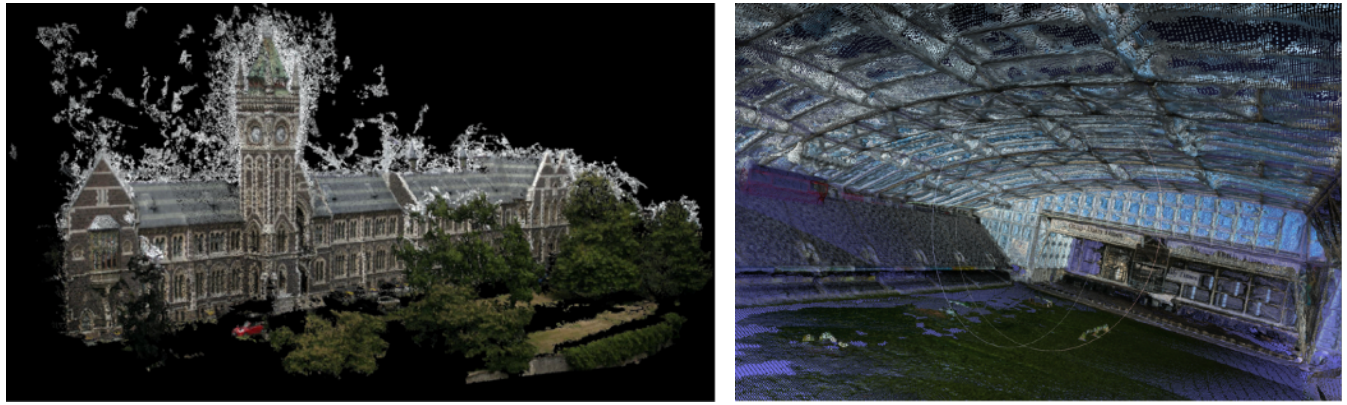


69

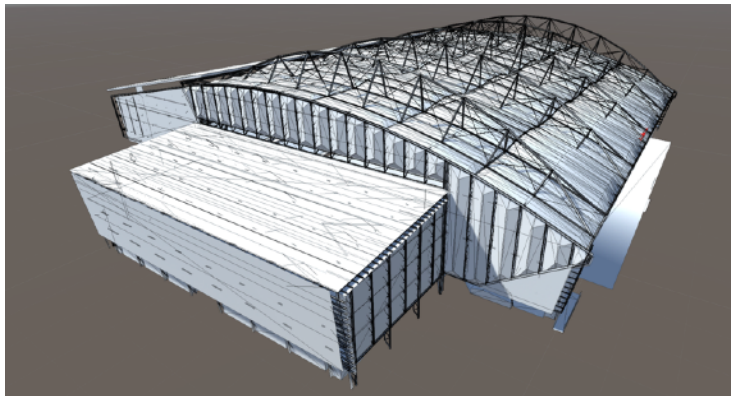
Quiz

Computer Graphics

What to do with all this data?



Point Clouds



3D Mesh data

3-Min Discussion:

Why not just keep doing what we did so far and use image processing tools to render our 3D models? What challenges would you expect?

Applications for real-time graphics

Games

Augmented Reality

Virtual Reality

Visualization

Real-Time Graphics

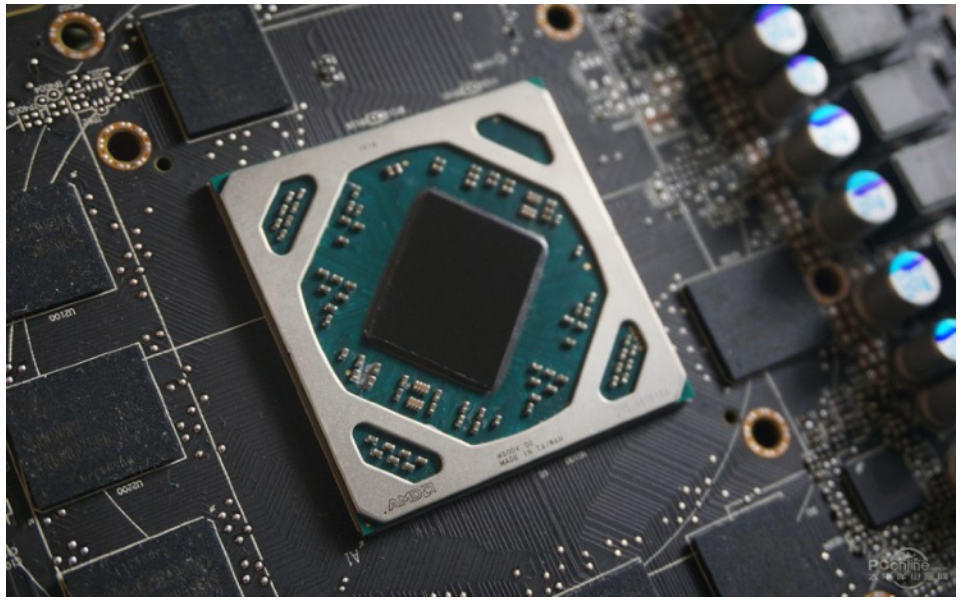
Real-Time Graphics

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



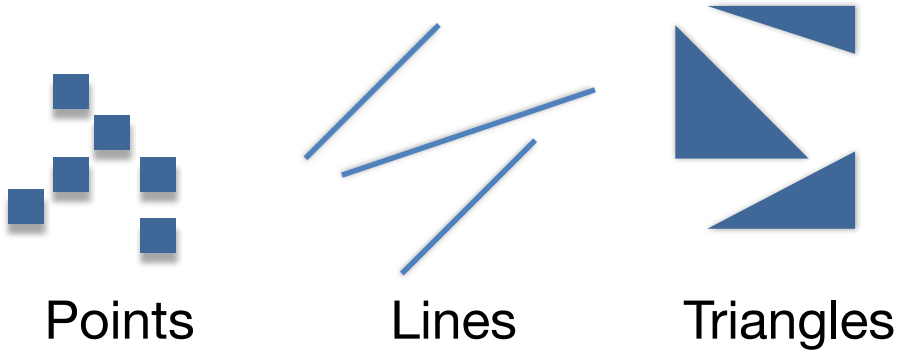
By Shawn Knight - http://www.techspot.com/news/65328-amd-radeon-rx-480-benchmarks-bare-pcb-photos.html, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=54979727

Real-Time Graphics

Based on rasterisation of graphic primitives:

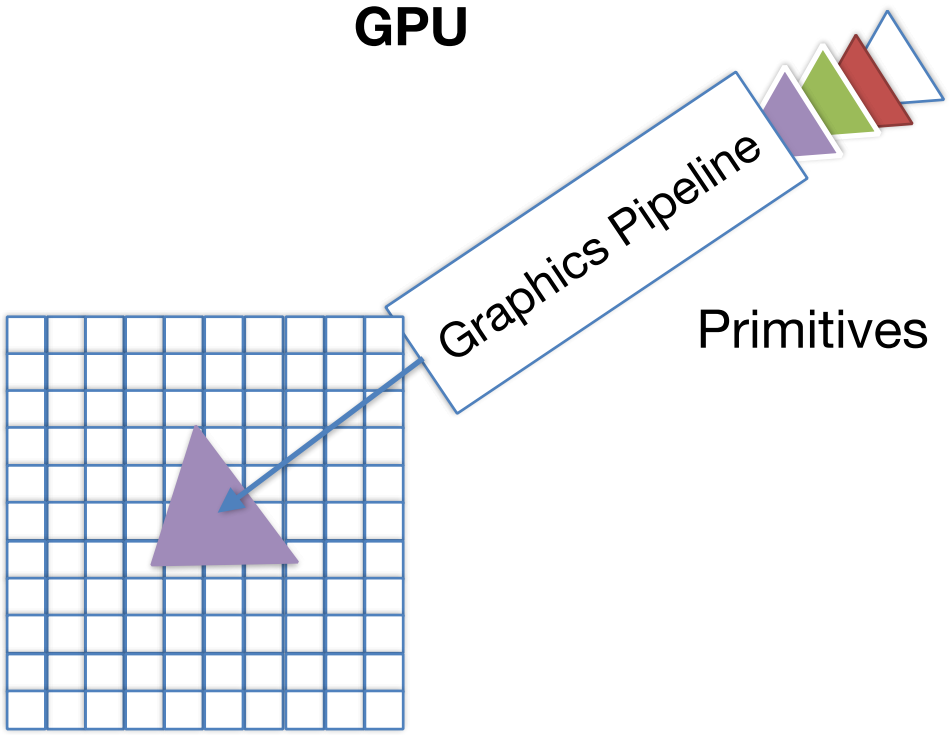
- Points
- Lines
- Triangles

- Implemented in hardware (GPU)



GPUs do Rasterisation

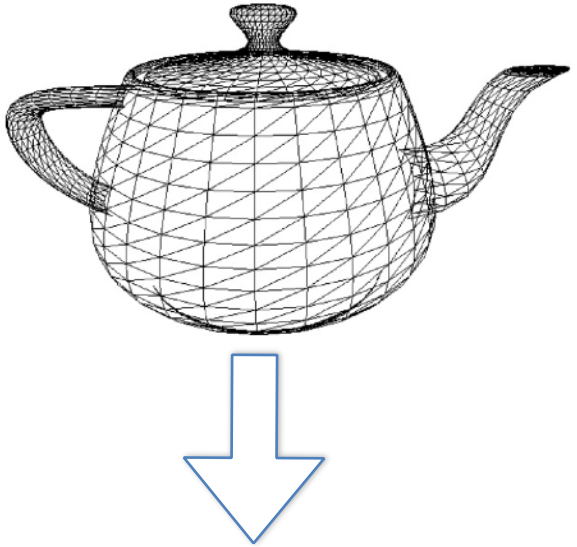
- Rasterisation
  - Compute for each triangle which pixel it covers
  - Triangle centric approach
  - Rasteriser processes one triangle at a time




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

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



Mapping Computer Vision Concepts to Graphics

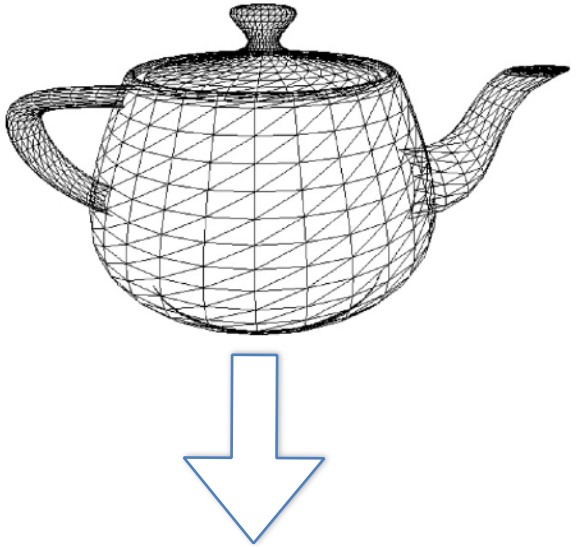
Vision	Graphics
[R|t]	View matrix
K	Projection matrix
Triangulated points	Vertex buffer
Reprojection	Rasterization
Image	Framebuffer

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

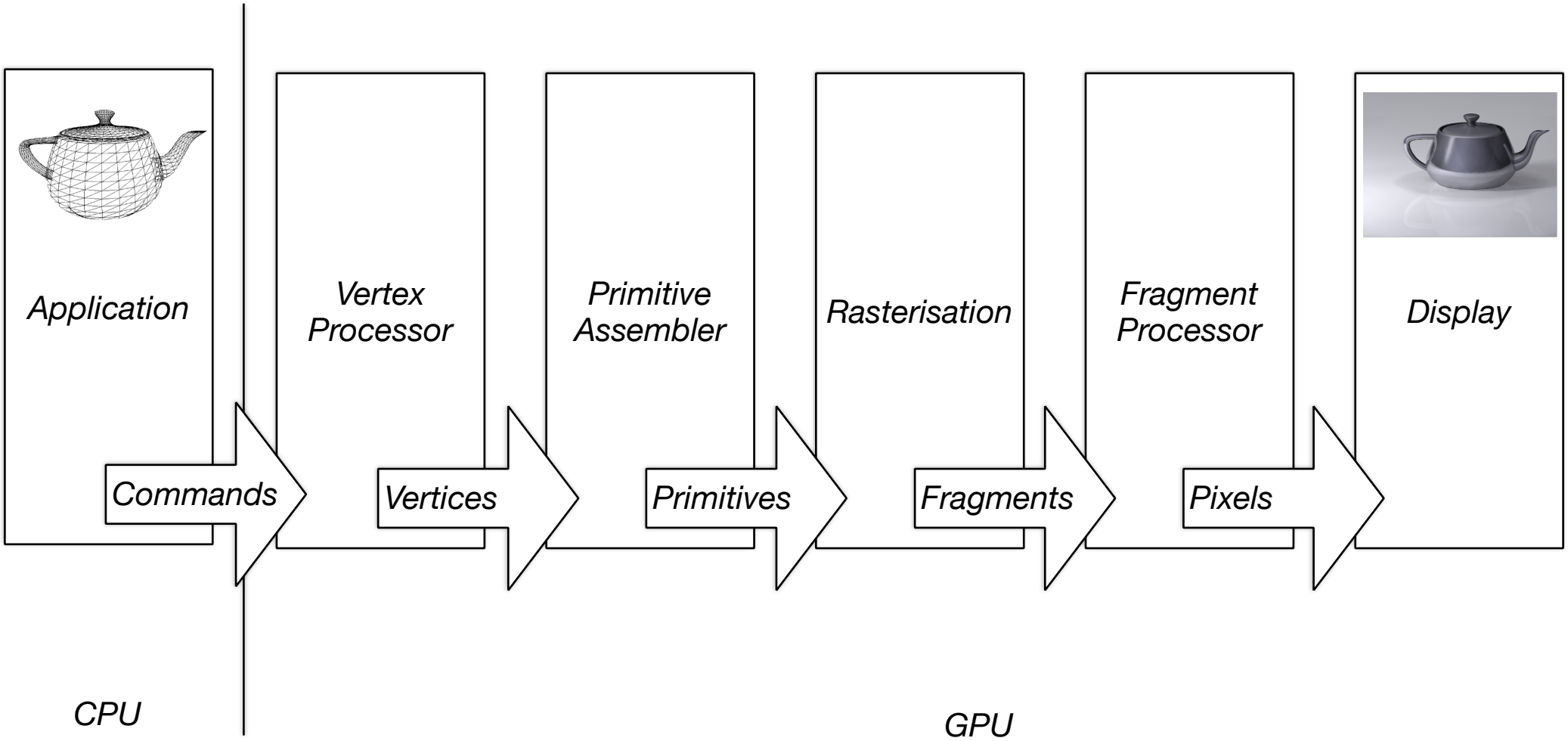
The Common Core: Projective Geometry

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

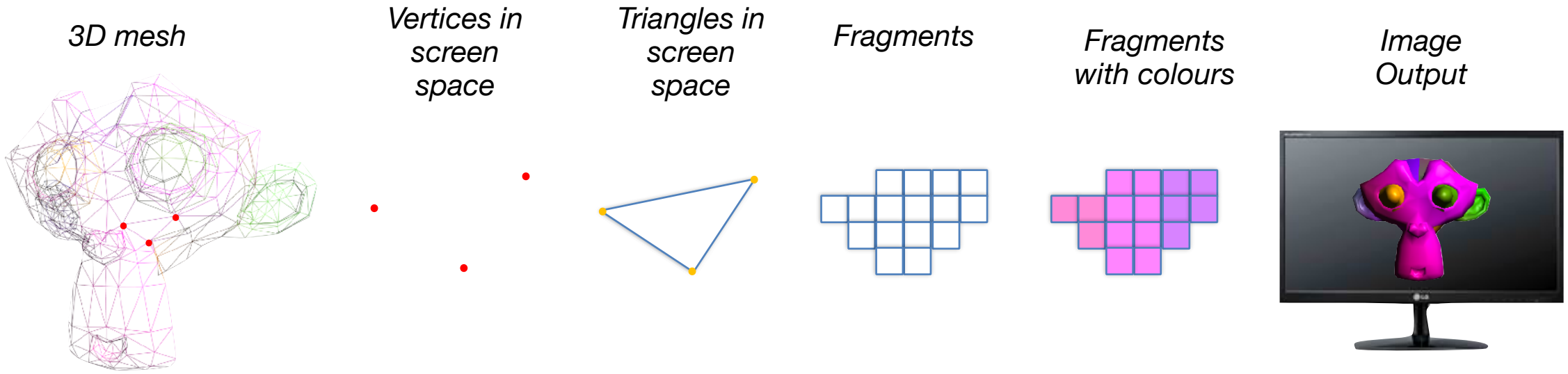
- Note:
  - Same intrinsic and extrinsic parameters
  - Only the direction of computation changes

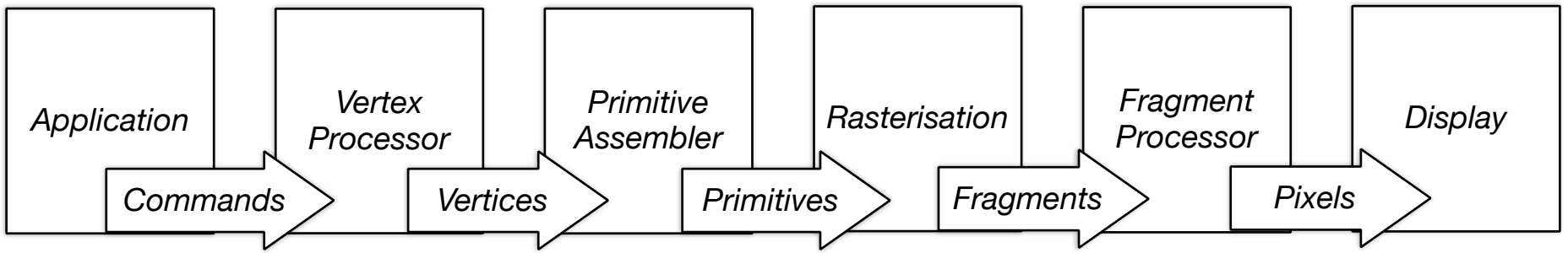


Graphics Pipeline



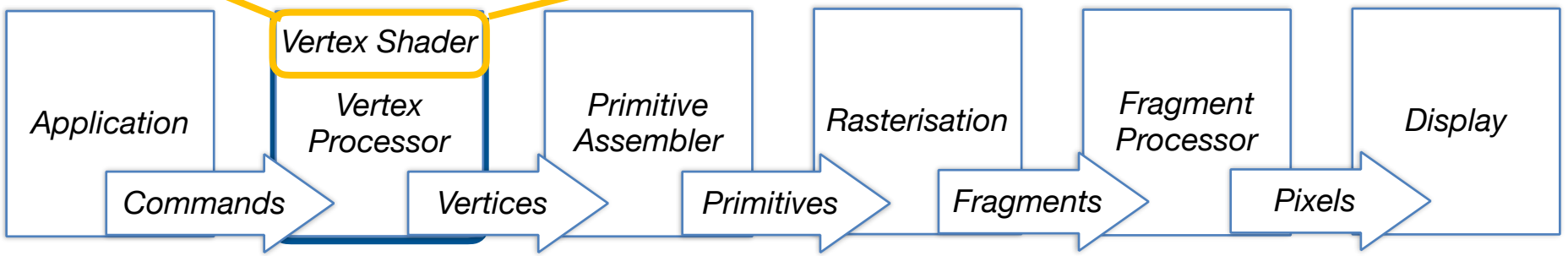
Graphics Pipeline



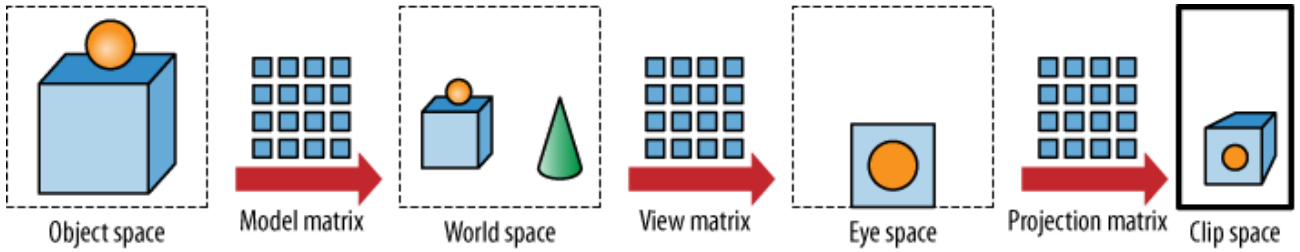


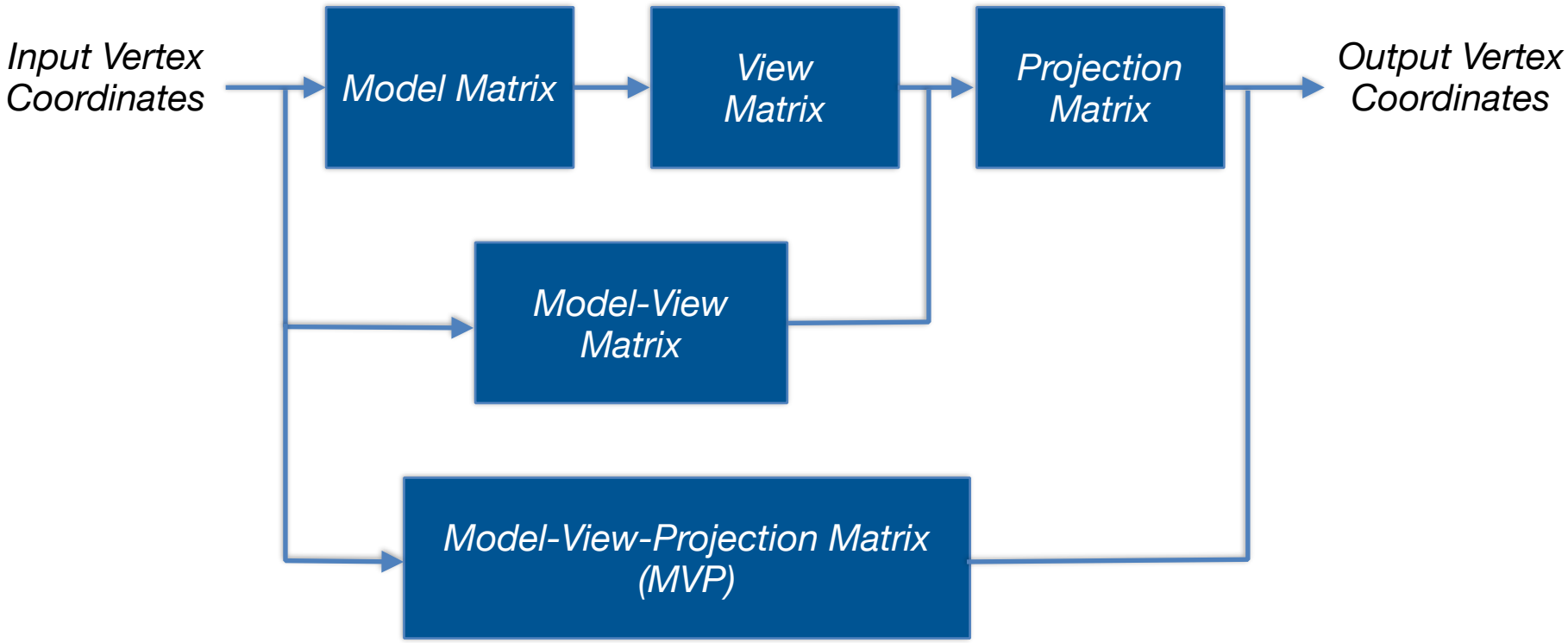
Vertex Processing

- 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)

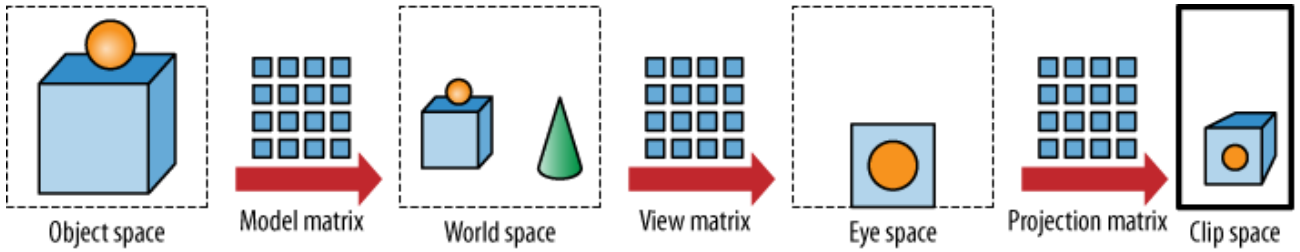


Vertex Shader: Transformations



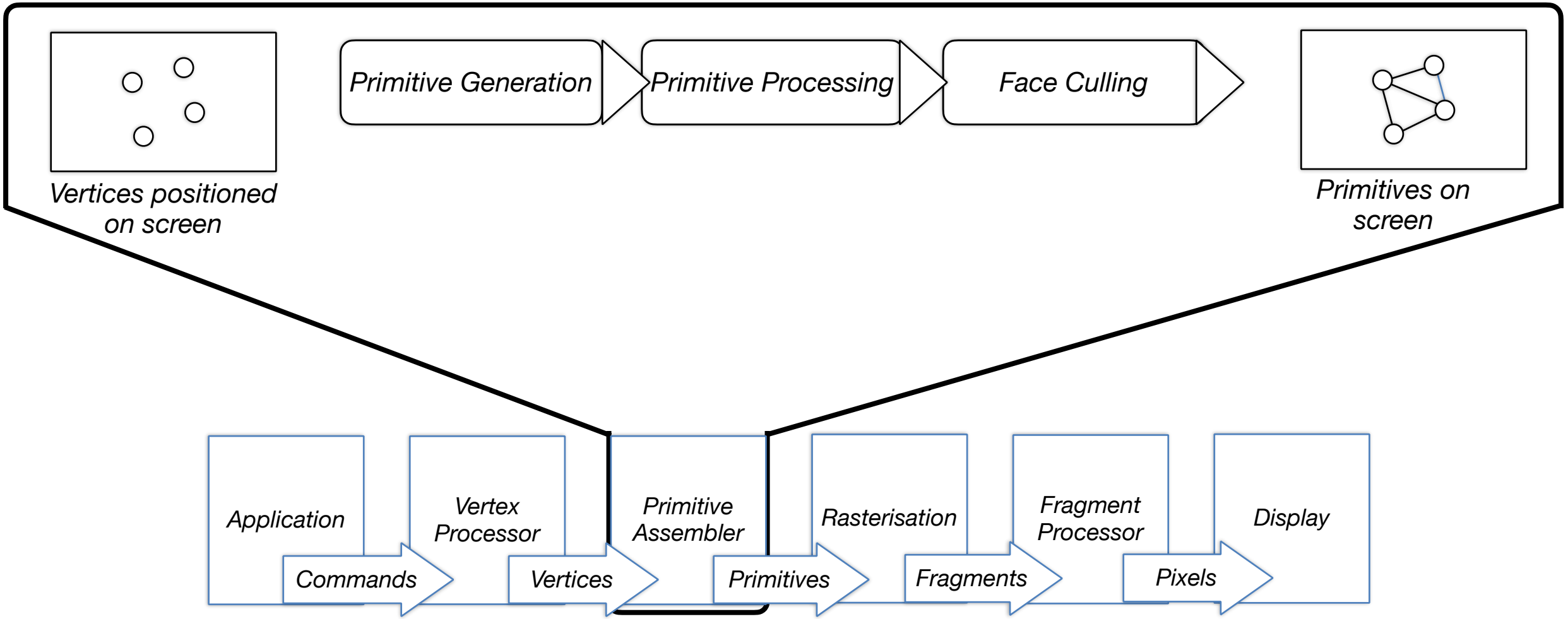


Vertex Shader: Example

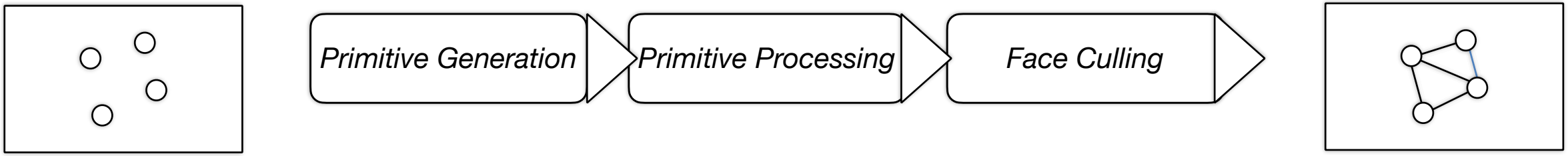



```
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosModelspace;

// 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

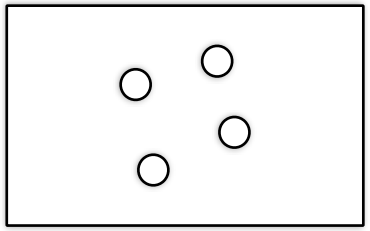


Vertices positioned on screen

Primitives on screen

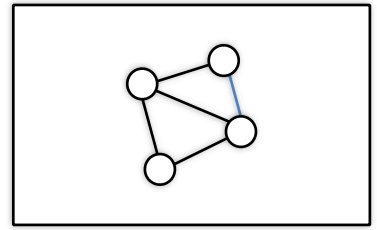
- Primitive assembly receives processed vertices, and the vertex connectivity information as input
- Divides them into a sequence of individual base primitives
- Primitives now have a certain facing
- Face culling discards faces based on their facing

Primitives Assembly: Primitives

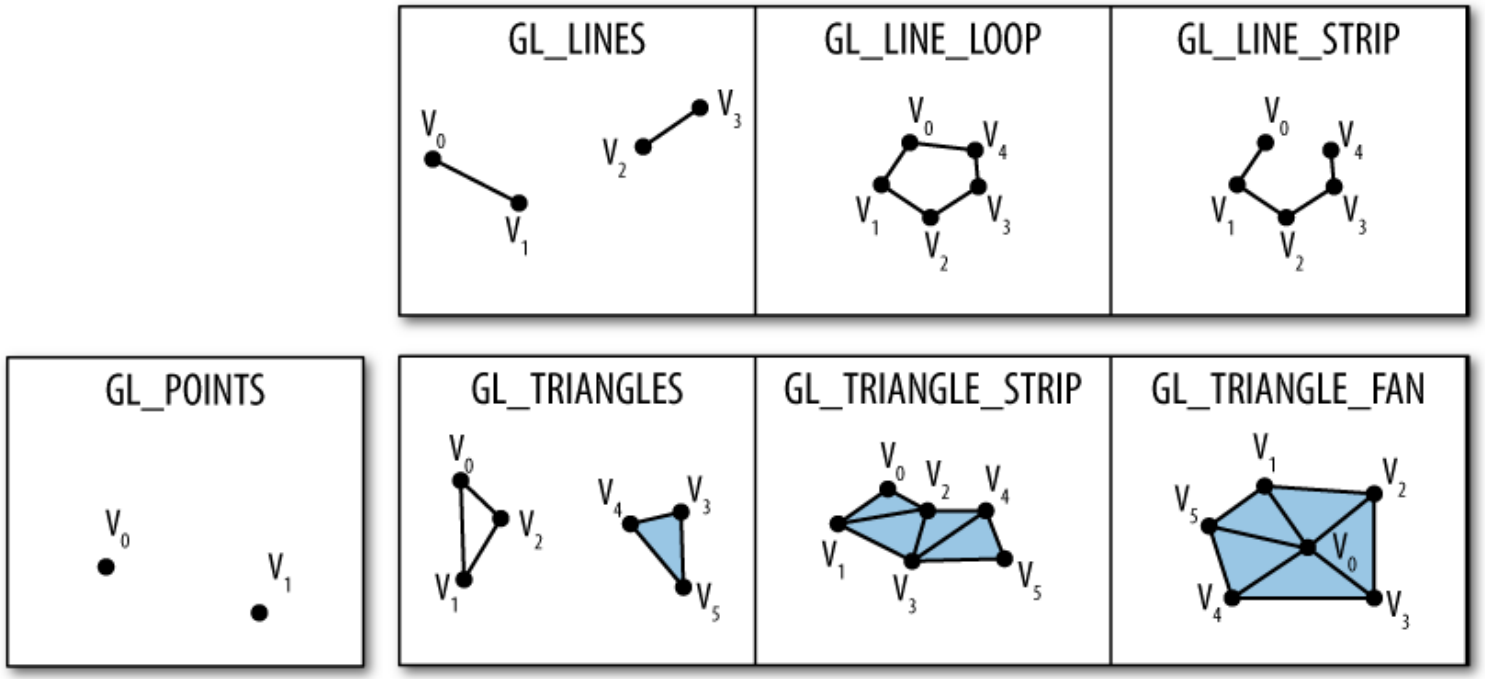


Vertices positioned on screen

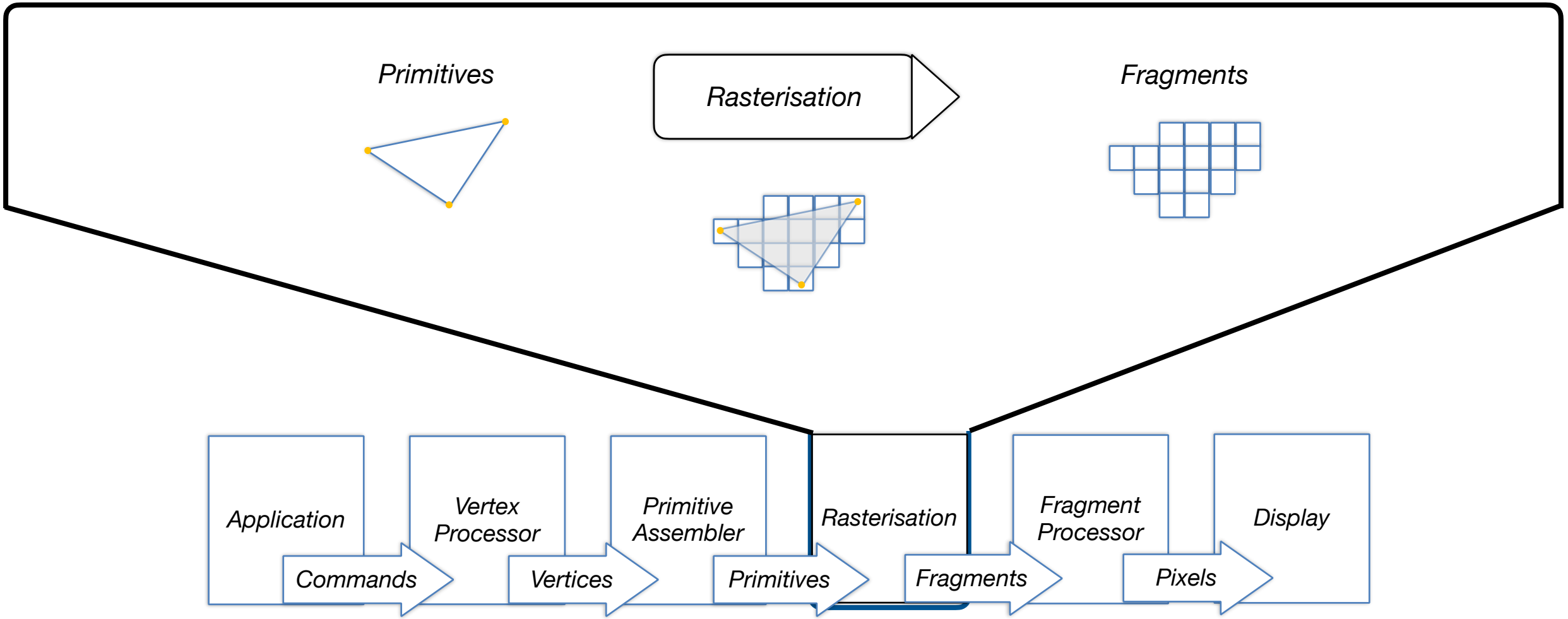
Primitive Generation → *Primitive Processing* → *Face Culling*



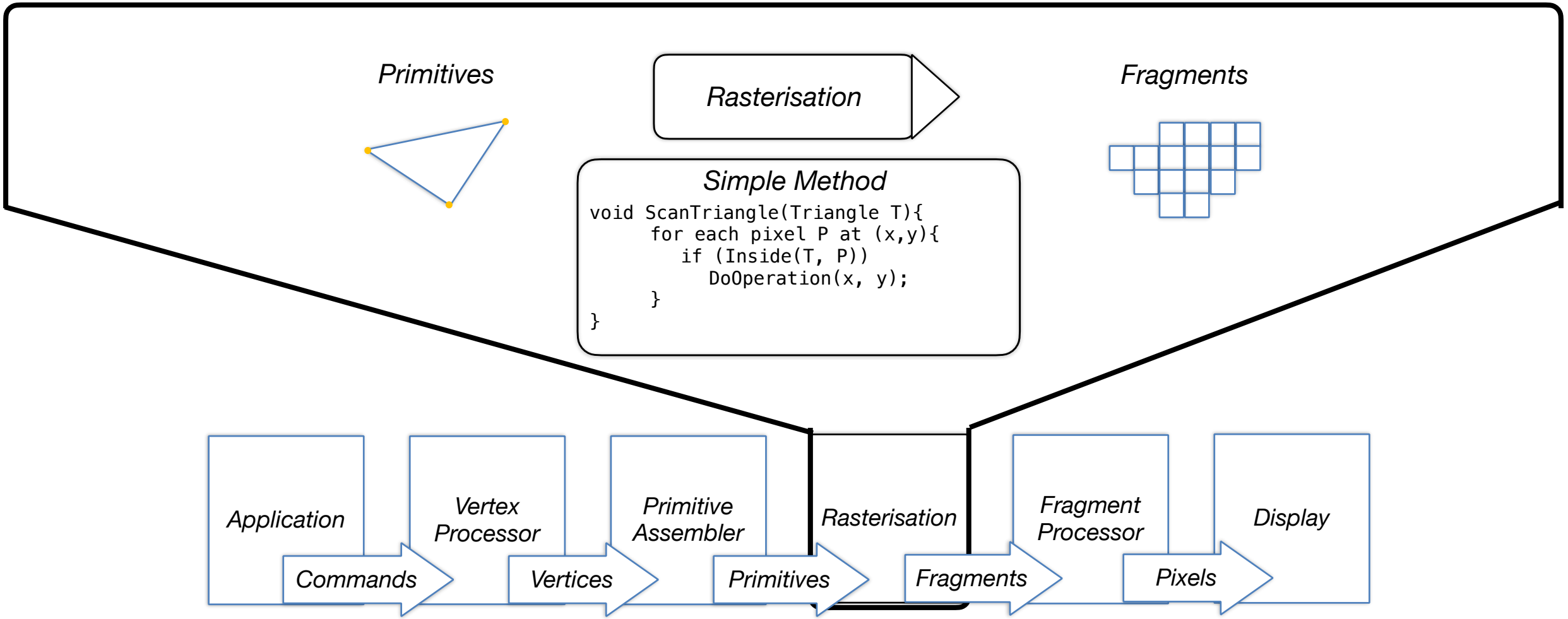
Primitives on screen



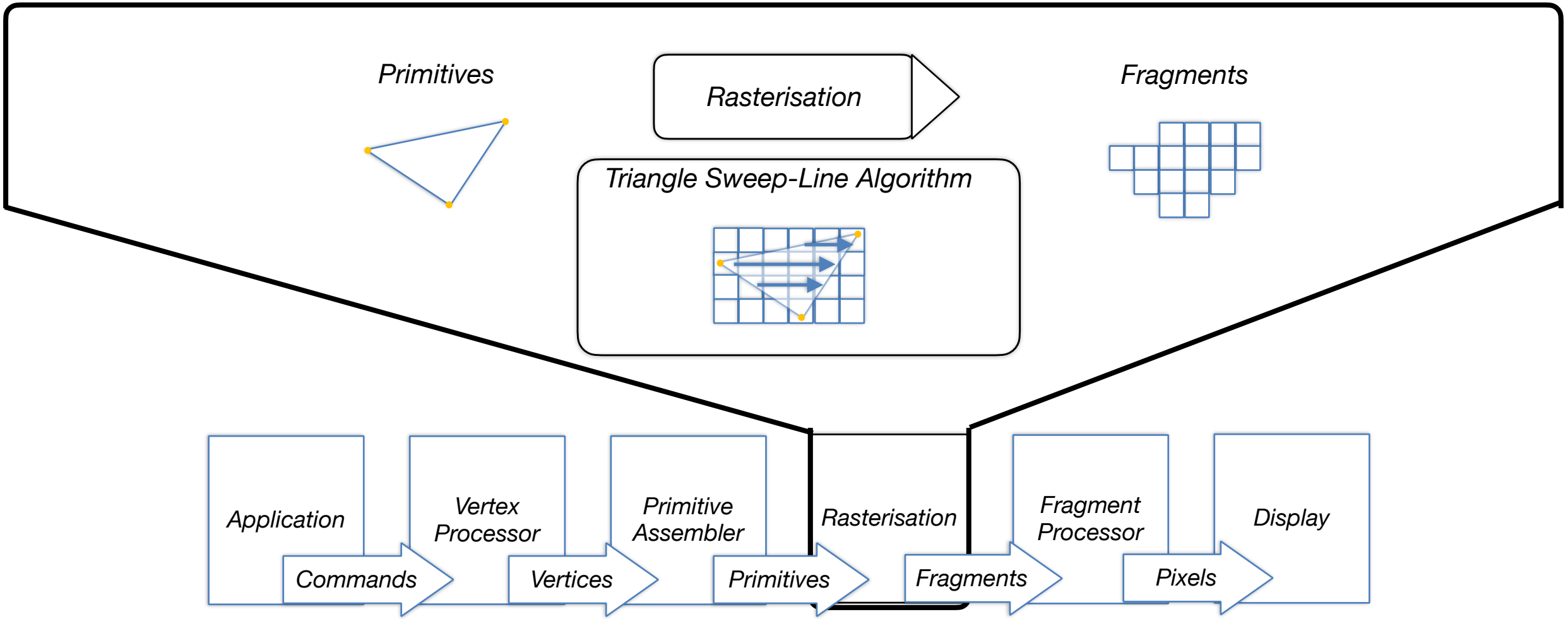
Rasterisation



Rasterisation

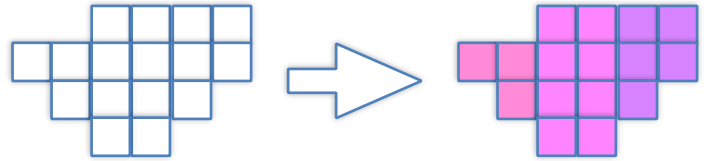


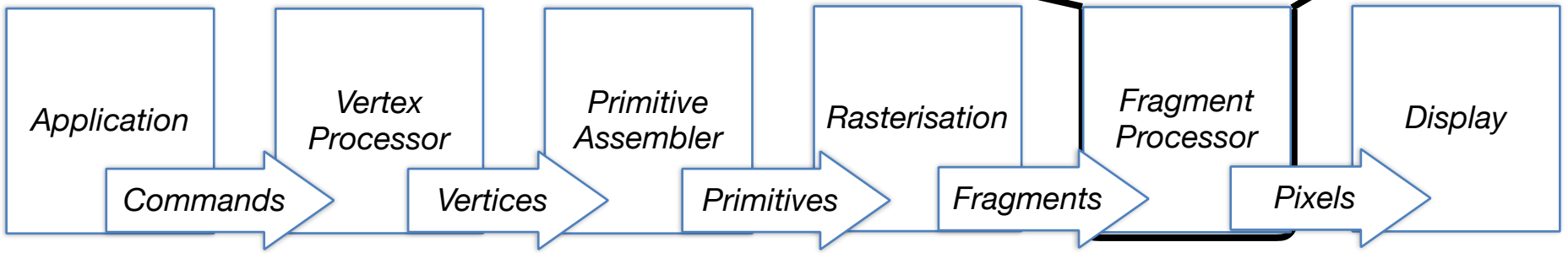
Rasterisation



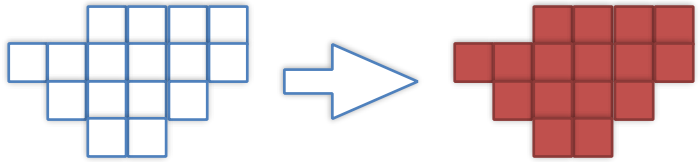
Fragment Processing

- 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;

}
```

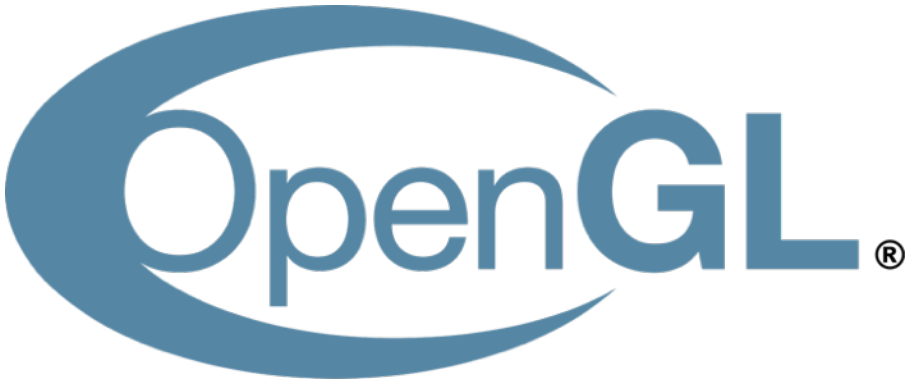
3-Min Discussion:

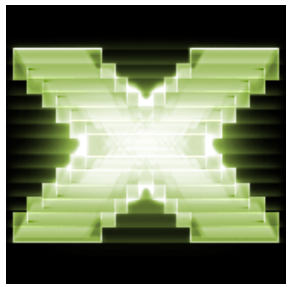
Why does modern graphics use a pipeline with multiple stages (vertex processing, rasterization, fragment shading) instead of doing everything in one big function?

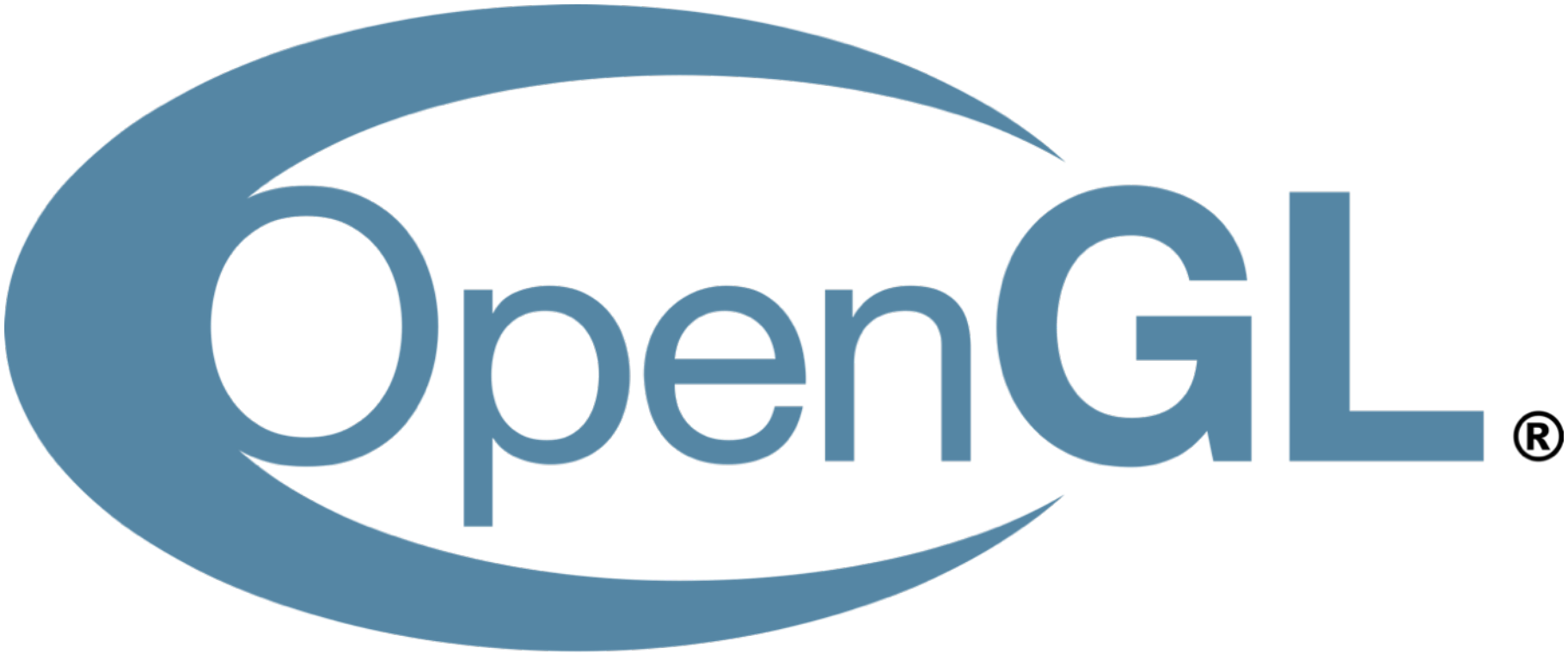
Let’s Get Practical

How to talk to the Graphics Hardware?

Graphics APIs

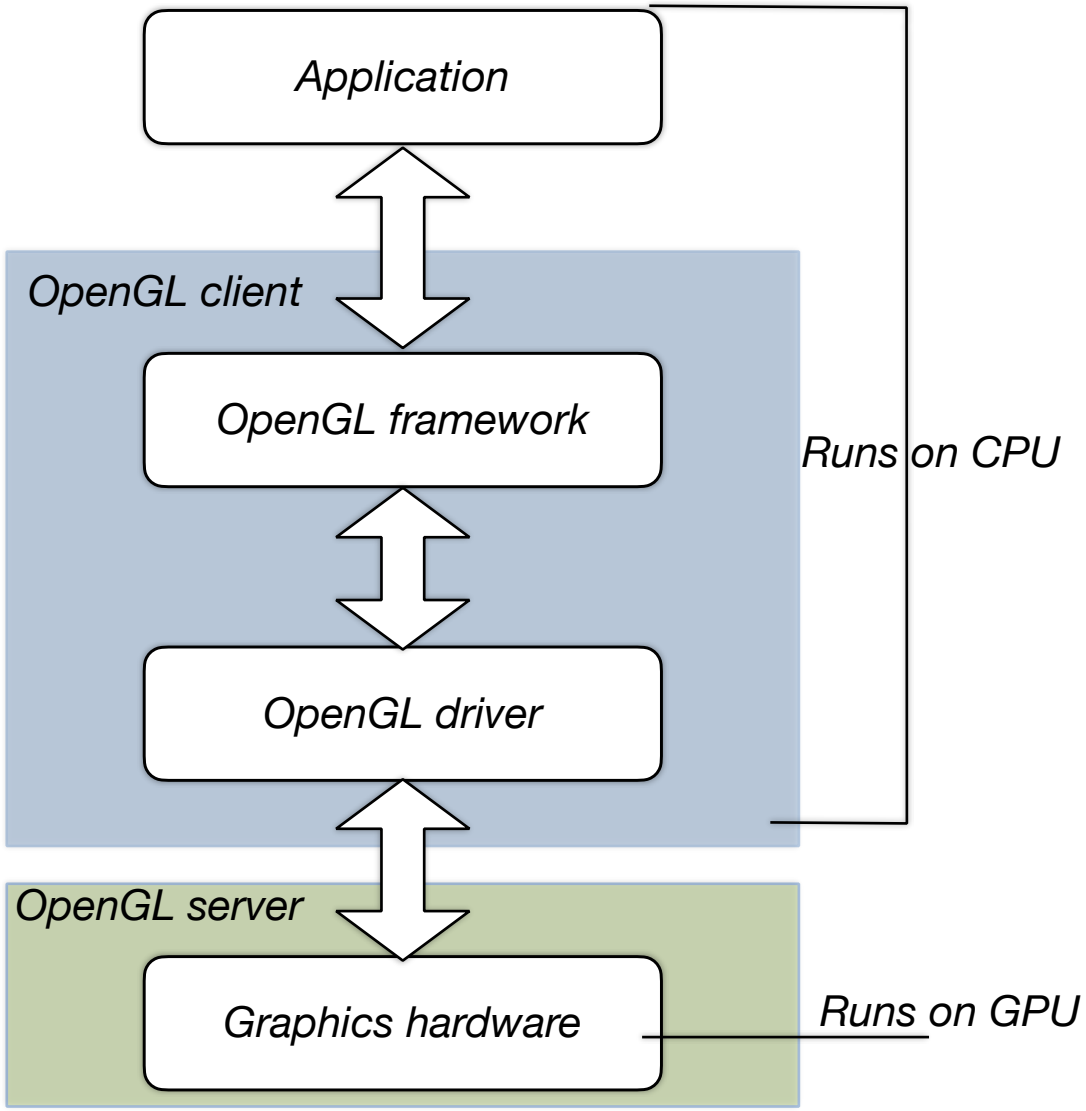





OpenGL®

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



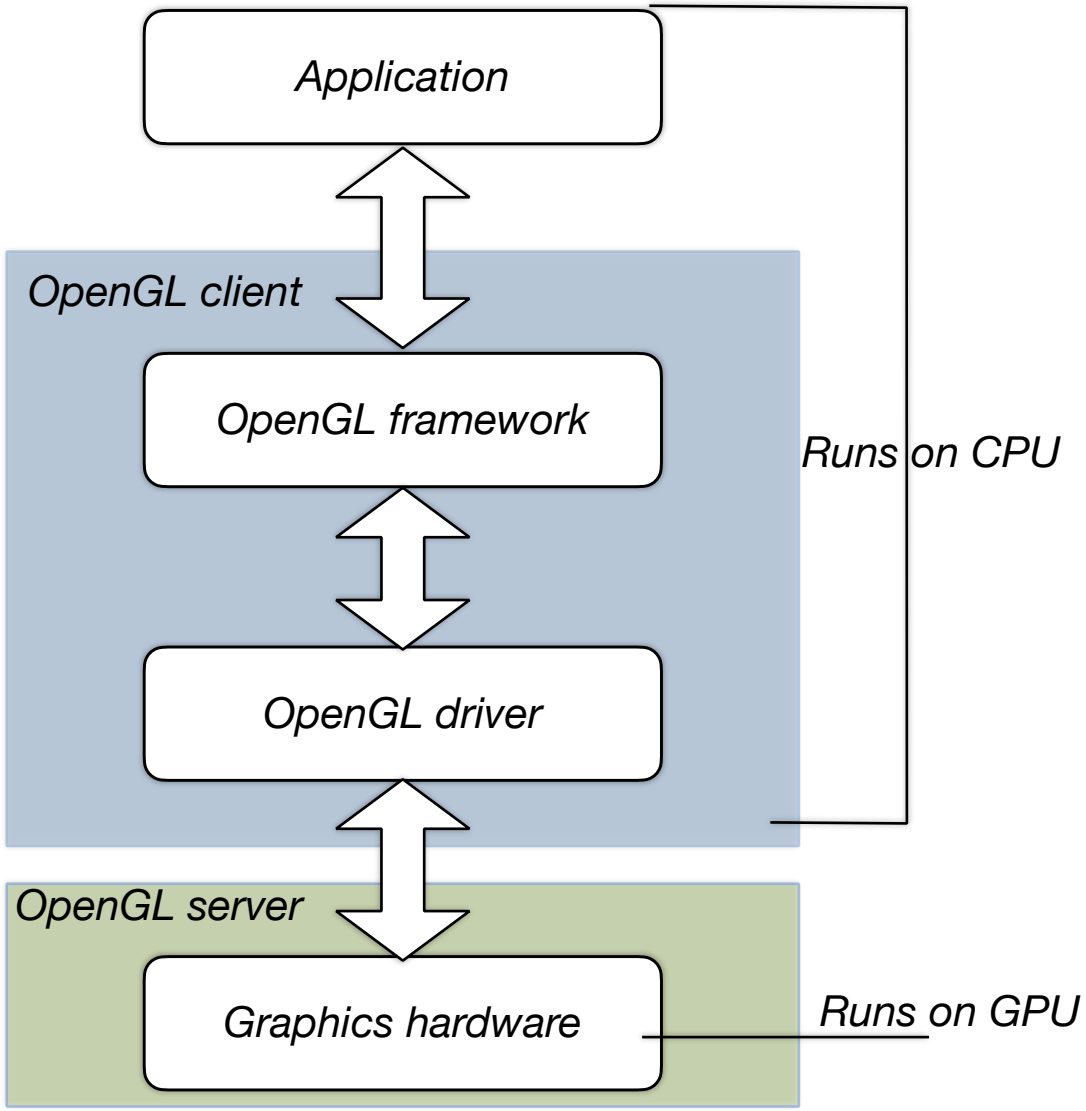
OpenGL

Important to note:

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

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

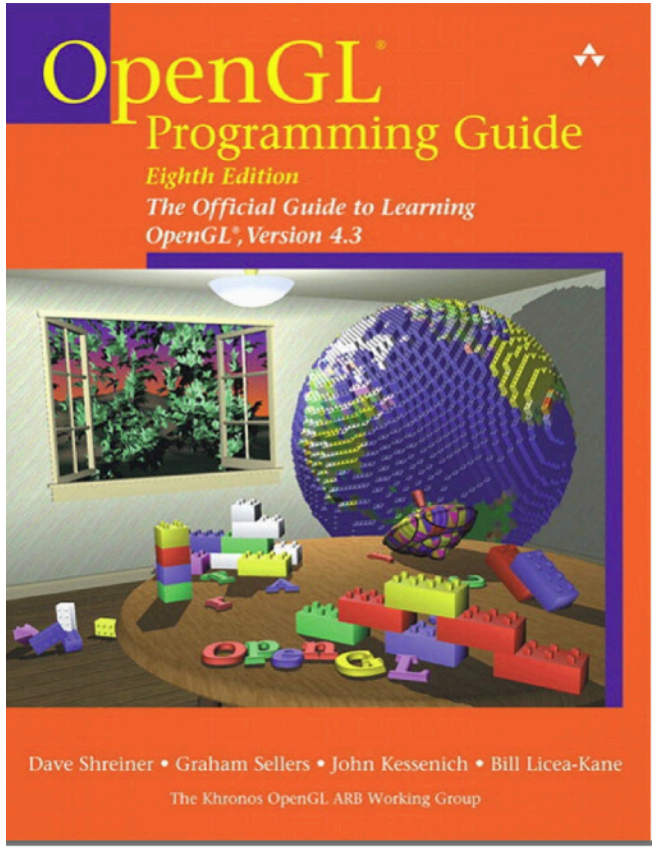
  - Working with identifier: no concept of permanent objects

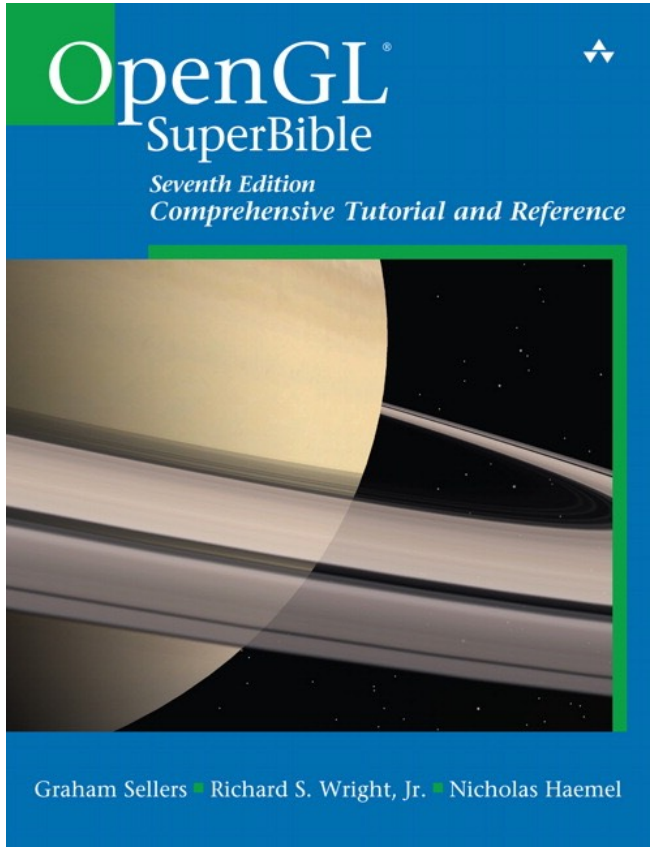


OpenGL - History

- 1992 – Originally released by Silicon Graphics Inc. (SGI) as a platform-independent graphics API for professional 3D applications
- 2006 – Management transferred to the Khronos Group, a non-profit industry consortium that also maintains Vulkan, WebGL, and OpenXR
- 2004 – OpenGL 2.0: Introduced the OpenGL Shading Language (GLSL), allowing programmable vertex and fragment processing
- 2008 – OpenGL 3.0: Major revision; deprecated the fixed-function pipeline and immediate mode (glBegin/glEnd) in favour of the programmable pipeline (using shaders and buffer objects)
- 2017 – OpenGL 4.6: The last version released by Khronos
- Now: OpenGL is stable but no longer actively developed. Khronos and GPU vendors continue to provide driver support, but new features and performance improvements are being developed under Vulkan, which is intended as its successor

OpenGL - Ressources





https://learnopengl.com/

OpenGL Concepts

- OpenGL Context
- OpenGL State
- OpenGL Object Model

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

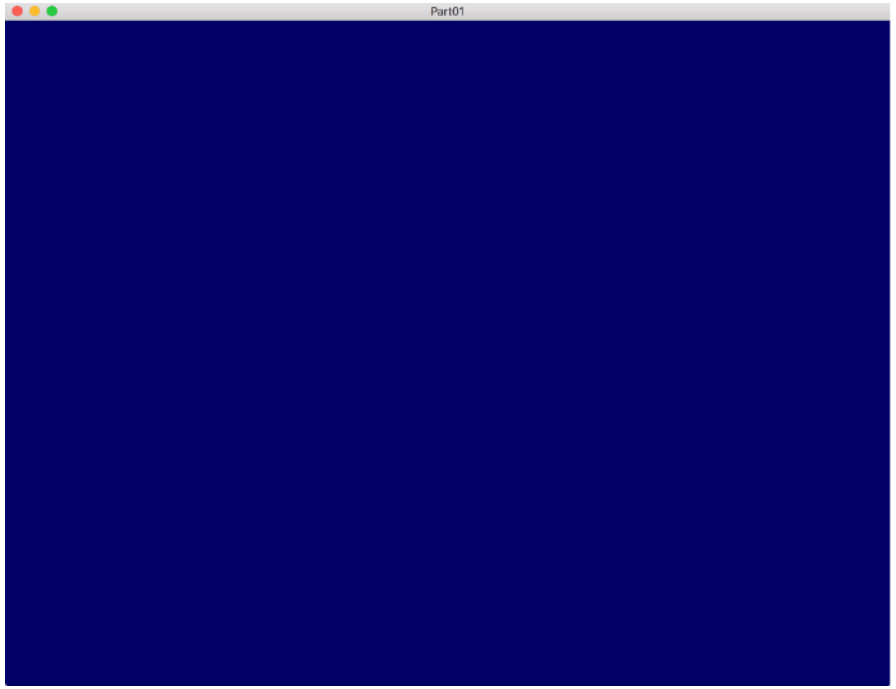
OpenGL Context

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?

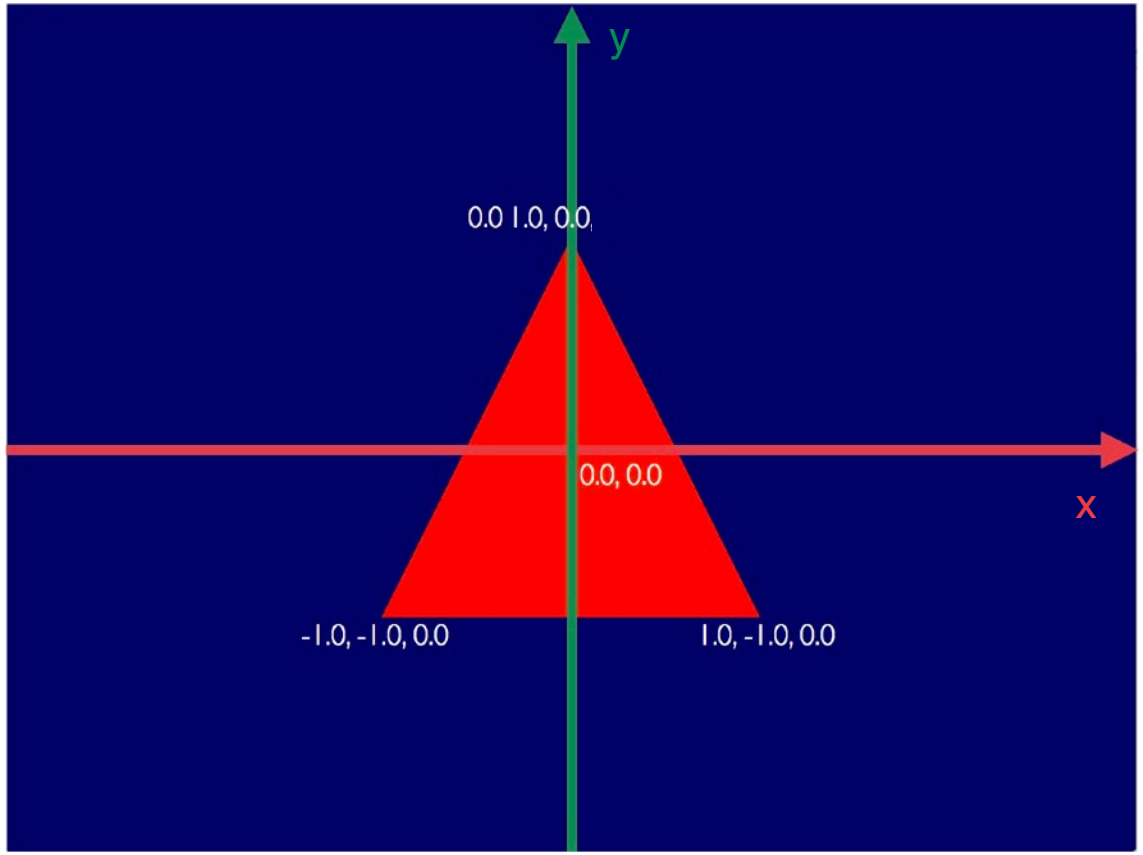
- target⇔type
- commands⇔methods

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,
    };
```

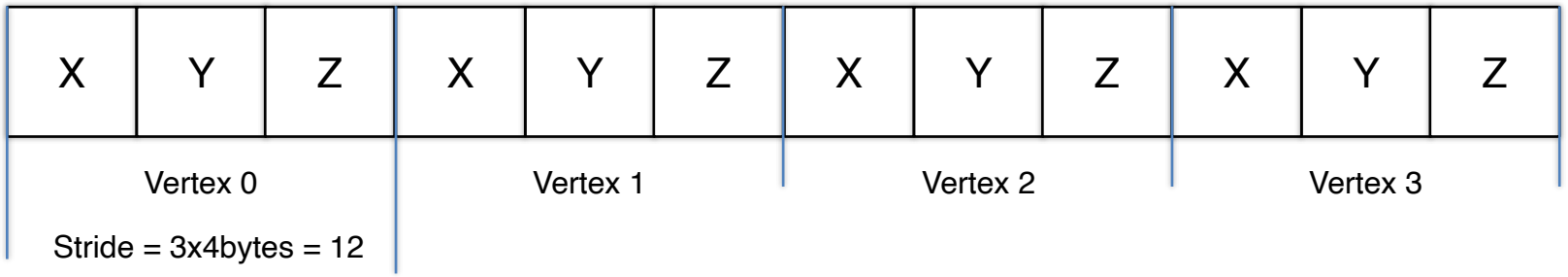
Example: Vertex Buffer Object

```
//  - - - - - - - - - - - - - 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
```

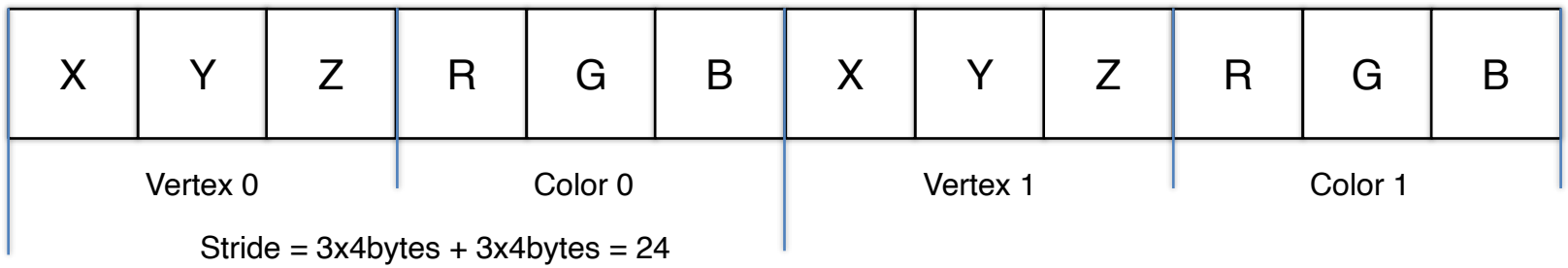
Stride

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

  - Tightly packed:



  - Interleaved:

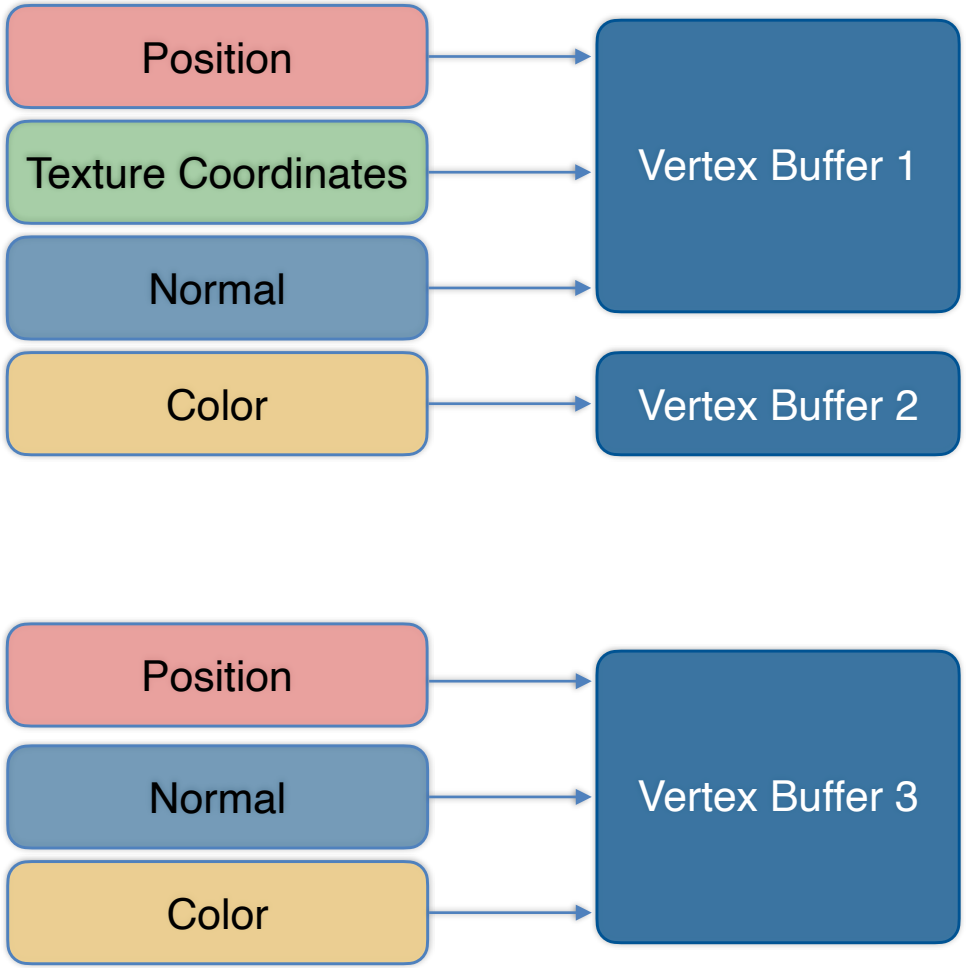


Example: Interleaved Vertex Buffer Object

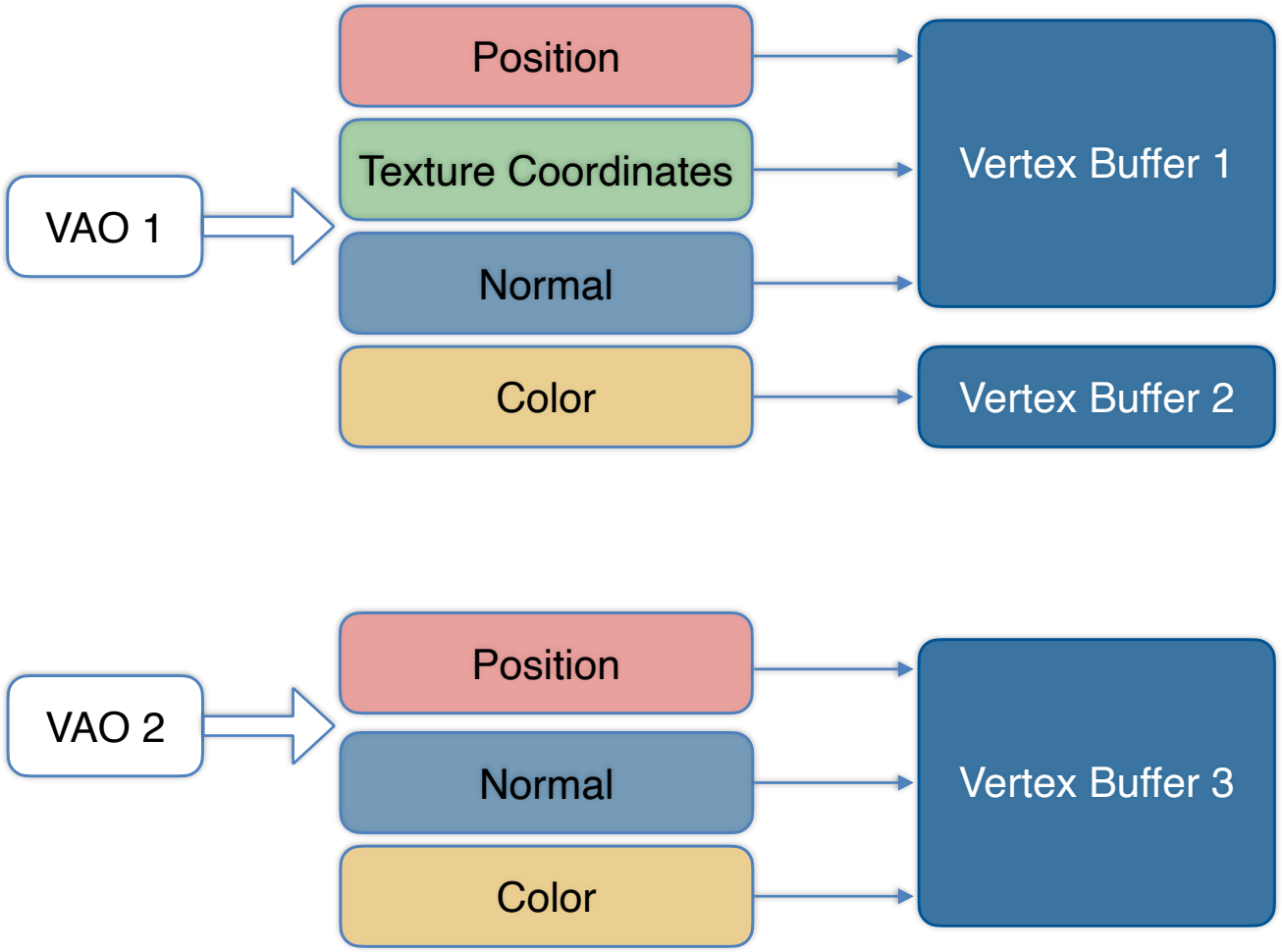
```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    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?
                          24,                  // interleaved data
                          (void*)0            // array buffer offset is 0 for vertices
                          );
//  - - - - - - - - - - - - - Actual drawing call - - - - - - - -
  glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

VERTEX ARRAY OBJECT

- If an application renders a lot of different objects, we would need to reconfigure the buffers
- To avoid this -> Use vertex array objects
- Vertex Array Object:
- Holds the state that configures the vertex specification stage:
  - Stores the data format of vertices
  - Buffer object bindings
  - Vertex attribute mapping



VERTEX ARRAY OBJECT



Example: Vertex ARRAY Object

```
//create a Vertex Array Object and set it as the current one
 GLuint VertexArrayID;
 glGenVertexArrays(1, &VertexArrayID);
 glBindVertexArray(VertexArrayID);
```

Important to note:

- We need at least on vertex array object in our application

VERTEX ARRAY OBJECT Example

```
GLuint vao2;
glGenVertexArrays(1, &vao2);
glBindVertexArray(vao2);

// Create and bind buffer object for vertex data
GLuint vbuffer;
glGenBuffers(1, &vbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vbuffer);

// copy data into the vertex buffer object
glBufferData(GL_ARRAY_BUFFER, NUM_VERTS * sizeof(Vertex), vertexdata, GL_STATIC_DRAW);


// set up vertex attributes
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, position));
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, normal));
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, colour));

glBindVertexArray(vao2);
```

Draw Call

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

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

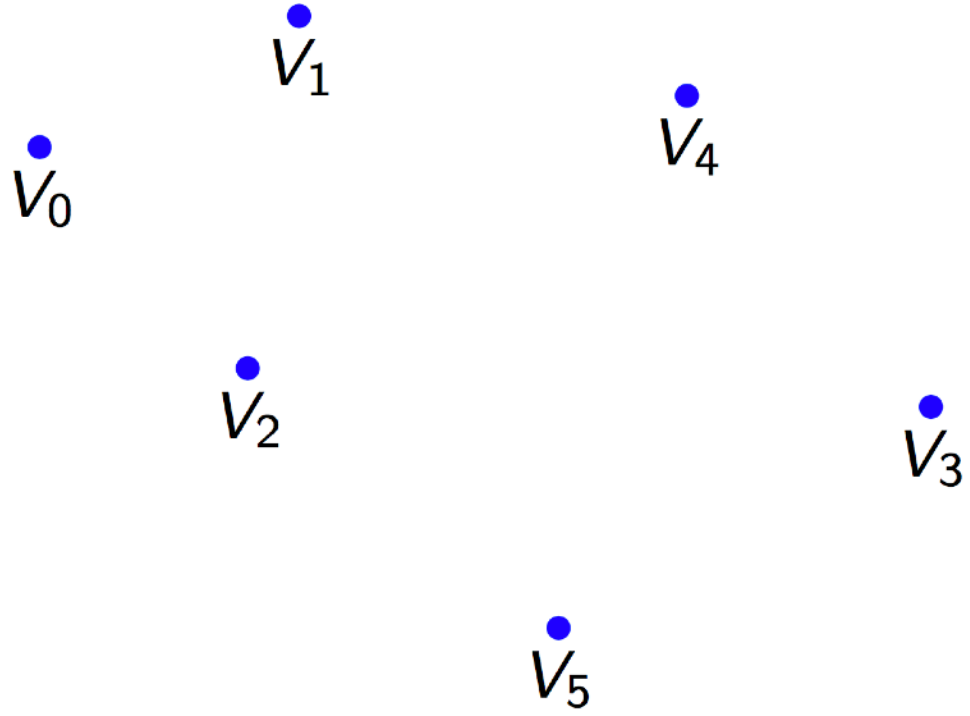
mode specifies what kind of primitives to render. e.g. GL_TRIANGLES

specifies the starting index in the enabled arrays.

count specifies the number of indices to be rendered.

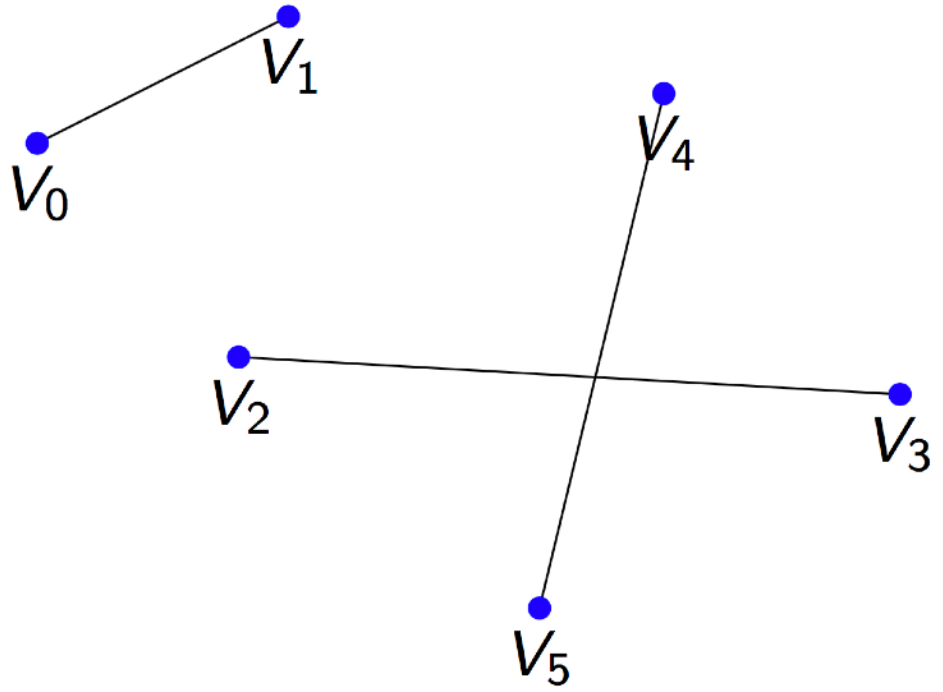
Primitive Types

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



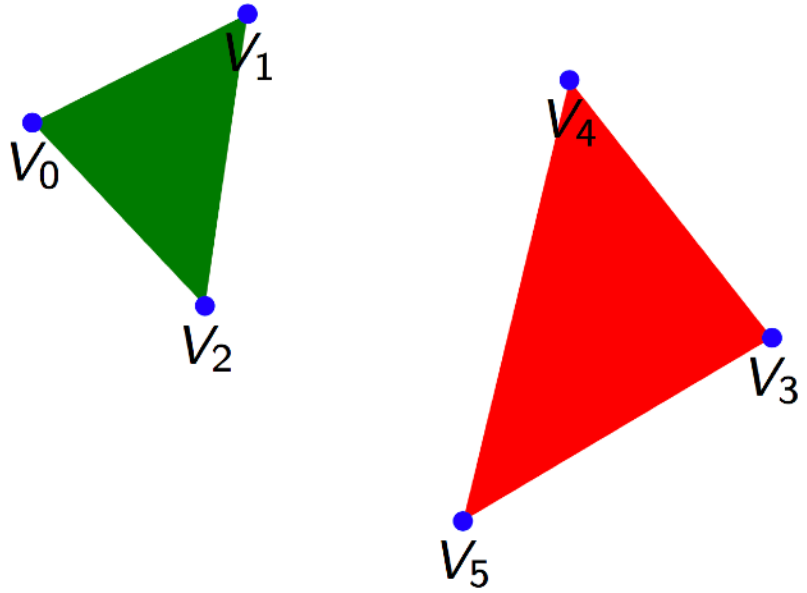
Primitive Types

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



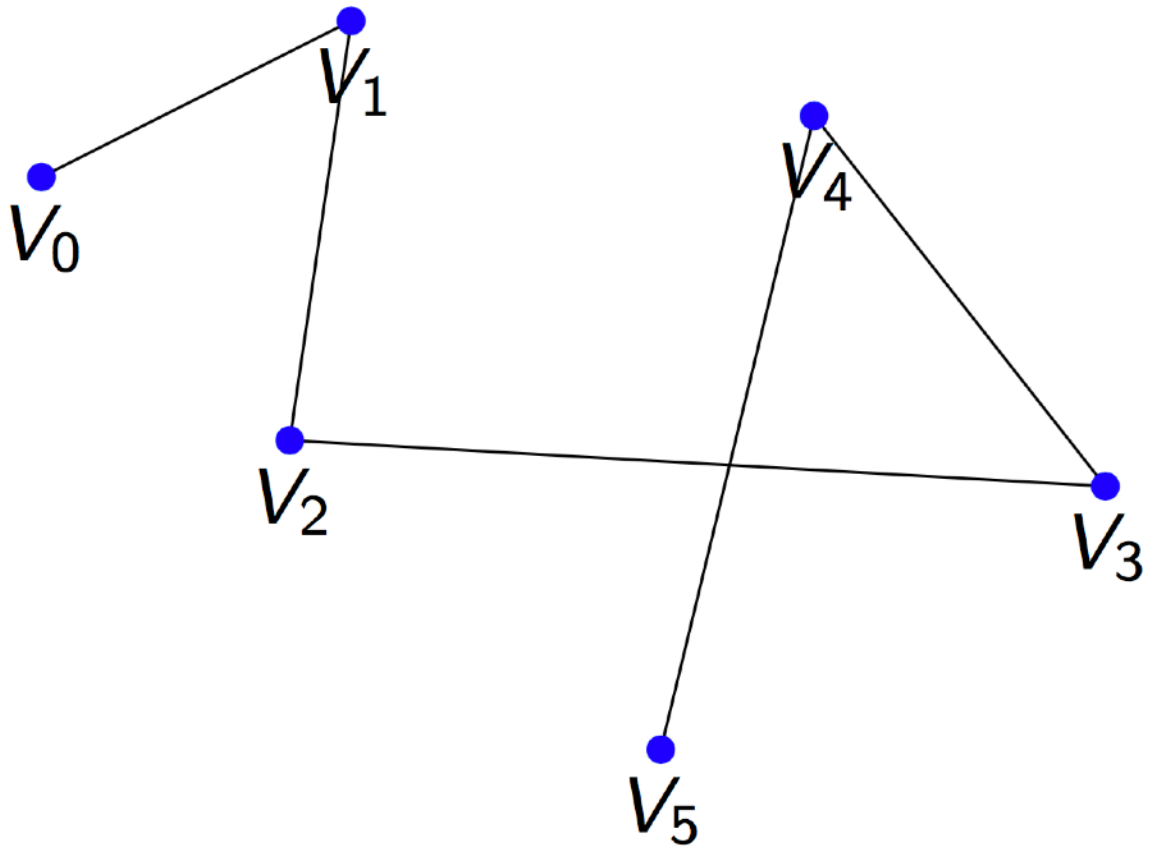
Primitive Types

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



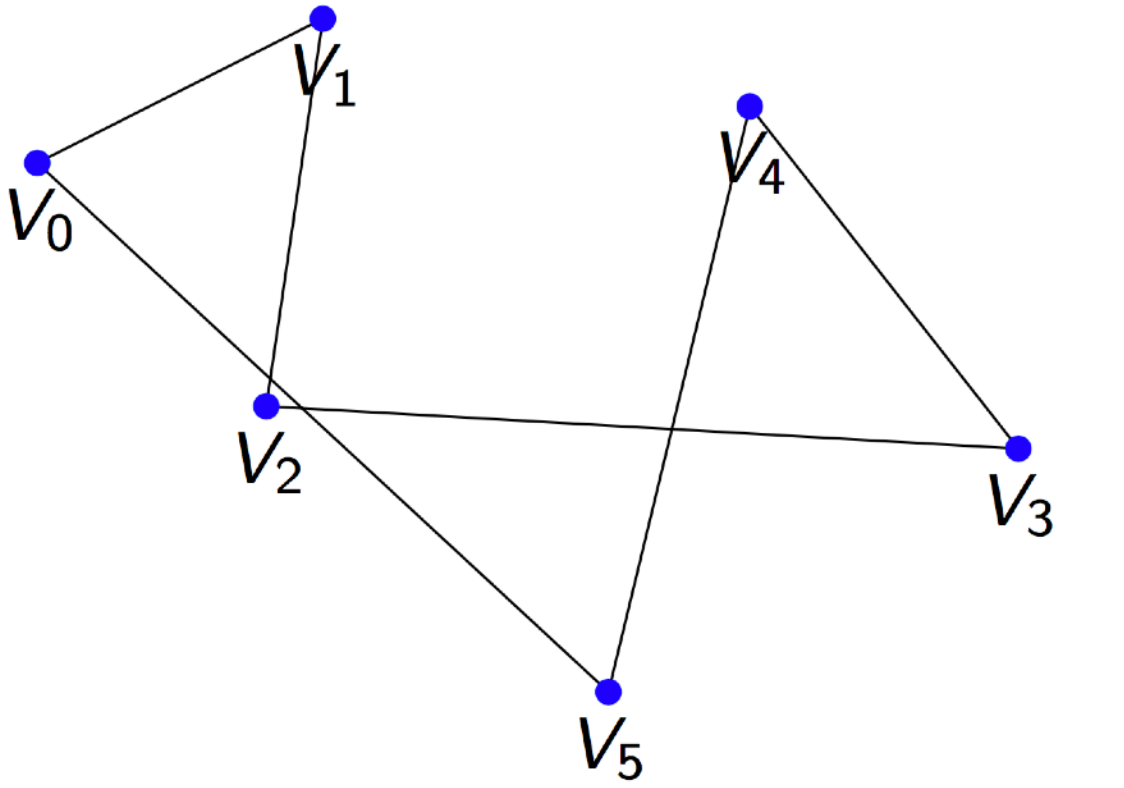
Primitive Types

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



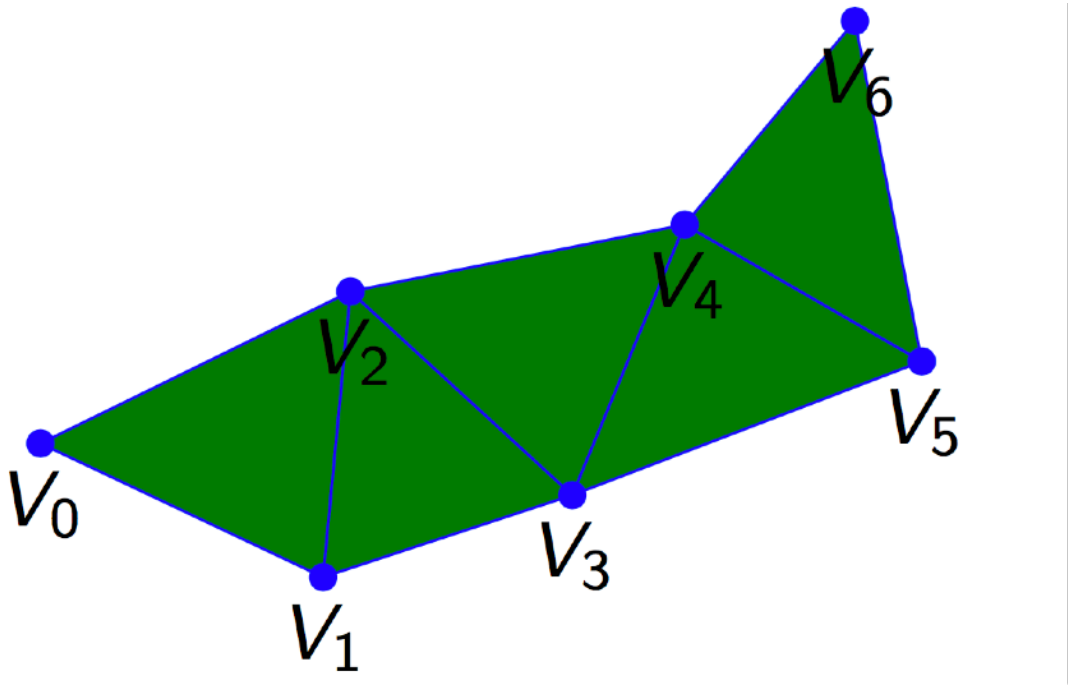
Primitive Types

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



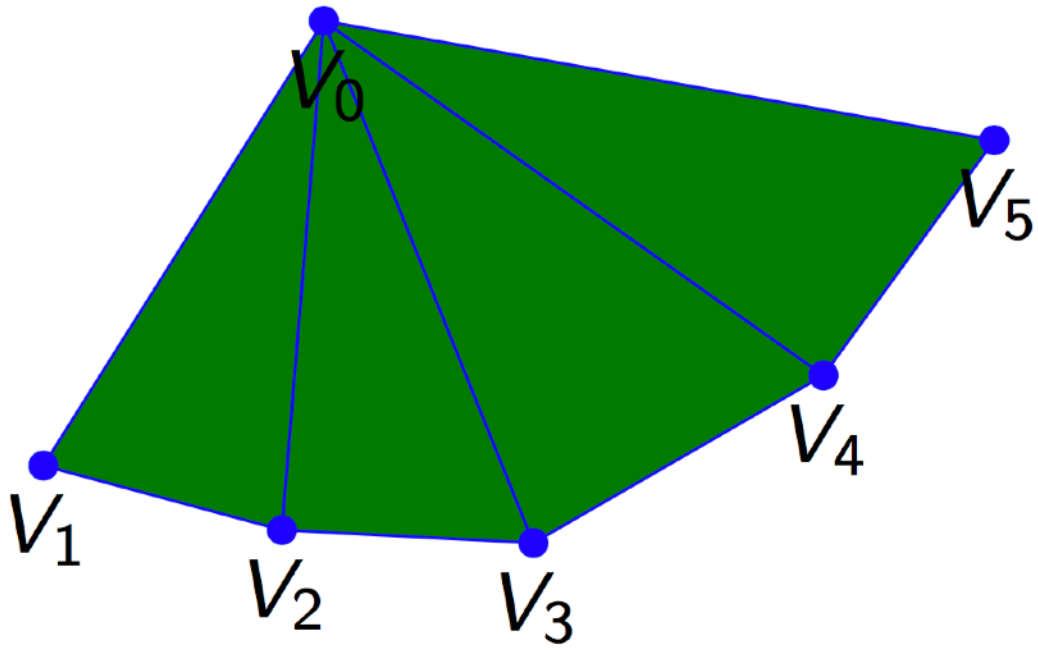
Primitive Types

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



Primitive Types

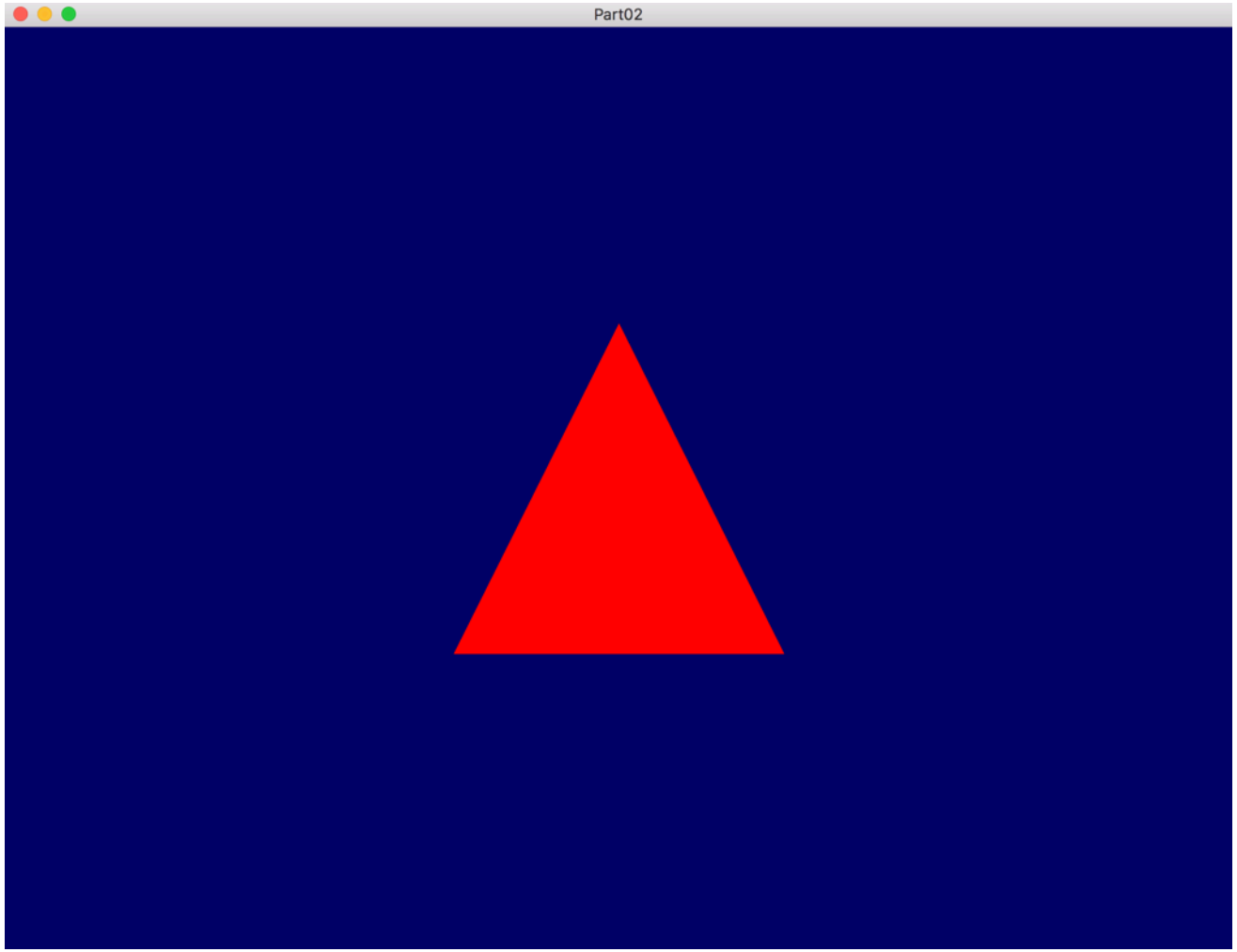
- Points
- Lines
- Triangles
- Line strips
- Lineloops
- Triangle strips
- Triangle fans



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

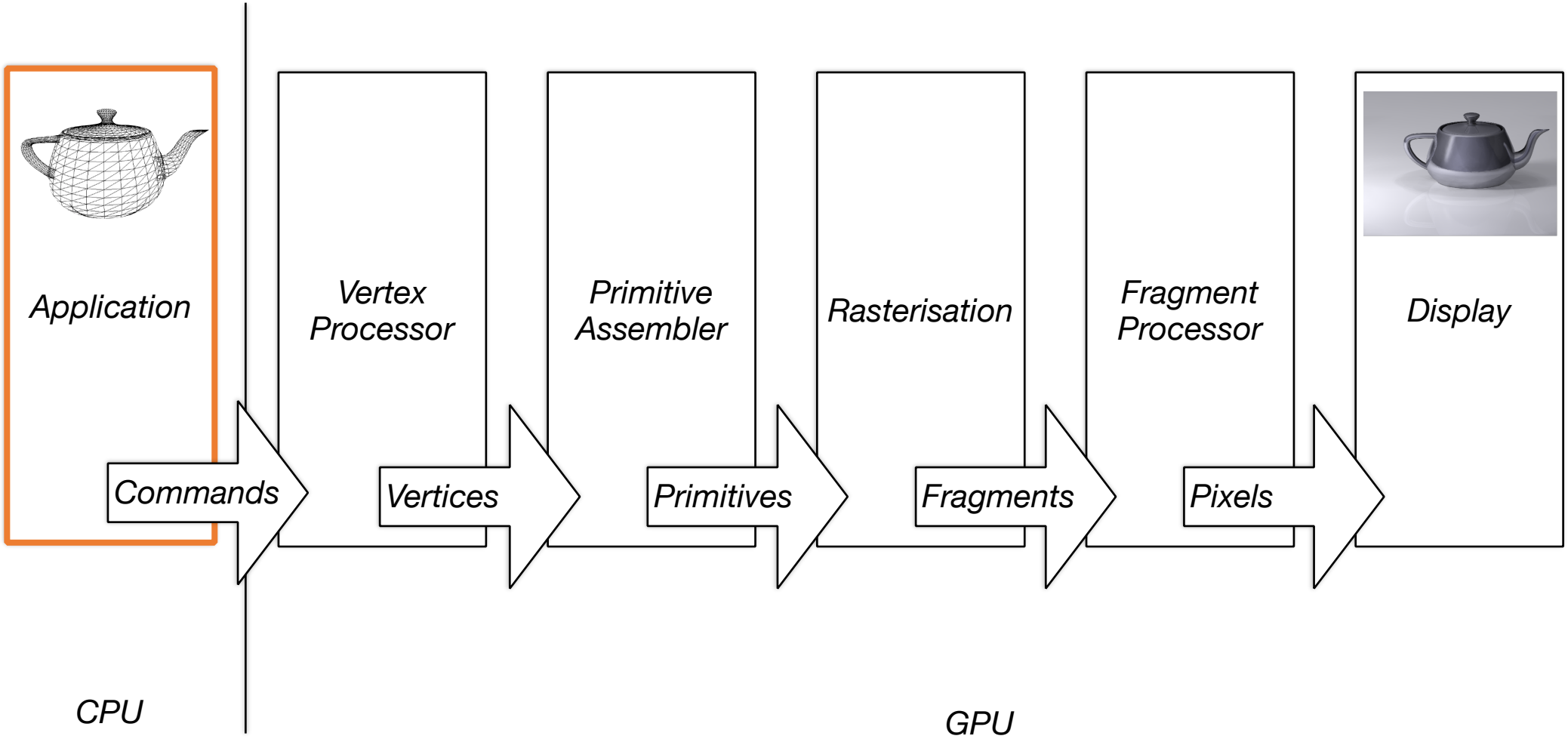
```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);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

3-Min Discussion:

So Far - Where are we in the graphics pipeline?

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 shading of a surface

Shader

Shading languages:

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

The end!