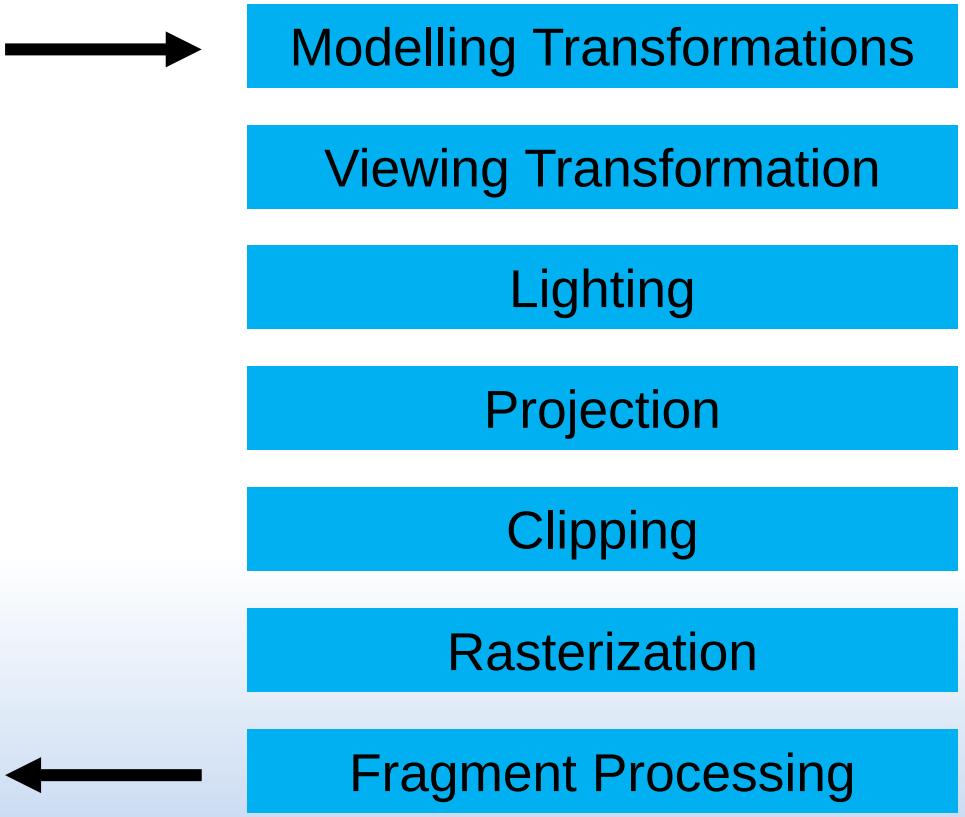
# Graphics Fundamentals

# The Computer Graphics Pipeline

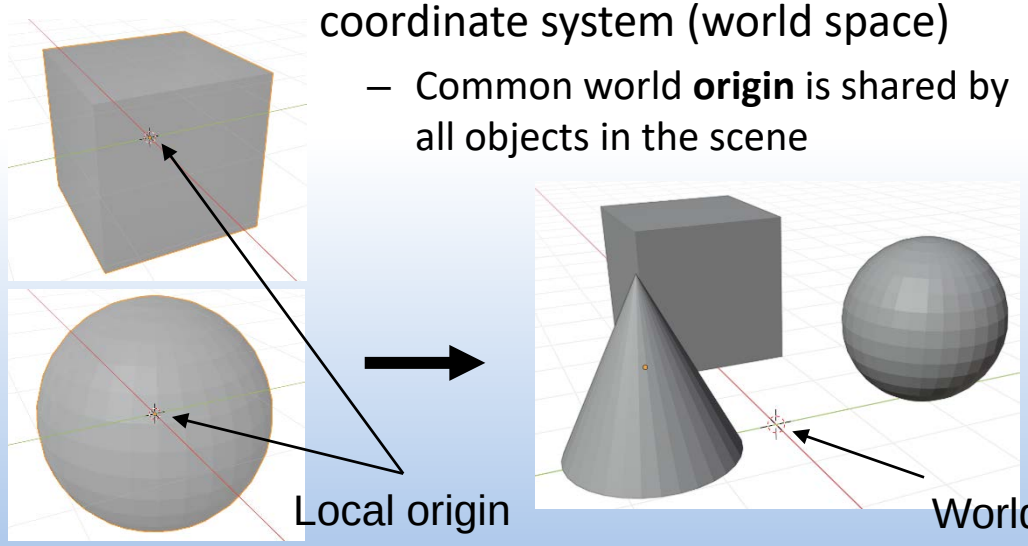
- Overview of stages in the graphics pipeline
  - ➢ Input
    - Vertex data
      - Geometric models

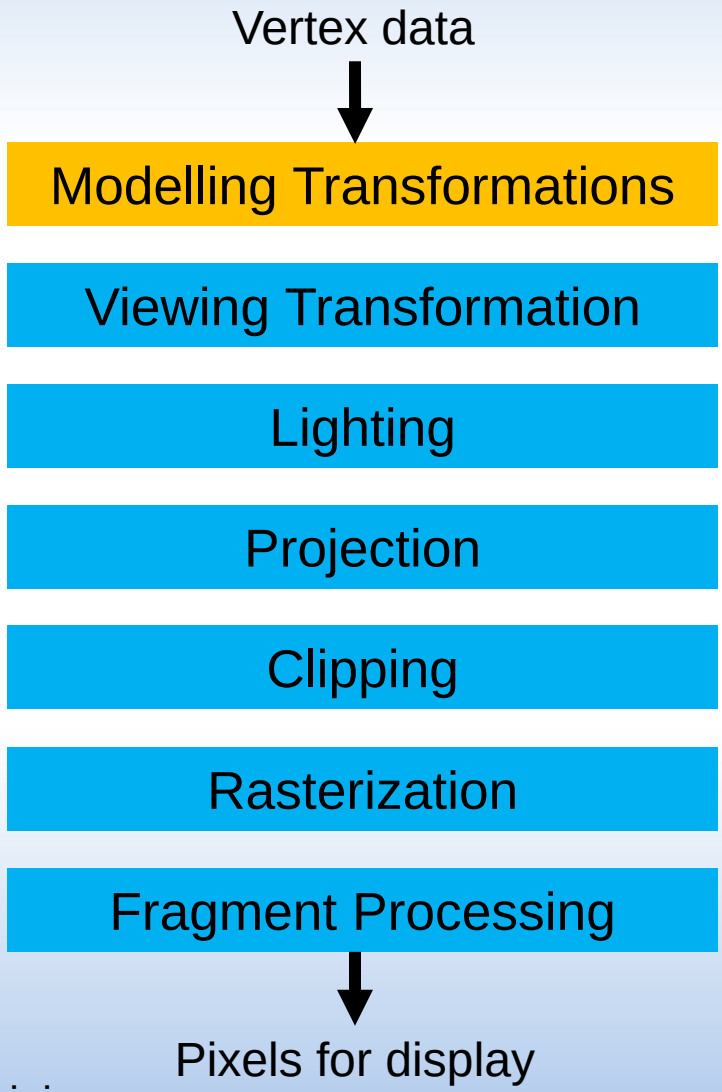



  - ➢ Output
    - Pixels for display

# The Computer Graphics Pipeline

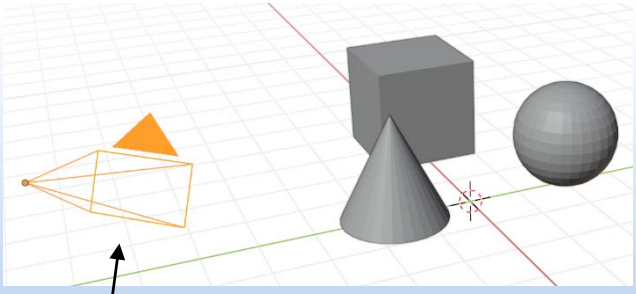
- Modelling transformations
  - **Model space** (or local space) to **world space**
    - Models typically defined in their own coordinate system
      - Each have their own local **origin**
    - Places objects in a common coordinate system (world space)
      - Common world **origin** is shared by all objects in the scene



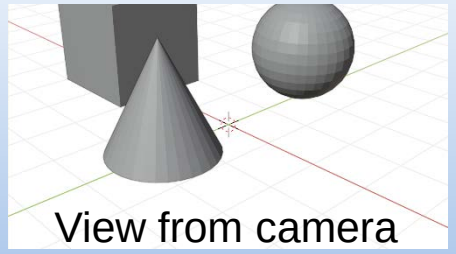



# The Computer Graphics Pipeline

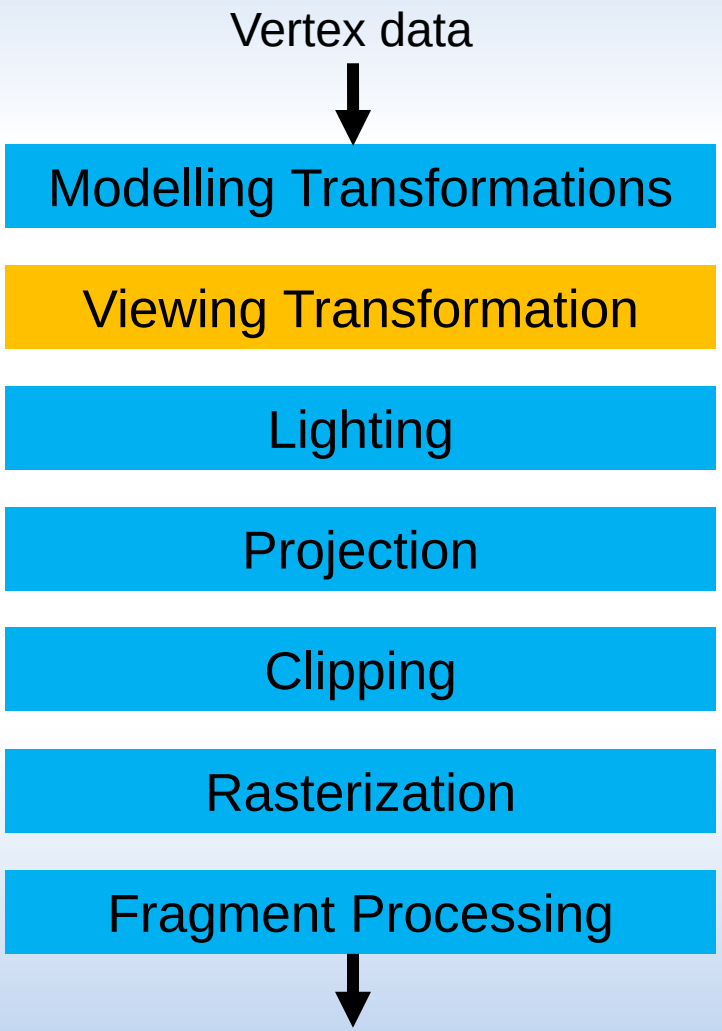
- Viewing transformations
  - ➢ World space to **view space** (or camera/eye space)
    - Viewing position transformed to origin and direction orientated along the z-axis
    - View seen from camera's viewpoint based on camera's orientation



Camera

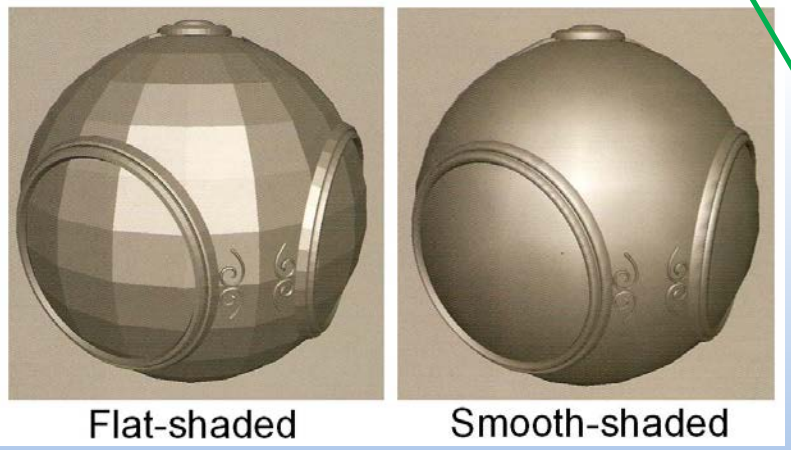


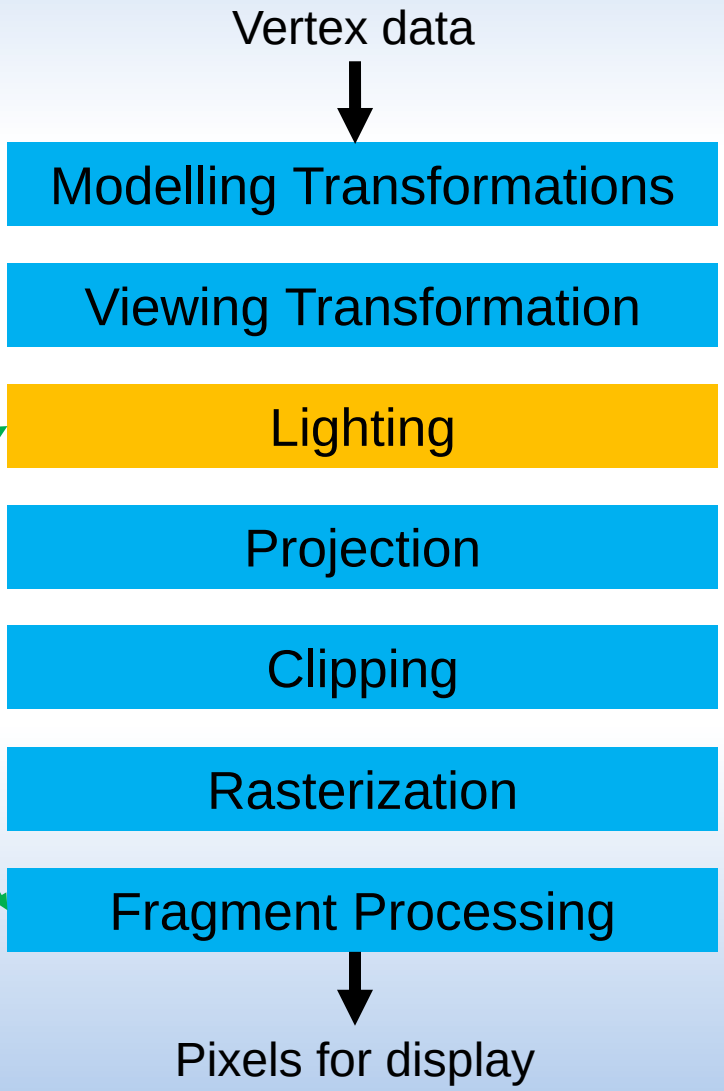
View from camera



# The Computer Graphics Pipeline

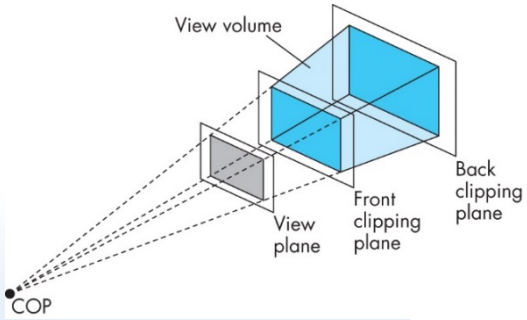
- Lighting
  - ➢ Local lighting model
    - Calculate intensity to shade 3D models
      - – Based on normals, light and material properties, etc.
      - – Per-vertex or per-pixel
  - ➢ Shading model

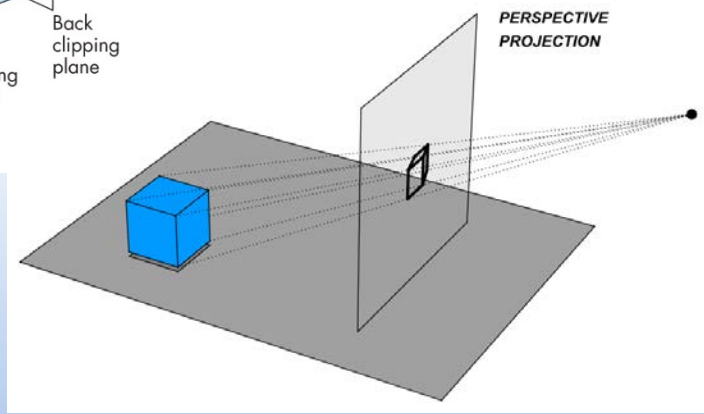


# The Computer Graphics Pipeline

- Projection
  - ➢ 3D scene projected to 2D surface
    - Defines a **view volume**
    - Transformed into normalized device coordinates (NDC)




Vertex data

↓

Modelling Transformations

Viewing Transformation

Lighting

Projection

Clipping

Rasterization

Fragment Processing

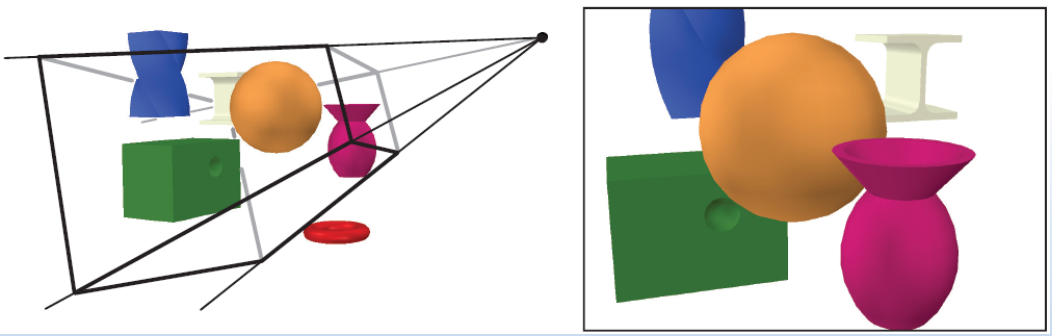
↓

Pixels for display

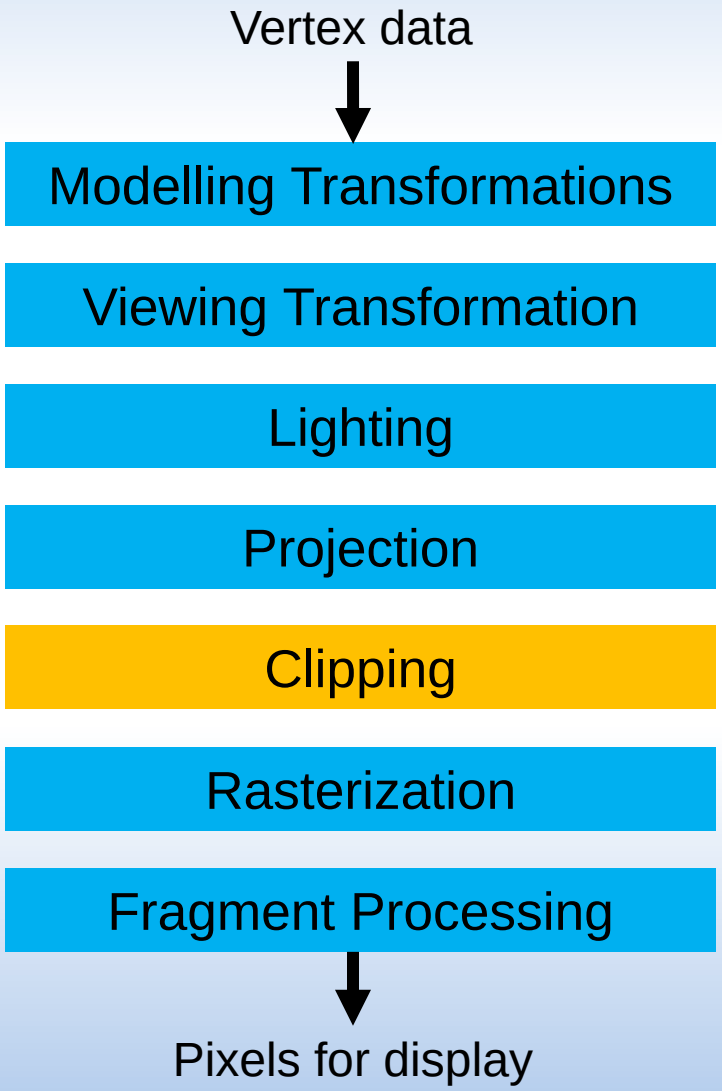
https://commons.wikimedia.org/wiki/File:Various_projections_of_cube_above_plane.svg

# The Computer Graphics Pipeline

- Clipping
  - ➢ Clip scene against view volume
    - Removes parts of scene not within view volume
  - ➢ **Clip space**
    - Coordinate system where clipping is performed

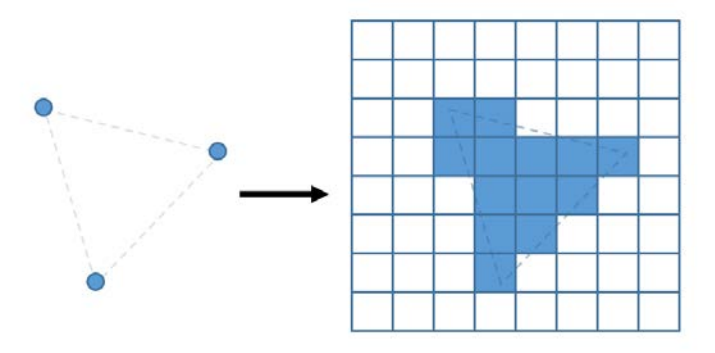


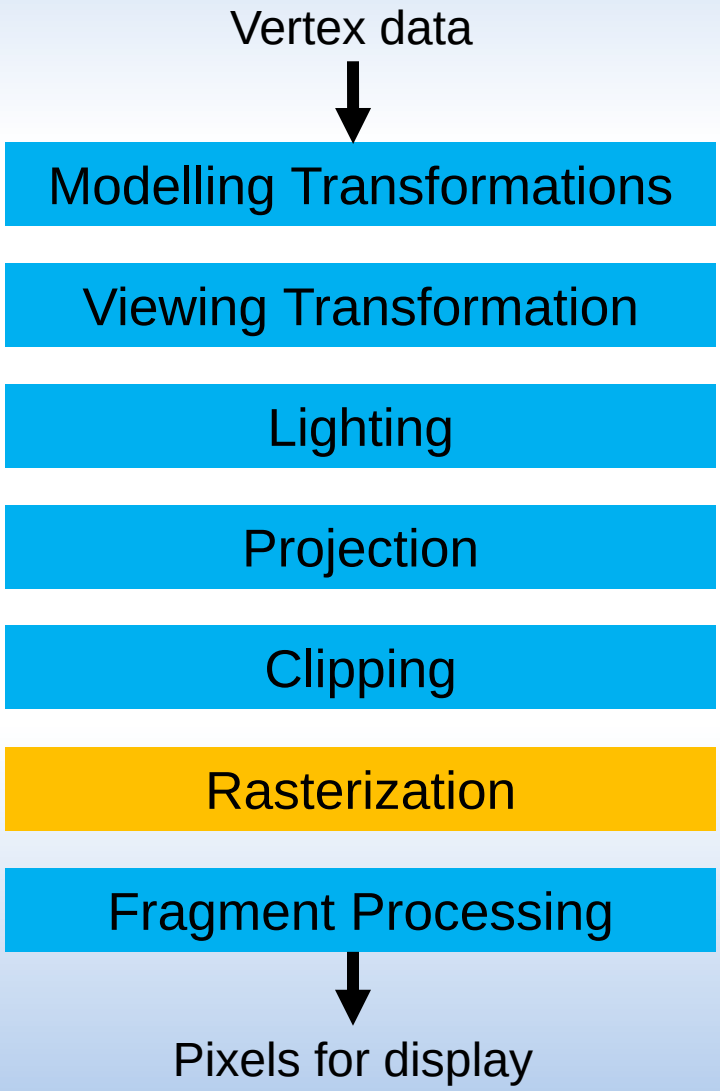




# The Computer Graphics Pipeline

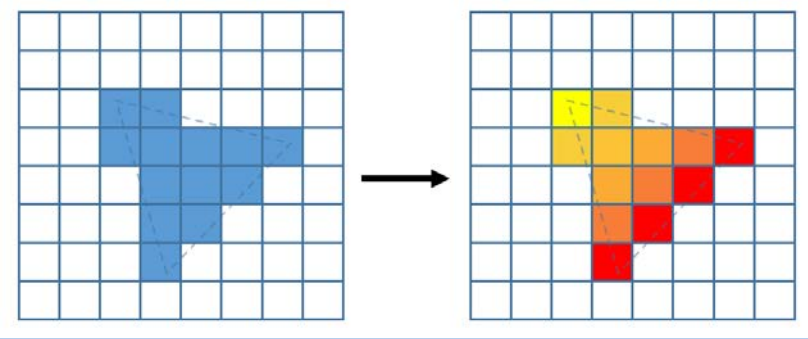
- Rasterization
  - Scan converts vertices into **screen space**
    - Input: vertices (after clipping)
    - Output: fragments
  - After rasterization, everything in fragments
    - No longer polygons or vertices

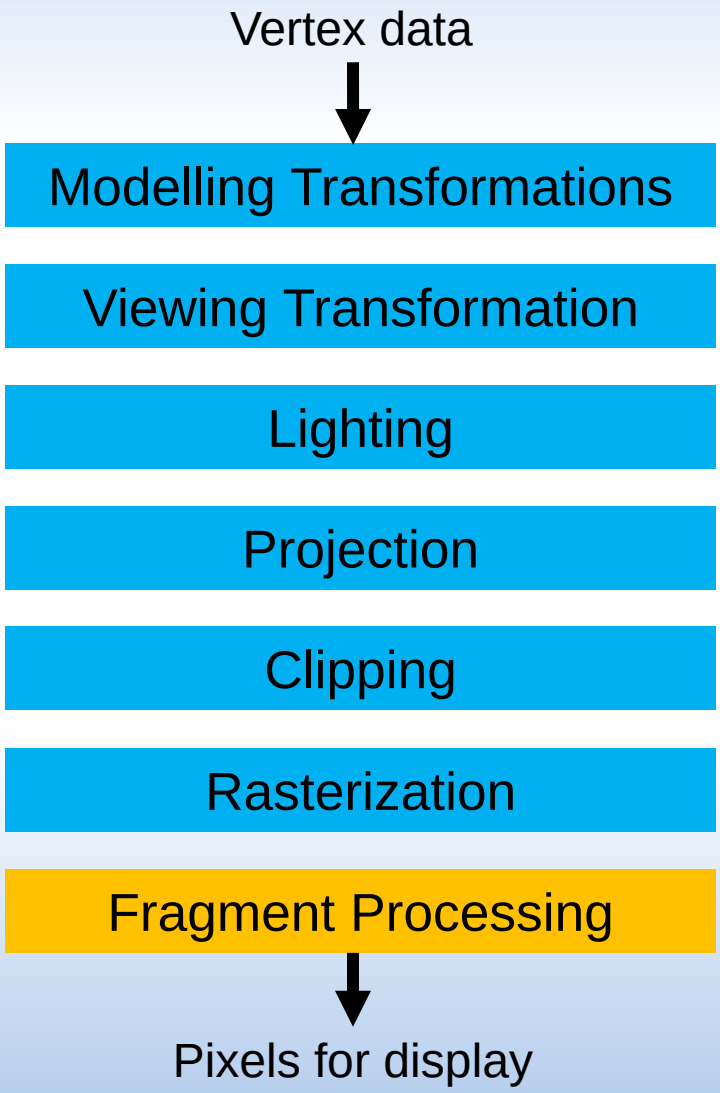




# The Computer Graphics Pipeline

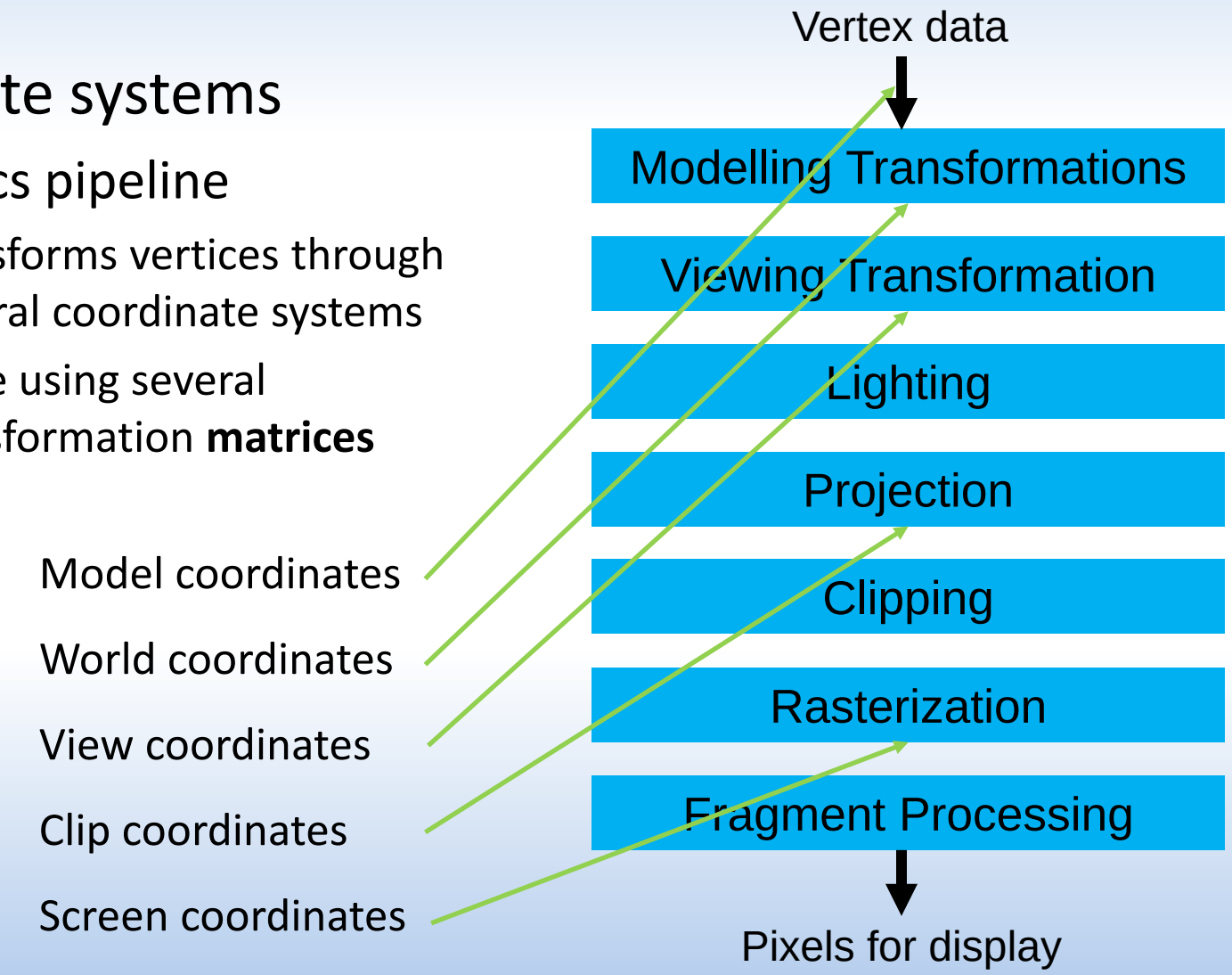
- Fragment processing
  - ➢ Operations to determines final pixel colour
    - Buffer operations
      - – Colour buffer, depth buffer, stencil buffer, etc.
    - Texture mapping
    - Blending
    - Per-pixel lighting





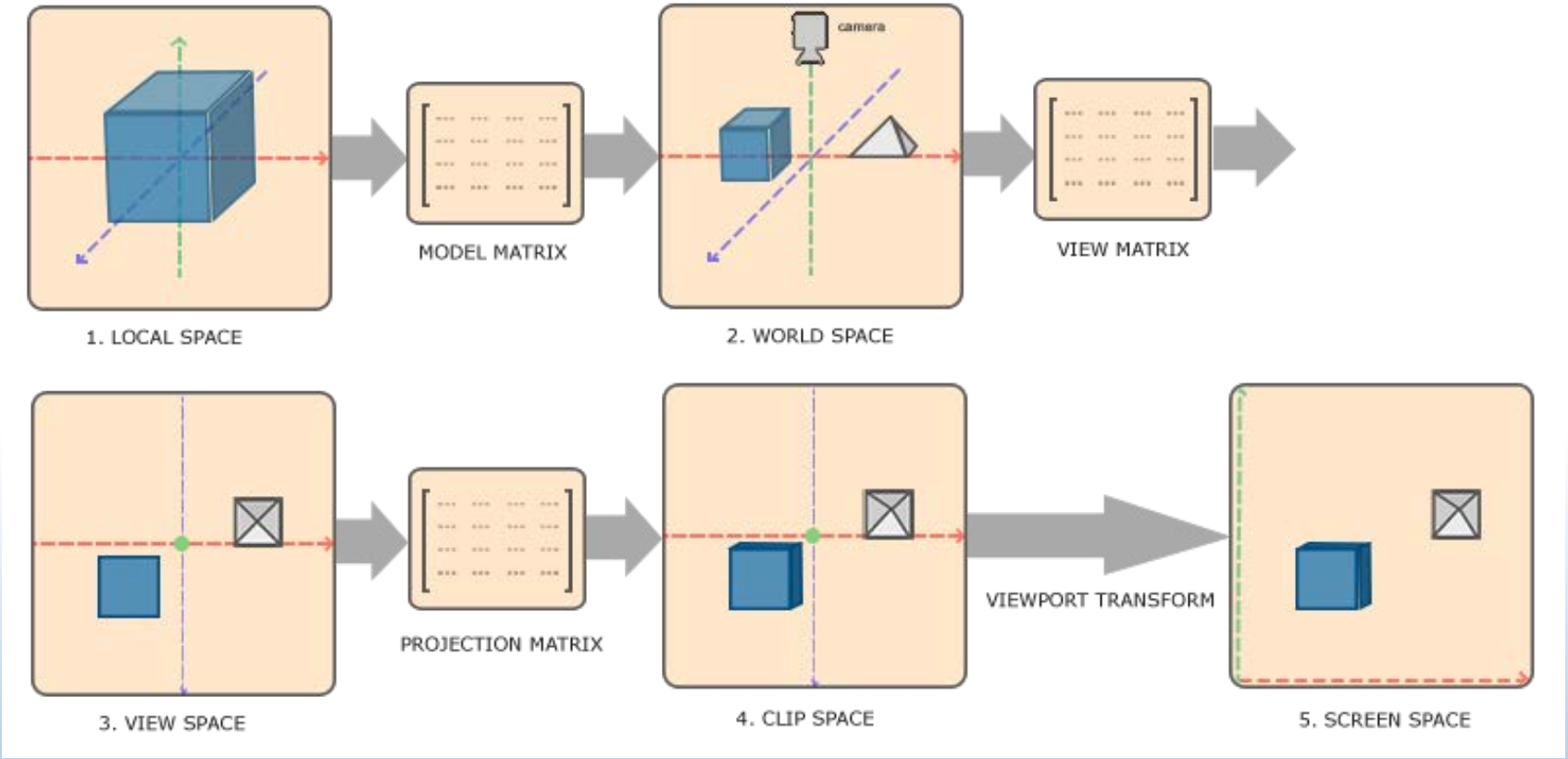
# The Computer Graphics Pipeline

- Coordinate systems
  - ➢ Graphics pipeline
    - Transforms vertices through several coordinate systems
    - Done using several transformation **matrices**



# The Computer Graphics Pipeline

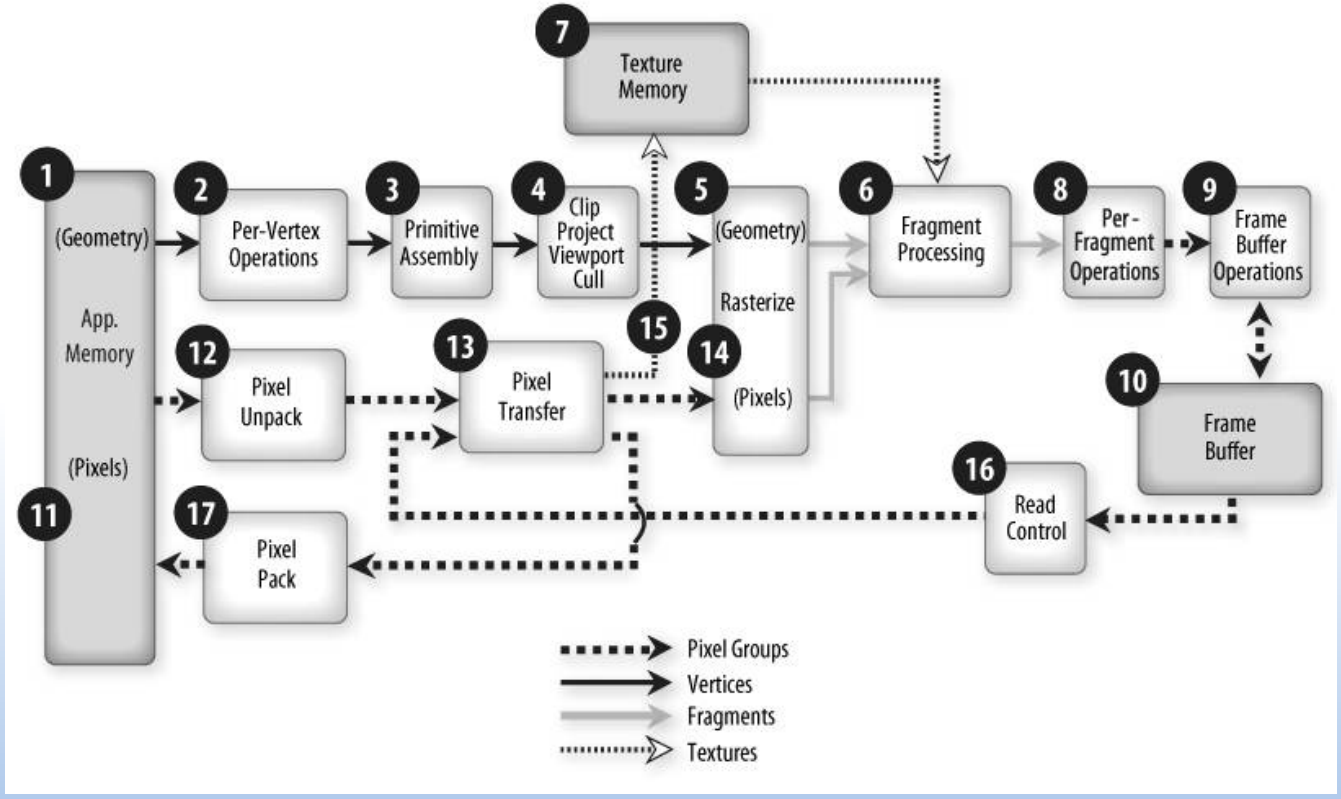
- Coordinate systems
  - Transformation matrices




https://learnopengl.com/Getting-started/Coordinate-Systems

# OpenGL

- Rendering pipeline
  - Simplified representation of the **fixed-function** OpenGL pipeline (legacy OpenGL)



# OpenGL

- Legacy OpenGL
  - Can still use due to backwards-compatibility
    - Programs will run
  - Performance implications
  - Implements fixed-functioned graphics pipeline
- Modern OpenGL
  - Core profile 3.3 and above
    - Latest is 4.6
  - Shader based
    - **Programmable pipeline**
      - Write custom operations for some pipeline stages

# OpenGL

- Example of legacy OpenGL code

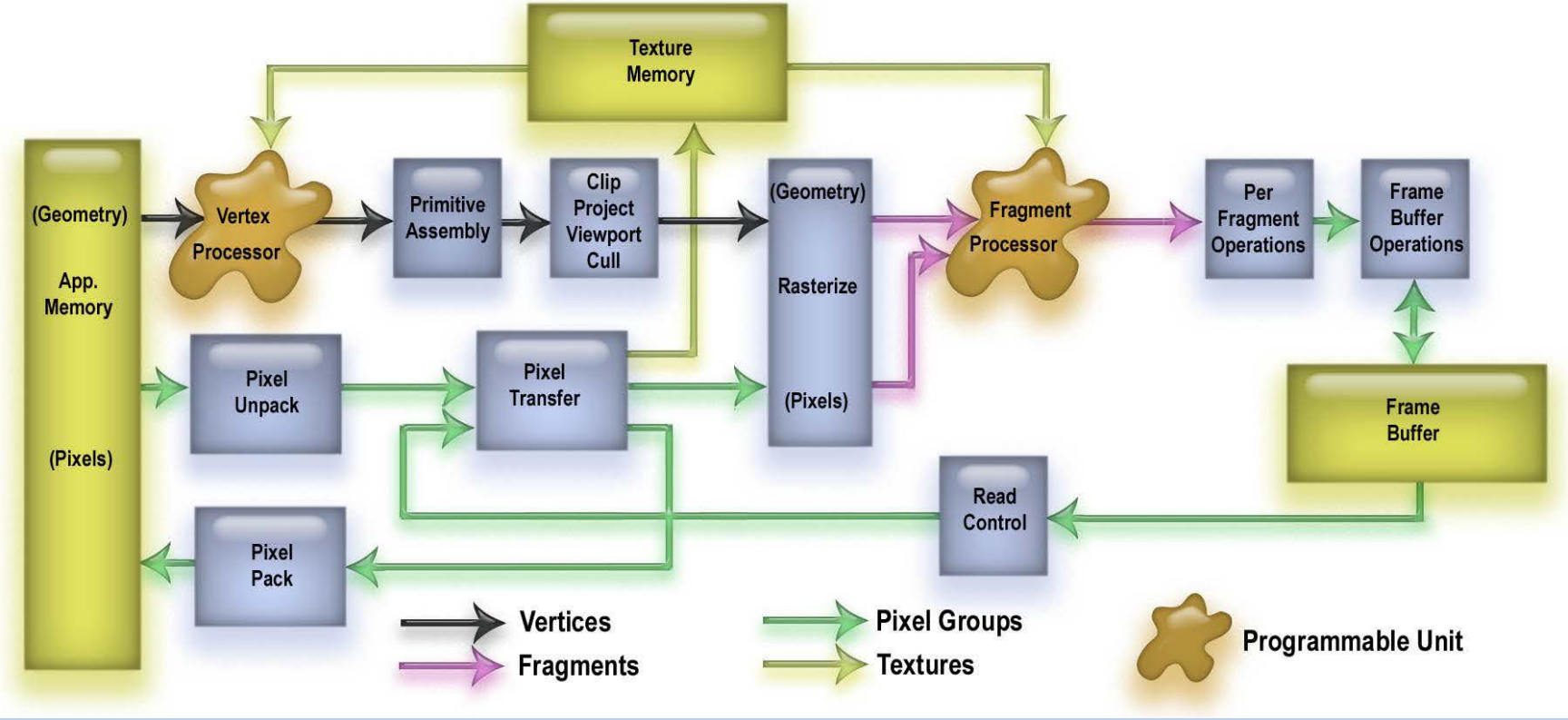
```
glBegin(GL_POLYGON);
  glVertex2f(-0.5, -0.5);
  glVertex2f(-0.5, 0.5);
  glVertex2f(0.5, 0.5);
  glVertex2f(0.5, -0.5);
glEnd();

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(90.0, 0.0, 1.0, 0.0);
glTranslatef(0.0, 0.0, -5.0);
```

DO NOT USE legacy OpenGL in this subject

# OpenGL

- Rendering pipeline
  - OpenGL programmable pipeline



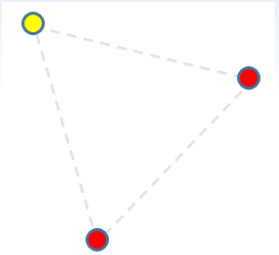
# Shaders

- What are shaders?
  - A **shader** is code intended for execution on one of the programmable processors
  - Compiled and linked to form executable code called a **program** (i.e. a shader program)
    - Contains one or more executables that can run on the programmable processing units
      - Much like a normal CPU program, but instead executed on a graphics processor
    - Normally small programs (compared with application programs)
      - Speed is of utmost importance
      - For parallelisation purposes

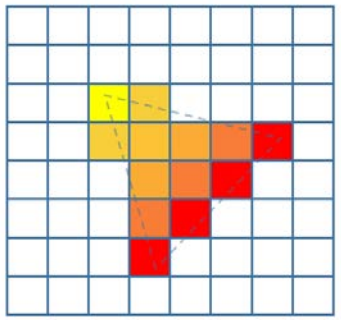
# Shaders

- Programmable processors
  - Started off with
    - Vertex processor
    - Fragment processor
      - Often referred to as pixel processor
  - Two more were introduced later (OpenGL 4.0)
    - Geometry and tessellation shaders (optional)
- Shading language
  - **GLSL** (OpenGL Shading Language)
    - C-style language

# Shaders



- Vertex shader
  - Used to process vertices
  - **Per-vertex** operations
    - Vertex transformations, per-vertex lighting, per-vertex colour, handling texture coordinates, etc.



- Fragment shader
  - To manipulate fragments
  - **Per-fragment** operations
    - Colour, texture lookup, per-pixel lighting, blending, etc.
  - Produces final RGBA colour for each pixel location in the frame buffer

# Simple Vertex Shader

```
#version 330 core

// input
layout(location = 0) in vec3 aPosition;

void main()
{
    // set vertex position
    gl_Position = vec4(aPosition, 1.0f);
}
```

# Simple Fragment Shader

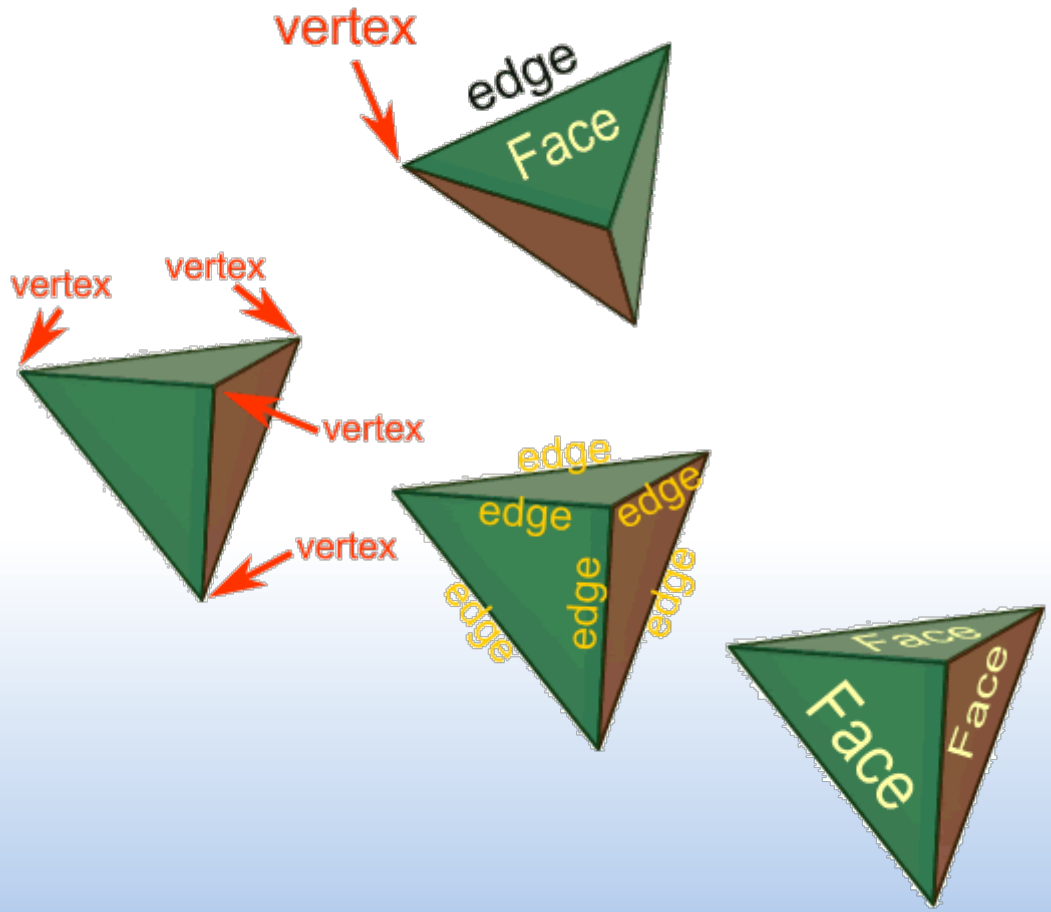
```
#version 330 core

// output data
out vec3 fColor;

void main()
{
    // set output color
    fColor = vec3(1.0f, 0.0f, 0.0f);
}
```

# The Computer Graphics Pipeline

- Vertex (plural: vertices)




Vertex data

↓

Modelling Transformations

Viewing Transformation

Lighting

Projection

Clipping

Rasterization

Fragment Processing

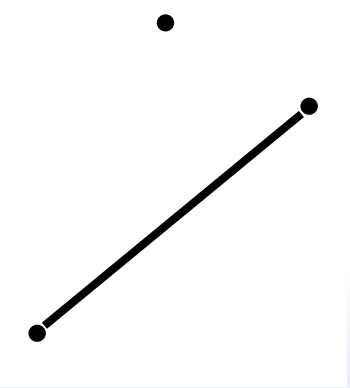
↓

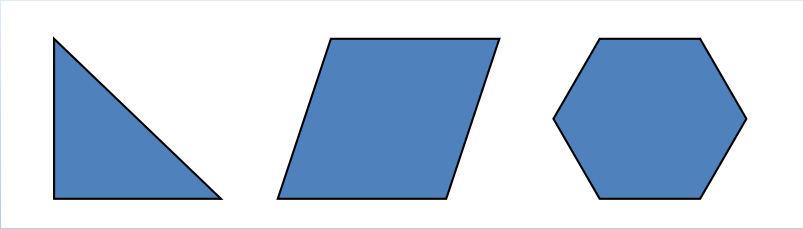
Pixels for display

https://www.mathsisfun.com/geometry/vertices-faces-edges.html

# Primitives

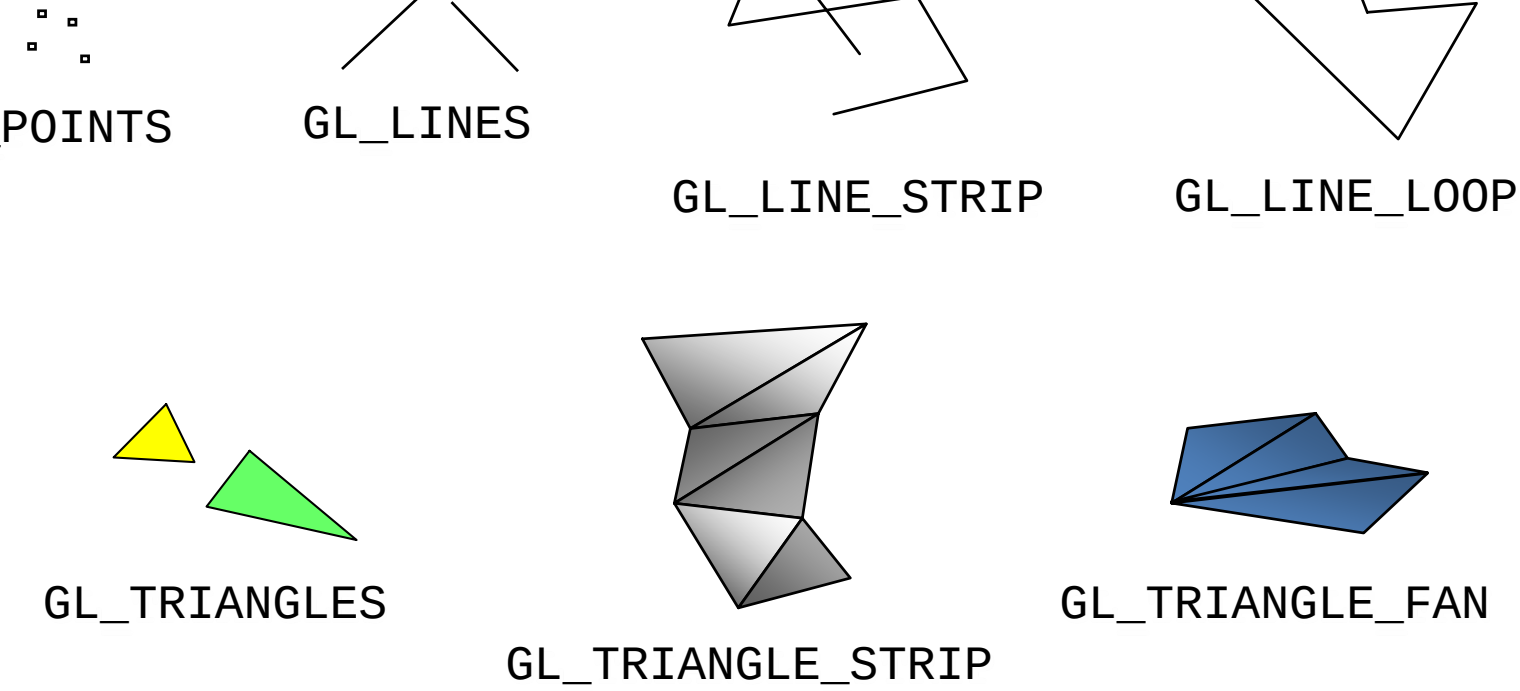
- **Primitive assembly**
  - Vertices must be collected into geometric objects before clipping and rasterization can take place
- Primitives
  - Points
  - Lines
    - Bounded by two end points
  - Polygons
    - Bounded by a closed circuit of lines





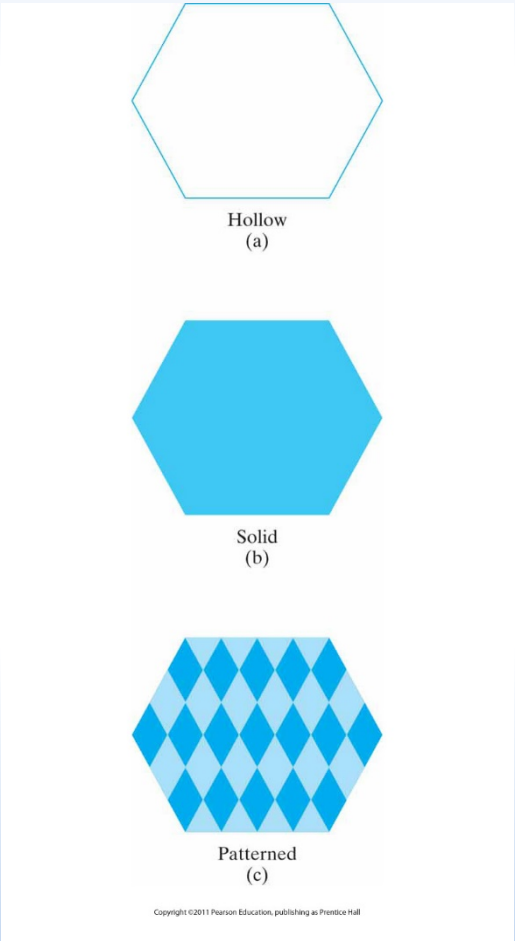
# Primitives

- OpenGL Primitives



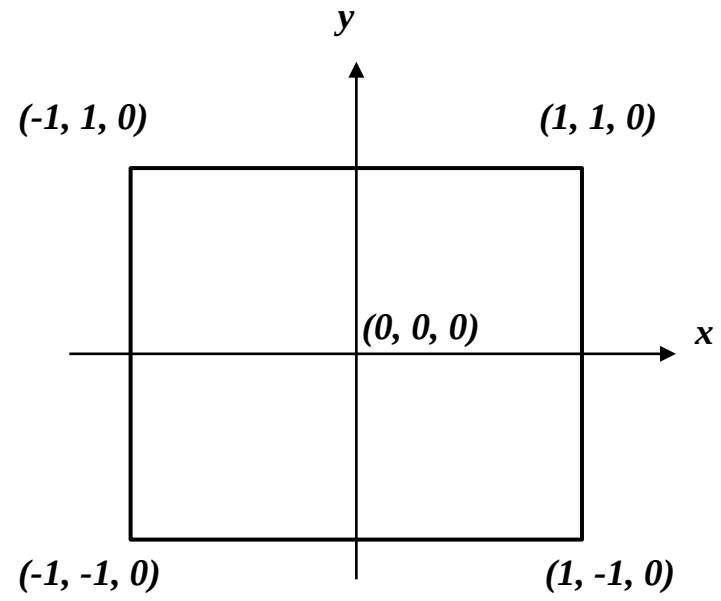
# Primitives

- Attributes
  - Vertex attributes
    - Position
    - Colour
  - Point size
    - `gl_PointSize` in vertex shader with `GL_PROGRAM_POINT_SIZE` enabled
    - These will use the nearest integer size
  - Wireframe/polygon fill
    - `glPolygonMode()`
  - Textured

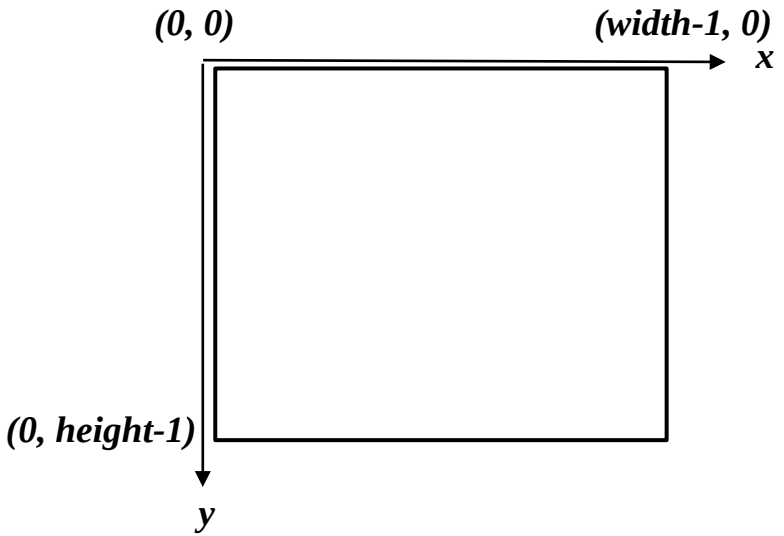


# Coordinate Systems

- Position
  - 2D coordinates: (x, y)
  - 3D coordinates: (x, y, z)




Default rendering coordinate system
Normalized device coordinates (NDC)

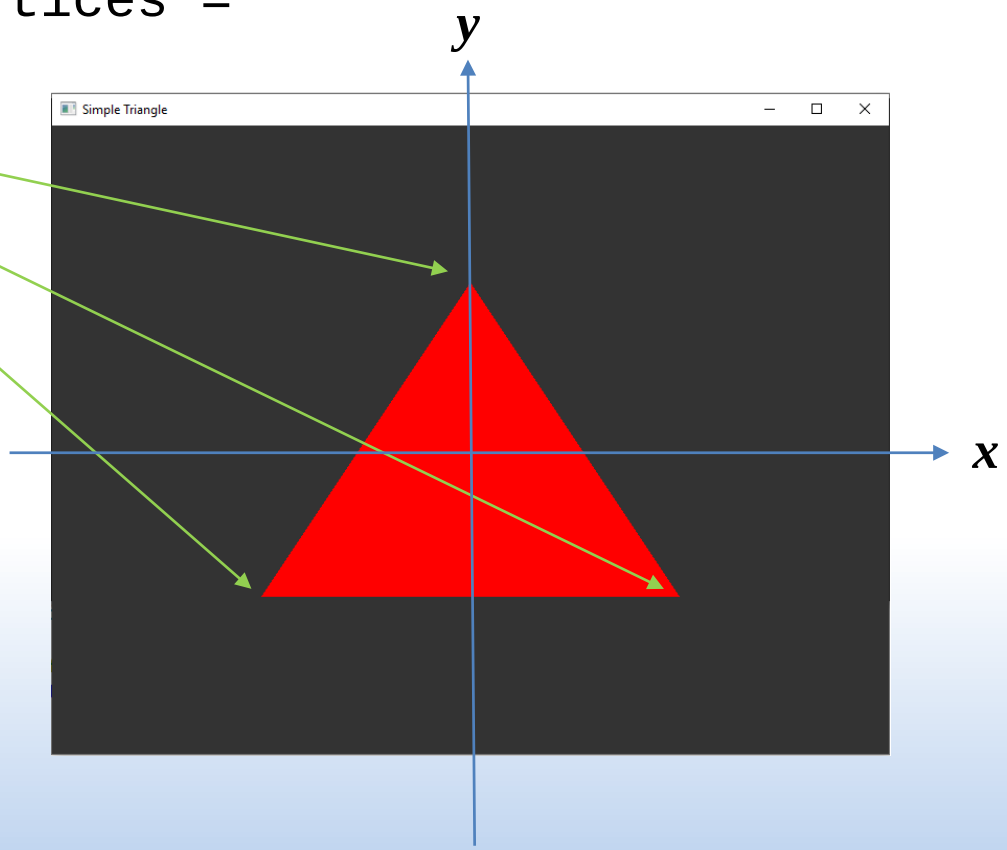



Mouse cursor coordinate system
Units are in pixels

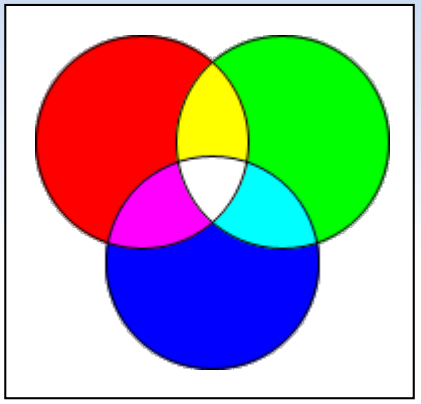
# Primitives

- Position

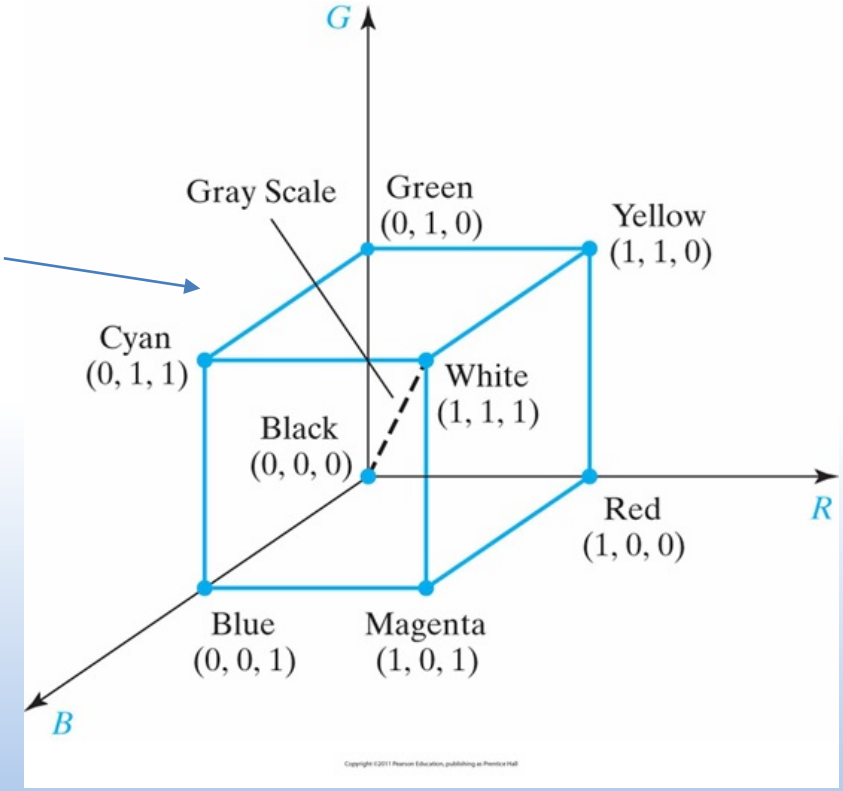
```
std::vector<GLfloat> vertices =
{
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f
};
```



# Primitives

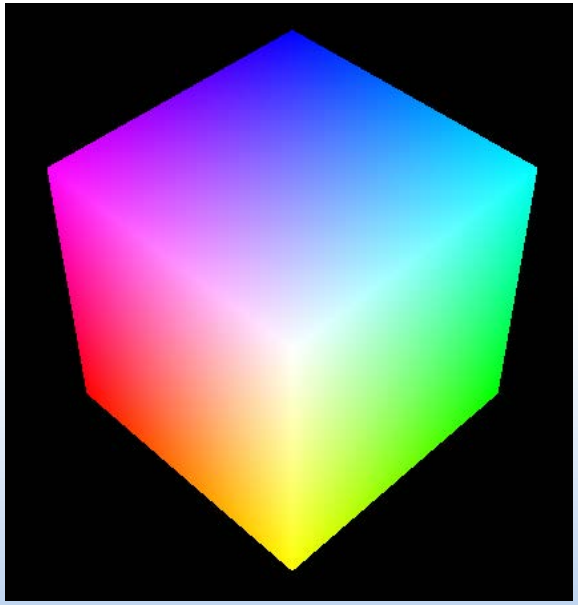


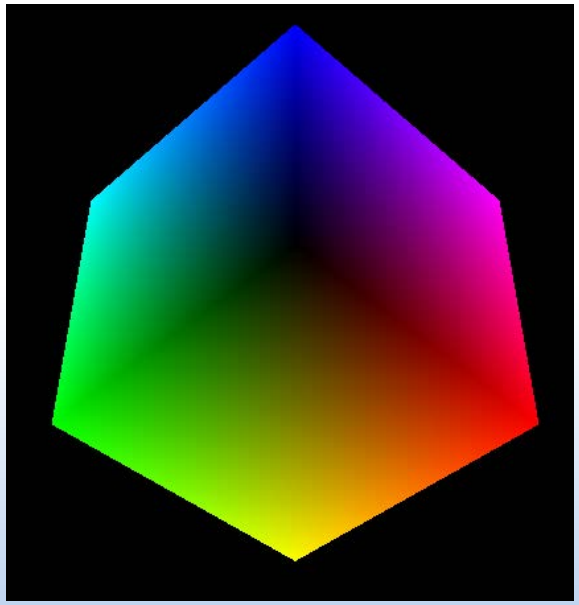
- RGB colour model
  - Additive colour model
  - Three primary colours
    - Red, green, blue
      - RGBA has an additional alpha component
    - Any colour within the unit cube can be described as an additive combination of the 3 primary colours



# Primitives

- RGB colour model
  - Each pixel has a separate RGB colour component, typically 8-bits each
    - i.e. 24-bits per pixel (or 32-bits for RGBA)





# Graphics Programming

- Vertex data
  - Submitting vertex data for rendering requires creating a stream of vertices
    - **Vertex buffer object** (VBO)
  - Tell OpenGL how to interpret data in the stream
    - Format of the data
      - `glVertexAttribPointer()`
    - **Vertex array object** (VAO)
    - Primitive assembly
      - Tell OpenGL what type of **OpenGL primitive** to render

# Graphics Programming

- Vertex buffer object (VBO)
  - A buffer used as the source for vertex data
    - Vertex coordinates (positions), colours, normals, texture coordinates, etc.
- Vertex array object (VAO)
  - Stores all of the state needed to supply vertex data
  - Stores format of the vertex data and buffer objects
- OpenGL is a "**state machine**"
  - Rendering based on the current state

# Graphics Programming

- Passing vertex data
  - Define vertices
    - Floating-point values representing positions

```
std::vector<GLfloat> vertices = {
   0.0f, 0.5f, 0.0f,
   0.5f, -0.5f, 0.0f,
   -0.5f, -0.5f, 0.0f
};
```

  - Use VBO to store and send to graphics device
  - Create VBO

```
GLuint gVBO = 0;
glGenBuffers(1, &gVBO);
```

# Graphics Programming

- Bind buffer object to make it active
  - For vertex attributes: GL_ARRAY_BUFFER

```
glBindBuffer(GL_ARRAY_BUFFER, gVBO);
```

- Buffer the data in graphics device memory

```
glBufferData(GL_ARRAY_BUFFER,
    sizeof(float) * vertices.size(),
    &vertices[0],
    GL_STATIC_DRAW
    );
```

Size of the data

Pointer to the data

Expected usage

```
vertices = {
   0.0f, 0.5f, 0.0f,
   0.5f, -0.5f, 0.0f,
   -0.5f, -0.5f, 0.0f
};
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.5 | 0.0 | 0.5 | -0.5 | 0.0 | -0.5 | -0.5 | 0.0 |

# Graphics Programming

- VAO
  - Store states, one or more VBOs and format of vertex data
  - Create VAO

```
GLuint gVAO = 0;
glGenVertexArrays(1, &gVAO);
```

  - Bind array object to make it active

```
glBindVertexArray(VAO);
```

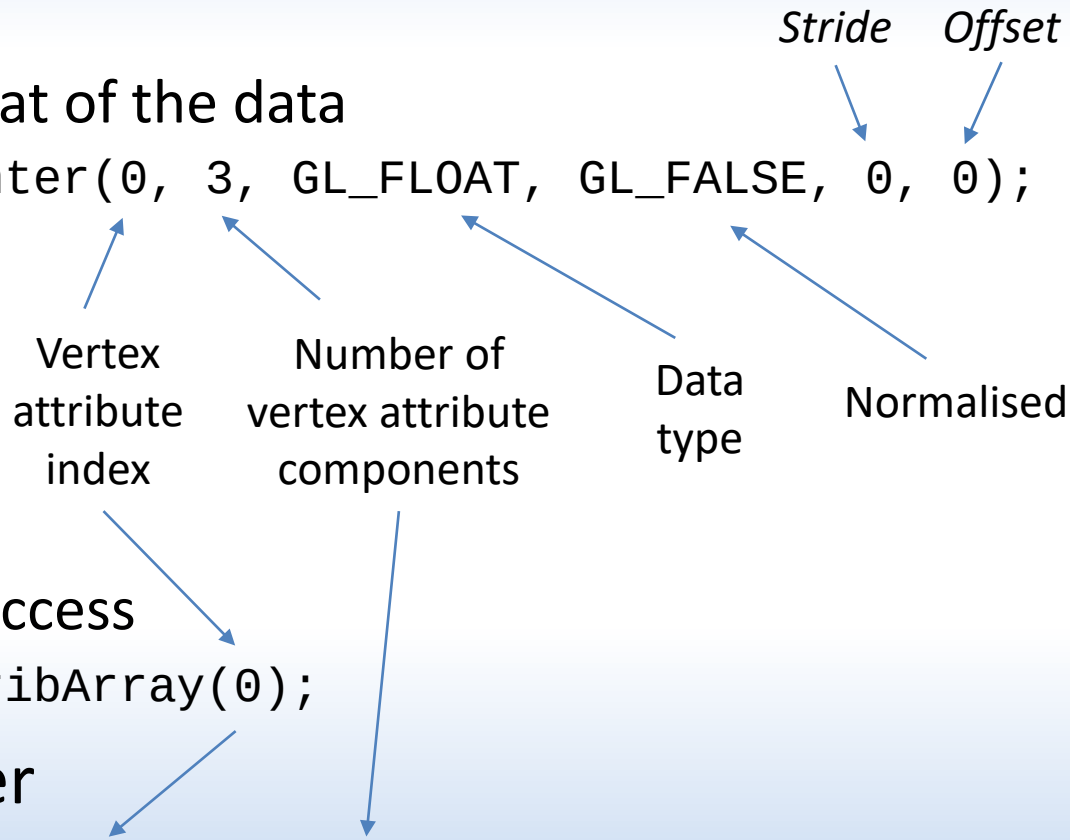
  - Associate it with one or more buffer objects

```
glBindBuffer(GL_ARRAY_BUFFER, gVBO);
```

# Graphics Programming

- VAO
  - Describe format of the data

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```




  - Enable array access

```
glEnableVertexAttribArray(0);
```

- In vertex shader

```
layout(location = 0) in vec3 aPosition;
```

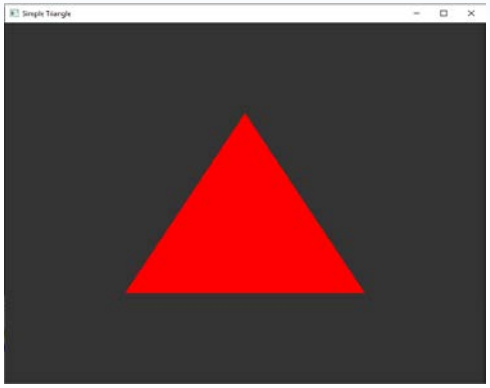
# Graphics Programming

- To render
  - Restore VAO states

```
glBindVertexArray(gVAO);
```

  - Render primitive

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



OpenGL primitive type

Starting index

Number of indices

```
vertices = {
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f
};
```

| 0.0 | 0.5 | 0.0 | 0.5 | -0.5 | 0.0 | -0.5 | -0.5 | 0.0 |
|---|---|---|---|---|---|---|---|---|

Vertex 0 (x, y, z)

Vertex 1 (x, y, z)

Vertex 2 (x, y, z)

# References

- Among others, material sourced from
  - Hearn, Baker & Carithers, “Computer Graphics with OpenGL”, Prentice-Hall
  - Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Addison Wesley
  - Bailey & Cunningham, “Graphics Shaders: Theory and Practice,” CRC Press
  - Randi Rost, “OpenGL Shading Language,” Addison-Wesley
  - Akenine-Moller, Haines & Hoffman, “Real-Time Rendering”, A.K. Peters
  - Joey de Vries, “Learn OpenGL,” https://learnopengl.com/
  - MIT OpenCourseWare, https://ocw.mit.edu/
  - https://www.khronos.org/opengl/wiki/
  - http://en.wikipedia.org/wiki/