OpenGL Tutorial

CS 334 Fundamentals of Computer Graphics

https://learnopengl.com/

What is OpenGL?

- A rendering library (written in C); API (Application Programming Interface)
- Bindings: C/C++ (official), C#, Python, etc.
- Version: 3.3

```
#version 330 core
```

OpenGL development environment

- Linux, Windows (Visual Studio, a demo project will be posted)
- Libraries: *Freeglut, GLEW, GLM*

```
#include <glm/glm.hpp>
#include <GL/glew.h>
#include <GL/glut.h>
```

- *Freeglut*: OpenGL utility toolkit, creating windows, handling input events
- *GLEW*: OpenGL Extension Wrangler Library, providing extensions
- *GLM*: OpenGL Mathematics

OpenGL development environment (*optional*)

- Visual Studio + *GLFW* + *GLAD* + *ImGui* + *GLM*
- *GLFW*: a library providing simple API for creating windows, contexts and surfaces, receiving input and events
- *GLAD*: a library simplifying the work
- *ImGui*: a graphical user interface (GUI)

```
#include <glad/glad.h>  // always put BEFORE GLFW
#include <GLFW/glfw3.h>
#include "imgui.h"
#include "imgui_impl_glfw.h"
#include "imgui_impl_opengl3.h"
```

OpenGL - A large state machine

State machine

- *Context*: the state of OpenGL; defines what it should operate now
- Definition: a collection of context variables

A common workflow:

- Creating an object -> storing reference as as id -> binding it to target location of the context -> unbinding

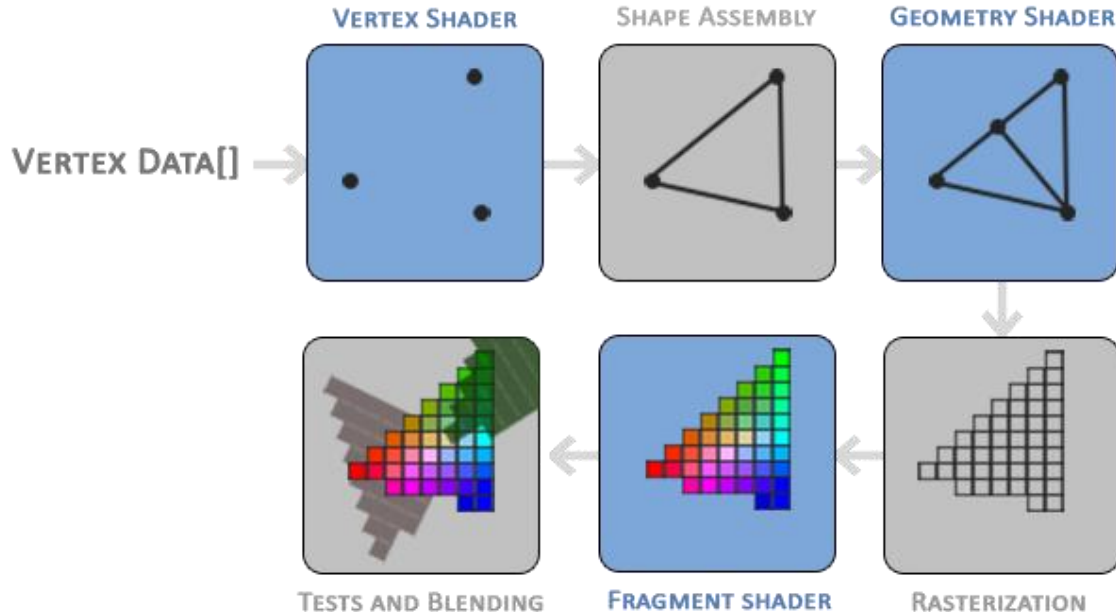
```
// create object
unsigned int objectId = 0;
glGenObject(1, &objectId);
// bind/assign object to context
glBindObject(GL_WINDOW_TARGET, objectId);
// set options of object currently bound to GL_WINDOW_TARGET
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH,  800);
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_HEIGHT, 600);
// set context target back to default
glBindObject(GL_WINDOW_TARGET, 0);
```

Graphics Pipeline - Overview

3D model coordinates -> 2D colored pixels on screen

Shaders: the steps can easily be executed in parallel

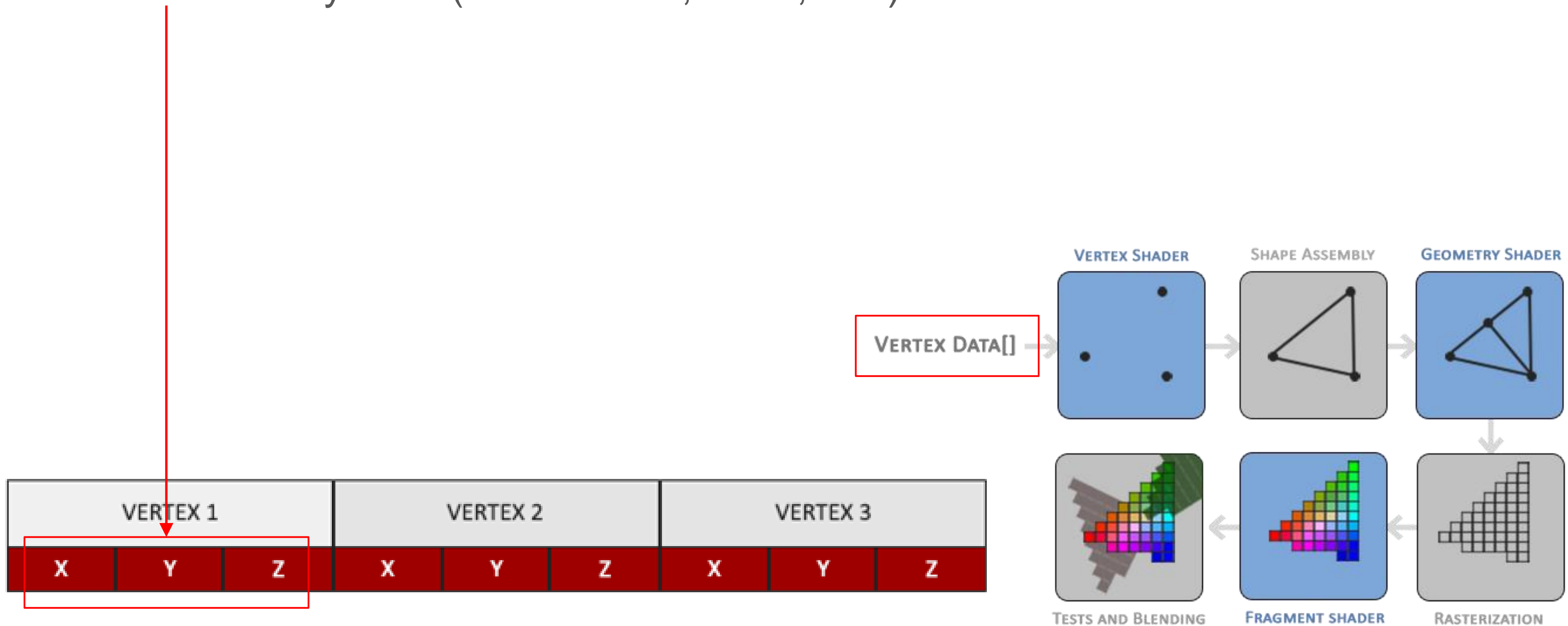
- Definition: tiny programs on GPU
- Replacement: *vertex shader* and *fragment shader*
- *GLSL*: OpenGL Shading Language



Graphics Pipeline 1 - Vertex data

Vertex data

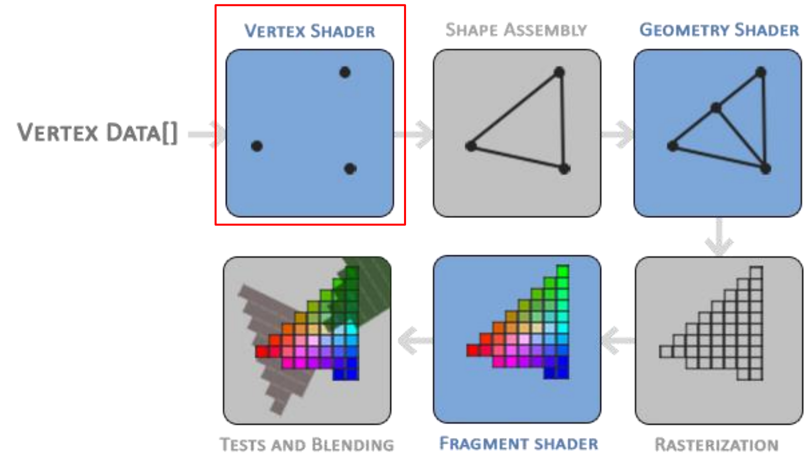
- A list of 3D coordinates / vertices
- *Attributes*: any data (coordinates, color, etc.) contained in a vertex



Graphics Pipeline 2 - Vertex shader

Vertex shader (required)

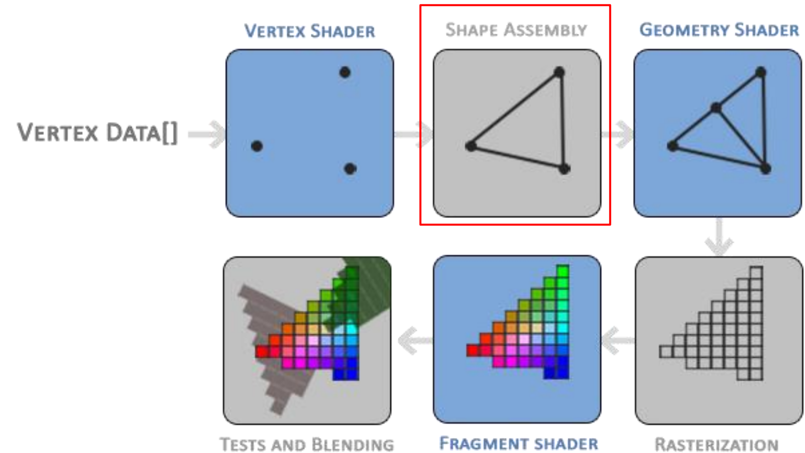
- Input: vertex data
- Transformation on 3D coordinates



Graphics Pipeline 3 - Shape Assembly

Shape assembly

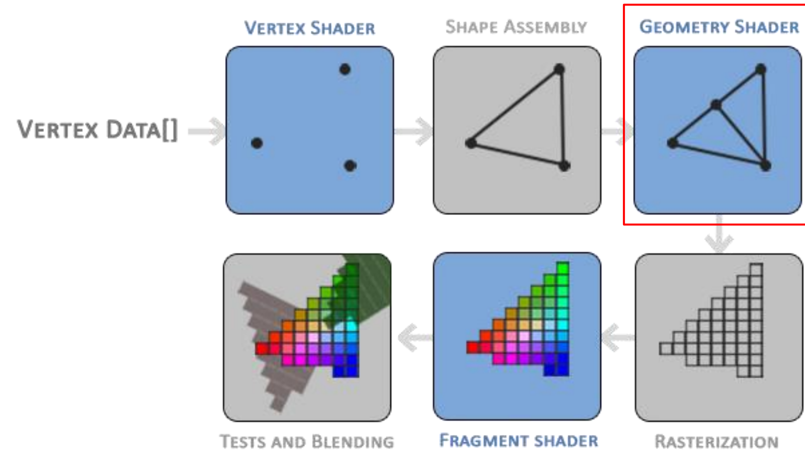
- Assemble all the points into a specific primitive shape (e.g., triangle, points, etc.)



Graphics Pipeline 4 - Geometry shader

Geometry shader

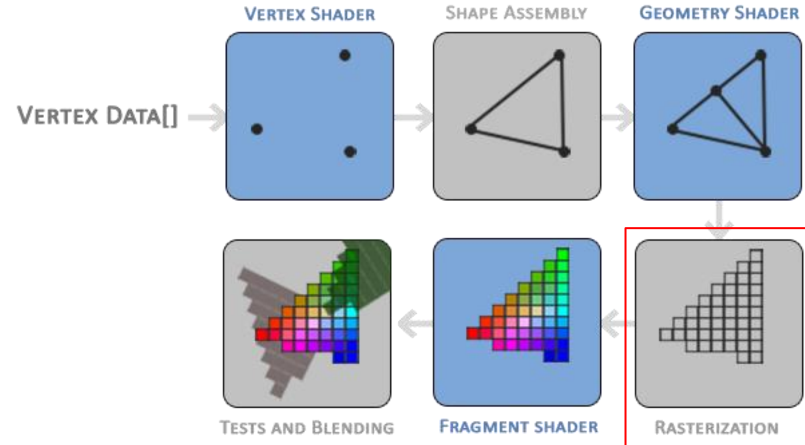
- Generate other shapes / primitives (e.g., insert vertices)
- Will not be used



Graphics Pipeline 5 - Rasterization

Rasterization stage

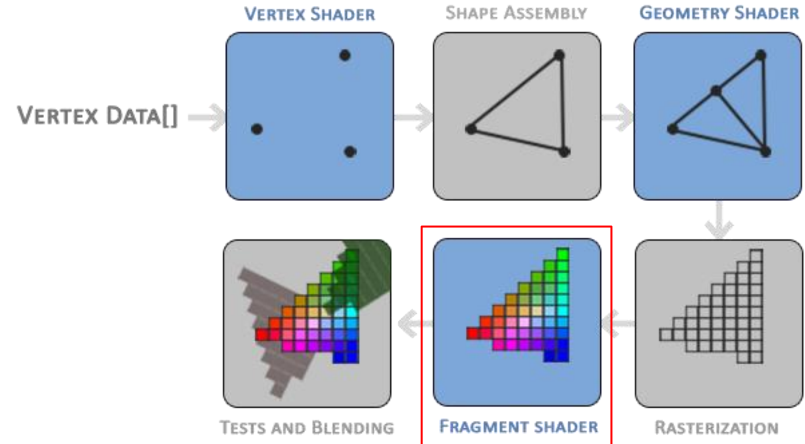
- Map the primitive shapes to pixels on the screen
- Clip the fragments falling outside the view



Graphics Pipeline 6 - Fragment Shader

Fragment shader (required)

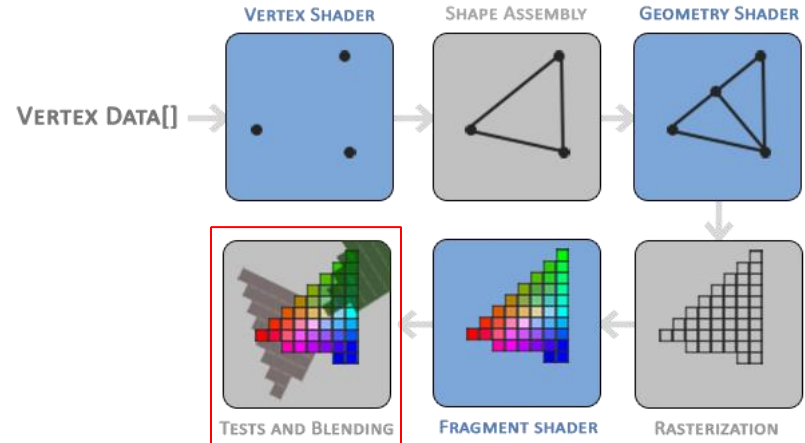
- Calculate the final color for each pixel
- Apply other high-level effects (illumination, shadow, etc.)



Graphics Pipeline 7 - Tests and Blending

Tests and blending

- Perform depth test and decide the occlusion
- Check opacity and blend the objects



Get started 1 - Vertex Input

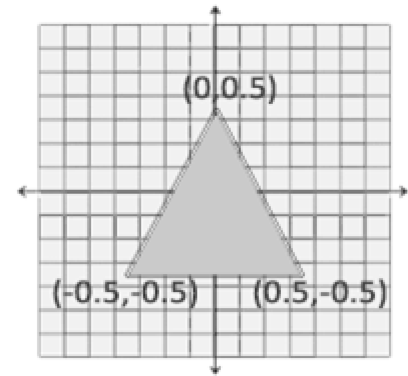
Ranges:

- Coordinates range: [-1.0, 1.0]; color range: [0.0, 1.0]

NDC (Normalized Device Coordinates):

- Vertex shader only processes NDC
- Data outside this range will be clipped (*recall*...)

```
float vertices[] = {  // attributes
    // positions            // colors
    0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // bottom left
    0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // top
};
```



Get started 2 - VBO

Memory management (before drawing):

- Copy the geometry data to GPU (send data all at once)
- Specify the vertex format

VBO: Vertex Buffer Objects

- The memory is managed via VBO which stores vertice data in GPU

Get started 2 - VBO (continue)

How to use VBO: recall the *common workflow*...

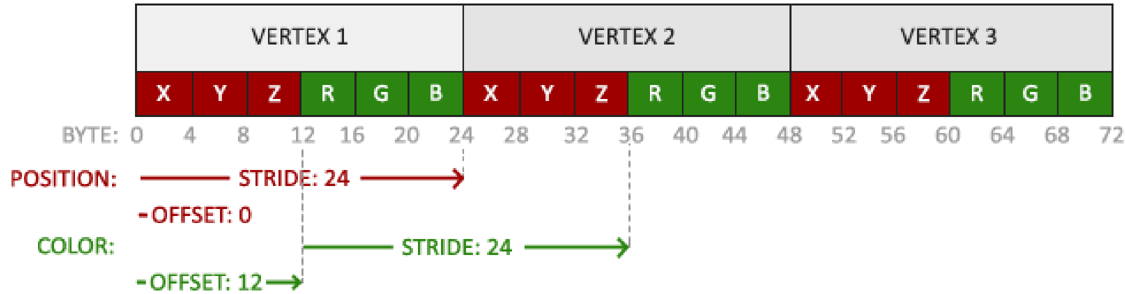
```
unsigned int VBO;  // generate a unique id
glGenBuffers(1, &VBO);  // generate id(s)
glBindBuffer(GL_ARRAY_BUFFER, VBO);  // bind the buffer to the target
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// copy data into the buffer’s memory
// 1st argument: the type of buffer to copy data to
// 2nd argument: specifies the size of data (bytes)
// 3rd argument: actual data to send
// 4th argument: the data is set once and used many times
glBindBuffer(GL_ARRAY_BUFFER, 0);  // unbind
```

```
unsigned int VBOs[5];  // generate multiple buffers
glGenBuffers(5, &VBOs);
```

Get started 3 - Vertex Specification

Specify the vertex data by specifying the attributes

- How many attributes
- Size of each attribute
- Where attributes are stored




```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Get started 3 - Vertex Specification (continue)

How to specify: via *attribute pointers*

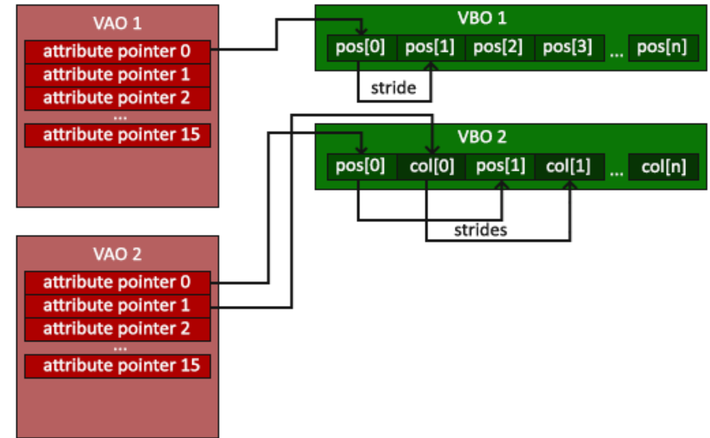
```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);

// 1st argument: location of the attribute
// 2nd argument: number of components
// 3rd argument: data type
// 4th argument: normalize integers (divide by 255 -> 0 or 1)
// 5th argument: stride
// 6th argument: offset

// glEnableVertexAttribArray: enable the attribute giving the location as argument
```

Get started 4 - VAO

Different objects have different vertex formats

VAO (Vertex Array Object):

- Store vertex specifications and which VBO to use
- Allow quickly switching between different vertex formats




```
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
// define VBO and specify vertex attributes using pointers (in previous pages)
glBindVertexArray(0);
```

Get started 5 - Vertex Shader

A pipeline: vertex data -> vertex shader -> interpolated values -> fragment shader -> output colors

Vertex shader (written in *GLSL*):

- Process each vertex in parallel
- Usually apply transformations: model -> world -> view -> projection (*later*)
- In the end it must write to `gl_Position` built-in variable

Vectors:

- A vector in GLSL has a max size of 4 (`vec.x, vec.y, vec.z, vec.w`)
- `vec.w`: perspective division (*later*)

```
#version 330 core
layout (location = 0) in vec3 aPos;  // the first attribute
layout (location = 1) in vec3 aColor;  // the second attribute
out vec3 ourColor;  // pass the color data to the fragment shader
void main() {
   gl_Position = vec4(aPos, 1.0);  // output
   ourColor = aColor;
};
```

Get started 6 - Fragment Shader

Fragment shader (written in *GLSL*):

- Calculate the color of the pixels that are inside the the primitives
- Colors representation: RGBA (red, geen, blue and alpha or opacity) within [0.0, 1.0]
- Calculate lighting, texture, etc.

```
#version 330 core
out vec4 FragColor;  // output color
in vec3 ourColor;  // vertex color, passed by the vertex shader
void main() {
   FragColor = vec4(ourColor, 1.0f);
}
```

Get started 7 - Shader Compiling

- Store the shader in a constant C string
- Must be compiled and linked at runtime
- Recall the *common workflow*…
- Check if the compilation is successful (see *test0.c*)

```
const char *vertexShaderSource = "#version 330 core\n"
  "layout (location = 0) in vec3 aPos;\n"
  "layout (location = 1) in vec3 aColor;\n"
  "out vec3 ourColor;\n"
  "void main()\n"
  "{\n"
  "   gl_Position = vec4(aPos, 1.0);\n"
  "   ourColor = aColor;\n"
  "}\0";
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);  // referenced by an id
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);  // attach the shader obj
// 1st argument: the object
// 2nd argument: how many strings
glCompileShader(vertexShader);
```

Get started 8 - Shader Linking

Shader program:

- The final linked version of all shaders combined
- Link the outputs of one shader to the inputs of the next shader

```
unsigned int shaderProgram = glCreateProgram();  // referenced by an id
glAttachShader(shaderProgram, vertexShader);  // attach the shaders to the program obj
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
// check for linking errors...
glDeleteShader(vertexShader);  // once we link the shader objects, they are no longer needed
glDeleteShader(fragmentShader);
glUseProgram(shaderProgram);  // activate the program
```

Get started 9 - Summary

The general process:

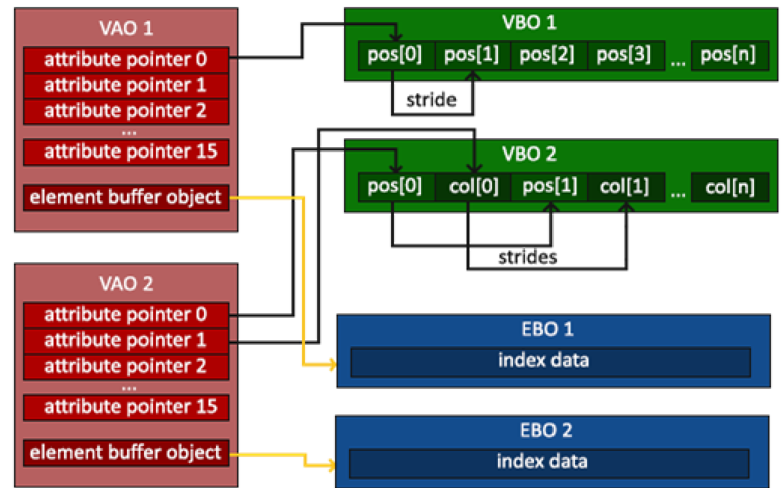
```
// Initialization code...
// 1. bind VAO
glBindVertexArray(VAO);
// 2. copy our vertices array to VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. set attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 4. use the shader program (compile and link)
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
// draw objects…
glDrawArrays(GL_TRIANGLES, 0, 3);
// 1st argument: primitive (can also be points, lines)
// 2nd argument: starting index of the vertex array
// 3rd argument: how many vertices to draw
// 5. unbind
```

Advanced features 1 - EBO

Problem: overlap in triangle vertices

EBO (element buffer objects):

- Store indices for deciding which vertices to draw
- Specify the *unique* vertices and the indices to draw

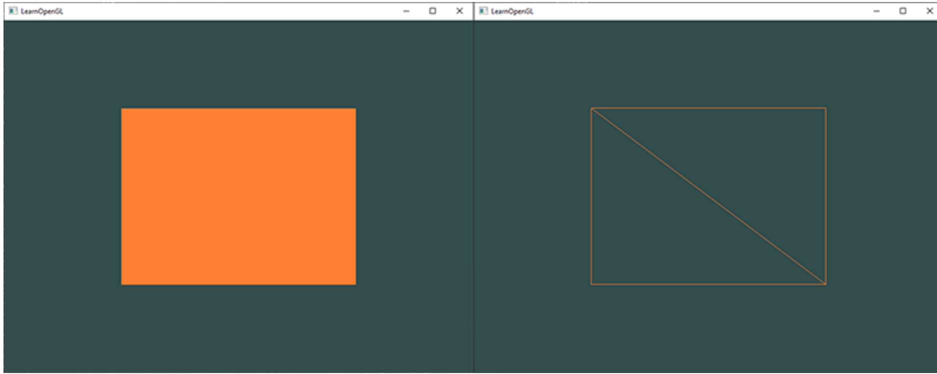



```
float vertices[] = {
    // first triangle
     0.5f,  0.5f, 0.0f,   // top right
     0.5f, -0.5f, 0.0f,   // bottom right
    -0.5f,  0.5f, 0.0f,   // top left
    // second triangle
     0.5f, -0.5f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f    // top left
};
```

```
float vertices[] = {
     0.5f,  0.5f, 0.0f,   // top right
     0.5f, -0.5f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f    // top left
};
unsigned int indices[] = {  // note that we
start from 0!
    0, 1, 3,   // first triangle
    1, 2, 3    // second triangle
};
```

Advanced features 1 - EBO (continue)

- The process is similar to VBO
- Replace `glDrawArrays` by `glDrawElements`



```
unsigned int EBO;  // the process is very similar to VBO
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);  // bind
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);  // copy
the data to the buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);  // take indices from EBO
// 2nd argument: number of elements to draw
// 4th argument: offset
```

Transformation 1 - definition

- In OpenGL, transformations are handled using matrices (4 * 4)
- Vertices in OpenGL can be represented as vectors (4 * 1)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

identity matrix and a vector *v* (*w*=1, explained later)

```
glm::vec4 Position = glm::vec4(glm::vec3(0.0), 1.0);
glm::mat4 Model = glm::mat4(1.0);
```

Transformation 2 - scaling

- Defined by 3 scaling variables (for x, y, z), *uniform* if they are the same else *non-uniform*
- Define the scaling matrix on any vector as:

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_1 * x \\ S_2 * y \\ S_3 * z \\ 1 \end{bmatrix}$$

Transformation 3 - translation

- Add another vector to the original
- Defined in the last column
- This can *NOT* be done using a 3 * 3 matrix

$$\begin{bmatrix} 1 & 0 & 0 & T_1 \\ 0 & 1 & 0 & T_2 \\ 0 & 0 & 1 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_1 \\ y + T_2 \\ z + T_3 \\ 1 \end{bmatrix}$$

Transformation 4 - rotation (continue)

- Rotation along x, y, z axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

X axis

$$\begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Y axis

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Z axis

Transformation 5 - combination

- The power of 4 * 4 matrix: combine scaling / rotation with translation
- Orders are important: read the matrices from right to left
- Suggestion: scaling -> rotation -> translation

$$\begin{bmatrix} sx + t_x \\ sy + t_y \\ sz + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Transformation 7 - GLM

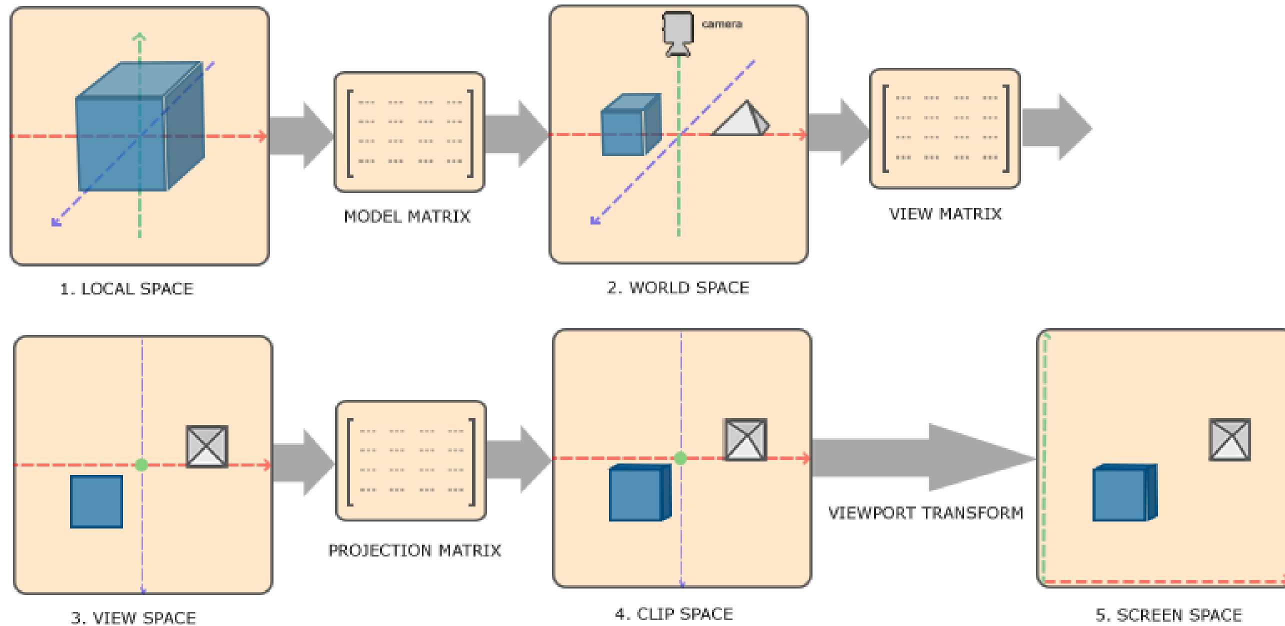
```
glm::mat4 trans = glm::mat4(1.0f);  // transformation matrix
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));  // around Z axis
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));

unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");  // pass the
"uniform" variable to the shader
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
// 1st argument: location of the "uniform"
// 2nd argument: how many matrices to send
// 3rd argument: whether transpose it
```

In shader:

```
layout (location = 0) in vec3 aPos;
uniform mat4 transform;  // receive the uniform

void main() {
    gl_Position = transform * vec4(aPos, 1.0f);
}
```

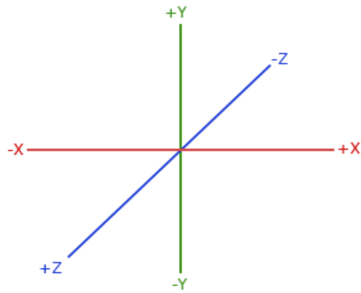
Transformation 8 - coordinate systems (covered later)

- Model space: local to an object
- World space: objects placed relative to each other
- View space: origin at the camera’s focal point
- Clip space: the volume the camera sees (near - large, far - small) (a *frustum*)
- Screen space: in pixels

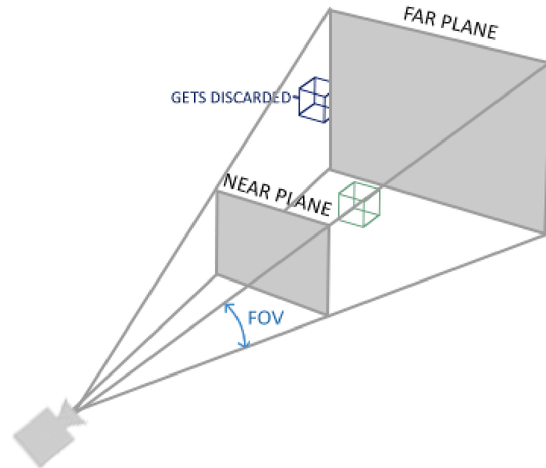


Transformation 8 - coordinate systems (covered later)

- Model space: local to an object
- World space: objects placed relative to each other
- View space: origin at the camera's focal point
- Clip space: the volume the camera sees (perspective: near - large, far - small) (a *frustum*)
- Screen space: in pixels



view space



clip space

Transformation 8 - coordinate systems (covered later)

Combination: $V_{clip} = M_{proj} * M_{view} * M_{model} * M_{local}$

Freeglut Quick Start 1 - window

Initialize with a window

```
void initGLUT(int* argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  // a single-buffered window
  glutInitWindowSize(SCR_WIDTH, SCR_HEIGHT);  // set window size
  glutCreateWindow("example window");
  glutDisplayFunc(display);  // set a callback function

  // set menu...
  // set callbacks...
}

void display (void)
{
  glClear(GL_COLOR_BUFFER_BIT); // clear window
  glDrawArrays(GL_TRIANGLES, 0, 3);
  glFlush();  // force execution of GL commands
  glutSwapBuffers();  // swap buffers if the window is double-buffered
}
```

Freeglut Quick Start 2 - callbacks

Put callback functions into the initialization
Note: put global declarations at the beginning of the source file

```
// declarations of callback functions
void display();
void keyRelease(unsigned char key, int x, int y);
void mouseBtn(int button, int state, int x, int y);
void mouseMove(int x, int y);

void initGLUT(int* argc, char** argv) {
  // init...
  // set menu...
  // set callbacks
  glutDisplayFunc(display);
  glutKeyboardUpFunc(keyRelease);
  glutMouseFunc(mouseBtn);
  glutMotionFunc(mouseMove);
}
```

Freeglut Quick Start 2 - callbacks - keypress

```
void keyRelease(unsigned char key, int x, int y) {
    switch (key) {
    case 27:  // Escape key
        menu(MENU_EXIT);
        break;
    }
}
```

Freeglut Quick Start 2 - callbacks - mouse move

```
void mouseMove(int x, int y) {  // x, y: mouse location in window relative coordinates
    ...
    glutPostRedisplay();  // update the window (redisplay)
}
```

Freeglut Quick Start 3 - menu

```
void menu(int cmd) {
    switch (cmd) {
    case MENU_VIEWMODE:
        ...
        glutPostRedisplay();  // redraw the screen
        break;

    case MENU_EXIT:
        glutLeaveMainLoop();
        break;
    }
}

void initGLUT(int* argc, char** argv) {
    // Create a menu
    glutCreateMenu(menu);
    glutAddMenuEntry("Toggle view mode", MENU_VIEWMODE);
    glutAddMenuEntry("Exit", MENU_EXIT);
    glutAttachMenu(GLUT_RIGHT_BUTTON);  // bind to the right button of the mouse
}
```