Lab 01 Basics

OpenGL Initialization

OpenGL is just a graphics library, it does not have functionality for creating windows, handling User I/O or implementing vector and matrix mathematics etc. For this reason we use 3rd party libraries.

We will be using the GLEW library[1]: *The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file. GLEW has been tested on a variety of operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris.*

```
// Include GLEW (always include first)
#include <GL/glew.h>
```

```
// Include GLFW to handle window and keyboard events
#include <glfw3.h>
```

Now we need to initialize GLFW.

```
// Initialize GLFW
if (!glfwInit())
{
    throw runtime_error("Failed to initialize GLFW\n");
}
```

Configure antialiasing and OpenGL version.

```
glfwWindowHint(GLFW_SAMPLES, 4);  // 4x antialiasing
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // We want OpenGL 3.3
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make MacOS happy
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); // We don't wa
nt the old OpenGL
```

Create window handle.

```
// Open a window and create its OpenGL context
window = glfwCreateWindow(W_WIDTH, W_HEIGHT, TITLE, NULL, NULL);
if (window == NULL)
{
    glfwTerminate();
    throw runtime_error(string(string("Failed to open GLFW window.") +
            " If you have an Intel GPU, they are not 3.3 compatible." +
            "Try the 2.1 version.\n"));
}
```

[1] http://www.glfw.org/docs/latest/index.html

```
glfwMakeContextCurrent(window);
```

Initialize GLEW.

```
// Initialize GLEW
if (glewInit() != GLEW_OK)
{
    glfwTerminate();
    throw runtime_error("Failed to initialize GLEW\n");
}
```

Enable keyboard events.

```
// Ensure we can capture the escape key being pressed below
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
```

Set background color. Color is normalized $[0,1]$.

```
// Set background color (gray) [r, g, b, a]
glClearColor(0.5f, 0.5f, 0.5f, 0.0f);
```

Create Context

OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate. The state of OpenGL is commonly referred to as the OpenGL . We communicate with the OpenGL by manipulating the variables and the data of buffers.

This is where we create the objects that are to be drawn on the screen. The process is as follows:

1. Load, compile and link the shader programs together. The shader programs are written in GL Shader Language (GLSL) and run directly on the graphics card. They perform basic numerical operations and are responsible for creating the different effects.
2. Load vertices, 3D object attributes, textures, color, etc. directly to the graphics card memory. There are some basic functions that control the transfer of information to the graphics memory.
3. Specify the interpretation the loaded data.

Let's draw a simple triangle. A triangle is defined by three points. When talking about "points" in 3D graphics, we usually use the word "vertex" ("vertices" in the plural). A vertex has 3 coordinates: X, Y and Z. You can think about these three coordinates in the following way:

- X in on your right
- Y is up
- Z is towards your back (yes, behind, not in front of you)

But here is a better way to visualize this: use the Right-Hand Rule

- X is your thumb
- Y is your index

- Z is your middle finger. If you put your thumb to the right and your index to the sky, it will point to your back, too.

Having the Z in this direction is weird, so why is it so? Short answer: because 100 years of Right Hand Rule Math will give you lots of useful tools. The only downside is an unintuitive Z.

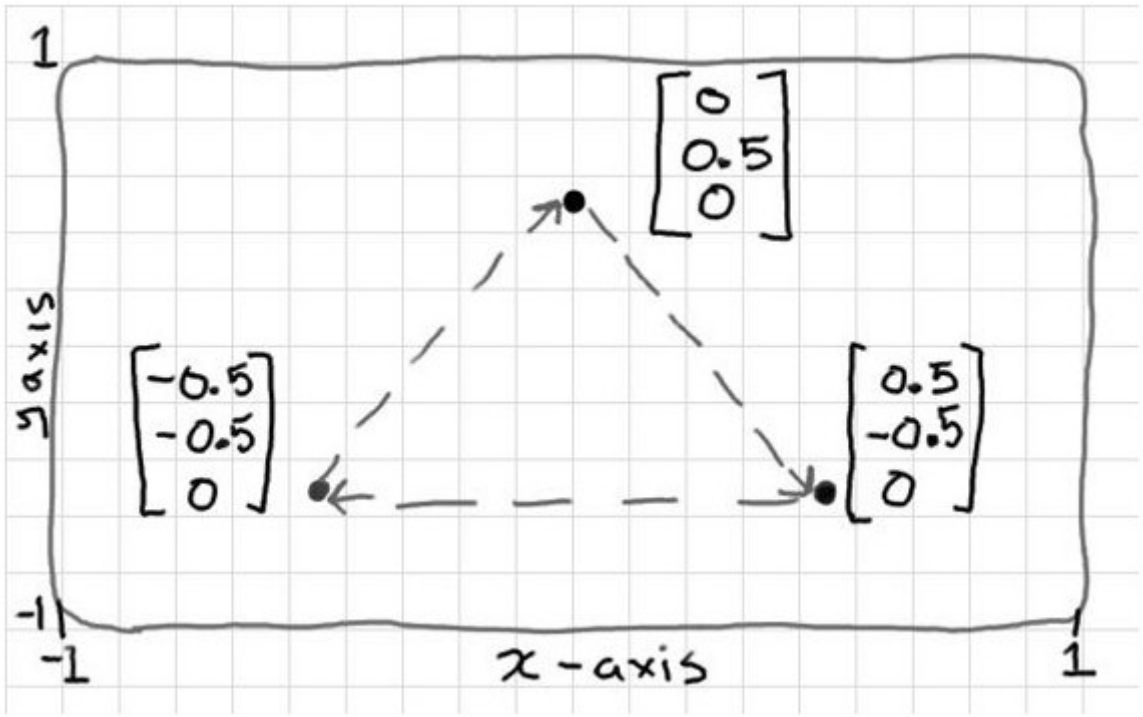


Figure 1 Coordinate System

Define the three vertices of the triangle.

```
/* geometry to use. these are 3 xyz points (9 floats total) to make a triangl
e */
const GLfloat vertices[] = {
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f
};
```

Create and bind a vertex array object (VAO). Since a shape can have many attributes (e.g., color, normal vectors, texture, etc.) we would like to identify it with a single pointer during the rendering step (imagine that we have many shapes in the scene). VAOs save us the hassle of having to redefine everything in the main render loop when we want to render a different object. Note that glBind* functions are used to set current working data.

```
/* The vertex array object (VAO) is a little descriptor that defines which
data from vertex buffer objects should be used as input variables to vertex
shaders. In our case - use our only VBO, and say 'every three floats is a
variable' */
glGenVertexArrays(1, &triangleVAO);
glBindVertexArray(triangleVAO);
```

Create the vertex buffer object (VBO) that will hold the vertex attributes (coordinates created previously). A shape can have many VBOs, each one defining some information about each vertex (like color or normals). The VBOs are transfered to the graphics memory and we define how this data is interpreted.

```
/* a vertex buffer object (VBO) is created here. this stores an array of
data on the graphics adapter's memory. in our case - the vertex points */
glGenBuffers(1, &verticesVBO);
glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(GLfloat), vertices, GL_STATIC_DRAW);
```

The vertex coordinates are the first attribute (“0”) of the shape. This attribute will be referenced in the **vertex shader** program as we will see later on. We need to specify the layout of the data for the verticesVBO.

```
// "attribute #0 is created from every 3 variables in the above buffer, of ty
pe
// float (i.e. make me vec3s)"
glVertexAttribPointer(
    0,                  // attribute 0 (must match the layout in the shader)
    3,                  // size: xyz
    GL_FLOAT,           // type
    GL_FALSE,           // normalized?
    0,                  // stride
    NULL                // array buffer offset
);
```

Every creation of a buffer, shader program and other resources that are related to the graphics card requires the de-allocation of the memory at the end of the program.

```
// Free allocated buffers
glDeleteBuffers(1, &verticesVBO);
glDeleteVertexArrays(1, &triangleVAO);
glDeleteProgram(shaderProgram);
```

Rendering (Main Loop)

The purpose of the main loop is to refresh the content (shapes), redraw them and listen for events (e.g., keyboard, mouse).

```
do
{
    // Clear the screen.
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw

    // Swap buffers
    glfwSwapBuffers(window);

    // Events
    glfwPollEvents();
} // Check if the ESC key was pressed or the window was closed
while (glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
        glfwWindowShouldClose(window) == 0);
```

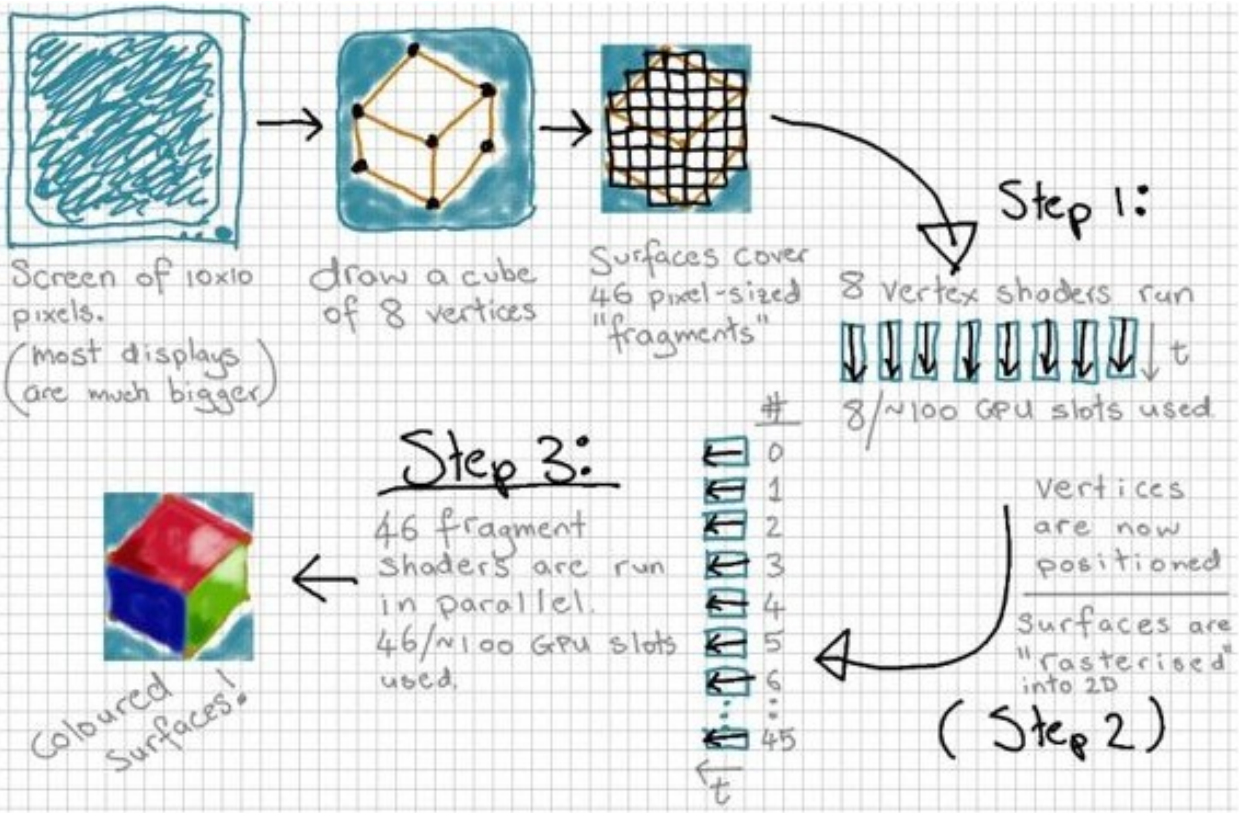
The swap buffers command uses two buffers, that is the first is used for rendering the current content and the second for drawing the new content. When the new content is ready the two buffers are swapped. The function glfwPollEvents is polling for events. The main loop terminates if we press the “ESC” key or if the window is closed (“X” button).

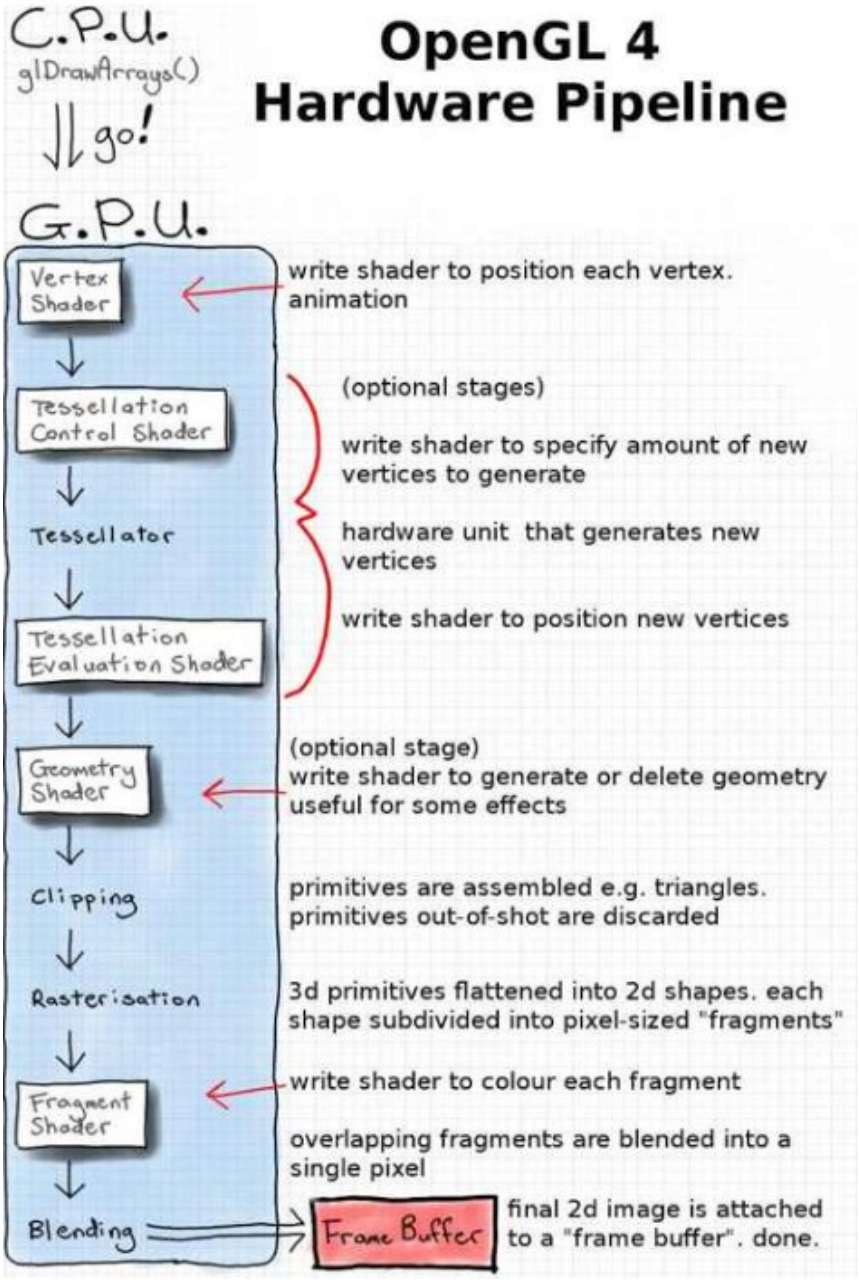
In order to draw something, we need 1) to bind the shader program, 2) bind the VAO of the shape, and 3) draw the shape.

```
// Bind shader program
glUseProgram(shaderProgram);

// Bind VAO
glBindVertexArray(triangleVAO);

//"attribute #0 and #1 should be enabled when this VAO is bound"
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

/* draw points 0-3 as triangles from the currently bound VAO with
current in-use shader*/
glDrawArrays(GL_TRIANGLES, 0, 3); // interpret as triangles read 3 data
```

Shaders

glDrawArrays invoices a specific pipeline that is followed by the graphics card in order to draw the shapes on the screen. In these labs sessions, we will mostly use the vertex and fragment shaders. The vertex shader is responsible for positioning each vertex. The fragment shader is responsible for assigning color to each fragment (something like pixel).


Screen of 10x10 pixels.
(most displays are much bigger)
draw a cube of 8 vertices
Surfaces cover 46 pixel-sized "fragments"
Step 1:
8 vertex shaders run
t
8/~100 GPU slots used
#
0
1
2
3
4
5
6
45
t
Vertices are now positioned
Surfaces are "rasterised" into 2D
(Step 2)
Step 3:
46 fragment shaders are run in parallel.
46/~100 GPU slots used.
Coloured Surfaces!



Let's define the two shader programs. We will be using GLSL 330 version of the language.

```
#version 330 core
```

The coordinate attribute is bound to "0" (see glVertexAttribPointer) and is interpreted as 3 floats. We can access the coordinates as follows:

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

Finally, assign the homogeneous position of the vertex (homogeneous coordinates are the native representation for 3D points in the graphics card).

```
void main()
{
    // assign vertex position
    gl_Position.xyz = vertexPosition_modelspace;
    gl_Position.w = 1.0;
    // or
    // gl_Position = vec4(vertexPosition_modelspace, 1.0);
}
```

gl_Position is a reserved variable.

The fragment shader assigns the red color to each vertex. Note that the output of the program is a vec3.

```
#version 330 core

// output data
out vec3 fragmentColor;

void main()
{
    fragmentColor = vec3(1, 0, 0);
}
```