Tutorial 1: Introduction to OpenGL Toolbox

Things to be covered in this tutorial:

- Overview of the OpenGL graphics rendering pipeline
- Introduction to framebuffer, buffer objects, OpenGL primitives, and shaders
- Implement a simple OpenGL application with keyboard events

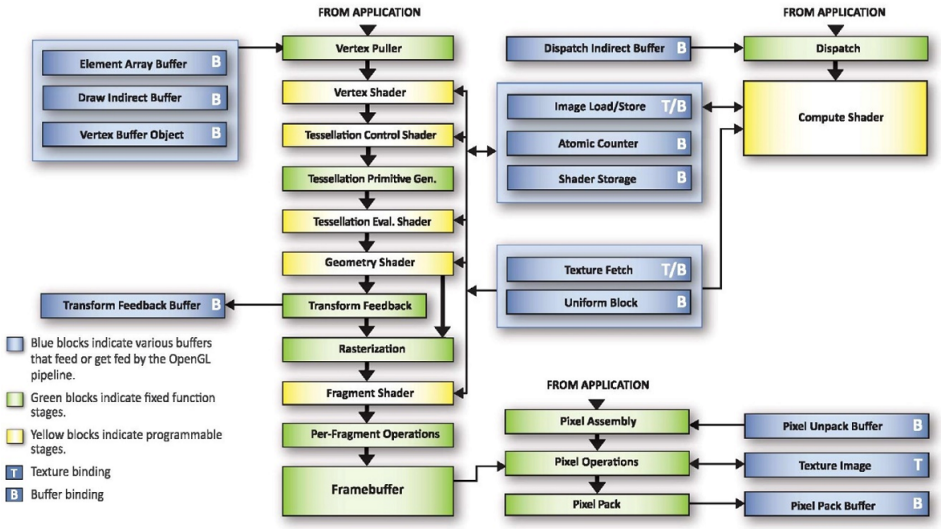
Prerequisites

- This tutorial will require completion of Tutorial 0.

Review of the OpenGL Graphics Pipeline

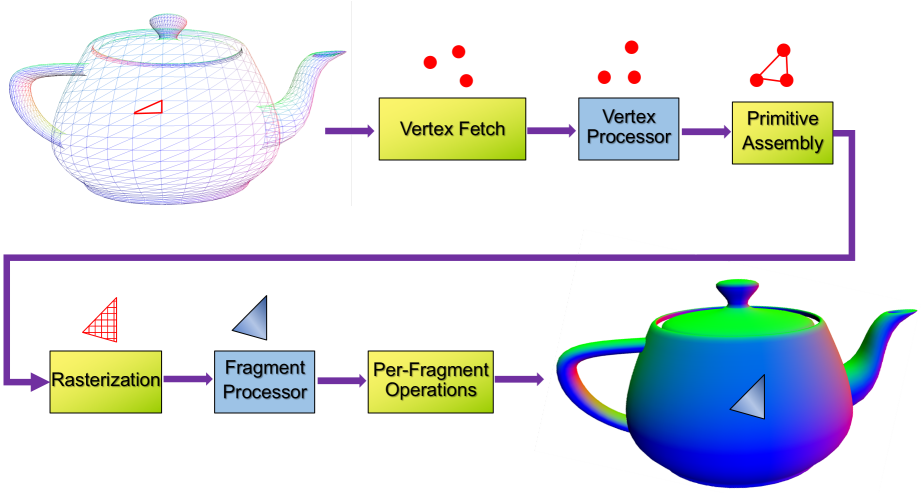
When you develop a graphics application with OpenGL API, you define geometry, viewing, projection, and a number of appearance properties. Objects' geometries are defined by their vertices, their normals, and their graphics primitives that encompass points, lines, or triangles. Viewing and projection are each defined with a specific matrix. Appearance is specified by defining color, shading, materials, and lighting, or texture mapping. For the hardware to generate correct images, this data must be associated with hardware commands. The process of going from an object's geometry to an on-screen image can be divided into a number of conceptual steps or stages. Orders issued to the hardware must have an explicit and well known order at every stage. Data and commands must follow a path and have to pass through some stages, and that path cannot be altered. This path is commonly called the *graphics rendering pipeline*. Think of it like a pipe where you insert some data into one end - vertices, textures, shaders - and they start to travel through some small machines that perform very precise and concrete operations on their input data to generate new data that is then transmitted to the next machine until the final machine generates output at the other end: the rendered image.

The following picture shows the official OpenGL version 4.6 graphics rendering pipeline that is nothing but a sequence of data processing stages where the output of one stage is used as the input to the subsequent stage:



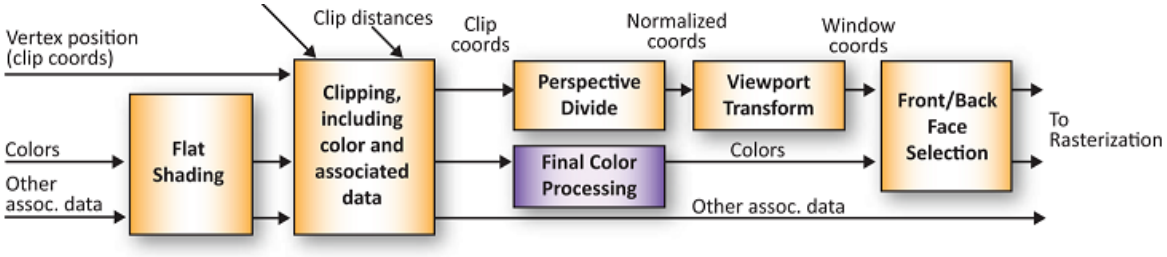
The OpenGL graphics pipeline accepts as input a geometrical representation of a 3D scene stored in GPU memory. The input data is processed by a variety of fixed-function and programmable processing stages into a final rendered image. To render a frame using OpenGL, the application configures the graphics pipeline and submits 3D models to be rendered. In general, each model will require a different pipeline configuration based on the model's specific geometric data and topology. 3D models can be represented using a variety of methods including polygonal surfaces, quadric surfaces, spline surfaces, procedural surfaces, physically based modeling methods for hair, fur, cloth, octree encoding, isosurface display, and volume rendering. Irrespective of the representation method, these models must be specified using OpenGL ***primitives*** that will get rendered as one of the three native types supported by all graphics hardware: ***points***, ***lines*** and ***triangles***. Line primitive types can be combined together to form ***strips*** and ***loops***; triangle primitive types can be combined together to form ***fans*** and ***strips***.

The main stages in the OpenGL pipeline of concern to us consist of ***vertex processing***, ***rasterization***, ***fragment processing***, and ***per-fragment operations***. The following picture uses the trip taken by a triangular polygon of a three-dimensional model to illustrate the main processes involved in the generation of the final rendered image.



The ***vertex processor*** is a programmable unit that is responsible for transforming incoming geometry into something suitable for consumption by subsequent stages. OpenGL requires geometric data to be presented to the GPU in the form of vertices. A vertex is a bundle of ***per-vertex data*** such as position coordinates, normals, tangents, colors, and texture coordinates. OpenGL requires vertices and the topology that connects vertices into primitives of points, lines, or triangles be stored in ***buffer objects***, which are chunks of memory managed by the GPU. The fixed-function ***vertex fetch*** stage supplies the vertex processor with vertices from buffer storage that specify a primitive such as a point, a line, or a triangle. The specific buffer memory regions and formats of per-vertex data stored in these buffers is determined by states set by specific OpenGL commands. Compilation units written in OpenGL Shading Language [GLSL] to be run on the vertex processor are called ***vertex shaders***. The vertex shader is a piece of code *executed once and only once for each vertex*. The vertex processor consists of many cores with each core executing one vertex shader that transforms one vertex, thereby allowing computation to be performed on several vertices in parallel, with the parallelization width dependent on the particular graphics hardware being used. Minimally, vertex shaders must transform position coordinates usually specified in *model coordinate system* to corresponding position coordinates in *clip coordinate system*. Vertex shaders can do more - they can transform and normalize normals, generate and transform texture coordinates, and implement lighting calculations at each vertex. Since vertex shaders are applied on individual vertices, they cannot perform operations such as clipping and culling that require knowledge of several vertices at once.

Using the primitive type specified by an OpenGL command [such as `glDrawArrays` or `glDrawElements`], the fixed-function ***primitive assembly*** stage organizes collections of vertices into a single, complete topological primitive that will then be operated on by fixed-function stages illustrated in the following picture:



1. ***clipping***: the primitive is clipped against the view frustum in clip coordinate system to eliminate portions that will be projected outside the *viewport*.
2. ***projection [perspective or orthographic] division***: the vertices of the clipped primitive are projected from clip coordinate system to NDC system.
3. ***viewport transform***: primitives in NDC system are transformed to window [or device or screen] coordinate system using the arguments specified in OpenGL command `glViewport`.
4. ***triangle culling***: triangles are discarded based on whether they're front-facing or back-facing with the interpretation controlled by calls to OpenGL commands `glFrontFace` and `glCullFace`.

Primitives that have not been clipped out nor culled because of their facing orientations are decomposed by fixed-function ***rasterizer*** stage into smaller units called ***fragments*** that correspond to pixels in the framebuffer. Primitives contain a set of attributes at each vertex - for example, the highlighted red triangle in this picture will have a set of attributes at each vertex with each set potentially containing texture coordinates, normals for lighting, colors from lighting calculations performed in the vertex shader, depth values computed during projection division, and any other data supplied by the application. The rasterizer will interpolate these per-vertex values across the primitive and assign the interpolated data to each fragment. After rasterization, fragments are processed by early per-fragment tests such as the pixel ownership test, scissor test, and if enabled the stencil, the depth buffer, and occlusion query sample counting tests are also performed. Think of each fragment generated by the rasterizer as a "may be" pixel because it is possible that the fragment may be rejected by these early per-fragment tests [and other actions performed by the fixed-function per-fragment operations stage] and may never update the corresponding pixel's color value [and if depth buffering is enabled, the corresponding depth buffer].

Surviving fragments are processed by the ***fragment processor*** which is a programmable unit that operates on fragment values. Compilation units written in GLSL that will run on the fragment processor are called ***fragment shaders***. The fragment shader is a piece of code *executed once and only once for each fragment* - the interpolated values contained in a fragment can be used in any computation such as plain color computations, texture mapping, per-pixel lighting, per-pixel fog, blending, and so on. The fragment processor has several cores with each core executing a fragment shader that operates on an individual fragment with the parallelization width depending on the particular graphics hardware being used. The minimum requirement for the fragment shader is to use the interpolated values at each fragment to update the corresponding location in the destination framebuffer's color buffer with a color value. The fragment shader may also modify or replace the interpolated depth values of fragments. Changes to a fragment's window position nor access to neighboring fragments is allowed.

After a fragment is modified by the programmable fragment processor to have a color and a depth value, it is then further modified, and possibly discarded by the fixed-function ***per-fragment operations*** stage. Operations performed in this stage are controlled by calls to OpenGL commands and include stencil test, depth buffer test, occlusion query test, blending, dithering, and logic operations. The stencil test, depth buffer test, occlusion query tests can be implemented as early per-fragment tests right after rasterization. Finally, if the fragment was not discarded, its computed color is used to update the destination framebuffer's color buffer and its depth value is used to update the destination framebuffer's depth buffer. The update locations within these buffers is specified by the fragment's window coordinates.

Introduction to tutorial

The purpose of this tutorial is to put together the necessary functionality that will render two triangles. These two triangles will take a trip through the OpenGL rendering pipeline [sort of] similar to the path taken by the highlighted red triangle in this picture.

OpenGL doesn't manage windows or handle input systems. Modern windowing systems that support OpenGL include an interface that provides the binding between an OpenGL context and the windowing system. Rather than dealing with these low-level operating system mechanisms, it is more convenient to use a cross-platform library that abstracts away operating system specific code to create and manage windows, create OpenGL contexts, and deal with user input in a platform independent manner. This course will use a popular toolkit called GLFW.

OpenGL is implemented in graphics drivers developed by GPU vendors. Programmers must download header file glcorearb.h to obtain interfaces to functions specified in core OpenGL specification and extensions and wglext.h for Windows-specific extension interfaces. Further, since OpenGL functions are implemented in graphics drivers, applications must call WGL function `wglGetProcAddress` to manually obtain pointers to OpenGL functions. Downloading appropriate header files and getting access to these functions through function pointers is tedious work for application programmers. This course will use a platform-independent library called GLEW to provide the necessary header files and expose the functionality of core OpenGL and extensions.

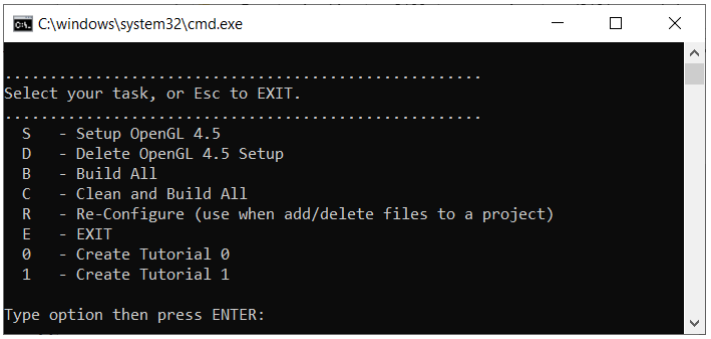
Since OpenGL interfaces, GLFW, and GLEW libraries are not class-based, their functions are non-polymorphic and don't have namespace associations. To prevent namespace corruption, name clashes, and to identify the parent library to which the function belongs, these C-based libraries use a simple naming convention. Every function name has a lowercase prefix:

- `gl`: Core OpenGL function.
- `glfw`: Function belonging to the windowing system interface GLFW.
- `glew`: Function belonging to the GL extension interface GLEW.

The rest of the function name indicates its purpose and follows the CamelCase convention. Detailed explanations about GLFW are available from GLFW's online documentation. Some of this document's code is borrowed from GLFW's "Getting Started" page.

🎯Task 1: Starter-Code

Overwrite the existing batch file csd2101.bat in csd2101-opengl-dev with a new version available on the assessment's web page. Executing the batch file generates the following menu:



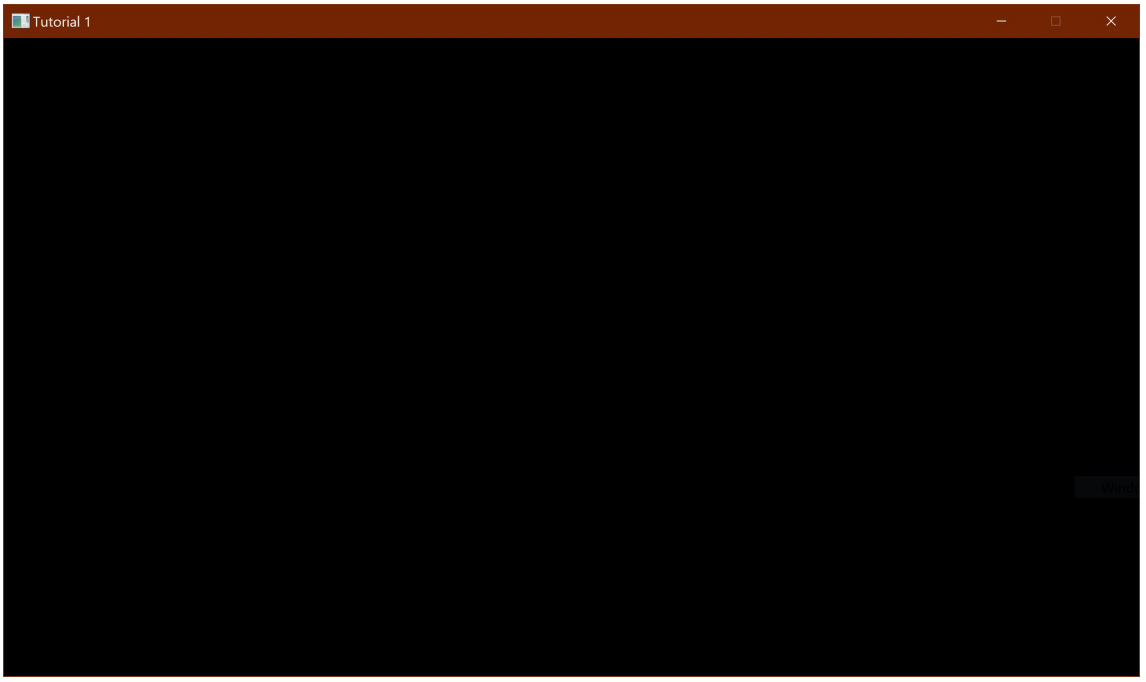
1. Choose option **1 - Create Tutorial 1** to create a directory projects/tutorial-1 whose layout is shown below:

```
csd2101-opengl-dev/       # Sandbox directory for all assessments
├─ build/                 # Build files will exist here
├─ projects/              # Tutorials and assignments must exist here
|  └─ tutorial-1          # Tutorial-1
|     └─ include          # Header files - *.hpp and *.h files
|     └─ src              # Source files - *.cpp and .c files
|        └─ my-tutorial-1.vert   # Vertex shader file
|        └─ my-tutorial-1.frag   # Fragment shader file
└─ csd2101.bat            # Automation Script
```

   Source and shader files main.cpp, glhelper.cpp, glslshader.cpp, my-tutorial-1.vert, and my-tutorial-1.frag are pulled into nested directory tutorial-1/src. Header files glhelper.h, glslshader.h, and glapp.h are pulled into nested directory /tutorial-1/include.

   This option also creates a Visual Studio 2022 project tutorial-1 in directory build which in turn will also update the Visual Studio 2022 solution file opengl-dev.sln [also in directory build] by adding a new project titled tutorial-1. If you're currently in Solution Explorer, a dialog box will be displayed requesting an update. Press Reload button to refresh the Solution Explorer with the updated solution file that includes project tutorial-1.

2. Right-click on project tutorial-1 in the Solution Explorer pane and select option Set as Startup Project. Building the project [using Ctrl + B] and then executing the application [using Ctrl+5] generates output resembling the following picture:



Opening a window: Contexts, profiles, framebuffers

`main()`: Entry point to OpenGL program

As seen from the listing of function `main` [in main.cpp], our OpenGL application consists of five distinct parts:

```
int main(int argc, char* argv[]) {
  // Part 1
  ParseArguments& args = ParseArguments::getInstance();
  if (!args.parseArguments(argc, argv)) return 0;

  // Part 2
  init(args.window_width, args.window_height, "Tutorial 1");

  // [!WARNING!] Do not alter/remove following line
  AUTOMATION_HOOK_RENDER(args); // Automation hook

  // Part 3
  while (!glfwWindowShouldClose(GLHelper::ptr_window)) {
    // Part 3a
    update();
    // Part 3b
    draw();
  }

  // Part 4
  cleanup();
}
```

Here's a high-level overview of these parts:

- Part 1 is a call to function `ParseArguments::parseArguments`, which parses the application's command line arguments. The use of command-line parameters was covered in Tutorial 0 and will not be repeated here.
- Part 2 is a call to function `init` that creates a display window, starts up an OpenGL context, and initializes graphical entities such as geometry buffer objects, shader programs, and texture objects that are used by the application to render images. Function `init` has the following definition [in main.cpp]:

```
static void init(GLint width, GLint height, std::string title) {
  // Part 2a
  if (!GLHelper::init(width, height, title)) {
    std::cout << "Unable to create OpenGL context" << std::endl;
    std::exit(EXIT_FAILURE);
  }

  // Part 2b
  //GLHelper::print_specs(); // to be uncommented by you ...

  // Part 2c
  GLApp::init();
}
```

The first two user-supplied arguments specify the dimensions of the display window [and the default framebuffer attached to this window]. Use values suitable for your specific hardware configuration. The third user-supplied argument specifies a string printed to the window title bar. Part 2a calls function `GLHelper::init` to do the work of creating a window and starting the OpenGL context. The purpose and intent of function `GLHelper::print_specs` is described in the next task. We'll look at function `GLApp::init` later in the document.

- Ignore the line that makes a call to function `AUTOMATION_HOOK_RENDER`. This is a sandbox hook that providing testing and debugging functionalities.
- Part 3 implements a rendering loop that uses function `update` to process user input and advance the application one step forward and then renders the updated scene in function `draw`.
- Part 4 uses function `cleanup` to return allocated resources back to the system.

`GLHelper::init()`: Start OpenGL context

Now, locate function `GLHelper::init` [in file glhelper.cpp]. The code fragments below replicate the code in function `GLHelper::init` in the order required to create a window and start up the OpenGL context. The first section consists of the following code:

```
1  bool GLHelper::init(GLint width, GLint height, std::string title) {
2    GLHelper::width = width;
3    GLHelper::height = height;
4    GLHelper::title = title;
5
6    // Part 1
7    if (!glfwInit()) {
8      std::cout << "GLFW init has failed - abort program!!!" << std::endl;
9      return false;
10   }
11
12   // In case a GLFW function fails, an error is reported to callback function
13   glfwSetErrorCallback(GLHelper::error_cb);
14
15   // more code..
16 }
```

Before using any GLFW function, GLFW's state must be initialized by calling function `glfwInit`. If the library was successfully initialized, the function will return `GLFW_TRUE`, otherwise `GLFW_FALSE`. Function `glfwSetErrorCallback` is a callback function that is called if a GLFW function fails. The callback function `GLHelper::error_cb` is authored [in glhelper.cpp] to print to standard output the reason for the error.

Three important concepts to understand about OpenGL are *contexts*, *profiles* and *versions*.

- The process of turning geometric primitives, images, and bitmaps into pixels on a display screen is controlled by a large number of state settings. Cumulatively, these state settings define the behavior of the OpenGL rendering pipeline and the way in which primitives are transformed into pixels. A ***context*** is an internal data structure that contains the internal state of OpenGL. Entities such as texture objects, vertex array objects, and buffer objects are stored in the context. The application must create at least one context and make it active in order to perform any rendering.

- OpenGL 3.0 introduced a deprecation model in which certain features are marked ***deprecated***. Deprecated features are expected to be completely removed from a future version of OpenGL. To aid developers in writing applications for future versions, it is possible to create an OpenGL context which doesn't support deprecated features. Such a context is called a ***forward compatible context***, while a context supporting all OpenGL features is called a ***full context***.
- ***Profiles*** define subsets of OpenGL functionality targeted to specific application domains. The ***core profile*** of OpenGL defines essential functionality for the modern programmable shading model introduced in OpenGL 2.0 but doesn't include features marked as deprecated for that version of the spec. The ***compatibility profile*** doesn't remove the functionality marked as deprecated. Each profile has versions indicated by two numbers, the ***major*** and ***minor version numbers***. For example, in OpenGL 4.5, the major and minor version numbers are 4 and 5, respectively. Specifications sharing the same major version number are backward-compatible. An application written for OpenGL 3.2 specification will compile, link, and run unmodified using an OpenGL 3.3 implementation. However, the same application may not run on an OpenGL 4.0 implementation.

Before an OpenGL context is created, specific window and context settings can be provided as "hints" to GLFW. Context settings include core or compatible OpenGL profiles, forward compatibility, specific OpenGL versions, single- or double-buffered framebuffer, framebuffer format and whether a depth or stencil buffer is required, and so on. In the code below, we specify that we want to use OpenGL version 4.5 with core profile that is forward compatible.

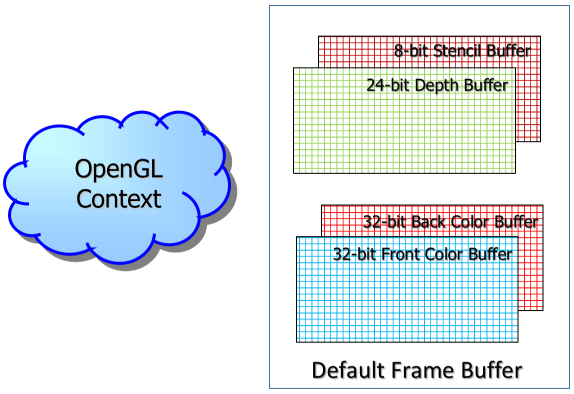
```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

The only physical manifestation of OpenGL is the ***default framebuffer***, which is used for the image that appears on the display screen. The default framebuffer will require at least a ***color buffer*** to hold color data for the image to be shown on the display screen, a ***depth buffer*** to hold depth values at pixels, and a ***stencil buffer***. Depth values in the depth buffer are used by a hidden surface removal algorithm called the ***depth buffer algorithm*** to generate correct images. The default framebuffer is created as part of the OpenGL context creation process with properties specified by the developer. Once created, the default framebuffer cannot be changed. However, to implement advanced rendering algorithms, the OpenGL application can create off-screen buffers called *framebuffer objects* [not discussed in this document] whose properties are controlled by the application.

In the code below, we ask for a double-buffered [explained later] framebuffer with a 32-bit RGBA color buffer and a 24-bit depth buffer.

```
glfwWindowHint(GLFW_DOUBLEBUFFER, GLFW_TRUE);
glfwWindowHint(GLFW_DEPTH_BITS, 24);
glfwWindowHint(GLFW_RED_BITS, 8); glfwWindowHint(GLFW_GREEN_BITS, 8);
glfwWindowHint(GLFW_BLUE_BITS, 8); glfwWindowHint(GLFW_ALPHA_BITS, 8);
```

The following picture conceptualizes the default framebuffer:



The window and OpenGL context are created at the same time using function `glfwCreateWindow` which returns a handle to the window that encapsulates the OpenGL context and default framebuffer. If the window or OpenGL context creation fails, a null pointer is returned, so it is necessary to check the return value.

```
GLHelper::ptr_window = glfwCreateWindow(width, height,
                                        title.c_str(), NULL, NULL);
if (!GLHelper::ptr_window) {
  std::cerr << "GLFW unable to create OpenGL context - abort program\n";
  glfwTerminate();
  return false;
}
```

OpenGL functions will affect the state of whichever context is *current*. Since we want OpenGL functions to impact state changes on the context that was just created, we make it current:

```
glfwMakeContextCurrent(GLHelper::ptr_window);
```

The next line is a call to function `setup_event_callbacks` that consolidates event handling functionality for the application:

```
setup_event_callbacks();
```

The definition of function `setup_event_callbacks` looks like this:

```
void GLHelper::setup_event_callbacks() {
  // [!WARNING!] Do not alter/remove following line
  AUTOMATION_HOOK_EVENTS(); // Automation hook

  glfwSetFramebufferSizeCallback(GLHelper::ptr_window, GLHelper::fbsize_cb);
  glfwSetKeyCallback(GLHelper::ptr_window, GLHelper::key_cb);
  glfwSetMouseButtonCallback(GLHelper::ptr_window, GLHelper::mousebutton_cb);
  glfwSetCursorPosCallback(GLHelper::ptr_window, GLHelper::mousepos_cb);
  glfwSetScrollCallback(GLHelper::ptr_window, GLHelper::mousescroll_cb);
}
```

The first statement is a call to function `AUTOMATION_HOOK_EVENTS` that provides internal testing and debugging functionalities and can be safely ignored by students. In the next set of statements, we register callback functions to be invoked by GLFW for specific input events such as mouse cursor movement, mouse buttons being clicked, and window size changes. The first argument of

each of the `glfwSet*` functions represents a handle to the window while the second argument specifies the particular callback function that is being registered.

GLFW function `glfwSetInputMode` allows you to configure input modes for a specific window and we use the default setting:

```
glfwSetInputMode(GLHelper::ptr_window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
```

Recall that core OpenGL functions and extensions are implemented in graphics drivers. After the OpenGL context is created, GLEW obtains entry points to these functions and initializes function pointers to implementations of both core OpenGL and extensions.

```
// Part 2: Initialize entry points to OpenGL functions and extensions
GLenum err = glewInit();
if (GLEW_OK != err) {
  std::cerr << "Unable to initialize GLEW - error: "
            << glewGetErrorString(err) << " abort program" << std::endl;
  return false;
}
if (GLEW_VERSION_4_5) {
  std::cout << "Using glew version: "
            << glewGetString(GLEW_VERSION) << std::endl;
  std::cout << "Driver supports OpenGL 4.5\n" << std::endl;
} else {
  std::cerr << "Warning: The driver may lack full compatibility "
            << "with OpenGL 4.5, potentially limiting access to "
            << "advanced features." << std::endl;
}
```

🎯Task 2: Accessing OpenGL context information

OpenGL command `glGetString` returns information from the graphics driver about the capabilities of the graphics hardware. For example, the following code fragment:

```
GLubyte const *str_ven = glGetString(GL_VENDOR);
std::cout << "Vendor: " << str_ven << std::endl;
```

prints NVIDIA Corporation if the machine has a NVIDIA GPU.

OpenGL commands abbreviated as `glGet` can be used to obtain values for simple state variables. For example, the following code fragment will print the largest texture map dimension that can be handled by the current OpenGL context:

```
GLint tex_size;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &tex_size);
std::cout << "Maximum texture size: " << tex_size << std::endl;
```

The above code fragment prints Maximum texture size: 32768 on my machine.

Begin by uncommenting the call to `GLHelper::print_specs` in function `init` [defined in main.cpp]. Next, declare and define a static member function `GLHelper::print_specs` to print useful information about your OpenGL context and graphics hardware to standard output:

1. Use `glGetString` to print name of vendor [shown above], name of OpenGL renderer, version number of OpenGL graphics driver, and version number of OpenGL shading language.
2. Use `glGetIntegerv` to print major number and minor number of OpenGL API supported by the current context, whether the current context is double-buffered or not, the maximum number of vertex array vertices, maximum number of vertex array indices, the largest texture that OpenGL can handle, and the maximum supported viewport width and height.

Output similar to the following text should now be printed to the console:

```
GPU Vendor: NVIDIA Corporation
GL Renderer: GeForce GTX 1050/PCIe/SSE2
GL Version: 4.5.0 NVIDIA 417.35
GL Shader Version: 4.50 NVIDIA
GL Major Version: 4
GL Minor Version: 5
Current OpenGL Context is double-buffered
Maximum Vertex Count: 1048576
Maximum Indices Count: 1048576
GL Maximum texture size: 32768
Maximum Viewport Dimensions: 32768 x 32768
Maximum generic vertex attributes: 16
Maximum vertex buffer bindings: 16

```

Understand double buffering

The term *framebuffer* traditionally refers to a chunk of memory that holds the color data for the image displayed on a computer screen. The memory is composed of a rectangular array of pixels having the same dimensions as the display screen with each pixel capable of displaying a color at the corresponding position on the display screen. Each pixel element is represented by several bytes of memory - standard implementations use a byte for each of the red, green, and blue components. In OpenGL, the term framebuffer is used in a broader sense to refer to a collection of color buffers, a depth buffer, and a stencil buffer. A color buffer - as implied by the name - holds color information. We've seen that during initialization, GLFW creates a window and attaches a default framebuffer to the window. The color buffer attached to the window holds the color information that's to be displayed to the display screen.

In a simple graphics system, there may be only a single color buffer, into which new graphics is drawn at the same time as the display device is refreshed from it. This ***single buffering*** is simple and requires minimal graphics memory. However, even if graphics rendering happens very fast, the rendering and display refresh rates are usually not synchronized with each other. This lack of synchronization causes images to be displayed with major artifacts.

Double buffering is a common programming idiom that is used to solve a variety of problems. In computer graphics, double buffering avoids tearing by rendering into a *back buffer* and notifying the system when the frame is completed. While the back buffer is being rendered into, the video controller [a unit of the graphics hardware] will use the contents of the *front buffer* to display the previous frame's rendered image on the display device. A platform-specific buffer swapping command is called to make the rendered image become visible. This effectively transfers the back buffer data into the front buffer. In a *true swap*, the back buffer becomes the front buffer and vice-versa by swapping pointers to the buffers. If you were to read from the back buffer after a true

swap, it would hold the *previous* contents of the front buffer. OpenGL does not *require* true swapping. All that is required is that the contents of the back buffer find their way into the front buffer. This could be via a true swap, or it could be via a copy from the back buffer into the front. The default framebuffer almost always has a double-buffered color buffer.

`GLApp::init()`: Set OpenGL state

So far, the initialization process has created a window, started up an OpenGL context, and initialized function pointers to core OpenGL functions and extensions. The next step is to initialize the OpenGL rendering pipeline to a useful state in function `GLApp::init` [in source file glapp.cpp] by adding the following lines:

```
1  void GLApp::init() {
2    // Part 1: clear color buffer with RGBA value in glClearColor ...
3    glClearColor(0.f, 1.f, 0.f, 1.f);
4
5    // Part 2: use entire window as viewport ...
6    glViewport(0, 0, GLHelper::width, GLHelper::height);
7  }
```

Most applications including games fill the back buffer with an image representing the background of the scene being rendered. In our application, we fill the back buffer with an RGBA color value specified by OpenGL command `glClearColor`. In Part 1, we're specifying the back buffer fill color to be green 🟢.

In Part 2, we use OpenGL command `glViewport` to specify the entire window be used to display rendered images. Think of the ***viewport*** as the rectangular region within the window where the application will display what is viewed by a camera in the scene.

ℹ️ Adding the above code to `GLApp::init` will not affect the picture - it should still have a black background. Read on to understand why the window is not filled with a green 🟢 color.

Rendering loop: Life cycle of a frame

The rendering loop in function `main` [defined in main.cpp] facilitates rendering of new frames one after another by repeatedly calling functions `update` and `draw` [both defined in main.cpp]:

```
1  // Part 3
2  while (!glfwWindowShouldClose(GLHelper::ptr_window)) {
3    // Part 3a
4    update();
5    // Part 3b
6    draw();
7  }
```

The loop continues to iterate until the user decides to close the window using the Esc key, Close button, or using keyboard shortcut ALT+F4 in which case GLFW function `glfwWindowShouldClose` returns true and the loop terminates.

At the beginning of each frame, function `update` calls GLFW function `glfwPollEvents` to process input events in the event queue by invoking the callbacks associated with these events. The interactive application advances one step by updating objects in the application using the application's update function `GLApp::update` [defined in glapp.cpp]. At present, objects have

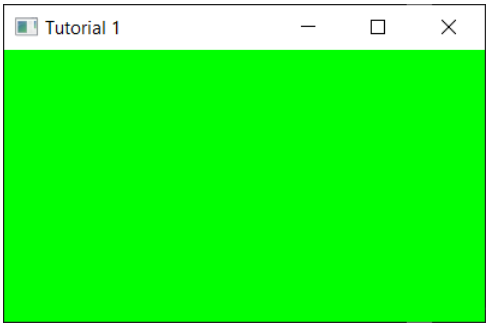
neither been defined nor initialized. Therefore the function doesn't do anything - we'll speak more about this matter later in this document.

```
static void update() {
  // Part 1
  glfwPollEvents();

  // Part 2:
  GLHelper::update_time(1.0); // update fps

  // Part 3
  GLApp::update();
}

static void draw() {
  // Part 1
  GLApp::draw();

  // Part 2: swap buffers: front <-> back
  glfwSwapBuffers(GLHelper::ptr_window);
}
```

The first objective of function `draw` is to invoke the application's rendering function `GLApp::draw` [defined in glapp.cpp] to render objects one after another into the back buffer. Before rendering, the application must clear the color back buffer, depth buffer, and other buffers [if used] in the default framebuffer. For example, the back buffer is generally cleared to a background color or image specific to the application. The depth buffer would be cleared to the maximum allowable depth in the scene for the depth buffer algorithm to be correctly implemented. OpenGL command `glClear` is used to inform the GPU that the back buffer must be filled with the color previously set by a call to `glClearColor` in `GLApp::init`. Although the previously created default framebuffer has a depth buffer, it is not required in this tutorial and will therefore not be cleared. Your version of function `GLApp::draw` must now look like this:

```
void GLApp::draw() {
  glClear(GL_COLOR_BUFFER_BIT);
}
```

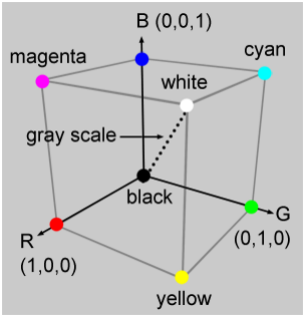
After `GLApp::draw` renders all necessary objects to the back buffer, function `draw` calls `glfwSwapBuffers` to display this newly rendered image. This GLFW function swaps the front and back buffers; the video controller sends the newly created image ([in the now front buffer] to the display device while the next ensuing frame will use the previous front buffer and now the current back buffer to render a new image into. At this point, your window should be displayed with a green 🟢 background.



Representing colors

Light is electromagnetic radiation of any wavelength with the visible range of a typical human eye between 400 and 700 nanometers. Each individual wavelength within the spectrum of visible light wavelength is representative of a particular color ranging from violet at 400 nanometers to blue at 470 nanometers to green at 550 nanometers to red at 700 nanometers where one nanometer is 10^{-9} meter. Color, on the other hand, is more of a perception in human minds than a part of objective reality. The inside surface of the eye called the retina contains sensors called cones. When light of a specific wavelength is incident on the retina, photochemical reactions occur in cones generating electrical impulses that are transmitted along optic nerves to the brain which then psychologically perceives these electrical impulses to represent a certain color. There are three kinds of cones labeled red, green, and blue because of their sensitivity to ranges of wavelengths of light associated with red, green, and blue colors, respectively. When light of a particular wavelength, say somewhere in the range between 560 to 590 nanometers strikes the retina, the red and green cones would transmit electrical pulses to the brain whose nerves would psychologically perceive the color to be yellow. A similar process occurs when all the wavelengths of the visible light spectrum strike the retina at the same time and white color is perceived. The sensation of white is not the result of a single color of light. Rather, the sensation of white is the result of a combination of all the colors of the visible light spectrum. In a similar manner, the absence of wavelengths of visible light spectrum is perceived as black.

Mimicking the behavior of red, green, and blue cones, one can use three *primary* colors [in computer graphics **R**ed, **G**reen, and **B**lue], the combinations of which create most colors that humans are capable of seeing. Absence of R, G, and B is black, adding R and G together produces yellow, G and B produce cyan, and R and B produce magenta. Adding equal amounts of R, G, and B produces gray shades ranging from black to white.



In OpenGL, light is represented as a $3-$tuple of arbitrary numbers denoting the amount of red, green, and blue light, each of which is clamped to range $[0, 1]$ before being stored in the color buffer. 1.0 means the maximum amount of light that can be displayed on a traditional display, and RGB $3-$tuple $(1.0, 1, 0, 1.0)$ indicates white light, $(1.0, 0.0, 0.0)$ provides bright red, and $(0.0, 0.0, 0.3)$ corresponds to dark blue. If $8-$bit integers are used to encode color components, 0 maps to 0.0 and 255 maps to 1.0.

In addition to color channels, OpenGL defines an additional alpha channel. The alpha channel doesn't have any inherent meaning or interpretation, but is usually used to perform blending operations between two different colors [for example, a foreground and a background color] or to encode the level of transparency on a material or surface. It is important to realize that pixels are not transparent, this is simply a convenient illusion that is accomplished by blending colors. Alpha values range from 0 [meaning completely transparent with 0% opacity] to 1.0 [meaning completely opaque with 100% opacity].

🎯Task 3: Clearing color buffer

In function `glApp::update` [in source file glapp.cpp], use a key event U to toggle the color for clearing the back color buffer between a static color and a dynamically computed color.

To handle key event U, declare a static variable `GLHelper::keystateU` of type `GLboolean` and initialize it with a default value `GL_FALSE`. Use callback function `GLHelper::key_cb` to toggle the value of `keystateU` when key U is pressed.

When the application begins execution, RGB values of the static background color are used to specify the arguments of `glClearColor`. Although the background color was set to green in Task 2, you can use any color you wish as the static background color. The sample uses red 🔴 color as the static background color.

Now, when key U is pressed, the background color must be dynamically computed by interpolating between the static color and a second color [although the sample uses blue color, you can choose any color you wish]. You can use `glfwGetTime` in conjunction with a periodic function to transition from the static color to the second color and back to the static color. Or, you can use some other method that you invent. The only requirement is that the computed background color must smoothly transition between these two colors and the resulting image must be pleasing to the eye. Abrupt transitions between color is not pleasing and should be avoided!!!

Pressing key U again must toggle the background color back to the static color.

🎯Task 4: Printing to Window's Title Bar

The sample prints the assignment's name followed by the author's name followed by the frames per second count with each field separated by a vertical bar:

You must do the same except that the author's name is replaced with your name and you must retain the sample's format so that information is legible. You'll have to use a GLFW function [that can be easily researched] to perform this activity. The few lines of code to complete this task must be inserted in function `GLApp::update` [in glapp.cpp]. Don't change the text in function `init` [defined in main.cpp] to complete this task since you'll not be submitting main.cpp.

🎯Task 5: Rendering triangles

To render a frame using OpenGL, the application configures the graphics pipeline and submits objects to be rendered. In general, each object will require a different pipeline configuration based on the object's specific geometric data, textures, and topology. Geometric data describes vertex positions, colors, and texture coordinates and topology information. This geometric data must be cached in ***buffer objects*** on the graphics server [that is, the GPU] to avoid the high costs of transferring geometry from client memory to server memory. OpenGL uses a container object called the ***vertex array object*** to encapsulate the format, description, and state of geometric data so that a single function call can present this data to the vertex shader. Our goal in the remainder of this document is to describe the programming constructs used in modern OpenGL programs to specify an object's geometry and topology to the vertex shader.

OpenGL primitives

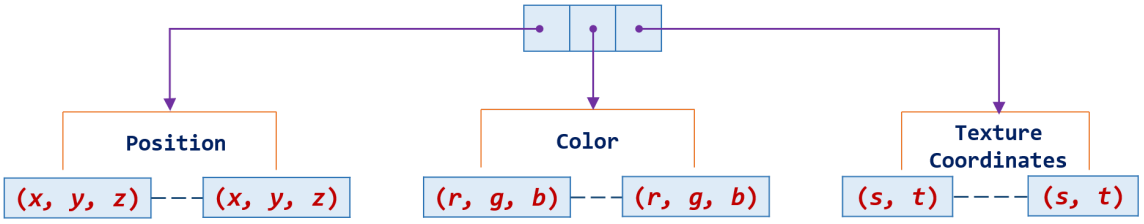
3D models are represented using boundary representation schemes, parametric spline surfaces, subdivision surfaces and boundary representation schemes. Although OpenGL supports rendering of complex geometric objects, it doesn't provide mechanisms to describe the complex objects themselves. Instead, 3D models - irrespective of their representation method in a modeling package - must be specified using one of the three native OpenGL types supported by vertex shaders [we'll ignore tessellation and geometry shaders in this document]: ***points***, ***lines*** and ***triangles***. Line primitive types can be combined together to form strips and loops; triangle primitive types can be combined together to form fans and strips. The following table maps primitive types to tokens representing them in OpenGL [for the sake of completeness, the primitives consumed by tessellation and geometry shaders are also included]:

Primitive Type	OpenGL Token
Individual vertices	`GL_POINTS`
Pair of vertices interpreted as line segments	`GL_LINES`
Series of interconnected line segments	`GL_LINE_STRIP`
Series of interconnected line segments with line segment between first and last vertices	`GL_LINE_LOOP`
Three vertices interconnected as triangles	`GL_TRIANGLES`
Series of interconnected triangles	`GL_TRIANGLE_STRIPS`
Series of triangles with common pivot	`GL_TRIANGLE_FANS`
For use by tessellation shaders: Pre-specified set of vertices processed by vertex shader followed by tessellation control shader	`GL_PATCH`
Only used by geometry shader: Send sequence of four-vertex primitives to shader with middle two vertices interpreted as line segment and first and last vertices providing adjacency information	`GL_LINES_ADJACENCY`
Only used by geometry shader: Same as before except that entire strip is considered primitive with one additional vertex at each end	`GL_LINE_STRIP_ADJACENCY`
Only used by geometry shader: Send sequence of six-vertex primitives to shader with first, third, and fifth vertices making up real triangle while second, fourth, and sixth vertices are considered in-between triangle's vertices	`GL_TRIANGLES_ADJACENCY`
Only used by geometry shader: Send sequence of vertices representing triangle strip with every other vertex [first, third, fifth, ...] forming the strip and in between vertices are considered adjacent vertices	`GL_TRIANGLE_STRIP_ADJACENCY`

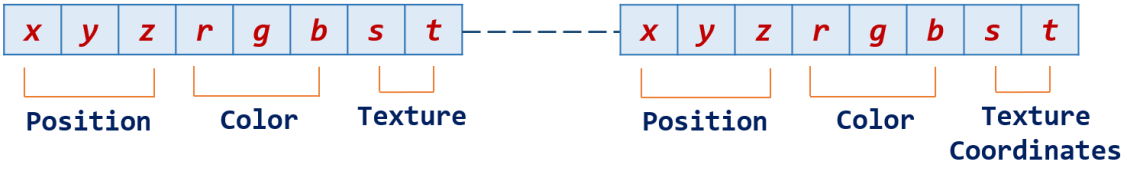
Memory layout for vertex data

The data associated with each vertex in the geometry consists of position coordinates plus a variety of other information such as normal vectors, color components, and texture coordinates. Consider a geometric model that is to be rendered to screen. In addition to position data, suppose each vertex also has color and texture data. What are the different possibilities to store this geometric information in server-side [or GPU] memory? Two methods are commonly used for allocating and storing per-vertex data in arrays: ***structure of arrays*** and ***array of structures***.

- ***Structure of arrays*** [SOA] represents a memory layout of geometric data in which per-vertex data is stored in individual arrays with a structure encapsulating these array addresses. Position data is laid out in memory as an array with each element representing position coordinates (x, y, z). Similarly, color data is laid out in an array with each element representing color coordinates (r, g, b). Texture coordinate data is stored as an array with each element representing texture coordinates (s, t). As shown in the following picture , a structure encapsulates the addresses of these three arrays.



- ***Array of structures*** [AOS] represents a memory layout in which geometric data is laid out in an array with each array element encapsulating per-vertex data. Suppose there is a structure with the first data member representing the position data (x, y, z), the second data member representing color data (r, g, b), with a final third data member representing texture coordinates (s, t). The following picture illustrates an array of elements with each array element representing the previously defined structure.



In OpenGL literature, AOS memory layout is said to be *interleaved* across each structure, while SOA memory layout is said to be *non-interleaved*. All graphics APIs and hardware allow both memory layouts. So, which memory layout provides optimal performance in terms of GPU performance? With AOS, each cache line is populated with a position (x, y, z), a color (r, g, b), and texture coordinates (s, t). However, if only position coordinates are required [say, because the object's shadow map is being rendered], AOS will not be efficient. In this situation, SOA will provide better bandwidth and cache utilization because each cache line can be populated with only position coordinates. In addition to shadow map generation, SOA is again preferred for 3D geometry viewers where per-vertex data may not be uniformly present for all objects and even if certain per-vertex data are present, it is up to the user to determine which per-vertex data is to be used. In general, an SOA should be optimal compared to AOS. However, you must always profile your applications to identify the most optimal memory layout.

SOA memory layout for rectangle model

Let's begin our discussion on buffer objects, vertex array objects [VAOs], and OpenGL draw commands by amending glapp.h and glapp.cpp. Append the following to glapp.h:

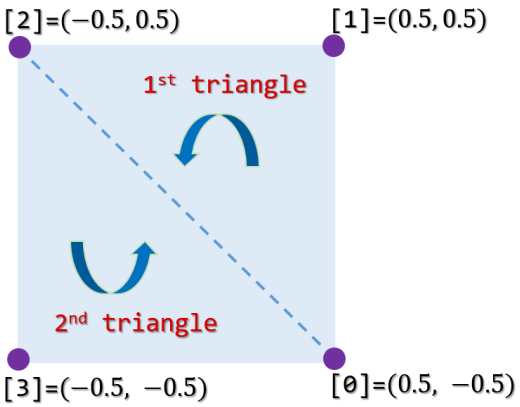
```
struct GLApp {
  // previous existing declarations

  // encapsulates state required to render a geometrical model
  struct GLModel {
    GLenum     primitive_type; // which OpenGL primitive to be rendered?
    GLSLShader shdr_pgm;       // which shader program?
    GLuint     vaoid;          // handle to VAO
    GLuint     vbo_hdl;        // handle to VBO
    GLuint     idx_elem_cnt;   // how many elements of primitive_type
                               // are to be rendered?

    // member functions defined in glapp.cpp
    void setup_vao();
    void setup_shdrpgm();
    void draw();
  };

  // data member to represent geometric model to be rendered
  // C++ requires this object to have a definition in glapp.cpp!!!
  static GLModel mdl;
};
```

`GLSLShader` is the type encapsulating a shader program and is defined in interface file glslshader.h with implementation in glslshader.cpp.

Next, we'd like to define member function `GLModel::setup_vao` [in source file glapp.cpp] that will set up a rectangle model using a SOA memory layout. We'll define the rectangle model directly in Normalized Device Coordinates [NDC] to avoid speaking of and using matrices in this tutorial. The viewport [which is the entire display window for this tutorial] is mapped to a NDC box with dimensions 2×2 with the viewport center, bottom-left corner, and top-right corner having NDC coordinates $(0, 0)$, $(-1, -1)$, $(1, 1)$, respectively.

> ***Carefully read the handout on Viewport Transform to understand the reasons behind the existence of something called NDC in a graphics pipe.***

We define a rectangle model that has one-fourth the area of the viewport with vertices and topology described in the following picture:



Begin the definition of function `GLModel::setup_vao` by defining the geometry of the rectangle model described in the above picture using a SOA memory layout:

```
std::array<glm::vec2, 4> pos_vtx { // vertex position attributes
  glm::vec2(0.5f, -0.5f), glm::vec2(0.5f, 0.5f),
  glm::vec2(-0.5f, 0.5f), glm::vec2(-0.5f, -0.5f)
};

std::array<glm::vec3, 4> clr_vtx { // vertex color attributes
  glm::vec3(1.f, 0.f, 0.f), glm::vec3(0.f, 1.f, 0.f),
  glm::vec3(0.f, 0.f, 1.f), glm::vec3(1.f, 1.f, 1.f)
};
```

Type `glm::vec2` is declared in GLM and is capable of holding two values of type `float`. You'll have to include the appropriate header file to make these types be known to the compiler. The geometry and topology of the rectangle model is defined in client-side memory. Next, this data must be transferred once from the client-side to server-side memory.

Vertex Buffer Objects

Buffer objects contain a data store holding a fixed-sized allocation of server-side GPU memory, and provide a mechanism to allocate, initialize, read from, and write to such memory. Under certain circumstances, the data store of a buffer object may be shared between the client and server and accessed simultaneously by both. Although, the OpenGL server requires many types of data supplied by the client, we'll take a closer look only at buffer objects whose data stores will hold per-vertex data.

Allocating buffer objects

In the most simplest form, an OpenGL *object* is a collection of multiple and related OpenGL state that is a subset of the OpenGL context. As is common with external libraries and APIs, the data structures representing these OpenGL objects are opaque and inaccessible to developers. Instead, developers must use OpenGL commands along with integer-based handles to create, modify, query, and destroy objects.

OpenGL uses many types of data supplied by the client. Buffer objects contain a data store holding a fixed-sized allocation of server memory, and provide a mechanism to allocate, initialize, read from, and write to such memory. A buffer object can be created with a call to `glCreateBuffers`:

```
glCreateBuffers(1, &vbo_hdl); // vbo_hdl is data member of GLApp::GLModel
```

The newly created buffer object is initialized to an unspecified target.

Allocating and filling data store

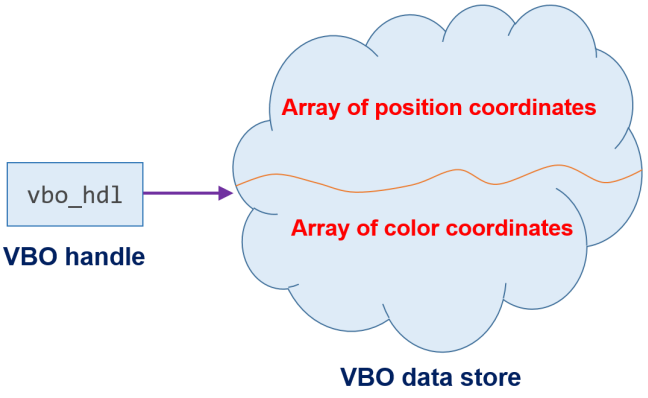
`glNamedBufferStorage` creates an immutable data store to store per-vertex data in server-side memory. Next, the client-side data is copied to the buffer object's data store using `glNamedBufferSubData`:

```
// compute and store values to simplify VBO and VAO management
GLsizei position_data_offset    = 0;
GLsizei position_attribute_size = sizeof(glm::vec2);
GLsizei position_data_size      = position_attribute_size *
```

```
                                  static_cast<GLsizei>(pos_vtx.size());
GLsizei color_data_offset       = position_data_size;
GLsizei color_attribute_size    = sizeof(glm::vec3);
GLsizei color_data_size         = color_attribute_size *
                                  static_cast<GLsizei>(clr_vtx.size());

glCreateBuffers(1, &vbo_hdl);
glNamedBufferStorage(vbo_hdl,
                     position_data_size + color_data_size,
                     nullptr, // nullptr means no data is transferred
                     GL_DYNAMIC_STORAGE_BIT);

/*
  + position_data_offset             + color_data_offset
  |                                  |
  v                                  v
  +----------------------------------------------------------------------+
  |           Vertex Data            |           Color Data              |
  +----------------------------------+-----------------------------------+
             position_data_size                  color_data_size
  <---------------------------------> <---------------------------------->
*/
glNamedBufferSubData(vbo_hdl, position_data_offset,
                     position_data_size, pos_vtx.data());
glNamedBufferSubData(vbo_hdl, color_data_offset,
                     color_data_size, clr_vtx.data());
```

The fourth argument `GL_DYNAMIC_STORAGE_BIT` for command `glNamedBufferStorage` says that the contents of the data store may be updated by calls to `glNamedBufferSubData`.

The following picture conceptualizes the code related to setting up the VBO pointed to by `vbo_hdl`:



Creating Vertex Array Object

Per-vertex data stored as vertex arrays in data stores of VBOs must be streamed as input to the vertex shader. An OpenGL application is free to define its own vertex data; each type of per-vertex data corresponds to an input *generic vertex attribute* variable in the vertex shader. All OpenGL implementations must support a minimum of 16 generic vertex attributes. The number of vertex attributes actually supported by the graphics hardware can be queried:

```
GLint max_vtx_attribs;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &max_vtx_attribs);
std::cout << "Maximum vertex attributes: " << max_vtx_attribs << '\n';
```

An ***attribute*** is defined as a vector consisting of one to four components. All components in the attribute must share a common data type. For example, a position attribute might be defined as three `GLfloat` components (x, y, z) or a color might be defined as four `GLubyte` components (r, g, b, a).

It is the job of the OpenGL application to store the per-vertex data in a VBO and to use a vertex array object to connect the VBO data with the input vertex attributes in the vertex shader. This task is implemented by the fixed function stage called ***vertex puller*** [also known as ***vertex fetch***]. At different instances of time, contents of vertex data arrays with varying formats stored in different VBOs are streamed as input to the vertex shader through its input vertex attributes. To ensure correct behavior, the vertex puller stage would require state information such as the component count of each vertex attribute, the data format of each element, and where these vertex arrays are stored in a VBOs' data store. The ***vertex array object*** [VAO] is a container object that represents a set of vertex attributes with each set specifying specific per-vertex data in the VBO's data store. Think of a VAO as containing information and state that allows the data stored in VBOs to be connected to input vertex attributes of a shader.

OpenGL 4.5 introduced command `glCreateVertexArrays` to directly create a VAO initialized to a default state and provide a handle to that VAO:

```
glCreateVertexArrays(1, &vaoid); // vaoid is data member of GLApp::GLModel
```

After creating and binding a VAO, it needs to be filled with the state information necessary for the vertex puller stage to stream vertex data from VBOs to the input vertex attributes of a vertex shader. We begin by calling OpenGL command `glEnableVertexArrayAttrib` to enable generic attribute index of VAO handle `vaoid`:

```
glEnableVertexArrayAttrib(vaoid, 8); // VAO's vertex attribute index is 8
```

Note that I'm enabling generic attribute index `8` while most examples enable attribute index `0`. I could have used any value between `0` and `GL_MAX_VERTEX_ATTRIBS - 1` [see above to know what this token means]. I'm simply using `8` rather than `0` to avoid confusion between two index types: an attribute index and a vertex buffer binding index [more on a binding index below].

Now, we need to update the VAO handle with the buffer containing the data associated with vertex attribute index `8`. In the following lines, we'll be associating the vertex position array in the data store pointed to by VBO handle `vbo_hdl` with vertex attribute index `8`.

We've previously created a VBO referenced by handle `vbo_hdl` containing vertex position and vertex color arrays. Before an OpenGL object can be used, it must be bound to the OpenGL context. When binding an object to the OpenGL context, the programmer must specify the kind of data stored in the buffer. The OpenGL specification uses the term *binding point* or *target* to indicate the specific nature and type of data that a buffer object is a data store for. Each binding point or target type corresponds to a distinct set of commands that manage objects of that type. The maximum number of binding points on a platform is determined by enumeration constant `GL_MAX_VERTEX_ATTRIB_BINDINGS`. We bind the vertex position array data in VBO handle `vbo_hdl` to vertex buffer binding point `3` and update VAO handle `vaoid` with this information:

```
glVertexArrayVertexBuffer(vaoid,
                          3,     // vertex buffer binding point
                          vbo_hdl,
                          position_data_offset,
                          position_attribute_size);
```

The fourth argument `position_data_offset` specifies the offset in the VBO's data store to access the vertex position array while the fifth argument `position_attribute_size` provides the size of each element in the vertex position array.

The previous step has updated VAO handle `vaoid` with *where* [which VBO?] the vertex position array data is store. The third step is to call OpenGL command `glVertexArrayAttribFormat` to provide VAO handle `vaoid` with the format and organization of the vertex position array data [number of coordinates, data type of coordinate element, and so on].

```
glVertexArrayAttribFormat(vaoid,
                          8,        // VAO's vertex attribute index
                          2,
                          GL_FLOAT,
                          GL_FALSE,
                          0);
```

Since our position coordinates are specified as $2-$tuples (x, y) with data type `GL_FLOAT`, the third and fourth arguments to the command are `2` and `GL_FLOAT`, respectively. Since we're using floating-point values for vertex position coordinates, the fifth argument is not used. Finally, the sixth argument `0` indicates that the vertex position coordinates are tightly packed next to each other with zero padding.

The final step is to provide an association between VAO's vertex attribute index [which in our example is `8`] and the VBO's vertex buffer binding point [which in our example is `3`] using the OpenGL command `glVertexArrayAttribBinding`:

```
glVertexArrayAttribBinding(vaoid,
                           8,  // VAO's vertex attribute index
                           3); // VBO's vertex buffer binding point
```

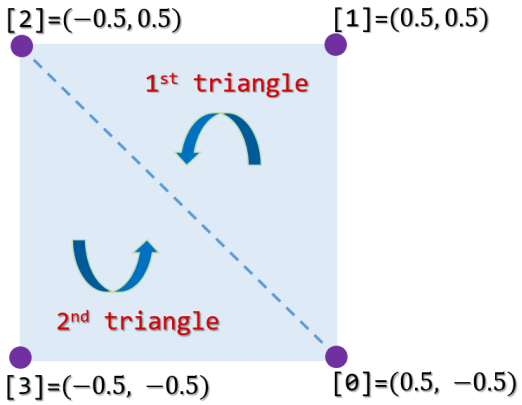
These steps need to be repeated for the vertex color attribute. The following code fragment contains the VAO setup for both position and color attributes in member function `GLModel::setup_vao`:

```
glCreateVertexArrays(1, &vaoid);

// for vertex position array, we use vertex attribute index 8
// and vertex buffer binding point 3
glEnableVertexArrayAttrib(vaoid, 8);
glVertexArrayVertexBuffer(vaoid, 3, vbo_hdl,
                          position_data_offset, position_attribute_size);
glVertexArrayAttribFormat(vaoid, 8, 2, GL_FLOAT, GL_FALSE, 0);
glVertexArrayAttribBinding(vaoid, 8, 3);

// for vertex color array, we use vertex attribute index 9
// and vertex buffer binding point 4
```

```
glEnableVertexArrayAttrib(vaoid, 9);
glVertexArrayVertexBuffer(vaoid, 4, vbo_hdl,
                          color_data_offset, color_attribute_size);
glVertexArrayAttribFormat(vaoid, 9, 3, GL_FLOAT, GL_FALSE, 0);
glVertexArrayAttribBinding(vaoid, 9, 4);
```

Setting up rectangle model topology

The final step involves defining the topology of the rectangle model which is pictured again below:



Since the rectangle model is rendered as two triangles, data member `GLModel::primitive_type` must be assigned OpenGL enumeration constant `GL_TRIANGLES`:

```
primitive_type = GL_TRIANGLES;
```

The rectangle model's topology consists of two counterclockwise oriented triangles. Each triangle is specified by a sequence of three indices with each index indicating elements in vertex attribute arrays where the corresponding position and color coordinates are stored. To render two triangles, six indices are required and they're sequentially stored in an array:

```
std::array<GLushort, 6> idx_vtx {
  0, 1, 2, // 1st triangle's position and color coordinates are stored
           // in indices 0, 1, 2 of vertex attribute arrays in VBO
  2, 3, 0  // 2nd triangle's position and color coordinates are stored
           // in indices 0, 1, 2 of vertex attribute arrays in VBO
};
```

Data member `GLModel::idx_elem_cnt` must be assigned the total number of indices. Since we'll be using OpenGL draw command `glDrawElements` to render geometry, the value assigned to `GLModel::idx_elem_cnt` will be used by OpenGL to deduce the number of triangles to be rendered. In our case, the number of indices is 6 and `glDrawElements` will deduce the number of triangles being rendered equivalent to 2:

```
idx_elem_cnt = static_cast<GLuint>(idx_vtx.size());
```

Next, this topology information must be transferred from client-side [that is, CPU] memory to server-side [that is, GPU] memory:

```
GLuint ebo_hdl;
glCreateBuffers(1, &ebo_hdl);
glNamedBufferStorage(ebo_hdl,
                     sizeof(GLushort) * idx_elem_cnt,
                     reinterpret_cast<GLvoid*>(idx_vtx.data()),
                     GL_DYNAMIC_STORAGE_BIT);
```

Unlike with vertex attribute arrays, we can both allocate server-side storage and copy data from client-side to server-side using OpenGL command `glNamedBufferStorage`. This is possible because the data is laid out linearly in a single client-side array. The flag `GL_DYNAMIC_STORAGE_BIT` says the contents of the data store may be updated in later calls to copy vertex attributes from client memory to server memory [this is potentially useful if the vertices are being animated by the CPU].

The next step is to configure the VAO's state with the buffer object whose data store contains the topology [OpenGL uses the term *elements* to refer to the topology of a model] data:

```
glVertexArrayElementBuffer(vaoid, ebo_hdl);
```

Once the VAO's state has been completely specified, it is important to break the VAO's binding to avoid any inadvertent changes to the VAO's state:

```
glBindVertexArray(0);
```

Definition of `GLModel::setup_vao`

The complete definition of function `GLModel::setup_vao` will look like this:

```
void GLApp::GLModel::setup_vao() {
  std::array<glm::vec2, 4> pos_vtx {
    glm::vec2(0.5f, -0.5f), glm::vec2(0.5f, 0.5f),
    glm::vec2(-0.5f, 0.5f), glm::vec2(-0.5f, -0.5f)
  };
  std::array<glm::vec3, 4> clr_vtx {
    glm::vec3(1.f, 0.f, 0.f), glm::vec3(0.f, 1.f, 0.f),
    glm::vec3(0.f, 0.f, 1.f), glm::vec3(1.f, 1.f, 1.f)
  };

  GLsizei position_data_offset    = 0;
  GLsizei position_attribute_size = sizeof(glm::vec2);
  GLsizei position_data_size      = position_attribute_size *
                                    static_cast<GLsizei>(pos_vtx.size());
  GLsizei color_data_offset       = position_data_size;
  GLsizei color_attribute_size    = sizeof(glm::vec3);
  GLsizei color_data_size         = color_attribute_size *
                                    static_cast<GLsizei>(clr_vtx.size());
  glCreateBuffers(1, &vbo_hdl);
  glNamedBufferStorage(vbo_hdl,
                       position_data_size + color_data_size,
                       nullptr,
                       GL_DYNAMIC_STORAGE_BIT);
  glNamedBufferSubData(vbo_hdl, position_data_offset,
                       position_data_size, pos_vtx.data());
```

```
  glNamedBufferSubData(vbo_hdl, color_data_offset,
                       color_data_size, clr_vtx.data());

  glCreateVertexArrays(1, &vaoid);
  glEnableVertexArrayAttrib(vaoid, 8);
  glVertexArrayVertexBuffer(vaoid, 3, vbo_hdl,
                            position_data_offset, position_attribute_size);
  glVertexArrayAttribFormat(vaoid, 8, 2, GL_FLOAT, GL_FALSE, 0);
  glVertexArrayAttribBinding(vaoid, 8, 3);

  glEnableVertexArrayAttrib(vaoid, 9);
  glVertexArrayVertexBuffer(vaoid, 4, vbo_hdl,
                            color_data_offset, color_attribute_size);
  glVertexArrayAttribFormat(vaoid, 9, 3, GL_FLOAT, GL_FALSE, 0);
  glVertexArrayAttribBinding(vaoid, 9, 4);

  primitive_type = GL_TRIANGLES;

  std::array<GLushort, 6> idx_vtx { 0, 1, 2, 2, 3, 0 };
  idx_elem_cnt = static_cast<GLuint>(idx_vtx.size());

  GLuint ebo_hdl;
  glCreateBuffers(1, &ebo_hdl);
  glNamedBufferStorage(ebo_hdl, sizeof(GLushort) * idx_elem_cnt,
                       reinterpret_cast<GLvoid*>(idx_vtx.data()),
                       GL_DYNAMIC_STORAGE_BIT);
  glVertexArrayElementBuffer(vaoid, ebo_hdl);
  glBindVertexArray(0);
}
```

GLSL shader program

To generate useful images, every modern OpenGL program must minimally define a vertex shader and a fragment shader. Geometry, tessellation, and compute shaders are outside the scope of this course. Shaders are authored in the OpenGL Shading Language which is a fundamental and integral part of the OpenGL API. Each shader executes within a different stage of the graphics rendering pipeline.

GLSL has a preprocessor whose behavior is quite similar to the C preprocessor. The preprocessor is a pre-compilation step over the shaders' source code that performs simple text substitution. Similar directives are provided including `#define`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`.

When authoring a shader, you've to specify the GLSL version you are coding for. GLSL version is always bound to a specific OpenGL version, so in order to choose a GLSL version that supports the features you need, you're also tied to the minimum OpenGL version that supports that GLSL version. The `#version` directive is meant to force the GLSL compiler to stick to a certain GLSL version. It has to be placed in the first line of the shader. `#version` is useful because it ensures better compatibility. If you want to use only the feature set of a certain GLSL version [for example, to avoid deprecated things in your shader code], the `#version` directive is critical because the compiler will throw an error when it detects that you are using newer or older GLSL features than the version you actually have specified. Since we've previously defined a OpenGL 4.5 core profile, the first line of every shader throughout this course will be:

```
#version 450 core
```

Raw string literals

We'll be using raw string literals to author shaders. Recall from HLP2 that raw string literals are string literals with a prefix containing `R`. The particular `R"(...)"` notation represents a raw string literal with everything between the delimiters `(` and `)` becoming part of the string. For example, the following code fragment:

```
std::cout << R"("a ""b """c)" << '\n';
char const *raw_str = R"("a ""b """c)";
std::cout << raw_str << '\n';
std::string str {R"("a ""b """c)"};
std::cout << str << '\n';
```

will write the following text to the standard output stream:

```
"a ""b """c
"a ""b """c
"a ""b """c

```

We're using raw string literals because everything between the delimiters including special C++ tokens such as `"` and `\n` become part of the string. This can be particularly useful when dealing with regular expressions, file paths, or any other strings containing a lot of special characters.

Vertex shader

In directory tutorial-1/src, edit the vertex shader file labeled my-tutorial-1.vert. Encode the `#version` directive and the rest of the shader code as a raw string literal like this:

```
R"( #version 450 core

  // other shader code here ...
)"
```

Typically, a vertex shader would perform the following per-vertex operations: transformation of position coordinates, transformation of normal coordinates [for real-time lighting], normalization of normal coordinates [for normal mapping], implement per-vertex lighting calculations, and computation of texture coordinates. However, a vertex shader doesn't replace all of the operations in the rendering pipeline. In particular, it doesn't replace the projection operations that map clip coordinates to window coordinates. The specific functions that are still done by the fixed-function stages include view volume clipping, perspective projection, viewport transforms, and back face culling. Minimally, every vertex shader must transform vertices that are usually given in model space coordinates into clip space coordinates.

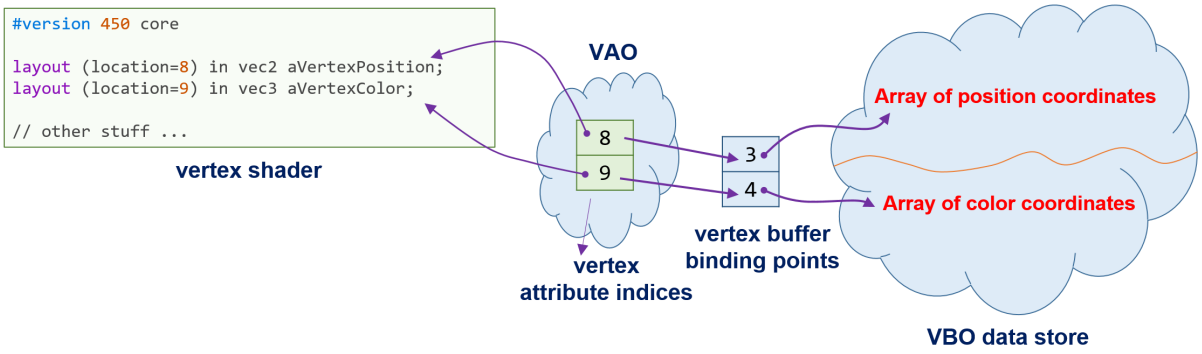
We must supply the vertex shader both vertex position and vertex color coordinates that were previously written into a VBO's data store. A vertex shader can receive data from the application through two kinds of inputs: ***attribute variables*** and ***uniform variables***. Vertex shaders use attribute variables to store per-vertex data that will be read from VBO data store(s). Uniform variables are used to store data that is common to all vertices and this data is transferred by the OpenGL application at runtime. Uniform variables will not be discussed any further in this tutorial.

Recall that we setup VAO handle `vaoid` to associate generic vertex attribute indices `8` and `9` with per-vertex position and color coordinates in the VBO's data store. We'll now use the GLSL `layout` qualifier to set up the vertex shader to read per-vertex data from these VBO vertex attribute indices:

```
// 8 is vertex attribute index that is associated with per-vertex
// position coordinates in VBO; this association was performed by
// this call: glEnableVertexArrayAttrib(vaoid, 8);
layout (location=8) in vec2 aVertexPosition;

// 9 is vertex attribute index that is associated with per-vertex
// position coordinates in VBO; this association was performed by
// this call: glEnableVertexArrayAttrib(vaoid, 9);
layout (location=9) in vec3 aVertexColor;
```

Keyword `in` is used to indicate that the values created by the application that are associated with vertex attribute indices `8` and `9` will be read into vertex attribute variables `aVertexPosition` and `aVertexColor`, respectively.

The following picture illustrates the relationship between VBO vertex binding points, VAO vertex attribute indices, and input variables of a vertex shader:



Notice that we're *explicitly* telling the shader the vertex attribute indices the OpenGL application has placed the per-vertex data into. This is only useful if you always remember to load per-vertex position coordinates into vertex attribute index `8` and per-vertex color coordinates into vertex attribute index `9`. For newer GLSL compiler versions including SPIR-V, you don't have an option - you're required to specify vertex attribute locations using the `layout` qualifier. Older tutorials on the web might use OpenGL commands `glBindAttribLocation` - this is no longer recommended!!!

`vec2` is a native GLSL type that specifies a 2—component vector where each component is a single-precision floating-point value. What does native type mean? Native types are types such as vectors and matrices [up to 4 dimensions] that GPU cores can process in a single cycle; similar to the way CPUs can process `int`, `float`, and `double` values in a single cycle.

Every shader must generate an output. A key function of every vertex shader is to use attribute variables to generate new values that are copied into `out` variables for subsequent shaders to use. Or, in some cases, a vertex shader might just "pass through" attribute variables contents through `out` variables for subsequent shaders to use. Vertex shaders have several kinds of output. The most important are transformed vertices and per-vertex texture, normal, and color coordinates. Of course, the vertex shader can compute or re-compute any of these coordinates. If you use a fragment shader, the vertex shader can use output variables to supply per per-vertex values to the rasterizer which then interpolates these values. In turn, the rasterizer can provide the fragment shader with these interpolated values using corresponding input `in` variables defined

by the fragment shader. The fragment shader can use these interpolated color, normal, and texture coordinate values to carry out sophisticated operations on each fragment. In our minimal example, `out` variable `vColor` outputs the unchanged copy of the per-vertex color in the `in` attribute variable `aVertexColor` to the fixed stage rasterizer:

```
layout (location=0) out vec3 vColor;
```

The rasterizer will receive the unchanged copies of the per-vertex color at the triangle's three vertices. It interpolates these per-vertex colors across the surface of the triangle and assigns the interpolated color computed at each fragment as input to the fragment shader.

Just like in C/C++, execution of a shader begins with function `main`. However, unlike C/C++ `main` functions, `main` functions in shaders cannot take parameters nor return a value.

Inside function `main`, we use a special built-in member `gl_Position` to hold transformed clip coordinates of the position coordinates stored in the `in` attribute variable `aVertexPosition`. This picture shows position coordinates transformed by the vertex shader into clip coordinates that are then supplied as inputs to certain fixed-function stages of the graphics rendering pipeline. This is the minimal requirement for every vertex shader and is its main purpose. Not placing a value in `gl_Position` will render the shader unusable and ill-formed and will cause a compiler error. Since the position coordinates in the VBO data store are stored in NDC coordinates, we simply convert the $2-$component (x, y) values to $4-$component clip coordinates $(x, y, 0.0, 1.0)$ using a `vec4` constructor. Finally, we assign the unchanged copy of the per-vertex color in the `in` attribute variable `aVertexColor` to the `out` variable `vColor`. Function `main` looks like this:

```
void main(void) {
  gl_Position = vec4(aVertexPosition, 0.0, 1.0);
  vColor = aVertexColor;
}
```

The minimal but complete vertex shader my-tutorial-1.vert will look like this:

```
R"( #version 450 core

  layout (location=8) in vec2 aVertexPosition;
  layout (location=9) in vec3 aVertexColor;

  layout (location=0) out vec3 vColor;

  void main() {
    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
    vColor      = aVertexColor;
  }
)"
```

The starter code already ensures that the vertex shader file is accessible within glapp.cpp.

```
const std::string my_tutorial_1_vs = {
  #include "my-tutorial-1.vert"
};
```

Fragment shader

Using the primitive type specified by an OpenGL command [such as `glDrawArrays` or `glDrawElements`], the fixed-function primitive assembly stage organizes collections of vertices into a single, complete topological primitive that will then be operated on by fixed-function stages consisting of clipping, perspective division, viewport transform followed by culling. Primitives that have not been clipped out nor culled because of their facing orientations are decomposed by fixed-function rasterizer stage into smaller units called *fragments* that correspond to pixels in the framebuffer. The rasterizer interpolates quantities such as colors, depths, and texture coordinates that have been specified as `out` values in the vertex shader. The fragment shader executes once per each generated fragment and uses these interpolated values, as well as other kinds of information, to minimally determine the color of the fragment's pixel. In addition, a fragment shader replaces or adds the following operations: texturing, per-pixel lighting, fog, and discarding fragments. However, a fragment shader does not replace all the operations in the rendering pipeline. In particular, a fragment shader does not replace several raster operations, including blending, stencil test, depth test, scissor test.

There are two important constraints on fragment shaders. First, a fragment shader can only write into the framebuffer but can't read from the framebuffer. Multi-pass rendering techniques are required if the fragment shader computations are to be combined with the existing framebuffer color. Second, a fragment shader is executed for a specific fragment that is located at a specific framebuffer location (x, y) and the shader cannot be used to modify fragments at other framebuffer locations.

As with any other shader, the first line of the fragment shader named my-tutorial-1.frag must look like this [ignoring the fact that we're encoding shaders as raw string literals]:

```
1 #version 450 core
```

Perhaps the most important inputs to fragment shaders are the variables that are passed to the fragment shader as `out` variables by the shader preceding the rasterizer. This could be a vertex, geometry, or tessellation shader. These variables are the data that are interpolated across a graphics primitive by the rasterizer in order to give the fragment shader enough information to set the colors of each fragment. The vertex shader my-tutorial-1.vert "passed through" its input per-vertex color in `aVertexColor` to `out` variable `vColor`. If we're rendering a triangle primitive, each triangle vertex will consist of three colors passed through by the vertex shader. By default, the rasterizer will use hyperbolic or perspective-correct interpolation [will be covered in the second graphics course] to determine the color at an interior fragment. Our fragment shader requires this interpolated color to determine the corresponding pixel's color. The fragment shader signals its intent to receive this interpolated color using the following declaration:

```
1 layout (location=0) in vec3 vInterpolatedColor;
```

The vertex shaders' outputs must correspond to the fragment shaders' inputs: inputs and outputs must match by name, type and qualifiers exactly. This strict correspondence between the names and types of the vertex shaders' outputs and the fragment shaders' inputs is a major source of errors. Such hard-to-track errors can be avoided by tagging `in` and `out` variables with an index, such that they correspond by the index instead of by name [or by type]. For this reason, even though `out` variable `vColor` in the vertex shader doesn't match the name of `in` variable `vInterpolatedColor` in the fragment shader, the dynamic linker of these two shaders will make

the correspondence between these `in` and `out` variables because both names are tagged with index `0` using the `layout` qualifier.

The primary purpose of a fragment shader is to compute color values for a fragment that may ultimately be written into the framebuffer. User-defined `out` variables in the fragment shader link the fragment outputs to fixed-function per-fragment operations at the back end of the OpenGL rendering pipeline, and from there to the framebuffer. Which framebuffer? The default framebuffer can contain up to four color buffers, named `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, and `GL_BACK_RIGHT`. Monoscopic contexts include only *left* buffers, and stereoscopic contexts include both *left* and *right* buffers. Likewise, single buffered contexts include only *front* buffers, and double-buffered contexts include both *front* and *back* buffers. Since our OpenGL context was initialized to be double-buffered and monoscopic, `GL_FRONT_LEFT` and `GL_BACK_LEFT` color buffers are available. The front buffer is, more or less, what we see on the screen. The back buffer is where the image is rendered to. When we want the rendered image to become visible, our application makes a call to GLFW function `glfwSwapBuffers` to swap the front and back buffers. Color buffers are identified by color buffer indices. OpenGL function glDrawBuffers can be used to specify the relationship between color buffer indices and the buffers into which fragment colors will be written. By default, `GL_BACK_LEFT` [the back buffer of the double-buffered framebuffer] has color buffer index 0. This is the color buffer where our fragment shaders' output must be written to and we say so using the following declaration:

```
layout (location = 0) out vec4 fFragColor;
```

The remaining part is to implement function `main` that expands (r, g, b) values in `vInterpColor` to (r, g, b, a) using a `vec4` constructor. The minimal but complete fragment shader is shown below:

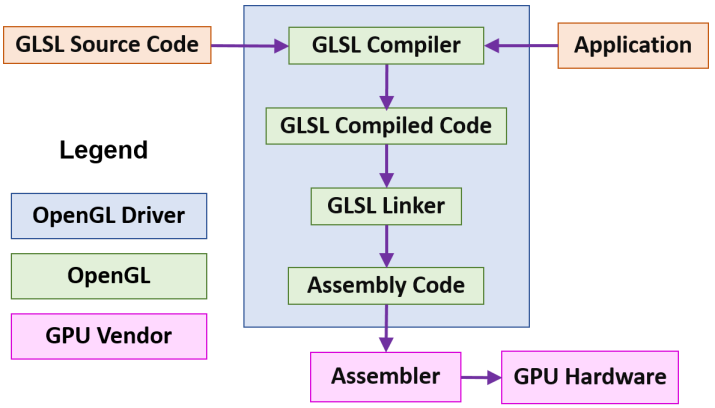
```
R"( #version 450 core
  layout (location=0) in vec3 vInterpColor;
  layout (location=0) out vec4 fFragColor;

  void main () {
    fFragColor = vec4(vInterpColor, 1.0);
  }
)"
```

Similar to vertex shader, the starter code already makes the fragment shader file accessible in glapp.cpp:

```
const std::string my_tutorial_1_fs = {
  #include "my-tutorial-1.frag"
};
```

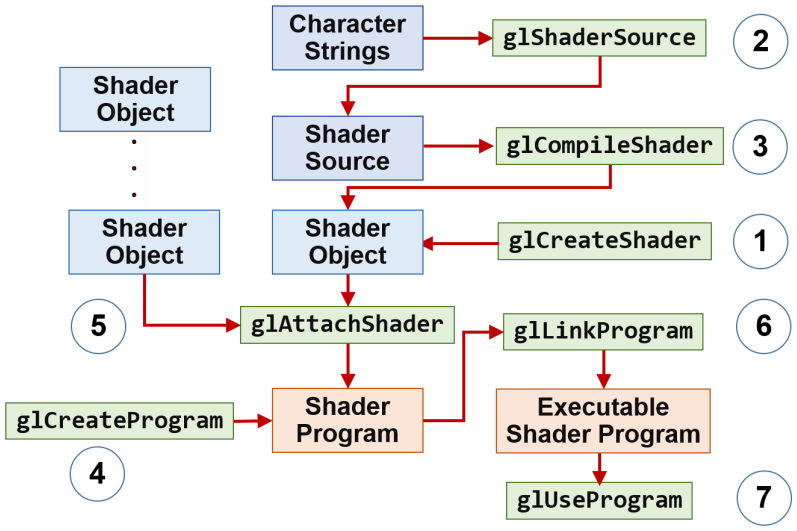
Creating a shader program object

Just as C/C++ code located in source file(s) must be compiled and linked to create an executable program, the GLSL vertex shader in my-tutorial-1.vert and fragment shader in my-tutorial-1.frag must be individually compiled and then linked together into a shader program. As shown by the following picture, the GLSL compiler/linker is built into the OpenGL API and shaders are compiled and linked by a running OpenGL program.



OpenGL 4.1 added the ability to save compiled shader programs to a file enabling OpenGL programs to avoid the overhead of shader compilation by loading precompiled shader programs. This process is not described in this document and is left as an exercise for the reader.

The steps involved in creating a shader program are illustrated in the following picture:



These steps have been encapsulated in class `GLSLShader`; both interface file glslshader.h and implementation source file glslshader.cpp are provided. Rather than describe the entire functionality of class `GLSLShader`, a general description of the seven steps illustrated in the above picture is provided:

1. Use OpenGL function `glCreateShader` to create a vertex shader object. The function returns a handle to the vertex shader object.
2. The vertex shader program - stored in a `std::string` object called `my_tutorial_1_vs` - is converted into a C-style string. Copy the vertex shader source in the C-style string into the vertex shader object using function `glShaderSource`.
3. Compile the shader source in the vertex shader object using function `glCompileShader`. The compilation status can be queried by calling `glGetShaderiv`. If the compile status is `GL_FALSE`, the shader log can be queried for additional details using function `glGetShaderInfoLog`. Repeat the above three steps for the fragment shader.
4. Supposing both vertex and fragment shader objects return their compilation status as `GL_TRUE`, call function `glCreateProgram` to create an empty shader program object and return a handle to that shader program object.
5. Attach the vertex shader object to the shader program object by calling function `glAttachShader`. Make a second call to the function to attach the fragment shader object to the shader program.

6. Link the vertex and fragment shader objects into a single shader program with a call to `glLinkProgram`. This step makes connections between `in` variables of one shader to `out` variables of a preceding shader and makes connections between `in` and `out` variables of a shader to appropriate buffers in the OpenGL application. The link status can be verified using function `glGetProgramiv` and if the link status is `GL_FALSE`, the program log can be queried for additional details using function `glGetProgramInfoLog`. The vertex and fragment shader objects can now be detached and then deleted using functions `glDetachShader` and `glDeleteShader`.
7. If linking in the previous step is successful, install the shader program in the OpenGL rendering pipeline with a call to `glUseProgram`.

Compiling the shaders and linking the shader objects into a shader program is facilitated by function `GLModel::setup_shdrpgm` [in source file glapp.cpp]:

```
1  void GLApp::GLModel::setup_shdrpgm() {
2    if (!shdr_pgm.CompileShaderFromString(GL_VERTEX_SHADER,
3                                          my_tutorial_1_vs)) {
4      std::cout << "Vertex shader failed to compile: ";
5      std::cout << shdr_pgm.GetLog() << std::endl;
6      std::exit(EXIT_FAILURE);
7    }
8    if (!shdr_pgm.CompileShaderFromString(GL_FRAGMENT_SHADER,
9                                          my_tutorial_1_fs)) {
10     std::cout << "Fragment shader failed to compile: ";
11     std::cout << shdr_pgm.GetLog() << std::endl;
12     std::exit(EXIT_FAILURE);
13   }
14
15   if (!shdr_pgm.Link()) {
16     std::cout << "Shader program failed to link!" << std::endl;
17     std::exit(EXIT_FAILURE);
18   }
19
20   if (!shdr_pgm.Validate()) {
21     std::cout << "Shader program failed to validate!" << std::endl;
22     std::exit(EXIT_FAILURE);
23   }
24 }
```

Rendering rectangle model

First, update function `GLApp::init` [in source file glapp.cpp] to invoke the newly defined functions `GLModel::setup_vao` and `GLModel::setup_shdrpgm`.

```
void GLApp::init() {
  // Part 1: clear color buffer with ...
  glClearColor(1.f, 0.f, 0.f, 1.f);

  // Part 2: use the entire window as viewport ...
  GLint w = GLHelper::width, h = GLHelper::height;
  glViewport(0, 0, w, h);

  // Part 3: set up model's VAO and create model's shader program
  mdl.setup_vao();     // mdl is a static object of type GLModel
  mdl.setup_shdrpgm(); // declared in type GLApp
}
```

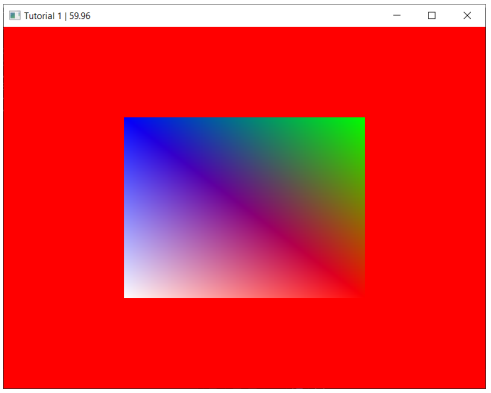
Next, define function `GLModel::draw` [in source file glapp.cpp]:

```
void GLApp::GLModel::draw() {
  // there are many shader programs initialized - here we're saying
  // which specific shader program should be used to render geometry
  shdr_pgm.Use();
  // there are many models, each with their own initialized VAO object
  // here, we're saying which VAO's state should be used to set up pipe
  glBindVertexArray(vaoid);
  // here, we're saying what primitive is to be rendered and how many
  // such primitives exist.
  // the graphics driver knows where to get the indices because the VAO
  // containing this state information has been made current ...
  glDrawElements(primitive_type, idx_elem_cnt, GL_UNSIGNED_SHORT, NULL);
  // after completing the rendering, we tell the driver that VAO
  // vaoid and current shader program are no longer current
  glBindVertexArray(0);
  shdr_pgm.UnUse();
}
```

Function `GLModel::draw` must be invoked from `GLApp::draw` [in glapp.cpp]:

```
void GLApp::draw() {
  glClear(GL_COLOR_BUFFER_BIT); // clear back buffer as before
  mdl.draw();                   // now, render rectangle model
}
```

The output must look like this:



🎯Task 6: Update rectangle model's VBO at runtime

In Task 3, key event U was used to toggle the color for clearing the back color buffer between a static color and an interpolated color computed at runtime. This task provides similar behavior but with respect to the rectangle model's vertex color attributes. When key event U is false, the rectangle model's color attributes must be equivalent to the values specified in Task 5:

```
std::array<glm::vec3, 4> clr_vtx {
  glm::vec3(1.f, 0.f, 0.f), glm::vec3(0.f, 1.f, 0.f),
  glm::vec3(0.f, 0.f, 1.f), glm::vec3(1.f, 1.f, 1.f)
};
```

When key event U is toggled, an interpolated color is computed using two colors in function `glApp::update` [in source file glapp.cpp]:

```
// you can choose any colors you wish ...
glm::vec3 init_clr  { 0.f,  1.f,  1.f };
glm::vec3 final_clr { 1.0f, 1.0f, 0.0f };

// compute per-vertex interpolated color between init_clr and final_clr
// so that the interpolated color is different for each vertex ...
clr_vtx[0] = ???
clr_vtx[1] = ???
clr_vtx[2] = ???
clr_vtx[3] = ???
```

When key event U is toggled again, the rectangle model's color attributes are again equivalent to the values specified in Task 5.

Every frame the vertex color attribute data in the rectangle model's VBO must be overwritten with the contents of array `clr_vtx`. Use command `glNamedBufferSubData` [used and explained in Task 5] to update vertex color attribute data in the rectangle model's VBO:

```
glNamedBufferSubData(mdl.vbo_hdl, color_data_offset,
                     color_data_size, clr_vtx.data());
```

Run the sample executable to better understand the expected behavior and output.

ℹ️In this tutorial, if you resize the window, the scene rendered with the quad will stay fixed at the specified vertex positions. This behavior is intentional. In future tutorials, we'll cover how to properly update your scene when the window size changes.

Submission

1. Source and header files for tutorial-1 must be placed in a directory labeled as: <login>-<tutorial-1>. If your Moodle student login is foo, then the directory would be labeled as foo-tutorial-1 and would have the following structure and layout:

```
foo-tutorial-1          # You're submitting Tutorial 1
├─ include              # Header files - *.hpp and *.h files
└─ src                  # Source files - *.cpp and .c files
   └─ my-tutorial-1.vert # Vertex shader file
   └─ my-tutorial-1.frag # Fragment shader file
```

2. *Zip* the directory and upload the resultant file foo-tutorial-1.zip to the assessment's submission page.

⚠ Before Submission: Verify and Test it ⚠

1. Unzip your archive file foo-tutorial-1.zip in directory test-submissions directory. Your directory and file layout should look like this:

```
csd2101-opengl-dev    # Sandbox directory for all assessments
├─ test-submissions   # ⚠ Test submissions here before uploading
|  └─ foo-tutorial-1 # Tutorial 1 is submitted by foo
|     └─ include     # Header files - *.hpp and *.h files
|     └─ src         # Source files - *.cpp and .c files
|        └─ my-tutorial-1.vert  # Vertex shader file
|        └─ my-tutorial-1.frag  # Fragment shader file
└─ csd2101.bat              # Automation Script
```

2. Select option R to reconfigure the solution with new project foo-tutorial-1.
3. Select option B to build the project. If there are no errors, an executable file foo-tutorial-1.exe will be created in directory build/Release. Alternatively, you can verify the project through the Visual Studio 2022 IDE.
4. Use the following checklist before uploading to the assessment's submission page:

Things to test before submission	Status
Assessment compiles without any errors	☐
All warnings resolved, zero warnings during compilation	☐
Executable generated and successfully launched in \text{Debug} and \text{Release} mode	☐
Directory is zipped, ensuring adherence to naming conventions as outlined in submission guidelines	☐
Upload zipped file to appropriate submission page	☐

ℹ The purpose of this verification step is not to guarantee the correctness of your submission. Instead, it verifies whether your submission meets the most basic rubrics [it compiles, it doesn't generate warnings, it links, it executes] or not and thus helps you avoid major grade deductions. And, remember you'll need every single point from these programming assessments to ensure a noteworthy grade!!! Read the section on Grading Rubrics for information on how your submission will be assigned grades.

Grading Rubrics

The core competencies assessed for this assessment are:

- [***core1***] Submitted code must build an executable file with **zero** warnings. This rubric is not satisfied if the generated executable crashes or displays nothing. In other words, if your source code doesn't compile nor link nor execute, there is nothing to grade.
- [***core2***] This competency indicates whether you're striving to be a professional software engineer, that is, are you implementing software that satisfies project requirements [are you submitting the required files? are they properly named? are they properly archived? are you submitting unnecessary files?], that is maintainable and easy to debug by you and others [are file and function headers properly annotated?]. Submitted source code must satisfy ***all*** requirements listed below. Any missing requirement will decrease your grade by one letter grade.
    - Source code must compile with ***zero*** warnings. Pay attention to all warnings generated by the compiler and fix them.
    - Source code file submitted is correctly named.
    - Source code file is *reasonably* structured into functions and *reasonably* commented. See next two points for more details.
    - If you've created a new source code file, it must have file and function header comments.
    - If you've edited a source code file provided by the instructor, the file header must be annotated to indicate your co-authorship. Similarly, if you're amending a previously defined function, you must annotate the function header to document your amendments. You must add a function header to every new function that you've defined.
- [***core3***] Completed $\mathbf{Task\ 1}$ that builds tutorial-1 project and executes the OpenGL application to display default [black] window.
- [***core4***] Completed $\mathbf{Task\ 2}$ to demonstrate ability to query and report OpenGL graphics driver capabilities.
- [***core5***] Completed $\mathbf{Task\ 3}$ to demonstrate basic competency of color representation and double buffering in OpenGL and responding to key event that toggles color for clearing back color buffer. Your implementation must be similar to or exceed the sample.
- [***core6***] Completed $\mathbf{Task\ 4}$ to demonstrate basic competency in researching an API and in following instructions laid out in the spec.
- [***core7***] Completed $\mathbf{Task\ 5}$ to demonstrate basic competency in programming constructs and toolboxes used in modern OpenGL programs to specify an object's geometry and topology to the vertex shader.
- [***core8***] Completed $\mathbf{Task\ 6}$ demonstrates ability to partially update VBOs at runtime by writing only vertex color attribute data [and not vertex position attribute data] to the model's VBO.

Mapping of Grading Rubrics to Letter Grades

The core competencies listed in the grading rubrics will be mapped to letter grades using the following table:

Grading Rubric Assessment	**Letter Grade**
There is no submission.	F

Grading Rubric Assessment	Letter Grade
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing.	F
If ***core2*** rubrics are not satisfied, final letter grade will be decreased by one. This means that if you had received a grade `A+` and ***core2*** is not satisfied, your grade will be recorded as `B+`, a `B+` would be recorded as `C+`, and so on.	
core3 rubric [$\mathbf{Task\ 1}$] is satisfied.	D
core4 rubric [$\mathbf{Task\ 2}$] is satisfied.	$D+$
core5 rubric [$\mathbf{Task\ 3}$] is satisfied.	C
core6 rubric [$\mathbf{Task\ 4}$] is satisfied.	B
core7 rubric [$\mathbf{Task\ 5}$] is satisfied.	A
core8 rubric [$\mathbf{Task\ 6}$] is satisfied.	$A+$

Things to research [not for submission]

1. The document doesn't say much about cleaning up when the application is terminated. For example, function `GLApp::cleanup` at present only calls `GLHelper::cleanup()` which destroy glfw window. Collate the resources obtained from the CPU and GPU and systematically release and return these resources back to the system(s). Otherwise, it is possible that both the CPU and GPU can become sluggish after repeatedly calling this program. To discard data in a buffer object, research OpenGL functions `glInvalidateBufferData` and `glDeleteBuffers`.
2. OpenGL functions are complex with lots of parameters with each parameter able to specify a variety of values. There's a lot to learn and it is easy to go wrong. This document doesn't say anything about how to check for errors when a function is called with incorrect or incompatible arguments. Begin your research by reading about OpenGL function `glGetError`. For more sophisticated debugging and profiling, you will need to learn about *debug contexts*. GLFW3 creates a default context when `glfwWindowHint` is called with argument `GLFW_OPENGL_DEBUG_CONTEXT` in function `GLHelper::init`. Creating a debug context enables debugging features. This page provides a simple example; read the ARB extension document for a more technical explanation.
3. Change the viewport settings to render to different portions of the back buffer.
4. Change the representation of geometry from *structure of arrays* format to *arrays of structure* format. What are the advantages and disadvantages of each of these geometry representation formats?
5. We dynamically modified the color attribute, and likewise, we can adjust the vertex attribute to change the positions of the vertices.
6. Discover methods for managing window resizing and ensuring the quad always renders in the center of the window.

7. How will you implement *instancing*? That is, using the same rectangle model defined previously, how can many instances of the model be rendered? What changes are required to the data structures?
8. Implement instancing by rendering the rectangle model to two different viewports.
9. Extend the idea of instancing to render thousands of instances of these rectangle models.
10. How will you implement linear and affine transforms to instances of the rectangle model?
11. How will you extend the rendering from NDC coordinates to the window to consider a scene in world coordinates with many different rectangle objects, each object having its own transformation matrix?
12. How can a camera be inserted in the scene so that the objects in the scene are rendered from the point of view of the camera?
13. How can this static camera be augmented to handle user input and move around the scene?