Short Circuit

OpenGL

Learn OpenGL
by Joey de Vries
June 2020
Up until Chapter 32
Skipped Chapter 27
Already did Chapter 47

Summary by Emiel Bos

1 Introduction

OpenGL is an API specification, developed and maintained by the Khronos Group. It specifies the input, output and performance of all functions, but the implementation is carried out by graphics card manufacturers. Each graphics card supports specific versions of OpenGL, which are included in its drivers. Higher versions of OpenGL are generally supported only by the latest GPUs. OpenGL supports *extensions*, in which graphics card manufacturers can supply additional, GPU-specific functionality or optmizations. The developer has to query whether any of these extensions are available before using them:

```
if(GL_ARB_extension_name) {
   // Do hardware supported modern stuff
}
else {
   // Do it the old OpenGL way
}
```

Before version 3.0, OpenGL only supported a fixed-function graphics pipeline in what is called *immediate mode*, which was easy-to-use but highly inflexible. Version 3.0 introduced programmable shaders that allowed programming different stages in the graphics pipeline, referred to as Modern OpenGL. Immediate mode functionality was deprecated in version 3.2, leaving OpenGL's *core-profile mode*, which is a division of OpenGL's specification that removed all old deprecated functionality.

OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate, referred to as the OpenGL *context*. We use state-changing functions to manipulate *objects*, which are collection of options that represents a subset of OpenGL's state. Objects as abstractions were introduced because OpenGL at its core is a C-library, and many of C's language constructs do not translate well to other high-level programming languages. Another slight downside of OpenGL being a C-library is that C does not support function overloading, and therefore OpenGL often defines multiple functions which you would expect to be overloaded. For example, if a function supports multiple data types, the function name likely needs a postfix, e.g. `glUniform4f()` expects four `float`s.

2 Setup

2.1 GLFW

In order to render, we need to create an OpenGL context and an application window. Since these operations are OS-specific and OpenGL tries to abstract itself from these operations, there are libraries that create a window, define a context, and handle user input. GLFW (Graphics Library FrameWork) is a basic one for OpenGL window and input and the one

we'll use here, but others are SDL(2) (bigger package that also supports sound, fonts, texture imports, some rendering abstractions), GLUT (very old and generally avoided) and SFML (generally avoided). GLFW needs to be built (with CMake) and linked (by adding `glfw3.lib` as a dependency in the linker settings), but they provide some precompiled binaries and header files on their website. Include GLFW using `#include <GLFW/glfw3.h>`.

2.2 GLAD

The location of OpenGL's functions are driver and OS specific, and therefore need to be queried during runtime. While you can manually retrieve the location of the functions you need and store them in function pointers, there are libraries that do this. We'll use GLAD; there's also GLEW but it doesn't seem very popular online. It offers a webservice where you can specify for which programming language (likely C++) and version of OpenGL you'd like to define and load all relevant OpenGL functions. Copy both include folders in your include directory (set in linker settings) and copy the `glad.c` file in your project. You should now be able to include GLAD using `#include <glad/glad.h>`. Include it before including GLFW; the include file for GLAD includes the required OpenGL headers behind the scenes, so those can be used by other includes that require OpenGL.

To create a GLFW window, initialize GLAD, and start rendering, define `main()` as follows:

```
#define WIDTH 800
#define HEIGHT 600

glfwInit(); // Initialize GLFW
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Give option GLFW_CONTEXT_VERSION_MAJOR value 3
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Tell GLFW to use OpenGL 3.3
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); // Tell GLFW to only use the core
    profile
//glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // Use on macOS


GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "App name", NULL, NULL);
if (window == NULL)
{
   std::cout << "Failed to create GLFW window" << std::endl;
   glfwTerminate();
   return -1;
}
glfwMakeContextCurrent(window); // Make the window context the main context on the current thread

// Initialize GLAD in order to use OpenGL functions
// Pass GLAD the GLFW function glfwGetProcAddress (GLFW defines the correct function based on OS),
    which loads the address of the OS-specific OpenGL function pointers
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
   std::cout << "Failed to initialize GLAD" << std::endl;
   return -1;
}

glViewport(0, 0, WIDTH, HEIGHT); // Tell OpenGL the size of the rendering window; the first two
    parameters set the location of the lower-left corner. OpenGL uses these dimensions to map from
    (-1, 1) to (0, WIDTH) and (0, HEIGHT).

glfwSetFramebufferSizeCallback(window, framebuffer_size_callback); // Register callback function at
    GLFW, defined below

// Render loop
while(!glfwWindowShouldClose(window))
{

   glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Specify the color to clear the screen with
   glClear(GL_COLOR_BUFFER_BIT); // Clear the specified buffer with the above color (black)

   render();
   processInput(window); // Call our function defined below
   glfwSwapBuffers(window); // Swap the color buffer and show it as output to the screen
   glfwPollEvents(); // Check if any (input) events are triggered, update the window state, and
       call corresponding callback functions
}

glfwTerminate(); // Delete all GLFW resources that were allocated
return 0;
```

The `framebuffer_size_callback()` callback function looks like:

```
// Define callback function that GLFW calls each time the window is resized
void framebuffer_size_callback(GLFWwindow* window, int w, int h) {
   glViewport(0, 0, w, h); // Pass the new dimensions to OpenGL
}
```

The `processInput()` function could be something like:

```
// Check whether ESC is pressed and, if so, close the window
void processInput(GLFWwindow *window) {
   if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
      glfwSetWindowShouldClose(window, true);
   }
}

glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED); // Tell GLFW to capture the mouse, but
    make it invisible
void mouse_callback(GLFWwindow* window, double xpos, double ypos) {
   float xoffset = xpos - lastX;
   float yoffset = lastY - ypos; // Reversed since y-coordinates range from bottom to top
   lastX = xpos;
   lastY = ypos;
}
glfwSetCursorPosCallback(window, mouse_callback); // Register mouse callback with GLFW

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
glfwSetScrollCallback(window, scroll_callback); // Register scroll callback with GLFW
```

A list of all available GLFW window handling options is available online.

2.3 GLM

GLM (OpenGL Mathematics) is a header-only library, meaning you can download and copy it to your `include` folder and include it in your code:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

It brings the vector and matrix manipulation functionality of GLSL, and more, to your host code. For example, to translate a vector (i.e. multiply the vector by a translation matrix):

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 matIdentity = glm::mat4(1.0f); // Identity matrix
glm::mat4 matTranslate = glm::translate(matIdentity, glm::vec3(1.0f, 1.0f, 0.0f)); // Multiply
    matIdentity by a translation matrix created from the given vector
vec = matTranslate * vec; // Translate vec

glm::mat4 matTransform = glm::mat4(1.0f); // (Will become a) scale + rotation matrix
matTransform = glm::rotate(matTransform, glm::radians(90.0f), glm::normalize(glm::vec3(1.3, 0.5,
    2.5))); // Rotate 90 degrees around the given (unit) vector
matTransform = glm::scale(matTransform, glm::vec3(0.5, 0.5, 0.5)); // Scale by 50%
```

Getting GLM variables to the shaders is easy (this will make sense when we talk about shaders in 5.3):

```
unsigned int location = glGetUniformLocation(shaderProgramHandle, "matrix");
glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(matTransform)); // Send 1 matrix without
    transposing it. GLM data doesn't always match OpenGL's specification, so we first convert it
```

3 Coordinate systems

A vertex goes from its definition in local space through multiple coordinate systems to screen space before it is rendered. The following lists the different coordinate systems (as numbered items) and the transforms between them (as arrow items):

1. *Local space*, or *object space*; the coordinates of an object are relative to the object's local origin. This is how an object is defined (e.g. with Blender).

↓ *Model matrix*; translates, scales and/or rotates your object to place it in the world.

2. *World space*; the coordinates of an object are relative to the world or scene's global origin. All objects are now in the same coordinate system.

↓ *View matrix*, or *LookAt matrix*; translates and/or rotates your object to place it in front of (or rather, relative to) the camera. It move the entire scene around inversed to where the camera is in world space. OpenGL itself doesn't have a notion of a camera, but we define it by its position in world space, the opposite of the direction it's looking at (obtained by subtracting the viewing target from the camera position), a vector pointing to the right from the camera (obtained as a cross product of the direction vector and the `(0, 1, 0)` "world-up" vector) and a vector pointing upwards from the camera (obtained as a cross product of the direction vector and the right vector). From these, we can obtain a view matrix:

$$\mathrm{V} = \begin{bmatrix} \boldsymbol{r}_x & \boldsymbol{r}_y & \boldsymbol{r}_z & 0 \\ \boldsymbol{u}_x & \boldsymbol{u}_y & \boldsymbol{u}_z & 0 \\ \boldsymbol{d}_x & \boldsymbol{d}_y & \boldsymbol{d}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -\boldsymbol{p}_x \\ 0 & 1 & 0 & -\boldsymbol{p}_y \\ 0 & 0 & 1 & -\boldsymbol{p}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where $\boldsymbol{r}$ is the right vector, $\boldsymbol{u}$ is the up vector, $\boldsymbol{d}$ is the inverse direction vector, and $\boldsymbol{p}$ is the position vector. What happens is we first translate all world-space coordinates to have the camera as origin, after which we multiply them with the inverse of the view-space's change-of-basis vector (which is equal to its transpose, since it's an orthonormal matrix) in order to get their view-space equivalents. GLM creates this matrix for you using:

```
glm::mat4 view = glm::lookAt(posCamera, posTarget, glm::vec3(0.0f, 1.0f, 0.0f));
```

3. *View space*, *eye space* or *camera space*; the coordinates are relative to the camera's point of view. The camera itself is the origin, positive x-axis extends the the right, the positive y-axis extends up, and the negative z-axis extends away from the camera's direction (i.e. OpenGL uses a right-handed system).

↓ *Projection matrix*; maps or *projects* coordinates within a specified range to clip space coordinates with respect to a so-called a *frustum*. There are two types of projection:

- *Orthographic projection*; objects further away are not smaller. Creates a rectangular frustum. This projection doesn't change the w component of coordinates, which means perspective division doesn't do anything, making the clip space coordinates equal to NDC.

```
glm::mat4 proj = glm::ortho(left, right, bottom, top, near, far); \\ Define
    projection matrix with the specified floats
```

- *Perspective projection*; objects further away are smaller. Creates a rectangular pyramid as frustum, that grows in girth away from the camera.

```
glm::mat4 proj = glm::perspective(glm::radians(fov), (float) resWidth / (float)
    resHeight, near, far);
```

4. *Clip space*; the coordinates are clip space coordinates relative to a frustum ranging from -w to w in the frustum. Coordinates outside of the frustum are *clipped*, but if only part of a triangle is clipped it is reconstructed it as one or more triangles that still fit in the frustum. This is the coordinate system in which the vertex shader should output its coordinates.

↓ *Viewport transform*; does *perspective division* to get the clip space coordinates to *normalized device coordinates* (NDC) that range from -1 to 1 by dividing the x, y, and z components by the w component. This is done automatically by OpenGL after the vertex shader. These NDC are mapped to screen coordinates (using the settings of `glViewport()`) and turned into fragments.

5. *Screen space*; the coordinates are pixel coordinates ranging from 0 to your window resolution. These are sent to the rasterizer to be turned into fragments.

4 Buffers

A *buffer* is simply an object that manages a certain piece of GPU memory. Meaning is given to a buffer by binding it to a target, e.g. `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`. OpenGL internally stores a reference to the buffer per target and, based on the target, processes the buffer differently.
The default way of getting data to a buffer is with `glBufferData()`, which copies data from the CPU to the GPU. If a `nullptr` is passed, the memory is only reserved/allocated but not initialized. We can then fill specific regions of a buffer with `glBufferSubData()`, which takes an offset and a size.
If a buffer doesn't need to be resized, use `glBufferStorage()` for performance reasons.
Another way of copying data is by mapping a buffer, useful for directly mapping data to a buffer without first storing it in temporary memory (e.g. copying directly from a file):

```
glBindBuffer(GL_ARRAY_BUFFER, bufferHandle);
void *ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY); // Get pointer
memcpy(ptr, data, sizeof(data)); // Copy data into memory
glUnmapBuffer(GL_ARRAY_BUFFER); // Tell OpenGL we're done with the pointer
```

Lastly, we can copy data from one buffer to another using `glCopyBufferSubData()`, which copies from the buffer bound to the specified read target to the buffer bound to the specified write target. For cases where both buffers happen to have the same target, OpenGL introduced two more buffer targets called `GL_COPY_READ_BUFFER` and `GL_COPY_WRITE_BUFFER`. You don't have to use both of them; you could for example bind one of your equityped buffers to `GL_COPY_READ_BUFFER` and then write to the other bound to its original target.

4.1 VBOs

We get vertices to the vertex shader using *vertex buffer objects* (VBOs), which can store a large number of vertices in the GPU's memory that are send at once, rather than copying one vertex at a time. VBOs is the common term for a normal buffer object when it is used as a source for vertex array data, but it is no different from any other buffer object. Vertices are defined by, or represented as *vertex attributes*; a vertex attribute is an input variable to a shader that is supplied with per-vertex data, for example positions, normals, or texture coordinates. The process of creating a VBO is highly similar to that of creating any other OpenGL object, so it's a pattern you'll see many times:

```
unsigned int vboHandle;
glGenBuffers(1, &vboHandle); // Generate 1 ID and store it in an int
glBindBuffer(GL_ARRAY_BUFFER, vboHandle); // Bind the VBO of type GL_ARRAY_BUFFER and initializes
    it. OpenGL has many other types of buffer; at any point, one buffer per type may be bound. In
    practice, the type doesn't matter for buffers as they're typeless; it is at most a driver or
    code reader hint.
// glCreateBuffers(1, vboHandle) // Only for OpenGL 4.5 and up. Same thing as glGenBuffers +
    glBindBuffer. The reason it doesn't take a target is because buffer objects are not typed and
    no-one cares how it is initialized.
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); // Copies vertex data
    from vertices (a float[]) to buffer currently bound to GL_ARRAY_BUFFER.
```

The last argument specifies how the data should be used and how the GPU can manage it: `GL_STREAM_DRAW` means the data is set once and used at most a few times, `GL_STATIC_DRAW` means the data is set once and used many times, and `GL_DYNAMIC_DRAW` means the data is changed a lot and used many times (it will be stored in memory that allows for faster writes).

Next, we tell OpenGL the structure of the data and what part of our input data goes to which vertex attribute in the vertex shader. Specifying a vertex attributes layout is done as follows (this example is for a *tightly packed* buffer containing only positions, in which there are no other values in between vertices):

```
glEnableVertexAttribArray(0); // Enable the attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0); // Configure input
    attribute with location 0 and associate it with the VBO that is currently bound to
    GL_ARRAY_BUFFER: vboHandle. Each vertex has 3 float values that we don't want to normalize. The
    fifth argument is the stride in bytes; the space between consecutive vertex attributes. The
    last argument is the offset at which the position data begins in the buffer.
```

We can specify multiple vertex attributes in the same (currently bound) buffer, as long as you specify offset and stride in the two `glVertexAttribPointer()` calls accordingly. These can be *interleaved* (i.e. pctpct), which may perform better due to attributes being closely aligned in memory, or *batched* (i.e. ppcctt), which is often a little easier when loading from file.

4.2 EBOs

Indexed drawing avoids having to define duplicate vertices in the VBO in case triangles overlap (which is the case for nearly all meshes) and is much more memory efficient in almost all cases. We define all unique vertices in a VBO, and then reference those indices – called *elements* in OpenGL jargon – in an *element buffer object* (EBO) that defines how the vertices constitute primitives.

```
unsigned int eboHandle;
glGenBuffers(1, &eboHandle);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, eboHandle);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW); // Copy indices
    (an unsigned int[]) to the EBO
```

When we draw from indices, we use `glDrawElements()` rather than `glDrawArrays()`.

4.3 VAOs

In order to avoid binding potentially hundreds of buffer objects and configuring potentially many vertex attributes for each of those objects, OpenGL requires us to configure *vertex array objects* (VAOs). A VAO is basically a list of vertex attributes, numbered from `0` to `GL_MAX_VERTEX_ATTRIBS - 1`, and for each stores its configuration and associated VBO – both set by calls to `glVertexAttribPointer()` – and whether it is enabled – set by calls to `glEnableVertexAttribArray()` / `glDisableVertexAttribArray()`. It also stores at most one EBO. A VAO can be bound just like a VBO, and subsequent vertex attribute calls from that point on will be stored inside the VAO. This means that you only have to configure vertex attribute pointers once and whenever you want to draw the object, you can just bind the corresponding VAO.

```
unsigned int vaoHandle;
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(VAO);
// Create, fill, and configure VBO(s) as we did before
//...
glBindVertexArray(VAO); // Bind VAO whenever you want to draw it
```

When using EBOs, the last element buffer object that gets bound while a VAO is bound, is stored as the VAO's element buffer object (but make sure you don't unbind the EBO before unbinding your VAO, since all `glBindBuffer()` calls with target `GL_ELEMENT_ARRAY_BUFFER` are recorded).

5 Shaders

5.1 Shader objects

Shaders are programs that run many times in parallel on GPU cores, and that allow for configurable stages of the graphics pipeline (which used to be a fixed-function pipeline). There are different types of shaders; for graphics, the most common two types are vertex and fragment shaders. Shaders have to be dynamically compiled at run-time from their source code (the process is identical for vertex and fragment shaders except for `GL_VERTEX_SHADER` vs. `GL_FRAGMENT_SHADER`):

```
unsigned int shaderHandle;
shaderHandle = glCreateShader(GL_VERTEX/FRAGMENT_SHADER); // Create shader object
glShaderSource(shaderHandle, 1, &shaderCode, NULL); // Attach shader source (as a C-string) to
    shader object
glCompileShader(shaderHandle); // Compile shader

int success;
char infoLog[512];
glGetShaderiv(shaderHandle, GL_COMPILE_STATUS, &success); // Check if compilation succeeded

if(!success) {
   glGetShaderInfoLog(shaderHandle, 512, NULL, infoLog); // Retrieve error message
   std::cout << "Shader compilation error: " << infoLog;
}
```

5.2 Shader program objects

Both vertex and fragment shader objects have to be *linked* into a *shader program*:

```
unsigned int shaderProgramHandle;
shaderProgramHandle = glCreateProgram();
glAttachShader(shaderProgram, vertexShaderHandle);
glAttachShader(shaderProgram, fragmentShaderHandle);
glLinkProgram(shaderProgramHandle);

glGetProgramiv(shaderProgramHandle, GL_LINK_STATUS, &success); // Check if linking succeeded
if(!success) {
   glGetProgramInfoLog(shaderProgramHandle, 512, NULL, infoLog); // Retrieve error message
   std::cout << "Shader linking error: " << infoLog;
}

glUseProgram(shaderProgram); // Activate shader program, which all rendering calls henceforth will
    use
glDeleteShader(vertexShader); // Don't need these after linking
glDeleteShader(fragmentShader);
```

And of course, we want to use our program to render stuff with:

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsWhateverIsInOurVAOs(); // More about this in section Rendering
```

5.3 Shaders

Shaders are written in the OpenGL Shading Language (GLSL) GLSL is similar to C(++) but tailored for graphics and matrix and vector manipulations. It supports C(++)'s basic types: `bool`, `int`, `uint`, `float` and `double`, but also has built-in types for vectors and matrices: if `n` is the number of components, ranging from 2 to 4, `vecn` is a float vector, `bvecn` is a boolean vector, `ivecn` is an integer vector, `uvecn` is a unsigned integer vector, and `dvecn` is a double vector. You can use `.x/.r/.s`, `.y/.g/.t`, `.z/.b/.p` and `.w/.a/.q` to access their components. You can even combine them to get new vectors of their components in any order using a feature called *swizzling*, e.g. `.zyyx`. You can also pass vectors to the constructors of other vectors, e.g.:

```
vec2 vector1 = vec3(0.5, 0.7, 0.9);
vec4 vector2 = vec4(vect0r.xy, 0.0, 0.0);
```

A *vertex* is a 3D coordinate, but vertex data can also include other data like color. A *primitive* is the thing that we want to draw and which the vertices constitute, e.g. `GL_POINTS`, `GL_LINES`, `GL_TRIANGLES`, etc. A *fragment* is (all the data for rendering) a pixel that overlaps with a primitive. Note that multiple fragments may be processed for the same pixel, in case multiple primitives overlap that pixel. At least a vertex shader and a fragment shader need to be defined.

Each shader generally looks like this (this happens to be a vertex shader):

```
#version 460 core // Declaration of GLSL version; corresponds with OpenGL version 4.6

in vec3 varIn; // Input variable
out vec2 varOut; // Output variable
uniform vec2 varUni; // Uniform variable
layout(binding = 0, std430) buffer BufferName { uint content[]; }; // Buffer binding

void main() {
   varOut = varIn.xy * varUni; // Output variables after processing
}
```

Wherever the name and type of an output variable matches with an input variable of the next shader stage, OpenGL links those variables (during the shader linking operation) and the variable can be passed along. When specifying a location (by prepending e.g. `layout(location = 0)`), the names can differ (so you can actually put "In" or "Out" in the variable names).
Uniforms are another way of passing (non-buffer/non-array) data to shaders, except they're global (i.e. unique per shader program object) and can be accessed from any shader at any stage, and they keep their value until they're reset or updates. If you don't use a uniform in a particular shader, don't declare it, because OpenGL will silently remove the variable from the compiled version which may cause frustrating errors. To define it in our C++ code:

```
int uniformLocation = glGetUniformLocation(shaderProgramHandle, "varUni"); \\ Query location
glUseProgram(shaderProgramHandle); \\ Uniforms are set on the currently active program, so we need
    to use it first
```

```
glUniform4f(uniformLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

The graphics pipeline is as follows:

1. **Vertex shader**, programmable: takes a vertex and does some processing on its attributes. In vertex shaders, input variables are vertex attributes, which are received straight from the vertex data in the bound VAO. At most `GL_MAX_VERTEX_ATTRIBS` input veriables are allowed (which is at least 16). Input variables/vertex attributes need to be decorated with a location in order to connect them to the correct binding in the VAO, e.g. `layout(location = 0)in vec3 posIn;`.[1]

   It should output coordinates via its predefined `vec4` output variable `gl_Position`, which expects coordinates to be in clip space coordinates.[2] that range from `-w` to `w` (vertices outside that range are not rendered). These will be transformed to screen-space coordinates via the *viewport transform* using the data you provided with `glViewport()`, which are in turn transformed to fragments for the fragment shader. Other user-defined output variables (e.g. for color) are of course also possible.

   When drawing `GL_POINTS` as primitives, you can set vertex-specific point radii by setting the built-in `float gl_PointSize` output variable. For that, you have to `glEnable(GL_PROGRAM_POINT_SIZE)`. You can set a default point size in the host code using `glPointSize()`.

   The built-in integer input variable `gl_VertexID` contains the ID of the current vertex, which is its index when doing indexed drawing (i.e. with `glDrawElements()`) or simply its sequence number when drawing without indices (i.e. with `glDrawArrays()`).

2. **Primitive assembly stage**, fixed: takes a collection of vertices that form the specified primitive and assembles those into the specified primitive shape.

3. **Geometry shader**: takes a set of vertices that form a primitive and generates other shapes by emitting new vertices to form new (or other) primitive(s). Because the shapes are generated dynamically on the GPU, this can be a lot faster than defining these shapes yourself within vertex buffers. This shader is optional and usually left to the default shader. Input and output qualifiers look like:

   ```
   #version 330 core
   layout (points) in;
   layout (line_strip, max_vertices = 2) out;
   ```

   When declaring the type of primitive input we're receiving from the vertex shader, the input layout qualifier (in front of `in`) can take any of the following primitive values:

   - `points` when drawing `GL_POINTS` (at least 1 vertex)
   - `lines` when drawing when drawing `GL_LINES` or `GL_LINE_STRIP` (at least 2 vertices)
   - `lines_adjacency` when drawing `GL_LINES_ADJACENCY` or `GL_LINE_STRIP_ADJACENCY` (at least 4 vertices)
   - `triangles` when drawing `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN` (at least 3 vertices)
   - `triangles_adjacency` when drawing `GL_TRIANGLES_ADJACENCY` or `GL_TRIANGLE_STRIP_ADJACENCY` (at least 6 vertices)

   The output qualifier is any of:

   - `points`
   - `line_strip`; every two adjacent vertices are considered a line/segment along a path, so if you pass n vertices, you will get $n - 1$ connected lines. E.g.: (0, 1), (1, 2), (2, 3), ...
   - `triangle_strip`; every three adjacent vertices will form a triangle, so if you pass n vertices, you will get $n - 2$ triangles. After the first triangle is drawn, each subsequent vertex generates another triangle next to the first triangle. E.g.: (0, 1, 2), (1, 2, 3), (2, 3, 4), ...

   You also need to set the maximum number of vertices it outputs (if you exceed this number, OpenGL won't draw the extra vertices), and of the ways to do that is within the layout qualifier of the `out` keyword.

[1] It is possible to omit the `layout(location = 0)` specifier and query for the attribute locations in your host code via `glGetAttribLocation()`, but this is easier to understand and saves you (and OpenGL) some work.

[2] Technically, *normalized device coordinates* (NDC) are obtained after perspective division on the clip coordinates, but the two terms are sometimes used interchangeably.

To retrieve the output from the previous shader stage, GLSL gives us a built-in variable called `gl_in` that internally (probably) looks something like an interface block:

```
in gl_Vertex
{
   vec4 gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
} gl_in[];
```

i.e., `gl_in` is an array of all vertices of a primitive, where each vertex is structured as above. Because we're operating on sets of vertices, any input data from the vertex shader is always represented as arrays of vertex data in the geometry shader. For example, when we want to pass another vertex attribute, an interface block looks like this in the vertex shader:

```
out vec3 color;
```

and this in the geometry shader:

```
in vec3 color[];
```

We generate new data by calling `EmitVertex()` for as many vertices that a primitive consists of, and then calling `EndPrimitive()`, which collects the emitted vertices into the specified output render primitive. Each time we call `EndPrimitive()`, the vector currently set to `gl_Position` – and the value/vector currently set to any custom defined output variable, e.g. color – are added to the output primitive. This `main()` body translates a single input vertex in two ways and combines the two emitted vertices into a single line strip:

```
void main() {
   gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
   EmitVertex();

   gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
   EmitVertex();

   EndPrimitive();
}
```

4. **Rasterization stage**, fixed: takes a primitive and maps it to the corresponding (overlapping) pixels on the screen, resulting in fragments. Also *clips*, i.e. discards, all fragments that are outside your view, increasing performance.

5. **Fragment shader**, programmable: takes fragment data/attributes which are the output variables from the vertex (or geometry) shader interpolated over the primitive. It should calculates RGBA pixel colors (each component between 0 and 1), which should be output via a user-defined vec4 output variable/attribute. Usually the stage where all the advanced OpenGL effects occur.
   The built-in `vec4 gl_FragCoord` input variable contains the fragment's screen-space coordinates in the x and y components (with (0,0) being the bottom-left corner and whatever you specified in `glViewport()` in the top-right corner) and the fragment depth in the z component. The built-in `float` output variable `gl_FragDepth` allows us to overwrite the depth value in `gl_FragCoord.z` and specify our own value between 0.0 and 1.0. However, doing this disables early depth testing and penalizes performance. OpenGL 4.2 allows to ameliorate this penalty and still do some early depth testing by redeclaring the `gl_FragDepth` variable with a depth condition:

   ```
   layout (<condition>)out float gl_FragDepth;
   ```

   where `<condition>` may be any of the following:

   - `depth_any`; the default value. Early depth testing is disabled.
   - `greater`; you can only make the depth value larger compared to `gl_FragCoord.z`.
   - `depth_less`; you can only make the depth value smaller compared to `gl_FragCoord.z`.
   - `depth_unchanged`; if you write to `gl_FragDepth`, you will write exactly `gl_FragCoord.z`.

   The fragment shader also has the `discard` command, which instructs the shader to stop processing and not output the fragment to the color buffer.

   The built-in `gl_FrontFacing` input variable is a `bool` that is `true` if the current fragment is part of a front face, and

`false` otherwise.

6. **Alpha test and blending stage**, fixed: checks if the resulting fragment is in front or behind other objects and should be discarded accordingly using the depth (and stencil) value and blends with the alpha value.

5.4 Interface blocks

Interface blocks are a handy mechanism for organizing input/output variables in shaders that are similar to structs. In the vertex shader, this looks like:

```
out ExampleBlock
{
   vec2 attr;
} blockOut;
```

If you write to `blockOut.attr`, it can be retrieved in the fragment shader as `blockIn.attr` if you declare the block as:

```
in ExampleBlock
{
   vec2 attr;
} blockIn;
```

As you can see, the *block name* `ExampleBlock` needs to be the same in order for them to get linked, but the *instance name* `blockOut`/`blockIn` can be different.

6 Rendering

When we compiled and linked our shaders, created and configured VAOs and associated VBOs, we can draw their contents. Draw commands are typically placed after `glUseProgram()` and `glBindVertexArray()` commands. Different drawing commands are:

```
glDrawArrays(GL_TRIANGLES, 0, 3); // Draw vertices as triangles directly from VAOs, starting at
    index 0 and drawing 3 vertices
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0); // Draw vertices as triangles using indexed
    drawing from an EBO, drawing 6 vertices/indices of type unsigned int, starting at index 0 in
    the EBO
glDrawArraysInstanced(GL_TRIANGLES, 0, 3); // Draws n instances of all vertices and advances a
    counter gl_InstanceID per iteration
glDrawElementsInstanced(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0, n); // Draws n instances of all
    elements and advances a counter gl_InstanceID per iteration

glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); // Tells OpenGL to draw both faces of polygons/triangles
    as lines, i.e. this enables wireframe mode
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // Tells OpenGL to draw both faces normally. This is the
    default, and running this disables wireframe mode
```

6.1 Instanced rendering

When drawing many instances of the same vertices, *instanced drawing* is much more performant, because we avoid OpenGL's prep work that needs to be done for each draw call (like telling the GPU which buffer to read data from, where to find vertex attributes and all this over the relatively slow CPU to GPU bus). The `...Instanced` versions of essentially repeat the call `n` times:

```
for (int i = 0; i < n ; i++) {
   instanceID = i;
   glDrawArrays(mode, first, count);
}
```

where `instanceID` can be read in shaders as `gl_InstanceID` and can be used to index into buffer object to retrieve data such as world coordinates/offsets or colors. Another option to get data in a shader that is called with an instanced drawing call are *instanced arrays*, which are vertex attributes that are updated per instance rather than per vertex. Where regular vertex attributes are retrieved per vertex and are useful for vertex-specific data, instanced vertex attributes are retrieved per instance and are useful for instance-specific data. Instanced vertex attributes are declared as usual, and there is no

need to index anything with `gl_InstanceID`. To make a vertex attribute instanced, we need to specify when to update the attribute to the next value/vector:

```
glVertexAttribDivisor(2, x); \\ Tell OpenGL to update the vertex attribute bound to index 2 every x
    instances. x = 0 is default and indicates updating per vertex
```

This command[3] is specified along with the other command in Section 4.1.

6.2 Transparency

The alpha channel can be used to draw certain texels transparently. When we want certain parts of a texture entirely invisible, we can `discard` those fragments in the fragment shader when an alpha value is below some threshold. For such textures, set texture wrapping of transparent textures to `GL_CLAMP_TO_EDGE`, or else transparent alpha values will be interpolated with solid values. To allow for partially transparent textures that blend with whatever is behind them (i.e. make use of more than just 0 and 1 alpha values), we have to `glEnable(GL_BLEND)` and tell OpenGL how to blend:

`glBlendFunc(GLenum sfactor, GLenum dfactor)` tells OpenGL how to set the fragment factor f_{fragment} (`sfactor`) and buffer factor f_{buffer} (`dfactor`) colors in the function

$$\boldsymbol{c}_{\text{fragment}} * f_{\text{fragment}} + \boldsymbol{c}_{\text{buffer}} * f_{\text{buffer}} \tag{2}$$

where $\boldsymbol{c}_{\text{fragment}}$ is the fragment color vector output calculated by the fragment shader and $\boldsymbol{c}_{\text{buffer}}$ is the color vector currently in the buffer. There are a plethora of possible options for both arguments, among them:

- `GL_ZERO`: factor is equal to 0.
- `GL_ONE`: factor is equal to 1.
- `GL_SRC_COLOR`: factor is equal to the fragment color vector $\boldsymbol{c}_{\text{fragment}}$.
- `GL_ONE_MINUS_SRC_COLOR`: factor is equal to $1 - \boldsymbol{c}_{\text{fragment}}$.
- `GL_DST_COLOR`: factor is equal to the buffer color vector $\boldsymbol{c}_{\text{buffer}}$.
- `GL_ONE_MINUS_DST_COLOR`: factor is equal to $1 - \boldsymbol{c}_{\text{buffer}}$.
- `GL_SRC_ALPHA`: factor is equal to the alpha component of the fragment color vector $\boldsymbol{c}_{\text{fragment}}$.
- `GL_ONE_MINUS_SRC_ALPHA`: factor is equal to $1 - \alpha$ of $\boldsymbol{c}_{\text{fragment}}$.
- `GL_DST_ALPHA`: factor is equal to the alpha component of the buffer color vector $\boldsymbol{c}_{\text{buffer}}$.
- `GL_ONE_MINUS_DST_ALPHA`: factor is equal to $1 - \alpha$ of $\boldsymbol{c}_{\text{buffer}}$.

`GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA` are often used for `sfactor` and `dfactor`, respectively. Instead of adding the weighted fragment and buffer colors, we can set a different operator, e.g. using `glBlendEquation(GL_FUNC_SUBTRACT)`. When drawing partially transparent textures, you have have to draw them in reversed order of their distance from the camera (and after all opaque objects have been rendered), because else the depth test will discard fragments that should be partially visible. Sorting the objects can be done using, for example, `std::map`.

6.3 Face culling

Each triangle has a *winding order* – the order in which its vertices are specified – and by default, triangles defined with a counter-clockwise winding order are treated as facing the camera.
Face culling is a technnique that discards fragments from triangles that are not facing the camera, saving on performmance if those fragments would've been rendered first (if they were rendered last, they would be discarded by depth testing anyways). It is mostly useful for closed shapes, and is disabled by default, so use `glEnable(GL_CULL_FACE)` to enable it. By default, back-faces are culled, but we could also cull front-faces instead using `glCullFace(GL_FRONT)`. We can even instruct OpenGL to treat clockwise faces as the front-faces using `glFrontFace(GL_CW)`.

[3]There is also the `glVertexBindingDivisor`, which is highly similar. [Ga dit nog maar eens lekker samenvatten: https://stackoverflow.com/questions/50650457/what-is-the-difference-between-glvertexattribdivisor-and-glvertexbindingdivisor]

7 Textures

OpenGL offers multiple options for *texture wrapping*, which dictate what happens when coordinates fall outside the range:

- `GL_REPEAT`: repeats the texture image. Default behavior.
- `GL_MIRRORED_REPEAT`: same as `GL_REPEAT` but mirrors the image with each repeat.
- `GL_CLAMP_TO_EDGE`: clamps outside coordinates to the nearest edge.
- `GL_CLAMP_TO_BORDER`: fills outside coordinates with a user-specified color.

These coordinates are specified – for the currently bound texture – on a per coordinate basis as follows:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
```

For the `GL_CLAMP_TO_BORDER` we need to specify the bordor color:

```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

Another set of options concerns *texture filtering*, which is about how to interpolate *texels* (texture pixels), because texture coordinates can be any floating point values. The most important options are:

- `GL_NEAREST`: nearest neighbour filtering; selects the texel of which the center is closest to the texture coordinate.
- `GL_LINEAR`: bilinear filtering; takes an interpolated value from the texture coordinate's neighboring texels, weighted by distance.

These can be set for both *magnifying* and *minifying* filtering operations, e.g.:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

7.1 Mipmaps

An important technique when rendering a high-resolution texture that may be viewed at varying distances from the camera is called *mipmapping*. In that case, the color of a fragment which contains a large portion of the texture (because it is far away) is difficult to determine, because there are so many texels overlapping the pixel. This will product artifacts and waste memory bandwidth if the object is far away. A mipmap is a collection of texture images based on the same base image, where each subsequent texture is twice as small compared to the previous one. After a certain distance threshold from the camera, OpenGL will pick the next (smaller) mipmap texture that best suits the distance to the object. OpenGL can automatically generate the different mipmap levels using `glGenerateMipmaps(GL_TEXTURE_2D` after texture creation. We can replace the original filtering option for `GL_TEXTURE_MIN_FILTER` (though not for `GL_TEXTURE_MAG_FILTER`, since mipmaps aren't used for magnification) with the following options that define filtering *between* mipmap levels:

- `GL_NEAREST_MIPMAP_NEAREST`: takes the nearest mipmap to match the pixel size and uses nearest neighbor interpolation for texture sampling.
- `GL_LINEAR_MIPMAP_NEAREST`: takes the nearest mipmap level and samples that level using linear interpolation.
- `GL_NEAREST_MIPMAP_LINEAR`: linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples the interpolated level via nearest neighbor interpolation.
- `GL_LINEAR_MIPMAP_LINEAR`: linearly interpolates between the two closest mipmaps and samples the interpolated level via linear interpolation.

7.2 `stb_image.h`

For loading image files from disk, `stb_image.h` is a very popular single header image loading library that is loads most popular file formats. Add the header file into your project and write

```
#define STB_IMAGE_IMPLEMENTATION // Modifies the header file such that it only contains the
    relevant definition source code, effectively turning the header file into a .cpp file
#include "stb_image.h"
```

To load and decompress an image:

```
int width, height, nrChannels; // stb_image.h will store properties in these
stbi_set_flip_vertically_on_load(true); // OpenGL expects the 0.0 coordinate on the y-axis to be on
    the bottom of the image, but images usually have 0.0 at the top of the y-axis
unsigned char *data = stbi_load("image.png", &width, &height, &nrChannels, 0);

if(!data) {
   std::cout << "Failed to load texture" << std::endl;
   return -1;
}
```

7.3 Textue objects

To create a texture with the loaded image data:

```
unsigned int textureHandle;
glGenTextures(1, &textureHandle);
glBindTexture(GL_TEXTURE_2D, textureHandle);
// Set texture wrapping/filtering options here
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data); //
    Specify texture currently bound to target GL_TEXTURE_2D and mipmap level 0 to have format
    GL_RGB and the given width and height. The sixth argument should always be 0 (legacy stuff).
    The last three arguments specify the format, datatype and actual data of the source image.
stbi_image_free(data); // Free image data
```

In order to apply/render this texture on a primitive, the `vec2` texture coordinates (also known as UV coordinates) will have to be passed to the vertex shader as a vertex attribute, and then in the vertex shader forward those to the fragment shader, which will interpolate the coordinates. Texture coordinates range from 0 to 1 in the x and y (and z) axes – which OpenGL calls the s and t (and r) axes when talking about textures. Coordinate (0,0) is in the lower-left corner.
Of course, the texture object itself has to be passed. We do this by assigning the texture to a *texture unit*, which is a location but for textures[4]:

```
glUniform1i(glGetUniformLocation(shaderProgramHandle, "texIn"), 0); // Assign texture unit 0 to the
    sampler uniform (this can be done outside render loop)
// ...inside render loop:
glActiveTexture(GL_TEXTURE0); // Activate texture unit 0 (goes up to at least GL_TEXTURE16, or,
    GL_TEXTURE0 + 16) first before binding texture
glBindTexture(GL_TEXTURE_2D, textureHandle); // Bind the texture to the currently active texture
    unit
glUseProgram(shaderProgramHandle); // Activate shader before setting uniforms
```

We can then sample this texture in the fragment shader using GLSL's built-in `sampler2D` (or `sampler1D` or `sampler3D`) data type, which we declare as a `uniform`:

```
#version 330 core
in vec2 texCoordIn; // Interpolated texture coordinate
out vec4 colorOut;
uniform sampler2D texIn;
// uniform sampler2D tex2In; // A potential texture bound to exture unit 2

void main() {
   colorOut = texture(texIn, texCoordIn); // Sample texture
}
```

If we first render our scene to a framebuffer and then render the resulting texture to a full-screen quad, we can do post-processing by using GLSL's `texture()` function to sample fragments that surround the current fragment using a *kernel* (or convolution matrix), which is a small matrix-like array of values centered on the current pixel that multiplies surrounding pixel values by its kernel values and adds them all together to form a single value.

[4] If you only use one texture in a fragment shader, on most (but not all) drivers you do not need to assign it to a texture unit (avoiding a call to `glUniform1i()` and `glActiveTexture()`), because those drivers bind texture unit `GL_TEXTURE0` by default.)

8 More Buffers

8.1 Color buffer

The *color buffer* is a two-dimensional buffer with the same dimensions as the viewport that stores all the fragment colors: the visual output. It is identified by the constant `GL_COLOR_BUFFER_BIT`, e.g. when clearing it using `glClear()`.

8.2 Depth buffer

The *depth buffer* is a two-dimensional buffer with the same dimensions as the viewport that stores per pixel how far its latest rendered fragment is as a 16, 24 or 32 bit float between 0 and 1. It is identified by the constant `GL_DEPTH_BUFFER_BIT`, e.g. when clearing it using `glClear()`. When depth testing is enabled (it's disabled by default, so use `glEnable(GL_DEPTH_TEST)`), new fragments at that pixel test their `gl_FragCoord`'s z value with the value in the depth buffer after the fragment shader has run[5] and after the stencil test. If the fragment is closer, it is rendered, else it is discarded. The depth value is often obtained as follows:

$$d = \frac{1/z - 1/n}{1/f - 1/n} \tag{3}$$

where z is the depth of the fragment in the view space, n is the depth of the near plane of the view frustum, and f is the depth of the far plane of the view frustum. The reason for not simply using a linear depth buffer ($\doteq \frac{z-n}{f-n}$) is to have more depth resolution near the near plane, where it is mostly needed. The farther away the fragment, the less accurate its depth.

If you only want to use the depth buffer but not write to it, make it read-only with `glDepthMask(GL_FALSE)`. We can also modify the comparison operators it uses for the depth test with `glDepthFunc(GLenum func)`, where `func` is any of:

- `GL_ALWAYS`: the depth test always passes.
- `GL_NEVER`: the depth test never passes.
- `GL_LESS`: passes if the fragment's depth value is less than the stored depth value. Default.
- `GL_EQUAL`: passes if the fragment's depth value is equal to the stored depth value.
- `GL_LEQUAL`: passes if the fragment's depth value is less than or equal to the stored depth value.
- `GL_GREATER`: passes if the fragment's depth value is greater than the stored depth value.
- `GL_NOTEQUAL`: passes if the fragment's depth value is not equal to the stored depth value.
- `GL_GEQUAL`: passes if the fragment's depth value is greater than or equal to the stored depth value.

Z-fighting is a common visual artifact that occurs when there isn't enough depth resolution to decide which of two primitives should be rendered "on top", resulting weird glitchy patterns that are generally more noticeable when objects are further away (because the depth buffer has less precision at larger z-values). Three tricks to prevent z-fighting are to always have some margin between primitives, to set the near plane further away, and to use a higher precision depth buffer.

8.3 Stencil buffer

The *stencil buffer* is a two-dimensional buffer with the same dimensions as the viewport that stores an eight-bit value (so 256 values possible) per pixel. It is identified by the constant `GL_STENCIL_BUFFER_BIT`, e.g. when clearing it using `glClear()`. Stencil testing occurs before depth testing, and similarly renders or discards fragments based on the respective fragment and stencil values. Not all windowing libraries create a stencil buffer by default (but GLFW does). Enable it using `glEnable(GL_STENCIL_TEST)`. It is basically a freeform screen buffer that you can read and write to from the fragment shader however you like, but generally you 1) enable stencil buffer writing to the stencil buffer 2) render objects while updating the stencil buffer 3) disable stencil buffer writing 4) render (other) objects using stencil buffer to test. It can for example be used to draw outlines around (selected) objects, or for rendering textures neatly inside rear-view mirrors.

`glStencilFunc(GLenum func, GLint ref, GLuint mask)` describes whether OpenGL should pass or discard fragments based on the stencil buffer's content. `func` sets the stencil test function, with the same possible options as the depth test's `func`. `ref` specifies the reference value to which the stencil buffer's content is compared. `mask` specifies a mask that is

[5] Most GPUs support a hardware feature called *early depth testing*, which depth tests before the fragment shader. This has one restriction: the fragment shader can't write to `gl_FragCoord.z` anymore.

ANDed with both the reference value and the stored stencil value before the test compares them, and is initially all-1s.

`glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)` describes how to update the stencil buffer. `sfail` specifies the action to take if the stencil test fails, `dpfail` specifies the action to take if the stencil test passes but the depth test fails, and `dppass` specifies the action to take if both the stencil and the depth test pass, with the actions any of the following

- `GL_KEEP`: the currently stored stencil value is kept.
- `GL_ZERO`: the stencil value is set to 0.
- `GL_REPLACE`: the stencil value is replaced with the reference value set with `glStencilFunc()`.
- `GL_INCR`: the stencil value is increased by 1 if it is lower than the maximum value.
- `GL_INCR_WRAP`: same as `GL_INCR`, but wraps it back to 0 as soon as the maximum value is exceeded.
- `GL_DECR`: the stencil value is decreased by 1 if it is higher than the minimum value.
- `GL_DECR_WRAP`: same as `GL_DECR`, but wraps it to the maximum value if it ends up lower than 0.
- `GL_INVERT`: bitwise inverts the current stencil buffer value.

8.4 Framebuffer

A *framebuffer* refers to the combination of color, depth, and stencil buffer. GLFW creates and configures the *default framebuffer* when you create a window, but OpenGL offers the flexibility to define your own framebuffer as another target to render to (called *off-screen rendering*), which can be useful for post-processing (by rendering to a texture and then rendering that texture on a screen-filling quad), mirrors, etc. You always render to (and depth and stencil test from) the currently bound `GL_FRAMEBUFFER`. If you were to omit for example a depth buffer, depth testing operations will not work.

```
unsigned int fboHandle;
glGenFramebuffers(1, &fboHandle);
glBindFramebuffer(GL_FRAMEBUFFER, fbo); // You can also bind to GL_READ_FRAMEBUFFER or
    GL_DRAW_FRAMEBUFFER, for when you only want to read or render to it
```

An *attachment* is memory that acts as a buffer for the framebuffer – think of it as an image – that comes in two forms.

- A *texture attachment* has all rendering commands render to the texture as if it was a normal color/depth/stencil buffer, after which we can use it in our shaders. Just create a texture as usual (although you likely want to pass `NULL` to `glTexImage2D()`'s `data` parameter to only allocate data and not initialize it and you shouldn't care about wrapping or mipmapping) and attach it:

  ```
  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, textureHandle,
      0); // Attach color attachment of type GL_TEXTURE_2D to the texture bound to
      GL_FRAMEBUFFER with mipmap level 0
  ```

  To attach a depth or stencil buffer, specify the texture's `format` and `internalformat` arguments to `glTexImage2D()` as `GL_DEPTH_COMPONENT` or `GL_STENCIL_INDEX` respectively, and specify the attachment argument to `glFramebufferTexture2D()` as `GL_DEPTH_ATTACHMENT` or `GL_STENCIL_ATTACHMENT`, respectively. You can also combine the two into one `GL_DEPTH_STENCIL_ATTACHMENT`, in which case use `GL_DEPTH24_STENCIL8` as `internalformat`, `GL_DEPTH_STENCIL` as `format`, and `GL_UNSIGNED_INT_24_8` as `type`.
  If you want to render your scene to a texture of a different size, you need to call `glViewport()` again with the texture dimensions before rendering to your framebuffer.

- A *renderbuffer object* is similar to a texture attachment but specifically meant as a framebuffer attachment, contrary to the general-purpose texture attachment. All the render data is written directly in a native format into the buffer, avoiding any conversions to texture-specific formats, making it faster to write to and swap from (using `glfwSwapBuffers()`) than texture attachment, but very slow to read from them directly. It is possible to read from them via the slow `glReadPixels()`, which returns a specified area of pixels from the currently bound framebuffer, but not directly from the attachment itself. Use renderbuffers if you don't need to sample their values. For this reason, they are mostly used as depth and stencil attachments, since depth and stencil testing use the native format, but we don't need to read from them/sample them. To create a renderbuffer:

  ```
  unsigned int rboHandle;
  glGenRenderbuffers(1, &rboHandle);
  ```

```
glBindRenderbuffer(GL_RENDERBUFFER, rboHandle);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600); // Specify the
    internal format, width and height of the renderbuffer bound to GL_RENDERBUFFER
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER,
    rbo); // Attach renderbuffer to framebuffer
```

Multiple texture buffers and renderbuffers can be bound interchangeably to the same framebuffer.

To use a framebuffer, it must be complete: it needs at least one (color, depth or stencil) buffer, it needs one color attachment, all attachments need to be complete, and each buffer should have the same number of samples. Check this:

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
   std::cout << "Error: framebuffer incomplete";
   return -1;
}

glBindFramebuffer(GL_FRAMEBUFFER, 0); // Bind the default framebuffer again
glDeleteFramebuffers(1, &fbo); // Delete framebuffer if we don't use it anymore
```

8.5 Uniform buffer objects

Uniform buffer objects are meant for global uniform variables that remain the same over any number of shader programs. Each frame, they're set only once in fixed GPU memory. For example, the projection and view matrices are often the same for all shaders (in contrast with the view matrix), and can therefore be put into a uniform buffer object. When you have many shader programs, this can avoid a lot of uniform setting, improving performance, code readability, and makes changing uniforms easier. Another advantage is that you can set a lot more uniforms in shaders using uniform buffer objects, because OpenGL can set only `GL_MAX_VERTEX_UNIFORM_COMPONENTS` regular uniforms.

```
unsigned int uboHandle;
glGenBuffers(1, &uboHandle);
glBindBuffer(GL_UNIFORM_BUFFER, uboHandle);
glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW); // Allocate 152 bytes of memory
glBufferSubData(GL_UNIFORM_BUFFER, 144, 4, &b) // Fill the uniform buffer (this sets only the
    boolean in ExampleBlock)
glBindBuffer(GL_UNIFORM_BUFFER, 0); // Unbind
glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboHandle); // Link uboHandle to binding point 2
// or, equivalently:
glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBlock, 0, 152); // Link uboHandle to binding
    point 2 with no offset. Allows to have multiple different uniform blocks linked to a single
    uniform buffer object.
```

In order to use this in shaders, we have to link both the uniform buffer object and the *uniform block index* of the shader's uniform block to the same *binding point*. Prior to OpenGL 4.2, we had to do this manually:

```
unsigned int exampleIndex = glGetUniformBlockIndex(shaderProgramHandle, "ExampleBlock"); //
    Retrieve uniform block index of ExampleBlock
glUniformBlockBinding(shaderProgramHandle, exampleIndex, 2); // Link uniform block ExampleBlock to
    binding point 2
```

But since OpenGL 4.2, we can explicitly link uniform blocks to binding points in the shader code, as you'll see next. Every shader can then use the uniform buffer:

```
layout (std140, binding 2) uniform ExampleBlock
{
                // base alignment // aligned offset
   float value; // 4           // 0
   vec3 vector; // 16          // 16 (offset must be multiple of 16 so 4->16)
   mat4 matrix; // 16          // 32 (column 0)
                // 16          // 48 (column 1)
                // 16          // 64 (column 2)
                // 16          // 80 (column 3)
   float values[3]; // 16      // 96 (values[0])
                // 16          // 112 (values[1])
                // 16          // 128 (values[2])
   bool boolean; // 4          // 144
   int integer; // 4           // 148
};
```

The variables can be accessed directly without prefixing the block name. Here we specify `std140` as *uniform block layout*, which specifies what parts of the reserved buffer memory correspond to which uniform variables in the shader. More specifically, it specifies the spacing between – and therefore the offsets of – the variables (the sizes are clearly defined). There are number of possible layouts:

- `shared`; the default layout, optimizes for space as long as the variables' order remains intact. The offsets are defined by the hardware and are consistently shared between multiple programs, hence the name. The downside is that you'd need to query the offsets using `glGetUniformIndices()`.
- `packed`; similar to `shared`, except allows the compiler to optimize uniform variables away from the uniform block, which may differ per shader and hence there is no guarantee that the layout remains the same between programs (not shared).
- `std140`; standardizes the variable offsets, which allows for figuring them out manually. Each variable has a predefined *base alignment* equal to the space it takes including padding. Some base aligments are: `int`, `float`, `bool` are four bytes; `vec2` is eight byte; `vec3`, `vec4` are sixteen bytes[6]; each element in a vectors has a base alignment of a `vec4`, etc. The aligned offset of a variable must be equal to a multiple of its base alignment.
- `std430`

9 DSA

Even though OpenGL was designed as a finite state machine, OpenGL 4.5 introduced *direct state access* (DSA), which allows for modifying objects without binding and unbinding them to the context. Up until now we have exclusively used non-DSA function to illustrate how OpenGL operates, but in general, use DSA functions wherever possible. You not only avoid a significant number of binds/unbinds/state switches (which can actually have performance impact), but it leads to much cleaner and more readable code, because you directly indicate the object you wanna operate on. In addition, some (kinda bad) functions have just not been ported to DSA, which is a nice way to avoid using them. Only draw calls still require binding a VAO.

DSA functions can be recognized in a couple of ways. They always share a manpage with their non-DSA counterpart, where the more verbose function name is always the DSA version. The DSA function name either fully spells the relevant object type (e.g. `Texture` instead of `Tex`) or prepends `Named` to it. There are also the many `glCreate...()` functions that are the DSA counterparts to `glGen...()` followed by `glBind...()`.[7]

10 Leftovers

`glGet<type>v()` functions retrieve the value of a specified OpenGL parameter, e.g.:

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes); // Retrieve value
std::cout << "Max vertex attribs: " << nrAttributes; // Print value
```

`glfwGetTime()` retrieves the running time of the application in seconds.

11 Debugging

Debugging execution on a GPU is more difficult than debugging CPU code; there is no console to output text to, no breakpoints to set on GLSL code, and no way of easily checking the state of GPU execution. There are two major ways of retrieving debug info from the GPU: `glGetError()`, and debug output.

11.0.1 `glGetError()`

`glGetError()` is the old and more annoying way. Whenever you use OpenGL incorrectly, an OpenGL error is generated and stored in a queue, until the error is retrieved. `glGetError()` fetches the next error in the queue and removes it. If the error queue is empty, it will return `GL_NO_ERROR`, if not, it will return one of

[6] Implementations sometimes get the layout wrong for `vec3` components, so either manually pad your `vec3` variables or avoid them altogether.

[7] However, when mixing DSA with non-DSA functions, don't forget to bind the relevant object anyways, because that's what the non-DSA functions need. Try to avoid this, though.

- `GL_INVALID_ENUM`: an enumeration parameter is not legal.
- `GL_INVALID_VALUE`: a value parameter is not legal.
- `GL_INVALID_OPERATION`: the state for a command is not legal for its given parameters.
- `GL_STACK_OVERFLOW`: a stack pushing operation causes a stack overflow.
- `GL_STACK_UNDERFLOW`: when a stack popping operation occurs while the stack is at its lowest point.
- `GL_OUT_OF_MEMORY`: a memory allocation operation cannot allocate (enough) memory.
- `GL_INVALID_FRAMEBUFFER_OPERATION`: reading or writing to a framebuffer that is not complete.

OpenGL's documentation lists for each function which error codes it can throw, and when. To fetch all of the errors currently in the queue, you would need to loop:

```
GLenum err;
while((err = glGetError()) != GL_NO_ERROR) {
// Process err.
}
```

This only prints error numbers. It often makes sense to write a small helper function to easily print out the error strings together with where the error check function was called:

```
void glAssert(const char *file, int line) {
GLenum errorCode;
while ((errorCode = glGetError()) != GL_NO_ERROR) {
   std::string errorString;
   switch (errorCode)
   {
      case GL_INVALID_ENUM:         errorString = "INVALID_ENUM"; break;
      case GL_INVALID_VALUE:        errorString = "INVALID_VALUE"; break;
      case GL_INVALID_OPERATION:    errorString = "INVALID_OPERATION"; break;
      case GL_STACK_OVERFLOW:       errorString = "STACK_OVERFLOW"; break;
      case GL_STACK_UNDERFLOW:      errorString = "STACK_UNDERFLOW"; break;
      case GL_OUT_OF_MEMORY:        errorString = "OUT_OF_MEMORY"; break;
      case GL_INVALID_FRAMEBUFFER_OPERATION: errorString = "INVALID_FRAMEBUFFER_OPERATION"; break;
   }
   std::cout << errorString << " | " << file << " (" << line << ")" << std::endl;
}
return errorCode;
}
#define glCheckError() glCheckError_(__FILE__, __LINE__)
```

`__FILE__` and `__LINE__` are preprocessor directive variables that get replaced during compile time with the respective file and line they were compiled in. The downside to this method of debugging is that you need to spam `glAssert()`s everywhere in your code where there could be an error in the queue, so that's after almost every API call. If, for example, you only `glAssert()` at the end of each frame, you wouldn't know where the error came from.

11.0.2 Debug output

Debug output is an OpenGL extension that became part of core OpenGL since version 4.3. With it, OpenGL itself will directly send an error or warning message – what they refer to as *message events* – to the user with a lot more details compared to `glCheckError()`, and at the exact location where the message event occurs. This also avoids the need to clutter your code with many `glAssert()`s.

Messages consist of:

- a `GLenum` indicating the source that produced the message. Any of `GL_DEBUG_SOURCE_API`, `GL_DEBUG_SOURCE_WINDOW_SYSTEM`, `GL_DEBUG_SOURCE_SHADER_COMPILER`, `GL_DEBUG_SOURCE_THIRD_PARTY`, `GL_DEBUG_SOURCE_APPLICATION`, `GL_DEBUG_SOURCE_OTHER`.
- a `GLenum` indicating the type of message. Any of `GL_DEBUG_TYPE_ERROR`, `GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR`, `GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR`, `GL_DEBUG_TYPE_PORTABILITY`, `GL_DEBUG_TYPE_PERFORMANCE`, `GL_DEBUG_TYPE_MARKER`, `GL_DEBUG_TYPE_PUSH_GROUP`, `GL_DEBUG_TYPE_POP_GROUP`, `GL_DEBUG_TYPE_OTHER`.
- a `GLenum` indicating the severity. Any of `GL_DEBUG_SEVERITY_HIGH`, `GL_DEBUG_SEVERITY_MEDIUM`, `GL_DEBUG_SEVERITY_LOW`, `GL_DEBUG_SEVERITY_NOTIFICATION`.

- a `GLuint` containing the ID.
- a null-terminated string describing the message.

You can enable debug output in two ways:

- Call `glEnable(GL_DEBUG_OUTPUT`. However, the OpenGL implementation may not generate messages this way. Messages are only guaranteed in a...
- Create, or rather request, a debug context from GLFW by calling `glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, true)` before calling `glfwCreateWindow()`. To check whether GLFW satisfied the request:

  ```
  int flags; glGetIntegerv(GL_CONTEXT_FLAGS, &flags);
  if (flags & GL_CONTEXT_FLAG_DEBUG_BIT)
  {
  // initialize debug output
  }
  ```

  Debug output starts enabled in a debug context. This context can be significantly slower compared to a non-debug context, so when working on optimizations or releasing your application you want to remove GLFW's debug request hint.

We then pass OpenGL an error logging function callback (similar to GLFW's input callbacks), which is called whenever debug output detects an OpenGL error:

```
// OpenGL expects this exact callback function prototype
void APIENTRY debugOutputCallback(GLenum source,
                  GLenum type,
                  unsigned int id,
                  GLenum severity,
                  GLsizei length,
                  const char *message,
                  const void *userParam)
{

std::cout << "---------------" << std::endl;
std::cout << "Debug message (" << id << "): " << message << std::endl;

switch (source)
{
   case GL_DEBUG_SOURCE_API:    std::cout << "Source: API"; break;
   case GL_DEBUG_SOURCE_WINDOW_SYSTEM: std::cout << "Source: Window System"; break;
   case GL_DEBUG_SOURCE_SHADER_COMPILER: std::cout << "Source: Shader Compiler"; break;
   case GL_DEBUG_SOURCE_THIRD_PARTY: std::cout << "Source: Third Party"; break;
   case GL_DEBUG_SOURCE_APPLICATION: std::cout << "Source: Application"; break;
   case GL_DEBUG_SOURCE_OTHER:  std::cout << "Source: Other"; break;
} std::cout << std::endl;

switch (type)
{
   case GL_DEBUG_TYPE_ERROR:     std::cout << "Type: Error"; break;
   case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR: std::cout << "Type: Deprecated Behaviour"; break;
   case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR: std::cout << "Type: Undefined Behaviour"; break;
   case GL_DEBUG_TYPE_PORTABILITY: std::cout << "Type: Portability"; break;
   case GL_DEBUG_TYPE_PERFORMANCE: std::cout << "Type: Performance"; break;
   case GL_DEBUG_TYPE_MARKER:    std::cout << "Type: Marker"; break;
   case GL_DEBUG_TYPE_PUSH_GROUP: std::cout << "Type: Push Group"; break;
   case GL_DEBUG_TYPE_POP_GROUP: std::cout << "Type: Pop Group"; break;
   case GL_DEBUG_TYPE_OTHER:     std::cout << "Type: Other"; break;
} std::cout << std::endl;

switch (severity)
{
   case GL_DEBUG_SEVERITY_HIGH: std::cout << "Severity: high"; break;
   case GL_DEBUG_SEVERITY_MEDIUM: std::cout << "Severity: medium"; break;
   case GL_DEBUG_SEVERITY_LOW: std::cout << "Severity: low"; break;
   case GL_DEBUG_SEVERITY_NOTIFICATION: std::cout << "Severity: notification"; break;
} std::cout << std::endl;
std::cout << std::endl;
}
```

With this callback, we can initialize debug output:

```
if (flags & GL_CONTEXT_FLAG_DEBUG_BIT)
{
glEnable(GL_DEBUG_OUTPUT);
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS); // Tells OpenGL to directly call the callback function the
    moment an error occurred
glDebugMessageCallback(debugOutputCallback, nullptr); // Register callback
glDebugMessageControl(GL_DEBUG_SOURCE_API, GL_DEBUG_TYPE_ERROR, GL_DEBUG_SEVERITY_HIGH, 0, nullptr,
    GL_TRUE); // Only show messages from the OpenGL API that are errors and have a high severity
}
```

If a callback is not registered, then the messages are stored in a log. The last command, `glDebugMessageControl()`, allows us to filter messages. It has two modes in which it can be used. In the first mode, as used above, it selects messages based on source (first argument), type (second argument), and severity (third argument), and then – depending on the last argument – those messages are set to be either emitted while all others are ignored or ignored while all others are emitted. For any of the first three arguments, you can use `GL_DONT_CARE` as a wildcard to select any messages at that level. The second mode of selecting messages uses (a pointer to) an array of specific message IDs to whitelist or blacklist (again, depending on the last argument), and also requires that arrays size. In this mode, the severity specifier must be `GL_DONT_CARE`. You can use any number of `glDebugMessageControl()` calls. This is a pretty essential call, since the majority of message events are not that interesing (e.g. "you created a buffer").

By setting a breakpoint in `debugOutputCallback()` at a specific error type (or at the top of the function if you don't care), you can figure out the exact line or call an error occurred.

You can also push your own messages to the debug output system:

```
glDebugMessageInsert(GLenum source, GLenum type, GLuint id, GLenum severity, GLsizei length, const
    char *message);
```

This is especially useful if you're hooking into other application or OpenGL code that makes use of a debug output context. Other developers can quickly figure out any reported bug that occurs in your custom OpenGL code. The `source` must be either `GL_DEBUG_SOURCE_APPLICATION` or `GL_DEBUG_SOURCE_THIRD_PARTY`, which will never be used for implementation-generated messages. The `id` is purely for the benefit of the user and may be any unsigned 32-bit integer value. `length` indicates the length of the message, and may be negative, but then that message must be `NULL`-terminated.

11.0.3 Debugging GLSL

GLSL report syntax errors when compiling, but debugging runtime GLSL is difficult, as there are no breakpoints or easy printing. You need to resort to methods such as sending all relevant variables in a shader directly to the fragment shader's output channel and inspecting the visual results. You can also display a framebuffer's content(s) in some pre-defined region of your screen, but this only works on texture attachments, not render buffer objects.

Because each vendor-specific driver has its own quirks and tidbits in how they enforce the OpenGL specification, if you want to be sure your shader code runs on all kinds of machines, you can directly check your shader code against the official specification using OpenGL's GLSL reference compiler. The GLSL lang validator can easily check your shader code by passing it as the binary's first argument; if it detects no error, it returns no output.

Lastly, there are third party applications that often inject themselves in the OpenGL drivers and are able to intercept all kinds of OpenGL calls, allowing you to profile OpenGL function usage, find bottlenecks, inspect buffer memory, and display textures and framebuffer attachments. Examples are RenderDoc (open source tool that capture one or more frames at the executable's current state and shows all OpenGL commands, buffer storage, and textures in use), CodeXL (for profiling), and NVIDIA Nsight (renders an overlay GUI system from within your application with a large host of run-time statistics regarding GPU usage and the frame-by-frame GPU state, but only works on NVIDIA cards).