Real time rendering with OpenGL

Gaëtan Blaise-Cazalet - Head of programming section

Display a triangle on your computer

Initialize glfw and glew

Boilerplate code to init everything :

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <stdio.h>

int main () {

  if (!glfwInit ()) {
    fprintf (stderr, "ERROR: could not start GLFW3\n");
    return 1;
  }

  GLFWwindow* window = glfwCreateWindow (640, 480, "Hello Triangle", NULL, NULL);
  if (!window) {
    fprintf (stderr, "ERROR: could not open window with GLFW3\n");
    glfwTerminate();
    return 1;
  }
  glfwMakeContextCurrent (window);

  glewExperimental = GL_TRUE;
  glewInit ();

  // get version info
  const GLubyte* renderer = glGetString (GL_RENDERER);
  const GLubyte* version = glGetString (GL_VERSION);
  printf ("Renderer: %s\n", renderer);
  printf ("OpenGL version supported %s\n", version);

  // tell GL to only draw onto a pixel if the shape is closer to the viewer
  glEnable (GL_DEPTH_TEST); // enable depth-testing
  glDepthFunc (GL_LESS); // depth-testing interprets a smaller value as "closer"

  /* CODE */

  /* END CODE */

  // close GL context and any other GLFW resources
  glfwTerminate();
  return 0;
}
```

Start GL context and O/S window using the GLFW helper library

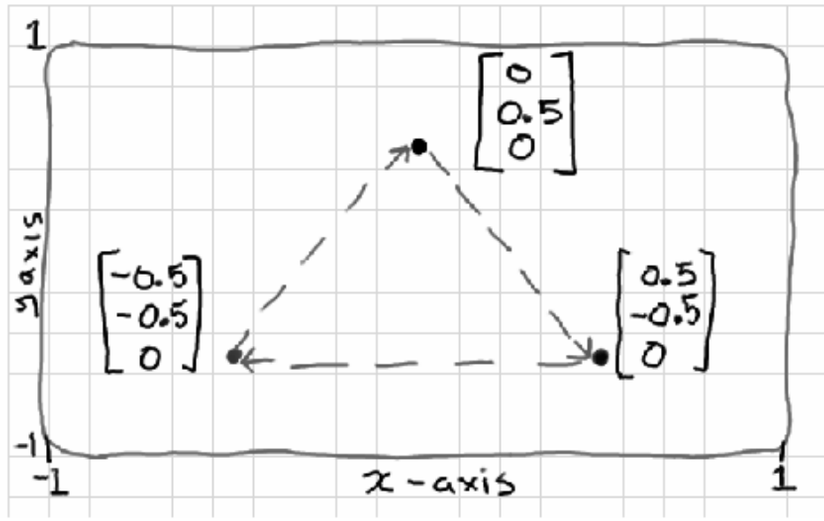
Create window

Start GLEW then config OpenGL

We will code here

Define a triangle in the vertex buffer

A triangle in screen view coordinate system



A triangle in a VBO (Vertex Buffer Object)

We pack all the triangle points, 3 by 3 in clockwise order, in a float array. Then we copy this array onto the graphics card using a VBO.

```
...
/* CODE */
GLfloat points[] = {
   0.0f,  0.5f,  0.0f,
   0.5f, -0.5f,  0.0f,
  - 0.5f, -0.5f,  0.0f
};
GLuint vbo = 0;
glGenBuffers (1, &vbo);
glBindBuffer (GL_ARRAY_BUFFER, vbo);
glBufferData (GL_ARRAY_BUFFER, sizeof (points), points, GL_STATIC_DRAW);
```

Generate an empty buffer
Set it as the current buffer in OpenGL state machine by binding
Copy the points on the bound buffer

A mesh in the VAO (Vertex Attribute Object)

Most meshes will use a collection of one or more vertex buffer objects to hold vertex points, texture-coordinates, vertex normals, etc. In older GL implementations we would have to bind each one, and define their memory layout, every time that we draw the mesh.

To simplify that, we have a thing called the vertex attribute object (VAO), which remembers all of the vertex buffers that you want to use, and the memory layout of each one. We set up the vertex array object once per mesh. When we want to draw, all we do then is bind the VAO and draw.

```
...
GLuint vao = 0;
glGenVertexArrays (1, &vao);
glBindVertexArray (vao);
glEnableVertexAttribArray (0);
glBindBuffer (GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

Generate a new VAO
Bind in focus in the OpenGL state machine
We have one VBO, so the attributo location will be 0
Define the layout for attribute 0, the variable are made of 3

Setting up the shaders

The shaders will define how to draw our shape from the VAO. We need at least a vertex shader (shader the points' positions of the mesh) and a fragment shader (shader to compute the pixels). For now shaders are simple, so we write them in a text variable.

Vertex and fragment shader

We pack all the triangle points, 3 by 3 in clockwise order, in a float array. Then we copy this array onto the graphics card using a VBO.

```
...
const char* vertex_shader =
"#version 410\n"
"in vec3 vp;"
"void main () {"
"  gl_Position = vec4 (vp, 1.0);"
"}";

const char* fragment_shader =
"#version 410
"
"out vec4 frag_colour;"
"void main () {"
"  frag_colour = vec4 (0.5, 0.5, 0.0, 1.0);"
"}";
```

- OpenGL Version
- Input variable is a vec3, which match the position of a point from the VAO
- We output gl_Position, which is the point with a 1 as a 4th coordinate (homogeneous w)
- We output a color in RGBA format, for each of the pixels that belong to the shape
- The color will be a nice golden yellow

Using the shaders

Before using the shaders we have to load the strings into a GL shader, and compile them. Then we put compiled shaders into a single program.

```
...
GLuint vs = glCreateShader (GL_VERTEX_SHADER);
glShaderSource (vs, 1, &vertex_shader, NULL);
glCompileShader (vs);
GLuint fs = glCreateShader (GL_FRAGMENT_SHADER);
glShaderSource (fs, 1, &fragment_shader, NULL);
glCompileShader (fs);

GLuint shader_programme = glCreateProgram ();
glAttachShader (shader_programme, fs);
glAttachShader (shader_programme, vs);
glLinkProgram (shader_programme);
```

- Compile vextex shader
- Compile fragment shader
- Create empty program
  Attach the shaders
- Link program

Drawing the triangle

In real time rendering, we clear the screen and redraw the scene each frame.

```
...
glUseProgram (shader_programme);

while (!glfwWindowShouldClose (window)) {
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBindVertexArray (vao);
  glDrawArrays (GL_TRIANGLES, 0, 3);

  glfwPollEvents ();

  glfwSwapBuffers (window);
}
/* END CODE */
```

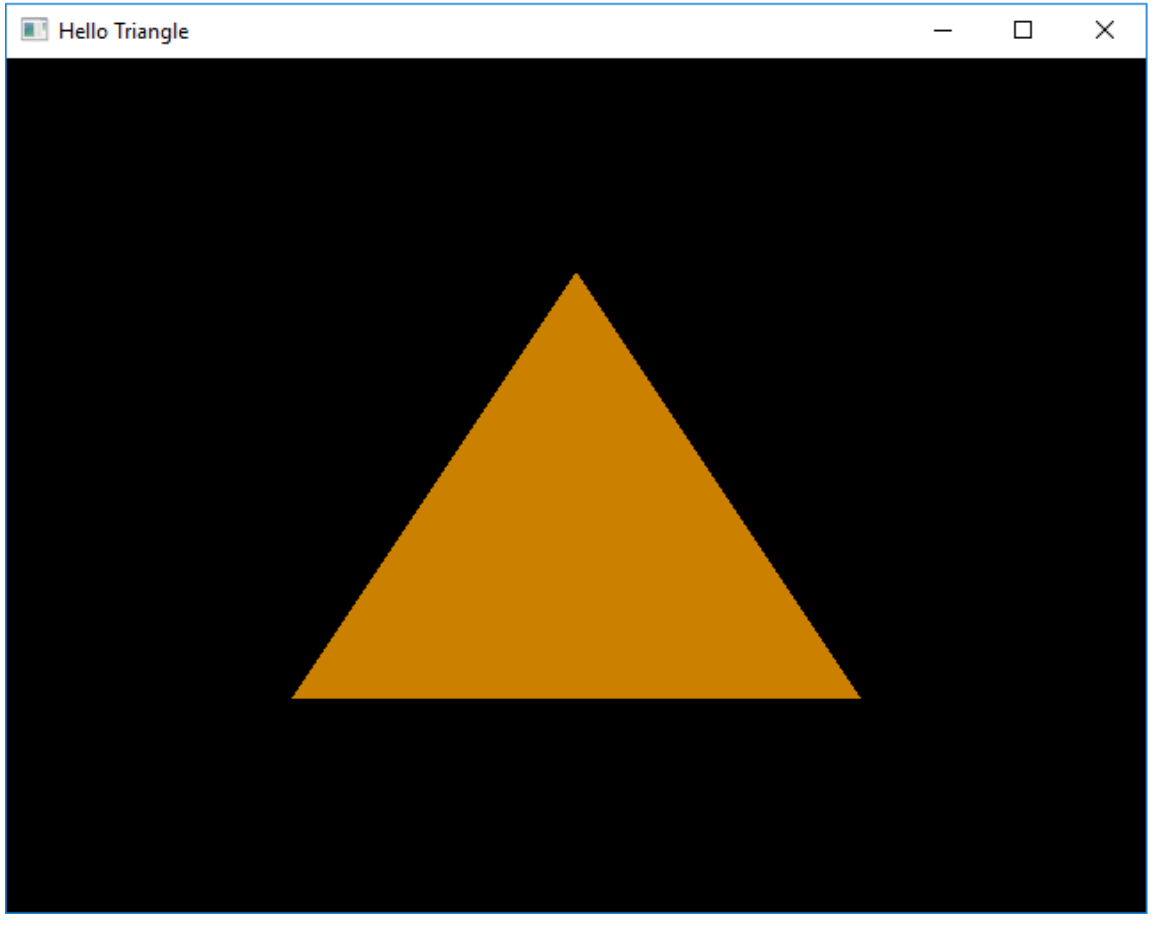
Use program

Wipe the screen

draw points 0-3 from the currently bound VAO with current in-use shader

update other events like input handling

put the drawing on the display (double buffering)



Exercices

- Load the shader strings from text files called test.vert and test.frag (a naming convention is handy).
- Change the colour of the triangle in the fragment shader.
- Try to move the shape in the vertex shader e.g. vec4 (vp.x, vp.y + 1.0, vp.z, 1.0);
- Try to add another triangle to the list of points and make a square shape. You will have to change several variables when setting up the buffer and drawing the shape. Which variables do you need to keep track of for each triangle? (hint: not much...).
- Try drawing with GL_LINE_STRIP or GL_LINES or GL_POINTS instead of triangles. Does it put the lines where you expect? How big are the points by default?
- Try changing the background colour by using glClearColor () before the rendering loop. Something grey-ish is usually fairly neutral; 0.6f, 0.6f, 0.8f, 1.0f.
- Try creating a second VAO, and drawing 2 shapes (remember to bind the second VAO before drawing again).
- Try creating a second shader programme, and draw the second shape a different colour (remember to "use" the second shader programme before drawing again).

Extended initialisation

A Logging system

Debuging graphical software is difficult, so we will start by implementing a Logging system.

Use cases

- Our logging system will log several log lines in a .log file
- It will have different level of logging
- It will have a config system that will allow us to choose which logging precision we want, and choose if we want to restart the logging
- Here is the logging use case :

```
LOG(Info) << "OpenGL version supported " << version;
```

- Here is the config setup :

```
LogConfig LOG_CONFIG = {};
LOG_CONFIG . reporting_level = Debug;
LOG_CONFIG . restart = true;
```

A Logging system

Header

```
// Inspired from http://www.drdobbs.com/cpp/logging-in-c/201804215
// and https://github.com/zuhd-org/easyloggingpp

#ifndef LOG_H
#define LOG_H

#include <time.h>
#include <stdio.h>
#include <fstream>
#include <sstream>

#define GL_LOG_FILE "gl.log"

enum LogLevel { Error, Warning, Info, Debug };

struct LogConfig {
   LogLevel reporting_level = Info;
   bool restart = false;
};

extern LogConfig LOG_CONFIG;

class Log {
public:
   Log();
   virtual ~Log();
   std::ostringstream& get(LogLevel level = Info);
   static void restart();

private:
   std::ostringstream os;
   static std::ofstream file;

   std::string get_label(LogLevel type);

   Log(const Log&);
   Log& operator=(const Log&);
};

#define LOG(level) \
if (level > LOG_CONFIG . reporting_level) ; \
else Log().get(level)

#endif
```

Log file name

Log config struct

We will use this global variable to store the config

restart will clear the log file, static function

restart need the file to be static too

We redefine basic operations as a security

This macro will allow us to optimize our logging command and give us syntactic sugar

A Logging system

Source

```
#include "Log.h"

Log::Log() {
    file.open(GL_LOG_FILE, std::fstream::app);
}

Log::~Log()
{
    os << std::endl;
    file << os.str();
    printf(os.str().c_str());
    os.clear();
    file.close();
}

std::ofstream Log::file;

void Log::restart() {
    file.open(GL_LOG_FILE, std::fstream::trunc);
    file.close();
}

std::ostringstream& Log::get(LogLevel level) {
    if(!file) return os;

    // Log
    time_t now;
    struct tm * timeinfo;
    char date[19];

    time(&now);
    timeinfo = localtime (&now);
    strftime (date, 19, "%y-%m-%d %H:%M:%S", timeinfo);

    // Log
    os << date << " " << get_label(level) << ": \t";
    return os;
}

std::string Log::get_label(LogLevel type) {
    std::string label;
    switch(type) {
        case Debug:     label = "DEBUG"; break;
        case Info:      label = "INFO "; break;
        case Warning:   label = "WARN "; break;
        case Error:     label = "ERROR"; break;
    }
    return label;
}
```

Log constructor opens file in append mode

Log destructor write the content of the buffer in the file and in the standard output (console)

Reset the file by opening in truncate mode and closing

Logging GLFW errors

GLFW triggers its own errors. Here is how to log them.

- Before the main function, define the LOG_CONGIG struc and a callback function

```
LogConfig LOG_CONFIG = {};
void glfw_error_callback(int error, const char* description) {
    LOG(Error) << "GLFW error code " << error << " | msg: " << description;
}
```

- Configure the logging right after declaring the window :

```
    LOG_CONFIG . reporting_level = Debug;
    LOG_CONFIG . restart = true;
    if(LOG_CONFIG . restart) {
        Log::restart();
    }
```

- Then initialize GLFW with a proper logging. Set the GLFW error callback function

```
    LOG(Info) << "Starts GLFW";
    glfwSetErrorCallback(glfw_error_callback);
    if (!glfwInit()) {
        LOG(Error) << "Could not start GLFW3";
        return -1;
    }
```

- Convert former printf to info or error logs

More options

We will set the version of openGL we want to use, with the "forward compatible, core profile" context, which forbid to use deprecated features.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

And we want a 4-samples anti aliasing

```
glfwWindowHint (GLFW_SAMPLES, 4);
```

Get graphics capabilities info

We want to get graphics capabilities. We will create a function that we will use just after GLEW init.

```
void log_gl_params () {
  GLenum params[] = {
     GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS,
     GL_MAX_CUBE_MAP_TEXTURE_SIZE,
     GL_MAX_DRAW_BUFFERS,
     GL_MAX_FRAGMENT_UNIFORM_COMPONENTS,
     GL_MAX_TEXTURE_IMAGE_UNITS,
     GL_MAX_TEXTURE_SIZE,
     GL_MAX_VARYING_FLOATS,
     GL_MAX_VERTEX_ATTRIBS,
     GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS,
     GL_MAX_VERTEX_UNIFORM_COMPONENTS,
     GL_MAX_VIEWPORT_DIMS,
     GL_STEREO,
  };
  const char* names[] = {
     "GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS",
     "GL_MAX_CUBE_MAP_TEXTURE_SIZE",
     "GL_MAX_DRAW_BUFFERS",
     "GL_MAX_FRAGMENT_UNIFORM_COMPONENTS",
     "GL_MAX_TEXTURE_IMAGE_UNITS",
     "GL_MAX_TEXTURE_SIZE",
     "GL_MAX_VARYING_FLOATS",
     "GL_MAX_VERTEX_ATTRIBS",
     "GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS",
     "GL_MAX_VERTEX_UNIFORM_COMPONENTS",
     "GL_MAX_VIEWPORT_DIMS",
     "GL_STEREO",
  };
  LOG(Info) << "-----------------";
  LOG(Info) << "GL Context Params:";
  // integers - only works if the order is 0-10 integer return types
  for (int i = 0; i < 10; i++) {
     int v = 0;
     glGetIntegerv (params[i], &v);
     LOG(Info) << names[i] << " " << v;
  }
  // others
  int v[2];
  v[0] = v[1] = 0;
  glGetIntegerv (params[10], v);
  LOG(Info) << names[10] << " " << v[0] << " " << v[1];
  unsigned char s = 0;
  glGetBooleanv (params[11], &s);
  LOG(Info) << names[11] << " " << (unsigned int)s;
  LOG(Info) << "-----------------";
}
```

Decipher graphics capabilities info

How to read our log result.

The log will output :

```
19-05-09 16:39:42 INFO :        ----------------
19-05-09 16:39:42 INFO :        GL Context Params:
19-05-09 16:39:42 INFO :        GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS 192
19-05-09 16:39:42 INFO :        GL_MAX_CUBE_MAP_TEXTURE_SIZE 16384
19-05-09 16:39:42 INFO :        GL_MAX_DRAW_BUFFERS 8
19-05-09 16:39:42 INFO :        GL_MAX_FRAGMENT_UNIFORM_COMPONENTS 4096
19-05-09 16:39:42 INFO :        GL_MAX_TEXTURE_IMAGE_UNITS 32
19-05-09 16:39:42 INFO :        GL_MAX_TEXTURE_SIZE 16384
19-05-09 16:39:42 INFO :        GL_MAX_VARYING_FLOATS 64
19-05-09 16:39:42 INFO :        GL_MAX_VERTEX_ATTRIBS 16
19-05-09 16:39:42 INFO :        GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS 32
19-05-09 16:39:42 INFO :        GL_MAX_VERTEX_UNIFORM_COMPONENTS 4096
19-05-09 16:39:42 INFO :        GL_MAX_VIEWPORT_DIMS 16384 16384
19-05-09 16:39:42 INFO :        GL_STEREO 0
19-05-09 16:39:42 INFO :        ----------------
```

This tells me that my shader programmes can use 192 different textures each. I can access 96 different textures in the vertex shader, and 96 more in the fragment shader. My laptop can support only 8 textures, so if I want to write programmes that run nicely on both you can see that I would make sure that they don't use more than 8 textures at once. In theory my texture resolution can be up to 16384x16384, but multiply this by 4 bytes (for red+green+blue+alpha channels), and we see that 16k textures will use up my memory pretty quickly. I might get away with 8224x8224x4 bytes.

The "max uniform components" means that I can send 4096 floats to each shader. Each matrix is going to use 32 floats. So we can say that we have plenty of space there.

"Varying" floats are those sent from the vertex shader to the fragment shaders. Usually these are vectors, so we can say that we can send 16 4d vectors between shaders. Varyings are computationally expensive, and some devices still have a limit of 16 floats (4 vectors), so it's best to keep these to a minimum.

Vertex attributes are variables loaded from a mesh e.g. vertex points, texture coordinates, normals, per-vertex colours, etc. OpenGL means 4d vectors here. I would struggle to come up with more than about 6 useful per-vertex attributes, so no problem here. Draw buffers is useful for more advanced effects where we want to split the output from our rendering into different images - we can split this into 8 parts. And, sadly, my video card doesn't support stereo rendering.

GL state machine

In the first lesson, we told about the OpenGL State Machine. This means that once we set a state (like transparency, for example), it is then globally enabled for all future drawing operations, until we change it again. In GL parlance, setting a state is referred to as "binding" (for buffers of data), "enabling" (for rendering modes), or "using" for shader programmes.

The state machine can be very confusing. Lots of errors in OpenGL programmes come from setting a state by accident, forgetting to unset a state, or mixing up the numbering of different OpenGL indices. Some of the most useful state machine variables can be fetched during run-time. You probably don't need to write a function to log all of these states, but keep in mind that, if it all gets a bit confusing, you can check individual states.

FPS counter

We will add a FPS counter at the top of our window, which will be updated each quarter of seconds.

```
double previous_seconds;
int frame_count;

void update_fps_counter(GLFWwindow* window) {
  double current_seconds;
  double elapsed_seconds;

  current_seconds = glfwGetTime();
  elapsed_seconds = current_seconds - previous_seconds;
  /* limit text updates to 4 per second */
  if (elapsed_seconds > 0.25) {
    previous_seconds = current_seconds;
    char tmp[128];
    double fps = (double)frame_count / elapsed_seconds;
    sprintf (tmp, "OpenGL @ fps: %.2f", fps);
    glfwSetWindowTitle (window, tmp);
    frame_count = 0;
  }
  frame_count++;
};
```

We use this function in the loop we created earlier.

```
  glUseProgram (shader_programme);
  // Loop until the user closes the window
  while (!glfwWindowShouldClose(window))
  {
     // Display FPS
     update_fps_counter(window);
     // wipe the drawing surface clear
     glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

     glBindVertexArray (vao);
     // draw points 0-3 from the currently bound VAO with current in-use shader
     glDrawArrays (GL_TRIANGLES, 0, 3);
     // update other events like input handling
     glfwPollEvents ();
     if (GLFW_PRESS == glfwGetKey (window, GLFW_KEY_ESCAPE)) {
        glfwSetWindowShouldClose (window, 1);
     }

     // Logic

     // put the stuff we've been drawing onto the display
     glfwSwapBuffers (window);
```

OOP refactoring

Now our OpenGL is initialized, we will refactor our code in a more OOP fashion.

We'll get most part of our initialization logic out of main.cpp to put it in a Window class, whose header will be :

```
#ifndef WINDOW_H
#define WINDOW_H

#include <GL/glew.h>
#include <GLFW/glfw3.h>

#include "Log.h"

extern LogConfig LOG_CONFIG;
extern void glfw_error_callback(int error, const char* description);

class Window {

public:
    Window();
    virtual ~Window();

    void init();
    void log_gl_params();
    bool should_close();
    void handle_close();
    void update_fps_counter();
    void clear();
    void swap_buffer();

private:
    GLFWwindow* glfw_window;
    double previous_seconds;
    int frame_count;

    Window(const Window&);
    Window& operator=(const Window&);
};

#endif
```

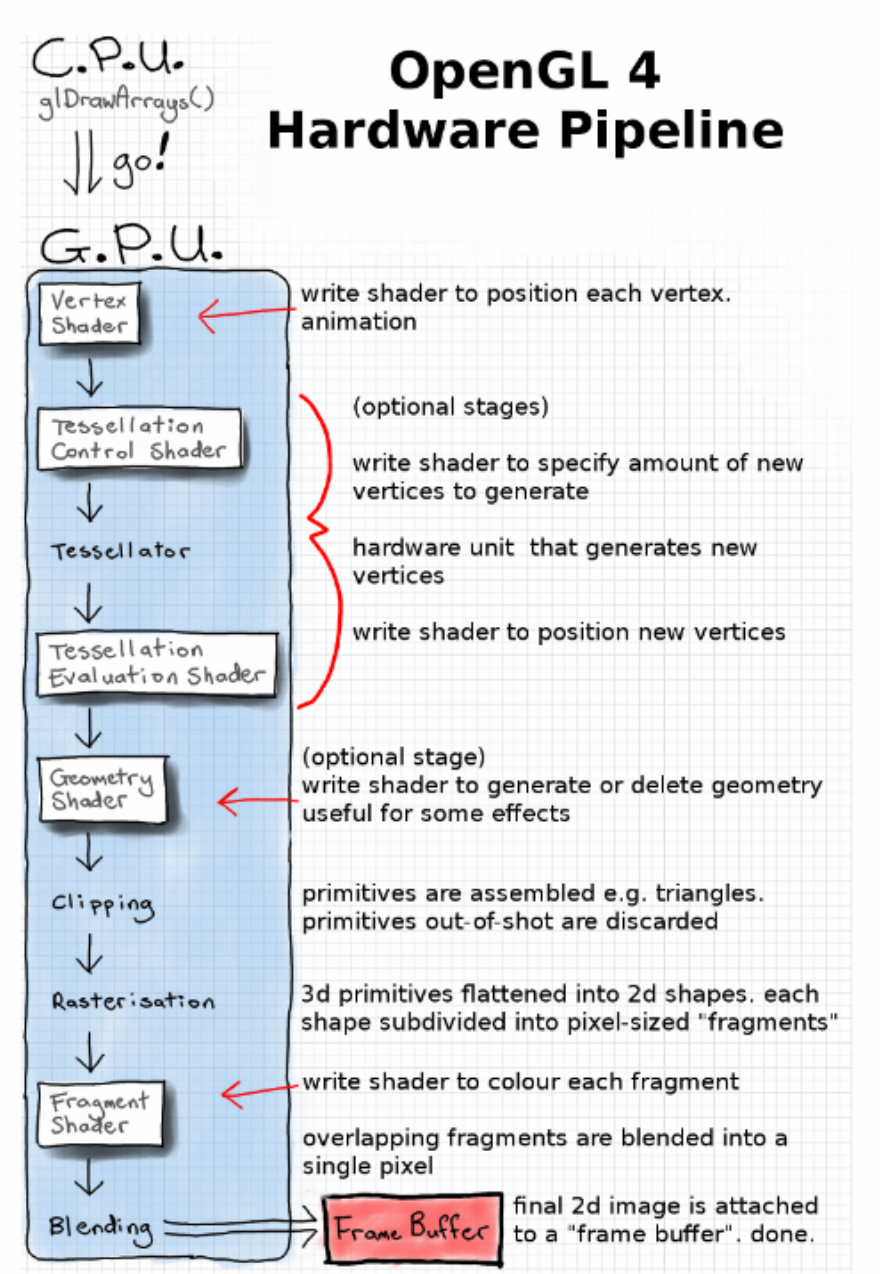
The glfw_error_callback is kept in main.cpp. You can replace all initialization code by proper call of window methods.

Shaders

Shaders

Shaders are the programmable part of a more general graphics pipeline. Graphics pipeline is the ensemble of steps the graphic card compute before sending pixels to display. It is a pipeline because inputs line to be computed one after an other. Graphics card are massively parallel systems, so they compute several inputs at the same time (several thousands shaders at a time).

Here is the OpenGL 4 graphics pipeline. Steps in rectangles are customizable.



A complete shader programme controls comprises a set of separate shader (mini-programmes) - one to control each stage. Each mini-programme - called a shader by OpenGL - is compiled, and the whole set are linked together to form the executable shader programme - called a program by OpenGL. Yes, that's the worst naming convention ever. If you look at the Quick Reference Card, you can see that the API differentiates functions into glShader and glProgram (note US spelling of "programme").

Each individual shader has a different job. At minimum, we usually have 1 vertex shader and 1 fragment shader per shader programme, but OpenGL 4 allows us to use some optional shaders too.

A shader is a small programme. OpenGL uses the GLSL, a C like programming language.

Fragment / Pixels

A pixel is a "picture element". In OpenGL lingo, pixels are the elements that make up the final 2d image that it draws inside a window on your display. A fragment is a pixel-sized area of a surface. A fragment shader determines the colour of each one. Sometimes surfaces overlap - we then have more than 1 fragment for 1 pixel. All of the fragments are drawn, even the hidden ones.

Each fragment is written into the framebuffer image that will be displayed as the final pixels. If depth testing is enabled it will paint the front-most fragments on top of the further-away fragments. In this case, when a farther-away fragment is drawn after a closer fragment, then the GPU is clever enough to skip drawing it, but it's actually quite tricky to organise the scene to take advantage of this, so we'll often end up executing huge numbers of redundant fragment shaders.

Vertex Shader

The vertex shader is responsible for transforming vertex positions into clip space, the final coordinate space that we must transform points to before OpenGL rasterises (flattens) our geometry into a 2d image. Vertex shaders can also be used to send data from the vertex buffer to fragment shaders. This vertex shader does nothing, except take in vertex positions that are already in clip space, and output them as final clip-space positions. We can write this into a plain text file called: test.vert.

```
#version 410

in vec3 vertex_position;

void main() {
  gl_Position = vec4(vertex_position, 1.0);
}
```

GLSL has some built-in data types that we can see here:

- vec3 is a 3d vector that can be used to store positions, directions, or colours.
- vec4 is the same but has a fourth component which, in this variable, is used to determine perspective. We will examine this in the virtual camera article, but for now we can leave it at 1.0, which means "don't calculate any perspective".

We can also see the in key-word for input to the programme from the previous stage. In this case the vertex_position_local is one of the vertex points from the object that we are drawing. GLSL also has an out keyword for sending a variable to the next stage.

The entry point to every shader is a void main() function.

The gl_Position variable is a built-in GLSL variable used to set the final clip-space position of each vertex.

The input to a vertex buffer (the in variables) are called per-vertex attributes, and come from blocks of memory on the graphics hardware memory called vertex buffers. We usually copy our vertex positions into vertex buffers before running our main loop. We will look at vertex buffers in the next tutorial. This vertex shader will run one instance for every vertex in the vertex buffer.

Fragment Shader

Once all of the vertex shaders have computed the position of every vertex in clip space, then the fragment shader is run once for every pixel-sized space (fragment) between vertices. The fragment shader is responsible for setting the colour of each fragment. Write a new plain-text file: test.frag.

```
#version 410

uniform vec4 inputColour;
out vec4 fragColour;

void main() {
  fragColour = inputColour;
}
```

The uniform keyword says that we are sending in a variable to the shader programme from the CPU. This variable is global to all shaders within the programme, so we could also access it in the vertex shader if we wanted to.

The hardware pipeline knows that the first vec4 it gets as output from the fragment shader should be the colour of the fragment. The colours are rgba, or red, green, blue, alpha. The values of each component are floats between 0.0 and 1.0, (not between 0 and 255). The alpha channel output can be used for a variety of effects, which you define by setting a blend mode in OpenGL. It is commonly used to indicate opacity (for transparent effects), but by default it does nothing.

Some OpenGL shader functions

For a complete list of OpenGL shader functions see the Quick Reference Card http://www.opengl.org/documentation/glsl/.

The most useful functions are tabulated below. We will implement all of these:

- glCreateShader() - create a variable for storing a shader's code in OpenGL. returns GLuint index to it.
- glShaderSource() - copy shader code from C string into an OpenGL shader variable
- glCompileShader() - compile an OpenGL shader variable that has code in it
- glGetShaderiv() - can be used to check if compile found errors
- glGetShaderInfoLog() - creates a string with any error information
- glDeleteShader() - free memory used by an OpenGL shader variable
- glCreateProgram() - create a variable for storing a combined shader programme in OpenGL. returns GLuint index to it.
- glAttachShader() - attach a compiled OpenGL shader variable to a shader programme variable
- glLinkProgram() - after all shaders are attached, link the parts into a complete shader programme
- glValidateProgram() - check if a program is ready to execute. information stored in a log
- glGetProgramiv() - can be used to check for link and validate errors
- glGetProgramInfoLog() - writes any information from link and validate to a C string
- glUseProgram() - switch to drawing with a specified shader programme
- glGetActiveAttrib() - get details of a numbered per-vertex attribute used in the shader
- glGetAttribLocation() - get the unique "location" identifier of a named per-vertex attribute
- glGetUniformLocation() - get the unique "location" identifier of a named uniform variable
- glGetActiveUniform() - get details of a named uniform variable used in the shader
- glUniform{1234}{ifd}() - set the value of a uniform variable of a given shader (function name varies by dimensionality and data type)
- glUniform{1234}{ifd}v() - same as above, but with a whole array of values
- glUniformMatrix{234}{fd}v() - same as above, but for matrices of dimensions 2x2,3x3, or 4x4

Logging shader and linking errors

Shader error

After the vertex shader compilation, add this code :

```
int params = -1;
glGetShaderiv(vs, GL_COMPILE_STATUS, &params);
if (params != GL_TRUE)
{
    LOG(Error) << "GL vertex shader index " << vs << " did not compile.";
    print_shader_info_log(vs);
    return -1;
}
```

With :

```
void print_shader_info_log(GLuint shader_index)
{
    int max_length = 2048;
    int actual_length = 0;
    char log[2048];
    glGetShaderInfoLog(shader_index, max_length, &actual_length, log);
    LOG(Info) << "Shader info log for GL index" << shader_index;
    LOG(Info) << log;
}
```

Do the same for the fragment shader.

Linking error

After the programme compilation, add this code :

```
params = -1;
glGetProgramiv(shader_programme, GL_LINK_STATUS, &params);
if (params != GL_TRUE)
{
    LOG(Error) << "Could not link shader programme GL index " << shader_programme;
    print_programme_info_log(shader_programme);
}
```

With :

```
void print_programme_info_log(GLuint programme)
{
    int max_length = 2048;
    int actual_length = 0;
    char log[2048];
    glGetProgramInfoLog(programme, max_length, &actual_length, log);
    LOG(Info) << "program info log for GL index" << programme;
    LOG(Info) << log;
}
```

Logging usual errors

One of the more common errors is mixing up the "location" of uniform variables. Another is where an attribute or uniform variable is not "active"; not actually used in the code of the shader, and has been optimised out by the shader compiler. We can check this by printing it, and we can print all sorts of other information as well. Here, programme is the index of the shader programme.

```
void print_all_params(GLuint programme)
{
 LOG(Info) << "------------";
 LOG(Info) << "Shader programme " << programme << " info:";
 int params = -1;
 glGetProgramiv(programme, GL_LINK_STATUS, &params);
 LOG(Info) << "GL_LINK_STATUS = " << params;

 glGetProgramiv(programme, GL_ATTACHED_SHADERS, &params);
 LOG(Info) << "GL_ATTACHED_SHADERS = " << params;

 glGetProgramiv(programme, GL_ACTIVE_ATTRIBUTES, &params);
 LOG(Info) << "GL_ACTIVE_ATTRIBUTES = " << params;
 for (GLuint i = 0; i < (GLuint)params; i++)
 {
  char name[64];
  int max_length = 64;
  int actual_length = 0;
  int size = 0;
  GLenum type;
  glGetActiveAttrib(programme, i, max_length, actual_length, size, &type, name);
  if (size > 1)
  {
   for (int j = 0; j < size; j++)
   {
    char long_name[77];
    sprintf(long_name, "%s[%i]", name, j);
    int location = glGetAttribLocation(programme, long_name);
    LOG(Info) << "  " << i << ") type:" << GL_type_to_string(type) << " name:" << long_name << " location:" << location;
   }
  }
  else
  {
   int location = glGetAttribLocation(programme, name);
   LOG(Info) << "  " << i << ") type:" << GL_type_to_string(type) << " name:" << name << " location:" << location;
  }
 }

 glGetProgramiv(programme, GL_ACTIVE_UNIFORMS, &params);
 LOG(Info) << "GL_ACTIVE_UNIFORMS = " << params;
 for (GLuint i = 0; i < (GLuint)params; i++)
 {
  char name[64];
  int max_length = 64;
  int actual_length = 0;
  int size = 0;
  GLenum type;
  glGetActiveUniform(programme, i, max_length, actual_length, size, &type, name);
  if (size > 1)
  {
   for (int j = 0; j < size; j++)
   {
    char long_name[77];
    sprintf(long_name, "%s[%i]", name, j);
    int location = glGetUniformLocation(programme, long_name);
    LOG(Info) << "  " << i << ") type:" << GL_type_to_string(type) << " name:" << long_name << " location:" << location;
   }
  }
  else
  {
   int location = glGetUniformLocation(programme, name);
   LOG(Info) << "  " << i << ") type:" << GL_type_to_string(type) << " name:" << name << " location:" << location;
  }
 }
 print_programme_info_log(programme);
}
```

The interesting thing here are the printing of the attribute and uniform "locations". Sometimes uniforms or attributes are themselves arrays of variables - when this happens size is > 1, and we loop through and print each index' location separately.

Here is a home-made function for printing the GL data type as a string (normally it is an enum which doesn't look very meaningful when printed as an integer) :

```
const char* GL_type_to_string(GLenum type)
{
 switch (type)
 {
 case GL_BOOL:
  return "bool";
 case GL_INT:
  return "int";
 case GL_FLOAT:
  return "float";
 case GL_FLOAT_VEC2:
  return "vec2";
 case GL_FLOAT_VEC3:
  return "vec3";
 case GL_FLOAT_VEC4:
  return "vec4";
 case GL_FLOAT_MAT2:
  return "mat2";
 case GL_FLOAT_MAT3:
  return "mat3";
 case GL_FLOAT_MAT4:
  return "mat4";
 case GL_SAMPLER_2D:
  return "sampler2D";
 case GL_SAMPLER_3D:
  return "sampler3D";
 case GL_SAMPLER_CUBE:
  return "samplerCube";
 case GL_SAMPLER_2D_SHADOW:
  return "sampler2DShadow";
 default:
  break;
 }
 return "other";
}
```

Program validation

You can also "validate" a shader programme before using it. Only do this during development, because it is quite computationally expensive. When a programme is not valid, the details will be written to the program info log. programme is the shader programme index.

```
bool is_valid(GLuint programme)
{
  glValidateProgram(programme);
  int params = -1;
  glGetProgramiv(programme, GL_VALIDATE_STATUS, &params);
  LOG(Info) << "program " << programme << " GL_VALIDATE_STATUS = " << params;
  if (params != GL_TRUE)
  {
    print_programme_info_log(programme);
    return false;
  }
  return true;
}
```

OOP refactorisation

Now create a Shader class header and source file, to store our shader functions. Header can be :

```
#ifndef SHADER_H
#define SHADER_H

#include "log.h"

class Shader
{
public:
   Shader();
   virtual ~Shader();

   void compile_vertex_shader();
   void compile_fragment_shader();
   void create_shader_programme();

   GLuint get_shader_programme() const;

private:
   GLuint vs;
   GLuint fs;
   GLuint programme;

   const char* vertex_shader =
      "#version 410\n"
      "in vec3 vp;"
      "void main () {"
      "  gl_Position = vec4 (vp, 1.0);"
      "}";

   const char* fragment_shader =
      "#version 410\n"
      "out vec4 frag_colour;"
      "void main () {"
      "  frag_colour = vec4 (0.8, 0.5, 0, 1.0);"
      "}";

   void print_shader_info_log(GLuint shader_index);
   void print_programme_info_log(GLuint programme);
   const char* GL_type_to_string(GLenum type);
   void print_all_params(GLuint programme);
   bool is_valid(GLuint programme);

   Shader(const Shader &);
   Shader& operator=(const Shader &);
};

#endif
```

Include <GL/glew.h> and <GLFW/glfw3.h> in the cpp file to avoid redundancies..

Best practices

- All uniform variables are initialised to 0 when a programme links, so you only need to initialise them if the initial value should be something else. Example: you might want to set matrices to the identity matrix, rather than a zeroed matrix.
- Calling glUniform is quite expensive during run-time. Structure your programme so that glUniform is only called when the value needs to change. This might be the case every time that you draw a new object (e.g. its position might be different), but some uniforms may not change often (e.g. projection matrix).
- Calling glGetUniformLocation during run-time can be expensive. It is best to do this during initialisation of the component that updates the uniform e.g. the virtual camera for the projection and view matrices, or a renderable object in the scene for the model matrix. Store the uniform locations once, then call glUniform as needed, rather than updating everything every frame.
- When calling glGetUniformLocation, it returns -1 if the uniform variable wasn't found to be active. You can check for this. Usually it means that either you've made a typo in the name, or the variable isn't actually used anywhere in the shader, and has been "optimised out" by the compiler/linker.
- Modifying attributes (vertex buffers) during run-time is extremely expensive. Avoid.
- Get your shaders to do as much work as is possible; because of their parallel nature they are much faster than looping on the CPU for most tasks.
- Drawing lots of separate, small objects at once does not make efficient use of the GPU, as most parallel shader slots will be empty, and separate objects must be drawn in series. Where possible, merge many, smaller objects into fewer, larger objects.

Extending your shader class

- Create a "reload my shaders" function. Bind this to a keyboard key. If you get this working your should be able to live edit your shaders and see what they do without restarting your programme. This should speed you up.
- Larger projects will use many different shader programmes. It would make sense to have a Shader Manager interface or class to load shaders, and to make sure that shaders are re-used, rather than loaded multiple times.
- In the future we will sometimes use geometry and tessellation shaders, so we can consider upgrading our shader functions to handle more than just vertex and fragment shaders. We can just set some boolean flags to true or false to indicate which types of shader have been loaded before attaching and linking.
- If you have a shader manager, and have written a function along the lines of setUniform (shader_index, value) from the manager, it could then check to make sure that shader is in use first (a common cause of error).

Vertex Buffer Objects

Why VBO ?

A vertex buffer object (VBO) is nothing fancy - it's just an array of data (usually floats). We already had a look at the most basic use of vertex buffer objects in the Hello Triangle tutorial. We know that we can describe a mesh in an array of floats; {x,y,z,x,y,z..x,y,z}. And we also know that we need to use a Vertex Array Object to tell OpenGL that the array is divided into variables of 3 floats each.

The key idea of VBOs is this: in the old, "immediate-mode" days of OpenGL, before VBOs, we would define the vertex data in main memory (RAM), and copy them one-by-one each time that we draw. With VBOs, we copy the whole lot into a buffer before drawing starts, and this sits on the graphics hardware memory instead. This is much more efficient for drawing because, although the bus between the CPU and the GPU is very wide, a bottle-neck for drawing performance is created by stalling drawing operations to send OpenGL commands from the CPU to the GPU. To avoid this we try to keep as much data and processing on the graphics hardware as we can.

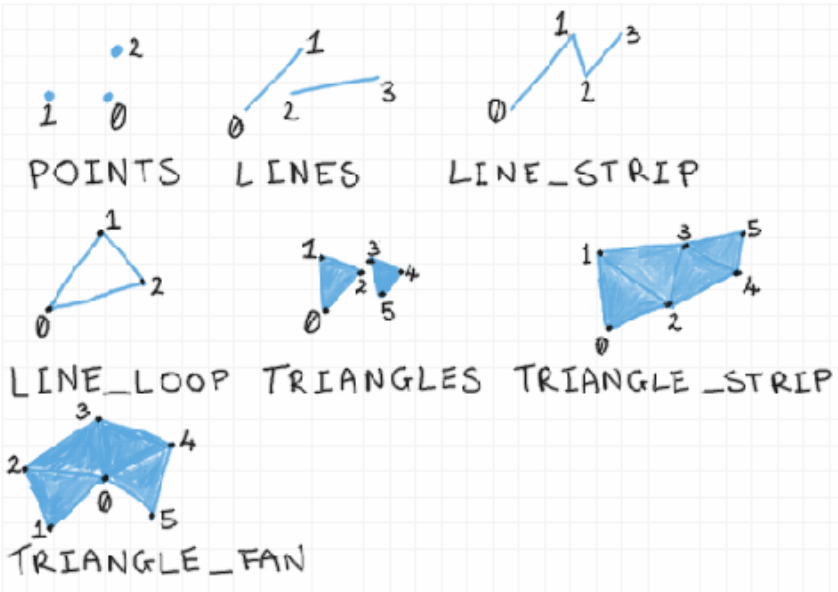
A VBO is not an "object" in the object-oriented programming sense, it is a simple array of data. OpenGL has several other types of data that it refers to as singular "buffer objects". The term "object" here implies that the GL gives us a handle or identifier number to the whole buffer to interact with, rather than a traditional address to the first element in the buffer.

We will look at managing vertex buffers in a little bit more depth. We will need this for enabling lighting and texturing effects later. We will break down the interface, and visualise how interpolation between the vertex shader and the fragment shader works.

Rendering Different Primitive Types

Drawing modes

We used glDrawArrays(GL_TRIANGLES, 0, 3) to draw a triangle. But there are several other drawing modes :



Try changing your GL_TRIANGLES parameter. You can see that points and lines are going to be useful for drawing thing like charts or outlines. Triangle strip is a slightly more efficient method for drawing ribbon-like shapes. You can actually change the size of the points in the vertex shader, so they are quite versatile.

Wire-frame

It's much easier to use the glPolygonMode built-in function than to attempt to draw everything with GL_LINES. Call glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); before rendering.

Using Multiple Vertex Buffers for One Object

We can store more than just 3d points in a vertex buffer. Common uses also include :

- 2d texture coordinates that describe how to fit an image to a surface
- 3d normals that describe which way a surface is facing so that we can calculate lighting.

So it's quite likely that most of your objects will have 2 or 3 vertex buffers each.

You can use vertex buffers to hold any data that you like; the key thing is that the data is retrieved once per vertex. If you tell OpenGL to draw an array of 3 points, using a given vertex array object, then it is going to launch 3 vertex shaders in parallel, and each vertex shader will get a one variable from each of the attached arrays; the first vertex shader will get the first 3d point, 2d texture coordinate, and 3d normal, the second vertex shader will get the second 3d point, 2d texture coordinate, and 3d normal, and so on, where the number of variables, and size of each variable is laid out in the vertex array object.

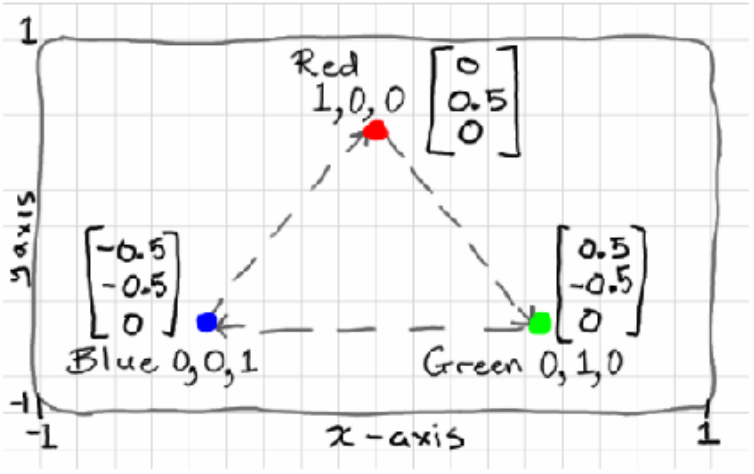
Colour interpolation on vertices 1/3

Data

Vertex colours are very seldom used in practise, but most modern GL tutorials will get you to create a buffer of colours as a second vertex buffer. Why? Because it's easy to visualise how interpolation works with colours.

```
GLfloat points[] = {
   0.0f,  0.5f,  0.0f,
   0.5f, -0.5f,  0.0f,
  -0.5f, 0.5f,  0.0f
};

GLfloat colours[] = {
  1.0f, 0.0f,  0.0f,
  0.0f, 1.0f,  0.0f,
  0.0f, 0.0f,  1.0f
};
```



VBO

Now for each one we can create a GL vertex buffer object, bind it in the state machine, and copy the array of values into it. Both colours, and points have 9 components each (3 vertices with 3 components per vertex). We set up one buffer after the other, because the state machine can only have 1 buffer bound at a time.

```
GLuint points_vbo = 0;
glGenBuffers (1, &points_vbo);
glBindBuffer (GL_ARRAY_BUFFER, points_vbo);
glBufferData (GL_ARRAY_BUFFER, sizeof (points), points, GL_STATIC_DRAW);

GLuint colours_vbo = 0;
glGenBuffers (1, &colours_vbo);
glBindBuffer (GL_ARRAY_BUFFER, colours_vbo);
glBufferData (GL_ARRAY_BUFFER, sizeof (colours), colours, GL_STATIC_DRAW);
```

Colour interpolation on vertices 2/3

VAO

Our object now consists of 2 vertex buffers, which will be input "attribute" variables to our vertex shader. We set up the layout of both of these with a single vertex attribute object - the VAO represents our complete object, so we no longer need to keep track of the individual VBOs.

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colours_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

Note that we have to bind each VBO before calling glVertexAttribPointer(), which describes the layout of each buffer to the VAO.

The first parameter of glVertexAttribPointer() asks for an index. This is going to map to the indices in our vertex shader, so we need to give each attribute here a unique index. I will give my points index 0, and the colours index 1. We will make these match up to the variables in our vertex shader later. If you accidentally leave both indices at 0 (easy enough to do when copy-pasting code), then your colours will be read from the position values so x→red y→green and z→blue.

Both buffers contain arrays of floating point values, hence GL_FLOAT, and each variable has 3 components each, hence the 3 in the second parameter. If you accidentally get this parameter wrong (quite a common mistake), then the vertex shaders will be be given variables made from the wrong components (e.g. position x, y, z gets values read from a y, z, x).

That is the mesh side taken care of - you only need to do this part once, when creating the object. There is no need to repeat this code inside the rendering loop. Just keep track of the VAO index for each type of mesh that you create.

Enable Vertex Arrays 0 and 1 in a VAO

Attributes are disabled by default in OpenGL 4. We need to explicitly enable them too. This is easy to get wrong or overlook, and is not well explained in the documentation. We use a function called glEnableVertexAttribArray() to enable each one. This function only affects the currently bound vertex array object. This means that when we do this now, it will only affect our attributes, above. We will need to bind every new vertex array and repeat this procedure for those too.

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

NB : Other GL incarnations don't have vertex attribute objects, and need to explicitly enable and disable the attributes every time a new type of object is drawn, which changes the enabled attributes globally in the state machine. We don't need to worry about that in OpenGL 4; the vertex attribute object will remember its enabled attributes. i.e. they are no longer global states. This isn't clear in the official documentation, nor is it explained properly in other tutorials - it's very easy to get a false-positive misunderstanding of how these things work and run into hair-pulling problems later when you're trying to draw 2 different shapes.

Colour interpolation on vertices 3/3

Change the vertex shader

We want our shader to render using the second vertex buffer, so we need to add a second attribute to the top of the vertex shader. We are going to add the OpenGL 4 layout prefix to each attribute. This lets us manually specify a location for each attribute - and we are going to match this up to the index that we gave each one in glVertexAttribPointer. If you don't specify a location, the shader programme linker will automatically assign one. You can query this, which we did in Shaders, but I prefer to specify it manually so I can see the values as I write.

```
const char *vertex_shader =
  "#version 430\n"

  "layout(location = 0) in vec3 vertex_position;"
  "layout(location = 1) in vec3 vertex_colour;"

  "out vec3 colour;"

  "void main ()"
  "{"
  "colour = vertex_colour;"
  "gl_Position = vec4 (vertex_position, 1.0);"
  "}";
```

Now, the other interesting change here is that I took the vertex_colour input attribute and gave it to an output variable, which I called colour. This is where it gets interesting. It has no effect on our vertex shader, but outputs it to the next stage in the programmable hardware pipeline. In our case - the fragment shader.

Change the fragment shader

We can rewrite the fragment shader to use our new colour variable as an input, which will colour each fragment directly. Note that we add an in prefix to retrieve a variable from a previous stage. This in/out convention is a little different to how other OpenGL versions work. Our output fragment colour needs to be r,g,b,a (4 components) so we can just add a 1 to the end of our 3-component colour by casting it as a vec4.

```
const char *fragment_shader =
  "#version 430\n"

  "in vec3 colour;"
  "out vec4 frag_colour;"

  "void main ()"
  "{"
  "frag_colour = vec4 (colour, 1.0);"
  "}";
```

Vertex Shader to Fragment Shader Interpolation

Now, remember, our triangle has only 3 vertices, but 1 fragment for every pixel-sized area of the surface. This means that we have 3 colour outputs from vertex shaders, and perhaps 100 colour inputs to each fragment shader. How does this work?

The answer is that each fragment shader gets an interpolated colour based on its position on the surface. The fragment exactly on the red corner of the triangle will be completely red (1.0, 0.0, 0.0). A fragment exactly half-way between the blue and red vertices, along the edge of the triangle, will be purple; half red, half blue, and no green: (0.5, 0.0, 0.5). A fragment exactly in the middle of the triangle will be an equal mixture of all 3 colours; (0.3333, 0.3333, 0.333).

Keep in mind that this will happen with any other vertex buffer attributes that you send to the fragment shader; normals will be interpolated, and texture coordinates will be interpolated to each fragment. This is really handy, and we will exploit it for lots of interesting per-pixel effects. But it's quite common to misunderstand this when getting started with shaders - vertex shader outputs are not sending a constant variable from a vertex shader to all fragment shaders. We use uniform variables for that.

"Winding" and Back-Face Culling

The last thing that you should know about is a built-in rendering optimisation called back-face culling. This gives a hint to GL so that it can throw away the hidden "back" or inside faces of a mesh. This should remove half of the vertex shaders, and half of the fragment shader instances from the GPU - allowing you to render things twice as large in the same time. It's not appropriate all of the time - you might want our 2d triangle to spin and show both sides.

The only things that you need specify are if clock-wise vertex "winding" order means the front, or the back of each face, and set the GL state to enable culling of that side. Our triangle, you can see, is given in clock-wise order. Most mesh formats actually go the other way, so it pays to test this before wondering why a mesh isn't showing up at all!

```
glEnable(GL_CULL_FACE);   // Cull face
glCullFace(GL_BACK);      // Cull back face
glFrontFace(GL_CW);       // GL_CCW for Counter Clock-Wise
```

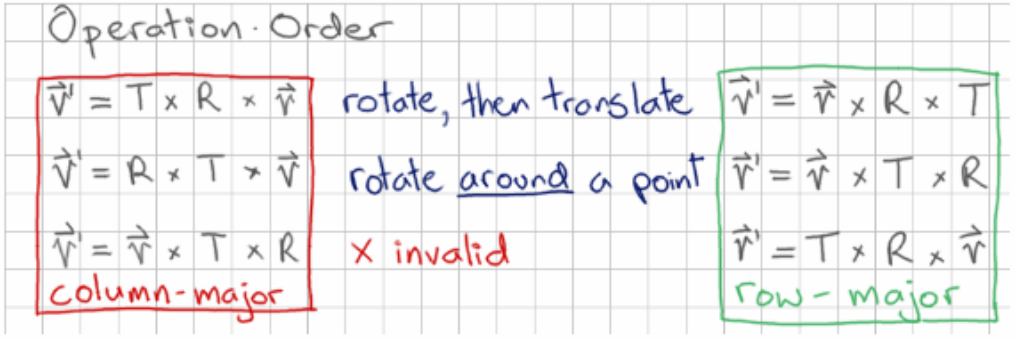
Try switching to counter clock-wise to make sure that the triangle disappears. If you were to rotate it around now you'd see the other side was visible. As with other GL states, this culling is enabled globally, in the state machine, you can enable and disable it between calls to glDrawArrays so that some objects are double-sided, and some are single-sided, etc. Keep in mind which winding order you are making new shapes in.

Using matrices with OpenGL

Row Order and Column Order

When you are computing vectors and matrices in C code, you need to stick to a layout convention. The two layouts are row major and column major. You can use either convention, but it needs to be consistent. For no particular reason, OpenGL people tend to favour column-major matrices, and Direct3D people row-major. You can swap between row-major and column-major by transposing a matrix. This flips it along the main diagonal (top-left to bottom-right diagonal). It's easy to spot if a 4X4 matrix is in row order because the transformation components will be on the bottom-row, and there will be zeros in the right-most column (as in the example drawn above). Column-major matrices will have the transformation in the right-most column.

The layout convention affects the order of multiplication, as detailed in below:



*Where R is a rotation matrix, T is a translation matrix, **v** is a vector and **v'** is the resulting vector.*

So, there is a specific order for multiplying matrices together, or vectors by matrices. We also see that 4d matrices are non-commutative - we get a different result if we multiply them in a different order. We will do vector and matrix multiplication in both C and in shaders, so it's worth remembering which convention is being used. In the column-order examples, you can see we are going right-to-left; e.g. input vector, rotate, then translate.

A minor point - in maths notation, row-major vectors are drawn in a row, and column-major vectors are drawn in a column. Matrices are expressed as a capital letter, and vectors a lower-case letter, either in bold-face, or with a harpoon/arrow symbol above them. Normalised (unit) vectors have a "hat" $\hat{v}$ rather than an arrow.

So, what have we learned? We can write a 4X4 matrix directly into a 1d array in C, and send that to a shader using glUniform, where it will appear as a uniform mat4. We can then use it to transform our vertex points using an affine transformation. We already have the layout of a translation matrix, so let's try that.

Translation

Start a simple project. You can use the Hello Triangle demo if you don't have one yet. Let's define a matrix that moves the triangle 0.5 units to the right. Remember that the edge of the view is - 1 on the left and 1 on the right, so we don't want to move it completely off the screen. Our matrix will look like this in column-major layout:

```
float matrix[] = {
    1.0f, 0.0f, 0.0f, 0.0f, // first column
    0.0f, 1.0f, 0.0f, 0.0f, // second column
    0.0f, 0.0f, 1.0f, 0.0f, // third column
    0.5f, 0.0f, 0.0f, 1.0f // fourth column
};
```

It looks transposed to the way we would write it down in matrix notation when we code it up like this, but I don't want to have to transpose it when I give it to GL.

Now we need to add the uniform variable into the vertex shader, and actually do the multiplication with the vertex point. The vertex shader will look something like this:

```
const char *vertex_shader =
    "#version 430\n"
    "layout(location = 0) in vec3 vertex_position;"
    "layout(location = 1) in vec3 vertex_colour;"

    "uniform mat4 matrix;"

    "out vec3 colour;"

    "void main ()"
    "{"
    "colour = vertex_colour;"
    "gl_Position = matrix * vec4(vertex_position, 1.0);"
    "}";
```

Note that the multiplication here is in column-major order as well. Now we should be able to retrieve the location of the new matrix uniform. Do this somewhere after where you link the shader programme, but before entering the main loop. Once we have that, we can "use" the shader programme, and send in our matrix.

```
GLuint shader_programme = shader.get_shader_programme();
int matrix_location = glGetUniformLocation(shader_programme, "matrix");
glUseProgram(shader_programme);
glUniformMatrix4fv(matrix_location, 1, GL_FALSE, matrix);

// Loop until the user closes the window
while (!window.should_close())
{
    ...
}
```

Moving the triangle

Start a simple project. You can use the Hello Triangle demo if you don't have one yet. Let's define a matrix that moves the triangle 0.5 units to the right. Remember that the edge of the view is - 1 on the left and 1 on the right, so we don't want to move it completely off the screen. Our matrix will look like this in column-major layout:

```
GLuint shader_programme = shader.get_shader_programme();
int matrix_location = glGetUniformLocation(shader_programme, "matrix");

float speed = 1.0f;
float last_position = 0.0f;
double previous_seconds = glfwGetTime();

while (!window.should_close())
{
   window.update_fps_counter();

   double current_seconds = glfwGetTime();
   double dt = current_seconds - previous_seconds;
   previous_seconds = current_seconds;

   // Inputs
   glfwPollEvents();
   window.handle_close();

   // Update
   if (fabs(last_position) > 1.0f) {
      speed = -speed;
   }

   matrix[12] = speed * dt + last_position;
   last_position = matrix[12];
   glUseProgram(shader_programme);
   glUniformMatrix4fv(matrix_location, 1, GL_FALSE, matrix);

   // Draw
   window.clear();
   glBindVertexArray(vao);
   glDrawArrays(GL_TRIANGLES, 0, 3);
   window.swap_buffer();
}
```

Rotating the triangle

Your turn ! Use your maths.

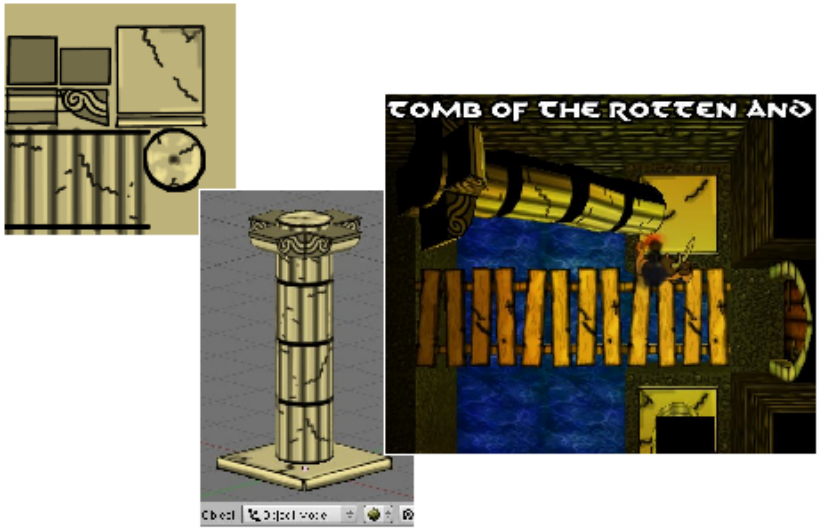
If you want to rotate + scale + translate, you can create your own math library or use GLM : https://glm.g-truc.net/0.9.9/index.html

Textures

Texture mapping

Texture mapping is another algorithm from the 1970s that real-time rendering technology wasn't ready for until the 1990s. Edwin Catmull (currently president of Pixar) is credited with creating the idea in 1974. The appropriate reference is his PhD thesis; "Computer display of curved surfaces".

In real-time rendering history, it was adopted in 1990 by the team building the still awesome Ultima Underworld. There's a nice article about that development on Wikipedia. It was beaten to market by Catacomb 3-D (Id Software) which used their own approach to texturing. The idea is, that instead of using simple colours for surfaces, we will load the surface colours from an image file, and map them to the surface. Not copy, but map - because our surfaces are not going to be the same size as the image, and some maths are required to match them up.



The texturing process; (left) a square image is loaded from a file as a texture. (centre) A designer will assign every vertex on a mesh with a point on the texture; creating a texture coordinate for each vertex. (right)

As the vertices are transformed in 3d, the rendering software will work out the texture coordinate of every fragment on the surface by interpolating the texture coordinates of its surrounding vertices, and sampling the colour of the corresponding point in the image.

We have some special terminology to avoid confusion between on-screen pixels, those from the original image, and the finals coloured elements on a surface:

- Texture, an image (usually loaded from a file) that can be mapped onto a surface, and drawn to match the 3d perspective of the surface.
- Pixels, or "picture elements", are the final on-screen, coloured spots rendered to the viewport. Your GL viewport might have 1024x768 pixels.
- Fragments are pixel-sized areas of a surface (the visible surface is divided into pixel-sized 2d fragments). As a 3d triangle gets closer to the camera, its surface will contain more and more fragments. Fragments can overlap, and we can blend them together for techniques like partial-transparency, but there is only 1 final pixel rendered.
- Texels, or "texture elements", are pixels loaded from an image. If we map a 512x512 texture to a surface that occupies only 30 pixels of the viewport, we have more texels than fragments.

To load textuxes, we will use stb_image. Create a stb_image.h file and copy the source code from :
https://github.com/nothings/stb/blob/master/stb_image.h

Loading texture

Create a class for the texture :

```
#include <string>

class Texture {
public:
    Texture();
    Texture(std::string _filename);
    ~Texture();

    void generate();

private:
    int x;
    int y;
    int n;
    std::string filename;
    const int force_channels = 4;
    unsigned char* image_data;
};
```

The texture will be loaded when instantiated :

```
#include "texture.h"
#include "log.h"

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#include "GL/glew.h"

Texture::Texture()
{
    filename = "";
    x = 0;
    y = 0;
    n = 0;
}

Texture::~Texture()
{
}

Texture::Texture(std::string filename) : filename(filename)
{
    image_data = stbi_load(filename.c_str(), x, y, &n, force_channels);
    if (!image_data)
    {
        LOG(Error) << "Could not load " << _filename;
    }
    if ((x & (x - 1)) != 0 || (y & (y - 1)) != 0)
    {
        LOG(Warning) << "Texture " << filename << " is not power-of-2 dimensions";
    }

    // Flip image vertically
    int width_in_bytes = x * 4;
    unsigned char *top = NULL;
    unsigned char *bottom = NULL;
    unsigned char temp = 0;
    int half_height = y / 2;

    for (int row = 0; row < half_height; row++)
    {
        top = image_data + row * width_in_bytes;
        bottom = image_data + (y - row - 1) * width_in_bytes;
        for (int col = 0; col < width_in_bytes; col++)
        {
            temp = *top;
            * top = *bottom;
            * bottom = temp;
            top++;
            bottom++;
        }
    }
}
```

The stb_image loading function gave me the dimensions of the loaded image, in x and y variables. Here the bitwise AND operator is used to check for non-power-of-two numbers. See if you can work out how that little bit-hack works with your knowledge of C operators.

The image_data is upside down. This is because OpenGL expects the 0 on the Y-axis to be at the bottom of the texture, but images usually have Y-axis 0 at the top. We have added some code to flip the image upside-down before creating the texture. We know the width of each row in the image in bytes, because stb_image gave us the width in pixels, and each pixel is 4 bytes big. We can don't need to copy the whole image ; we can just swap the top and the bottom half over. To do this we will put a pointer at the top row, and a pointer at the bottom row. We can swap the data of these rows over, then move the top pointer down a row, and the bottom pointer up a row. We just need to stop when we get to the middle.

The interesting thing to note is that each time that we move a pointer we multiply the move by 4, because a pixel is 4 bytes wide (RGBA remember). All of our image data is contiguous, so that's okay; our current address + 4 will be the location of the next RGBA pixel. We use a temp variable to remember the value at each location as we swap their values over. Remember that, in C, top is a pointer and just holds a number representing an address in memory. To get to the value in that address we "dereference" the pointer: *top.

Copy image data to OpenGL texture

Create the texture

We need to generate a new texture object, which will give us an unsigned integer to refer to it with later. Before we can copy data into the texture we have to bind it into focus. Any time that we bind a texture it will be moved into one of several active texture slots. We don't need to worry about this yet, but it's good to specify a slot now, otherwise it will use the most recently activated slot, which might create unexpected interference with another part of your programme - remember GL is a state machine. . This is done in the get function :

```
void Texture::generate()
{
    GLuint tex = 0;
    glGenTextures(1, &tex);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, tex);
    glTexImage2D(
        GL_TEXTURE_2D,
        0,
        GL_RGBA,
        x,
        y,
        0,
        GL_RGBA,
        GL_UNSIGNED_BYTE,
        image_data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
}
```

Look up the glTexImage2D function in the official documentation. It has a lot of parameters, but all of them should relate to how we loaded our image. At this stage we are just using bog-standard 2d textures, but we have to specify this whenever we bind or create a texture anyway. We aren't going to set up any custom levels-of-detail texture, so we just put 0 for this parameter. There are lots of options for texture formats - it's a good idea to have a look at the tables of options. We know for sure that our loaded image is RGBA format now, so we can say GL_RGBA. We give it the width and height in pixels, which were returned by our image loader. The next "0" is an unused parameter. Again, we can put RGBA as our internal format. It's possible to have the bytes of each pixel ordered differently. Finally, we know that our image data is in unsigned char (byte) format, and we give it the pointer to our data. The last four lines before the return are a good, safe, default "wrapping" mode to use for loaded textures, and a good default for anti-aliasing. We'll get onto both of these topics later.

Active Texture Slots

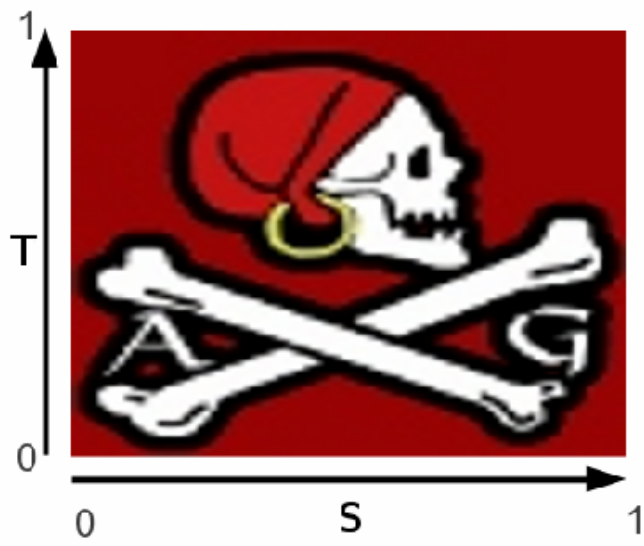
The texture is now loaded, and we can use it whenever we want to render by activating a texture slot, and binding the texture. It will stay in that slot until another texture is bound into it, so if your programme only has 1 texture then you can just do that once.

OpenGL can load a large number of textures into memory, but your GPU can only read from a small number of textures at one time (most GPUs will read 8 or so). To manage this, the state machine has a number of what it calls "active textures". We need to swap our textures in and out of these active texture slots as we render different objects. Yes, we have to do this low-level tedium ourselves, and it's very easy to make mistakes. The default active texture slot is number 0.

Texture in fragment shader

Texture coordinates

The main problem is that the surfaces are not going to have the same number of fragments as the image has texels, and this will change as the surface is viewed at viewpoints and perspectives. The obvious solution is to calculate a very simple interpolation function for each fragment of the wall, to find out the closest-matching texel from the image.



*GL texture co-ordinates for s (horizontal) from left to right, and t (vertical) from bottom to top.**Hint:** test your texture mapping software with an image that has text in it; this will help you spot if you've got one of the axes reversed.*

To keep interpolation of texels independent of the image size, we use texture coordinates, which are a horizontal and vertical value between 0.0 and 1.0. In OpenGL 0.0 is the left or bottom of a texture, and 1.0 is the right, or top of a texture. Direct3D has 0.0 at the top, and 1.0 at the bottom - don't mix them up!

So we can make another per-vertex variable to keep track of what part of the image each vertex should map to; a texture coordinate. This will be 2 floats for each vertex, and we will make another vertex buffer to store these. Our previous vertex buffers have all had 3 floats - don't accidentally tell GL that it is 3d, or it will get the memory "stride" (ordering) wrong, and you'll have mixed-up values.

We use x,y, z,w to refer to 3d space, and r,g,b,a for colours. In most 3d libraries u,v are used to refer to 2d space, but OpenGL prefers s,t. The GLSL language vector data-types (vec2, vec3, vec4) have built-in short-cuts called swizzle operators. You may have used something like vec3 pos = result.xyz; or float red = Kd.r; already. You can also use .s, .t, and .st.

Creating Texture Coordinates

For a wall mesh, made up of 2 triangles, we will create 18 vertices and 6 texture coordinates, and put them in a vertex buffer.

```
GLfloat points[] = {
   - 0.5f, -0.5f, 0.0f,
   0.5f, -0.5f, 0.0f,
   0.5f, 0.5f, 0.0f,

   0.5f, 0.5f, 0.0f,
   -0.5f, 0.5f, 0.0f,
   - 0.5f, -0.5f, 0.0f
};

GLfloat texcoords[] = {
   0.0f, 0.0f, 1.0f,
   0.0f, 1.0f, 1.0f,
   1.0f, 1.0f, 0.0f,
   1.0f, 0.0f, 0.0f
};

GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);

GLuint vt_vbo;
glGenBuffers(1, &vt_vbo);
glBindBuffer(GL_ARRAY_BUFFER, vt_vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(texcoords), texcoords, GL_STATIC_DRAW);

GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, vt_vbo);
int dimensions = 2; // 2d data for texture coords
glVertexAttribPointer(1, dimensions, GL_FLOAT, GL_FALSE, 0, NULL);

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

Texture texture("assets/wall_texture.png");
texture.generate();
```

It would be optimum to mix vertices and texture coordinates, and to use an other draw mode, we will do that later.

Don't forget to draw the two triangles after biding the VAO :

```
    glDrawArrays(GL_TRIANGLES, 0, 6);
```

Example : A Breakout Game

Game class

We will start by creating a simple game engine, so our breakout game can run. Let's start by creating a simple game class.

```
#ifndef GAME_H
#define GAME_H

#include <GL/glew.h>
#include <GLFW/glfw3.h>

enum GameState
{
    GameActive,
    GameMenu,
    GameWin
};

class Game
{
public:
    Game(GLuint width, GLuint height);
    ~Game();

    GameState state;
    GLboolean keys[1024];
    GLuint width, height;

    void init();
    void process_input(GLfloat dt);
    void update(GLfloat dt);
    void draw();

    Game(const Game &);
    Game operator=(const Game );
};

#endif
```

For now, the cpp file will be empty :

```
#include "game.h"

Game::Game(GLuint width, GLuint height)
        : state(GameActive), keys(), width(width), height(height)
{

}

Game::~Game()
{

}

void Game::init()
{

}

void Game::update(GLfloat dt)
{

}


void Game::process_input(GLfloat dt)
{

}
```

Shader class updated 1/2

We will update shader class to iinclude uniform change, compiling and using inside. Il will support geometry shaders too.

```
#ifndef SHADER_H
#define SHADER_H

#include "log.h"
#include <string>
#include <GL/glew.h>
#include <glm.hpp>
#include <gtc/type_ptr.hpp>

class Shader
{
public:
    Shader();
    virtual ~Shader();

    GLuint id;

    void compile(const GLchar vertex_source, const GLchar fragment_source, const GLchar *geometry_source = nullptr);
    Shader &use();

    void set_float    (const GLchar *name, GLfloat value, GLboolean use_shader = false);
    void set_integer  (const GLchar *name, GLint value, GLboolean use_shader = false);
    void set_vector2f (const GLchar *name, GLfloat x, GLfloat y, GLboolean use_shader = false);
    void set_vector2f (const GLchar *name, const glm::vec2 &value, GLboolean use_shader = false);
    void set_vector3f (const GLchar *name, GLfloat x, GLfloat y, GLfloat z, GLboolean use_shader = false);
    void set_vector3f (const GLchar *name, const glm::vec3 &value, GLboolean use_shader = false);
    void set_vector4f (const GLchar *name, GLfloat x, GLfloat y, GLfloat z, GLfloat w, GLboolean use_shader = false);
    void set_vector4f (const GLchar *name, const glm::vec4 &value, GLboolean use_shader = false);
    void set_matrix4  (const GLchar *name, const glm::mat4 &matrix, GLboolean use_shader = false);

private:
    GLuint vs;
    GLuint fs;
    GLuint gs;

    void compile_vertex_shader(const GLchar *vertex_source);
    void compile_fragment_shader(const GLchar *fragment_source);
    bool compile_geometry_shader(const GLchar *geometry_source);
    void create_shader_program(bool geometry_shader_exists);

    void check_shader_errors(GLuint shader, std::string shader_type);
    void print_shader_info_log(GLuint shader_index);
    void print_programme_info_log(GLuint programme);
    const char *GL_type_to_string(GLenum type);
    void print_all_params(GLuint programme);
    bool is_valid(GLuint programme);
};

#endif
```

Shader class updated 2/2

Only updated functions are listed below. Constructors and log functions remain the same.

```
void Shader::check_shader_errors(GLuint shader, std::string shader_type)
{
    int params = -1;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &params);
    if (params != GL_TRUE)
    {
        LOG(Error) << "GL " << shader_type << " shader index " << shader << " did not compile.";
        print_shader_info_log(shader);
    }
}

void Shader::compile_vertex_shader(const GLchar *vertex_source)
{
    vs = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vs, 1, &vertex_source, NULL);
    glCompileShader(vs);
    check_shader_errors(vs, "vertex");
}

void Shader::compile_fragment_shader(const GLchar *fragment_source)
{
    fs = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fs, 1, &fragment_source, NULL);
    glCompileShader(fs);
    check_shader_errors(fs, "fragment");
}

bool Shader::compile_geometry_shader(const GLchar *geometry_source)
{
    if (geometry_source == nullptr)
    {
        return false;
    }

    gs = glCreateShader(GL_GEOMETRY_SHADER);
    glShaderSource(gs, 1, &geometry_source, NULL);
    glCompileShader(gs);
    check_shader_errors(gs, "geometry");

    return true;
}

void Shader::create_shader_program(bool geometry_shader_exists)
{
    // Create program
    id = glCreateProgram();
    glAttachShader(id, fs);
    glAttachShader(id, vs);
    if (geometry_shader_exists)
    {
        glAttachShader(id, gs);
    }
    glLinkProgram(id);

    // Check for linking error
    int params = -1;
    glGetProgramiv(id, GL_LINK_STATUS, &params);
    if (params != GL_TRUE)
    {
        LOG(Error) << "Could not link shader programme GL index " << id;
        print_programme_info_log(id);
    }
    // Delete shaders for they are no longer used
    glDeleteShader(vs);
    glDeleteShader(fs);
    if (geometry_shader_exists)
    {
        glDeleteShader(gs);
    }
}

void Shader::compile(const GLchar vertex_source, const GLchar fragment_source, const GLchar *geometry_source)
{
    compile_vertex_shader(vertex_source);
    compile_fragment_shader(fragment_source);
    bool gs_exists = compile_geometry_shader(geometry_source);
    create_shader_program(gs_exists);
}

Shader &Shader::use()
{
    glUseProgram(id);
    return *this;
}

void Shader::set_float(const GLchar *name, GLfloat value, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform1f(glGetUniformLocation(id, name), value);
}
void Shader::set_integer(const GLchar *name, GLint value, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform1i(glGetUniformLocation(id, name), value);
}
void Shader::set_vector2f(const GLchar *name, GLfloat x, GLfloat y, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform2f(glGetUniformLocation(id, name), x, y);
}
void Shader::set_vector2f(const GLchar *name, const glm::vec2 &value, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform2f(glGetUniformLocation(id, name), value.x, value.y);
}
void Shader::set_vector3f(const GLchar *name, GLfloat x, GLfloat y, GLfloat z, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform3f(glGetUniformLocation(id, name), x, y, z);
}
void Shader::set_vector3f(const GLchar *name, const glm::vec3 &value, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform3f(glGetUniformLocation(id, name), value.x, value.y, value.z);
}
void Shader::set_vector4f(const GLchar *name, GLfloat x, GLfloat y, GLfloat z, GLfloat w, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform4f(glGetUniformLocation(id, name), x, y, z, w);
}
void Shader::set_vector4f(const GLchar *name, const glm::vec4 &value, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniform4f(glGetUniformLocation(id, name), value.x, value.y, value.z, value.w);
}
void Shader::set_matrix4(const GLchar *name, const glm::mat4 &matrix, GLboolean use_shader)
{
    if (use_shader)
        use();
    glUniformMatrix4fv(glGetUniformLocation(id, name), 1, GL_FALSE, glm::value_ptr(matrix));
}
```

Texture class updated

We will change our texture class to simplify it. We get rid of the file loading logic, which will be integrated, for better architecture and memory management, in a new ressource loading class.

Our texture class will now work only for 2D textures. Breakout is a 2D game so no problem with that.

```
#ifndef TEXTURE_H
#define TEXTURE_H

#include <GL/glew.h>

class Texture {
public:
    Texture();
    ~Texture();

    void generate(int w, int h, unsigned char *image_data);
    void bind() const;

    GLuint id;

    inline GLuint get_image_format() const { return image_format; }
    inline void set_internal_format(GLuint _internal_format) { image_format = _internal_format; }
    inline void set_image_format(GLuint _image_format) { image_format = _image_format; }

private:
    int w, h;

    GLuint internal_format;
    GLuint image_format;
    GLuint wrap_s, wrap_t;
    GLuint filter_min, filter_max;
};

#endif
```

```
#include "texture.h"

Texture::Texture() : w(0), h(0), internal_format(GL_RGBA), image_format(GL_RGBA),
                     wrap_s(GL_REPEAT), wrap_t(GL_REPEAT), filter_min(GL_LINEAR), filter_max(GL_LINEAR)
{
    glGenTextures(1, &id);
}

Texture::~Texture()
{
}

void Texture::bind() const
{
    glBindTexture(GL_TEXTURE_2D, id);
}

void Texture::generate(int w, int h, unsigned char *image_data)
{
    w = _w;
    h = _h;
    glBindTexture(GL_TEXTURE_2D, id);
    glTexImage2D(
        GL_TEXTURE_2D,
        0,
        internal_format,
        w,
        h,
        0,
        image_format,
        GL_UNSIGNED_BYTE,
        image_data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap_s);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap_t);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter_min);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter_max);
     // Unbind texture
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Resource manager

It is often considered a more organized approach to create a single entity designed for loading game-related resources called a resource manager. There are several approaches to creating a resource manager; for this game we chose to use a singleton static resource manager that is (due to its static nature) always available throughout the project hosting all loaded resources and its relevant loading functionality.

Using a singleton class with static functionality has several advantages and disadvantages with its disadvantages mostly being losing OOP properties and losing control over construction/destruction. However, for relative small projects like these it is easy to work with.

```
#ifndef RESOURCE_MANAGER_H
#define RESOURCE_MANAGER_H

#include <map>
#include <string>

#include <GL/glew.h>

#include "texture.h"
#include "shader.h"

class ResourceManager
{
public:
    // Storage
    static std::map<std::string, Shader> shaders;
    static std::map<std::string, Texture> textures;

    // Management
    static Shader load_shader(const GLchar *vs_file, const GLchar *fs_file, const GLchar *gs_file, std::string name);
    static Shader get_shader(std::string name);
    static Texture load_texture(const GLchar *file, GLboolean alpha, std::string name);
    static Texture get_texture(std::string name);
    static void clear();

private:
    ResourceManager() { } // Singleton

    static Shader load_shader_from_file(const GLchar *vs_file, const GLchar *fs_file, const GLchar *gs_file = nullptr);
    static Texture load_texture_from_file(const GLchar *file, GLboolean alpha);
};

#endif
```

```
#include "resource_manager.h"
#include "log.h"

#include <iostream>
#include <sstream>
#include <fstream>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

// Instantiate static variables
std::map<std::string, Texture> ResourceManager::textures;
std::map<std::string, Shader> ResourceManager::shaders;


Shader ResourceManager::load_shader(const GLchar *vs_file, const GLchar *fs_file, const GLchar *gs_file, std::string
name)
{
    shaders[name] = load_shader_from_file(vs_file, fs_file, gs_file);
    return shaders[name];
}

Shader ResourceManager::get_shader(std::string name)
{
    return shaders[name];
}

Texture ResourceManager::load_texture(const GLchar *file, GLboolean alpha, std::string name)
{
    textures[name] = load_texture_from_file(file, alpha);
    return textures[name];
}

Texture ResourceManager::get_texture(std::string name)
{
    return textures[name];
}

void ResourceManager::clear()
{
    // (Properly) delete all shaders
    for (auto iter : shaders)
        glDeleteProgram(iter.second.id);
    // (Properly) delete all textures
    for (auto iter : textures)
        glDeleteTextures(1, &iter.second.id);
}

Shader ResourceManager::load_shader_from_file(const GLchar *vs_file, const GLchar *fs_file, const GLchar *gs_file)
{
    // 1. Retrieve the vertex/fragment source code from filePath
    std::string vertex_code;
    std::string fragment_code;
    std::string geometry_code;
    try
    {
        // Open files
        std::ifstream vs_file_stream(vs_file);
        std::ifstream fs_file_stream(fs_file);
        std::stringstream vs_str_stream, fs_str_stream;
        // Read file's buffer contents into streams
        vs_str_stream << vs_file_stream.rdbuf();
        fs_str_stream << fs_file_stream.rdbuf();
        // close file handlers
        vs_file_stream.close();
        fs_file_stream.close();
        // Convert stream into string
        vertex_code = vs_str_stream.str();
        fragment_code = fs_str_stream.str();
        // If geometry shader path is present, also load a geometry shader
        if (gs_file != nullptr)
        {
            std::ifstream gs_file_stream(gs_file);
            std::stringstream gs_str_stream;
            gs_str_stream << gs_file_stream.rdbuf();
            gs_file_stream.close();
            geometry_code = gs_str_stream.str();
        }
    }
    catch (std::exception e)
    {
        LOG(Error) << "ERROR::SHADER: Failed to read shader files";
    }

    Shader shader;
    shader.compile(vertex_code.c_str(), fragment_code.c_str(), gs_file != nullptr ? geometry_code.c_str() : nullptr);
    return shader;
}

Texture ResourceManager::load_texture_from_file(const GLchar *file, GLboolean alpha)
{
    // Create Texture object
    Texture texture;
    if (!alpha)
    {
        texture.set_internal_format(GL_RGB);
        texture.set_image_format(GL_RGB);
    }
    // Load image
    int width, height;
    int channels;
    stbi_set_flip_vertically_on_load(true);
    unsigned char* image_data = stbi_load(file, width, height, &channels, texture.get_image_format());
    if (!image_data)
    {
        LOG(Error) << "Could not load " << file;
    }
    if ((width & (width - 1)) != 0 || (height & (height - 1)) != 0)
    {
        LOG(Warning) << "Texture " << file << " is not power-of-2 dimensions";
    }
    // Now generate texture
    texture.generate(width, height, image_data);
    // And finally free image data
    stbi_image_free(image_data);
    return texture;
}
```

Window update

We update the window class to initialize keyboard handling and optimize FPS counter

```
#ifndef WINDOW_H
#define WINDOW_H

#include <GL/glew.h>
#include <GLFW/glfw3.h>

#include "log.h"

extern LogConfig LOG_CONFIG;
extern const GLuint SCREEN_WIDTH;
extern const GLuint SCREEN_HEIGHT;

extern void glfw_error_callback(int error, const char *description);
extern void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode);

class Window
{
public:
    Window();
    virtual ~Window();

    void init();
    void log_gl_params();
    bool should_close();
    void handle_close();
    void update_fps_counter(double dt);
    void clear();
    void swap_buffer();

private:
    GLFWwindow *glfw_window;
    double previous_seconds;
    double current_seconds;
    int frame_count;

    Window(const Window &);
    Window &operator=(const Window &);
};

#endif
```

```
void Window::init()
{
    ...
    // Create a windowed mode window and its OpenGL context
    glfw_window = glfwCreateWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Breakout", nullptr, nullptr);
    if (!glfw_window)
    {
        glfwTerminate();
    }

    // Make the window's context current
    glfwMakeContextCurrent(glfw_window);
    glfwSetKeyCallback(glfw_window, key_callback);
    ...

    glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
    glEnable(GL_CULL_FACE);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    previous_seconds = 0.0;
    current_seconds = 0.0;
}

void Window::update_fps_counter(double dt)
{
    double elapsed_seconds;

    current_seconds += dt;
    elapsed_seconds = current_seconds - previous_seconds;
    /* limit text updates to 4 per second */
    if (elapsed_seconds > 0.25)
    {
        previous_seconds = current_seconds;
        char tmp[128];
        double fps = (double)frame_count / elapsed_seconds;
        sprintf(tmp, "OpenGL @ fps: %.2f", fps);
        glfwSetWindowTitle(glfw_window, tmp);
        frame_count = 0;
    }
    frame_count++;
};
```

Main

Now we can setup our framework in the main class.

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <cmath>

#include "log.h"
#include "window.h"
#include "game.h"


// ======================= Globals =======================

LogConfig LOG_CONFIG = {};
const GLuint SCREEN_WIDTH = 800;
const GLuint SCREEN_HEIGHT = 600;
Game game(SCREEN_WIDTH, SCREEN_HEIGHT);

void glfw_error_callback(int error, const char *description)
{
    LOG(Error) << "GLFW error code " << error << " | msg: " << description;
};

void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    if (key >= 0 && key < 1024)
    {
        if (action == GLFW_PRESS)
            game.keys[key] = GL_TRUE;
        else if (action == GLFW_RELEASE)
            game.keys[key] = GL_FALSE;
    }
}

// ======================= Main =======================

int main(void)
{
    Window window;

    // Init logging
    LOG_CONFIG.reporting_level = Debug;
    LOG_CONFIG.restart = true;
    if (LOG_CONFIG.restart)
    {
        Log::restart();
    }

    window.init();
    window.log_gl_params();

    game.init();
    game.state = GameActive;

    double dt = 0;
    double last_frame = 0;

    while (!window.should_close()) {
        double current_frame = glfwGetTime();
        dt = current_frame - last_frame;
        last_frame = current_frame;

        glfwPollEvents();
        game.process_input(dt);
        game.update(dt);

        window.clear();
        game.draw();
        window.swap_buffer();
        window.update_fps_counter(dt);
    }

    glfwTerminate();
    return 0;
}
```