1. An OpenGL Crash Course (c) Stephen J. Guy, 2018

The revolution in the quality of modern 3D games has been largely driven by a rapid improvement in graphics rendering hardware. Today, most computers have a specialized Graphics Processing Unit (GPU) designed to accelerate common tasks relating to 2D and 3D graphics. Here, we show how to use two key libraries to enable easy access to hardware accelerated 3D programs. The first is SDL, which allows a cross platform way to create windows, play sounds, and access the mouse and keyboard. The second is OpenGL, which is an API designed for interfacing with the GPU.

1.1 A Basic OpenGL Program

Most OpenGL programs follows the same basic structure, they first load common files and data which will be used for rendering and then have a central *game loop* that displays a rendering of the data once per time through the loop. This loop also gives the program a chance to update the state of the (virtual) world, and respond to any mouse or keyboard events. Here, we'll start with the progam `ColoredTriangle.cpp` that will draw a single 2D triangle.

8 ⟨*ColoredTriangle.cpp* 8⟩≡

```
⟨GLAD, SDL, and OpenGL Includes 9a⟩
⟨Basic 2D Vertex Shader 9c⟩
⟨Basic Fragment Shader 10⟩
⟨Globals 9b⟩

int main(int argc, char *argv[]) {
  ⟨Initialize SDL 11⟩
  ⟨Initialize OpenGL through GLAD 12a⟩
  ⟨Load Models and Shaders 12b⟩
  ⟨Main Game Loop 15a⟩
  ⟨Cleanup 17⟩
  return 0;
}
```

There are several header files needed to allow us to access SDL and OpenGL:

9a ⟨*GLAD, SDL, and OpenGL Includes* 9a⟩≡ (8 18 25 31a)
```
#include "glad/glad.h"  //Include order can matter here
#if defined(__APPLE__) || defined(__linux__)
 #include <SDL2/SDL.h>
 #include <SDL2/SDL_opengl.h>
#else
 #include <SDL.h>
 #include <SDL_opengl.h>
#endif
```

We will also need a global value to store if we are in fullscreen mode (vs. windowed rendering), along with size of the window.

9b ⟨*Globals* 9b⟩≡ (8 18 25 31a) 13a▸
```
bool fullscreen = false;
int screenWidth = 800;
int screenHeight = 600;
```

Definition 1.1 (Shader Programs) A shader is a custom program that runs on the GPU. Historically, shaders were used primarily to compute the colors of each pixel according to custom lighting (or shading) models. More recently, shaders have evolved to be used to perform any computation relating to rendering graphics. Almost every graphics program will have a *vertex shader* to compute the 2D positions of the 3D vertices and a *pixel shader* to compute the final color of each pixel. Shaders can also be used to create, modify, and distort 3D geometry, pre-compute key elements of lighting equations, and even run arbitrary computations.

We will start with the simplest possible shaders. First, our vertex shader takes a 2D position of vertices as input, and sets the z-component as 0 and homogeneous coordinate as 1. This new vector is passed to special output `gl_Position`, which is automatically interpreted by OpenGL to determine where to draw each vertex. We also will set a custom output called `Color` which sends the current color (as read from the GPU data) to the fragment shader.

9c ⟨*Basic 2D Vertex Shader* 9c⟩≡ (8)
```
const GLchar* vertexSource =
"#version 150 core\n"
"in vec2 position;"
"in vec3 inColor;"
"out vec3 Color;"
"void main() {"
"   Color = inColor;"
"   gl_Position = vec4(position, 0.0, 1.0);"
"}";
```

By default, OepnGL will interpolate any values computed in the vertex shader along each pixel in the triangle passing them to the fragment shader. Our fragment shader will simply send these (interpolated) colors directly along to be rendered without modification.

10 ⟨*Basic Fragment Shader* 10⟩≡ ⟨8 19⟩

```
const GLchar* fragmentSource =
    "#version 150 core\n"
    "in vec3 Color;"
    "out vec4 outColor;"
    "void main() {"
    "   outColor = vec4(Color, 1.0);"  //(Red, Green, Blue, Alpha)
    "}";
```

For more complex shaders, we would typically load them from files rather than directly writing them to string.

Tip **Fragment vs Pixel** Conceptually a *fragment* shader is computing the colors of a pixel. We call it a fragment shader because it tells us how to color (shade) each pixel-sized fragment of a triangle. In practice, multiple fragments may need to be computed to determine the final value of a pixel. For example, rendering the view out the window of a car with tinted windows involves shading one fragment for the window and one fragment for whatever object is visible outside the window.

Definition 1.2 (Homogeneous coordinates) Typically we represent 3D positions and vectors in OpenGL with a *homogeneous coordinate* of four floats labeled x, y, z and w. When $w \neq 0$ we interpert (x, y, z, w) as the 3D point (x/w, y/w, z/w), when $w = 0$ we interpert (x, y, z, w) as the 3D vector representing the direction (x, y, z). Using a 4D vector to represent 3D structures allows additional flexibility is representing transformations as matrix multiplications.

We will use the Simple Direct Media Layer 2 (SDL2) library in order to comunicate with the opperating systems for tasks like creating a window and reading keyboard inputs. In order to use SDL we must initalize it, and connect the library to OpenGL. Here we request to use OpenGL version 3.2 which is old enough to be broadly supported by many systems, but recent enough to have important features needed for modern game engines.

11 ⟨*Initialize SDL* 11⟩≡ (8 18 25 31a)

```
SDL_Init(SDL_INIT_VIDEO);  //Initialize Graphics (for OpenGL)

//Print the version of SDL we are using
SDL_version comp; SDL_version linked;
SDL_VERSION(&comp); SDL_GetVersion(&linked);
printf("\nCompiled against SDL version %d.%d.%d\n", comp.major, comp.minor, comp.patch);
printf("Linked SDL version %d.%d.%d.\n", linked.major, linked.minor, linked.patch);

//Ask SDL to get a recent version of OpenGL (3.2 or greater)
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);

//Create a window (offsetx, offsety, width, height, flags)
SDL_Window* window = SDL_CreateWindow("My OpenGL Program", 100, 100,
                                      screenWidth, screenHeight, SDL_WINDOW_OPENGL);
if (!window){
  printf("Could not create window: %s\n", SDL_GetError());
  return EXIT_FAILURE; //Exit as SDL failed
}
float aspect = screenWidth/(float)screenHeight; //aspect ratio needs update on resize

SDL_GLContext context = SDL_GL_CreateContext(window); //Bind OpenGL to the window
```

Tip **Window Creation.** The last argument to the function `SDL_CreateWindow()` can take a variety of flags. Passing `SDL_WINDOW_FULLSCREEN|SDL_WINDOW_OPENGL` starts the window as fullscreen whereas passing `SDL_WINDOW_RESIZABLE|SDL_WINDOW_OPENGL` allows the window to be resized by the user (though care should be taken to update the aspect ratio upon resize). A common approach used in games is to set a borderless window exactly the resolution of the desktop to allow easy switching between the OpenGL application and other windows. This can be accomplished in SDL by passing the flags of `SDL_WINDOW_FULLSCREEN_DESKTOP|SDL_WINDOW_OPENGL` (and passing 0 instead of `screenWidth` and `screenHeight`).

A last step before we can actually make calls to OpenGL functions is to ensure that the pointers to all OpenGL calls are loaded and refer to the active OpenGL instance. Here, we use the GLAD loader generator (https://github.com/Dav1dde/glad) to perfrom this loading process. Be sure to compile glad.cpp together with the rest of the program in order for the library to work.

12a ⟨*Initialize OpenGL through GLAD* 12a⟩≡ (8 18 25 31a)

```
if (gladLoadGLLoader(SDL_GL_GetProcAddress)){
  printf("OpenGL loaded\n");
  printf("Vendor:   %s\n", glGetString(GL_VENDOR));
  printf("Renderer: %s\n", glGetString(GL_RENDERER));
  printf("Version:  %s\n", glGetString(GL_VERSION));
}
else {
  printf("ERROR: Failed to initialize OpenGL context.\n");
  return -1;
}
```

In order to ensure good preformance its often best to send data to the GPU memory before the main loop actually starts, especially for data that does not frequently change as the application is running. We have three types of data we will transfer to the GPU. First, the shader programs which tell the GPU how to render our scene. Second, the data relating to the models we wish to display such as the vertex positions and colors; this model data will be stored in space on the GPU called a Vertex Buffer Object (VBO). Lastly, we need to send a mapping between which parts of our data in the VBO correpsond to which variables in our shaders; this information is also stored on the GPU in what is called a Vertex Array Object (VAO).

12b ⟨*Load Models and Shaders* 12b⟩≡ (8)

```
⟨Load and Compile Shaders 13b⟩
⟨Load Model Data 13c⟩
⟨Create VBO 14a⟩
⟨Create and Bind VAO 14b⟩
⟨Bind Attributes to VBO 14c⟩
```

We first make a helper function to load the shader and ensure it compiles without error.

13a ⟨*Globals* 9b⟩+≡ (8 18 25 31a) ◂9b 20▸
```
void loadShader(GLuint shaderID, const GLchar* shaderSource){
  glShaderSource(shaderID, 1, &shaderSource, NULL);
  glCompileShader(shaderID);

  //Let's double check the shader compiled
  GLint status;
  glGetShaderiv(shaderID, GL_COMPILE_STATUS, &status); //Check for errors
  if (!status){
    char buffer[512]; glGetShaderInfoLog(shaderID, 512, NULL, buffer);
    printf("Shader Compile Failed. Info:\n\n%s\n",buffer);
  }
}
```

We then load and compile the vertex and fragment shaders, and link them into a single shader program that computes the final output color `outColor`.

13b ⟨*Load and Compile Shaders* 13b⟩≡ (12b 18 25 31a)
```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
loadShader(vertexShader, vertexSource);
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
loadShader(fragmentShader, fragmentSource);

//Join the vertex and fragment shaders together into one program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor"); // set output
glLinkProgram(shaderProgram); //run the linker
```

We should next load the model data that we wish to render. Typically we should load this data from a model file, but for now we'll hardcode all the information. Here, we'll use the convention of storing data for each vertex sequentially. For each vertex, we first store the 2D X and Y positions and then store the RGB color. The below code loads the data for a single triangle with a different color on each vertex.

13c ⟨*Load Model Data* 13c⟩≡ (12b)
```
GLfloat vertices[] = {
  0.0f,  0.5f, 1.0f, 0.0f, 0.0f, // Vertex 1: postion = (0,.5) color = Red
  0.5f, -0.5f, 0.0f, 1.0f, 0.0f, // Vertex 2: postion = (.5,-.5) color = Green
  -0.5f, -0.5f, 0.0f, 0.0f, 1.0f}; // Vertex 3: postion = (-.5,-.5) color = Blue
```

Because our shader wil pass the x and y values directly as 2D coordinates for rendering, these positions need to be in *Normalized Device Coordinate* (NDC) space. Normalized Device Coordinates are in a special space where (-1,-1) is the bottom left of the image and (1,1) the top right; specifying points in NDC allows us to do all our math without worrying about the exact size of the image in pixels.

Tip **Representing Colors.** Colors are typically stored as a combination of red, green, and blue components. By combining different amounts of these three components, a wide range of different colors can be specified. For example, equal parts red and green will create yellow, and equal parts blue and red create magenta.

We now need to move the data from the `vertices` array (which lives on the CPU) to memory which is on the GPU. Once on the GPU, the data is stored in a special structure called a Vertex Buffer Object (VBO). We must first allocate space for a VBO on the GPU and then copy the data over.

14a ⟨*Create VBO* 14a⟩≡ (12b 18)

```
GLuint vbo;
glGenBuffers(1, &vbo);  //Create 1 buffer called vbo
glBindBuffer(GL_ARRAY_BUFFER, vbo); //(Only one buffer can be bound at a time)
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Here, we use `GL_STATIC_DRAW` to hint to the GPU that this data is unlikely to change each frame allowing OpenGL to better optimize the usage of GPU memory. If the data is changing infrequntly `GL_DYNAMIC_DRAW` may be better, and `GL_STREAM_DRAW` is best used when the data changes frequently.

We can now create a Vertex Array Object (VAO), which stores the mapping between the data stored in the VBO and the inputs to our shaders. First we must create and bind a VAO.

14b ⟨*Create and Bind VAO* 14b⟩≡ (12b 22a 25 31a)

```
GLuint vao;
glGenVertexArrays(1, &vao); //Create a VAO
glBindVertexArray(vao); //Bind the above created VAO to the current context
```

Then, we can set the mappings from the VBO to the fragment shader. First, we use `glVertexAttribPointer()` to record that a vertex position information is two floats long and starts every fifth element. Likewise, we record that the vertex color is three floats long, starts every five position, and that the first vertex comes after 2 floats.

14c ⟨*Bind Attributes to VBO* 14c⟩≡ (12b)

```
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), 0);
//(above params: Attribute, vals/attrib., type, isNormalized, stride, offset)
glEnableVertexAttribArray(posAttrib); //Mark the attribute's location as valid

GLint colAttrib = glGetAttribLocation(shaderProgram, "inColor");
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
                      5*sizeof(float), (void*)(2*sizeof(float)));
glEnableVertexAttribArray(colAttrib);

glBindVertexArray(0); //Unbind the VAO so we don't accidentally modify it
```

With our shaders compiled, linked, and loaded, the model data loaded onto the VBO, and the mappings set-up on the VAO we are now ready to set-up the main game loop. This is the heart of our program, looping continuously until the program is stopped (and `quit` is set to true). Each time through this loop we first look to see if there is any new SDL keyboard or mouse events to process, and then draw the actual graphics. As a last step, we swap buffers to display the rendering.

15a ⟨*Main Game Loop* 15a⟩≡ (8)
```
SDL_Event windowEvent;
bool quit = false;
while (!quit){
  ⟨Process Keyboard and Mouse Events 15b⟩
  ⟨Draw Content 16⟩
  SDL_GL_SwapWindow(window); //Double buffering
}
```

Tip **Double buffering.** Double buffering is a common graphics technique to reduce the look of "tearing" that can occur when the screen gets refreshed in the middle of an image being rendered. To avoid this, we render the image to an unseen back buffer, which is then swapped to be the front buffer between each frame. The command `SDL_GL_SwapWindow()` waits to update the new image being drawn to the screen until after the current image has been fully displayed. This process of waiting for the monitor to update the image is called *vsynch*. This fixes tearing artifacts at the cost of locking the games update rate to the monitor's display rate, and can cause some additional lag in processing input. *Triple buffering* uses a third buffer in order to better balance these tradeoffs.

There are various mouse, screen or keyboard events that we might be interested in our game responding to. For example, we want to exit our progam run any needed cleanup rutines whenever the user closes the window (creating the `SDL_QUIT` event) or hits the escape key (creating an `SDL_KEYUP` event). For some events there are additional paramters to check to understand the details of the event. For example, for `SDL_KEYUP` event, we can compare `windowEvent.key.keysym.sym` to a list of known keycodes in oder to see exactly what key was pressed. A list of keycodes supported by SDL can be found at: `https://wiki.libsdl.org/SDL_Keycode`. For now, we want to exit when escaped is pressed, and toggle fullscreen when the f-key is pressed.

15b ⟨*Process Keyboard and Mouse Events* 15b⟩≡ (15a 18 25 31a)
```
while (SDL_PollEvent(&windowEvent)){
  if (windowEvent.type == SDL_QUIT) quit = true; //Exit Game Loop
  if (windowEvent.type == SDL_KEYUP && windowEvent.key.keysym.sym == SDLK_ESCAPE)
    quit = true; //Exit Game Loop
  if (windowEvent.type == SDL_KEYUP && windowEvent.key.keysym.sym == SDLK_f){
    fullscreen = !fullscreen;
    SDL_SetWindowFullscreen(window, fullscreen ? SDL_WINDOW_FULLSCREEN : 0);
  }
}
```

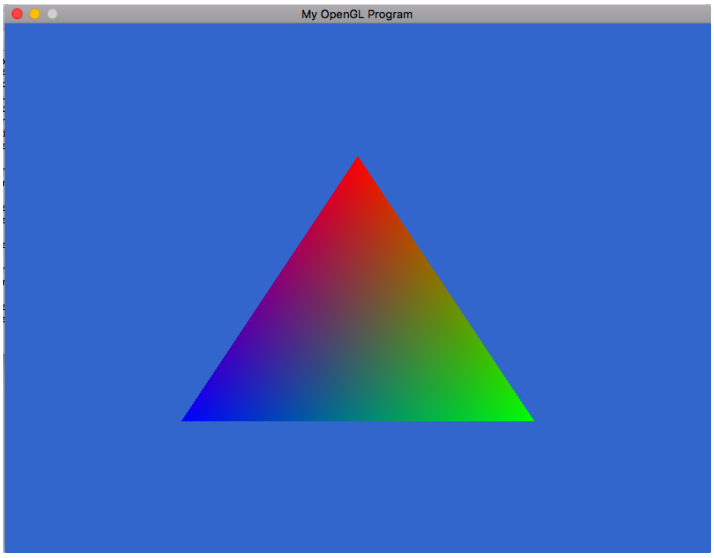

Figure 1.1: A Colorful Triangle rendered by *3DTriangle.cpp*

Tip **KeyUp vs KeyDown Events.** When looking for key presses we are using SDL_KEYUP, which only gets triggered when the user releases a key. You can also look for SDL_KEYDOWN events, but on most operating systems the key_down events get generated repeatedly as long as the key is pressed. When we want each press of a button to trigger an event only once it's best to look for SDL_KEYUP event.

With the vertex data loaded on the VBO, the vertex and pixel shaders compiled, linked, and loaded, and the VAO storing the mappings from the data to the shaders we are now ready to render our model. We first clear the screen to get a fresh image each frame, and then load the VAO. To draw the contents of the VBO, we can call glDrawArrays. Here, we specify that we are drawing 3 verticies worth of traingles (i.e., one triangle).

16 ⟨*Draw Content* 16⟩≡ (15a)

```
glClearColor(.2f, 0.4f, 0.8f, 1.0f);  // Clear the screen to blue
glClear(GL_COLOR_BUFFER_BIT);

glUseProgram(shaderProgram); //Set the active shader program
glBindVertexArray(vao);  //Bind the VAO for the shader(s) we are using
glDrawArrays(GL_TRIANGLES, 0, 3);
```

We've made it! The output of the code should be a multi-colored triangle as shown in Figure 1.1. Before exiting, its best to do some cleanup to free up the memory we've reserved, and safely destroy the SDL and OpenGL contexts we have created.

17 ⟨*Cleanup* 17⟩≡ (8 18 31a)

```
glDeleteProgram(shaderProgram);
glDeleteShader(fragmentShader);
glDeleteShader(vertexShader);
glDeleteBuffers(1, &vbo);
glDeleteVertexArrays(1, &vao);
SDL_GL_DeleteContext(context);
SDL_Quit();
```

Tip **Compiling Graphics Applications.** When compiling graphics applications a key step is including all of the relevant framework files, header files, and .c/.cpp files needed to actually use the libraries we are relying on. The way to accomplish this varies based on your operating system. For example on Mac OS we can compile *ColoredTriangle.cpp* form the command line with the following command:

```
g++ ColoredTriangle.cpp -x c glad/glad.c -g -F/Library/Frameworks
  -framework SDL2 -framework OpenGL -o ColoredTriangle
```

Whereas on linux the command would be:

```
g++ ColoredTriangle.cpp -x c glad/glad.c -g -lSDL2 -lSDL2main -lGL -ldl
  -I/usr/include/SDL2/ -o square
```

Regardless of your OS, if is important that the dynamically linked aspects of these libraries (e.g., the DLLs in windows) are installed in locations that are accessible to your executable.

1.2 Moving to 3D

We are now ready to move on to rendering 3D models. The same basic framework we used in the 2D triangle example can be easily extended to support 3D rendering. Just as in the above example, we need to include the header files needes for Glad, OpenGL, and SDL; load and compile our shaders, move data to VBOs, set-up VAO mappings, etc.

18 ⟨*Cube3D.cpp* 18⟩≡

```
⟨GLAD, SDL, and OpenGL Includes 9a⟩
⟨Basic 3D Shaders 19⟩
⟨Globals 9b⟩

int main(int argc, char *argv[]) {
  ⟨Initialize SDL 11⟩
  ⟨Initialize OpenGL through GLAD 12a⟩

  ⟨Load Cube Model 21⟩
  ⟨Load and Compile Shaders 13b⟩
  ⟨Create VBO 14a⟩
  ⟨Set-up Cube VAO mappings 22a⟩

  glEnable(GL_DEPTH_TEST);

  SDL_Event windowEvent;
  bool quit = false;
  while (!quit){
    ⟨Process Keyboard and Mouse Events 15b⟩
    ⟨Draw Cube Scene 22b⟩
    SDL_GL_SwapWindow(window); //Double buffering
  }
  ⟨Cleanup 17⟩
  return 0;
```

```
}
```

Tip **Transforming with Matrices** A 4x4 matrix in homogeneous coordinates can represent any affine transformation (e.g., any combination of scaling, rotating, and shearing) to a 3D point or 3D vector simply by a matrix multiplication. Our shaders will use this fact to compute the position of each vertex relative to the camera as a matrix multiply (e.g., `view*model*vec3(position,1.0)`). A 4x4 homogeneous matrix can also represent perspective transformations (i.e. transformations which shrink objects based on their distance from the camera). Affine transformation matrices for 3D space have a bottom row of [0 0 0 1], projective transformation matrices will have replaces some zeros in the bottom row. The most common bottom row of a projection matrix is [0 0 1 0] which, when combined with the definition of homogeneous coordinates, results in a division of every coordinate by Z (this distance of the vertex form the camera).

One key change is the shaders themselves. Our vertex shader now has to project the 3D positions of the triangle vertices in our models to 2D screen coordinates. We do this through the multiplication of several matrices: a `model` matrix which stores the location and orientation of our cube, a `view` matrix which stores the location and orientation of the camera, and a `proj` matrix which encodes the 3D to 2D projection. Each of these matrices can be viewed as a transformation which changes coordinate systems (`model` transforms from an the arbitrary coordinate system the model was created in to the world coordinate system, `view` transforms from world coordinates to a position relative to the camera, and `proj` transforms from 3D to 2D). Our vertex shader simply multiplies these three transformation matricies together, and then multiplies the resulting matrix with the orginal 3D position of each vertex in the model to find its new 2D position. Our fragment shader is unchanged from before.

19 ⟨*Basic 3D Shaders* 19⟩≡ (18)

```
// Shader sources
const GLchar* vertexSource =
    "#version 150 core\n"
    "in vec3 position;"
    "in vec3 inColor;"
    "out vec3 Color;"
    "uniform mat4 model;"
    "uniform mat4 view;"
    "uniform mat4 proj;"
    "void main() {"
    "   Color = inColor;"
    "   gl_Position = proj * view * model * vec4(position,1.0);"
    "}";
```

⟨*Basic Fragment Shader* 10⟩

Tip **Uniforms in Shaders.** Shader values that are marked as `uniform` don't change across a model (in constrast to e.g., `position` which is different each vertex). Because all vertices in a given model share the same `model`, `view`, and `proj` matrices we mark them an `uniform` in our vertex shader. The values for uniforms are set through specialized calls such as `glUniformMatrix4fv()` rather than being specified through the

VBO/VAO.

In order to compute the values of `model`, `view`, and `proj` matrices we will make use of the excellent GLM library. We need to include the relevant header files to make use of this library later on.

20 ⟨*Globals* 9b⟩+≡ (8 18 25 31a) ◁13a

```
#define GLM_FORCE_RADIANS //ensure we are using radians
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/type_ptr.hpp"
```

Tip **Triangles.** While there are many options for storing 3D models, far and way the most common approach used in games is to represent them as a collection of 3D triangles. This is because any surface, no matter how complex or detailed, can be well-approximated as series of triangles. Here, we'll show results on a model of a cube made of up 12 triangles, but the method is identical for more complex models.

Specifying the cube model takes a lot more vertices than the single triangle example in Section 1.1. Each of the six faces of the cube is made up of two triangles, with each triangle consisting of three verticies (leading to 48 total vertices to specify). Below we give the position (X,Y,Z) and color (Red, Green, Blue) for each vertex. We also specify a special 2D coordinate (U,V), which we can make use of if we want to add a texture to the model.

21 ⟨*Load Cube Model* 21⟩≡ (18 25)

```
GLfloat vertices[] = {
// X      Y     Z     R     G      B      U      V
 -0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, //Red face
  0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
  0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
  0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
 -0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f,
 -0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,

 -0.5f, -0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, //Green face
  0.5f, -0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f,
  0.5f,  0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,
  0.5f,  0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,
 -0.5f,  0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
 -0.5f, -0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,

 -0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, //Yellow face
 -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
 -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
 -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
 -0.5f, -0.5f,  0.5f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f,
 -0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f,

  0.5f,  0.5f,  0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, //Blue face
  0.5f,  0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f,
  0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f,
  0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f,
  0.5f, -0.5f,  0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
  0.5f,  0.5f,  0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f,

 -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, //Black face
  0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
  0.5f, -0.5f,  0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
  0.5f, -0.5f,  0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
 -0.5f, -0.5f,  0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
```

```
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,

    -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, //White face
     0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
     0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f,
     0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f,
    -0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f
  };
```

As before, we also need to use a VAO to specify how the data in the VBO is mapped to the inputs of the shaders. The set-up is similar to our previous VAO, though now the position takes up three floats, and the stride between subsequent vertices is 8 floats.

22a ⟨*Set-up Cube VAO mappings* 22a⟩≡ (18)

```
  ⟨Create and Bind VAO 14b⟩
  GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
  glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), 0);
  //(above params: Attribute, vals/attrib., type, isNormalized, stride, offset)
  glEnableVertexAttribArray(posAttrib); //Set attribute location as active

  GLint colAttrib = glGetAttribLocation(shaderProgram, "inColor");
  glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
                        8*sizeof(float), (void*)(3*sizeof(float)));
  glEnableVertexAttribArray(colAttrib);


  glBindVertexArray(0); //Unbind the VAO
```

Drawing the actual cube is accomplished through a call to `glDrawArrays()`. Here, we also use the function `SDL_GetTicks()` to keep track of how much time has past since the start of the code. We will use this value to help control the animation.

22b ⟨*Draw Cube Scene* 22b⟩≡ (18 25)

```
  // Clear the screen to default color
  glClearColor(.2f, 0.4f, 0.8f, 1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  ⟨Set-up transformation matrices 23a⟩

  glUseProgram(shaderProgram); //Set the active shader program
  glBindVertexArray(vao);  //Bind the VAO for the shaders we are using
  glDrawArrays(GL_TRIANGLES, 0, 36); //Number of vertices
```

23a ⟨*Set-up transformation matrices* 23a⟩≡ (22b 34b)
```
float time = SDL_GetTicks()/1000.f;
⟨Set model matrix 23b⟩
⟨Set view matrix 23c⟩
⟨Set proj matrix 24⟩
```

> Tip **Model Storage.** The `glDrawArrays()` command supports many storage formats for the model data. Here, we are using `GL_TRIANGLES`, which indicates that the first set of 3 vertices make the first triangle, the second set of vertices make the second triangle, and so on. Also common is to store triangle using `GL_TRIANGLES_STRIP` which treats every consecutive set of three points as a new triangle. This allows models to use almost 3X less storage, though we may need to re-arrange the vertices to support this format.

Before making calling `glDrawArrays()`, we need to set up the `model`, `view`, and `proj` matrices, which will be used in the process of rendering. Here, we use the GLM library to create a matrix will rotate the model by an amount based on how much time has passed. After creating this matrix, we use glUniformMatrix4fv() to associate the location of the matrix with the input of the shader that is expecting the `model` matrix (as found by the call to `glGetUniformLocation()`).

23b ⟨*Set* `model` *matrix* 23b⟩≡ (23a)
```
glm::mat4 model;
model = glm::rotate(model,time * 3.14f/2,glm::vec3(0.0f, 1.0f, 1.0f));
model = glm::rotate(model,time * 3.14f/4,glm::vec3(1.0f, 0.0f, 0.0f));
GLint uniModel = glGetUniformLocation(shaderProgram, "model");
glUniformMatrix4fv(uniModel, 1, GL_FALSE, glm::value_ptr(model));
```

We can set up a virtual camera which specifies where the user is viewing the scene from. Here, we hard-code the camera's position, the direction it is looking at, and the up direction. We then use the GLM library to compute a `view` matrix that matches the camera properties, which we send to the shader.

23c ⟨*Set* `view` *matrix* 23c⟩≡ (23a)
```
glm::mat4 view = glm::lookAt(
  glm::vec3(3.0f, 0.0f, 0.0f),  //Cam Position
  glm::vec3(0.0f, 0.0f, 0.0f),  //Look at point
  glm::vec3(0.0f, 0.0f, 1.0f)); //Up
GLint uniView = glGetUniformLocation(shaderProgram, "view");
glUniformMatrix4fv(uniView, 1, GL_FALSE, glm::value_ptr(view));
```

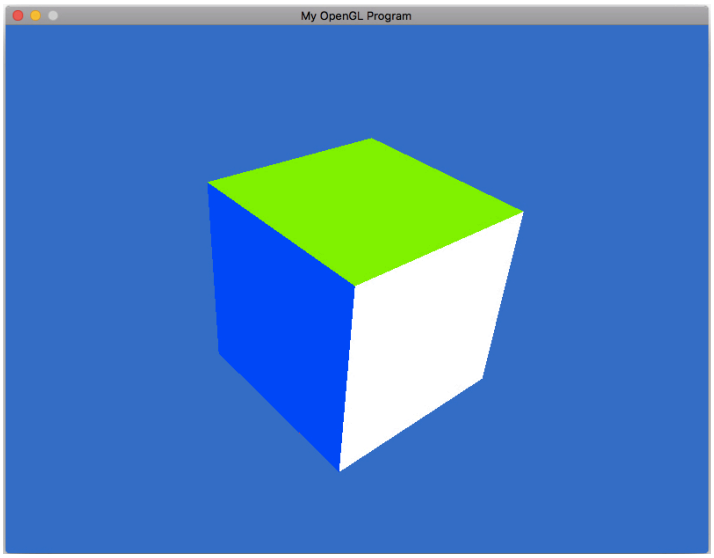

Figure 1.2: A spinning 3D cube rendered by *3DCube.cpp*

> Tip **Dynamic Cameras.** The values for the cameras positon and direction should typically be set by user input. One common approach is to map keyboard input (e.g., W, A, S and D keys) to the X and Y camera positions, and map mouse input to rotate the point the camera is looking at.

Our virtual camera has additional intrinsic parameters stored in the `proj` matrix that represent the view frustum, including the camera's field of view (FOV), the aspect ratio of the display, and the near and far plane (which specify the closest and furtherest away thing which will be rendered).

24 ⟨*Set* `proj` *matrix* 24⟩≡ (23a)

```
glm::mat4 proj = glm::perspective(3.14f/4, aspect, 1.0f, 10.0f);
                                  //FOV, aspect ratio, near, far
GLint uniProj = glGetUniformLocation(shaderProgram, "proj");
glUniformMatrix4fv(uniProj, 1, GL_FALSE, glm::value_ptr(proj));
```

Definition 1.3 (Viewing Frustum) A viewing frustum is a truncated, pyramid-like shape that captures the volume of space that your camera can see. The left-right and the top-bottom opening angles of this frustum are captured by the camera's FOV, the aspect ratio it is displaying at. The near and far planes can be set to any values, but the computer uses a fixed number of bits to represent the depth of an object between these two panes. If the near plane is too close to 0, or the far plane is too big, then the lack of precision will lead to artifacts relating to difficulties resolving which triangle is closest to the camera for a given pixel.

That's it. With the rendering matrices computed, and the VBOs, VAOs, shaders and uniforms all set, we can now render a 3D cube. The result is shown in Figure 1.2.

1.3 Lighting the Scene

We can greatly increase the quality of our 3D rendering by improving the lighting model used to compute the color of each pixel. While generating physically realistic lighting is a complex tasks, there are two key components of lighting that are most important to capture, the diffuse falloff and specular highlights. The diffuse falloff in lighting captures the diminishing effect a light source has on a surface as its angle approaches perpendicular to the surface being lit (this same effect causes seasons on earth). Specular highlights capture the bright, local reflection of a light source off of a shiny material.

The overall structure for our lighting code is the same as the other examples in this chapter.

25 ⟨*CubeLit.cpp* 25⟩≡

```
⟨GLAD, SDL, and OpenGL Includes 9a⟩
⟨Globals 9b⟩
⟨Phong 3D Vertex Shader 27c⟩
⟨Phong 3D Fragment Shader 29⟩

int main(int argc, char *argv[]) {
  ⟨Initialize SDL 11⟩
  ⟨Initialize OpenGL through GLAD 12a⟩

  ⟨Load Cube Model 21⟩
  ⟨Load Cube Normals 26a⟩
  ⟨Load and Compile Shaders 13b⟩
  ⟨Create and Bind VAO 14b⟩
  ⟨Create and Bind 2 VBOs 26b⟩
  ⟨Set-up 1st VBO mappings 27a⟩
  ⟨Set-up 2nd VBO mappings 27b⟩

  glEnable(GL_DEPTH_TEST);

  SDL_Event windowEvent;
  bool quit = false;
  while (!quit){
    ⟨Process Keyboard and Mouse Events 15b⟩
```

```
    ⟨Draw Cube Scene 22b⟩
    SDL_GL_SwapWindow(window); //Double buffering
  }
  ⟨Cleanup with 2 VBOs 30⟩
  return 0;
}
```

There are two main aspects we need to update from the *3DCube.cpp* to enable dynamic lighting. First, we need to store the local curvature of the model in the form of *normals*, that is, vectors which are perpendicular to the model. Second, we need to update our shader code to use these normals, along with other information about the geometry and lighting to better compute a color for each pixel.

Typically, we would store the normals for a model in the same VBO as the rest of the information (color, vertex postion, etc.). However, for illustration purposes, let's look at an altenative option where we store all the normals in a separate VBO. This gives us a chance to see multiple VBOs in action. First, we need to load the one 3D normal for each vertex.

Definition 1.4 (Normals) When representing a 3D model we typically care about two aspects of the models geometry: one, it's overall shape (i.e., where its surface is located) and two, the curvature of the surface. The location of the surface tells us which pixels to light up, the curvature tells of how to light these pixel (i.e., how bright the local lighting is). There are many ways to represent the local curvature of a model, but the most common approach in graphics it to store a vector which is normal to (that is, perpendicular to) the surfrace and every vertex. This vector should be normal to the true underlying model we wish to represent, which may not be exactly normal to the low dimensional polygonal version we are storing. This allows the GPU to interpolate the original normals and better recover the true curvature of the original model.

26a ⟨Load Cube Normals 26a⟩≡ (25)

```
float normals[] = { //Normals for 36 vertices
 0.f,0.f,-1.f,  0.f,0.f,-1.f,  0.f,0.f,-1.f,  0.f,0.f,-1.f,  //1-4
 0.f,0.f,-1.f,  0.f,0.f,-1.f,  0.f,0.f,1.f,   0.f,0.f,1.f,   //5-8
 0.f,0.f,1.f,   0.f,0.f,1.f,   0.f,0.f,1.f,   0.f,0.f,1.f,   //9-12
-1.f,0.f,0.f,  -1.f,0.f,0.f,  -1.f,0.f,0.f,  -1.f,0.f,0.f,   //13-16
-1.f,0.f,0.f,  -1.f,0.f,0.f,   1.f,0.f,0.f,   1.f,0.f,0.f,   //17-20
 1.f,0.f,0.f,   1.f,0.f,0.f,   1.f,0.f,0.f,   1.f,0.f,0.f,   //21-24
 0.f,-1.f,0.f,  0.f,-1.f,0.f,  0.f,-1.f,0.f,  0.f,-1.f,0.f,  //25-28
 0.f,-1.f,0.f,  0.f,-1.f,0.f,  0.f,1.f,0.f,   0.f,1.f,0.f,   //29-32
 0.f,1.f,0.f,   0.f,1.f,0.f,   0.f,1.f,0.f,   0.f,1.f,0.f,   //33-36
};
```

We then need to create two VBOs. One for the normals, and one for the rest of the model information.

26b ⟨Create and Bind 2 VBOs 26b⟩≡ (25)

```
GLuint vbo[2];
glGenBuffers(2, vbo);  //Create 1 buffer called vbo
```

The VAO will now contain two sets of mappings. One set of mappings connects data in the first VBO to the position and color inputs of our shader.

27a ⟨*Set-up 1st VBO mappings* 27a⟩≡ (25)

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]); //Set the first vbo as the active  buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), 0);
                //Attribute, vals/attrib., type, isNormalized, stride, offset
glEnableVertexAttribArray(posAttrib);

GLint colAttrib = glGetAttribLocation(shaderProgram, "inColor");
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
                      8*sizeof(float), (void*)(3*sizeof(float)));
glEnableVertexAttribArray(colAttrib);
```

An additional mapping connects data in the second VBO to the input in our shader that reads the triangle's normal.

27b ⟨*Set-up 2nd VBO mappings* 27b⟩≡ (25)

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(normals), normals, GL_STATIC_DRAW); //upload normals
GLint normAttrib = glGetAttribLocation(shaderProgram, "inNormal");
glVertexAttribPointer(normAttrib, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(normAttrib);
```

Notice, no stride or offset is needed for this attribute mapping as the normals in this VBO are stored sequentially, and not interleaved with other data as they would be if we put everything in one VBO.

We can now update our shaders to implement a model to compute lighting for each pixel. There are many options for what is computed per-vertex (in the vertex shader) and what is computed per pixel (in the fragment shader). We will use the vertex shader to transform all of 3D vertices to positions relative to the camera which is stored in `gl_Position`. The x and y positions store what pixel on the screen the vertex covers, the z position stores the depth of the vertex relative to the camera. We will also use the vertex shader to transform the position and directions of all the lights in the scene to be stored in positions relative to the camera.

27c ⟨*Phong 3D Vertex Shader* 27c⟩≡ (25)

```
const GLchar* vertexSource =
"#version 150 core\n"
⟨Declare Phong 3D Shader Inputs 28a⟩
⟨Declare Phong 3D Shader Uniforms 28b⟩
⟨Declare Phong 3D Shader Outputs 28c⟩
⟨Phong 3D Apply Model and Viewing Transformations 28d⟩
```

The vertex shader will take as input a position, color, and normal for each vertex in the model.

28a ⟨*Declare Phong 3D Shader Inputs* 28a⟩≡ (27c)
```
"in vec3 position;"
"in vec3 inColor;"
"in vec3 inNormal;"
```

We only need to load one `model`, `view`, and `proj` matrix for the entire model so we declare them as uniform values. For now, we will only use a single light from a fixed direction so we can set as once at the time the shader is compiled using the keyword const.

28b ⟨*Declare Phong 3D Shader Uniforms* 28b⟩≡ (27c 32)
```
"const vec3 inlightDir = normalize(vec3(1,0,0));"
"uniform mat4 model;"
"uniform mat4 view;"
"uniform mat4 proj;"
```

We need to output to the fragment shader any information that is needed to compute the lighting model.

28c ⟨*Declare Phong 3D Shader Outputs* 28c⟩≡ (27c 32)
```
"out vec3 Color;"
"out vec3 normal;"
"out vec3 pos;"
"out vec3 eyePos;"
"out vec3 lightDir;"
```

The actual lighting will take place in the fragment shader on a per-pixel basis. The vertex shader is responsable for transforming the object from the world coordinates to coordinates that are relative to the camera. The `model` and `view` matrices must both be applied to each vertex to find its new position relative to the camera. We must also apply the `model` and `view` transformations to an object's normal vectors in order to rotate the normals along with the object. The light is assumed to be placed independent of the model so we don't need to apply the `model` transformation to it, but we still need to multiply by the `view` transformation to account for camera motion. Finally we can compute 2D position of the vertex for `gl_Position` by applying the `proj` transformation.

28d ⟨*Phong 3D Apply Model and Viewing Transformations* 28d⟩≡ (27c 32)
```
"void main() {"
"   Color = inColor;"
"   vec4 pos4 = view * model * vec4(position,1.0);"
"   pos = pos4.xyz/pos4.w;"  //Homogeneous coordinate divide
"   vec4 norm4 = transpose(inverse(view*model)) * vec4(inNormal,0.0);"
"   normal = norm4.xyz;"
"   lightDir = (view * vec4(inlightDir,0)).xyz;"  //Transform light into to view space
"   gl_Position = proj * pos4;"
"}";
```

Tip **Transforming Normals** A 4x4 matrix in homogeneous coordinates can be used to transform vectors as well as points. We can exploit this to transform our normals by the same `model` and `view` matrices that we used to transform a model's vertices. However, when an object is sheared we don't want to shear the normal, but rather need to keep in perpendicular to the sheared surfrace. Multiplying the normal by the inverse transpose of the transformation applied to the model will keep the normal perpendicular. In practice, many game engines approximate this step by simply ignoring the inverse transpose step, and instead renormalizing the resulting vector in case it was distorted in length: `vec4 norm4 = normalize(view*model*vec4(inNormal,0.0));` This avoids a potentially slow call to matrix inverse, and is mathematically sound as long as only translation, rotations, and uniform scales are applied to the model.

The fragment shader will be responsible for computing the actual final color. Here, we use the Phong lighting model which compiles a base ambient color, plus a diffuse term (dependent on the angle of the light), and a specular highlight term (dependent on both the angle of the light and the camera). Here, we will follow the popular Phong Lighting Model. The final color of a given pixel will be determined by the combination of three factors. First, an *ambient* component which simply provides a small amount of constant lighting to every object. Second, a *diffuse* component, where the brightness of the pixel depends on the cosine of the angle between the vector to the light and the object's normal. If the light direction and object normal vectors are normalized, this value can be computed quickly as the dot product between the two vectors. Third, a *specular* component, were brightness depends on how close the viewing direction is to the mirror reflection of the light along the objects normal. The Phong model, approximates this function as the dot product of the reflected light direction with the camera direction, raised to a power. This exponent is set by the artist, and controls how large the specular highlight is. The ambient, diffuse, and specular terms are summed together for each color channel (red, green, and blue) separately.

29 ⟨*Phong 3D Fragment Shader* 29⟩≡ (25 31a)

```
const GLchar* fragmentSource =
  "#version 150 core\n"
  "in vec3 Color;"
  "in vec3 normal;"
  "in vec3 pos;"
  "in vec3 eyePos;"
  "in vec3 lightDir;"
  "out vec4 outColor;"
  "const float ambient = .3;"
  "void main() {"
  "   vec3 N = normalize(normal);" //Re-normalized the interpolated normals
  "   vec3 diffuseC = Color*max(dot(lightDir,N),0.0);"
  "   vec3 ambC = Color*ambient;"
  "   vec3 reflectDir = reflect(-lightDir,N);"
  "   vec3 viewDir = normalize(-pos);"  //We know the eye is at 0,0
  "   float spec = max(dot(reflectDir,viewDir),0.0);"
  "   if (dot(lightDir,N) <= 0.0) spec = 0;"
  "   vec3 specC = vec3(.8,.8,.8)*pow(spec,4);"
  "   outColor = vec4(ambC+diffuseC+specC, 1.0);"
```

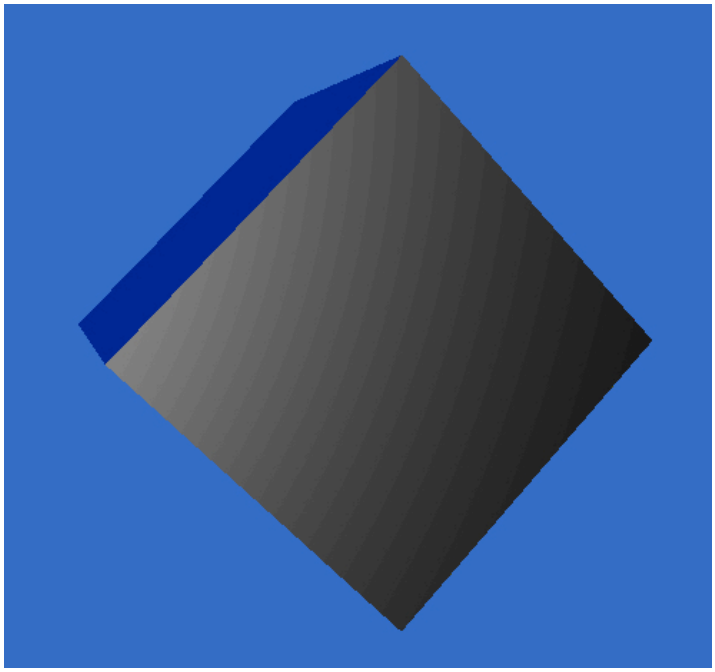

Figure 1.3: A spinning lit cube rendered by *CubeLit.cpp*

```
"}";
```

Definition 1.5 (Physically-Based Lighting) Computing how a to color a pixel depends on several factors including how close the part of the object seen by the pixel is to the light, the angle between that portion of the object and the light, the angle to the viewer, and the type of material the object is made of. In this chapter we focus on the Phong lighting model, which is a popular approach that is very fast to compute, but not particularly accurate to the actual physics of lighting. While these models are slower to compute than Phong, they can capture a much wider range of materials with higher quality than the simple Phong model. As computers have gotten faster, there has been a trend to include more physically-based lighting models in both offline graphics (e.g., movies) and real-time graphics (e.g., games).

30 ⟨*Cleanup with 2 VBOs* 30⟩≡ (25)

```
glDeleteProgram(shaderProgram);
glDeleteShader(fragmentShader);
glDeleteShader(vertexShader);
glDeleteBuffers(2, vbo);
glDeleteVertexArrays(1, &vao);
SDL_GL_DeleteContext(context);
SDL_Quit();
```

1.4 Loading Models

In practice, we would not normally hard code the vertex positions and other model data into the code for the graphics program as was done in previous sections. Rather, we would like to load all the data relating to the model such as the positions, normals, and texture coordinates from a file. Here we'll extend the above program to render a model loaded from a file. As before, we will start with the same basic structure used above:

31a ⟨*ModelLoad.cpp* 31a⟩≡

```
⟨GLAD, SDL, and OpenGL Includes 9a⟩
⟨File Reading Includes 31b⟩
⟨Globals 9b⟩
⟨Phong 3D Vertex Shader with Const Color 32⟩
⟨Phong 3D Fragment Shader 29⟩

int main(int argc, char *argv[]) {
  ⟨Initialize SDL 11⟩
  ⟨Initialize OpenGL through GLAD 12a⟩

  ⟨Load Model from File 33a⟩
  ⟨Load and Compile Shaders 13b⟩

  ⟨Create and Bind VAO 14b⟩
  ⟨Create Model VBO with Data from File 33b⟩
  ⟨Set-up Model VBO mappings 34a⟩

  glEnable(GL_DEPTH_TEST);

  SDL_Event windowEvent;
  bool quit = false;
  while (!quit){
    ⟨Process Keyboard and Mouse Events 15b⟩
    ⟨Draw Model Scene 34b⟩
    SDL_GL_SwapWindow(window); //Double buffering
  }
  ⟨Cleanup 17⟩
  return 0;
}
```

We need to include <fstream> to support reading files.

31b ⟨*File Reading Includes* 31b⟩≡ (31a)

```
#include <fstream>
using std::ifstream;
```

Often, 3D models do not store any color with model but rather expect all of the color information will be read from a texture (based on U,V coordinates that map a given part of the model to a specific part of a texture). For simplicity, we'll ignore any textures and texture coordinates and instead render our models with a uniform green color. To do so we need to update our vertex shader to use this fixed color.

32 ⟨*Phong 3D Vertex Shader with Const Color* 32⟩≡ (31a)

```
const GLchar* vertexSource =
"#version 150 core\n"
"in vec3 position;"
"const vec3 inColor = vec3(0.f,0.7f,0.f);"
"in vec3 inNormal;"
⟨Declare Phong 3D Shader Uniforms 28b⟩
⟨Declare Phong 3D Shader Outputs 28c⟩
⟨Phong 3D Apply Model and Viewing Transformations 28d⟩
```

Again for simplicity we will consider a very basic model format. Here, we assume the model is an ASCII text file with one number per line. The first line contains the number of lines remaining in the file. Then for each vertex, 8 lines are written sequentially containing the vertex's X, Y, and Z position; the U, V texture coordinates; and the X, Y, and Z component of the objects normal sampled at the vertex. An example object file might look as follows:

```
23808    Number of lines (8 times number of vertices)
0.1779   Vertex 1 X Position
0.1238   Vertex 1 Y Position
0.0      Vertex 1 Z Position
0.5      Vertex 1 U Texture Coordinate
1.0      Vertex 1 V Texture Coordinate
-0.967   Vertex 1 Normal Direction X Value
-0.256   Vertex 1 Normal Direction Y Value
0.0      Vertex 1 Normal Direction Z Value
0.1616   Vertex 2 X Position
⋮
```

This layout is exactly how we would like to store the data in our VBO, so we can directly read the model into an array. Note, we can determine the number of vertices in our model by dividing the number contained in the first line of the model file by eight.

33a ⟨*Load Model from File* 33a⟩≡ (31a)

```
ifstream modelFile;
modelFile.open("models/teapot.txt");
int numLines = 0;
modelFile >> numLines;
float* modelData = new float[numLines];
for (int i = 0; i < numLines; i++){
        modelFile >> modelData[i];
}
printf("Mode line count: %d\n",numLines);
float numVerts = numLines/8;
```

Once the data is read in, we can load it directly into a VBO.

33b ⟨*Create Model VBO with Data from File* 33b⟩≡ (31a)

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, numLines*sizeof(float), modelData, GL_STATIC_DRAW);
```

Because this model does not have any color information, the mappings of the VBO to the shader that we store in the VAO must be slightly different than in the cube example. Here, the first 3 elements are still position data, but the normals now start at the fifth position in the VBO.

34a ⟨*Set-up Model VBO mappings* 34a⟩≡ (31a)

```
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), 0);
                          //Attribute, vals/attrib., type, isNormalized, stride, offset
glEnableVertexAttribArray(posAttrib);


GLint normAttrib = glGetAttribLocation(shaderProgram, "inNormal");
glVertexAttribPointer(normAttrib, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), (void*)(5*sizeof
glEnableVertexAttribArray(normAttrib);
```

We are now ready to draw the model. As with the cube we need to clear the screen and the depth buffer, set up the `model`, `view`, and `proj` transformation matrices, bind the VAO that stores the mappings between the VBO and the shader inputs, and finally call `glDrawArrays` to actually draw the content stored in the VBO. Note that the last parameter of the call to `glDrawArrays` is the number of vertices in the model, a value we computed when loading the model.

34b ⟨*Draw Model Scene* 34b⟩≡ (31a)

```
// Clear the screen to default color
glClearColor(.2f, 0.4f, 0.8f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

⟨Set-up transformation matrices 23a⟩

glUseProgram(shaderProgram);
glBindVertexArray(vao);  //Bind the VAO for the shaders we are using
glDrawArrays(GL_TRIANGLES, 0, numVerts); //Number of vertices
```

The final results is a spinning teapot shown in Figure 1.4. Changing the model file from `teapot.txt` to some other model will change what is rendered.

Figure 1.4: A spinning Teapot model rendered by *ModelLoad.cpp*

Quick Check

Quick Check 1.1 When calling `SDL_CreateWindow()` we passed 100 for the second and third parameters. Try changing these numbers, what is the effect?

Quick Check 1.2 In *Cube3D.cpp* we call `glEnable(GL_DEPTH_TEST)`. Remove this call, how do you explain the resulting image?

Quick Check 1.3 Currently, our examples exits when escape is pressed. Update the code to also exit when the "Q" key is pressed.

Quick Check 1.4 Create a new model file, by hand, that contains a single large triangle. Load this model instead of `teapot.txt`.

Programming Exercises

Computer exercise 1.1 Add a key-board based camera controller to one of the 3D examples. One simple approach is to add new variables which get passed into the call of `glm::lookAt()`. These variables should then be modified based on key press events.

Computer exercise 1.2 Update *ModelLoad.cpp* to rotate and translate the model based on key presses.

Computer exercise 1.3 Replace the phong specularity model in *ModelLoad.cpp* with Blinn-Phong. What happens to the size of the highlight?

Computer exercise 1.4 The vertex shader in *ModelLoad.cpp* hard codes the color to green. Update this code to instead draw the model loaded with a random color. Each time the user presses the 'c' key change the model to a new random color.

Computer exercise 1.5 The vertex shader in *CubeLit.cpp* and *ModelLoad.cpp* hard codes the light to be shining in the direction of (1,0,0). Change this value from a const vec3 to a uniform, then dynamically update the direction to rotate around the model.

Computer exercise 1.6 The shaders used in *CubeLit.cpp* and *ModelLoad.cpp* support a directional light model. This means the strength of the lighting depends only on the direction of the light, not it's distance. Update the code to support a point light. This change involves two steps. One, the direction that was statically stored in `inlightDir` should now be dynamically computed to point from the `pos` of the part of the model that is being rendered to current position of the light. Two, the strength of the light should be divided by the distance form the light, squared (i.e., further way lights should have less impact).

Computer exercise 1.7 The code shown in *CubeLit.cpp* and *ModelLoad.cpp* computes the full Phong lighting model for each pixel. This leads to a nice image, but can be expensive to compute. Re-write the shaders to move the entire lighting computation to the vertex shader. This approach is known as Gouraud shading, and can be much faster as the lighting is only computed once per vertex. By default, the graphics card will quickly interpolate the computer color between vertices. Is the rendering quality noticeably different? Which is faster?

Computer exercise 1.8 The Wavefront OBJ file format is a popular way to store 3D content, that is widely supported by many tools. The OBJ format is only slightly more complex than the format used here. The main change is that it first lists all vertex positions, then an UV texture coordinates, then all vertex normals, and finally builds the triangles out of vertices by choosing one of the positions, texture coordinates, and vertex normals listed at the start of the file. You can read a detailed description of the format at `https://en.wikipedia.org/wiki/Wavefront_.obj_file`. Write a program that converts the vertex data stored in OBJ files to the format used here.

Tip 1: OBJ files allow arbitrary convex polygons (not just triangles). But you can safely assume these vertices will be sorted (typically counter-clockwise), allowing you to easily convert a polygon of n verticies into $n-2$ triangles.
Tip 2: This type of file conversion is typically best done in a scripting language such as python.

By adding OBJ suppor 3D projects can include a wide variety of models readily available online, greatly enhancing the quality of the results, and the fun of working on the project.

3D Math

Problem 1.1 Consider multiplying the 3D matrix M_{proj} by the point P as defined below.

$$M_{proj} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

a. Compute the resulting 4D homogeneous coordinate of $M_{proj} * P$.
b. What 3D point does this homogeneous coordinate correspond to?
c. Show that X/Z and Y/Z is the correct perspective transform for an object at point (X,Y) transformed onto the plane $Z = 1$ from a camera looking down the Z axis.

Problem 1.2 Consider multiplying the 3D matrix M_{trans} by the point P or the vector V as defined below.

$$M_{transj} = \begin{bmatrix} 1 & 0 & 0 & s \\ 0 & 1 & 0 & t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad V = \begin{bmatrix} X \\ Y \\ Z \\ 0 \end{bmatrix}$$

a. Show that multiplying $M_{trans} * P$ results in translating P by the amount (s, t).
b. Show that multiplying $M_{trans} * V$ does not result in any translation.
c. Find a matrix M_{scale} such that $M_{scale} * P$ scales P by 2X and $M_{scale} * V$ also scales V by 2X.
d. Find a matrix M_{rot} such that $M_{rot} * P$ rotates P by 90° clockwise and $M_{rot} * V$ also rotates V by 90° clockwise.
e. Mathematically speaking, why would we want a representation for points which can be translated, rotated, and scaled, but a representation for vectors which can be scaled and rotated, but not translated?