OpenGL tutorial

Chapter 1: The main application window

The Windows window.

To begin with we will build a simple windows application, connect it with OpenGL and get into the very basics.

The program loop.

In windows we substitute the *main* function of a typical C/C++ program with the windows specific *WinMain* function.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPTSTR lpCmdLine, int nCmdShow)
```

The parameters of the function are:

- *hInstance* is the handle of the current instance of the program. It is a value assigned by the operating system to identify the program in memory.
- *hPrevInstance* is always NULL. It was used in 16-bit windows.
- *lpCmdLine* the command line of the program
- *nCmdShow* is a flag indicating how the main window will be upon startup.

Here is a sample implementation:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPTSTR lpCmdLine, int nCmdShow)
{
    initialize_application(hInstance, nCmdShow);

    // load keyboard shortcuts
    HACCEL hAccelTable = LoadAccelerators(hInstance,
                  MAKEINTRESOURCE(IDC_BASICWINDOW));

    // message (msg) variable encodes information about user interaction
    // or other system wide event
    MSG msg;
    // Main message loop:
    // waits for user interaction, decodes it and passes the message (msg)
    // to WndProc below
    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            // let windows process the message
            TranslateMessage(&msg);
            // ask windows to pass the message to WndProc
            DispatchMessage(&msg);
        }
    }

    return (int)msg.wParam;
}
```

We will just focus on the *DispatchMessage* function. It simply redirects the user triggered event to the WndProc function which will take the appropriate action based on the event:

```
// the callback windows calls when needed via DispatchMessage
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    // menu command
    case WM_COMMAND:
    {
        int wmId = LOWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        // exit menu command
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
    }
    break;
    // draw window contents
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);
        // TODO: Add any drawing code that uses hdc here...
        EndPaint(hWnd, &ps);
    }
    break;
    // destroy the window, here we can do some cleanup
    case WM_DESTROY:
        // ok to exit
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Adding OpenGL

The next step is to create a window that connects with OpenGL to give us access to the advanced drawing it provides.

Windows drawing basics

Whenever the system wants to update the contents of our window it sends a *WM_PAINT* message to our *WndProc* function. Our program will not use this because we do things using OpenGL. So we will just write some basic code that does nothing special:

```
case WM_PAINT:
{
    // details about the command are in this struct
    PAINTSTRUCT ps;
    // HDC in our connection to the video driver
    HDC hdc = BeginPaint(hWnd, &ps);
    // EndPaint tells the system we are done
    EndPaint(hWnd, &ps);
}
break;
```

Initializing OpenGL window

OpenGL programs need a permanent connection with the video driver to achieve smooth drawing. Furthermore we will use a technique called *double buffering* for smoother drawing and animation. For this purpose we create two virtual screens. One is presented to the user while we draw in the other. When we are ready we simply swap the two. We need to predefine some attributes of these virtual screens upon the creation of our application window.

All the necessary initializations are handled in the *initialize_window* function. Key to the quality of the image we see is the *PIXELFORMATDESCRIPTOR* structure. This structures passes information about the color depth we will use, the double buffering, the need for a stencil buffer and other techniques OpenGL provides and we will use in our program.

Terminating OpenGL window

Creating an OpenGL window reserves many system resources. It is our responsibility to release them when we no longer need them. So we call the *destroy_GL_window* function after the main loop and before our program exits.

Modifying the program loop

Now we will modify the main loop of our program to accommodate for OpenGL drawing.

First we will stop using *GetMessage* and use *PeekMessage* instead. This function checks the message queue and if a message is pending it behaves like *GetMessage*. If no message is pending it returns allowing us to perform any tasks we want. Here is how our program loop looks like:

```
bool bLooping = true;
MSG msg;
while (bLooping)
{
    // check for windows messages and process them
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) != 0)
    {
        // destroy window was invoked (escape or Alt-F4)
        if (msg.message == WM_QUIT)
            bLooping = false;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else  // no messages, just loop for next frame.
    {
        if (g_window.isMinimized)    // if window is minimized
        {
            // yield back to system, do not waste processing power
            WaitMessage();
        }
        else
        {
            // here we do our own processing and drawing
        }
    }
}
```

When our program window is minimized we yield control to the system. This allows other processes to work without a problem. Otherwise we perform any operation we want.

Drawing in OpenGL

When we created the main window of our program, we initialized the *PIXELFORMATDESCRIPTOR* structure with *PFD_DOUBLEBUFFER*. This means that we have two drawing surfaces for smooth drawing.

From our main loop, first we call our function that does the dirty work of drawing everything, and then we call the *SwapBuffers* function to swap the buffers and display the contents of the hidden one.

```
// render scene
frame_render();
// Swap Buffers (Double Buffering)
SwapBuffers(g_window.hDC);
```

Let us take a look at the *frame_render* function and take a look at our first scene.

```
void frame_render()
{
    // first step: set up our camera
    // black background
    glClearColor(0.0f, .0f, 0.0f, 1.f);
    // clear screen and depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // set the viewport to the whole window
    glViewport(0, 0, (GLsizei)(g_window.vwidth), (GLsizei)(g_window.vheight));
    // select the projection matrix
    glMatrixMode(GL_PROJECTION);
    // reset the projection matrix
    glLoadIdentity();
    // set the viewing angle to 45 degrees
    // set the aspect ratio of the window
    // and draw everything between 1 and 1000 units from the viewer
    gluPerspective(45.0f, (float)(g_window.vwidth) /
                     (float)(g_window.vheight), 1.0f, 1000.0f);

    // second step: the world we want to picture
    // select the modelview matrix
    glMatrixMode(GL_MODELVIEW);
    // reset the modelview matrix
    glLoadIdentity();

    // move 6.0 units apart
    glTranslatef(0.f, 0.0f, -6.0f);

    // drawing using triangles
    glBegin(GL_TRIANGLES);

    // set the color to red
    glColor3f(1.0f, 0.0f, 0.0f);
    // top vertex
    glVertex3f(0.0f, 1.0f, 0.0f);

    // set the color to green
    glColor3f(0.0f, 1.0f, 0.0f);
    // bottom left vertex
    glVertex3f(-1.0f, -1.0f, 0.0f);

    // set the color to blue
    glColor3f(0.0f, 0.0f, 1.0f);
    // bottom right vertex
    glVertex3f(1.0f, -1.0f, 0.0f);

    // finished drawing the triangle
    glEnd();
}
```

This function is made up of two distinct parts. The camera that takes a picture, and the world we want to take a picture of.

The sensor of our camera is our window. In the beginning we clear the camera sensor. OpenGL will automatically remove hidden surfaces and objects. Then we define the portion of the sensor we want to use. Our camera is more capable than any ordinary camera. The next step is to set up our lens. For our sample we have a lens with a viewing angle of $45°$, set the aspect ratio of the sensor and tell it to capture anything from one unit to one thousand units from our location. Note here that we are specifying the *nearest* and *furthest* distances, so they are positive. The default position of the camera is at the position $v(0,0,0)$, with the up direction being at the $z-axis$, and looking towards the negative $z-axis$.

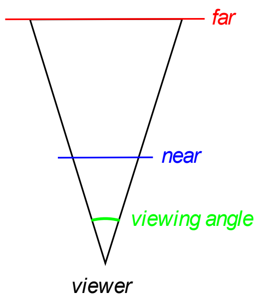

Figure 1: The view perspective

When we are done with the camera, we start describing our objects. OpenGL automatically calculates the transformations and draws everything, taking care of hidden surface removal.

In this sample we draw a simple triangle six units towards the negative of the $z-axis$.

OpenGL and the transformation matrix

OpenGL uses matrices to perform transformations and projections. Here we are going to take a closer look at how it uses them to perform all is calculations.

In OpenGL there is one active matrix for the camera settings, identified as *GL_PROJECTION* , one for the model world identified as *GL_MODELVIEW*, and some more that we will see in time. We select them calling *glMatrixMode*, with the appropriate matrix ID. These matrices are initialized to the identity matrix when we call *glLoadIdentity*. From that point on every change is accumulated, so every time we are left with the latest version of the matrix.

In our sample we start with the *camera* matrix, or *projection* matrix as it is actually called. We set it to identity matrix and then we call utility function *gluPerspective*, to set it appropriately in order to behave like the camera we have in mind.

A similar operation is performed for the *world*, or *modelview* matrix. First, we set it to identity matrix and then we modify it to move all the objects 6 units apart from the viewer with the *glTranslatef* function. This tells OpenGL where the objects are and then we draw them using coordinates local to them. This allows us to draw similar objects, say boxes, in many locations in a scene, saving us a lot of memory and processing time.

Keeping track of time

If we want to do animations in our application we will need a precision timer. This way we can tell how much time has elapsed since the last frame we drew and make the necessary changes in the next, thus presenting animations that look natural.

For this reason we created a high-resolution clock we can use to get the time. In this sample it is used to calculate the frames per second:

```
// frame counting mechanism
// the higher the frame rate the faster the system
// if frame rate drops below 30 we are in deep trouble
// we should either optimize the program or buy a new computer
static int m_nFrames = 0;                              // frame Counter
static float tot = 0;                                  // time counter
tot += fElapsed;                                       // increment counters
m_nFrames++;
if (tot >= 1.f)                                        // one second reached
{
    char txt[200];
    sprintf_s(txt, "Altair, fps:%d", m_nFrames);
    SetWindowText(g_window.hWnd, txt);
    tot = 0;                                           // reset counters
    m_nFrames = 0;
}
```

Chapter 2: The pipeline

Then was then and now is now. The OpenGL code we saw in chapter 1 is quite old fashioned. It was very good when graphics cards did not have enough memory to store our geometry and we had to pass the objects one by one.

Modern graphics hardware is very powerful and massively parallel. This means that they can perform too many operations per unit of time. All this results in spectacular images in real time. Our programs and animations can use all this power.

So, instead of passing the objects one by one we can load all our geometry data in the video memory and let the GPU do all the hard work. We can take advantage of the hardware and make it perform complex tasks. This is done with the use of an OpenGL specific programming language, the *GL Shading Language* or *GLSL*.

The steps we need to take are clearly defined by the OpenGL standard and they are called *the pipeline*.

Walk the pipeline.

There are certain steps in the OpenGL pipeline. We have access only to a number of them. Our main tool is the *GL Shading Language*, or *GLSL* for short. Some are fixed processes caried by the system. Here is a brief description of the OpenGL pipeline:

- *Vertex Specification*: The first thing we have to do when we program in OpenGL, is to collect our data. An object on the screen is made up of certain attributes. These are the vertices, a.k.a. the points, the edges, and the faces they define, the colors and the textures, or any other attributes we may need.
- *Vertex Shader*: The first actual calculation is the Vertex Shader. The information we collected in the previous step passes through this. The objective of this step is to calculate the final position of every vertex in the scene. This is a good place to put our 3D calculations, since the GPU will execute them.
- *Tessellation*: in this step the primitives are divided into a smoother mesh of triangles. This step is optional.
- *Geometry Shader*: In this step we can further manipulate our primitives. We can break them into smaller ones, organize them differently i.e., convert points to triangles, or even remove some of them. This step is also optional.
- *Vertex Post Processing*: In this stage, OpenGL decides what is in our field of view and what is not. This process is called *clipping*. There is no way we can intervene in this stage.
- *Primitive Assembly*: This stage collects the vertex data into an ordered sequence of simple primitives. It is also an internal stage we cannot modify.
- *Rasterization*: This stage creates fragments. It is a very important step of the pipeline.
- *Fragment Shader*: This stage calculates the color of each fragment calculated by the previous step. We can write code in GLSL and manipulate what the user sees. It is optional to do so, but taking in mind the advantages it gives us, it is clear that this is a very good point to add our code.
- *Per-Sample Operations*: This is the final step of the pipeline. In this stage the engine performs some final tests like *Stencil Test, Depth Test* or *Scissor Test*.

One step at a time

In the previous chapter we saw a sample program that was throwing some drawing commands to OpenGL. It was a very straight forward and simple solution. On the other hand, looking at the description of the pipeline is quite intimidating. It seems to be a lot of work that has to be done in order to draw a simple triangle on the screen.

using the pipeline though we end up with a much faster rendering code. All the data are stored in video memory and the calculations are performed by the GPU which was actually designed for this purpose.

Structure of the program

In the ***pipeline*** sample, we break the code into files so we can focus only on the things we need to.

Here we introduce a small math module to handle the math we need for our 3D calculations. We could use *glm*, a math module available on internet but it is too big for our needs. Then we also include *OpenGL Extension Wrangler Library*, which enables access to all the OpenGL functions because the development libraries provided do not.

Organizing the data

The first step in the pipeline is the *Vertex Specification*. In this step we gather and organize the data that represent our meshes and models, and we load our shader programs.

This is done in our *init_game* function. There the first thing we do is load the shaders into the GPU memory. These little programs will guide the GPU when we render the triangle.

Then we allocate space in the memory of our graphics card and store the triangle geometry and color attributes to be used for drawing. We are starting with the geometry data. The shaders will be examined a little later. For the time being, we compile and load them to get things going.

The first thing we have to do is allocate a *Vertex Array* in which we will store vertex data.

```
// create the main storage
glGenVertexArrays(1, &vertex_array);
// bind and use it
glBindVertexArray(vertex_array);
```

glGenVertexArrays allocates the memory and stores its identifier in the *vertex_array* variable. OpenGL depends heavily on its state. After we allocate the array, we bind it using *glBindVertexArray*. From that point on subsequent calls are appended to this array. Any other allocations are stored within this array. So, our next step which allocates memory for the vertices of our triangle will allocate the buffer inside the *vertex_array.*

```
// create a buffer for the vertices
glGenBuffers(1, &vertex_buffer);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
glEnableVertexAttribArray(0);
```

This allocated buffer is inside the vertex array and after we bind it *glBufferData* copies the vertex coordinates inside the *vertex_buffer*.

OpenGL buffers can contain more than just data. They can also store commands like *glVertexAttribPointer* which instructs the library where to ‘find’ the data. Storing these commands within the vertex buffers speeds the drawing process dramatically. You can visualize the buffers as programs containing both data and code that are stored in superfast GPU memory and run int GPU as well.

Similarly to the vertex data we allocate buffer for the colors. We can add buffers containing any information we like. The most common attributes are colors, textures coordinates and normal vectors.

Vertex shader

As we know from everyday experience, what we see is subject to our location, the direction of our sight, the objects' location, and orientation. On the other hand every object can have its own coordinate system and it have its geometry expressed based on that system.

This is the way we define objects in 3D handling software, be it Computer Aided Design or Game software, or any other kind of software of which we can think.

It is the responsibility of the *Vertex Shader* to translate the local coordinates of the vertices to real world coordinates and then to view coordinates in order to draw the object.

The shader can take as input the transformation matrices for the view and for the objects along with the coordinates and other attributes of the vertices. The transformation matrices and the coordinates are used to calculate the final location of the vertices on the screen. Here is a sample shader:

```
#version 410 core
// input to the vertex shader
// location where the vertex coordinates are stored
layout(location = 0) in vec3 aPos;
// location where the vertex color is stored
layout(location = 1) in vec3 aCol;
// 4x4 matrix for the model
uniform mat4 model;
// 4x4 matrix with the view parameters
uniform mat4 view;

// output of the vertex shader
// the vertex color is passed to the fragment shader
out vec4 vs_color;

// 'main' the entry point to the shader
void main(){
    // calculate the final position for the vertex
    gl_Position = view * model * vec4(aPos, 1.0);
    // the vertex color
    vs_color = vec4(aCol, 1.0);
}
```

The first line defines the minimum OpenGL version requirement, which is 4.1 in this case.

Then we describe how our *vertex array* is organized. The first buffer contains the vertices in a three-dimensional vectors, and the second contains the colors again as three-dimensional vectors.

The next two variables are the transformation matrices to use. One for the model positioning and orientation, and one for the 'camera' transformation.

Finally we define our output variable *vs_color* that we are going to use so we can pass the user color to the *fragment shader*. More on this shader when its time comes. Now let us focus on what *vertex shader* does.

OpenGL has some built in variables we can set. One such variable is *gl_Position*. In this variable we set the final position of the vertex. Here we multiply the view matrix with model matrix and the vertex position. The matrices must be organized column major because this is the convention in OpenGL.

The final variable we set is *vs_color* which is used to read the color from the vertex attributes and pass it to the next step.

In our program we introduced some new code to handle our requirements in mathematics. The code is in the 'math' files. the first thing we introduced is the 4x4 matrix. We need this for the transformations we are going to use in this sample.

```
mat4 view = perspective_matrix(pi / 4.f, (float)(g_window.vwidth) / (float)(g_window.vheight),
1.f, 1000.f);
mat4 model = translation_matrix(0,0,-6);

// use shader
glUseProgram(shaderID);
set_mat4(shaderID, "model", (float *)model);
set_mat4(shaderID, "view", (float *)view);
// draw the triangle (vertex coordinates are in the shader)
glBindVertexArray(vertex_array);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glDrawArrays(GL_TRIANGLES, 0, 3);
glBindVertexArray(0);
```

The first matrix represents the camera like the call to *gluPerspective* we saw in the previous example, and the second is the translation matrix which is equivalent to the *glTranslatef* call. We pass these matrices to the shader, before we start drawing, giving them the variable names we set earlier in the shader.

Tessellation

Tessellation is the process of breaking up a large area into smaller pieces. This is what we do when we apply small tiles on a large surface. In computer graphics we divide a large polygon, usually a triangle, into smaller ones.

This process is very useful when we want to apply higher detail to objects that are closer to the viewer, while those that are further away do not need high detailed drawing.

Tessellation follows the vertex shader, and it is done in three steps.

The first of the three tessellation phases is the *tessellation control shader*. This shader takes its input from the *vertex shader* and is primarily responsible for two things: the determination of the level of tessellation that will be sent to the tessellation engine, and the generation of data that will be sent to the tessellation evaluation shader that is run after tessellation has occurred.

Second is the *Tessellation Engine* that generates the new vertices. It is a fixed function engine, and we can only set its parameters in the *tessellation control shader*. it produces a number of output vertices representing the primitives it has generated. These are passed to the tessellation evaluation shader.

Third and final step of the process is the *tessellation evaluation shader*. The tessellation evaluation shader runs an invocation for each vertex produced by the tessellation engine. When the tessellation levels are high, the tessellation evaluation shader could run an extremely large number of times. For this reason, you should be careful with complex evaluation shaders and high tessellation levels.

First. we create a simpler *vertex shader*. The new one just reads the model coordinates and passes sets them in the OpenGL variable *gl_Position*. The view and model matrices will be applied in the final step of *tessellation evaluation*.

```
#version 410 core
layout(location = 0) in vec3 aPos;
void main(){
    gl_Position = vec4(aPos, 1.0);
}
```

Here are two simple shaders that we can use to tessellate a simple triangle:

```
// first is the control shader
#version 410 core
layout(vertices = 3) out;
void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 3.0;
        // number of tessellations on each of the outer edges
        gl_TessLevelOuter[0] = 3.0;
        gl_TessLevelOuter[1] = 3.0;
        gl_TessLevelOuter[2] = 3.0;
    }
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}

// second is the evaluation shader
#version 410 core
layout(triangles, equal_spacing, cw) in;
uniform mat4 model;
uniform mat4 view;
out vec4 es_color;
void main(void)
{
    vec4 pos = (gl_TessCoord.x * gl_in[0].gl_Position) +
        (gl_TessCoord.y * gl_in[1].gl_Position) +
        (gl_TessCoord.z * gl_in[2].gl_Position);
    es_color = vec4(gl_TessCoord,1);
    gl_Position = view*model*pos;
}
```

More on tessellations when we talk about level of detail in scenes. This is just an example of how it attaches in the pipeline.

The Fragment Shader

OpenGL has finished rasterizing the image. That means the image is ready pixel by pixel. It knows what each pixel on the screen represents in terms of objects depth and orientation. This is our last chance, based on knowing what we need to draw, to tell OpenGL what color to set for each pixel.

We created a *fragment shader* before talking about tessellation. There we set the pixel color. Now we are asked to set the pixel color for a tessellated object. For this reason we calculated the color in the *evaluation shader*. There we had access to the coordinates generated by the tessellation engine. We used that information to calculate a color and create a fancy result.

```
#version 420 core
out vec4 color;
in vec4 es_color;
void main(void)
{
    color = es_color;
}
```

Chapter 3: Shaders and Shader Language

In this chapter we will cover the *shaders* and the *shader language GLSL*. Shaders are fundamental in OpenGL. They are the tool to take advantage of the graphics hardware that is specifically designed and optimized to perform all the operations required for all the effects we need in our programs.

As we have seen in the previous chapter after we organize our data the control goes to OpenGL and the underlying hardware. Using shaders is the key to fast and eye-catching graphics that will make our programs stand out.

The language

The *language* we use for shaders is GLSL. Its syntax is based loosely on the C programming language. Here is a simple shader program:

```
#version 410 core
uniform mat4 view;
in vec4 color;
out vec4 vs_color;
void main() {
    vs_color = color;
    float f = 2.0f;
    vec4 v = vec4(1,2,3,4)
}
```

As we can see statements are very simple and easy to understand and write. We can create blocks of code by enclosing the statements in curly brackets. The syntax is very easy to understand if you have some experience with C or C++.

Since the language is so close to the C/C++ syntax we will just do a brief pass over its features.

The Preprocessor

Just like C/C++ GLSL has a preprocessor. It takes the input shader code and modifies it according to the command we issue. Here is the list of the preprocessor commands:

#version : this command must be in the first line of the code. It instructs the compiler to stick to a certain version of the language. This means that the compiler will throw an error if it encounters a feature not supported by the specified version.

#extension : used to enable or disable an OpenGL extension that is not yet part of the core implementation of OpenGL.

#pragma : allows compiler control. We can use it to enable or disable optimizations or debug mode compile.

#error, #define, #undef, #if, #ifdef, #ifndef, #else, #elif, #endif : are the same as in C/C++ and need no extra explanation.

```
#version 410
#extension ARB_explicit_attrib_location : enable
#pragma optimize(on)
#pragma optimize(off)
#pragma debug(on)
#pragma debug(off)
#error Some error occurred
#define MY_VALUE 2
#undef SOME_VALUE
#ifdef MY_VALUE
#ifndef SOME_VALUE
#if MY_VALUE == 3
#else
#elif MY_VALUE == 4
#endif
```

Data types

This is a topic that needs a thorough approach. GLSL introduces some data types we do not have in C/C++. These data types are required to handle the geometry of objects and the transformations that must be applied. Here is the list of data types in GLSL.

- *bool*: Boolean, only *true,* or *false*
- *int*: Integer value, *negative* or *positive* value
- *uint*: *Positive integer* value
- *float*: *floating point* number
- *sampler*: this type represents texture sampling, it can be one of these.
    - *sampler1D*
    - *sampler2D*
    - *sampler4D*
- *vectors*: built in vectors of 2, 3, or 4 elements, based on the type of the elements they can be.
    - *bvec2, bvec3, bvec4*: vectors of Booleans
    - *ivec2, ivec3, ivec4*: vectors of integers
    - *uvec2, uvec3, uvec4*: vectors of unsigned integers
    - *vec2, vec3, vec4*: vectors of single precision floats
    - *dvec2, dvec3, dvec4*: vectors of double precision floats
- *matrices*: matrices in GLSL are always made of floating-point numbers either of single or double precision. The later take the prefix ‘d’.
    - *square matrices*: these can be 2x2, 3x3 or 4x4 matrices.
        - *mat2, dmat2*: 2x2 matrix
        - *mat3, dmat3*: 3x3 matrix
        - *mat4, dmat4*: 4x4 matrix
    - *arbitrary matrices*: these matrices can be 2x3, 2x4, 3x2, 3x4, 4x2, 4x3. The first number refers to their columns and the second to their rows.
        - *mat2x3, mat2x4, mat3x2, mat3x4, mat4x2, mat4x3*
        - *dmat2x3, dmat2x4, dmat3x2, dmat3x4, dmat4x2, dmat4x3*

All the matrices in OpenGL are column major. That means that in a 4x4 matrix for example the first four elements are those of the first column. One more thing we must keep in mind is that when we talk about the dimensions of a matrix, the first number is the number of columns.

Arrays

Vectors and matrices are collections of numeric data organized in certain ways, based on the mathematical concepts behind them. If we want collections of data of arbitrary size we need arrays. Arrays in GLSL are declared like arrays in C++.

```
int indices[5];
vec3 vertices[] = {vec3(0,0,0), vec3(1,1,1), vec3(2,2,2)};
```

Accessing the elements in an array is similar as well.

```
vec3 pos = vertices[1];
indices[2] = 3;
```

Vectors

Vectors are a very important and flexible data type in GLSL. They can be used to access vertex positions, normal vectors, colors, and textures. Vectors can be found anywhere in the shader programs we create and can help us accomplish complex tasks.

First let us see the members of the vector type.

- {x, y, z, w}, used when accessing position data.
- {r, g, b, a}, used when dealing with colors.
- {s, t, p, q}, used for texture coordinates.

vec2 has only the first two members, *vec3* the first three, and *vec4* has them all. We can access them either using the index of their position in the vector, or using the corresponding name:

If v is a vector, $v[0]$ is equivalent to $v.x$ and $v.r$ and $v.s$. Be careful though because if v is a $vec2$, then $v.z$ is not defined and will yield an error.

Accessing the elements of a vector using their names is very flexible. Let us assume that v is a vector of type $vec4$. Here is what we get when accessing its members:

- *v.x* : yields a floating-point number.
- *v*.xy : yields a *vec2.*
- *v.yzx* : yields a vec3 initialized like this {v.y, v.z, v.x}
- *v.yxwz* : yields a reordered vec4.
- *v.xyxy* : yields a vec4 based on *x* and *x* and *y* elements.

It is obvious that there are numerous combinations we can have when accessing the elements of a vector.

Another thig worth mentioning is the initialization of vectors. The language accepts many combinations of vectors, and values:

```
// initialize a vector with floats
vec2 v = vec2(1, 2);
// initialize using another vector and float
vec3 u = vec3(v, 1);
// or using a float and a vector
vec3 u2 = vec3(1, v);
// use part of u and v
vec4 t = vec4(u.xy, v);
// use x of u for all of t elements
vec4 t = vec4(u.xxxx);
// use 1 for all of t elements
vec4 t2 = vec4(1);
// member access can be used to set values as well
t2.xy = t.zw;
```

We can use another vector to initialize one, or if the vector we use is short we can fill in with another vector or values. The only limit to the combinations we use is the actual problem we are trying to solve and the data we have available.

Matrices

We have cleared that matrices in OpenGL are column major, so $m[0]$ is a vector of the elements of the first column. Its length depends on the number of rows in the matrix.

```
vec2 v = vec2(1, 2);
vec3 u = vec3(v, 1);
mat4 m;
m[0] = vec4(v, 2, 3);
m[1][0] = 5;
m[2] = vec4(u.xy, v);
```

Operations between matrices and vectors

We said earlier that vectors can be used to represent vertex positions and matrices can hold transformations. In this section we will see the operations we can do with vectors and matrices to do all the calculations we need.

Here is a sample code snippet with the operations we can do:

```
mat3 t, r;  // translation, rotation
vec3 v, u;  // two vectors
float f;    //
// assume the variables are initialized

u = f * v;  // scaling vector v
// or
u = t * v;  // translate v
// or
u = r * v;  // rotate v
// or
// translate and then rotate v
u = r * t * v;
// setting the final location of vertex_pos
// using view and model matrices set by our program
gl_Position = view * model * vertex_pos;
```

Just a little reminder here. As we saw in part 2, although matrix multiplication is performed from left to right, the results propagate from right to left. So when we perform the operation $u = r * t * v$, we end up applying translation and the rotation.

Flow control

GLSL, like any other language, supports flow control of the code depending on conditions. Being modeled after the C programming language means that whatever we present here is similar to thing we already know from our experience with C++, which is based on C to ease transition.

In GLSL we have *if/else* and *switch* to direct code execution based on conditions.

```
in int a_variable;
out vec4 s_color;
void s_conditional()
{
    if (a_variable == 2)
    {
        gl_Position = vec4(1, 2, 3, 4);
        s_color = vec4(1, 1, 1, 1);
    }
    else
    {
        gl_Position = vec4(4, 3, 2, 1);
        s_color = vec4(1, 1, 1, 0.5);
    }

    // or
    switch (a_variable)
    {
    case 1:   // set color to white.
        s_color = vec4(1, 1, 1, 1);
        break;
    case 2:   // set color to red.
        s_color = vec4(1, 0, 0, 1);
        break;
    case 2:   // set color to green.
        s_color = vec4(0, 1, 0, 1);
        break;
    default:   // set color to blue.
        s_color = vec4(0, 0, 1, 1);
        break;
    }
}
```

Loops

GLSL supports the three basic loops we know. These are the *for, while,* and *do/while* loops. Whatever we said about these loops when we were introduced to them in C++ is valid for GLSL loops as well. They behave in exactly the same way. *for* and *while* loops perform the check in the beginning of the loop, while the *do/while* loop performs the check at the end. This means that the first two may never enter the code inside the loop, and the third will execute that cod at least once.

Their syntax is the same as in C/C++ we know already:

```
void loops()
{
    for (int i = 0; i < 10; i = i + 1)
    {
        // do something
    }
    int j = 0;
    while (j < 10)
    {
        // do something
        j = j + 1; // remember to increase counter!
    }
    int j = 10;
    do {
        // do something
        j = j - 1; // remember to decrease counter!
    } while (j > 0);
}
```

Structures

As we saw in C++ structures are actually custom data types. They give the opportunity to put together data describing one entity.

```
struct my_vertex {
    vec3 pos; // position
    vec2 tex; // texture coordinates
    vec3 normal; // normal vector
};
// here we have an array of structures
my_vertex mv[10];
mv[0].pos = vec3(1, 1, 1);
// initializing an array
my_vertex v = my_vertex(vec2(0, 0, 0), vec2(1, 1), vec3(1, 0, 0));
```

We see that GLSL structures are simple, and they do not defer much from C++ structures. Declaring, accessing, and initializing a structure is modeled after the simplest versions of their C/C++ counterparts.

Functions

Now is the time to see how we can organize our shader code into meaningful and reusable blocks. As in C/C++ functions can take arguments and return values.

Let us start with the entry point to our shader. The first function of our shader called by the system is as you may have guessed the *main* function. This function takes no arguments and returns nothing.

Our shaders are supposed to perform whatever calculations we need and finally set predefined OpenGL variables that will be used to create the scene. Here is a *main* function that sets the final position of a vertex:

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

As we do in any programming task that is reasonably complex and big, putting all the code in one function is not a good idea. The following example shows how we can use a function that takes a direction vector and returns a new direction.

```
vec3 redirect(vec3 dir)
{
    // this might be the result of a complex calculation
    return vec3(-dir.x, dir.y, dir.z);
}
```

We have seen the benefits of creating and using functions when we talked about the subject in C++.

There are many built-in functions in GLSL to assist us in developing our code. These include trigonometric functions, logarithmic and exponential functions, common mathematical functions, geometric functions, relation and logical functions, and texture functions. Appendix A has a list of the built-in GLSL functions.

Shader input and output

Data defined by our application is input for the shader. The shader performs some calculations and then outputs the results.

The *vertex shader* for example, accepts as input the vertex coordinates and converts them into clip coordinates, possibly using view and model transformations, for the *fragment shader* to use.

Input variables

Shaders know nothing about our world. They are great in exploiting the graphics hardware and performing complex calculations. We can layout our vertex data and attributes so that OpenGL can pass them to the shader.

Input variables to the shader, are declared with the keyword *in*:

```
in vec4 color;
```

Our first job is to 'tell' the shader how our data is organized. Let us assume that our vertex has its coordinates organized as a three-dimensional vector. For this information se are creating a variable of type *vec3*. As we saw in the previous chapter this information is the first we put in the *vertex array.* This is accessed by the shader with a *layout qualifier*. The *layout qualifier* specifies where the variable is stored within the *vertex array*.

To read the coordinates of the vertex we will create a variable of type *vec3* and call it *aPos*. In order to read the actual vertex data we must use the *layout qualifier*.

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

This means that in the first *buffer* of the *vertex array* and in the form of a *vec3* structure we have stored the vertex coordinates. Remember how we set up the vertex data:

```
// create a buffer for the vertices
glGenBuffers(1, &vertex_buffer);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
glEnableVertexAttribArray(0);
```

OpenGL knows how to '*walk'* through our data and set the data pointers correctly, and now our shader knows how to read that data.

The vertex sampler can have similar input, the texture coordinate of the vertex. This information has to be passed to the fragment shader. We will see how the vertex shader outputs the information in the next section, now we are interested in how the fragment shader reads it. Reading input requires the *in* directive at the variable declaration. Variables that are passed from one step to the other do not require the *layout qualifier*.

```
in vec2 texCoord;
```

Another type of input mechanism is the *uniform*. With this we pass names variables of any type we need to the shader. A *uniform* is a global storage location that is accessed by its name. we can write data to it from our C/C++ code and ten read that data in our shader.

We use the *glUniform* family of functions to write to a location. We can pass the camera matrix like this:

```
glUniformMatrix4v(glGetUniformLocation(program, "view"), 1, GL_FALSE, viewmat);
```

And then we read it in our shader like this:

```
uniform mat4 view;
```

Uniforms are accessible in all stages of the pipeline.

Output variables

As we have said shaders must generate some output. This either by setting OpenGL internal variables like vertex position in clip coordinates, or by setting variables for the next step in the pipeline.

gl_Position is a built-in function in OpenGL that holds the vertex location after we have finished our calculations about it. A list of the built-in functions can be found in Appendix B.

Here we are going to examine the second type, the variables passed to the next step. They are preceded with the keyword *out*, letting the compiler generate the code to link this variable with the next step. To complete the example of the previous section, where we needed to read the vertex texture coordinate in the vertex shader and pass it to the fragment shader, here is how we output the variable:

```
out vec2 texCoord;
```

Chapter 4: Drawing simple objects in 3D.

The two chapters that went before gave us some clue as of how to draw in OpenGL. We can create a window for our program and draw a simple tringle inside it. We can even use shaders and vertex arrays to store our data in the video memory so that the GPU can access it faster and speed up drawing.

The next step is to start creating solid objects on three dimensions. We must learn how to draw them and create some realistic three-dimensional effects.

We must also get a good understanding on how user vision works. Understand the mathematics behind the camera and its lens. How to handle the zoom and the field of view. What is the effect of camera position, orientation, and view direction?

For the needs of this tutorial I have created a small application framework. It is not a production framework, yet it has all the key components and architecture to be used as a tutorial of how applications can be made.

The first two components of the engine we need are the ***graphics*** and the ***math***. The first has all the Windows and OpenGL functionality we need, and the second the mathematics to support all our work.

We name the types as the *shading language* to make reading of the code easier. We did not use the ***glm*** library because it is big and harder to browse through the code.

The other components of the engine will be discussed in time when we need them as we proceed with our journey in OpenGL development.

The 3d mesh

A mesh is a collection of *vertices, edges,* and *faces*. A line between two vertices is an edge, and a closed polygon is a face. The polygons can be triangles, quadrilaterals or any convex or concave polygon. In OpenGL programming we prefer triangular meshes because they are simpler to render.

Meshes represent the surface and the volume of 3D objects. So we use them in our programs to draw our objects or to check for collisions.

The ***math*** library is based on triangular mesh to represent any solid object. This simplifies drawing and unifies our calculations.

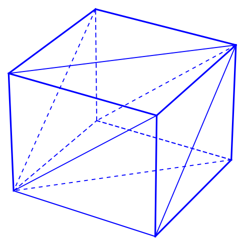

Figure 2: A meshed cube

An eye to the word

Before we go any further we should have some basic understanding about the way OpenGL defines and handles what we see and how.

As you might expect, starting with view definition in OpenGL is not that hard. Figure 3 shows the view volume as defined in OpenGL.

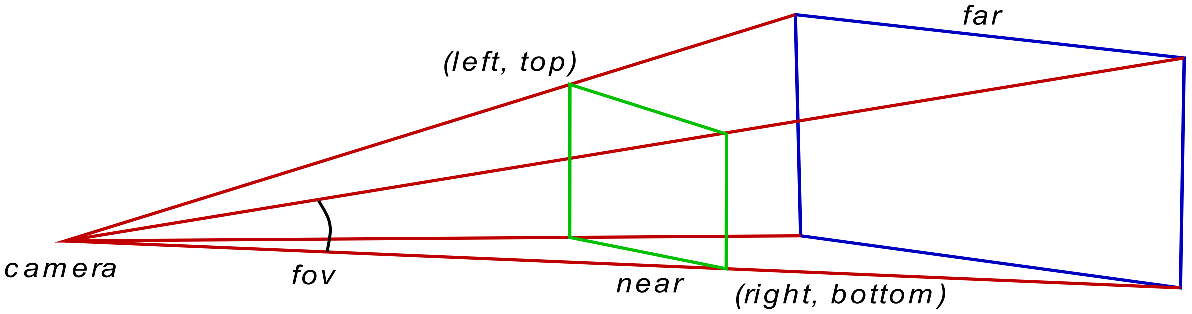

Figure 3: The OpenGL view volume

All we have to do is define the distance of the *near* and the *far* plane from the *camera position*, and the *(left, top)* and *(right, bottom)* points. Imagine a pyramid with its apex at the camera. The *far* plane is the base and all we have to do is define the *near* plane. OpenGL displays everything inside this volume and ignores anything else.

The geometric solid that defines the view volume is actually a frustum because it is part of a view cone. It is the shape of the computer screen that defines a pyramid, and this is the reason we talk about frustum but draw and calculate a pyramid.

The viewport projection matrix

The dimensions of the *view window* and the *near* and *far* distances are used to form what we call the *frustum matrix* or *viewport matrix*.

There is a functions for this purpose in the ***math*** library. It is in the file *cg_matrix.cpp* and this is its prototype:

```
void frustum_matrix(float* matrix, float left, float right, float bottom, float top,
                    float znear, float zfar);
```

The *left-top* and *right-bottom* points hardly make any sense to humans. We can better understand the angle indicating the *field of view*. This is a metric completely independent of the view distance. If you notice camera lenses and zoom are also measured by this angle. Focal length of a lens is actually a function of this angle.

The *cg_viewport* class which handles the viewport specifics in the ***graphics*** library has the function *set_fov* which sets this angle. Notice that the angle is in the vertical axis and not in the horizontal.

Having this angle it is a matter of simple trigonometry to calculate the two corner points and build the transformation matrix for the projection.

The camera matrix

Setting the viewport is independent of the location we are standing, the direction we are looking at, and our orientation. These parameters are moving the view volume around bringing objects in and out of view.

We can use three vectors to handle the camera. The first vector will hold the actual location of the camera, the second the point we are targeting, and the third will point to our *UP* direction.

All this is handled by the *cg_camera* class. It requires the three vectors we just described and returns us the corresponding transformation matrix by calling the *perspective* function.

The first thing we do when initializing the application is create a viewport and camera object to handle the way we see the world.

Shading things

As we saw in earlier the best practice when drawing in OpenGL is to exploit the GPU and use the pipeline.

The shaders we are going to use for the time being are very simple. Our primary goal is to explain how to handle the basic transformations in three dimensions. Drawing a simple triangle as we did before is not enough to master OpenGL.

So for our first real 3D drawing we are only using a *vertex* and a *fragment* shader. The first is used to calculate the transformations of the vertices and the second to set the drawing color.

The ***graphics*** library supports reading the shaders from external files and this is how we use them here.

These are very simple shaders so we can go with OpenGL version 3.3.

```
// vertex shader
#version 330 core
// vertex position in local coordinates
layout(location = 0) in vec3 aPos;
// camera has the combined viewing matrix
uniform mat4 camera;
// model has the combined matrix of object position and orientation
uniform mat4 model;

void main() {
    // calculate final vertex position in the 3D space
    gl_Position = camera * model * vec4(aPos, 1);
}

// fragment shader
#version 330 core
// drawing color for OpenGL to use
out vec4 color;
void main() {
    color = vec4(1, 1, 1, 1);
}
```

The only thing we need to pay attention to is the *vertex shader* which calculates the final position of the vertex. The *camera* matrix is the combined projection matrix for the *viewport* and the *camera*.

We pass a combined matrix for speed reasons. This shader is called for every vertex, and if the scene has many complex objects it will be called thousands of times. So it is better to save a matrix multiplication and perform it only once instead doing it for every vertex.

Take a good look at the *fragment* shader. In this sample it is just setting the drawing color to white. Here is the point where most of the visual effects take place. It will get more complex as we add effects.

Creating a simple solid

The first solid object we are going to create is a simple animated cube. We use a cube because it is a simple object, and we can easily understand its orientation.

I mentioned earlier that the base of all solid objects is the *mesh*. The cube is no exception. When we create a cube the library generates a mesh. This mesh will be used later for all physics related calculations. But for the time being we will stick to the mesh and how to use it in OpenGL.

When we create a *cg_gl_cube* the ***graphics*** library creates a mesh to represent the cube. In mathematics the mesh needs three things. The first element is the *vertices* of the mesh, the second element is the *normal vector* at each vertex and finally the order in which to access the vertices that represent the faces of the mesh and draw them. Our implementation contains one extra element the texture coordinates.

Normal vectors will be used later when we add lighting and texture coordinates when we add textures. Right now we are drawing in wireframe mode because drawing without light generates a flat monochrome image.

Solid objects can be moved and rotated. For this reason they have two member variables of type *vec3*, the *position* and the *rotation*. We are using *Euler angles* for the rotation and not *quaternions* to keep things simple. In the *frame_move* function we rotate and move the cube around the screen.

Drawing the cube

We saw that the cube is rotated around its axes and moved in an elliptical motion. These movements simply change the value of its internal variables. These values must be given to OpenGL somehow to render the cube correctly. All this is done in the *frame_render* function.

```
// set the viewport to the whole window
m_view->set_viewport(); // calls the glViewport function.

// clear screen
glClearColor(.0f, .0f, .0f, 1.f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// combine the view and camera matrices into one
mat4 cam_matrix = m_view->perspective() * m_cam->perspective();
// enable the shader
m_shader->use(); // glUseProgram
// set the combined view matrix in the shader
m_shader->set_mat4("camera", cam_matrix);
// and finally draw the cube
simple_cube->render(m_shader);
```

First we set the viewport to the whole screen, and we clear it. And then we start the actual drawing routine.

We start by calculating the combined projection matrix, we activate the shader, set the projection matrix in the shader, and finally tell the cube to draw itself. This is what happens when the cube draws itself.

```
// calculate object matrix
mat4 ob_matrix = tmat * rmat * smat;
// pass transformation to shader
_shader->set_mat4("model", ob_matrix);
glBindVertexArray(vao);
if (draw_elements)
{
    // setup drawing
    glFrontFace(GL_CCW);
    glPolygonMode(GL_FRONT, draw_mode);
    glDrawElements(GL_TRIANGLES, (unsigned int)m_mesh_data->indices.size(),
                   GL_UNSIGNED_SHORT, 0);
}
glBindVertexArray(0);
```

We start by combining the position, the orientation, and the scale of the object in one *model* matrix , which we pass to the shader. Then, after we setup the drawing mode, we draw the triangles.

We do not really need the scale of the object. We just included it to show how to use the scale matrix in the process.

Adding textures

Realistic 3D environments, especially in games, are based on illusions. Illusions generated by images. These are the textures applied on surfaces. It is a lot easier, and faster, to use the image of a complicated object, instead of drawing its geometry. Take for example the tire of a car. The tread is rather complex and drawing it requires a lot of graphics memory to store the geometry and GPU power to process it.

Now suppose we want to draw the earth rotating to show the change between day and night. Can you imagine the amount of data required to represent the earth's surface? If we apply a good image of the earth showing the continents and the sea on a simple sphere, we can create the illusion of the earth. We can then rotate the sphere any way we want and get a good view of the earth from any point of view we want.

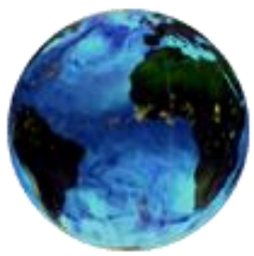

Figure 4: Textured sphere

We start by creating a sphere. Just like the cube in the previous section our little framework has a sphere generation built in. The process of creating a sphere is the same as creating a cube, we just call *new* to create an instance of the *cg_gl_sphere* class, which takes care of the *mesh* generation and all the *vertex buffers* and *vertex arrays*.

The difference from the cube we created before is in the texture we are going to load, and the shaders that set the drawing color based on this texture.

Textures are data that we store in the GPU memory. They are used to add detail to our objects. Here we focus on images that we wrap around the objects to give them the illusion of detail.

We load an image file, only *targa* images are supported, by calling the *load_texture* function, which returns us the texture ID on success, or -1 on failure.

Textures are similar to vertex buffers. We must generate them, a.k.a. allocate storage for them, and then bind them to apply any operations on them, like we do with vertex buffers.

```
GLuint load_texture(const char* fname)
{
    GLuint tex = -1; // default return is failure

    cg_image img;    // try to load the TGA image.
    if (!img.load(fname))
       return tex;   // return failure (invalid OpenGL id)

    // how bytes are aligned in memory
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    // enable textures for the following commands
    glEnable(GL_TEXTURE_2D);

    // we will generate a texture with mipmaps
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
    // generate texture
    glGenTextures(1, &tex);
    // and make it the current processing object
    glBindTexture(GL_TEXTURE_2D, tex);

    // how texture values (colors) are interpreted
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    // when texture area is small, bilinear filter the closest mipmap
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
    // when texture area is large, bilinear filter the original
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // the texture wraps over at the edges (repeat)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // now build the mipmaps
    gluBuild2DMipmaps(GL_TEXTURE_2D, 4,
       img.get_width(), img.get_height(),
       img.get_image_format(), img.get_data_type(), img.get_image());
    glBindTexture(GL_TEXTURE_2D, 0);
    img.release();// release the TGA image.

    return tex;
}
```

After we setup the memory alignment, we enable the use of textures, and we declare the textures to have *mipmaps*. *Mipmaps* are precalculated images, each of which is in lower resolution than the previous.

Figure 5:mipmap image

Figure 5:mipmap image shows a *mipmap* image. This image contains copies of progressively lower resolution of the original image. Using *mipmaps* allows OpenGL to use predefined images of lower resolution when needed instead of resampling the original image every time.

OpenGL can generate images like this when loading an image and use them when needed.

Then we generate and bind the texture so that all subsequent calls will act upon it. For our needs we setup our bitmaps to use the closest *mipmap* image and to repeat the texture when needed. Now that we are done parametrizing OpenGL we pass the image data and invoke the mipmap building engine.

When all is done we unbind the texture and release the image.

Mapping the texture

The texture was loaded into the GPU memory but how can we map the texture to the object? It is clear that there must be some mapping between the image and the surface of our object. There must be some way we can tell OpenGL how to draw using colors from the image.

Here is how OpenGL addresses the problem. First the texture is given coordinates from 0 to 1 in each direction, as we see in Figure 6. Then we can assign an *s* and *t* (for *horizontal* and *vertical*) between 0 and 1, to each vertex of our mesh. A simplified mapping of an image on a simple mesh is shown in Figure 6.

When it renders the image it samples according to these coordinates to color the vertices and it interpolates for the image coordinates corresponding to the rest of the triangles surface.

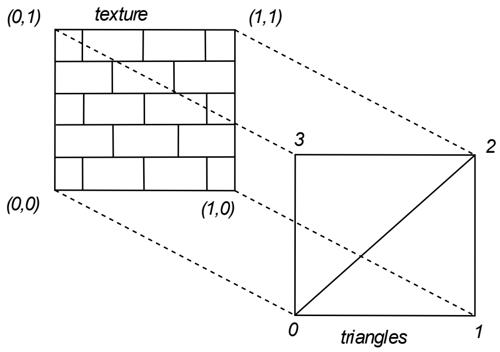

Figure 6: Texture mapping

Sampling the image

The sampling of the image is done in the *fragment shader*. There we read the pixel from the image and pass it to OpenGL to finally draw. But there are some things that have to be done before that.

It all starts in the *frame_render* function, which takes care of all our drawing. First we enable the shader we have written for this purpose:

```
// vertex shader
#version 330 core
// vertex position in local coordinates
layout(location = 0) in vec3 aPos;
// the texture coordinates
layout(location = 2) in vec2 aTexCoord;
// texture coordinate output for fragment shader
out vec2 texCoord;
// camera has the combined viewing matrix
uniform mat4 camera;
// model has the combined matrix of object position and orientation
uniform mat4 model;
void main() {
    // calculate final vertex position in the 3D space
    gl_Position = camera * model * vec4(aPos, 1);
    texCoord = aTexCoord;
}

// fragment shader
#version 330 core
// the texture coordinate
in vec2 texCoord;
// the image we take samples from
uniform sampler2D textureSampler;
// drawing color for OpenGL to use
out vec4 color;
void main() {
    color = vec4(texture(textureSampler, texCoord).rgb,1);
}
```

The vertex shader apart from setting the vertex coordinates has one more job to do. It reads the texture coordinates of the vertex and passes them to the fragment shader. The fragment shader then uses these coordinates to read the color data from the texture image and pass that value to OpenGL.

Now there is only one thing left to clear. Where do we find this image we sample from? This is done by binding the texture right before we call the sphere to render itself. The texture sampler accesses the last texture that was bound.

Chapter 5: Lights please

The main perception of three dimensions comes from the shades. The human eye can only see colors and not the outlines. It is the transition from one color to the other that signals the change into our brain. Similarly the change in the shadows and the difference of light and color perception between our eyes, informs our brain about the third dimension.

Color perception depends on light. The light that reaches our eyes stimulates our optical nerve. This light is emitted from various sources and reflected by the surfaces all around. The color of a surface is the combined result of the light that falls on that surface and its physical characteristics thar reflect or absorb part of that light.

In this chapter we will try to clear up things about the light and see how we can use all that in our advantage.

The color of things

We use special materials, usually liquid, to dye the surfaces of objects to our desired color. These liquids are called paints. No big news so far. But how do these paints work? How do they change what we see?

The color we see is actually the color of the light that reaches our eyes. The light of the sun is what we call the *white* light. It is the mixture of all the colors we know. Sir Isaak Newton was the first to understand it back in 1666.

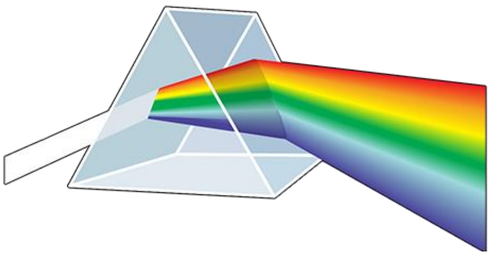

Figure 7: Color spectrum of white light

The various light sources emit anything from white light to light of a specific color or combination of colors that gives them their characteristic color. The light travels in a straight line but the dust particles in the air refract it and this is a part of the *ambient* light in the scene.

The light emitting sources are point sources. The light emitted from them goes radially. When the source is too far, say like the sun, we consider the light to be emitted in parallel. This *direct* light makes our scene bright, and the objects shine.

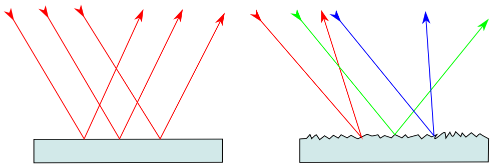

Figure 8: Reflection and diffusion

The light rays that reach the surface of an object are reflected back. Some of the light though is absorbed by the object and some is reflected. This results in the color we see for the object. This is the job of the paint, it absorbs all the colors but those we see.

Object surfaces are not as smooth as think. They are a little rough, enough to reflect the light to various directions. Figure 8 shows the ideal reflection on the left and the actual on the right. This phenomenon is called *diffusion* and scatters the light adding to the *ambient* light. Ambient light allows us to see in areas shadowed by objects.

The light that we see is the sum of the direct light that reaches our eyes, the reflected light from the surfaces of the objects in the form of *diffuse* and *specular* light, and the ambient light. The color of the objects is also a function of the color of the light that falls on them. If we drop red light on a blue surface we cannot expect to see the surface blue. It will be black if the light is pure red or some version of purple. The color of the light is described by the *diffuse* attribute of the light.

RGB

Red, Green, Blue. These three are the components that define any color. Our eyes are like camera sensors, or to put it right, we modeled camera sensors after our eyes. There are light sensing cells which help see the light and mainly the color.

These sensors are sensitive to red, green, and blue light. The combination of these three colors makes the colors we actually see. Deep in the human eye there are light receptors which are sensitive to these colors. This is the actual reason we have modeled the camera sensors like this. There are more sensors in the eye, but they are responsible for other features of our perception of the world around us.

OpenGL has modeled anything that has to do with color with *red, green,* and *blue.*

Reflections

The rays of light that are reflected of a surface are not scattered in random directions. Reflection follows certain rules. We are all familiar with these rules either by playing with a mirror in the sun or any framework with a ball that hits a wall and comes back.

The **law of reflection** indicates that the ***angle of incidence*** equals the ***angle of reflection***. This might mean nothing but taking a look at Figure 9 it will get clear. Here we have a ray falling on a smooth surface and we mark with red the normal vector of it. ***Angle of incidence*** is the angle between the ray moving towards the surface and the normal vector and is marked as θ. ***Angle of reflection*** is he angle between the ray moving away from the surface and the normal vector and it is marked with φ. These two angles are *always equal*.

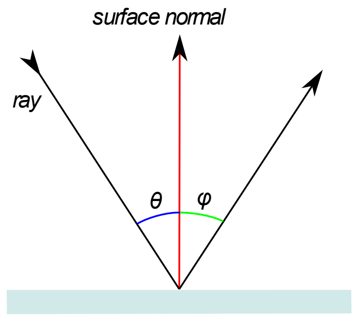

Figure 9: The law of reflection

If the surface is rough we might think that this is not the case. If we take a closer look though we will see that the surface at the point of incidence is not in the direction we thought it was. Surfaces as we said are rough at microscopic level, making us think that light behaves strangely, and not according to the laws of nature.

Putting it all together

We have seen how a simple source of light behaves. It is about time to add a light source into our scene and start adding some life to it.

For our needs we have created the ***lights_maerials*** sample. In this sample we create some lighting models of increasing complexity to show some techniques we can use to add realism in our scenes.

Our framework has a class to handle the light in the scene and we are going to use it. This is the *cg_light* class. It allows us to store the characteristics of the light in one place and pass them to the shader in a clean and organized way.

The light has three major attributes: *ambient, diffuse,* and *specular.* We will start the description from the end.

Specular light is the light that is directly reflected off the surface of an object to our eyes. Pretty much like the reflection of the sun light on a mirror. The color of this light is the product of the color of the light source and the color of the object. Its intensity depends on the orientation of the object.

Diffuse light is generated by the roughness of object surfaces. Because of the roughness of any surface, part of the light that falls on them is reflected at random directions. Some of this randomly reflected light reaches our eyes. Its color is again a product of the *diffuse* color of the light and the color of the object.

Ambient light is the light that has no direct source and seems to come from all directions. It is the result of consecutive reflections of the light on various surfaces in the scene, diffusions that may occur and even diffusion caused by the atmosphere. Exact calculation of the ambient light is too complex and time consuming, so we assume that it is a constant light that is the same in all directions.

All the calculations are done in the shader. Starting with the vertex shader which, apart from calculating the vertex position in the view and extracting the texture coordinates, has to calculate the position of the vertex in the 3D space as well as its *normal vector*. These two values will be needed to calculate the lighting.

Now that we have calculated the position and the normal vector of the vertex we can calculate the color of the fragment. We are starting with the easy part of our calculation, the *ambient* light. This is a simple multiplication between the *ambient* factor of the light and the light color. We do not have to calculate any direction or reflection, so we are done with one of the light components.

The other two components are the *specular* light, and the *diffused* light. These require the calculation of the light direction, the view direction, and the reflection of the light on the surface.

For the *specular* part of the light we calculate the strength of the reflection. This determines how the reflection spreads over the surface of the object, a.k.a. how big is the bright spot on the surface. Then we combine it with the properties of the material, how shiny the object is. We will return to this later when we talk about materials, for the moment we just give it a value.

The *diffusion* determines the amount of light that comes to our eyes when the object surface is not reflecting the light to us. This is why we need the orientation calculations. We need to know if the light falls directly on it, and if so how bright it may seem to us.

The amount and the hue of the light that falls on the object is the sum of the three components we calculated. Multiplying it with the color of the object we obtain the final color we see.

This is the vertex shader we just described:

```
// we use the same vertex shader for all the shaders in this chapter
#version 330 core
// these are set in the array buffer
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;

// the transformation matrices
uniform mat4 camera;
uniform mat4 model;

// data we pass to the fragment shader
out vec2 texCoord;
out vec3 Normal;
out vec3 pos;

void main() {
    // some code optimization
    // this product appears twice in our code
    // so we calculate it once
    vec4 p = model * vec4(aPos, 1);
    // first occurrence in gl_Position calculation
    gl_Position = camera * p; // model* vec4(aPos, 1);
    texCoord = aTexCoord;
    // vertex position in space (no view)
    // second occurrence in vertex position in 3D
    pos = vec3(p); //  model* vec4(aPos, 1));
    Normal = mat3(transpose(inverse(model))) * aNormal;
}
```

And this is the fragment shader:

```
#version 330 core

struct lightsource {
    int type;              // SPOTLIGHT=1, DIRLIGHT=2
    vec3 pos_or_dir;       // light location
    vec3 ambient;          // the ambience property of the light
    vec3 diffuse;          // the diffuse property of the light.
    vec3 specular;         // the specular property of the light
};
uniform lightsource light;
uniform vec3 cameraPos;   // viewer location
uniform vec3 objectColor; // object color

in vec3 Normal;           // surface normal
in vec3 pos;              // drawing position

out vec3 color;           // resulting drawing color.

void main() {
    // ambient color
    vec3 ambient = light.ambient;
    // normalize is a built-in function in GLSL
    // surface normal
    vec3 norm = normalize(Normal);

    // light direction
    vec3 lightDir;
    if (light.type == 1) // SPOTLIGHT
        lightDir = normalize(light.pos_or_dir - pos);
    else                 // DIRLIGHT
        lightDir = normalize(-light.pos_or_dir);

    // view direction
    vec3 viewDir = normalize(cameraPos - pos);

    // reflection vector, reflect is a built-in function in GLSL
    vec3 reflectDir = reflect(-lightDir, norm);

    // pow, max, dot are built in functions in GLSL
    // specular strength
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 128);
    // specular color
    vec3 specular = spec * light.specular;

    // diffusion factor (calculated by the reflection angle)
    float diff = max(dot(norm, lightDir), 0.0);

    // and diffusion color
    vec3 diffuse = diff * light.diffuse;

    // light emitted from object is (ambient + diffuse + specular)
    color = (ambient + diffuse + specular) * objectColor;
}
```

Materials

In the previous section we introduced several parameters that involved the materials of the objects and we just used arbitrary values without any explanation. In this section we are going to explain these parameters and see how they change the appearance on the objects.

So far the only parameter we had for the material of the object was the color. We all know that this is not enough. Different materials behave differently under light. Obsidian for instance is different to black rubber of a car’s tires, although they are both black.

Light and color equations are designed to consider the nature of the material and not only the color. all we have to do is create another communication channel between our program and the OpenGL pipeline to pass these parameters. Then with some minor adjustments to the shader we will reach our goal.

To make all our samples comparable and clear we will use a point light source, pure white that emits light evenly to all directions. This will allow us to see and compare the effect of the materials on the result.

We use three different types of reflection to describe the light emitted from an object when the light hits it. These are *ambient*, *diffuse,* and *specular* reflections.

Ambient reflection is caused by the ambient light that falls on the surface of the object. Some of this light is absorbed and some is reflected. Since there is no clear direction in the light, ambient reflection is uniform at every point on the surface. The color reflected depends on the color of the surface.

Diffuse reflection is caused by the roughness of the surface. This makes the light reflect in random directions.

Specular reflection is the clear reflection we get from smooth surfaces like polished metal. This gives us the shiny look of the objects. Apart from the color reflected, *specular* reflection has one more parameter, the ***shininess*** of the object.

All reflections have three components for each of the *red, green,* and *blue* components of the white light, except *shininess* which is a scalar value. In the following table you can see the values for some common materials as defined by OpenGL.

The *cg_material* class packs all these parameters and handles the communication with the shading pipeline, writing the values in the shader program memory. The shader we created has a new structure called *material* which holds the same variables as we did before with the light.

When we have material properties we do not consider the *ambient* component of the *lightsource* structure, but we use the ambient component of the *material* instead.

The *vertex* shader is the same, but the *fragment* shader has been modified to use the properties of the material.

```glsl
#version 330 core

struct lightsource {
    int type;           // SPOTLIGHT=1, DIRLIGHT=2
    vec3 pos_or_dir;    // light location
    vec3 ambient;      // the ambience property of the light
    vec3 diffuse;      // the ambience property of the light.
    vec3 specular;      // the ambience property of the light
};
uniform lightsource light;

struct material {
    // how the material reacts to light
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shine;  // surface shine
};
uniform material mat;

in vec3 Normal;            // surface normal
in vec3 pos;               // drawing position

uniform vec3 cameraPos;    // viewer location

out vec4 color;            // resulting drawing color.

void main() {
    // ambient color based on material properties as well
    vec3 ambient = light.ambient * mat.ambient;

    // surface normal
    vec3 norm = normalize(Normal);

    // light direction
    vec3 lightDir;
    if (light.type == 1)
        lightDir = normalize(light.pos_or_dir - pos);
    else
        lightDir = normalize(-light.pos_or_dir);

    // view direction
    vec3 viewDir = normalize(cameraPos - pos);
    // reflection vector
    vec3 reflectDir = reflect(-lightDir, norm);
    // specular strength
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), mat.shine);
    // at last, specular color
    vec3 specular = (spec * mat.specular) * light.specular;

    // diffusion factor (calculated by the reflection angle)
    float diff = max(dot(norm, lightDir), 0.0);
    // and diffusion color
    vec3 diffuse = light.diffuse * (diff * mat.diffuse);

    // light emitted from object is (ambient + diffuse + specular)
    vec3 result = (ambient + diffuse + specular);

    color = vec4(result, 1);
}
```

Color maps

The previous approach assumes a solid material like a metallic object, or an evenly painted surface. We all know that this is not the case for all objects. In the previous chapter we learned how to apply textures and make our objects more realistic. Here we will learn how to combine textures with material properties for even better results.

Textures

We are starting with textures. We are going to use a texture as material surface. Color is not enough to represent the details of the surface of the object. It is not enough when we want to draw a wooden box or the earth.

In the previous chapter we used an image of the earth as a texture for a sphere. Here we will use a similar image on a sphere and an image of wooden planks on a box. Only this time we will treat them as material properties rather than general textures. This will let us select from different images just by selecting different materials.

OpenGL can load several images as textures and we can address them as *GL_TEXTURE0*, *GL_TEXTURE1* and so on. Then we can activate the one we want to use and start drawing.

When we are dealing with a simple texture as we did before all we do is activate the first texture buffer in OpenGL and bind the texture image to that, like this.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
```

This simple approach gives meaning to the *textureSampler* variable in the *fragment shader,* and we can use it to extract parts of the image and render them on the scene.

The same principles apply when we use material techniques. We see that in the fragment shader we request the color using the *texture* function, passing it the texture buffer id, in our case *GL_TEXTURE0* etc. , and the texture coordinate. Now instead of having the texture sampler as a global *uniform*, we put it in the material structure, which now has one more member the *diffuse_map.*

Things in the fragment shader are straight forward. The color of our object is: `texture(mat.diffuse_map, texCoord.rgb)`, where *mat* is the variable holding the material properties.

Now we know how to use the texture and we have reorganized our fragment shader. The next step is to learn what goes before that. How to prepare and set the textures in our program.

If we use *GL_TEXTURE0* our *sampler2D* parameter should be 0, for *GL_TEXTURE1* it should be 1, and so on. This was the default behavior of OpenGL when we used one texture, everything was set to 0 and we used *GL_TEXTURE0.* So our *material* class has two new variables: *diffuse_map* and *diffuse_index*. *Diffuse_map* stores the texture id if the loaded image, and the *diffuse_index* is the active texture index. Here is the code to set the parameters:

```
shader->set_int("mat.diffuse_map", diffuse_index); // GL_TEXTURE0=0...
glActiveTexture(GL_TEXTURE0 + diffuse_index);
glBindTexture(GL_TEXTURE_2D, diffuse_map);
```

The fragment color derived from the texture image is used in the calculation of the *ambient* and the *diffuse* color of the object:

```
// ambient color
vec3 ambient = light.ambient * texture(mat.diffuse_map, texCoord).rgb;
// diffusion color
vec3 diffuse = light.diffuse * diff * texture(mat.diffuse_map, texCoord).rgb;
```

Specular reflections

The use of textures adds to the realism of the scene, but it also raises a new problem. The light is not reflected uniformly on the surface of the object. The object may have materials that do not behave the same under the light. Take the earth for example, the water of the oceans reflects the light more than the land.

To achieve this kind of lighting we need to pass a per fragment information to the shader. This is done using the same technique as the texture map. We create an image like the texture image, only this time we are interested in the amount of shine each fragment will receive. In the case of our hypothetical example of earth, we paint the oceans white and the land black, to define maximum shine for the water and no shine for the land.

Now we add one more *sampler2D* texture to our material. This means that our material will activate and bind two texture buffers in OpenGL, one for the texture itself, and one for the specula map.

The complete code in the material class that sets both the diffuse and specular maps is like this:

```
if (diffuse_index >= 0)
{
    shdr->set_int("mat.diffuse_map", diffuse_index); // GL_TEXTURE0=0...
    glActiveTexture(GL_TEXTURE0 + diffuse_index);
    glBindTexture(GL_TEXTURE_2D, diffuse_map);
}
if (specular_index >= 0)
{
    shdr->set_int("mat.specular_map", specular_index); // GL_TEXTURE0=0...
    glActiveTexture(GL_TEXTURE0 + specular_index);
    glBindTexture(GL_TEXTURE_2D, specular_map);
}
```

As you can see the code is practically identical. Now in our program we set the material like this:

```
m_mat5->set_diffuse(diffuse_texture, 0);  // use GL_TEXTURE0 for texture
m_mat5->set_specular(specular_texture, 1);// use GL_TEXTURE1 for specular
```

We set the texture images we loaded, and the texture buffers we want to use. The complete fragment shader is this:

```glsl
#version 330 core

struct lightsource {
    int type;          // SPOTLIGHT=1, DIRLIGHT=2
    vec3 pos_or_dir;   // light location
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform lightsource light;
struct material {
    // how the material reacts to light
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shine;  // surface shine
    sampler2D diffuse_map;   // the surface texture.
    sampler2D specular_map;  // the specular reflection map
};
uniform material mat;
in vec2 texCoord;

in vec3 Normal;            // surface normal
in vec3 pos;               // drawing position
uniform vec3 cameraPos;    // viewer location
out vec4 color;            // resulting drawing color.

void main() {
    // ambient color
    vec3 ambient = light.ambient * texture(mat.diffuse_map, texCoord).rgb;

    // surface normal
    vec3 norm = normalize(Normal);
    // light direction
    vec3 lightDir;
    if (light.type == 1)
        lightDir = normalize(light.pos_or_dir - pos);
    else
        lightDir = normalize(-light.pos_or_dir);
    // diffusion factor (calculated by the reflection angle)
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = light.diffuse * diff *
                   texture(mat.diffuse_map, texCoord).rgb;
    // specular
    vec3 viewDir = normalize(cameraPos - pos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), mat.shine);
    vec3 specular = light.specular * spec *
                    texture(mat.specular_map, texCoord).rgb;

    // light emitted from object is (ambient + diffuse + specular)
    vec3 result = ambient + diffuse + specular;

    color = vec4(result, 1);
}
```

Chapter 6: Advanced concepts

In this chapter we will learn some advanced OpenGL techniques that will spice up our scenes. Apart from learning these techniques our main goal is to see how they can make our programs a little more interesting and eye-catching.

For example how can we implement the rear-view mirror of a car, or even how can we see through a keyhole or a glass window? And then we were introduced in the use of light, but light casts shadows, how do we deal with them?

Stencil Buffers

In graphic arts one of the most used tools is the stencil. This tool allows us to draw a pattern by applying ink or paint. It might be one of the oldest drawing tools ever used. The most common application of a stencil in the prehistoric time is the human hand on the walls of the caves, where people painted around their hands.

Figure 10: Stencil

This idea evolved, and now we have created tools that allow us to draw from simple text to complex patterns, using the same principle. By cutting out specific shapes on a piece of paper, we can create a stencil with letters as we see in Figure 10.

OpenGL has a similar mechanism we can use to keep our drawing inside any shape.

What we do is create a 'canvas', or stencil buffer in technical terms, the size of the screen. Then we 'cut' out the openings that allow the paint to pass and finally draw. Here is a description of how we do it.

As you may have noticed in the beginning of our drawing function we clear the screen

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

only in this case we have added a third option, the `GL_STENCIL_BUFFER_BIT` which clears the stencil canvas as well and initializes it to screen size. The stencil buffer is typically initialized to black. Everything we draw on it is done in white and that marks the cut outs that will enable us to do the stencil drawing of our scene.

Using stencil buffers is a four-step process as opposed to normal drawing which is a one step process. First we enable writing to the stencil buffer. Then we draw our stencil shape. Third we enable normal drawing, and finally we draw our scene.

```
virtual void frame_render() {
    // set the viewport to the whole window
    m_view->set_viewport();

    // GL_STENCIL_BUFFER_BIT  needs mask=0xFF
    glStencilMask(0xff);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    // we are using stencil
    glEnable(GL_STENCIL_TEST);
    // prepare the stencil buffer
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);
    glStencilFunc(GL_NEVER, 1, 0xFF);
    glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP);  // draw 1s on test fail (always)

    // draw stencil pattern
    m_shader->use();
    mat4 ob_matrix;
    ob_matrix.loadIdentity();  // at the center of the screen
    m_shader->set_mat4("model", ob_matrix);
    m_shader->set_mat4("camera", ob_matrix);
    // move the window
    m_shader->set_vec3("displacement", vec3(xcen, ycen, 0));
    // disabling these to speed up drawing
    glDisable(GL_CULL_FACE);
    glDisable(GL_DEPTH_TEST);
    pstencil->render(NULL);

    // return to normal drawing
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
    glDepthMask(GL_TRUE);
    glStencilMask(0x00); // do NOT draw on black.
    // draw only where stencil's value is 1
    glStencilFunc(GL_EQUAL, 1, 0xFF);
    // enable them before we start normal drawing
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);

    // stop moving around
    m_shader->set_vec3("displacement", vec3(0, 0, 0));
    // and proceed with normal drawing
    mat4 cam_matrix = m_view->perspective() * m_cam->perspective();
    m_light->apply(m_shader);
    m_shader->set_mat4("camera", cam_matrix);
    m_shader->set_vec3("cameraPos", m_cam->vLocation);

    // draw a rectangle behind the cube to act as background
    // this will make the stencil shape visible
    m_shader->set_vec3("objectColor", vec3(0.75f, 0.85f, 0.85f));
    pbackgound->render(m_shader);
    // and now draw the cube
    m_shader->set_vec3("objectColor", vec3(.1f, .2f, .9f));
    m_cube->render(m_shader);
    // stop using stencil
    glDisable(GL_STENCIL_TEST);
    glUseProgram(0);
}
```

Blending

Blending in OpenGL means the mixing of colors when we see our scene through a semitransparent medium. This is the case when we see the world through our sunglasses, for example.

Applying glass color is very common in virtual reality simulation and everyday practice in games.

The technique is very simple, and the results are sometimes spectacular. So let us start demystifying it.

We will start with the object's material. We will not use the materials from the previous chapter but only the object's color, to keep things simple and focus only on what is important.

The color of the object can have a fourth component, apart from the *red, green,* and *blue,* we used so far. This is usually referred to as *alpha* channel or *transparency*. An opaque material has the value of 1 and a fully transparent has the value 0. Common objects have values anywhere between these limits.

Drawing with blending is simple but it requires some work on our side. It has to be done in a specific order to achieve the desired and correct results.

First we upgrade our color to have four components adding the opacity in the alpha channel. So our color variable for the objects' colors are now of type *vec4* instead of *vec3.* Then we start drawing all our opaque objects. In our sample this is a spinning cube.

When we are done with the opaque objects, we draw our transparent objects. This is the process we will analyze more because here is where all the magic happens.

Our transparent objects must be ordered starting from the farthest and finishing to the closest to the viewer, and then be drawn in that order. All color calculations required for blending are performed as we pass the objects to the pipeline, and drawing order really makes a difference.

We can use texture images as textures for our transparent objects. This is a lot more flexible than using a color value for the entire surface. We can add a byte per pixel in the target image, as the alpha channel, and encode in it 256 different levels of opacity. This can give our object varying opacity over its area. We see this in our example as the blue 'glass' changes from completely opaque to completely transparent. It is combined with a red uniform glass to show how two transparent objects can be combined.

```
m_shader->use();
m_light->apply(m_shader);
// set the texture we will use
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);

mat4 cam_matrix = m_view->perspective() * m_cam->perspective();
m_shader->set_mat4("camera", cam_matrix);
m_shader->set_vec3("cameraPos", m_cam->vLocation);

// draw opaque objects first
m_shader->set_vec4("objectColor", vec4(0, 1, 0, 1));
// use object color parameter, do not look for texture
m_shader->set_int("useColor", 1);
m_cube->render(m_shader);

// start blending
glEnable(GL_BLEND);
glBlendFunc(modes[source_mode], modes[dest_mode]);
// and now transparent objects (sorted)
if (draw_order == 0) // the blue glass goes back
{
    // ignore object color and use texture
    m_shader->set_int("useColor", 0);
    pglass->move_to(vec3(0, 0, -1.5f));
    pglass->render(m_shader);

    m_shader->set_vec4("objectColor", vec4(.9f, .1f, .1f, 0.5f));
    m_shader->set_int("useColor", 1);
    pglass->move_to(vec3(0, 0, 0));
    pglass->render(m_shader);
}
else  // the red glass goes back.
{
    m_shader->set_vec4("objectColor", vec4(.9f, .1f, .1f, 0.5f));
    m_shader->set_int("useColor", 1);
    pglass->move_to(vec3(0, 0, -1.5f));
    pglass->render(m_shader);

    m_shader->set_int("useColor", 0);
    pglass->move_to(vec3(0, 0, 0));
    pglass->render(m_shader);
}
glBindTexture(GL_TEXTURE_2D, 0);
// stop blending
glDisable(GL_BLEND);
```

As we said before the color of the incoming fragment is combined with the color of the corresponding fragment in the frame buffer.

The following equation is applied.

$$Color_{final} = Color_{source} * Factor_{source} + Color_{dest} * Factor_{dest}$$

Where *source* is the output of the fragment shader, and *dest* is the content of the frame buffer. The *Factor* is calculated according to the parameters given to the *glBlendFunc*.

Option	Factor is equal to
GL_ZERO	0
GL_ONE	1
GL_SRC_COLOR	$Color_{source}$
GL_ONE_MINUS_SRC_COLOR	$(1,1,1,1) - Color_{source}$

GL_DST_COLOR	$Color_{dest}$
GL_ONE_MINUS_DST_COLOR	$(1,1,1,1) - Color_{dest}$
GL_SRC_ALPHA	$(A_{source}, A_{source}, A_{source}, A_{source})$
GL_ONE_MINUS_SRC_ALPHA	$(1,1,1,1) - (A_{source}, A_{source}, A_{source}, A_{source})$
GL_DST_ALPHA	$(A_{dest}, A_{dest}, A_{dest}, A_{dest})$
GL_ONE_MINUS_DST_ALPHA	$(1,1,1,1) - (A_{dest}, A_{dest}, A_{dest}, A_{dest})$

You can experiment with the blending options and see how OpenGL handles it. Blending in OpenGL is not difficult. There is only one thing to remember. Never break the order of execution:

- Render opaque objects.
- Render transparent objects sorted according to their distance from the viewer.

Face culling

The number of faces in 3D applications increases as the available computation power increases. The situation in computer games is even worse. The system has to perform all those tasks required by the gameplay raising the barrier even higher. If only there was a way to skip drawing some of the faces of an object.

We know that a great part of the surface of a solid object is invisible to us. The part that faces away from us, the back of the object as we call it. The viewer never sees those faces, and they usually measure more than 50% of the total number of faces we must draw in our scene. Imagine the amount of work required to calculate how to draw things that will finally be rejected because they are behind other objects.

There is an easy way to define if a face is pointing towards the viewer. We will use a simple triangle to illustrate the method. As you can see in Figure 11, we can define the triangle issuing its vertices in a counter clock or clockwise manner. If we define all our triangles consistently, then the triangles in the back facing surfaces will be drawn the opposite way due to the space transformation. This is a clear sign that we can skip them.

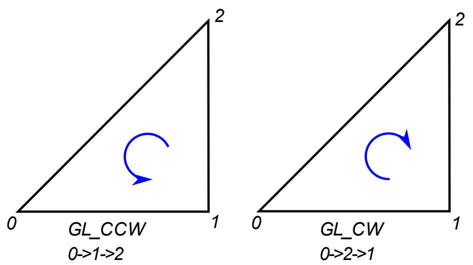

Figure 11: Clock and Counterclockwise definition of a triangle

We can 'tell' OpenGL which rotation is front facing and which is back facing, as well as which is to be omitted from drawing. I prefer to define counterclockwise as the positive rotation because it follows the right-hand rule which I find really helpful when setting up the calculations for my game. The basic rule is to be consistent, and you can follow any rule you feel more comfortable with.

To achieve back face culling is relatively easy. We start by enabling the feature in OpenGL.

```
glEnable(GL_CULL_FACE);
```

Then we define which rotation faces front, and which we want to draw, front facing, back facing or both.

```
glFrontFace(GL_CCW);
glPolygonMode(GL_FRONT, draw_mode);
```

The benefit of using this technique in small programs like the samples of this book is negligible. As the points and faces count increases though it makes a huge difference.

Cubemap / Skybox

Cubemaps simplify the texture we can apply on a cube, hence the name cubemap. It is an easy way to draw the distant horizon, the sky above and the earth below.

Imagine yourself being in a huge cubic box, the faces of which display carefully selected images of the surrounding environment, creating a panorama of whatever is around us.

These are the steps we need to follow to create a *cubemap*. The first step is to generate a texture, just like any other texture by calling the glGenTextures function. Then we have to activate this texture by binding it with glBindTexture. Here is the first difference from ordinary textures. Instead of using the GL_TEXTURE_2D parameter, we use the GL_TEXTURE_CUBE_MAP. This is used in all the texture functions we use for the *cubemap*.

The next step is to load the images. Loading the images has no new feature to talk about. After loading the images we store them in OpenGL using glTexImage2D. The first parameter of this function is one of the cube face ids. Let me explain this in more detail.

The first value defined in OpenGL is GL_TEXTURE_CUBE_MAP_POSITIVE_X. This means that the corresponding image will be used for the face lying on the YZ plane and has positive X coordinate. The five other values are GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, mapping the other five faces.

At the end we have created a texture with six layers, as referred to in OpenGL documentation. Each layer has an ID based on its location and is assigned a texture image.

The coordinate system for *cubemaps* used internally by OpenGL is left-handed. So positive X points to the RIGHT, positive Y points UP, and positive Z points to the from the viewer to the screen. On the other hand, our framework uses a right-handed coordinate system. This combination creates a problem when displaying *cubemaps*. The textures are rotated by 180° and left and right are swapped. To fix this we rotate the images after we have loaded them, and we swap the left and right images.

Here is the code that loads the texture images for the cubemap.

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

for (auto i = 0; i < 6; i++)
{
    cg_image* img = new cg_image;
    img->load(faces[i].c_str());
    // the coordinate system for cubemaps is left-handed
    // and the coordinate system of our engine is right-handed
    // so we rotate the images by 180 degrees
    img->rotate180();
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, img->format(),
        img->width(), img->height(), 0, img->format(),
        GL_UNSIGNED_BYTE, img->image());
    delete img;
}
```

The naming and coordinate convention we need to follow when we use the underlying engine to draw a skybox is as follows.

Coordinate macro	Relative position
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Up
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Down
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Front
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Back

Our cubemap texture is now ready. The next thing we are going to analyze is the drawing of the skybox. We expect drawing to be easy, after all this is the reason we use this technique.

And it is really easy to use the skybox. To begin with all we need is the vertex coordinates. No texture mapping coordinates or vertex normal are required. Since our object is a simple cube and the texture maps directly to the vertices of each face, OpenGL can easily choose the correct image and mapping based on the vertex coordinates alone.

The other thing we need is the location and orientation of the camera, the point in space we are looking to, and the angle of our field of view. In other words the *view* and *camera* matrices. We should note here that we remove the positioning part from the camera matrix. This is done because we are placing our viewer in the center of the cube, which is supposed to be really big.

The viewer matrix actually defines the view vector which is used by OpenGL to determine the fragment of the texture to render. It will automatically select the image or layer to use and apply all the necessary transformations. The shader code is in the *cg_skybox.cpp* file for easier installation of the engine and the samples.

```
// the skybox vertex shader
#version 330 core

layout(location = 0) in vec3 aPos;
uniform mat4 view;

out vec3 texCoords;

void main()
{
    // vertex coordinates are used for texture coordinates
    texCoords = aPos;
    vec4 pos = view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}

// the skybox fragment shader
#version 330 core

in vec3 texCoords;
uniform samplerCube cg_skybox;

out vec4 color;

void main()
{
    color = texture(cg_skybox, texCoords);
}
```

Environment mapping

We can dramatically improve our drawing and make it more realistic if we calculate the reflections of the environment onto the surfaces of our objects. The technique is called *raytracing* and it is a simulation of whatever actually happens in nature.

Unfortunately this is a very computationally intensive technique, and complex scenes take too long to render. Even with state-of-the-art hardware, the rendering speed is not good for gaming or any other real-time application.

One technique to help us achieve equally great results is *Environment mapping.* In this we create reflections of texture images on the surfaces we want. This is very efficient since all the light effects are precalculated and encoded in the texture image.

This technique complements the skybox we saw before. Instead of rendering the images on the surrounding cube, we render their reflections on the objects in the scene. The reflections can be rendered even if we do not draw the skybox.

We just setup the texture as we do in the skybox and then with the help of our shaders we calculate the reflections on the objects. In the sample code (chapter27c.cpp) you can comment out the rendering of the skybox and still see it rendered on the objects.

To use *environment mapping* we create a skybox and use its texture. The skybox class contains all the code required to load the texture images and then use them to generate OpenGL textures.

All the work is done in the shader and more precisely in the *fragment shader*. In the *vertex shader* we perform the usual calculations we did in when we had to apply light and texture:

```
// vertex shader
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;
uniform mat4 camera;
uniform mat4 model;

out vec3 Normal;
out vec3 pos;
out vec2 texCoord;

// nothing unusual in the vertex shader
void main() {
    gl_Position = camera * model * vec4(aPos, 1);
    pos = vec3(model * vec4(aPos, 1));
    Normal = mat3(transpose(inverse(model))) * aNormal;
    texCoord = aTexCoord;
}
```

In the *fragment shader* on the other hand we calculate the vector from the camera position to fragment position and calculate its reflection on the surface with the help of the normal vector. The reflected vector points to the location on the *skybox* and that is where we get the color to render.

```
// fragment shader
#version 330 core

uniform int useColor;        // use color=1 or texture=0
uniform vec4 objectColor;    // object color to use.
uniform vec3 cameraPos;      // viewer location
uniform samplerCube skybox; // skybox texture

in vec3 Normal;              // surface normal
in vec3 pos;                 // drawing position

out vec4 color;              // resulting drawing color.

void main() {
    // used when drawing text
    if (useColor == 1)
    {
        color = objectColor;
    }
    // used for textured objects
    else
    {
        // calculate reflection and not direct view
        vec3 view = normalize(pos - cameraPos);
        // reflect is a built-in function in GLSL
        vec3 reflection = reflect(view, normalize(Normal));
        // the reflection vector determines the output color
        color = vec4(texture(skybox, reflection).rgb, 1.0);
    }
}
```

The reflection is calculated by the *reflect* function which is a built-in function in GLSL. It takes two arguments, the vector to be reflected and the normal vector of the surface at the reflection point.

Frame buffers

According to the official definition, a *Frame buffer* in OpenGL is a buffer that can be used as the destination for rendering. There are two types of frame buffers. First is the default frame buffer, which we have used so far for rendering, and the user-created frame buffer which is usually called Frame Buffer Object of FBO for short.

You may wonder why we need frame buffers and what we can do with them. Well consider the following situation. Somewhere in your game scene there is a television which is on, playing some animation. Using a frame buffer can help you create that effect. You can render the animation using a frame buffer and then use that buffer as a texture for the television screen when rendering the main scene.

As is the case with all the OpenGL tools we have seen so far, a frame buffer has to be created before we can start using it. you can view the frame buffer as a painter's canvas on which we can draw and then hang it on our wall as a piece of art that decorates our scene.

Here is the code that creates the frame buffer.

```
unsigned int cg_texture_buffer::create(int w, int h)
{
    glGenFramebuffers(1, &m_buffer);
    glBindFramebuffer(GL_FRAMEBUFFER, m_buffer);
    // create a color attachment texture
    glGenTextures(1, &m_texture);
    glBindTexture(GL_TEXTURE_2D, m_texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0,
                           GL_RGB, GL_UNSIGNED_BYTE, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, m_texture, 0);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    return m_buffer;
}
```

You will notice the great resemblance between this and the creation of other objects we have seen so far. In this case we create a frame buffer and then we create a texture which we attach to the frame buffer. Something like setting a frame and a canvas.

After its creation we can leave the frame buffer until we need to draw on it. When that time comes we must call the `glBindFramebuffer` function passing the buffer ID like this:

```
glBindFramebuffer(GL_FRAMEBUFFER, m_buffer);
```

From that point on all our drawing is redirected to the frame buffer. To stop the redirection we call the same function with a value of 0 as a frame buffer ID. Now the frame buffer contains an image that can be used for anything we want. The most common use is as a texture image somewhere in our program.

The texture object inside the frame buffer class is just like any other image texture we have used so far. All we have to do is bind and use it in our drawing.

In the example *frame_buffers* I use two different shaders, one while rendering in the frame buffer and one while rendering the actual scene. A close look at the shader code will make clear that the only reason for this is to demonstrate that they can be two completely different scenes.

In the first part of the rendering function we draw in the frame buffer a rotating cube. Then in the second part we use that image as a texture for a rotating sphere.

Shadows

We are all familiar with shadows in our everyday life. So far we have learned how to deal with light in games. We have learned how it reflects on the surfaces of objects and creates all the nice effects. However our scenes seem empty. Our brain expects the light to cast shadows and when we do not see them we get that strange feeling.

According to the dictionary shadow is *the dark figure cast upon a surface by a body intercepting the rays from a source of light*. This definition really shows us what to do to get some realistic renderings with shadows.

The best way to calculate shadows is the technique called *raytracing*. In this method we trace the rays of light and simulate the effect they have when the hit an object. This way we can generate all the shininess and reflection we have seen so far. This automatically generates shadows since the areas the light cannot reach remain unlit.

Raytracing is the method of choice for high quality rendering and modern graphic cards provide specialized hardware to assist calculations. However it is still far from becoming the mainstream technique for games. You see ray-tracing algorithms are very intensive and the time needed for a scene is too long compared to the time available when playing a game.

To overcome the speed problem and still have fairly realistic scenes other techniques have been proposed and are used by game developers. The technique used by the most video games is *shadow mapping*. This is an easy to understand and implement technique, as well as fast with great results. All these make it an excellent choice for video games.

We will start by explaining the idea behind shadow mapping. First take a look at Figure 12. The areas marked in red are in the way of light and their color can be calculated as we have seen before. The rest of the surfaces are not lit directly because the light is obscured.

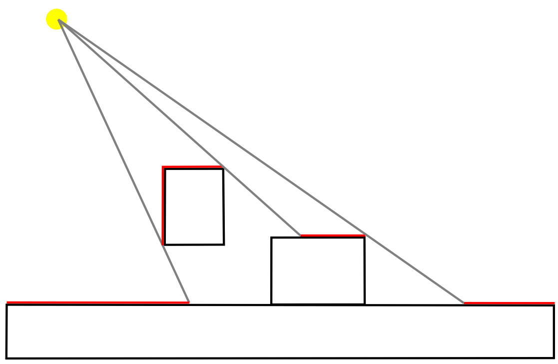

Figure 12: Objects in the shadow

A closer look at this image reveals the actual algorithm on which shadow mapping is based.

In this algorithm we create an image from the light point of view. We imagine our camera is at the position of the light source and it is looking to the direction the light points.

This approach reveals the fragments of the scene that receive direct light. Here come the real trick. In the frame buffer we store the distance of the fragment from the light source, or our imaginary camera. Every other fragment in that direction is definitely in the shadow.

In the next step we render our scene as normal based on our camera. Before deciding the final color of a fragment we calculate its distance from the light source and compare it with the distance stored in the shadow buffer. If the distance calculated is greater than the distance stored then the fragment is in the shadow and should be treated appropriately.

The approach presented here is a simple one and the light is treated as directional light. This means that we treat all light rays as parallel as the light rays from a very distant light source like the sun and not as rays from a small source nearby where the rays are radial.

The added complexity required for accurate shadow calculations for point light and especially when there are more than one light sources belongs in an OpenGL focused book.

Now let us see the details of shadow mapping. As we said first we render our scene in an off-screen buffer. OpenGL provides the type of buffer we need for this purpose. Actually we create a `GL_FRAMEBUFFER` like we used in the previous section. This time the texture we attach to it has the `GL_DEPTH_ATTACHMENT` attribute. This instructs OpenGL that we will store distances and not colors. Our framework has a class called *cg_depth_buffer* which takes care of the trivial tasks. For our needs a 1024x1024 pixel buffer is enough. Here is how we create the buffer:

```
unsigned int cg_depth_buffer::create(int w, int h)
{
    // configure depth map FBO
    glGenFramebuffers(1, &m_buffer);
    glBindFramebuffer(GL_FRAMEBUFFER, m_buffer);

    // create depth texture
    glGenTextures(1, &m_texture);
    glBindTexture(GL_TEXTURE_2D, m_texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, w, h, 0,
                 GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    // attach depth texture as FBO's depth buffer
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
                           m_texture, 0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
        m_buffer = 0;
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    return m_buffer;
}
```

```
virtual void frame_render()
{
    // create the depth buffer from the light point of view
    mat4 l_projection(ortho(-10, 10, -10, 10, 0.1f, 70.f));
    mat4 l_view(lookAt(m_light->get_position(),vec3(0.0f),vec3(0.0,1.0,0.0)));
    mat4 lightSpaceMatrix(l_projection * l_view);
    // render scene from light's point of view
    depth_shader->use();
    // the projection matrix in light space
    depth_shader->set_mat4("lightSpaceMatrix", lightSpaceMatrix);
    glViewport(0, 0, 1024, 1024);
    d_buffer->bind();
    glClear(GL_DEPTH_BUFFER_BIT);
    glCullFace(GL_FRONT);
    // render our scene
    render_scene(depth_shader);
    glCullFace(GL_BACK);
    d_buffer->release();

    // ---------------------------------------------------------
    // set the viewport to the whole window
    m_view->set_viewport();
    glClearColor(0.2f, 0.2f, 0.2f, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    m_shader->use();
    m_light->apply(m_shader);

    mat4 projection = m_view->perspective();
    mat4 view = m_cam->perspective();
    m_shader->set_mat4("projection", projection);
    m_shader->set_mat4("view", view);
    // the projection matrix in light space
    m_shader->set_mat4("lightSpaceMatrix", lightSpaceMatrix);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, d_buffer->texture());
    m_shader->set_int("shadowMap", 0);
    // render our scene
    render_scene(m_shader);

    // the following will be analyzed in the next chapter
    m_shader->set_vec4("objectColor", vec4(.9f, .1f, .1f, 0.5f));
    font2D->set_position(5, 5);
    font2D->render(m_shader, "press Esc to exit");

    glUseProgram(0);
}
```

The 2D texture attribute we use means that this image will be used as a “texture” at the second stage of our rendering operation. Our rendering function looks like this:

The first part of the function renders the scene from the light point of view almost like rendering on the screen. Then in the second part we bind the texture we created and render our scene from the player’s point of view.

Now we are going to take a look at the shader code used. We start with the shader we use for the depth buffer. Here we only need the position of the point which is calculated in the vertex shader and stored in the *gl_Position* predefined variable. The fragment shader can be empty or omitted completely.

```
#version 330 core
layout(location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}
```

This builds our depth buffer, and we proceed to our normal rendering where all the magic happens. The vertex shader is calculates the final position if the vertex and stores it in *gl_Position* as usual, but it also calculates the vertex position in the light space as did the depth vertex shader. This light space position will be used to check if the final fragment is in the shadow.

The fragment shader introduces a new function we call *shadow_calculation* which reads the depth buffer to determine whether the fragment is in the shadow or not.

```
#version 330 core

in vec3 pos;
in vec3 Normal;
in vec2 TexCoords;
in vec4 posLightSpace;

struct lightsource {
    int type;
    vec3 pos_or_dir;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform lightsource light;
uniform sampler2D shadowMap;
uniform vec3 cameraPos;
uniform vec3 objectColor; // object color

out vec4 color;
```

```glsl
float shadow_calculation()
{
    // perform perspective divide
    vec3 projCoords = posLightSpace.xyz / posLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.499;
    // get closest depth value from light's perspective
    // (using [0,1] range fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // check whether current frag pos is in shadow
    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;

    return shadow;
}

void main()
{
    vec3 normal = normalize(Normal);

    vec3 ambient = light.ambient;

    vec3 lightDir = normalize(light.pos_or_dir);// -pos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * light.diffuse;

    vec3 viewDir = normalize(cameraPos - pos);
    vec3 reflectDir = reflect(-light.pos_or_dir, normal);
    vec3 halfwayDir = normalize(light.pos_or_dir + viewDir);
    float spec = pow(max(dot(normal, halfwayDir), 0.0), 128.0);
    vec3 specular = spec * light.specular;

    float shadow = shadow_calculation();

    vec3 result = (ambient + (1.0 - shadow)*(diffuse + specular))*objectColor;

    color = vec4(result, 1);
}
```

You may have noticed that during the creation of the shadow map we called the *glCullFace* function and inverted which is the face to render. We used this trick to overcome an artifact called *shadow acne*. This is because the shadow map is made up of samples and the actual surface is continuous. This leads to spots where the shadow function fails, and we end up with black stripes. This trick makes the shadow map more continuous and eliminates the problem.

Chapter 7: Text

In the previous chapter we saw the need for text display in our games. From the simplest task to display some basic information to the user, to the display of a series of menu options for the user to select.

Displaying text from a simple program is easy. In C we have the *printf* function which prints text to the screen. As we saw in the first part of the book in C++ the equivalent id the *std::out* stream.

In complex environments though like OpenGL things are not so easy. Here we have to create the objects that represent the characters we want to display.

The simplest way to display text is to create a texture image on which to draw all the characters of a font, and then use it to render quads with those characters. This technique relies on the things we have learned before about drawing with textures.

Here I am going to present a simpler technique that relies on the Windows system and generates a drawing list based on any system font. This drawing list can be used to render any text on the screen.

In the second part of this chapter we will see how we can use the Windows system again to create a font with solid characters that can be used in any scene.

2D text rendering

I gave a brief description how to render 2D text on the screen using a texture. The technique we are going to see here is very similar to that, only this time, most of the job is done by the system. It is Windows and OpenGL that do all the dirty work of creating the texture and generating the render commands required to draw each character.

First we generate a storage for the drawing commands for each character. This is the *list* that we generate which can store 256 characters. Then we ask Windows to generate a font as if we were going to draw some text on the screen. Finally we instruct the Windows implementation of OpenGL to create the font textures and store the appropriate drawing commands in the character list we created.

```
HDC hDC = wglGetCurrentDC();
listBase = glGenLists(256);

if (strcmp(name, "symbol") == 0)
{
    hFont = CreateFontA(-size, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
        SYMBOL_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
        ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH, name);
}
else
{
    hFont = CreateFontA(-size, 0, 0, 0, FW_NORMAL, FALSE, FALSE, FALSE,
        ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
        ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH, name);
}

SelectObject(hDC, hFont);
wglUseFontBitmaps(hDC, 0, 256, listBase);
```

Now that we have everything set up, we can use the font to render our text on the screen.

```
glColor4f(color2d.x, color2d.y, color2d.z, color2d.w);
glWindowPos2i(screenPos.x, screenPos.y);
glPushAttrib(GL_LIST_BIT);
glListBase(listBase);
glCallLists((GLsizei)strlen(text), GL_UNSIGNED_BYTE, text);
glPopAttrib();
```

Our font class has does not need a shader because it is only used to set the rendering color. OpenGL has several functions to do it. We set all four components of the color using *glColor4f* function which takes four floating point values, one for each color component. Then we do some housekeeping, by storing the current state of OpenGL before we activate the bitmap font storage and call it to render our text. Then we restore the state of OpenGL, and we return from the rendering function.

3D text rendering

Our next tool to display text is fancier. It creates three dimensional objects from the character description in the system font. This way we have real 3D objects that can be incorporated in our scenes like any other object we have seen so far.

The creation process of the 3D font is almost the same as the 2D font we saw earlier. The main difference when we generate 3D fonts is the function *wglUseFontOutlines* we call to generate the 3d objects. This function has two arguments that we need to pay some attention to.

The first is the *depth* of the character. These characters are created by extrusion, so we need to provide the depth. The second argument is the buffer that receives the dimensions of each character. We will need these dimensions when drawing the text.

```
HDC hDC = wglGetCurrentDC();
listBase = glGenLists(256);      // create storage for 96 characters

if (strcmp(name, "symbol") == 0)
{
    hFont = CreateFontA(-size, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
        SYMBOL_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
        ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH, name);
}
else
{
    hFont = CreateFontA(size, 0, 0, 0, FW_NORMAL, FALSE, FALSE, FALSE,
        ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
        ANTIALIASED_QUALITY, FF_DONTCARE | DEFAULT_PITCH, name);
}

if (!hFont)
    return;

SelectObject(hDC, hFont);
wglUseFontOutlines(hDC, 0, 255, listBase, (float)0.0f, (float)depth, WGL_FONT_POLYGONS, gmf);
```

Since this font creates 3D objects we do not need a custom shader. Instead we use the normal shader of our scene.

Our font class allows us to align the text at the bottom-left, bottom-center, or bottom-right point. This requires us to calculate the length of the text prior to rendering in order to calculate how to place the text in our scene.

There is one thing we need to address when we render our text. Every character is a different object and has to be positioned in the scene. So we define a different *model* matrix for each one changing the translation matrix based on the alignment and stepping along as we move to the next character.

```
void cg_font::render(cg_shader* shader, int alignment, const char* str, ...)
{
    float length = 0;
    char text[512];
    va_list args;

    if ((str == NULL))
        return;

    va_start(args, str);
    vsprintf(text, str, args);
    va_end(args);

    // center the text
    // find length of text
    for (unsigned int loop = 0; loop < (strlen(text)); loop++)
    {
        // increase length by character's width
        length += gmf[text[loop]].gmfCellIncX;
    }
    vec3 offset(0, 0, 0);
    switch (alignment)
    {
    case ALIGN_CENTER:
        offset.x = -length / 2;
        break;
    case ALIGN_LEFT:
        break;
    case ALIGN_RIGHT:
        offset.x = -length;
        break;
    }

    // draw the text
    glPushAttrib(GL_LIST_BIT);
    glListBase(listBase);
    for (auto i = 0; i < strlen(text); ++i)
    {
        mat4 ttm = tmat;
        translate_matrix(ttm, offset);
        // putting translation at the end treats the text as a unit
        mat4 ob_matrix = rmat * smat * ttm;
        shader->set_mat4("model", ob_matrix);
        glCallLists(1, GL_UNSIGNED_BYTE, &text[i]);
        offset.x += gmf[text[i]].gmfCellIncX;
    }
    glPopAttrib();
}
```