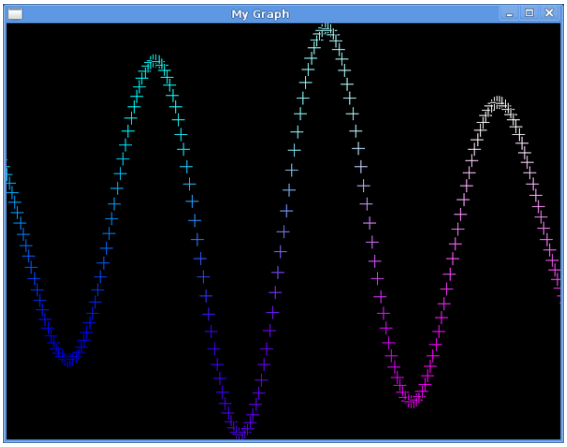
ogl-wikibooks

Contents

Chapter 1

OpenGL Programming/Scientific OpenGL Tutorial 01



Our first graph

Chapter 2

Introduction

Although OpenGL is widely known for its use in games, it has also many other applications. One of these is the visualization of scientific data. Technically, there is not a great difference between drawing datasets and drawing game graphics, but the emphasis is different. Instead of a perspective view of our data, the scientist usually wants an orthographic view. Instead of specular highlights, reflections and shadow, scientific data is usually presented with primary colors and just a bit of smooth shading. It may sound like only simple OpenGL features are used, but in return a scientist wants the data rendered with a high accuracy, without any artifacts, and without arbitrary clipping of geometry or lighting. Also, raw data might need a lot of transformation before it can be rendered, and these transformations cannot always be implemented as matrix multiplications. Before the advent of the programmable shaders, scientific visualisation was lot harder to do on graphics cards.

In the following tutorial, we will assume you have already read up to tutorial 06.

Chapter 3

Plotting a function in 2D

A basic scientific visualisation task is to make a graph of a function or some data points. We will start by plotting the following function:

$y = \frac{\sin(10x)}{1+x^2}$

Which looks like waves which have an amplitude of 1 near the origin and decay as you go away from the origin. If you have gnuplot installed, then you can easily plot this function with these commands:

```
f(x) = sin(10 * x) / (1 + x * x) plot f(x)
```

Just like gnuplot does, we first need to evaluate the function in a number of points, then we can draw lines through those points. We will evaluate the function at 2000 points in the range $-10 < x < 10$.

Since our function does not change over time, it would be nice if we could send the points that we calculated to the GPU only once. To do that, we will store the points in a Vertex Buffer Object. This allows us to hand over ownership of the data to the GPU, which can then store a copy in its own memory for example.

We also want to zoom in and out, and move around, to explore the function in greater detail. To do that, we will have a scale and offset variable. The vertex shader will use these to transform our "raw" data points to screen coordinates.

Just drawing a line through some points is not the only way to plot a function. It is also possible to apply color to the line, depending on the original x and y coordinates. Or one can draw different shapes. Compare for example the results of the following gnuplot commands:

```
plot f(x) with lines plot f(x) with dots plot f(x) with points
```

The first two forms are easily implemented by drawing the vertices using GL_LINES and GL_POINTS. The last form draws + signs at each point. It so happens that OpenGL has a function called "point sprites", which basically allows you to draw the contents of a texture on a square centered around the vertices, which makes it very easy to copy gnuplot's with points drawing style.

Chapter 4

Vertex buffer objects

Vertex buffer objects (VBOs) are just buffer objects that hold vertex data. It is very similar to using vertex arrays, with some exceptions. First, OpenGL will allocate and deallocate the storage space for us. Second, we must explicitly tell OpenGL when we want access to the VBO. The idea is that when we don't want to access the VBO ourself, the GPU can have exclusive access to the contents of the VBO, and can even store the contents in its own memory, so it doesn't need to fetch the data from the slow main memory every time it needs the vertices.

First, we create our own array of 2000 2D data points and fill it in:

struct point { GLfloat x; GLfloat y; }; point graph[2000]; for(int i = 0; i < 2000; i++) { float x = (i - 1000.0) / 100.0; graph[i].x = x; graph[i].y = sin(x * 10.0) / (1.0 + x * x); }

Then we create a new buffer object:

GLuint vbo; glGenBuffers(1, &vbo); glBindBuffer(GL_ARRAY_BUFFER, vbo);

The glGenBuffers() and glBindBuffer() functions work just like those for other objects in OpenGL. And, also similar to how we allocate and upload a texture with glTexImage*(), we upload our graph with the following command:

glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph, GL_STATIC_DRAW);

GL_STATIC_DRAW indicates that we will not write to this buffer often, and that the GPU should keep a copy of it in its own memory. It is always possible to write new values to the VBO. If the data changes once per frame or more often, you should use GL_DYNAMIC_DRAW or GL_STREAM_DRAW. When overwriting data in an existing VBO, you should use the glBufferSubData() function, which is the analogue of glTexSubImage*().

Suppose we have already set up everything else, and we are ready to draw a line through these points. Then we just need to do the following:

glBindBuffer(GL_ARRAY_BUFFER, vbo); glEnableVertexAttribArray(attribute_coord2d); glVertexAttribPointer(attribute_coord2d, // attribute 2, // number of elements per vertex, here (x,y) GL_FLOAT, // the type of each element GL_FALSE, // take our values as-is 0, // no space between values 0 // use the vertex buffer object); glDrawArrays(GL_LINE_STRIP, 0, 2000); glDisableVertexAttribArray(attribute_coord2d); glBindBuffer(GL_ARRAY_BUFFER, 0);

The first line tells us to use the VBO with our graph in it. Then we tell OpenGL that we are giving it an array of vertices. The last parameter of glVertexAttribPointer() used to be a pointer to the vertex array. However, we set this to 0 to tell OpenGL that it should use data from the currently bound buffer object instead. So we *do not* need to map our buffer object to a pointer here! Doing so would destroy all the performance benefits of VBOs. Then, we draw using the usual glDraw commands. The GL_LINE_STRIP mode tells OpenGL to draw line segments between consecutive vertices, such that there is a continous line that goes through all the vertices. Afterwards, we can tell OpenGL we no longer want to use vertex arrays and our buffer object.

Exercises (do them after you implemented the shaders mentioned below):

- Try drawing using GL_LINES, GL_LINE_LOOP, GL_POINTS or GL_TRIANGLE_STRIP instead.
- Try to draw only the subset of the points that are visible by changing the parameters of glDrawArrays().
- Try to draw only the even numbered points by modifying the parameters of glVertexAttribPointer().
- Try changing part of the graph using glBufferSubData().

Chapter 5

Mapping a buffer to memory

There is an alternative way to access the data that is in a VBO. Instead of telling OpenGL to copy data from our own memory to the graphics card, we can ask OpenGL to map the VBO into main memory. Depending on the graphics card, this might avoid the need to perform a copy, so it could be faster. On the other hand, the mapping itself could be expensive, or it might not really map anything but perform a copy anyway. That said, this is how it works:

glBindBuffer(GL_ARRAY_BUFFER, vbo); glBufferData(GL_ARRAY_BUFFER, 2000 * sizeof(point), NULL, GL_STATIC_DRAW); point *graph = (point *)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY); for(int i = 0; i < 2000; i++) { float x = (i - 1000.0) / 100.0; graph[i].x = x; graph[i].y = sin(x * 10.0) / (1.0 + x * x); } glUnmapBuffer(GL_ARRAY_BUFFER);

After binding to our VBO, we call glBufferData() like before, except that we pass a NULL pointer. This will tell OpenGL to allocate the memory for our 2000 data points. Then we "map" the buffer into main memory using the glMapBuffer() function. We indicate that we only write to the memory with GL_WRITE_ONLY. This tells the GPU it never has to copy GPU memory back to main memory, which might be expensive on some architectures. After we have a pointer to the buffer, we can write to it as usual. The final command tells OpenGL we are done with it. This unmaps the buffer from main memory (or it might cause our array to be uploaded to the GPU, for example). From then on, we cannot not use the graph pointer anymore.

Strictly, glMapBuffer() is not part of the core OpenGL ES 2.0 language, so you should not rely on it being always available.

- Try to find out which method of changing the contents of a VBO is faster on your system: glBufferData(), glBufferSubData() or glMapBuffer().

Chapter 6

Shaders

As mentioned before, our shaders will be very simple. Let's begin with the vertex shader:

attribute vec2 coord2d; varying vec4 f_color; uniform float offset_x; uniform float scale_x; void main(void) { gl_Position = vec4((coord2d.x + offset_x) * scale_x, coord2d.y, 0, 1); f_color = vec4(coord2d.xy / 2.0 + 0.5, 1, 1); }

As you can see, we perform very little transformations on our coordinates. Remember that by default, the OpenGL coordinate (1,1) correspond to the upper right corner of the window, and $(-1,-1)$ the lower left corner. Our x values go from -10 to 10, and our y values from -1 to 1. If we would not apply any transformation, we would only see the $-1 < x < 1$ part of our graph. So, we introduce two uniform variables that allow us to zoom in and out and move around: offset_x and scale_x. We add offset_x to the x coordinate, and the multiply the result by scale_x.

- What happens if you multiply first and then add the offset? Which is more efficient? What would you have to change in the C++ code to get the same behavior as before?
- In principle a MVP matrix would also allow us to move and zoom. However, try to change the vertex shader so it draws a logarithmic plot instead.

We also have a varying f_color which we can use to assign a color to each point, depending on the original coordinates. Although it is just for show here, it can be used to add more information to the plot. The fragment shader is very simple:

varying vec4 f_color; void main(void) { gl_FragColor = f_color; }

Chapter 7

Keyboard interaction

Now that we have the uniforms offset_x and scale_x, we want some way of controlling them. In a more sophisticated program, one would use a toolkit like Qt or Gtk, and use scrollbars or mouse controls to zoom and move around. In GLUT, we can very easily implement a keyboard handler that lets us interact with the program. Assume we have the following global variables:

GLint uniform_offset_x; GLint uniform_scale_x; float offset_x = 0.0; float scale_x = 1.0;

By now, you should know how to get references to the uniform variables in the shaders, and how to set their values in the display() function. So, we will just look at our keyboard handling function instead:

void special(int key, int x, int y) { switch(key) { case GLUT_KEY_LEFT: offset_x -= 0.1; break; case GLUT_KEY_RIGHT: offset_x += 0.1; break; case GLUT_KEY_UP: scale_x *= 1.5; break; case GLUT_KEY_DOWN: scale_x /= 1.5; break; case GLUT_KEY_HOME: offset_x = 0.0; scale_x = 1.0; break; } glutPostRedisplay(); }

It is called special() because GLUT makes a distinction between the “normal” alphanumeric keys and “special” keys like function keys, cursor keys, and so on. To tell GLUT to call our function whenever a special key is being pressed, we have the following in main():

if (init_resources()) { glutDisplayFunc(display); glutSpecialFunc(special); glutMainLoop(); }

- Experiment with the cursor keys. Try holding buttons for a long time.
- What do you think the x and y parameters are?
- Make it so you can switch between drawing GL_LINE_STRIP and GL_POINTS with the F1 and F2 keys.

Chapter 8

Point sprites

When plotting measurement data instead of mathematical functions, scientists usually draw little symbols at the data points, like crosses and squares. We can do this by drawing those symbols with GL_LINES for example, but we could also have those symbols as textures, and draw those on little squares centered on the data points. You should know how to do this from tutorial 06. However, instead of having to draw a quad or two triangles ourself, we can let OpenGL handle this for us using the point sprite functionality, which will let us reuse our vertex buffer without any changes.

You should know how to load and enable textures from the aforementioned tutorial. You should make a mostly transparent texture with a small opaque symbol drawn on it, like a +. To properly draw transparent textures, and to enable point sprites, we call these functions:

glEnable(GL_BLEND); glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); glEnable(GL_POINT_SPRITE); glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);

The last two commands enables point sprite functionality and the ability to control the point size from within the vertex shader. For OpenGL ES 2.0, these commands should not be necessary, as this functionality is always enabled. However, it may be required for your graphics card to function correctly (although some do have issues with vertex shader point size control).

Contrary to what you would use in most games, we want to *disable* interpolation for the texture, otherwise our symbols will look fuzzy and unsharp, which is not desirable in a plot. (You can compare this to the "hinting" of fonts.)

glBindTexture(GL_TEXTURE_2D, texture_id); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

To draw the points in our VBO with point sprites, we call:

glUniform1f(uniform_point_size, res_texture.width); glDrawArrays(GL_POINTS, 0, 2000);

The first line passes the desired point size (equal to the width of the texture in our case) to the vertex shader in a uniform. We should change the vertex shader to:

attribute vec2 coord2d; varying vec4 f_color; uniform float offset_x; uniform float scale_x; uniform float point_size; void main(void) { gl_Position = vec4((coord2d.x + offset_x) * scale_x, coord2d.y, 0, 1); f_color = vec4(coord2d.xy / 2.0 + 0.5, 1, 1); gl_PointSize = point_size; }

Notice that we draw just by using GL_POINTS. If we run the program like this, you will not see your point sprites, but just some colored squares! What remains is to change our fragment shader to actually draw the texture instead of a solid color:

#version 120 uniform sampler2D mytexture; varying vec4 f_color; void main(void) { gl_FragColor = texture2D(mytexture, gl_PointCoord) * f_color; }

This does not look too different from a regular texture shader. However, we are now using the gl_PointCoord variable in the fragment shader. It will run from (0,0) in the top left of the square to (1,1) in the bottom right, exactly what we need to get the right texture coordinates. This functionality is only available in GLSL version 1.20 and later, therefore we should put #version 120 at the top of the shader source code.

- Try changing the texture filters to GL_LINEAR. Research how exactly point sprites are drawn.
- Try to change the point size in the C++ program. Try really small and really big sizes.
- Research how you can change the point size using glPointSize() in the C++ program instead of using the vertex shader (this is not OpenGL ES 2.0 compatible though).
- Try to rotate the point sprites 45 degrees by changing the fragment shader.
- Try to draw circular point sprites.

- Make it so you can switch between drawing GL_LINE_STRIP, normal GL_POINTS and point sprites by pressing F1, F2 and F3.

- Comment on this page

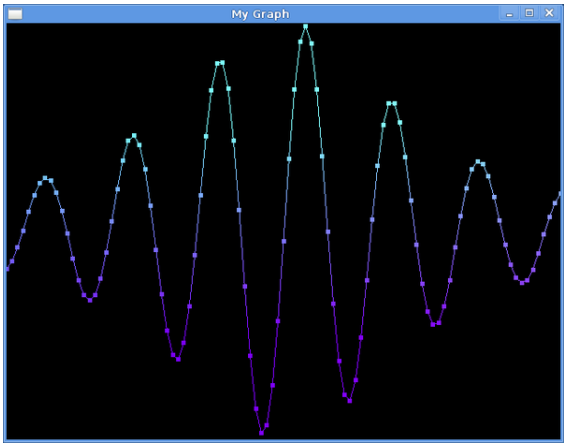
- Recent stats

< OpenGL Programming

Browse & download complete code

Chapter 9

OpenGL Programming/Scientific OpenGL Tutorial 02



Plotting a graph from a texture

Chapter 10

Introduction

In the first graph tutorial, we plotted a function by creating 2-dimensional vertices for all the data points we had. This is straightforward and works very well, if we do not have too much data. In this tutorial, we will approach the problem of plotting data points very differently.

First, we are usually more interested in the y coordinates, the x coordinates are just equally spaced in the domain of interest. We don't want to store the x coordinates in memory if they can easily be recovered programmatically. In fact, when taking data from an ADC (such as from a microphone connected to a sound card), we only get a stream of y coordinates. It would be very nice if we could put that into a buffer without any further processing, and have the graphics card do something useful with it.

Secondly, if we have thousands of data points, it really is useless to plot all of them in a window that may not even be a thousand pixels wide. So it would be nice if we can separate the number of vertices that we draw from the number of data points that we have. We also do not want to change the vertices as we move around or zoom in and out of the graph.

The solution is simple, we put fixed x coordinates in a 1-dimension vertex buffer object (VBO), and put the y coordinates in a 1-dimensional texture, and have the vertex shader combine the two.

NOTE: although core OpenGL ES 2.0 supports texture lookups in the vertex shader, it is allowed that graphics cards have zero texture units available in the vertex shader. It is therefore possible that this technique does not work on your card. To check the number of vertex texture units available, use this code fragment:

int vertex_texture_units; glGetIntegerv(GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS, &vertex_texture_units); if(!vertex_texture_units) { fprintf(stderr, “Your graphics cards does not support texture lookups in the vertex shader!\n”); // exit here or use another method to render the graph }

Chapter 11

1-dimensional VBO

Although vertices are usually two- or three-dimensional, OpenGL has nothing against using one-dimensional vertices. Remember that by default, the x coordinates of the window go from -1 to 1. So we will create a VBO with 101 x coordinates that go from -1 to 1.

glGenBuffers(1, &vbo); glBindBuffer(GL_ARRAY_BUFFER, vbo); GLfloat line[101]; for(int i = 0; i < 101; i++) { line[i] = (i - 50) / 50.0; } glBufferData(GL_ARRAY_BUFFER, sizeof line, line, GL_STATIC_DRAW);

We also rename our vertex attributes to “coord1d":

GLint attribute_coord1d = glGetAttribLocation(program, attribute_name);

Then, we can draw our “line” almost exactly like we would if we had 2D vertices:

glBindBuffer(GL_ARRAY_BUFFER, vbo); glEnableVertexAttribArray(attribute_coord1d); glVertexAttribPointer(attribute_coord1d, // attribute 1, // number of elements per vertex, here just x GL_FLOAT, // the type of each element GL_FALSE, // take our values as-is 0, // no space between values 0 // use the vertex buffer object); glDrawArrays(GL_LINE_STRIP, 0, 101);

In our vertex shader, we have to come up with the y coordinates on our own:

attribute float coord1d; void main(void) { float y = ...; gl_Position = vec4(coord1d, y, 0.0, 1.0); }

Exercise:

- Try various ways of calculating y values in the vertex shader. In fact, you can even let OpenGL evaluate the function we used in the first tutorial!

Chapter 12

Putting y-values in a texture

Depending on the graphics card and drivers that you have, textures can either be very flexible or very restricted. Some cards allow textures in a wide range of formats, including 16 bit integers, floating point or even fixed point formats. If your input data matches a format supported by the card, you don't have to do any conversions and rendering will be very fast. If you try to be OpenGL ES 2.0 compliant however, there is the restriction that it only supports 8 bit integers for texture data. However, might just be enough. Take for example the function we used in the previous tutorial, and map the y coordinates from $-1..1$ to $0..255$:

GLbyte graph[2048]; for(int i = 0; i < 2048; i++) { float x = (i - 1024.0) / 100.0; float y = sin(x * 10.0) / (1.0 + x * x); graph[i] = roundf(y * 128 + 128); }

Now we can create a one-dimensional texture. Again, OpenGL ES has a limitation; it doesn't explicitly support 1 dimensional textures. However, nothing prevents us from making a texture that is very wide but only one pixel high:

glActiveTexture(GL_TEXTURE0); glGenTextures(1, &texture_id); glBindTexture(GL_TEXTURE_2D, texture_id); glTexImage2D(GL_TEXTURE_2D, // target 0, // level, 0 = base, no minimap, GL_LUMINANCE, // internalformat 2048, // width 1, // height 0, // border, always 0 in OpenGL ES GL_LUMINANCE, // format GL_UNSIGNED_BYTE, // type graph);

Here we used the GL_LUMINANCE format to indicate that we only have one color component.

Exercises:

- Try to find out which texture formats your card supports.
- Is there a limit to the size of a one-dimensional texture?
- Try changing part of the graph using glTexSubImage2D().
- OpenGL ES also supports the GL_RGBA format, essentially giving us 32 bits per pixel. Could we use that to get better accuracy of our y values?

Chapter 13

The vertex shader

Now that we have our VBO with x coordinates and our texture with y coordinates, we will combine them in our vertex shader. Remember that texture coordinates go from 0 to 1, while our x coordinates go from -1 to 1. Also, we want to pan and zoom, so we will use the offset_x and scale_x variables from the previous tutorial. In this case however, since we do not change our x coordinates, we need to apply the offset and scale transformations in reverse to get the texture coordinates! Once we have all the coordinates, we can also use it to color the graph similar to how we did that in the previous tutorial. Here is the full vertex shader source:

```
attribute float coord1d; varying vec4 f_color; uniform float offset_x; uniform float scale_x; uniform sampler2D mytexture; void main(void) { float x = (coord1d / scale_x) - offset_x; float y = (texture2D(mytexture, vec2(x / 10.24 / 2.0 + 0.5, 0)).r - 0.5) * 2.0; gl_Position = vec4(coord1d, y, 0.0, 1.0); f_color = vec4(x / 2.0 + 0.5, y / 2.0 + 0.5, 1.0, 1.0); }
```

As you can see, nothing prevents you from using textures in the vertex shader (although on some graphics cards, especially older ones, it might be slower to access them from the vertex shader than from the fragment shader). Since we have a GL_LUMINANCE format texture, we have to read the red component, the other components are undefined. Also note that the texture2D() function returns floating point values in the range 0..1, not integers in the range 0..255. The fragment shader is the same from the previous tutorial, simply setting gl_FragColor to f_color.

Chapter 14

Interpolation and wrapping

If you zoom in very far, you will notice that the lines are not smooth anymore, but look like a staircase. You might think that this is because of the low accuracy of our 8-bit integer y values. However, the height of the steps vary, and in the steepest parts of the function, the height will be much more than can be explained by 8-bit integers. Instead, the problem is caused by the fact that there now are more vertices horizontally than pixels in the texture. Nearest-neighbor interpolation will cause clusters of vertices to all have the same y value. To our smooth curve back, we should enable linear interpolation:

```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER,  GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

If you pan or zoom out very far, you will notice something very interesting: the function is repeating itself! This is because by default, OpenGL will wrap the texture coordinates. We could clip the texture coordinates ourself in the vertex shader, but we can also tell OpenGL to do that automatically:

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

Exercises:

- Make it so you can toggle interpolation and wrapping modes by pressing F1 and F2.
- Make it so that if you press F3, it draws the graph twice, once with GL_LINE_STRIP and once with GL_POINTS, use a point size of 5 pixels.
- The MIN_FILTER does not seem to be doing very much. Research how GL_LINEAR works for both MIN_FILTER and MAG_FILTER.
- Would mipmaps be useful?
- Think again about using GL_RGBA to get 32 bit accuracy.

- Comment on this page

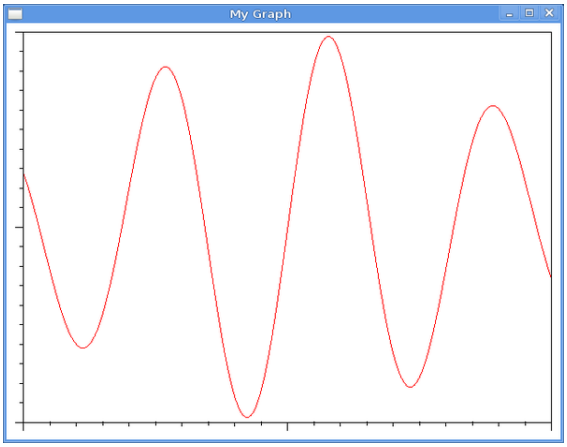
- Recent stats

< OpenGL Programming

Browse & download complete code

Chapter 15

OpenGL Programming/Scientific OpenGL Tutorial 03



Borders and axes

Chapter 16

Introduction

The previous two tutorials focused on plotting a curve, but if you use any serious plotting tool, you will notice that graphs come with titles, axes, tick marks, grid lines, legends and (most important, but often forgotten) axis labels. Most of the work of drawing a graph is actually drawing everything around it.

The biggest problem you will encounter is the fact that you have your graph coordinates on one hand, which we can relatively easily transform to position the graph correctly on the screen, and pixel coordinates on the other hand, which you want to use for everything around the graph. Right around the border of the graph, where you want to draw tick marks and coordinates, the two coordinate spaces will meet.

In this tutorial we will see how we can draw our plot in a somewhat smaller rectangle than the window, without the curve leaking out. We will also see how to correctly draw tick marks on the left and bottom of the graph, such that they match up with the curve. The result will start to resemble the output of gnuplot.

Chapter 17

Shaders

This time we will use very simple but generic shaders. The vertex shader will just apply a transformation matrix to 2D vertices:

attribute vec2 coord2d; uniform mat4 transform; void main(void) { gl_Position = transform * vec4(coord2d.xy, 0, 1); }

We will use fixed, solid colors, passed directly to the fragment shader in a uniform:

uniform vec4 color; void main(void) { gl_FragColor = color; }

Chapter 18

Viewports and scissors

Let's draw exactly the same graph as in the first graph tutorial, using a vertex buffer object to store our data, and having the variables offset_x and scale_x that we can change using the keyboard. To make it look more professional, we can plot it in red on a white background.

Remember that we now have a vertex shader that wants a transformation matrix instead of using offset_x and scale_x itself. Using GLM, we can create the matrix from those variables ourself, simply by applying a scale and a translate operation on the identity matrix. We can then send it to the vertex shader:

GLint uniform_transform = glGetUniformLocation(program, "transform"); glm::mat4 transform = glm::translate(glm::scale(glm::mat4(1.0f), glm::vec3(scale_x, 1, 1)), glm::vec3(offset_x, 0, 0)); glUniformMatrix4fv(uniform_transform, 1, GL_FALSE, glm::value_ptr(transform));

If we would now draw our graph, it should look the same as in the first tutorial (except that it is red on white). The graph still covers the entire screen. Instead, we want to scale down our graph a little bit, and make it so there is some space around it for tick marks and perhaps other things. Let's reserve some space for tick marks on the left and bottom side of the graph, and also have a margin around everything. We want the space to be independent of the size of the window, or put otherwise, it should be a fixed amount of pixels. Let's define that now:

const int margin = 20; const int ticksize = 10;

Of course, we can easily scale and translate our graph now that we have a transformation matrix. But however we change the matrix, that will not prevent it from being drawn across the whole screen. We could manually determine which vertices are inside the area designated for our plot and draw only those, but that would be a whole lot of work. We could also draw the plot first, and then clear the area around it by drawing filled rectangles. But all that is silly, we just want to tell the GPU to clip the plot for us.

There are several OpenGL methods we can use to clip our plot. We start with the glViewport() function. This defines the area inside the window, in pixels, in which to draw. We can use glutGet() calls to find out the actual size of our window, and calculate the exact area like this:

int window_width = glutGet(GLUT_WINDOW_WIDTH); int window_height = glutGet(GLUT_WINDOW_HEIGHT); glViewport(margin + ticksize, margin + ticksize, window_width - margin * 2 - ticksize, window_height - margin * 2 - ticksize);

The first two parameters to glViewport() are the x and y offsets in pixels from the lower left of the window. The second two are the width and height of or viewport in pixels. Try adding this at the top of the display() function. You should see that there now is indeed a clear margin around the plot. If you increase the margin or ticksize constants, or try resizing the window, you will notice that the viewport not only clips, but also rescales the plot, so that everything that fit in to the whole window before will now fit exactly in the viewport area. For our purposes, that is fine, because we then don't have to come up with our own transformations to compensate for the margins around the plot.

At this point, you may think that glViewport() really clips all the pixel outside the specified area. However, that is not exactly true. What happens is that the *geometry* gets clipped, so that all the vertices that will be drawn lie within the viewport area. There is no guarantee that fragments will get clipped, although some cards (nVidia for example) may automatically do that too. You can try to make the lines very thick using the glLineWidth() function, and depending on your video card, you can see that the center of the line stops at the edge of the viewport area, but due to its thickness pixels can leak outside of it. (It might help to see where the viewport ends if you draw the box mentioned in the next section.) To ensure all fragments also get clipped, you have to set the scissor area and enable the scissor test right after you set the viewport:

glScissor(margin + ticksize, margin + ticksize, window_width - margin * 2 - ticksize, window_height - margin * 2 - ticksize); glEnable(GL_SCISSOR_TEST);

Exercises:

- Is there any advantage of glViewport() over glScissor(), except the extra coordinate transformation the former does?
- Try moving the call to glClear() to between glViewport() and glScissor(). Does it make a difference? What happens if you call it right after both?
- Try drawing a few GL_POINTS with a very big point size, some right inside and some right outside the viewport area.
- Try drawing those GL_POINTS again right inside and right outside the window, without ever calling glViewport(). Can you think of a way to "fix" this behavior?

Chapter 19

Drawing a box around the graph

The next step is to draw a box with tick marks around the plot. This time we don't want any clipping to happen, so we should reset the viewport and disable the scissors to cover the whole window again:

glViewport(0, 0, window_width, window_height); glDisable(GL_SCISSOR_TEST);

The problem now is that we lose the automatic coordinate transformation of before, so we can no longer draw a box with corners $(-1, -1)$ and $(1, 1)$. Unfortunately, there is no easy function to get the same transformation as glViewport() applies, so we will write our own:

glm::mat4 viewport_transform(float x, float y, float width, float height) { // Calculate how to translate the x and y coordinates: float offset_x = (2.0 * x + (width - window_width)) / window_width; float offset_y = (2.0 * y + (height - window_height)) / window_height; // Calculate how to rescale the x and y coordinates: float scale_x = width / window_width; float scale_y = height / window_height; return glm::scale(glm::translate(glm::mat4(1), glm::vec3(offset_x, offset_y, 0)), glm::vec3(scale_x, scale_y, 1)); }

To understand this function, just imagine that you have to shift the center of the window to the center of the new viewport, and that you have to scale down from the width of the window to the width of the viewport. We can now call this function with the same parameters as we gave glViewport(), and give the result to the vertex shader:

transform = viewport_transform(margin + ticksize, margin + ticksize, window_width - margin * 2 - ticksize, window_height - margin * 2 - ticksize,); glUniformMatrix4fv(uniform_transform, 1, GL_FALSE, glm::value_ptr(transform));

Then we draw our box, in black:

GLuint box_vbo; glGenBuffers(1, &box_vbo); glBindBuffer(GL_ARRAY_BUFFER, box_vbo); static const point box[4] = {{−1, −1}, {1, −1}, {1, 1}, {−1, 1}}; glBufferData(GL_ARRAY_BUFFER, sizeof box, box, GL_STATIC_DRAW); GLfloat black[4] = {0, 0, 0, 1}; glUniform4fv(uniform_color, 1, black); glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0, 0); glDrawArrays(GL_LINE_LOOP, 0, 4);

Exercises:

- Could we have drawn the box before the graph? Or maybe right after it while still using the same glViewport()?

Chapter 20

Drawing y tick marks

Now that we have plotted the curve and drawn the box around it, it is time to draw the tick marks. These little lines are usually placed at integer values, or very round subdivisions thereof, and make it easier to estimate the value of function at a specific point. You can also think of a ruler, with the major ticks indicating centimeters, and minor ticks indicating millimeters.

We will start with the ticks on the left side of the box, also known as the y axis. Since our plot has a fixed y range of $-1..1$, it is going to be easy to figure out the right coordinates. We will try to draw 21 tick marks from -1 to 1, with a spacing of 0.1.

At this point, it is easiest if we keep using the same transformation matrix as we used to draw the box. That way, $x = -1$ corresponds exactly to the left edge of the box, and $y = -1$ and $y = 1$ to the bottom and top of it. But how do we start from there and draw lines that are exactly ticksize *pixels* long? We need to convert between our graph coordinates and pixel coordinates. Most importantly, we need to know how big one pixel is in graph units. Remember that the coordinates we used to draw the box range from -1 to 1 (so it is 2 units wide), but we set our viewport to be window_width - border * 2 - ticksize wide. We can use the same reasoning for the height. So our pixel scaling factors will be:

float pixel_x = 2.0 / (window_width - border * 2 - ticksize); float pixel_y = 2.0 / (window_height - border * 2 - ticksize);

Now that we know that, we can calculate the coordinates of the 42 vertices we need to draw 21 tick marks, and put those in a VBO:

GLuint ticks_vbo; glGenBuffers(1, &ticks_vbo); glBindBuffer(GL_ARRAY_BUFFER, ticks_vbo); point ticks[42]; for(int i = 0; i <= 20; i++) { float y = −1 + i * 0.1; ticks[i * 2].x = −1; ticks[i * 2].y = y; ticks[i * 2 + 1].x = −1 - ticksize * pixel_x; ticks[i * 2 + 1].y = y; } glBufferData(GL_ARRAY_BUFFER, sizeof ticks, ticks, GL_STREAM_DRAW); glVertexAttrib-Pointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0, 0); glDrawArrays(GL_LINES, 0, 42);

Notice the use GL_STREAM_DRAW here. Although the y ticks are always the same in this tutorial, they would be variable in a real plotting program. We will also reuse this VBO for the x ticks later. GL_STREAM_DRAW indicates that we will draw with these vertices only once.

We can further distinguish between major tickmarks (at unit values) and minor tickmarks (every 0.1 units), by replacing every second x coordinate with:

float tickscale = (i % 10) ? 0.5 : 1; ticks[i * 2 + 1].x = −1 - ticksize * tickscale * pixel_x;

Exercises:

- Try to put the tick marks on the other border, or inside the graph instead of outside.
- Instead of drawing tick marks, draw horizontal grid lines in a light grey color. Can you think of a few ways to make the lines appear under the curve?
- The major ticks are now drawn every 1 unit. Change that to be every 2.54 units, and minor ticks every 0.254 units.

Chapter 21

Drawing x tick marks

The y tick marks were easy because we never scale or translate our graph in the y axis. We know exactly where to start and end. But with the x axis, we have two difficulties. First, when we translate our graph, our tick marks should move along with the graph. But as we shift the graph to the left, tick marks should disappear at the left edge of the box, and new ones should appear at the right edge. Second, if we change the scale of the graph, the space between the tick marks should be adjusted accordingly. But if we zoom out a lot, then we don't want to have thousands of tick marks on the bottom. Instead, we want them to be decimated each time more than, say, 20 tick marks would be visible. Similarly, if we zoom in a lot, the density in the tick marks should be increased by a factor of 10 each time less than 2 tick marks would be visible.

The desired spacing of the tick marks can be found out relatively easily. Basically, we know the scale of the graph from the scale_x variable. We want to scale the spacing between our tick marks with it, but every time scale_x crosses a power of 10, "reset" it back to 1. We can do that by taking the base 10 logarithm of scale_x, rounding that down to an integer, and then raise 10 to the power of that integer to get a logarithmically rounded scaling factor (0.1, 1, 10, 100, et cetera). In graph units, the desired space between minor tick marks is:

```
float tickspacing = 0.1 * powf(10, -floor(log10(scale_x)));
```

To find out where the left- and right most ticks should be drawn, we will first figure out what the graph coordinates are of the left- and right most visible part of the graph. We know the x coordinates are -1 and 1 in the coordinate system we have drawn the box with, so we have to apply the inverse of the transformation matrix we used to draw the graph. Since we are only interested in the x coordinate, and the transformation is fairly simple, we can do that by hand instead of using GLM:

```
float left = -1.0 / scale_x - offset_x; float right = 1.0 / scale_x - offset_x;
```

There is no guarantee that these coordinates coincide with tick marks however. We do know that there is at least a tick mark at the origin, and we know the space between them. Let's number the tick marks, starting with 0 at the origin. Then we can determine the numbers of the two tick mark closest to, but still between, the left and right edges:

```
int left_i = ceil(left / tickspacing); int right_i = floor(right / tickspacing);
```

We then know that the coordinate of the left most tick mark, in graph coordinates, is simply left_i * tickspacing. The difference between the left border and the left most tick mark, *in graph units*, is then as follows:

```
float rem = left_i * tickspacing - left;
```

Now we can calculate the coordinate of the left most tick mark in the coordinate system we are going to draw with:

```
float firsttick = -1.0 + rem * scale_x;
```

We can also easily calculate what the distance between tick marks is in drawing coordinates, and we know how many tick marks to draw simply by looking at the left_i and right_i variables. If we did everything right, that should never be more than 21 ticks, however it is always best to strictly impose a limit, since funny things can happen when doing calculations on very large or small numbers (such as when you zoom in or out very far). Since we have numbered our tick marks, we can also apply the same trick we used for the y ticks to distinguish between major and minor ticks. Now we are ready to draw the x ticks:

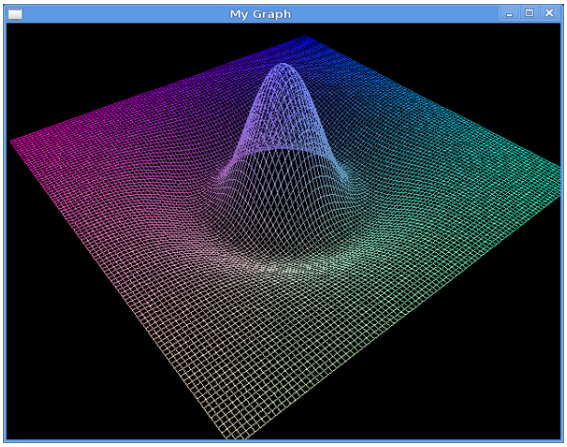
```
int nticks = right_i - left_i + 1; if(nticks > 21) nticks = 21; for(int i = 0; i < nticks; i++) { float x = firsttick + i * tickspacing * scale_x; float tickscale = ((i + left_i) % 10) ? 0.5 : 1; ticks[i * 2].x = x; ticks[i * 2].y = -1; ticks[i * 2 + 1].x = x; ticks[i * 2 + 1].y = -1 - ticksize * tickscale * pixel_y; } glBufferData(GL_ARRAY_BUFFER, nticks * sizeof *ticks, ticks, GL_STREAM_DRAW); glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0, 0); glDrawArrays(GL_LINES, 0, nticks * 2);
```

Exercises:

- Make every fourth tick mark a major one. Make the tick spacing reset every time it crosses a power of 4 instead of 10.
- Calculate the left and right variables using the inverse of the transform matrix.

Chapter 22

OpenGL Programming/Scientific OpenGL Tutorial 04



A three-dimensional plot

Chapter 23

Introduction

Now that we have a grip on drawing two-dimensional plots, it is time to tackle three-dimensional plots. It is not too different from drawing two-dimensional plots, it is just a matter of adding the third dimension and choosing a suitable model-view-projection (MVP) matrix to present the three-dimensional data in a clear way.

One significant difference however is that we now have much more data to plot, as the number of data points is now squared. The strategy from the second graph tutorial to separate the number of vertices drawn from the number of data points will now really pay off, so we will use it in this tutorial as well.

We will also see that when drawing grid lines, vertices are used more than once. To ensure we reuse vertices, and to work around some other problems, we will use an Index Buffer Object (IBO).

Chapter 24

Putting our function in a texture

We will use a 3D version of the Mexican hat function. Basically, this is a function of only one variable, but we will make a rotation symmetrical version of it by using the distance to the origin as that variable:

$$\psi(t) = (1 - t^2)e^{-\frac{1}{2}t^2}, \quad t = \sqrt{x^2 + y^2}$$

We will evaluate this function in a grid of N by N points. We will make it so it is easy to change the exact number of points at compile time:

```
#define N 256 GLbyte graph[N][N]; for(int i = 0; i < N; i++) { for(int j = 0; j < N; j++) { float x = (i - N / 2) / (N / 2.0); float y = (j - N / 2) / (N / 2.0); float t = hypotf(x, y) * 4.0; float z = (1 - t * t) * expf(t * t / −2.0); graph[i][j] = roundf(z * 127 + 128); } }
```

The hypot*() functions are not well known but they are part of the C99 standard, and are very convenient. We scaled the function somewhat to ensure the hat fits nicely within the range $-1..1$. We can now tell OpenGL to use this data as a two dimensional texture:

```
glActiveTexture(GL_TEXTURE0); glGenTextures(1, &texture_id); glBindTexture(GL_TEXTURE_2D, texture_id); glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, N, N, 0, GL_LUMINANCE, GL_UNSIGNED_BYTE, graph);
```

Exercises:

- Try different values of N. What is the maximum size supported by your video card? How much memory does the texture consume?
- We don't need to evaluate all the N x N points in one go. Make it so you evaluate only N points at a time, and put the partial results in the texture with glTexSubImage2D().
- Try using an image instead of a function (you can use the texture from Modern OpenGL Tutorial 06 for example.
- Make it so you can switch texture interpolation and wrapping on and off with the F1 and F2 keys.

Chapter 25

Shaders

The vertex shader we will use works more or less the same as the one from the second graph tutorial. However, there we had the luxury of plotting a 2D function on a 2D screen, so we didn't have to transform our vertex coordinates, we just moved the texture around a little. When plotting a 3D function, we really have to project our vertices somehow to make sense of all the three dimensions. Also, we can move the texture around in two dimensions now. So it makes sense two just have two generic transformation matrices; one for the texture coordinates and one for the vertex coordinates. This is what our shader will look like:

```
attribute vec2 coord2d; uniform mat4 texture_transform; uniform mat4 vertex_transform; uniform sampler2D mytexture; varying vec4 graph_coord; void main(void) { graph_coord = texture_transform * vec4(coord2d, 0, 1); graph_coord.z = texture2D(mytexture, graph_coord.xy / 2.0 + 0.5).r; gl_Position = vertex_transform * vec4(coord2d, graph_coord.z, 1); }
```

The attribute coord2d has the same function as the attribute coord1d from the second graph tutorial. The uniform matrix texture_transform takes over the role of the uniforms offset_x and scale_x. The uniform matrix vertex_transform is new, and will be used to change our view on the graph. In the main function, we recover the graph coordinates by applying the texture_transform matrix to the 2D coordinates that we feed it. Once we know that, we can recover the z coordinate by doing a texture lookup with those coordinates. We keep the graph coordinates in a varying vec4 so they can be used by the fragment shader to give the graph nice colors. The gl_Position variable is calculated the same was as it was done in the second tutorial, except for the new transformation matrix that is applied.

We will use the following fragment shader:

```
varying vec4 graph_coord; void main(void) { gl_FragColor = graph_coord / 2.0 + 0.5; }
```

Chapter 26

Calculating the texture and vertex transformation matrices

If you have followed the previous tutorials, you have already seen how we created a transformation matrix from offset and scale variables. This time is no different, except we have two offset variables, for both the x and y axis:

```
glm::mat4 texture_transform = glm::translate(glm::scale(glm::mat4(1.0f), glm::vec3(scale, scale, 1)), glm::vec3(offset_x, offset_y, 0)); glUniformMatrix4fv(uniform_texture_transform, 1, GL_FALSE, glm::value_ptr(texture_transform));
```

Also, from Modern OpenGL Tutorial 05 we have seen how to create model, view and projection matrices. Let's keep our vertices in a box from $(-1, -1, -1)$ to (1, 1, 1), so our model transformation matrix is just the identity matrix. Then we can position the camera at say, $(0, -2, 2)$, where the vector (0, 0, 1) is the up direction. So we are a bit in front of and above the graph, looking down on it. Finally, we will use the same perspective projection as used in the other tutorials. The resulting MVP matrix is calculated as follows:

```
glm::mat4 model = glm::mat4(1.0f); glm::mat4 view = glm::lookAt(glm::vec3(0.0, −2.0, 2.0), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 0.0, 1.0)); glm::mat4 projection = glm::perspective(45.0f, 1.0f * 640 / 480, 0.1f, 10.0f); glm::mat4 vertex_transform = projection * view * model; glUniformMatrix4fv(uniform_vertex_transform, 1, GL_FALSE, glm::value_ptr(vertex_transform));
```

Exercises:

- Make it so you can change offset_y using the up and down keys, and change the scale with page-up and page-down.
- Change the model matrix such that the graph is slowly rotating around the z axis in time.
- Make it so you can toggle rotation by pressing the F3 key.

Chapter 27

Drawing a grid

One way to plot a three-dimensional function is to draw grid lines. This is exactly what gnuplot does by default when you use the splot command, so we will do that as well. We used 101 points before for the two-dimensional plot, so we will use a grid of 101 by 101 points now, and put that in our VBO:

```
struct point { GLfloat x; GLfloat y; }; point vertices[101][101]; for(int i = 0; i < 101; i++) { for(int j = 0; j < 101; j++) { vertices[i][j].x = (j - 50) / 50.0; vertices[i][j].y = (i - 50) / 50.0; } } glBindBuffer(GL_ARRAY_BUFFER, vbo); glBufferData(GL_ARRAY_BUFFER, sizeof vertices, vertices, GL_STATIC_DRAW);
```

The vertices are in the right order to draw horizontal lines (i.e., those with constant y coordinates). The only problem is that we cannot just make a single call to glDrawArrays(GL_LINE_STRIP), as it would not know when it has reached the right edge of the graph and moves back to the left edge. Instead, it would create a zig-zag line pattern. The most simple solution is to just manually draw 101 lines:

```
glBindBuffer(GL_ARRAY_BUFFER); glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0, 0); for(int i = 0; i < 101; i++) glDrawArrays(GL_LINE_STRIP, 101 * i, 101);
```

That works, although we did make 101 OpenGL calls, which is not so much, but one would rather avoid doing that. We also need to draw vertical lines, but the vertices are not in the right order! Although, in this case, we can cheat by using the stride and pointer parameters:

```
for(int i = 0; i < 101; i++) { glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 101 * sizeof(point), (void *)(i * sizeof(point))); glDrawArrays(GL_LINE_STRIP, 0, 101); }
```

Exercises:

- Why did we have to use an offset pointer in glVertexAttribPointer()? Couldn't we have left it 0 and used glDrawArrays(GL_LINE_STRIP, i, 101) instead?
- Think of a way to create a vertex array with duplicate vertices such that you can draw all horizontal and vertical lines with just one call to glDrawArrays(GL_LINE_STRIP).
- If you are not limited by OpenGL ES, have a look at the glMultiDrawArrays() command.
- Suppose you want to draw triangles to fill the whole 101 x 101 square. Can you order the vertices such that you can draw everything with one call to glDrawArrays(GL_TRIANGLE_STRIP) without wasting vertices? What about when using multiple calls to glDrawArrays()?

Chapter 28

Using an IBO to prevent wasting vertices and OpenGL calls

If we wanted to draw the graph with triangles, to get a filled surface instead of a grid, we could not reuse our VBO anymore, since the order of the vertices for grid lines is completely different than for triangles. If we wanted to draw both a filled surface *and* grid lines on top, we would need multiple copies for all of our vertices, just because of ordering problems, and we would need many glDrawArrays() commands.

Luckily there is a way to decouple a set of vertices from the order in which to draw them. With the glDrawElements() function, we can have a second array which contains indices to the vertex array (or any other attribute array for that matter). Unfortunately, there is still no way to draw with GL_LINE_STRIP, because the index array also cannot tell OpenGL where to start and end the line strips. But we can draw with GL_LINES! Now, you may think that then we have a lot of duplication again, because we would have to draw a line from vertex index 0 to 1, from 1 to 2, and so on. However, indices are small numbers, usually one or two bytes big, while attributes are usually much bigger. Even in our simple case, our 2D vertex attributes are already 8 bytes big. So the overhead is much smaller. The advantages are that we can draw all the line segments for both horizontal and vertical grid lines with one call to glDrawElements(), without any unnecessary pixels drawn. Of course, we can also store the indices in the memory of the GPU, by using Index Buffer Objects. Here goes:

GLushort indices[2 * 100 * 101 * 2]; int i = 0; // Horizontal grid lines for(int y = 0; y < 101; y++) { for(int x = 0; x < 100; x++) { indices[i++] = y * 101 + x; indices[i++] = y * 101 + x + 1; } } // Vertical grid lines for(int x = 0; x < 101; x++) { for(int y = 0; y < 100; y++) { indices[i++] = y * 101 + x; indices[i++] = (y + 1) * 101 + x; } } GLint ibo; glGenBuffers(1, &ibo); glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof indices, indices, GL_STATIC_DRAW);

The amount of indices is two per line segment, and we have 100 segments per grid line. Then, we have two times 101 grid lines. Here is how we finally draw the grid using vertices from our VBO and indices from our IBO:

glEnableVertexAttribArray(attribute_coord2d); glBindBuffer(GL_ARRAY_BUFFER, vbo); glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0, 0); glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); glDrawElements(GL_LINES, 2 * 100 * 101 * 2, GL_UNSIGNED_SHORT, 0);

Exercises:

- Find out how you can draw only the vertical grid lines with a single call to glDrawElements() without changing anything else.
- Is there a limit to the number of vertices that can be drawn with glDrawElements()?
- Create an array of indices to draw a filled surface with GL_TRIANGLES.
- Can you reuse the same IBO with another VBO? Or the same VBO with another IBO?
- Suppose you have two attribute arrays, one for vertices and one for colors. Find out how glDrawElements() works in that case.

- Comment on this page

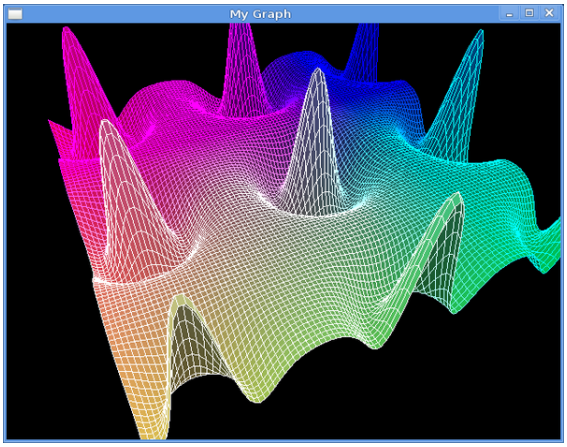
- Recent stats

< OpenGL Programming

Browse & download complete code

Chapter 29

OpenGL Programming/Scientific OpenGL Tutorial 05



Hidden line removal

Chapter 30

Introduction

In the previous tutorial, we have drawn a three-dimensional graph using grid lines. This allowed you to look through the graph, but for more complex functions, this can be rather confusing. It would be much nicer to draw the graph as a continuous, opaque surface. In this tutorial we will see how to do that.

Chapter 31

Front and back faces

Since our graph is not a close surface, but rather a sheet with some bumps in it, it is possible, depending on the orientation, to see both sides of the graph. When OpenGL draws a triangle, it automatically determines which side of the triangle is facing the camera, the front or the back face (see also the two-sided surfaces GLSL tutorial). We will change the fragment shader from the previous tutorial slightly to draw back facing triangles only half as bright as front facing triangles, making it easy to distinguish which side of the graph we are looking at. We will also introduce the uniform color, so that we can modulate the color depending on whether we are drawing the surface or the grid.

#version 120 varying vec4 graph_coord; uniform vec4 color; void main(void) { float factor; if(gl_FrontFacing) factor = 1.0; else factor = 0.5; gl_FragColor = (graph_coord / 2.0 + 0.5) * color * factor; }

Chapter 32

Drawing a surface

In the previous tutorial, we created a VBO which contained all the x and y coordinates of the graph. We also created an IBO that traced the horizontal and vertical grid lines. To draw a surface, we will reuse the VBO, but we will have to create a new IBO that describes how to draw the triangles that make up the surface of the graph:

```
GLushort indices[100 * 100 * 6]; int i = 0; // Triangles for(int y = 0; y < 100; y++) { for(int x = 0; x < 100; x++) { indices[i++] = y * 101 + x; indices[i++] = y * 101 + x + 1; indices[i++] = (y + 1) * 101 + x + 1; indices[i++] = y * 101 + x; indices[i++] = (y + 1) * 101 + x + 1; indices[i++] = (y + 1) * 101 + x; } } GLuint surface_ibo; glGenBuffers(1, &surface_ibo); glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, surface_ibo); glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof indices, indices, GL_STATIC_DRAW);
```

To draw the surface with the new shader, we use the following commands:

```
GLfloat white[4] = {1, 1, 1, 1}; glUniform4fv(uniform_color, 1, white); glEnableVertexAttribArray(attribute_coord2d); glBindBuffer(GL_ARRAY_BUFFER, vbo); glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0, 0); glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, surface_ibo); glDrawElements(GL_TRIANGLES, 100 * 100 * 6, GL_UNSIGNED_SHORT, 0);
```

If you do this, and rotate the graph, you will notice that depending on the orientation, the surface is not always drawn correctly. In orientations where the back most triangles are drawn first, there is no problem. However, when the front most triangles are drawn first, triangles that are more to the back can overwrite the triangles that are in the front. To prevent this from happening, we need to enable the depth buffer of course. In the main() function, use:

```
glutInitDisplayMode(GLUT_RGBA|GLUT_DEPTH|GLUT_DOUBLE);
```

And in init_resources() add:

```
glEnable(GL_DEPTH_TEST);
```

Also do not forget to clear the depth buffer before drawing a new frame:

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

Exercises:

- Could we not simply reverse the order in which we draw the triangles when necessary? Perhaps with a negative stride parameter in the glVertexAttribPointer() call?
- Try changing the order of the vertices in the triangles in various ways.
- Try to get the same result without specifying backside triangles by using glEnable(GL_CULL_FACE) and glCullFace(GL_FRONT_AND_BACK).

Chapter 33

Drawing the grid on top of the surface

Although the surface has its appeals, it is much harder to see subtle curves. It would be nice to draw the grid on top of the surface. We already know how to do that, so we just switch back to our grid IBO and draw using GL_LINES after we have drawn the surface. However, we have to give the grid lines a different color than the surface. Let's make them twice as bright as the surface:

GLfloat bright[4] = {2, 2, 2, 1}; glUniform4fv(uniform_color, 1, bright); glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); glDrawElements(GL_LINES, 100 * 101 * 4, GL_UNSIGNED_SHORT, 0);

When you do this, you will notice two things. The grid lines look horrible, but you can no longer see "hidden" grid lines. This is all because the depth test, by default, will ensure only fragments are drawn that are closer to the camera than any existing fragment at that position. The surface was drawn first, so no fragments will be drawn that are behind it. But if the grid lines are using the same vertices as the surface, you would expect the depth to be exactly the same. With the default depth test, you would think no grid lines should appear at all. In reality though, a line is not drawn in the same way as a triangle, and floating point rounding errors will cause some grid fragments to be slightly in front, and some slightly behind the surface.

To fix the grid lines, they should be drawn slightly closer to the camera than the surface, or the surface should be drawn slightly farther away. We could do that by applying a translation to the MVP matrix, but there is an OpenGL call that specifically addresses this problem:

glPolygonOffset(1, 1); glEnable(GL_POLYGON_OFFSET_FILL);

This will cause triangles (but not lines or points) to be drawn with a slightly increased depth value, with the result that the grid lines now appear as they should.

Exercises:

- Make it so you can turn the polygon offset on and off with the F4 key.
- The default depth test function is "lesser than" (GL_LESS). Try changing it to "lesser than or equal" with glDepthFunc(GL_LEQUAL). Does it help?
- Try different values for glPolygonOffset(), larger and/or negative.
- Between drawing the surface and the grid, try clearing either the depth buffer or the color buffer.
- Would it be possible to draw the grid before drawing the surface?

- Comment on this page

- Recent stats

< OpenGL Programming

Browse & download complete code

Chapter 34

OpenGL Programming/Modern OpenGL Tutorial 06

Chapter 35

Loading a texture

Our texture, in 2D

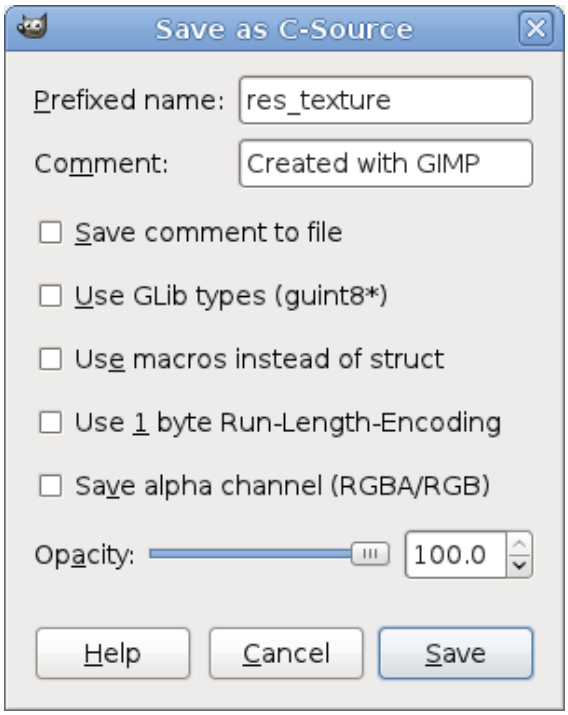
To load a texture, we need code to load images in a particular format, like JPEG or PNG. Usually, your final program will use generic libraries such as SDL_Image, SFML or Irrlicht, that support various image formats, so you won't have to write your own image-loading code. Specialized libraries such as SOIL (see below) may interest you as well.

In a first step, we need to manipulate the image at a low level to understand the basics, so we'll use a trick : GIMP can export an image as C source code, that we can read as-is from our program! I used the save options in the GIMp screenshot on the right.

If there's demand, we may provide a special tutorial to read a simple format like PNM, or a subset of BMP or TGA (these two are also simple, but support compression and various formats so it's hard to support all their options).

Note: Bundling images as C code is not super-memory-efficient, so don't generalize it. Technically: it's stored in the program BSS segment rather than in the heap, so it cannot be freed.

Note 2: you can find the GIMP source as res_texture.xcf in the code repository.



Exporting image as C from GIMP

To automatically rebuild the application when you modify res_texture.c, add this to the Makefile:

```
cube.o: res_texture.c
```

Chapter 36

Creating a texture OpenGL buffer

The buffer is basically a memory slot inside the graphic card, so OpenGL can access it very quickly.

/* Globals */ GLuint texture_id, program_id; GLint uniform_mytexture;
/* init_resources */ glGenTextures(1, &texture_id); glBindTexture(GL_TEXTURE_2D, texture_id); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexImage2D(GL_TEXTURE_2D, // target 0, // level, 0 = base, no minimap, GL_RGB, // internalformat res_texture.width, // width res_texture.height, // height 0, // border, always 0 in OpenGL ES GL_RGB, // format GL_UNSIGNED_BYTE, // type res_texture.pixel_data);
/* render */ glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, texture_id); uniform_mytexture = glGetUniformLocation(program_id, “mytexture”); glUniform1i(uniform_mytexture, /*GL_TEXTURE*/0);
/* free_resources */ glDeleteTextures(1, &texture_id);

Chapter 37

Texture coordinates

We now need to say where each vertex is located on our texture. For this, we'll replace the v_color attribute to the vertex shader with a texcoord:

GLint attribute_coord3d, attribute_v_color**, attribute_texcoord**;
/* init_resources */ attribute_name = “texcoord"; attribute_texcoord = glGetAttribLocation(program, attribute_name); if (attribute_texcoord == −1) { fprintf(stderr, “Could not bind attribute %s\n”, attribute_name); return 0; }

Now, what part of our texture do we map to, say, the top-left corner of the front face? Well, it depends:

- for the front face: the top-left corner of our texture
- for the top face: the bottom-left corner of our texture

We see that multiple texture points will be attached to the same vertex. The vertex shader won't be able to decide which one to pick.

So we need rewrite the cube by using 4 vertices per face, no reused vertices.

For a start though, we'll just work on the front face. Easy! We just have to only display the 2 first triangles (6 first vertices):

glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);

So, our texture coordinates are in the [0, 1] range, with x axis from left to right, and y axis from bottom to top:

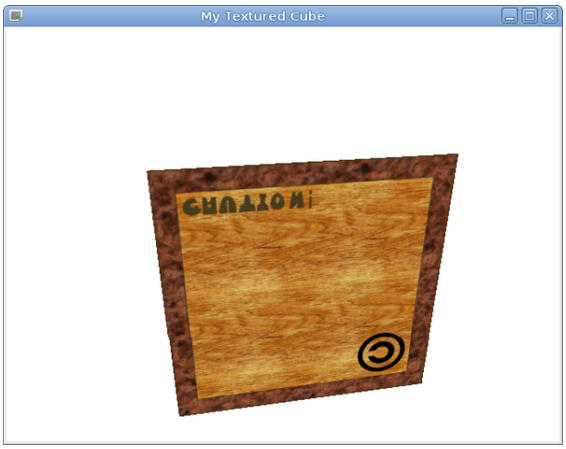
/* init_resources */ GLfloat cube_texcoords[] = { // front 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, }; glGenBuffers(1, &vbo_cube_texcoords); glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_texcoords); glBufferData(GL_ARRAY_BUFFER, sizeof(cube_texcoords), cube_texcoords, GL_STATIC_DRAW);
/* onDisplay */ glEnableVertexAttribArray(attribute_texcoord); glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_texcoords); glVertexAttribPointer(attribute_texcoord, // attribute 2, // number of elements per vertex, here (x,y) GL_FLOAT, // the type of each element GL_FALSE, // take our values as-is 0, // no extra data between each position 0 // offset of first element);

Vertex shader:

attribute vec3 coord3d; attribute vec2 texcoord; varying vec2 f_texcoord; uniform mat4 mvp; void main(void) { gl_Position = mvp * vec4(coord3d, 1.0); f_texcoord = texcoord; }

Fragment shader:

varying vec2 f_texcoord; uniform sampler2D mytexture; void main(void) { gl_FragColor = texture2D(mytexture, f_texcoord); }
But what happens? Our texture is upside-down!



Something is wrong...

The OpenGL convention (origin at the bottom-left corner) is different than in 2D applications (origin at the top-left corner). To fix this we can either:

- read the pixels lines from bottom to top
- swap the pixel lines
- swap the texture Y coordinates

Most graphics libraries return a pixels array in the 2D convention. However, DevIL has an option to position the origin and avoid this issue. Alternatively, some formats such as BMP and TGA store pixel lines from bottom to top natively (which may explain a certain popularity of the otherwise heavy TGA format among 3D developers), useful if you write a custom loader for them.

Swapping the pixel lines can be done in the C code at run time, too. If you program in high-level languages such as Python this can even be done in one line. The drawback is that texture loading will be somewhat slower because of this extra step.

Reversing the texture coordinates is the easiest way for us, we can do that in the fragment shader:

```
void main(void) { vec2 flipped_texcoord = vec2(f_texcoord.x, 1.0 - f_texcoord.y); gl_FragColor = texture2D(mytexture, flipped_texcoord); }
```

OK, technically we could have written the texture coordinates in the other direction in the first place - but other 3D applications tend to work the way we describe.

Chapter 38

Bumping to a full cube

So as we discussed, we specify independent vertices for each faces:

GLfloat cube_vertices[] = { // front −1.0, −1.0, 1.0, 1.0, −1.0, 1.0, 1.0, 1.0, 1.0, −1.0, 1.0, 1.0, // top −1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, −1.0, −1.0, 1.0, −1.0, // back 1.0, −1.0, −1.0, −1.0, −1.0, −1.0, −1.0, 1.0, −1.0, 1.0, 1.0, −1.0, // bottom −1.0, −1.0, −1.0, 1.0, −1.0, −1.0, 1.0, −1.0, 1.0, −1.0, −1.0, 1.0, // left −1.0, −1.0, −1.0, −1.0, −1.0, 1.0, −1.0, 1.0, 1.0, −1.0, 1.0, −1.0, // right 1.0, −1.0, 1.0, 1.0, −1.0, −1.0, 1.0, 1.0, −1.0, 1.0, 1.0, 1.0, };

For each face, vertices are added counter-clockwise (when the viewer is facing that face). Consequently, the texture mapping will be identical for all faces:

GLfloat cube_texcoords[2*4*6] = { // front 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, }; for (int i = 1; i < 6; i++) memcpy(&cube_texcoords[i*4*2], &cube_texcoords[0], 2*4*sizeof(GLfloat));

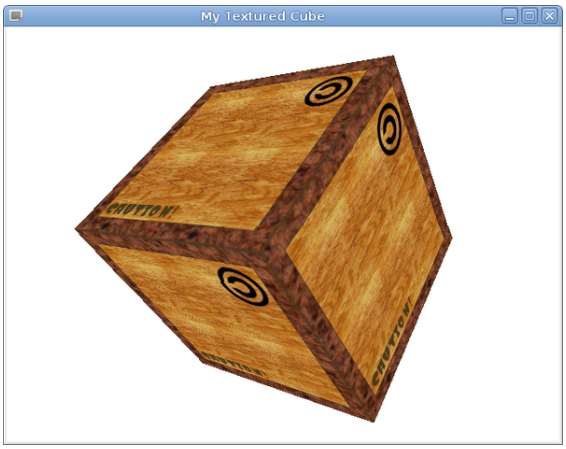
Here we specified the mapping for the front face, and copied it on all remaining 5 faces.

If a face were clockwise instead of counter-clockwise, then the texture would be shown mirrored. There's no convention on the orientation, you just have to make sure that the texture coordinates are properly mapped to the vertices.

The cube elements are also written similarly, with 2 triangle with indices (x, x+1, x+2), (x+2, x+3, x):

GLushort cube_elements[] = { // front 0, 1, 2, 2, 3, 0, // top 4, 5, 6, 6, 7, 4, // back 8, 9, 10, 10, 11, 8, // bottom 12, 13, 14, 14, 15, 12, // left 16, 17, 18, 18, 19, 16, // right 20, 21, 22, 22, 23, 20, }; ... glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_cube_elements); int size; glGetBufferParameteriv(GL_ELEMENT_ARRAY_BUFFER, GL_BUFFER_SIZE, &size); glDrawElements(GL_TRIANGLES, size/sizeof(GLushort), GL_UNSIGNED_SHORT, 0);

For additional fun, and to check the bottom face, let's implement the 3-rotations movement showcased in NeHe's flying cube tutorial, in onIdle:



Fly, cube, fly!

float angle = glutGet(GLUT_ELAPSED_TIME) / 1000.0 * 15; // base 15° per second glm::mat4 anim = \ glm::rotate(glm::mat4(1.0f), angle*3.0f, glm::vec3(1, 0, 0)) * // X axis glm::rotate(glm::mat4(1.0f), angle*2.0f, glm::vec3(0, 1, 0)) * // Y axis glm::rotate(glm::mat4(1.0f), angle*4.0f, glm::vec3(0, 0, 1)); // Z axis

We're done!

Chapter 39

Using SOIL

WIP

SOIL provides a way to load an image file in PNG, JPG and a few other formats, designed for OpenGL integration. It's a pretty minimal library with no dependency. It's used under the hood by SFML (although SFML also uses libjpeg and libpng directly).

Install it (look for a package named libsoil, libsoil-dev, or something similar).

Reference it in your Makefile:

LDLIBS=-lglut -lSOIL -lGLEW -lGL -lm

Include the soil header:

#include <SOIL/SOIL.h>

One high-level function allows you to upload it directly to the OpenGL context:

glActiveTexture(GL_TEXTURE0); GLuint texture_id = SOIL_load_OGL_texture (“res_texture.png”, SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_INVERT_Y); if(texture_id == 0) cerr << “SOIL loading error: '" << SOIL_last_result() << "' (" << “res_texture.png” << ")" << endl;

- SOIL_FLAG_INVERT_Y deals with the reverse-Y-coordinates issue we experienced above.
- SOIL also adapt NPOT (non power of 2) textures, when the graphic card doesn't handle these directly

Note that with this method, you do not have access to the image dimensions. To get them, you need to use a lower-level API:

int img_width, img_height; unsigned char* img = SOIL_load_image(“res_texture.png”, &img_width, &img_height, NULL, 0); glGenTextures(1, &texture_id); glBindTexture(GL_TEXTURE_2D, texture_id); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, img_width, img_height, 0, GL_RGB, GL_UNSIGNED_BYTE, img);

Chapter 40

Further reading

- Textures in the legacy OpenGL 1.x section
- SOIL homepage

- Comment on this page

- Recent stats

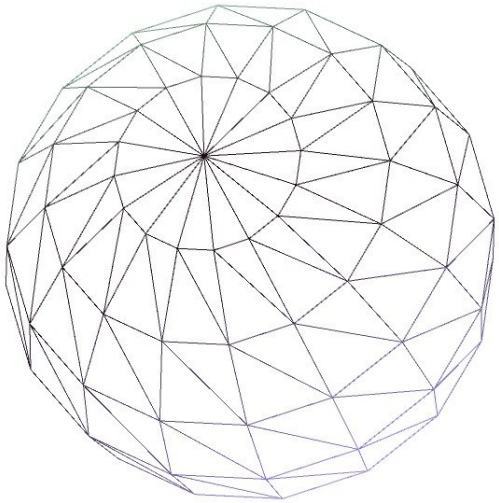
< OpenGL Programming

Browse & download complete code

Chapter 41

GLSL Programming/GLUT/Textured Spheres

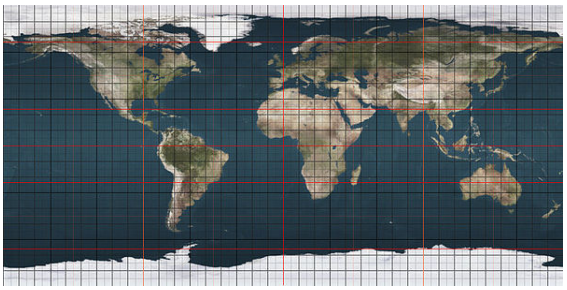
The Earth seen from Apollo 17. The shape of the Earth is close to a quite smooth sphere.



A triangle mesh approximating a sphere.

This tutorial introduces **texture mapping**.

It's the first in a series of tutorials about texturing in GLSL shaders in OpenGL 2.x. In this tutorial, we start with a single texture map on a sphere. More specifically, we map an image of the Earth's surface onto a sphere. Based on this, further tutorials cover topics such as lighting of textured surfaces, transparent textures, multitexturing, gloss mapping, etc.



An image of the Earth's surface. The horizontal coordinate represents the longitude, the vertical coordinate the latitude.

41.0.1 Texture Mapping

The basic idea of "texture mapping" (or "texturing") is to map an image (i.e. a "texture" or a "texture map") onto a triangle mesh; in other words, to put a flat image onto the surface of a three-dimensional shape.

To this end, "texture coordinates" are defined, which simply specify the position in the texture (i.e. image). The horizontal coordinate is officially called S and the vertical coordinate T. However, it is very common to refer to them as x and y. In animation and modeling tools, texture coordinates are usually called U and V.

In order to map the texture image to a mesh, every vertex of the mesh is given a pair of texture coordinates. (This process (and the result) is sometimes called "UV mapping" since each vertex is mapped to a point in the UV-space.) Thus, every vertex is mapped to a point in the texture image. The texture coordinates of the vertices can then be interpolated for each point of any triangle between three vertices and thus every point of all triangles of the mesh can have a pair of (interpolated) texture

coordinates. These texture coordinates map each point of the mesh to a specific position in the texture map and therefore to the color at this position. Thus, rendering a texture-mapped mesh consists of two steps for all visible points: interpolation of texture coordinates and a look-up of the color of the texture image at the position specified by the interpolated texture coordinates.

In OpenGL, any valid floating-point number is a valid texture coordinate. However, when the GPU is asked to look up a pixel (or "texel") of a texture image (e.g. with the "texture2D" instruction described below), it will internally map the texture coordinates to the range between 0 and 1 in a way depending on the "Wrap Mode" that is specified when importing the texture: wrap mode "repeat" basically uses the fractional part of the texture coordinates to determine texture coordinates in the range between 0 and 1. On the other hand, wrap mode "clamp" clamps the texture coordinates to this range. These internal texture coordinates in the range between 0 and 1 are then used to determine the position in the texture image: $(0, 0)$ specifies the lower, left corner of the texture image; $(1, 0)$ the lower, right corner; $(0, 1)$ the upper, left corner; etc.

41.0.2 Texturing a Sphere

To map the image of the Earth's surface onto a sphere, you first have to load the image. For this, use SOIL as explained in OpenGL Programming Tutorial 06.

glActiveTexture(GL_TEXTURE0); GLuint texture_id = SOIL_load_OGL_texture ("Earthmap720x360_grid.jpg", SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_INVERT_Y | SOIL_FLAG_TEXTURE_REPEATS);

SOIL_FLAG_TEXTURE_REPEATS will make the texture repeat itself when using texture coordinates outside of [0, 1].

Vertex shader:

attribute vec3 v_coord; varying vec4 texCoords; uniform mat4 m,v,p; void main(void) { mat4 mvp = p*v*m; gl_Position = mvp * vec4(v_coord, 1.0); texCoords = vec4(v_coord, 1.0); }

Fragment shader:

varying vec4 texCoords; uniform sampler2D mytexture; void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); // processing of the texture coordinates; // this is unnecessary if correct texture coordinates are specified by the application gl_FragColor = texture2D(mytexture, longitudeLatitude); // look up the color of the texture image specified by the uniform "mytexture" // at the position specified by "longitudeLatitude.x" and // "longitudeLatitude.y" and return it in "gl_FragColor" }

If everything went right, the texture image should now appear on the sphere. Congratulations!

41.0.3 How It Works

Since many techniques use texture mapping, it pays off very well to understand what is happening here. Therefore, let's review the shader code:

The vertices of the sphere object come from the FreeGLUT glutSolidSphere function. We'll need them in the fragment shader to convert them to texture coordinates in the space of the texture image.

The vertex shader then writes the texture coordinates of each vertex to the varying variable texCoords. For each fragment of a triangle (i.e. each covered pixel), the values of this varying at the three triangle vertices are interpolated (see the description in "Rasterization") and the interpolated texture coordinates are given to the fragment shader. The fragment shader then uses them to look up a color in the texture image specified by the uniform mytexture at the interpolated position in texture space and returns this color in gl_FragColor, which is then written to the framebuffer and displayed on the screen.

In this case, we generate the texture coordinates algorithmically, but usually they are specified through your 3D modeler, and passed as additional vertex attributes.

It is crucial that you gain a good idea of these steps in order to understand the more complicated texture mapping techniques presented in other tutorials.

41.0.4 Repeating and Moving Textures

In some 3D frameworks, you might have met parameters **Tiling** and **Offset**, each with an **x** and a **y** component. These parameters allow you to repeat the texture (by shrinking the texture image in texture coordinate space) and move the texture image on the surface (by offsetting it in texture coordinate space). To reproduce this behavior, another uniform has to be defined:

uniform vec4 mytexture_ST; // tiling and offset parameters

We can specify such a vec4 uniform for each texture. (Remember: "S" and "T" are the official names of the texture coordinates, which are usually called "U" and "V", or "x" and "y".) This uniform holds the **x** and **y** components of the **Tiling** parameter in mytexture_ST.x and mytexture_ST.y, while the **x** and **y** components of the **Offset** parameter are stored in mytexture_ST.w and mytexture_ST.z. The uniform should be used like this:

gl_FragColor = texture2D(mytexture, mytexture_ST.xy * texCoords.xy + mytexture_ST.zw);

This makes the shader behave like the built-in shaders. In the other tutorials, this feature is usually not implemented in order to keep the shader code a bit cleaner.

And just for completeness, here is the new fragment shader code with this feature:

varying vec4 texCoords; uniform sampler2D mytexture; uniform vec4 mytexture_ST; // tiling and offset parameters void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); // Apply tiling and offset vec2 texCoordsTransformed = longitudeLatitude * mytexture_ST.xy + mytexture_ST.zw; gl_FragColor = texture2D(mytexture, texCoordsTransformed); }

You can try, for instance, to duplicate all continents (scale 2x horizontally to see the texture twice - make sure your texture is GL_REPEATing), and reduce the north pole (start at −0.05 vertically to reduce the top):

glUniform4f(uniform_mytexture_ST, 2,1, 0,−.05);

41.0.5 Summary

You have reached the end of one of the most important tutorials. We have looked at:

- How to import a texture image and how to attach it to a texture property of a shader.
- How a vertex shader and a fragment shader work together to map a texture image onto a mesh.
- How tiling and offset parameters for textures work and how to implement them.

41.0.6 Further Reading

If you want to know more

- about the data flow in and out of vertex shaders and fragment shaders (i.e. vertex attributes, varyings, etc.), you should read the description of the "OpenGL ES 2.0 Pipeline".
- about the interpolation of varying variables for the fragment shader, you should read the discussion of the "Rasterization".

< GLSL Programming/GLUT

Chapter 42

GLSL Programming/GLUT/Lighting Textured Surfaces

Earthrise as seen from Apollo 8.

This tutorial covers **per-vertex lighting of textured surfaces**.

It combines the shader code of the tutorial on textured spheres and the tutorial on specular highlights to compute lighting with a diffuse material color determined by a texture. If you haven't read the tutorial on textured spheres or the tutorial on specular highlights, this would be a very good opportunity to read them.

42.0.7 Texturing and Diffuse Per-Vertex Lighting

In the tutorial on textured spheres, the texture color was used as output of the fragment shader. However, it is also possible to use the texture color as any of the parameters in lighting computations, in particular the material constant k_{diffuse} for diffuse reflection, which was introduced in the tutorial on diffuse reflection. It appears in the diffuse part of the Phong reflection model:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \max(0, \mathbf{N} \cdot \mathbf{L})$$

where this equation is used with different material constants for the three color components red, green, and blue. By using a texture to determine these material constants, they can vary over the surface.

42.0.8 Shader Code

In comparison to the per-vertex lighting in the tutorial on specular highlights, the vertex shader here computes two varying colors: diffuseColor is multiplied with the texture color in the fragment shader and specularColor is just the specular term, which shouldn't be multiplied with the texture color. This makes perfect sense but for historically reasons (i.e. older graphics hardware that was less capable) this is sometimes referred to as “separate specular color”.

attribute vec3 v_coord; attribute vec3 v_normal; uniform mat4 m, v, p; uniform mat3 m_3x3_inv_transp; uniform mat4 v_inv; varying vec3 diffuseColor; // the diffuse Phong lighting computed in the vertex shader varying vec3 specularColor; // the specular Phong lighting computed in the vertex shader varying vec4 texCoords; // the texture coordinates struct lightSource { vec4 position; vec4 diffuse; vec4 specular; float constantAttenuation, linearAttenuation, quadraticAttenuation; float spotCutoff, spotExponent; vec3 spotDirection; }; lightSource light0 = lightSource(vec4(0.0, 1.0, 2.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), 0.0, 1.0, 0.0, 180.0, 0.0, vec3(0.0, 0.0, 0.0)); vec4 scene_ambient = vec4(0.2, 0.2, 0.2, 1.0); struct material { vec4 ambient; vec4 diffuse; vec4 specular; float shininess; }; material mymaterial = material(vec4(0.2, 0.2, 0.2, 1.0), vec4(1.0, 0.8, 0.8, 1.0), vec4(1.0, 1.0, 1.0, 1.0), 5.0); void main(void) { vec4 v_coord4 = vec4(v_coord, 1.0); mat4 mvp = p*v*m; vec3 normalDirection = normalize(m_3x3_inv_transp * v_normal); vec3 viewDirection = normalize(vec3(v_inv * vec4(0.0, 0.0, 0.0, 1.0) - m * v_coord4)); vec3 lightDirection; float attenuation; if (light0.position.w == 0.0) // directional light { attenuation = 1.0; // no attenuation lightDirection = normalize(vec3(light0.position)); } else // point or spot light (or other kind of light) { vec3 vertexToLight-

Source = vec3(light0.position - m * v_coord4); float distance = length(vertexToLightSource); lightDirection = normalize(vertexToLightSource); attenuation = 1.0 / (light0.constantAttenuation + light0.linearAttenuation * distance + light0.quadraticAttenuation * distance * distance); if (light0.spotCutoff <= 90.0) // spotlight { float clampedCosine = max(0.0, dot(-lightDirection, normalize(light0.spotDirection))); if (clampedCosine < cos(radians(light0.spotCutoff))) // outside of spotlight cone { attenuation = 0.0; } else { attenuation = attenuation * pow(clampedCosine, light0.spotExponent); } } } vec3 ambientLighting = vec3(scene_ambient); // without material color! vec3 diffuseReflection = attenuation * vec3(light0.diffuse) * max(0.0, dot(normalDirection, lightDirection)); // without material color! vec3 specularReflection; if (dot(normalDirection, lightDirection) < 0.0) // light source on the wrong side? { specularReflection = vec3(0.0, 0.0, 0.0); // no specular reflection } else // light source on the right side { specularReflection = attenuation * vec3(light0.specular) * vec3(mymaterial.specular) * pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), mymaterial.shininess); } diffuseColor = ambientLighting + diffuseReflection; specularColor = specularReflection; texCoords = v_coord4; gl_Position = mvp * v_coord4; }

And the fragment shader modulates the diffuseColor with the texture color and adds the specularColor:

varying vec3 diffuseColor; // the interpolated diffuse Phong lighting varying vec3 specularColor; // the interpolated specular Phong lighting varying vec4 texCoords; // the interpolated texture coordinates uniform sampler2D mytexture; void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); // unusual processing of texture coordinates gl_FragColor = vec4(diffuseColor * vec3(texture2D(mytexture, longitudeLatitude)) + specularColor, 1.0); }

In order to assign a texture image to this shader, you should follow the steps discussed in the tutorial on textured spheres.

42.0.9 Summary

Congratulations, you have reached the end. We have looked at:

- How texturing and per-vertex lighting are usually combined.
- What a “separate specular color” is.

42.0.10 Further Reading

If you still want to know more

- about the diffuse reflection term of the Phong reflection model, you should read the tutorial on diffuse reflection.
- about per-vertex lighting or the rest of the Phong reflection model, i.e. the ambient and the specular term, you should read tutorial on specular highlights.
- about the basics of texturing, you should read tutorial on textured spheres.

< GLSL Programming/GLUT

Chapter 43

GLSL Programming/GLUT/Glossy Textures

Sun set with a specular highlight in the Pacific Ocean as seen from the International Space Station (ISS).

This tutorial covers **per-pixel lighting of partially glossy, textured surfaces**.

It combines the shader code of the tutorial on textured spheres and the tutorial on smooth specular highlights to compute per-pixel lighting with a material color for diffuse reflection that is determined by the RGB components of a texture and an intensity of the specular reflection that is determined by the A component of the same texture. If you haven't read the tutorial on textured spheres or the tutorial on smooth specular highlights, this would be a very good opportunity to read them.

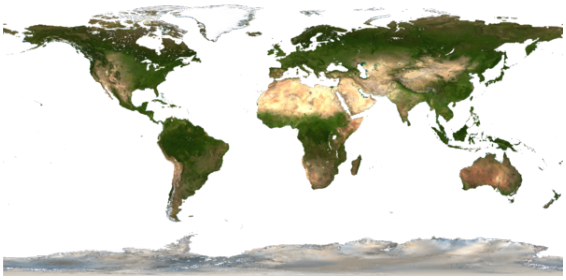
43.0.11 Gloss Mapping

The tutorial on lighting textured surfaces introduced the concept of determining the material constant for the diffuse reflection by the RGB components of a texture image. Here we extend this technique and determine the strength of thea specular reflection by the A (alpha) component of the same texture image. Using only one texture offers a significant performance advantage, in particular because an RGBA texture lookup is under certain circumstances just as expensive as an RGB texture lookup.

If the "gloss" of a texture image (i.e. the strength of the specular reflection) is encoded in the A (alpha) component of an RGBA texture image, we can simply multiply the material constant for the specular reflection k_{specular} with the alpha component of the texture image. k_{specular} was introduced in the tutorial on specular highlights and appears in the specular reflection term of the Phong reflection model:

$$I_{\text{specular}} = I_{\text{incoming}}\, k_{\text{specular}} \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$$

If multiplied with the alpha component of the texture image, this term reaches its maximum (i.e. the surface is glossy) where alpha is 1, and it is 0 (i.e. the surface is not glossy at all) where alpha is 0.



Map of the Earth with transparent water, i.e. the alpha component is 0 for water and 1 for land.

43.0.12 Shader Code for Per-Pixel Lighting

The shader code is a combination of the per-pixel lighting from the tutorial on smooth specular highlights and the texturing from the tutorial on textured spheres. Similarly to the tutorial on lighting textured surfaces, the RGB components of the texture color in textureColor is multiplied to the ambient and diffuse lighting.

In the particular texture image to the left, the alpha component is 0 for water and 1 for land. However, it should be the water that is glossy and the land that isn't. Thus, with this particular image, we should multiply the specular material color with (1.0 - textureColor.a). On the other hand, usual gloss maps would require a multiplication with textureColor.a. (Note how easy it is to make this kind of changes to a shader program.)

The vertex shader is then:

attribute vec3 v_coord; attribute vec3 v_normal; varying vec4 position; // position of the vertex (and fragment) in world space varying vec3 varyingNormalDirection; // surface normal vector in world space varying vec4 texCoords; // the texture coordinates uniform mat4 m, v, p; uniform mat3 m_3x3_inv_transp; void main() { vec4 v_coord4 = vec4(v_coord, 1.0); mat4 mvp = p*v*m; position = m * v_coord4; varyingNormalDirection = normalize(m_3x3_inv_transp * v_normal); texCoords = v_coord4; gl_Position = mvp * v_coord4; }

And the fragment shader becomes:

varying vec4 position; // position of the vertex (and fragment) in world space varying vec3 varyingNormalDirection; // surface normal vector in world space varying vec4 texCoords; // the texture coordinates uniform mat4 m, v, p; uniform mat4 v_inv; uniform sampler2D mytexture; struct lightSource { vec4 position; vec4 diffuse; vec4 specular; float constantAttenuation, linearAttenuation, quadraticAttenuation; float spotCutoff, spotExponent; vec3 spotDirection; }; lightSource light0 = lightSource(vec4(0.0, 1.0, 0.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), 0.0, 1.0, 0.0, 180.0, 0.0, vec3(0.0, 0.0, 0.0)); vec4 scene_ambient = vec4(0.2, 0.2, 0.2, 1.0); struct material { vec4 ambient; vec4 diffuse; vec4 specular; float shininess; }; material frontMaterial = material(vec4(0.2, 0.2, 0.2, 1.0), vec4(1.0, 0.8, 0.8, 1.0), vec4(1.0, 1.0, 1.0, 1.0), 5.0); void main() { vec3 normalDirection = normalize(varyingNormalDirection); vec3 viewDirection = normalize(vec3(v_inv * vec4(0.0, 0.0, 0.0, 1.0) - position)); vec3 lightDirection; float attenuation; vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); // unusual processing of texture coordinates vec4 textureColor = texture2D(mytexture, longitudeLatitude); if (0.0 == light0.position.w) // directional light? { attenuation = 1.0; // no attenuation lightDirection = normalize(vec3(light0.position)); } else // point light or spotlight (or other kind of light) { vec3 positionToLightSource = vec3(light0.position - position); float distance = length(positionToLightSource); lightDirection = normalize(positionToLightSource); attenuation = 1.0 / (light0.constantAttenuation + light0.linearAttenuation * distance + light0.quadraticAttenuation * distance * distance); if (light0.spotCutoff <= 90.0) // spotlight? { float clampedCosine = max(0.0, dot(-lightDirection, light0.spotDirection)); if (clampedCosine < cos(radians(light0.spotCutoff))) // outside of spotlight cone? { attenuation = 0.0; } else { attenuation = attenuation * pow(clampedCosine, light0.spotExponent); } } } vec3 ambientLighting = vec3(scene_ambient) * vec3(textureColor); vec3 diffuseReflection = attenuation * vec3(light0.diffuse) * vec3(textureColor) * max(0.0, dot(normalDirection, lightDirection)); vec3 specularReflection; if (dot(normalDirection, lightDirection) < 0.0) // light source on the wrong side? { specularReflection = vec3(0.0, 0.0, 0.0); // no specular reflection } else // light source on the right side { specularReflection = attenuation * vec3(light0.specular) * vec3(frontMaterial.specular) * (1.0 - textureColor.a) // for usual gloss maps: "* textureColor.a” * pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), frontMaterial.shininess); } gl_FragColor = vec4(ambientLighting + diffuseReflection + specularReflection, 1.0); }

The texture and sphere have to be set up as described in the tutorial on textured spheres.

A useful modification of this shader for the particular texture image above, would be to set the diffuse material color to a dark blue where the alpha component is 0.

43.0.13 Shader Code for Per-Vertex Lighting

As discussed in the tutorial on smooth specular highlights, specular highlights are usually not rendered very well with per-vertex lighting. Sometimes, however, there is no choice because of performance limitations. In order to include gloss mapping in the shader code of the tutorial on lighting textured surfaces, the fragment shader should be replaced with this code:

varying vec3 diffuseColor; // the interpolated diffuse Phong lighting varying vec3 specularColor; // the interpolated specular Phong lighting varying vec4 texCoords; // the interpolated texture coordinates uniform sampler2D mytexture; void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); // unusual processing of texture coordinates vec4 textureColor = texture2D(mytexture, longitudeLatitude); gl_FragColor = vec4(diffuseColor * vec3(textureColor) + specularColor * (1.0 - textureColor.a), 1.0); }

Note that a usual gloss map would require a multiplication with textureColor.a instead of (1.0 - textureColor.a).

43.0.14 Summary

Congratulations! You finished an important tutorial about gloss mapping. We have looked at:

- What gloss mapping is.
- How to implement it for per-pixel lighting.
- How to implement it for per-vertex lighting.

43.0.15 Further Reading

If you still want to learn more

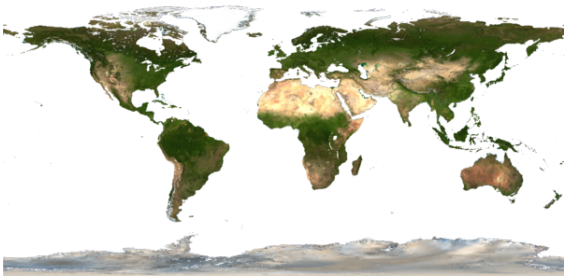
- about per-pixel lighting (without texturing), you should read tutorial on smooth specular highlights.
- about texturing, you should read tutorial on textured spheres.
- about per-vertex lighting with texturing, you should read tutorial on lighting textured surfaces.

page traffic for 90 days

< GLSL Programming/GLUT

Chapter 44

GLSL Programming/GLUT/Transparent Textures



Map of the Earth with transparent water, i.e. the alpha component is 0 for water and 1 for land.

This tutorial covers various common uses of **alpha texture maps**, i.e. RGBA texture images with an A (alpha) component that specifies the opacity of texels.

It combines the shader code of the tutorial on textured spheres with concepts that were introduced in the tutorial on cutaways and the tutorial on transparency.

If you haven't read these tutorials, this would be a very good opportunity to read them.

44.0.16 Discarding Transparent Fragments

Let's start with discarding fragments as explained in the tutorial on cutaways. Follow the steps described in the tutorial on textured spheres and assign the image to the left to the material of a sphere with the following fragment shader (keep the same vertex shader):

varying vec4 texCoords; uniform sampler2D mytexture; float cutoff = 0.1; void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); gl_FragColor = texture2D(mytexture, longitudeLatitude); if (gl_FragColor.a < cutoff) // alpha value less than user-specified threshold? { discard; // yes: discard this fragment } }

(You could pass the cutoff variable as a uniform.)

If you start the application now, the fragment shader should read the RGBA texture and compare the alpha value against the threshold specified in the variable cutoff. If the alpha value is less than the threshold, the fragment is discarded and the surface appears transparent.

Since we can look through the transparent parts, it makes sense to not to enable backface culling as described in the tutorial on cutaways.

44.0.17 Alpha Testing

OpenGL (non-ES) has a fixed-function feature, similar to the stencil buffer, to accept or discard a fragment based on its alpha value, through glAlphaFunc.

glAlphaFunc(GL_GREATER, 0.1); glEnable(GL_ALPHA_TEST);

44.0.18 Blending

The tutorial on transparency described how to render semitransparent objects with alpha blending.

Let's remind that proper transparency support requires sorting the triangles by distance to the camera, as we have to disable depth test to write "behind" a transparent triangle. To avoid sorting triangles for a single spherical object, we use culling to draw the back faces first, and then the front faces.

So take an RGBA texture, the textured spheres shaders, and this OpenGL configuration:

glDisable(GL_DEPTH_TEST); glEnable(GL_CULL_FACE); glEnable(GL_BLEND); glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); glCullFace(GL_FRONT); glutSolidSphere(1.0,30,30); glCullFace(GL_BACK); glutSolidSphere(1.0,30,30);

It should be mentioned that this particular texture image contains only alpha values of either 0 or 1. Thus, there

are relatively few fragments that receive an alpha value in between 0 and 1 due to interpolation of the alpha values of neighboring texels. It is only for those fragments that the order of rendering is important. If one accepts potential rendering artifacts for these fragments, one might improve the performance of the shader by enabling the depth test:

glEnable(GL_DEPTH_TEST);

Note that all texels with an alpha value of 0 are black in this particular texture image. In fact, the colors in this texture image are "premultiplied" with their alpha value. (Such colors are also called "opacity-weighted.") Thus, for this particular image, we should actually specify the blend equation for premultiplied colors in order to avoid another multiplication of the colors with their alpha value in the blend equation. Therefore, an improvement of the shader (for this particular texture image) is to employ the following blend specification:

glBlendFunc(GL_ONE,
GL_ONE_MINUS_SRC_ALPHA);

Semitransparent globes are often used for logos and trailers.

44.0.19 Blending with Customized Colors

We should not end this tutorial without a somewhat more practical application of the presented techniques. To the left is an image of a globe with semitransparent blue oceans, which I found on Wikimedia Commons. There is some lighting (or silhouette enhancement) going on, which I didn't try to reproduce. Instead, I only tried to reproduce the basic idea of semitransparent oceans with the following shader, which ignores the RGB colors of the texture map and replaces them by specific colors based on the alpha value:

varying vec4 texCoords; uniform sampler2D mytexture; void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); gl_FragColor = texture2D(mytexture, longitudeLatitude); if (gl_FragColor.a > 0.5) // opaque { gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0); // opaque green } else // transparent { gl_FragColor = vec4(0.0, 0.0, 0.5, 0.7); // semitransparent dark blue } }

Of course, it would be interesting to add lighting and silhouette enhancement to this shader. One could also change the opaque, green color in order to take the texture color into account, e.g. with:

gl_FragColor = vec4(0.5 * gl_FragColor.r, 2.0 * gl_FragColor.g, 0.5 * gl_FragColor.b, 1.0);

which emphasizes the green component by multiplying it with 2 and dims the red and blue components by multiplying them with 0.5 . However, this results in oversaturated green that is clamped to the maximum intensity. This can be avoided by halvening the difference of the green component to the maximum intensity 1. This difference is 1.0 - gl_FragColor.g; half of it is 0.5 * (1.0 - gl_FragColor.g) and the value corresponding to this reduced distance to the maximum intensity is: 1.0 - 0.5 * (1.0 - gl_FragColor.g). Thus, in order to avoid oversaturation of green, we could use (instead of the opaque green color):

gl_FragColor = vec4(0.5 * gl_FragColor.r, 1.0 - 0.5 * (1.0 - gl_FragColor.g), 0.5 * gl_FragColor.b, 1.0);

In practice, one has to try various possibilities for such color transformations. To this end, the use of numeric shader properties (e.g. for the factors 0.5 in the line above) is particularly useful to interactively explore the possibilities.

44.0.20 Summary

Congratulations! You have reached the end of this rather long tutorial. We have looked at:

- How discarding fragments can be combined with alpha texture maps.
- How the alpha test can be used to achieve the same effect.
- How alpha texture maps can be used for blending.
- How alpha texture maps can be used to determine colors.

44.0.21 Further Reading

If you still want to know more

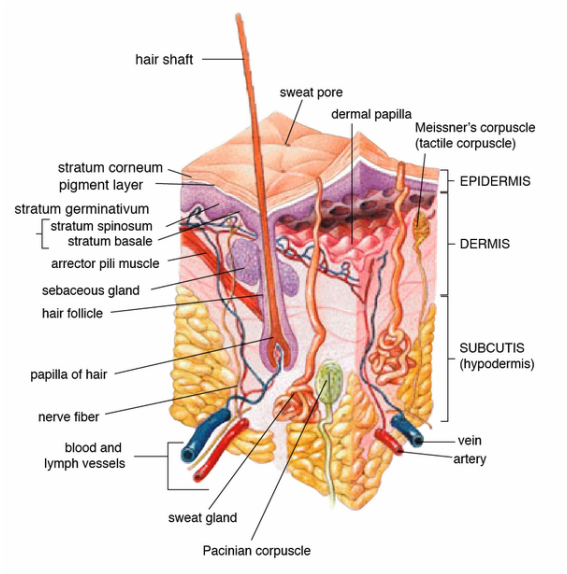
- about texturing, you should read the tutorial on textured spheres.
- about discarding fragments, you should read the tutorial on cutaways.
- about blending, you should read the tutorial on transparency.

page traffic for 90 days

< GLSL Programming/GLUT

Chapter 45

GLSL Programming/GLUT/Layers of Textures



Layers of human skin.

This tutorial introduces **multitexturing**, i.e. the use of multiple texture images in a shader.

It extends the shader code of the tutorial on textured spheres to multiple textures and shows a way of combining them. If you haven't read that tutorials, this would be a very good opportunity to read it.

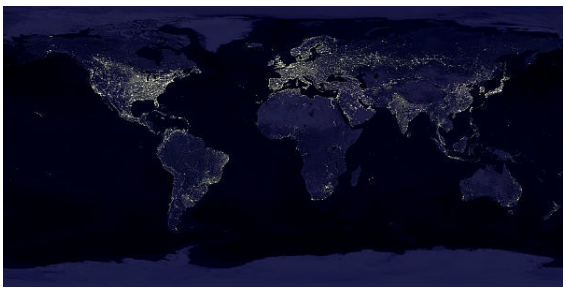
45.0.22 Layers of Surfaces

Many real surfaces (e.g. the human skin illustrated in the image to the left) consist of several layers of different colors, transparencies, reflectivities, etc. If the topmost layer is opaque and doesn't transmit any light, this doesn't really matter for rendering the surface. However, in many cases the topmost layer is (semi)transparent and therefore an accurate rendering of the surface has to take multiple layers into account.

In fact, the specular reflection that is included in the Phong reflection model (see the tutorial on specular highlights) often corresponds to a transparent layer that reflects light: sweat on human skin, wax on fruits, transparent plastics with embedded pigment particles, etc. On the other hand, the diffuse reflection corresponds to the layer(s) below the topmost transparent layer.

Lighting such layered surfaces doesn't require a geometric model of the layers: they can be represented by a single, infinitely thin polygon mesh. However, the lighting computation has to compute different reflections for different layers and has to take the transmission of light between layers into account (both when light enters the layer and when it exits the layer). Examples of this approach are included in the "Dawn" demo by Nvidia (see Chapter 3 of the book "GPU Gems", which is available online) and the "Human Head" demo by Nvidia (see Chapter 14 of the book "GPU Gems 3", which is also available online).

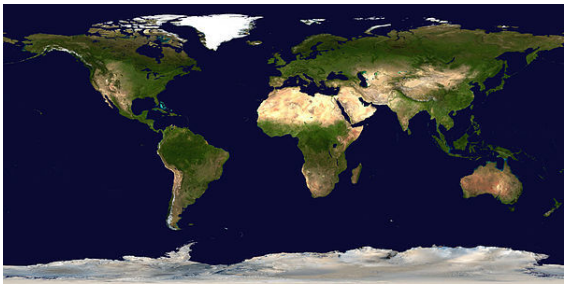
A full description of these processes is beyond the scope of this tutorial. Suffice to say that layers are often associated with texture images to specify their characteristics. Here we just show how to use two textures and one particular way of combining them. The example is in fact not related to layers and therefore illustrates that multitexturing has more applications than layers of surfaces.



Map of the unlit Earth.

45.0.23 Lit and Unlit Earth

Due to human activities, the unlit side of the Earth is not completely dark. Instead, artificial lights mark the position and extension of cities as shown in the image to the left. Therefore, diffuse lighting of the Earth should not



Map of the sunlit Earth.

just dim the texture image for the sunlit surface but actually blend it to the unlit texture image. Note that the sunlit Earth is far brighter than human-made lights on the unlit side; however, we reduce this contrast in order to show off the nighttime texture.

The shader code extends the code from the tutorial on textured spheres to two texture images and uses the computation described in the tutorial on diffuse reflection for a single, directional light source:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \max(0, \mathbf{N} \cdot \mathbf{L})$$

According to this equation, the level of diffuse lighting levelOfLighting is max(0, **N**·**L**). We then blend the colors of the daytime texture and the nighttime texture based on levelOfLighting. This could be achieved by multiplying the daytime color with levelOfLighting and multiplying the nighttime color with 1.0 - levelOfLighting before adding them to determine the fragment's color. Alternatively, the built-in GLSL function mix can be used (mix(a, b, w) = b*w + a*(1.0-w)), which is likely to be more efficient. Thus, the fragment shader could be (again with our particular computation of texture coordinates in longitudeLatitude):

void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); vec4 nighttimeColor = texture2D(mytexture, longitudeLatitude); vec4 daytimeColor = texture2D(mytexture_sunlit, longitudeLatitude); gl_FragColor = mix(nighttimeColor, daytimeColor, levelOfLighting); }

Note that this blending is very similar to the alpha blending that was discussed in the tutorial on transparency except that we perform the blending inside a fragment shader and use levelOfLighting instead of the alpha component (i.e. the opacity) of the texture that should be blended "over" the other texture. In fact, if daytimeTexture specified an alpha component (see the tutorial on transparent textures), we could use this alpha component to blend mytexture_sunlit over mytexture. This would correspond to a layer which is partially transparent on top of an opaque layer that is visible where the topmost layer is transparent.

45.0.24 Complete Shader Code

varying float levelOfLighting; varying vec4 texCoords; uniform sampler2D mytexture; uniform sampler2D mytexture_sunlit; void main(void) { vec2 longitudeLatitude = vec2((atan(texCoords.y, texCoords.x) / 3.1415926 + 1.0) * 0.5, (asin(texCoords.z) / 3.1415926 + 0.5)); vec4 nighttimeColor = texture2D(mytexture, longitudeLatitude); vec4 daytimeColor = texture2D(mytexture_sunlit, longitudeLatitude); gl_FragColor = mix(nighttimeColor, daytimeColor, levelOfLighting); }
attribute vec3 v_coord; attribute vec3 v_normal; uniform mat4 m, v, p; uniform mat3 m_3x3_inv_transp; uniform mat4 v_inv; varying float levelOfLighting; // the level of diffuse // lighting that is computed in the vertex shader varying vec4 texCoords; struct lightSource { vec4 position; vec4 diffuse; vec4 specular; float constantAttenuation, linearAttenuation, quadraticAttenuation; float spotCutoff, spotExponent; vec3 spotDirection; }; lightSource light0 = lightSource(vec4(2.0, 1.0, 1.0, 0.0), vec4(1.0, 1.0, 1.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), 0.0, 1.0, 0.0, 180.0, 0.0, vec3(0.0, 0.0, 0.0)); void main(void) { vec4 v_coord4 = vec4(v_coord, 1.0); mat4 mvp = p*v*m; vec3 normalDirection = normalize(m_3x3_inv_transp * v_normal); vec3 viewDirection = normalize(vec3(v_inv * vec4(0.0, 0.0, 0.0, 1.0) - m * v_coord4)); vec3 lightDirection; if (light0.position.w == 0.0) // directional light { lightDirection = normalize(vec3(light0.position)); } else // point or spot light (or other kind of light) { vec3 vertexToLightSource = vec3(light0.position - m * v_coord4); lightDirection = normalize(vertexToLightSource); } levelOfLighting = max(0.0, dot(normalDirection, lightDirection)); texCoords = v_coord4; gl_Position = mvp * v_coord4; }

45.0.25 Summary

Congratulations! You have reached the end of the last tutorial on basic texturing. We have looked at:

- How layers of surfaces can influence the appearance of materials (e.g. human skin, waxed fruits, plastics, etc.)
- How artificial lights on the unlit side can be taken into account when texturing a sphere representing the Earth.
- How to implement this technique in a shader.
- How this is related to blending an alpha texture over a second opaque texture.

45.0.26 Further Reading

If you still want to know more

- about basic texturing, you should read the tutorial on textured spheres.
- about diffuse reflection, you should read the tutorial on diffuse reflection.
- about alpha textures, you should read the tutorial on transparent textures.
- about advanced skin rendering, you could read Chapter 3 “Skin in the ‘Dawn’ Demo” by Curtis Beeson and Kevin Bjorke of the book “GPU Gems” by Randima Fernando (editor) published 2004 by Addison-Wesley, which is available online, and Chapter 14 “Advanced Techniques for Realistic Real-Time Skin Rendering” by Eugene d’Eon and David Luebke of the book “GPU Gems 3” by Hubert Nguyen (editor) published 2007 by Addison-Wesley, which is also available online.

page traffic for 90 days

< GLSL Programming/GLUT

Chapter 46

GLSL Programming/GLUT/Lighting of Bumpy Surfaces

"The Incredulity of Saint Thomas" by Caravaggio, 1601-1603.

This tutorial covers **normal mapping**.

It's the first of two tutorials about texturing techniques that go beyond two-dimensional surfaces (or layers of surfaces). In this tutorial, we start with normal mapping, which is a very well established technique to fake the lighting of small bumps and dents — even on coarse polygon meshes. The code of this tutorial is based on the tutorial on smooth specular highlights and the tutorial on textured spheres.

46.0.27 Perceiving Shapes Based on Lighting

The painting by Caravaggio that is depicted to the left is about the incredulity of Saint Thomas, who did not believe in Christ's resurrection until he put his finger in Christ's side. The furrowed brows of the apostles not only symbolize this incredulity but clearly convey it by means of a common facial expression. However, why do we know that their foreheads are actually furrowed instead of being painted with some light and dark lines? After all, this is just a flat painting. In fact, viewers intuitively make the assumption that these are furrowed instead of painted brows — even though the painting itself allows for both interpretations. The lesson is: bumps on smooth surfaces can often be convincingly conveyed by the lighting alone without any other cues (shadows, occlusions, parallax effects, stereo, etc.).

46.0.28 Normal Mapping

Normal mapping tries to convey bumps on smooth surfaces (i.e. coarse triangle meshes with interpolated normals) by changing the surface normal vectors according to some virtual bumps. When the lighting is computed with these modified normal vectors, viewers will often perceive the virtual bumps — even though a perfectly flat triangle has been rendered. The illusion can certainly break down (in particular at silhouettes) but in many cases it is very convincing.

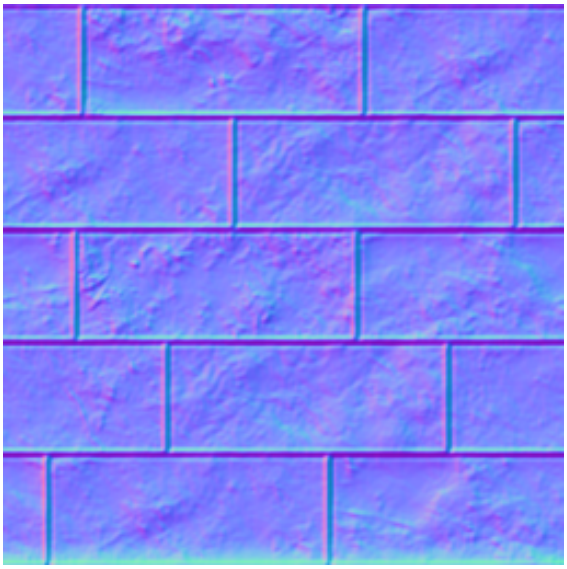
More specifically, the normal vectors that represent the virtual bumps are first **encoded** in a texture image (i.e. a normal map). A fragment shader then looks up these vectors in the texture image and computes the lighting based on them. That's about it. The problem, of course, is the encoding of the normal vectors in a texture image. There are different possibilities and the fragment shader has to be adapted to the specific encoding that was used to generate the normal map.

46.0.29 Normal Mapping

We will use the normal map to the left and write a GLSL shader to use it.

Normal maps can be tested and created with Blender (among others); see the description in the Blender 3D: Noob to Pro wikibook.

For this tutorial, you should use a cube mesh instead of the UV sphere that was used in the tutorial on textured spheres. Apart from that you can follow the same steps to assign a material and the texture image to the object. Note that you should specify a default **UV Map** in the **Properties window > Object Data tab**. Furthermore, you should specify **Coordinates > UV** in the **Properties window > Textures tab > Mapping**.



A typical example for the appearance of an encoded normal map.

When decoding the normal information, it would be best to know how the data was encoded. However, there are not so many choices; thus, even if you don't know how the normal map was encoded, a bit of experimentation can often lead to sufficiently good results. First of all, the RGB components are numbers between 0 and 1; however, they usually represent coordinates between -1 and 1 in a local surface coordinate system (since the vector is normalized, none of the coordinates can be greater than +1 or less than -1). Thus, the mapping from RGB components to coordinates of the normal vector **n** $= (n_x, n_y, n_z)$ could be:

$n_x = 2R - 1$, $n_y = 2G - 1$, and $n_z = 2B - 1$

However, the n_z coordinate is usually positive (because surface normals are not allowed to point inwards). This can be exploited by using a different mapping for n_z :

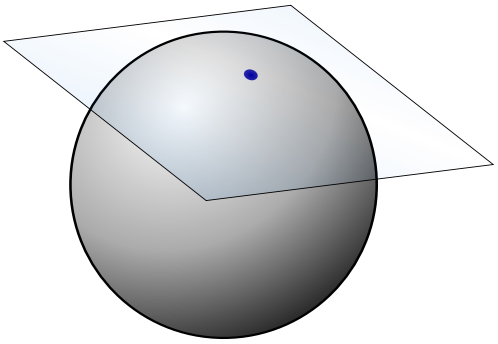
$n_x = 2R - 1$, $n_y = 2G - 1$, and $n_z = B$

If in doubt, the latter decoding should be chosen because it will never generate surface normals that point inwards. Furthermore, it is often necessary to normalize the resulting vector.

An implementation in a fragment shader that computes the normalized vector **n** $= (n_x, n_y, n_z)$ in the variable localCoords could be:

vec4 encodedNormal = texture2D(normalmap, texCoords); vec3 localCoords = 2.0 * encodedNormal.rgb - vec3(1.0);

Usually, a local surface coordinate systems for each point of the surface is used to specify normal vectors in the normal map. The z axis of this local coordinates system is given by the smooth, interpolated normal vector **N** and the $x - y$ plane is a tangent plane to the surface as illustrated in the image to the left. Specifically, the x axis



Tangent plane to a point on a sphere.

is specified by the tangent attribute **T** that the 3D engine provides to vertices. Given the x and z axis, the y axis can be computed by a cross product in the vertex shader, e.g. **B** = **T** × **N**. (The letter **B** refers to the traditional name "binormal" for this vector.)

Note that the normal vector **N** is transformed with the transpose of the inverse model-view matrix from object space to view space (because it is orthogonal to a surface; see "Applying Matrix Transformations") while the tangent vector **T** specifies a direction between points on a surface and is therefore transformed with the model-view matrix. The binormal vector **B** represents a third class of vectors which are transformed differently. (If you really want to know: the skew-symmetric matrix B corresponding to "**B**×" is transformed like a quadratic form.) Thus, the best choice is to first transform **N** and **T** to view space, and then to compute **B** in view space using the cross product of the transformed vectors.

Also note that the configuration of these axes depends on the tangent data that is provided, the encoding of the normal map, and the texture coordinates. However, the axes are practically always orthogonal and a bluish tint of the normal map indicates that the blue component is in the direction of the interpolated normal vector.

With the normalized directions **T**, **B**, and **N** in view space, we can easily form a matrix that maps any normal vector **n** of the normal map from the local surface coordinate system to view space because the columns of such a matrix are just the vectors of the axes; thus, the 3×3 matrix for the mapping of **n** to view space is:

$$\mathrm{M}_{\text{surface}\to\text{view}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

These calculations are performed by the vertex shader, for example this way:

attribute vec3 v_tangent; varying mat3 localSurface2View; // mapping from // local surface coordinates to view coordinates varying vec4 texCoords; // texture coordinates varying vec4 position; // position in view coordinates void main() { mat4 mvp = p*v*m; position = m * v_coord; // the signs and whether tangent is in

localSurface2View[1] or // localSurface2View[0] depends on the tangent attribute, texture // coordinates, and the encoding of the normal map localSurface2World[0] = normalize(vec3(m * vec4(v_tangent, 0.0))); localSurface2World[2] = normalize(m_3x3_inv_transp * v_normal); localSurface2World[1] = normalize(cross(localSurface2World[2], localSurface2World[0])); varyingNormal = normalize(m_3x3_inv_transp * v_normal); texCoords = v_texcoords; gl_Position = mvp * v_coord; }

In the fragment shader, we multiply this matrix with **n** (i.e. localCoords). For example, with this line:

vec3 normalDirection = normalize(localSurface2World * localCoords);

With the new normal vector in view space, we can compute the lighting as in the tutorial on smooth specular highlights.

46.0.30 Complete Shader Code

The complete fragment shader simply integrates all the snippets and the per-pixel lighting from the tutorial on smooth specular highlights.

attribute vec4 v_coord; attribute vec3 v_normal; attribute vec2 v_texcoords; attribute vec3 v_tangent; uniform mat4 m, v, p; uniform mat3 m_3x3_inv_transp; varying vec4 position; // position of the vertex (and fragment) in world space varying vec2 texCoords; varying mat3 localSurface2World; // mapping from local surface coordinates to world coordinates void main() { mat4 mvp = p*v*m; position = m * v_coord; // the signs and whether tangent is in localSurface2View[1] or // localSurface2View[0] depends on the tangent attribute, texture // coordinates, and the encoding of the normal map localSurface2World[0] = normalize(vec3(m * vec4(v_tangent, 0.0))); localSurface2World[2] = normalize(m_3x3_inv_transp * v_normal); localSurface2World[1] = normalize(cross(localSurface2World[2], localSurface2World[0])); texCoords = v_texcoords; gl_Position = mvp * v_coord; }
uniform mat4 m, v, p; uniform mat4 v_inv; uniform sampler2D normalmap; varying vec4 position; // position of the vertex (and fragment) in world space varying vec2 texCoords; // the texture coordinates varying mat3 localSurface2World; // mapping from local surface coordinates to world coordinates struct lightSource { vec4 position; vec4 diffuse; vec4 specular; float constantAttenuation, linearAttenuation, quadraticAttenuation; float spotCutoff, spotExponent; vec3 spotDirection; }; lightSource light0 = lightSource(vec4(0.0, 2.0, −1.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), vec4(1.0, 1.0, 1.0, 1.0), 0.0, 1.0, 0.0, 180.0, 0.0, vec3(0.0, 0.0, 0.0)); vec4 scene_ambient = vec4(0.2, 0.2, 0.2, 1.0); struct material { vec4 ambient; vec4 diffuse; vec4 specular; float shininess; }; material frontMaterial = material(vec4(0.2, 0.2, 0.2, 1.0), vec4(0.920, 0.471, 0.439, 1.0), vec4(0.870, 0.801, 0.756, 0.5), 50.0); void main() { vec4 encodedNormal = texture2D(normalmap, texCoords); vec3 localCoords = 2.0 * encodedNormal.rgb - vec3(1.0); vec3 normalDirection = normalize(localSurface2World * localCoords); vec3 viewDirection = normalize(vec3(v_inv * vec4(0.0, 0.0, 0.0, 1.0) - position)); vec3 lightDirection; float attenuation; if (0.0 == light0.position.w) // directional light? { attenuation = 1.0; // no attenuation lightDirection = normalize(vec3(light0.position)); } else // point light or spotlight (or other kind of light) { vec3 positionToLightSource = vec3(light0.position - position); float distance = length(positionToLightSource); lightDirection = normalize(positionToLightSource); attenuation = 1.0 / (light0.constantAttenuation + light0.linearAttenuation * distance + light0.quadraticAttenuation * distance * distance); if (light0.spotCutoff <= 90.0) // spotlight? { float clampedCosine = max(0.0, dot(-lightDirection, light0.spotDirection)); if (clampedCosine < cos(radians(light0.spotCutoff))) // outside of spotlight cone? { attenuation = 0.0; } else { attenuation = attenuation * pow(clampedCosine, light0.spotExponent); } } } vec3 ambientLighting = vec3(scene_ambient) * vec3(frontMaterial.ambient); vec3 diffuseReflection = attenuation * vec3(light0.diffuse) * vec3(frontMaterial.diffuse) * max(0.0, dot(normalDirection, lightDirection)); vec3 specularReflection; if (dot(normalDirection, lightDirection) < 0.0) // light source on the wrong side? { specularReflection = vec3(0.0, 0.0, 0.0); // no specular reflection } else // light source on the right side { specularReflection = attenuation * vec3(light0.specular) * vec3(frontMaterial.specular) * pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), frontMaterial.shininess); } gl_FragColor = vec4(ambientLighting + diffuseReflection + specularReflection, 1.0); }

46.0.31 Summary

Congratulations! You finished this tutorial! We have looked at:

- How human perception of shapes often relies on lighting.
- What normal mapping is.
- How to decode common normal maps.
- How a fragment shader can decode a normal map and use it for per-pixel lighting.

46.0.32 Further Reading

If you still want to know more

- about texture mapping (including tiling and offseting), you should read the tutorial on textured spheres.
- about per-pixel lighting with the Phong reflection model, you should read the tutorial on smooth specular highlights.
- about transforming normal vectors, you should read "Applying Matrix Transformations".
- about normal mapping, you could read Mark J. Kilgard: "A Practical and Robust Bump-mapping Technique for Today's GPUs", GDC 2000: Advanced OpenGL Game Development, which is available online.

page traffic for 90 days

< GLSL Programming/GLUT

46.1 Text and image sources, contributors, and licenses

46.1.1 Text

- **OpenGL Programming/Scientific OpenGL Tutorial 01** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Scientific%20OpenGL%20Tutorial%2001?oldid=2291465 *Contributors:* Beuc, QuiteUnusual, Jfmantis, Guus and QUBot
- **OpenGL Programming/Scientific OpenGL Tutorial 02** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Scientific%20OpenGL%20Tutorial%2002?oldid=2291464 *Contributors:* Beuc, QuiteUnusual, Guus and QUBot
- **OpenGL Programming/Scientific OpenGL Tutorial 03** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Scientific%20OpenGL%20Tutorial%2003?oldid=2291463 *Contributors:* Beuc, QuiteUnusual, Guus and QUBot
- **OpenGL Programming/Scientific OpenGL Tutorial 04** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Scientific%20OpenGL%20Tutorial%2004?oldid=2291462 *Contributors:* Beuc, QuiteUnusual, Guus and QUBot
- **OpenGL Programming/Scientific OpenGL Tutorial 05** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Scientific%20OpenGL%20Tutorial%2005?oldid=2666770 *Contributors:* Beuc, QuiteUnusual, Guus, QUBot and Anonymous: 1
- **OpenGL Programming/Modern OpenGL Tutorial 06** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Modern%20OpenGL%20Tutorial%2006?oldid=2684737 *Contributors:* Beuc, QuiteUnusual, Ambrevar, QUBot, Anthonybakermpls and Anonymous: 6
- **GLSL Programming/GLUT/Textured Spheres** *Source:* http://en.wikibooks.org/wiki/GLSL%20Programming/GLUT/Textured%20Spheres?oldid=2528748 *Contributors:* Beuc and Anonymous: 2
- **GLSL Programming/GLUT/Lighting Textured Surfaces** *Source:* http://en.wikibooks.org/wiki/GLSL%20Programming/GLUT/Lighting%20Textured%20Surfaces?oldid=2349798 *Contributors:* Beuc
- **GLSL Programming/GLUT/Glossy Textures** *Source:* http://en.wikibooks.org/wiki/GLSL%20Programming/GLUT/Glossy%20Textures?oldid=2350054 *Contributors:* Beuc
- **GLSL Programming/GLUT/Transparent Textures** *Source:* http://en.wikibooks.org/wiki/GLSL%20Programming/GLUT/Transparent%20Textures?oldid=2350274 *Contributors:* Beuc
- **GLSL Programming/GLUT/Layers of Textures** *Source:* http://en.wikibooks.org/wiki/GLSL%20Programming/GLUT/Layers%20of%20Textures?oldid=2627964 *Contributors:* Beuc, Denniss and Anonymous: 1
- **GLSL Programming/GLUT/Lighting of Bumpy Surfaces** *Source:* http://en.wikibooks.org/wiki/GLSL%20Programming/GLUT/Lighting%20of%20Bumpy%20Surfaces?oldid=2658578 *Contributors:* Beuc and Anonymous: 2

46.1.2 Images

- **File:Earthlights_dmsp.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/21/Earthlights_dmsp.jpg *License:* Public domain *Contributors:* http://eoimages.gsfc.nasa.gov/ve//1438/land_lights_16384.tif *Original artist:* Data courtesy Marc Imhoff of NASA GSFC and Christopher Elvidge of NOAA NGDC.
- **File:Earthmap720x360_grid.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1f/Earthmap720x360_grid.jpg *License:* Copyrighted free use *Contributors:* modification of Image:Earthmap1000x500.jpg *Original artist:* based on map by jimht at shaw dot ca
- **File:Globe.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/fa/Globe.svg *License:* Public domain *Contributors:* Own work, based on shoreline data from GSHHS ("crude" level), a public-domain source. *Original artist:* The original uploader was Augiasstallputzer at
- **File:Image_Tangent-plane.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/6/66/Image_Tangent-plane.svg *License:* Public domain *Contributors:* Transferred from en.wikipedia; Transfer was stated to be made by User:Ylebru. *Original artist:* Original uploader was Alexwright at en.wikipedia Later version(s) were uploaded by BenFrantzDale at en.wikipedia.
- **File:IntP_Brick_NormalMap.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/8/86/IntP_Brick_NormalMap.png *License:* Copyrighted free use *Contributors:* http://www.texturenwelt.de *Original artist:* DwX
- **File:Iss007e10807_darker.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/4d/Iss007e10807_darker.jpg *License:* Public domain *Contributors:* http://spaceflight.nasa.gov *Original artist:* NASA
- **File:Land_shallow_topo_2048.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/9/91/Land_shallow_topo_2048.jpg *License:* Public domain *Contributors:* http://visibleearth.nasa.gov/view.php?id=57752 *Original artist:* NASA
- **File:Land_shallow_topo_alpha_2048.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/25/Land_shallow_topo_alpha_2048.png *License:* Public domain *Contributors:*
- Land_shallow_topo_2048.jpg *Original artist:* Land_shallow_topo_2048.jpg: NASA
- **File:Le_Caravage_-_L'incrédulité_de_Saint_Thomas.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/07/Le_Caravage_-_L%27incr%C3%A9dulit%C3%A9_de_Saint_Thomas.jpg *License:* Public domain *Contributors:* artrenewal.org picture n°3757 *Original artist:* Michelangelo Merisi da Caravaggio
- **File:NASA-Apollo8-Dec24-Earthrise.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/a/a8/NASA-Apollo8-Dec24-Earthrise.jpg *License:* Public domain *Contributors:* http://www.hq.nasa.gov/office/pao/History/alsj/a410/AS8-14-2383HR.jpg *Original artist:* NASA / Bill Anders
- **File:OpenGL_Tutorial_Cube_textured.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/4c/OpenGL_Tutorial_Cube_textured.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Beuc

- **File:OpenGL_Tutorial_Gimp_export_as_C.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/ce/OpenGL_Tutorial_Gimp_export_as_C.png *License:* GPL *Contributors:* Own work *Original artist:* Gimp team
- **File:OpenGL_Tutorial_Graph_01.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/b/b6/OpenGL_Tutorial_Graph_01.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Gsliepen
- **File:OpenGL_Tutorial_Graph_02.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/22/OpenGL_Tutorial_Graph_02.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Gsliepen
- **File:OpenGL_Tutorial_Graph_03.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/6/66/OpenGL_Tutorial_Graph_03.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Gsliepen
- **File:OpenGL_Tutorial_Graph_04.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/24/OpenGL_Tutorial_Graph_04.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Gsliepen
- **File:OpenGL_Tutorial_Graph_05.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/a/af/OpenGL_Tutorial_Graph_05.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Gsliepen
- **File:OpenGL_Tutorial_Texture.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/0e/OpenGL_Tutorial_Texture.jpg *License:* GFDL *Contributors:* Own work *Original artist:* Beuc
- **File:OpenGL_Tutorial_Texture_Flipped.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/fa/OpenGL_Tutorial_Texture_Flipped.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Beuc
- **File:Skin.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/27/Skin.png *License:* Public domain *Contributors:* http://training.seer.cancer.gov/ss_module14_melanoma/images/illu_skin01.jpg (as last archived 2008-06-12 09:37:35 at http://web.archive.org/web/20080612093735/http://training.seer.cancer.gov/ss_module14_melanoma/images/illu_skin01.jpg) as displayed by Anatomy of the Skin (as last archived 2008-06-12 09:37:35 at Anatomy of the Skin) *Original artist:* US-Gov
- **File:The_Blue_Marble_4463x4163.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/ca/The_Blue_Marble_4463x4163.jpg *License:* Public domain *Contributors:*
- Smaller version: ftp://nssdcftp.gsfc.nasa.gov/miscellaneous/planetary/apollo/a17_h_148_22727.tiff as linked and thumbnailed by http://nssdc.gsfc.nasa.gov/imgcat/midres/a17_h_148_22727.gif on http://nssdc.gsfc.nasa.gov/imgcat/html/object_page/a17_h_148_22727.html and converted to JPEG and uploaded by Ed g2s 22:41, 29 December 2004 (UTC) *Original artist:* NASA. Photo taken by either Harrison Schmitt or Ron Evans (of the Apollo 17 crew).
- **File:WireSphereLowTass.MaxDZ8.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/cd/WireSphereLowTass.MaxDZ8.jpg *License:* Public domain *Contributors:* Snapshot from a program I've written. *Original artist:* MaxDZ8

46.1.3 Content license

- Creative Commons Attribution-Share Alike 3.0