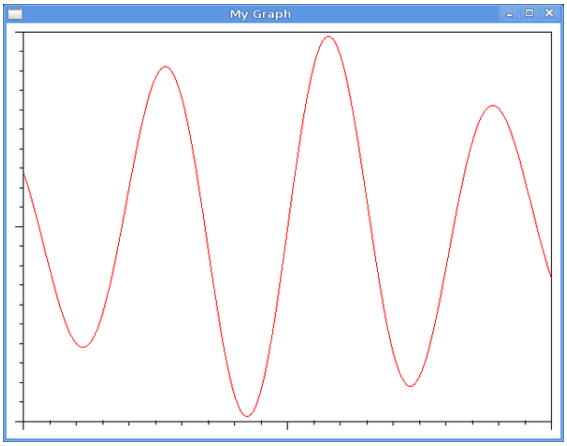
OpenGL Programming/Scientific OpenGL Tutorial 03



Borders and axes

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.

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; }

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?

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()?

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); glVertexAttribPointer(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.

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.

- Comment on this page

- Recent stats

< OpenGL Programming

Browse & download complete code

1 Text and image sources, contributors, and licenses

1.1 Text

- **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

1.2 Images

- **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

1.3 Content license

- Creative Commons Attribution-Share Alike 3.0