Graphics

Carl Kjaergaard, s123075

January 2016

1 Week 1 - Getting started

In this first week, we're introduced to WebGL and how to create basic programs that can be used for rendering.

In this week, we learn about the two types of shaders, the vertex shader and fragment shader, as well as how they communicate. This is done by implementing some different scenes in 2D that we then render.

In the first example, we set up three points and the fragment shader colors them black, we only feed them positions.

See worksheet1v1.html

In the second example we implement coloring. Now, every vertex position is associated with a color and the fragment shader interpolates based on the distance between each vertex, what the color in a certain pixel should be.

See worksheet1v2.html

In the next part, we have to implement moving the drawings. The tricky part here is to identify that in order to move the objects, we can either constantly rewrite their values in the buffers or we can make it possible to move the objects using a uniform variable that we can update readily.

See worksheet1v3.html

For the last part of worksheet 1, we have to do the same as with worksheet part 2, where we render a primitive shape and make it move. Like in part 2, the movement is made possible using a uniform variable. The trick to this part is to notice that there is no direct way of drawing a sphere and instead we approximate it by making use of a large number of triangles.

See worksheet1v4.html

2 Week 2 - Input and Interaction

In this week, we're going to be looking at how WebGL interacts with input from websites.

For part 1, we have to get the location of a mouseclick and print a dot at the location. The trick here is that we must add points to the buffer after its first initialization. This means that whenever we want to add a point, we must write these new points to the buffer and then redraw.

See worksheet2v1.html

In part 2, we must do the same thing, but now we wish to be able to add colors to the points, clear the screen and choose a clear color.

See worksheet2v2.html

In part 3, we must do the same thing as before, but now we must implement an algorithm that keeps track of points while we make up the triangle.

See worksheet2v3.html

In part 4, we add a new button to take care of adding circles for us. We can make the circles the same way we did during week 1. The secret difficulty in this exercise is to notice that you need to keep track of how you store your vertex data as well as how you draw it. Ordering this well matters a lot.

See worksheet2v4.html

3 Week 3 - Matrix Algebra

The previous weeks exercises can be considered warm up for this exercise and the following. In the previous weeks, most of the problems were solved implementing algorithms, but now we need to make use of mathematics as well and the math we use can quickly turn overwhelming if you are unprepared.

In this week, we are introduced to model-view and projection matrices, both of which are very important when it comes to 3d modelling in WebGL. In worksheet3v1.html, we see the first example of actual 3d rendering using webgl.

In worksheet3v2.html, we have draw the same cube but with three different view matrices.

We make use of two transformation vertices called the modelView matrix, which is what we use to move the camera around the scene and the projection matrix, which we use to define the clip-space of our scene, that is, what we render in front of the camera.

That is, for the point v1, we use the modelView matrix M and the projection matrix P. First, we multiply M with P so that we get our modelViewProjection matrix MVP = M*P. Then, when we have this matrix, we multiply the vector v1 with MVP, so that we get our point v2 = MVP*v1, which we then use to draw the scene.

For the one-point cube, the equation for our CTM is:
P = [[1,0,0,-0.5], [0,1,0,-0.5], [0,0,1,-3], [0,0,0,1]]
M = [[2.414,0,0,0],[0,2.414,0,0],[0,0,-1,-0.0002],[0,0,-1,0]]
CTM = M*P

For the two-point cube, the equation is:
P = [[0.894,0,0.447,-0.447],[0,1,0,-0.5],[-0.447,0,0.894,-3.130],[0,0,0,1]]
CTM = M*P

and finally, for the three-point cube
P = [[0.894, 0, 0.447, -0.447],[-0.182,0.912,0.365,-0.365],[-0.408,-0.408,0.816,-3.265],[0,0,0,1]]
CTM = M*P

4 Week 4 - Illumination

In this week, we will be making much more extensive use of the shaders to calculate the colors we will use based on two lighting models. The first thing we do is set up a model that is more complex than a cube in terms of the normals that it requires.

In part 1, we set up a 3d sphere with the ability to control how well defined the sphere is.

See worksheet4v1.html

In the next part, we make the color on the sphere uneven by having each point on the sphere be colored according to what position it has. We notice that the coloring doesn't seem "right" and that is because we draw too much of the circle. When dealing with 3D, we don't expect to see items that are behind other items, and since we have several points that are behind each other, we draw a lot of points that we shouldn't draw.

In order to avoid this, we tell WebGL to make use of a depth buffer and backface culling.

See worksheet4v2.html

The next three steps involve adding shading to the sphere. The first thing we do is add diffuse shading calculated in the vertices. This coloring is then interpolated between each of the vertices in the fragment shader and we get Goraud shading.

See worksheet4v3.html

In part 3, we only included diffuse shading, but we can improve on our shading model by including ambient and specular colors. We include also four sliders in our webpage, that allows us to freely change the values that we use to compute these different parts of the shading.

See worksheet4v4.html

For the final part of this week, we have to implement the shading in the fragment shader, meaning that instead of interpolating shading values between vertices, we calculate the shading value in each fragment.
See worksheet4v5.html

Finally, we have to answer some questions:
a)
The big difference is that phong lighting describes the set of equations we can use to calculate the shading values, whereas phong shading then describes more or less how to do it. The phong lighting works by calculating the ambient, diffuse and specular values of reflected light, given some material properties, a light position, light emission properties as well as a surface normal.
Phong shading is then the technique of calculating this value by interpolating normals for each pixel given that each vertex has a known normal.

b)
Flat shading reuses the same color values across an entire polygon, Goraud shading interpolates different shading values for each pixel given that the shading is calculated in the vertices and Phong shading calculates shading per pixel.
Performance is one big difference, given that the more calculations we have to perform, the slower the rendering process is.
On the other hand, the more precise we make our calculations, the more natural it will look.

c)
The difference between directional light and point light is that directional light has one "direction". Directional lights will thus always have the same angle of incidence on two surfaces in the same plane no matter where they are. This is a simpler lighting model than the pointlight, because pointlights shine equally in all directions from where they are put.
This makes calculating the angle of incidence harder, because we can no longer assume that it is the same for equal surfaces.

d)
It depends on how the question is interpreted. If we look on the backside of an object, we are going to see different parts of the object and therefore different shading and likewise different viewing angles on the same spot will cause the reflectance to change behaviour, but the shading properties of the object never changes.
So, the angle of the observer does matter to how the object is perceived but the object itself is not influenced by it.

e)

The sphere looks better. We no longer have the very bright area on our sphere that previously showed where the light was hitting the sphere. This can also be observed by pulling the slider for the specular coefficient to 0.

f)

We reduce the effect of the specular area. There is an illustration that shows this effect on page 317. If we set the shininess coefficient to 1, we will have all the diffuse area be specular.

g)

In the view space. We multiply the position of the vertices with the modelview matrix before we perform our calculations on it.

5 Week 5 - Loading .obj files

In this exercise, we have to load a premade model composed of triangle meshes. In this exercise, we are introduced to a way of storing mesh information for complex meshes as well as how to draw using the drawElements function.

The result of loading the file "teapot.obj" can be seen in worksheet5.html

The surface normals are obtained by loading the file in teapot.obj, where the normals are associated with a pixel. In the .obj file, each vertex can have an associated normal. A description of these normals can be found on page 417 in "Load and display" of the webgl programming guide.

One of the advantages of triangle meshes is that we can smooth them out much in the same way we did when we created our sphere from a tetrahedron. Each triangle surface can be split up and transformed into new triangles, that can then also be split up into even more triangles. Using this technique, we can create very smooth surfaces.

6 Week 6 - Textures

This week has two parts that are somewhat different. In the first part, we render a texture on a basic ground quad using different methods. In the second part, we render a texture on a sphere while adding shading.

For the first part, we see that clamping causes the texture colors to stretch when they reach outside the texture coordinates, and that repeated patterns allow us to paint the entire quad in chess texture even if the quad seems too small.

The nearest filtering looks very good closer to the screen, but becomes very disturbed further away, the linear filtering looks blurry up close but much better further away and the mipmapping removes most of the problems of both by making the edge of the square "grey" presumably because it blends the colors of black and white so much at that point. Overall, the mipmap looks best.

See worksheet6v1.html

For the next part, we have to render a textured sphere with a very large texture. The size of the texture means that our calculations become very imprecise and artefacts star to appear on the sphere. My preferred choice of texture filtering was mipmapping, because while there seem to be issues on the edges of the texture, the mountain ranges have been smoothed out considerably. This is presumably because of how mipmapping "blurs out" a lot of detail when the highest resolution texture becomes too complex, so instead we use a smaller resolution one where the color differences are not as jarring.

See worksheet6v2.html

7 Week 7 - Projection shadows

For this week, we'll be looking at implementing shadows. The way we implement shadows in rasterisation is that we render the scene in one way or another and use the information we gain from the rendering to determine the shadows. In this week, we will be implementing shadows by drawing our shadowed objects from the point of view of the light and use this rendering to draw on top of already existing geometry.

The difficulty in this is that we have to transform our modelview matrix so that we are rendering it from the light position, but translate it back to coordinates in our own view space.

We also added blending, which is where we look at the alpha values that are drawn and use these to determine whether colors should "blend". This allows us to make the shadows have an ambient coloring rather than hard black.

The result can be seen in worksheet7.html.

8 Week 8 - Shadow mapping

I was not able to fully implement this week, as it seems that the shadows created by shadow mapping disappear from time to time. I write to the shadowmap, read from it to determine whether I am in shadow or not, but something mishappens and causes the shadows to disappear every part of a rotation.

This week had a lot of things that needed to be implemented. The first thing that we had to do was some programming exercises, in that we had to allow rendering using two different sets of shaders as well as different drawing modes.

While there was not necessarily any reason to divide up the programs into 2 parts, one for each object, seeing as we can simply change the way we draw by using uniform variables, having a complex shader that has several vastly different drawing modes can presumably lead to a lot of misplaced effort.

When the two shader programs are set up, we need to implement a third shader as well as a a new set of buffers. The third shader is responsible for drawing the distances of the objects in the scene that is rendered from the point of view of the light. The distances are then written into a texture, that is send to the object that has to actually render the shadow. If the distance to a certain pixel from the light is not the same value as what we rendered into our texture, then the pixel is in shadow and we make it darker.

One big disadvantage that shadow projections will have that will ultimately leave them inferior to shadow maps is that shadow projections are projected to a single plane. For more complex geometry, this would mean that we would have to calculate all these different planes as well as project each shadow onto them and all of this would have to be done in the right order.

With shadow mapping, a single rendering into the shadow map will allow us to determine whether a pixel is in shadow or not by looking in the shadow map and comparing distances there.

Therein lies a disadvantage to shadow maps however, as each visible object in the scene would have to personally check whether part of it is in shadow or not, where as shadow projections allow us to make shadows without requiring any extra effort from the rest of the geometry.

What is ultimately the best choice is uncertain, but considering how relaxed about resources we have been in this course, it seems unlikely that adding a texture lookup adds much weight.

9 Week 9 - Reflections

In this week, we're focusing on reflections. Compared to shadow maps, reflections are a somewhat simpler thing to determine the location of, but choosing when and how to actually render pixels becomes problematic as we cannot make use of the depth buffer. In order to alleviate this issue, we make use of the stencil buffer as well as a modified view frustum that allows us to cut off unwanted parts of the reflection.

The first thing we have to do when reflecting objects is to create the reflection. Because we make use of rasterisation, we have to do this by rendering the scene one extra time. It becomes very evident here, because of the way we create the illusion of reflection, that rasterisation is inferior to ray tracing in terms of reflection, because each reflecting material will have to be processed on its own.

Creating the reflection matrix allows us to mirror each point above the reflecting plane to a location below it that is equally distant from the plane. In order to mirror the lighting, we also multiply the light position with the reflection matrix, although this means that the shading will not be reflected 1-to-1.

At this point, we notice that there are two things that happens that should

not be happening. The first is that the reflected teapot appears outside of the reflecting plane and the second is that the reflection happens when it should not be happening.

The first problem is solved by using a somewhat familiar method, which is to draw into a buffer that helps us determine whether we should draw certain pixels or not. What we are doing with the stencil buffer is essentially this:

First, we draw the reflecting plane after having disallowed drawing colors. The stencil buffer fills up with values equal to the depth of the reflecting plane. Then we draw the scene again, but this time we only allow filling in the highest possible depth value in the depth buffer for places where the reflecting plane is present and the smallest value where it is not. The result is that we can now draw the reflected object and it will always pass the depth test within the reflecting plane and always fail outside it.

This only solved one of our problems however, as we also need to make sure that we don't draw above the plane. We do this by changing the near clip plane to the reflecting plane. This means that the reflection will not be drawn above it because it will be "behind" the camera plane.

See worksheet9.html

10 Week 10 - Environment mapping

In this exercise, we deal with environment mapping, which in webgl is implemented using cubemaps. The cubemap accepts a 3d vec, instead of the normal 2d one and uses this to determine a color value. Implementing this with sphere normals can be seen in worksheet10v1.html.

For the second part, we have to do it on a quad. Unfortunately, the quad does not have normals that can be used to immediately determine the directional values that we wish to send the map. Instead, we have to go backwards to determine the direction we wish to use.

The idea behind going backwards is a bit strange, because we already have some usable world directions, which is essentially our world coordinates. In order to reverse our position, we make reverse transformations. Normally when we multiply our modelviewprojection matrix on our position, we go from world-space coordinates to clip space, which is what we then render. In order to go from clip-space to world-space coordinates, we have to multiply the inverse of these matrices to the transformed vector to transform it back, as a matrix multiplied with its inverse is equal to the identity matrix, meaning that we will essentially have gone back in position.

We also have to look up the reflected direction instead of the direction of the normal on the mirror ball. To do this, we make use of the normals that we already have, as well as the position of the eye to calculate the reflected direction. The resulting reflected direction is then what we use to look up in the environment map.

This can be seen in worksheet10v2.html.

For the final exercise, we have to implement bump-mapping. This is done by loading the bump-map and reading the color values as xyz values of a vector that is tangent to the actual normal and we then perform a rotation on the vector so that we get a new value for our normal. Doing this gives us a very cool "bumpy" sphere.

This can be seen in worksheet10v3.html

11 Project - Vertex Movement

For the project, I have decided to implement a system that lets you pick certain vertices as well as move them around. In order to do this, we need two things:

a) A way to determine what vertex was clicked on b) A way to move the vertex

Determining the selected vertex can be done by having each vertex render using a color that corresponds to a certain ID value but how that ID value should be interpreted, among other things, can be difficult to determine and should correspond to how the program is set up. What we have been used to until now is linearly drawing arrays of vertex data, with the exception of when we draw the teapot, so this is what I have chosen to use here as well.

One thing to notice is that one vertex may be part of many triangles in this setup, like how a corner in a cube is part of many triangles, so we associate each "vertex" like that to a table of vertex information, that shows at what offsets we can find the points of the corner. Then, when we want to "move" the vertex, we rewrite the values at the locations where the vertex is present.

Currently, the way the vertexInfo structures are created has to be custom for each primitive, but can theoretically be made to fit geometry regardless of shape.

One issue that is worth mentioning is that when you draw to the framebuffer, the framebuffer does not actually make use of the depth buffer, so in order to implement a depth buffer, one has to associate it with a renderbuffer.

One more thing to note is that while we are currently limited to at most 256 vertices because we only use one color byte, we can extend this to use all four colour bytes for a nigh infinite amount of different vertex identities.