Double Buffering The fragment shader sets a pixel in a texture (image). All the fragments are done in parallel, or as parallel as possible. Once all the fragments are processed, the resulting texture is swapped with the one we currently see on the screen. Then that texture is cleared and the process begins again. This is called "double buffering". By using two texture buffers, the system avoids showing us the drawing "in progress".

Depth Map Multiple fragments correspond to the same pixel. How does it decide on the final color? The fragments might be processed in any order. Who wins? Thanks to double buffering, we won't ever see the wrong color, but what if that fragment is set later than the correct color? The answer is a hidden color corresponding to the depth. The z is converted to the range 0 to 1 with 1 the initial color. Whenever the fragment shader sets a pixel, it checks to see if the new depth (color) is less than the old depth (color). If so, it sets the pixel with the new color and depth.

Texture Maps The GPU has to be able to send a depth buffer outward, so we can see it on the screen. It also has to be able to read and write texture maps that it creates. For texture mapping, we send other textures to the GPU and allowed to fragment shader to read them using texture coordinates.

"New" Technology The GPU can read and write textures. Instead of just having the texture it is currently working on plus the ones we give it (Mandill and Moon, etc.), it can read a different texture that it created. That must means having more than two display buffers. No big deal.

(What we do in the) Shadows A fragment is in shadow if the light cannot see it. Why can't the light see it? Render the scene from the light's point of view. The light cannot see a fragment if there is another fragment in the light's clip box that has the same x and y but a lesser z. In other words, something else is in the way of the light.

Remember that we figure out the closest fragment at each x and y by storing the z for each visible fragment. If another fragment has closer z, then we store it instead. We also store its color, which is usually what matters, but now we only care about the z because we don't actually want to *see* the scene from the light's point of view. We just want to know *which* fragments it sees.

We will make a texture that is the z. Since x, y, and z go from -1 to 1 but texture coordinates and colors go from 0 to 1, we need to multiply by 0.5 and add 0.5. Given a fragment at x,y,z in the light's clip box, we look up the value of z in the texture. If z equals the value in the texture, then the light source sees it, so it is lit. If fragment's z is greater than texture value, then the light source does not see it (because something is closer at that pixel), so that fragment is in shadow.

Look at http://media.tojicode.com/webgl-samples/depth-texture.html for a nice example. Notice how the figure casts a shadow on itself. Just doing the floor would not be as big a deal.

We will do a simpler example with two objects, a sphere and a cube. This has nothing to do with the actual shadow calculations. We just need two objects because our objects are convex so they can't cast a shadow on themselves. Each object has its own model2world. We can calculate model2world for the sphere in the usual manner (product of which three matrices?). The cube will be fixed in the world (no rotation or translation by dragging) and we will call its matrix cube2world. Instead of passing model2clip to the vertex shader, we will pass model2world or cube2world and world2clip. Either way, the vertex shader will call it model2world and multiply these times the point:

- p = world2clip * (model2world * position)

We set up a view2clip matrix for the light source. We will use a square texture (1024 by 1024) to record what the light "sees" so the aspect ratio is 1. That gives us a different (simple) proj2clipL and clip2projL. Combining these with the existing view2proj gives us view2clipL, which is view to clip for the light source. We also set up a world2viewL matrix for the light source. Again, that is easy because we just call lookAt with the position of the light instead of the position of the eye. The product of view2clipL and world2viewL is world2clipL, the world to clip matrix for the light source.

We render the models from the point of view of the light, passing model2world or cube2world as (uniform) model2world and passing world2clipL as world2clip. The depth of each point is recorded in a texture map. This texture will be available to the regular rendering for the eye as depthTex.

Next, we render from the eye as usual using model2world or cube2world and world2clip. Also pass in world2clipL. We know a position ends up in the light's clip box at

- positionL = world2clipL * (model2world * position);

We can homogenize this:

- positionL = positionL / positionL.w;

This is a point in the clip box with coordinates from -1 to 1. To get the coordinates from 0 to 1, use

- positionL * 0.5 + 0.5;

depthTexXY are the x and y and depth is the z. depthTexXY are the texture coordinates where the first fs stored the depth. depth is the depth of that point.

Look up d = texture2D(depthTex, depthTexXY). If it "equals" depth, then that fragment is lit. However, if d is definitely less than depth, then the fragment is in shadow. I say "equals" because there is rounding error and interpolation error, so you cannot expect the two values to be exactly equal. So definitely less means less by at least DEPTH_EPSILON = 0.1.