Omnidirectional Shadow Mapping

Paul Stocker, Wei Wang

January 27, 2016

Abstract

This report discusses the implementation of a movable light source inside a scene in WebGL. The light source emits a fading light, creating shadows on nearby objects. To create the shadows we use shadown mapping for a cube map, omnidirectional shadow mapping.

1 Introduction

The goal of the project was to create a scene containing a movable light source, shining light onto some geometric objects, as seen in Figure 1. The main challenges encountered during the programming where the movement of the light, creating a light radius and creating the shadow. We chose the light source to be a bright sphere surrounded by a pyramid, a cylinder and a cube. Furthermore, we drew a background quad and a ground with texture to give the scene a more interesting look. The scene is introduced in more details in Section 2.

In Section 3 we discuss how we implemented mouse interaction on the canvas in order to move the light sphere. Then, in Section 4, we show how we created a glow surrounding the light source and decaying in the distance, opposite to the light source known from the exercises that was placed outside the canvas shining uniform light on everything. Finally, in Section 5, we create shadows using shadow mapping on a cubemap, often referred to as omnidirectional shadow mapping.

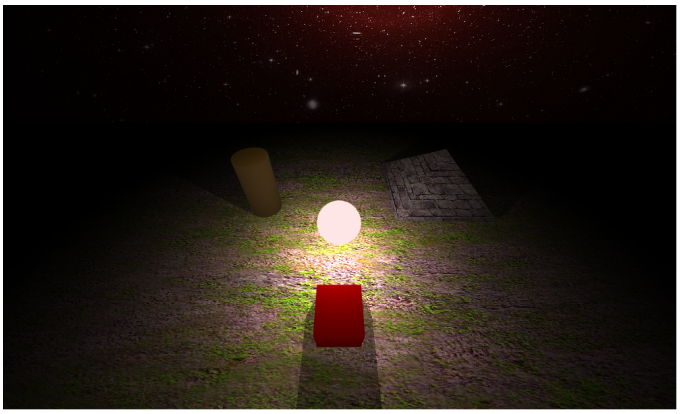

Figure 1: Goal and result of the project

2 Setting the Scene

To create the scene we used two different shaders, one for objects with texture and one for objects with color. Since we knew already that we will have to deal with a

frame buffer object (FBO) for shadow mapping (see Section 5) we stored all vertices in one array, by concatenating. This made it easier to buffer the scene into the FBO, later on.

We started out with the objects we wanted texture on, this is the ground, background and pyramid. The vertices and texture coordinates were hard coded and buffered into the first shader, the texture shader. The background quad is drawn in clip space coordinates in order to fill the background. To load the texture images we used the class Image, which loads an image using the method .url and after doing so calls the function defined in .callback to continue the program.

Next, we created the objects we wanted to draw with the color shader. The sphere was created by refining triangles and normalizing there distance to the origin, as learned in class. The cylinder was created by drawing two circles, using the same refinement technique, and connecting those vertices to create the side of the cylinder. The cube was, again, hard coded.

3 Mouse Interaction

We wanted to be able to move the light sphere using the mouse. Therefore, we added an event listener that would create a coordinate system if the mouse is clicked ('mousedown') on the canvas. At the point of the click the courser would be the origin of the system. If the mouse is moved we would update the model matrix of the sphere by a ratio of that movement. We wanted the sphere to move on the ground, hence we used the function 'translate' provided by [1] to create a translation matrix in the x-z-plain. This is seen in the code below.

```
// Mouse interaction
function mouse(){
   canvas.addEventListener("mousedown", function(event){
      rect = canvas.getBoundingClientRect();
      originX = event.clientX - rect.left;
      originY = event.clientY - rect.top;
       canvas.addEventListener("mousemove",mouseMoveHandler,false);
      }
   )

   canvas.addEventListener("mouseup", function(event) {
        canvas.removeEventListener('mousemove',mouseMoveHandler,false);
    });

   function mouseMoveHandler(ev) {
        offsetX = (ev.clientX - originX);
        offsetY = (ev.clientY - originY);
      MouseModelMatrix = mult(translate(vec3(offsetX/(30*screenwidth), 0.0,
          offsetY/(30*screenhight))), MouseModelMatrix);
    }
}
```

4 Light Attenuation

We wanted the light emitted by the sphere to decay in the distance to create a glowing effect, as seen in Figure 1. For this we calculated the distance of each point to the light source while drawing in the fragment shaders (texture and color shader) and reduced the color of the fragment by a factor. This factor is referred

to as attenuation and in [2] we found the formula

$$f_{att} = \frac{1}{(\frac{d}{r}+1)^2}$$

where d is the distance between the light source (we chose the center of the sphere) and the point P being shaded, and r is the light sphere radius. We used a cutoff to get pitch black areas and introduced the scalar lightRadius that achieve the wanted light intensity. The function shown below returns the correct light color and intensity.

```
vec3 DirectIllumination(vec3 P, vec3 lightCentre, float r, vec3
    lightColour, float lightRadius, float cutoff){
  // calc distance to light source
  float distance = length( lightCentre - P );
  float d = max(distance - r, 0.0)/lightRadius;
  L /= distance;

  // calculate basic attenuation
  float attenuation = 1.0 / pow((d/r + 1.0),2);

  // apply cutoff
  attenuation = (attenuation - cutoff) / (1.0 - cutoff);
  attenuation = max(attenuation, 0.0);

  return lightColour * attenuation;
}
```

5 Omnidirectional Shadow Mapping

The basic idea of shadow mapping is to draw the whole scene from the view point of the light source. This is done off canvas into a frame buffer object (FBO) which requires its own shader. From the light source's view we cannot see any shadow, this means that every point we cannot see has to be drawn in shadow.

To communicate from the FBO to the other shaders which points to draw in shadow and which not, we abuse a texture by storing the depth values as color values, this texture is referred to as *shadow map*. From the light source's point of view we store the depth value of each *visible* point. Later, when drawing the scene, we compare this depth value with the depth value of each point we draw. If the depth value is bigger then the stored depth value we know that this point has to be drawn in shadow. This is shown in Figure 2.

It is important to remember that we can only compare depth values from the light source's point of view. Therefore, while drawing, we need to shift each point in order to compare it to the values we got from the FBO, which are in $[0, 1]$.

5.1 Shadow Cube Map

The above process was already done during the exercises, however, in order to deal with the light source inside the scene we cannot use a 2D texture to store the depth map, as instead we need a cube map. The basic idea is to imagine the light source in a cube, and as we look at the scene from the light source's point of view we need to look 'out' of each side of the cube to see the full scene around us.

The main issues to keep track of during the rendering into the FBO are to: set the perspective angle to 90°, set the viewport so a $2^n \times 2^n$ square, set the viewpoint

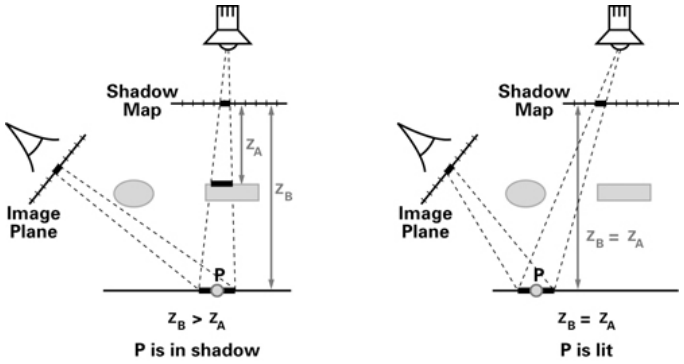

Figure 2: Depth values stored in shadow map. Figure from [3]. Left: the point P is in shadow since the depth values are not equal. Right: the point P is lit since it is visible from the light source's point of view and therefore the depth values are equal.

into the light source and look down each axis into positive and negative direction. Hence, we need to render six times, onto each face of the cubemap, as we look at the different directions.

5.2 Look Up

In order to perform the correct comparison of the depth value stored in the cubemap and the depth value of the point P we are drawing we need to shift the point P into the light source's view. This is done by finding the vector from the light source to the point P, seen in the first line of the program given below. Next, we need to look up the depth value for comparison in the cubemap. Since the light source is a sphere we can simply use the normalized vector to perform the look up. The depth value of the point P is given by its largest component. Lastly, we need to adjust the depth of the point P to values in $[0, 1]$ as used in the texture. This is done by using the formula given in [5],

$$z' = \left(-\frac{2 \cdot near \cdot far}{z(far - near)} + \frac{far - near}{far + near} + 1 \right) 0.5$$

where $near$ and far are the values of the near and far plain. This is done in the code below.

```
// shadow mapping
vec3 vPositionFromLight = (P.xyz/P.w-lightSource);
vec3 shadowCoord = normalize(vPositionFromLight.xyz);
vec4 rgbaDepth = textureCube(depthCubeMap, shadowCoord);
float localZcomp = max(abs(vPositionFromLight.x),
    max(abs(vPositionFromLight.y), abs(vPositionFromLight.z)));
float depth = rgbaDepth.r;
float objdepth =
    (1.0/localZcomp*(-2.0*5.0*0.1/(5.0-0.1))+(5.0+0.1)/(5.0-0.1)+1.0)*0.5;
float visibility = ( objdepth > depth + 0.005) ? 0.6 : 1.0;
```

References

[1] Edward Angel, Dave Shreiner: *Interactive Computer Graphics. A Top-Down Approach with WebGL* Pearson (2014)

[2] https://imdoingitwrong.wordpress.com/2011/01/31/light-attenuation/, 27.1.2016

[3] http://http.developer.nvidia.com/GPUGems/gpugems_ch12.html, 27.1.2016

[4] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html, 27.1.2016

[5] https://en.wikipedia.org/wiki/Z-buffering#Mathematics, 27.1.2016