**IMPLEMENTAION OF**

**SHADOW MAPPING**

**Subject:** Computer Graphics

CS 6366

**Prepared by:** Arvind Pandiyan <axp141630>

Nagabharan Nagendran <nxn141730>

5th May, 2015

**PROBLEM SUMMARY**

Shadows add a significant amount of realism to a scene by conveying information about the geometry of objects and the location and characteristics of light. Shadow mapping or projective shadowing is a process by which shadows are added to 3D computer graphics. Shadows is often a problem for polygon drawing engines such as OpenGL and DirectX.

The simple solution for shadows is to cast each object onto a plane and draw the resulting figure in black. But this solution does not allow objects to cast shadows on themselves nor on other objects. If the plane isn't infinite then the shadow will float in the air when the object reaches the edge of the plane.

There are several methods for shadow rendering such as ray tracing, shadow mapping, shadow volumes etc. We studied each of the solutions and realized that the algorithm for shadow volumes could actually be implemented rather easily in OpenGL because one of the major drawbacks was an easy problem for OpenGL to solve.

The goal of this project is to understand the concepts of shadow mapping [Williams 1978] and shadow volumes [Tim 1991], since these are the most commonly used real time shadow algorithms, and also to implement the same.

**DESCRIPTION OF WORK**

**SHADOW MAPPING:**

The basic shadow-map algorithm consists in two passes. First, the scene is rendered from the point of view of the light. Only the depth of each fragment is computed. Next, the scene is rendered as usual, but with an extra test to see it the current fragment is in the shadow.

The “being in the shadow” test is actually quite simple. If the current sample is further from the light than the shadow-map at the same point, this means that the scene contains an object that is closer to the light. In other words, the current fragment is in the shadow.

The shadow-map is rendered off-screen via an openGL Framebuffer Object (FBO). Only depth value are saved during rendition, there is no color texture bound to the FBO and color writes are disabled via glColorMask(GL\_FALSE, GL\_FALSE, GL\_FALSE, GL\_FALSE).

We transform the vertex with the camera matrices, the same vertex with the light POV matrix and we get the fragment color. The shadow variable holds the shadowed test result. As you see, the goal is to compare the z value (in light POV) of the vertex rendered, with what was rendered to the shadow-map.

**SHADOW VOLUME:**

A shadow volume is basically a cone capped on both ends. The origin of the cone is the light source. The cone is initially capped at the silhouette of an object and capped again at some other arbitrary place as long as the second capping results in a cone that passes though the ground plane. The silhouette of an object is defined to be any edge of the object such that one side of the edge faces the light and the other side does not. Implemented in OpenGL using the stencil buffer by the following algorithm:

1. Draw Scene Normally
2. Turn off depth buffer and color buffer.
3. Enable writing to stencil buffer.
4. Draw Shadow polygons facing the eye. Then increment values in stencil buffer.
5. Draw Shadow polygons not facing the eye. Then decrement values in stencil buffer.
6. Disable writing to stencil buffer.
7. Turn on depth buffer and color buffer.
8. Draw Scene again but with the stencil buffer test and the lights off.

**CHALLENGES FACED:**

There were multiple challenges and roadblocks encountered during development. The first and most significant challenge was development of a framework that rendered the model and lighting in proper order.

The next challenge was altering the framework from using basic GL commands to using shaders. This required understanding how the application level interacts with the shader level and understanding the rendering pipeline more in depth than we did for the first version of the framework.

In order to test a point against the depth map, its position in the scene coordinates must be transformed into the equivalent position as seen by the light. The first implementation of the shadow mapping involved calculation of the above transformation matrix. These values were passed to the fragment shader (and interpolated along the way) and then displayed. The result was not the expected, smooth result. In fact it was quite ugly and not correct. This required us to realize that the values I computed in the transformation incorrect and instead we used the bias matrix multiplication to map those −1 to 1 values to 0 to 1, which are more usual coordinates for depth map (texture map) lookup. This scaling can be done before the perspective division, and is easily folded into the previous transformation calculation by multiplying that matrix with the following:

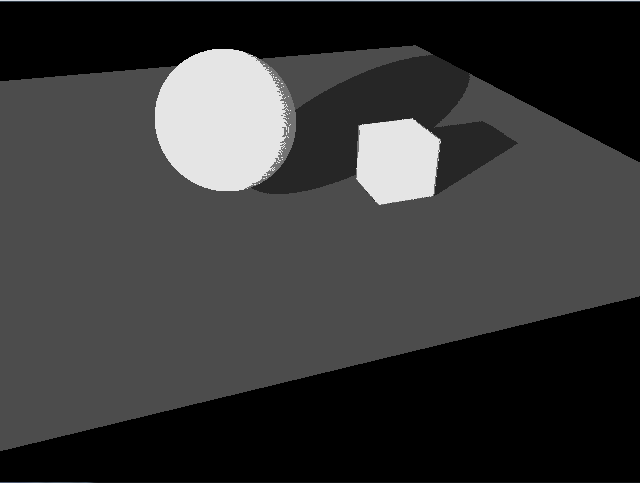
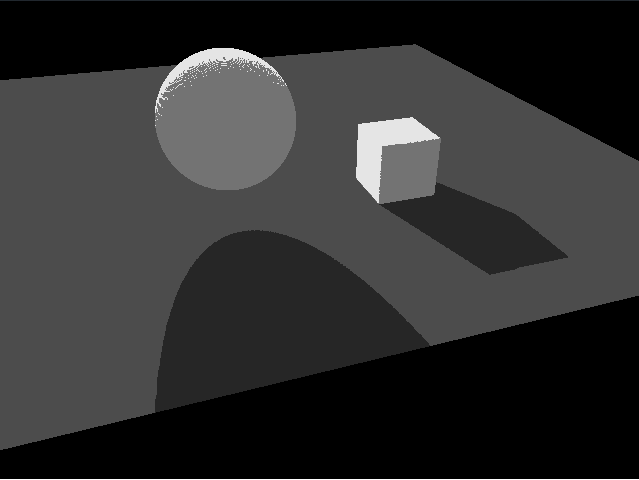

\begin{bmatrix}
0.5 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0.5 \\
0 & 0 & 0.5 & 0.5 \\
0 & 0 & 0 & 1 \end{bmatrix}


After using this transformation matrix the implementation became much better.

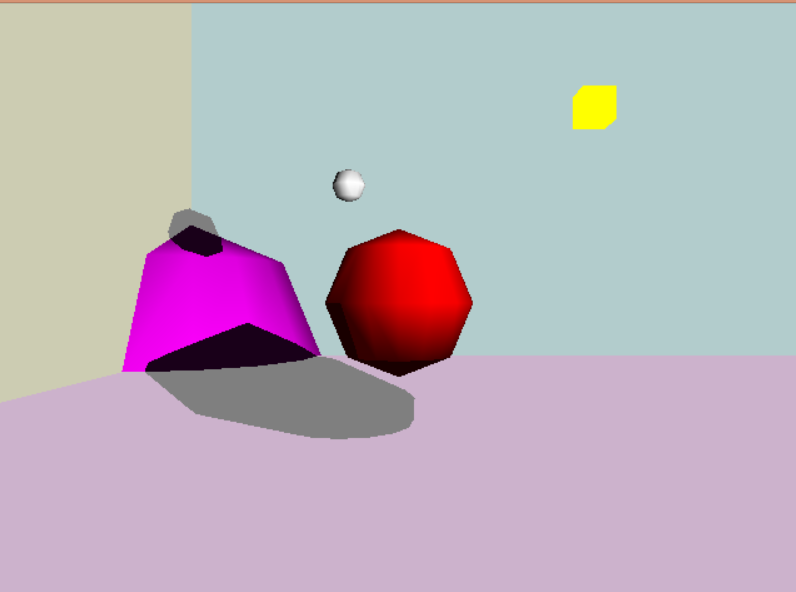
In the case of shadow volumes the implementation was straight forward using the algorithm described by Tim. There was no necessity for using shaders since we used the stencil buffer to generate the required shadows.

**RESULTS**

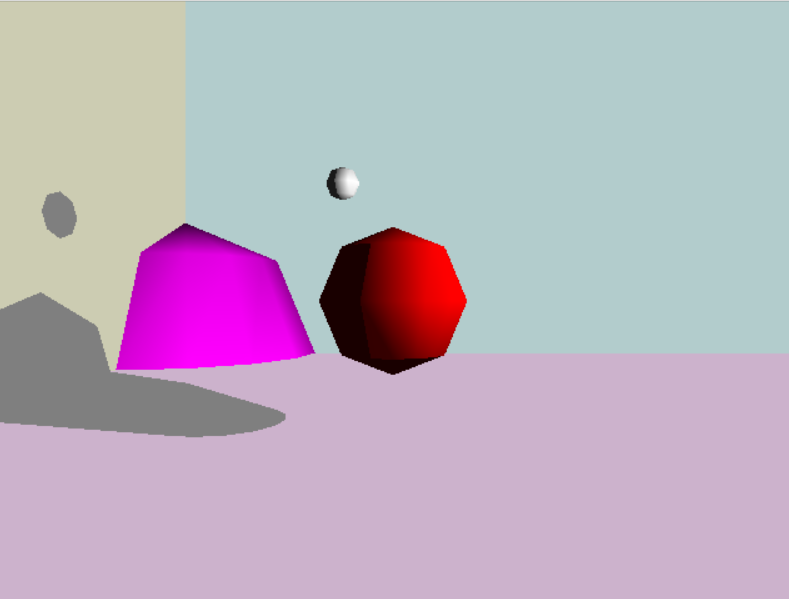
Shadow mapping:



Shadow volume:



After Changing position of light source



**ANALYSIS & CONCLUSION**

The initial Goals set out were:

• Understand exactly what shadows means; what does the effect produce?

• Understand the different concepts required: texturing, mapping, coloring, lighting, etc.

• This project will build on the skills obtained from programming assignment one and two.

• Use C++ and OpenGL to render models with shadows concurrently.

• Create appropriate shaders using GLSL to perform shading

Shadows were created successfully using shadow mapping and shadow volumes. Only basic structures were used for the objects. We couldn’t integrate the models given in class for the scene due to time constraints. Self-shadowing occurs because of the depth buffer limited precision. This is also known as Z-fighting. This only affect polygons facing the light because it's what was rendered to the shadow-map.

In case of shadow volumes, objects can cast shadows on anything including themselves or on curved surfaces. The algorithm is a two pass scheme, but since the second pass doesn't require lighting you should lose no more than half the frame rate. Since the volumes need to be recomputed for any individual object that moves or for all objects when the light moves, the actually loss will depend on the object complexity.

Most of the goals set out were met. The bias matrix was key to having proper shadows using shaders. There is definitely further scope for improvement and changes in shadow mapping.

**BIBLIOGRAPHY**

1. Crow, Franklin C., "Shadow Algorithms for Computer Graphics," Computer Graphics (SIGGRAPH '77 Proceedings), pp. 242-248, July 1977.
2. Williams, Lance, "Casting Curved Shadows on Curved Surfaces," Computer Graphics (SIGGRAPH '78 Proceedings), pp. 270-274, August 1978.
3. http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/
4. https://www.opengl.org/archives/resources/faq/technical/lights.htm
5. http://nehe.gamedev.net/tutorial/shadows/16010/
6. http://ogldev.atspace.co.uk/www/tutorial23/tutorial23.html
7. https://graphics.stanford.edu/~mdfisher/Shadows.html