Visible Space with Shadow

Caleb Chan 100768143

# Introduction

The goal of the project was to find the visible space from a point of view using shadows. Finding the visible space can aid in many different techniques from path finding to collision detection. For this to be possible the visible space must be in a format, like a 2D array, that can be used. In addition, to aid in collision detection, the visible space must be generated in real time.

There are a couple of solutions to find the visible space. One is an adaptation of back face culling and the other is to use shadows. Using shadows provides an interesting avenue for solving the visible space problem. The dark part of a shadow on a surface can be a kin to the lights non-visibility of the surface. The following sections will introduce different shadow generation techniques and explain how to convert a scene with shadows to a visible space format.

# Generating Shadows

There are three common techniques for generating shadow. They are shadow mapping, shadow volume and ray tracing. Shadow mapping is the easiest to implement and requires the use of the Z-buffer. Shadow volume is more complicated to implement then shadow mapping and requires the stencil buffer in addition to the Z-buffer to create shadows. Lastly, ray tracing is the hardest to implement because it requires generating multiple rays from the camera to the object.

In this project the shadow mapping and the shadow volume techniques were looked at and explored as possible solutions to view visible space.

# Shadow Mapping

The shadow mapping is technique for creating shadows. The technique was introduced in 1978 by Lance William . When the algorithm was first introduced, the key advantages were that it could handle indefinitely large environment and it had a linear cost growth. The advantages are the result using the Z-buffer / depth buffer. As a result of using the Z-buffer, objects being rendered do not need to be sorted beforehand and are implicitly sorted by the Z-buffer.

Shadow mapping is a two step process that uses the Z-buffer. The first step renders the scene from the lights point of view and store only the depth in the Z-buffer. The Z-buffer holds the depth information as floats for the rendered scene. The closer the object, the lower the depth value and the farther the object, the higher the depth value. A visual representation of the Z-buffer is shown in Figure 1. In the visual representation, the closer objects are darker and thus have a lower depth value while the farther objects and the background have a higher value and are lighter.



Figure : Visual representation of the Z-buffer

The second step renders the same scene from the camera's point of view. For each point on the screen, the shadow mapping technique compares the distance from the camera to the point and the distance from the light to the point. If the distance from the light to the point is smaller than the distance from the camera to the point, there is an object in front of that point and thus the point color should be darkened because it is a shadow point.

To explain this step further let us consider Figure 2. In the figure we are given three points, the view from the camera and the view from the Z-buffer from step 1. Let us first look at P1. When the checking if that point is a shadow it compares the value in the previous Z-buffer for that point to the value in the current Z-buffer for that point. If the point in the previous Z-buffer is less, there is an object blocking that point. In this case for P1, both depth values are the same and thus there is no point blocking P1 and the point is not in a shadow. Now let us look at P2. As with P1 the values in the both Z-buffers are compared for P2. In the previous Z-buffer, the value for P2 is lower than in the depth value for current depth buffer. This means that there is something blocking the P2 and thus P2 is a shadow and should be darkened. When we look at P3 we see it has the same outcome as for P1 and thus there is no need to darken the point.

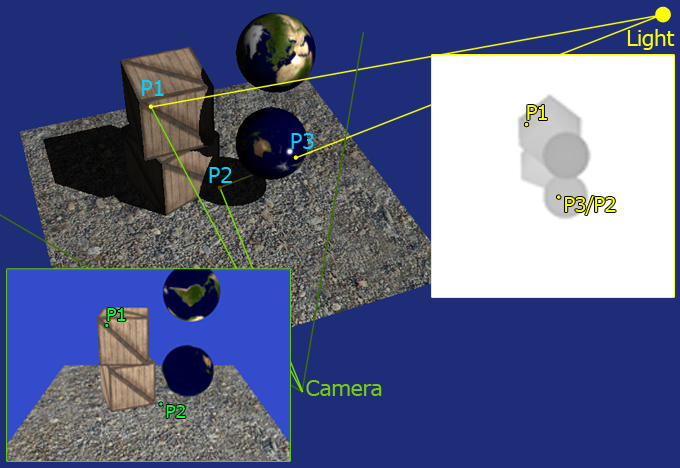


Figure : A comparison of the Z-buffer view and the camera view for three different points in 3D space

The comparison of the different depth values can be done using the fragment Shader. Using the Shader it allows for scanning through all the pixels to be done by the GPU. Since the GPU are optimized to compute the pixel color, this comparison operation can be performed quickly.

## Implementing Caveats of the Shadow Mapping Technique

Implementing the shadow mapping technique is a pretty straight forward. During the first step, the Z-buffer needs to be saved locally. This is done because rendering in the second step will override the values in the depth buffer. To save the depth buffer a Frame Buffer Object(FBO) and a 2D texture must be created. When creating the FBO we can assign which buffer to use. For example we can assign to FBO to only accept writing to the color buffer and discard all writes to the Z-buffer. For the shadow mapping, we just want writes to the depth buffer and to ignore all other buffers. The texture will need to be linked with the FBO so whenever anything is written to the FBO it will get written to the texture. Once we have create this FBO, whenever we want to write to it we just have to set it as the current frame buffer.

The second caveat that is the depth texture follows a UV coordinate system and it is different from the normal homogenous system. The UV coordinate system is a [0, 1] coordinate system where as the homogenous system is a [-1, 1] system. This means that when trying the access any of the values in the depth texture we first need to shift the coordinates. The matrix below will shift the coordinates from homogenous to UV.

Lastly the act of writing to the depth texture does produce some errors and thus another bias is applied to the depth value of the current Z-buffer.

## Technical Challenges for Shadow Mapping

While programming the shadow mapping algorithm the one major technical challenge was the appearance of artifacts. Figure 3 shows a comparison of the same view with artifacts and without artifacts. In the artifact picture we can see random horizontal lines, the shadow at the bottom is detached from the object and the shadow on the right of the image has jagged lines.

There are a couple of easy fixes to this problem. Most of the problems can be either fixed or reduced by either adjusting the bias on the current Z-buffer, changing the projection matrix for the light or both. By adjusting the bias, the error on the depth texture can be minimized. A projection matrix with a extreme parameters will yield an incorrect depth projection of the world from the light's point of view. This could lead the shadow mapping to incorrectly discern which is a shadow and which is not.

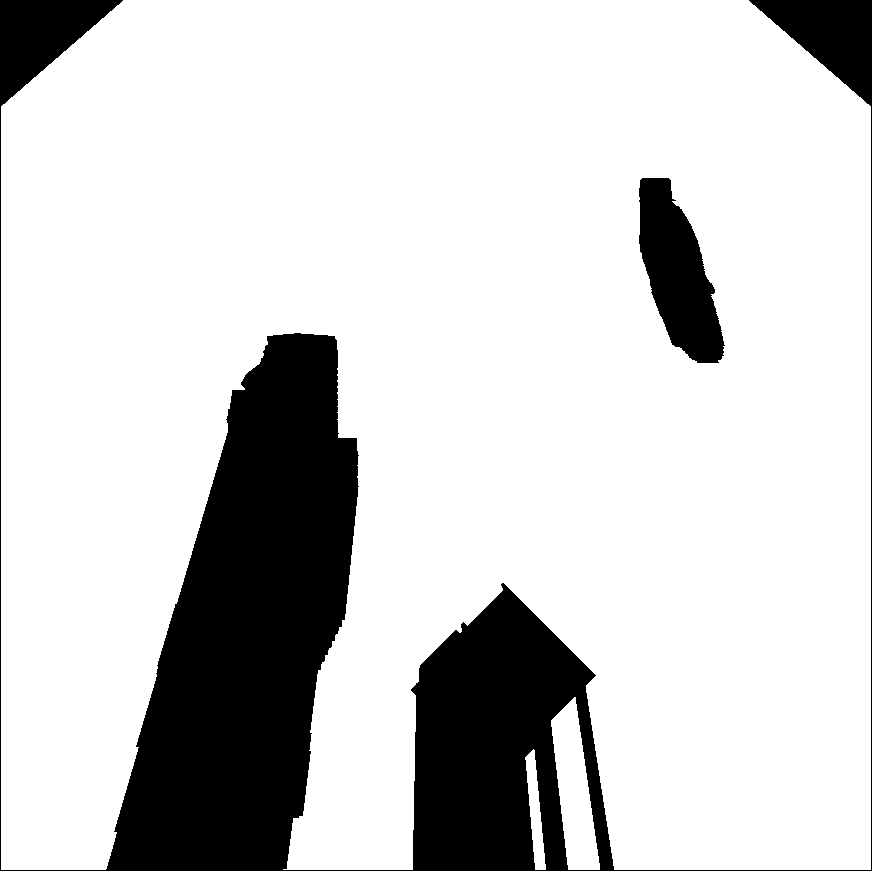
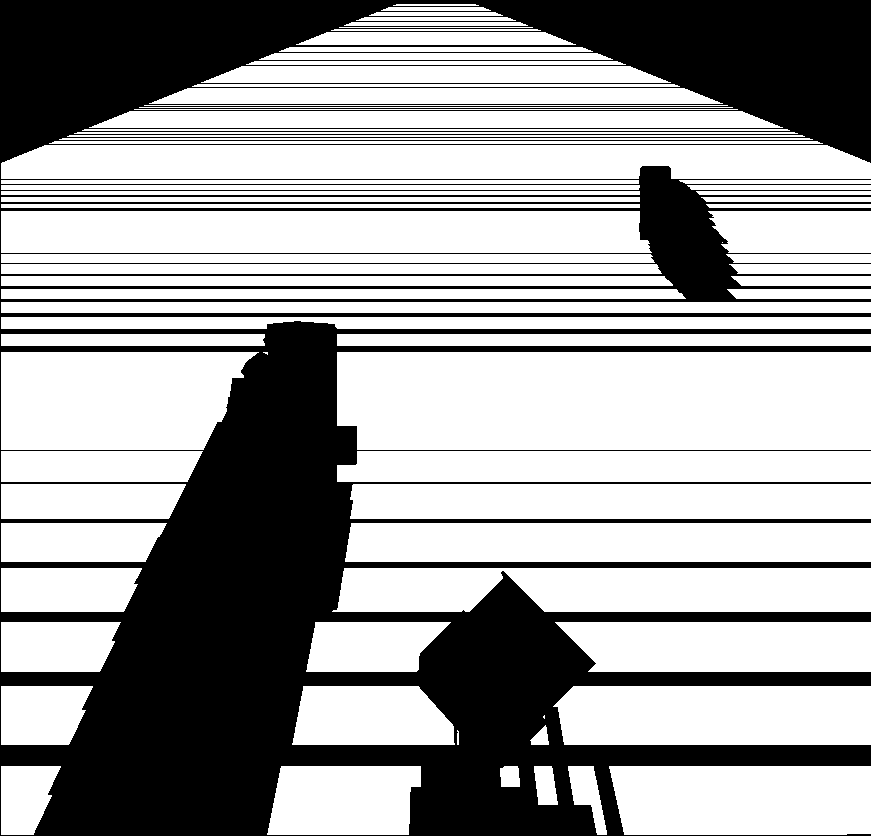


Figure : A comparison of the same image with artifacts(left) and without artifacts(right)

# Shadow Volume

The second technique for creating shadow is called shadow volume. As with the shadow mapping technique it is also a multi-pass rendering technique. The main difference is that it uses the stencil buffer in conjunction with the depth buffer. The purpose of the stencil buffer is to include or exclude parts of the screen for rendering. As with the depth buffer it is a 2D structure, but each location in the stencil buffer is integers instead of floats. The inclusion or exclusion of the pixels of the rendered image is determined by whether the value in the stencil buffer meets a certain condition.

The shadow volume technique is a multi-step process. The steps to the shadow volume technique are as follows.

1. Rendering to the depth buffer;
2. Creation of silhouettes of the object with respect to the light source;
3. Extending the silhouettes away from the light;
4. Render the scene to the stencil buffer with the depth fail conditions set
5. Render to screen taking into account stencil buffer

The silhouette is used to track where the shadow resides. In Figure 4, the solid blue lines show the silhouette that is created by the light source. The silhouette rests on the light facing face of the object and extends toward infinity. The silhouette can be generated by finding the faces that are facing the light to create the light cap. Once this is create, the edges of the light cap can be extended towards infinity in the direction of the light rays as shown by the red lines in Figure 5.

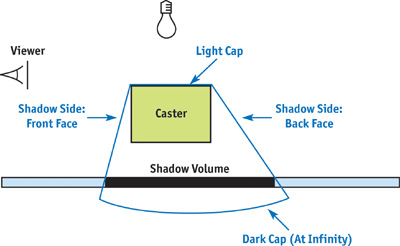


Figure : The silhouette generated by the light

There are a couple of ways to use the stencil buffer to generate shadows. They are depth fail and depth pass. For this project I focused on using depth fail. When using the stencil buffer for depth fail, there are certain update rules for the stencil buffer that must be set. The first being if the back face of the shadow polygons is rendered, we increment the appropriate value in the stencil buffer. The second rule is that if we render the front face of the shadow polygon, we decrement the value in the shadow buffer.

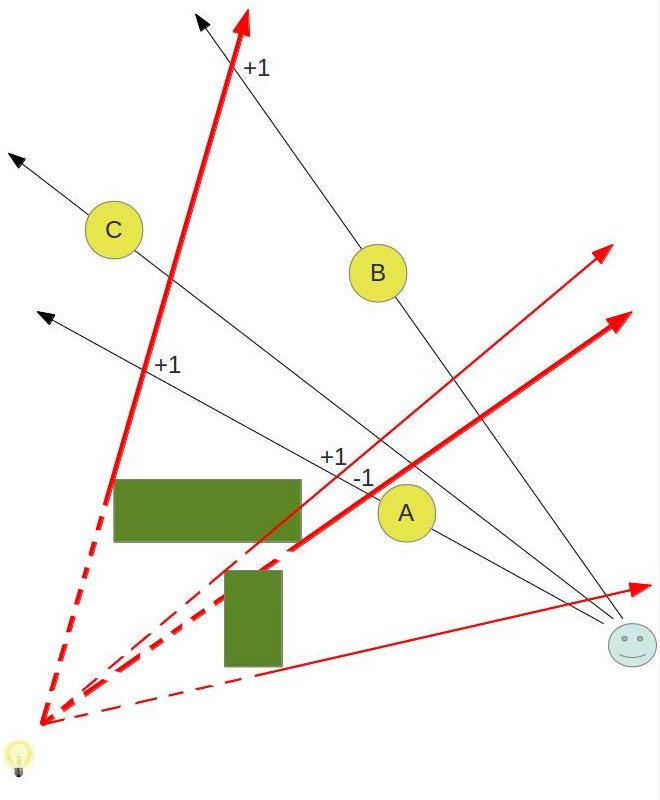


Figure : A figure showing the shadow volume with the increment and decrement values of the stencil buffer

Consider the image shown in Figure 5. The +1 and -1 represent changes to the stencil buffer when rendering the shadow silhouette. Any object (A, B or C) that has a stencil value that is zero will get rendered. This means that it does not appear in any silhouettes and will get rendered.

## Technical Challenges for Shadow Mapping

The biggest technical challenge was changing the ordering of how the original code drew triangles. By changing how triangles were draw, the geometry Shader could be used to add new vertices and create the silhouette. The program was changed from accepting a list of triangles to accepting a list of triangles with their adjacent triangles. By do thing allows for the an easy way to find the edge of the silhouette to extend away from the light.

To change the drawing procedure, the number of points needed to pass to the Shader is 6 from 3 points. The increase in number of points is due to the main triangle and its neighbor triangles connected by an edge. As a result of using the new format, the list of points needed to be changed. Unfortunately this change would increase the amount of information needed to be saved because the position, normals and any other vertex information would double. The work around to this problem is to save each unique vertex information only once and just reference the vertices by their index number. As a result, the amount of data needed to store each object is reduced to a list of unique vertices and a list of index numbers representing the format new format of the faces.

Another technical challenge was the algorithm used to generate silhouettes require that every face must have 3 neighbor faces. The meshes that were provided had faces with one or two neighbor faces that that cause may errors. To fix this problem, the original meshes were changed to compete meshes when rendering the shadow volume.

# Putting Everything Together

To find the visible space was the easiest part of the project. If you point the camera above the whole scene you can see the shadow that is being case from a light source close to the ground. This provides the visible space from the light source. In Figure 6, the light source representing a person is located at the top of the screen. The shadow it cast is shown in the figure. To get the visible space, everything that cannot be seen through (objects) is changed to the color black. The surface all the objects are on changed to the color white and it is assumed that its visible to the person. To use the visible space, I rendered the screen to another FBO to store the screen into a texture. This allows anyone to use the visible space after.

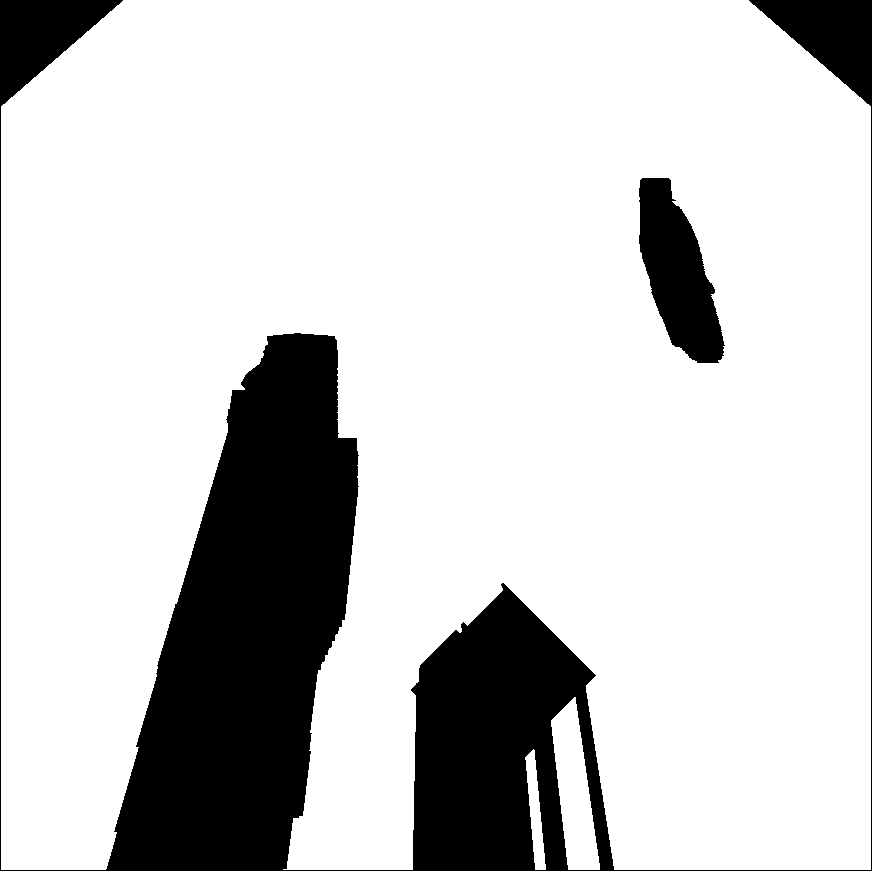


Figure : Visible space view

# Conclusion

In conclusion, I have shown that generation of visible space can be possible using shadows. I have shown that by using shadow mapping or shadow volume the visible space can be generated. By leveraging the Opengl framework and the Shader framework, the generation of shadows can be done quickly.

This project can be further improved by using the visible space to move the person(light) around. In this project, using the visible space was not done and this project mainly focused on generating the visible space.

Another thing that can be improved is using a directed light instead of a point light source for shadow volume. In the implementation of the shadow volume technique, the light source is a point light source. This means that it radiates light in all directions. Although it simulates visible space in front of it, it does not take into account that a real person cannot see behind them. As a result no shadow is generated behind the light like in the shadow mapping implementation.

Another direction to take this project is to use multiple light sources. For example, the problem of mapping a terrain using or a complicated floor plan using multiple independent visibility maps. By using many different small visibility maps of each room or part of the terrain, can one generate a larger visibility map that is accurate.

# Bibliography

|  |  |
| --- | --- |
| [1] | L. Williams, "CASTING CURVED SHADOWS ON CURVED SURFACES," *SIGGRAPH Comput. Graph.,* vol. 12, no. 3, pp. 270--274, Aug 1978. |
| [2] | M. Alamia, "Tutorial - Deferred Rendering Shadow Mapping," [Online]. Available: http://www.codinglabs.net/tutorial\_opengl\_deferred\_rendering\_shadow\_mapping.aspx. [Accessed 16 April 2015]. |
| [3] | M. McGuire, "Chapter 9. Efficient Shadow Volume Rendering," 2004. [Online]. Available: http://http.developer.nvidia.com/GPUGems/gpugems\_ch09.html. [Accessed 16 April 2015]. |
| [4] | E. Meiri, "Tutorial 40:Stencil Shadow Volume," [Online]. Available: http://ogldev.atspace.co.uk/www/tutorial40/tutorial40.html. [Accessed 16 April 2015]. |