Implementation and Assessment of Software-based Shadow Mapping Techniques: An Extension to the Gz Library

Sijie Bu
sbu@usc.edu

Jessi Bustos
jrbustos@usc.edu

Shao Ling Tan
shaoling@usc.edu

ABSTRACT

The Gz library originally only supported local shading, and thus lacked the ability to produce shadows. While ray tracing and radiosity techniques can be used to produce photorealistic shadows, these methods are computationally expensive and do not integrate well into the object-based rendering pipeline of the Gz library. Another method, known as shadow mapping, involves re-using the z-buffering algorithm that is originally used for hidden surface removal. While this algorithm is not capable of producing photorealistic shadows, can still provide a reasonable approximation, and can be easily integrated into the standard rendering pipeline of the Gz library.

In this report, we will be discussing the basics of the shadow mapping algorithm, and demonstrate the steps needed to add a simple shadow mapping algorithm to the Gz library. Finally, we will also be discussing some of the caveats that were encountered in the implementation process, and some important considerations.

1. INTRODUCTION

Shadow mapping, an algorithm first described by Lance Williams of New York Institute of Technology in 1978, is a relatively simple method of approximating shadows in a computer-generated scene with the same z-buffering logic that is being used for hidden surface removal [1]. Originally, Williams noted that this approach excels in logical, computational, and implementational simplicity but has a significant memory requirement [1]. However, with the advent in modern computer hardware, such drawback has since become less of an issue, and thus making shadow mapping an attractive method of approximating shadows in an object-based renderer that utilizes z-buffering as the principal method of hidden surface removal.

The Gz library, which had been under several iterations of improvements since the beginning of this semester, currently supports local illumination based on the Gouraud and Phong shading models as well as simple texture mapping. However, it lacks the ability to render any true shadows. We extended the Gz library with shadow abilities by reusing the core z-buffered rasterizer, and our implementation was tested against two fixed light sources and the standard Utah Teapot input files that were used to test the previous iterations of the Gz library. The Utah Teapot was specifically chosen because as implied by course materials from the University of Utah, it is capable of casting shadows onto itself [2].

1.1 Base Library Description

This experiment uses the 6th iteration (HW6) of the Gz library from Sijie Bu (combined with the driver/application code from HW5) as a starting point. The 6th iteration contains all the base logic that is required to implement the shadow mapping logic, and the driver code from the 5th iteration contains a good starting point, without any extra logic that might interfere with the experiment.

The starting point library (starting point) uses LEE as the rasterization method and has passed all conformance tests from HW1 through HW6 without any revealed bugs or issues. Therefore, the starting point serves as a solid foundation for implementing additional features that rely on core rendering logic such as z-buffering.

1.2 Implemented Algorithms

This experiment implemented the original shadow mapping algorithm that was described by Williams, along with William's solution of resolving aliasing of the shadows or shadow acne that involves low-pass filtering [1]. In the process, we also referred to an OpenGL implementation of shadow mapping [3] for clarification on details of the algorithm.

To successfully implement a shadow mapping algorithm, one also needs to be able to invert a projection matrix (see algorithm description for details). For this purpose, we adapted a generic matrix inversion algorithm, which uses determinants, from the Mesa 3D graphics library [4]. No other external code or dependency besides this function was introduced.

We also implemented a simple nearest-neighbor subroutine based on the descriptions from MathWorks [5]. This subroutine is being used by the shadow mapping logic to interpolate possible fraction points to precise integer points, which is required for framebuffer and depth buffer lookups.

Finally, since the key to accurate shadow and highlight representation is per-pixel shading computation [1] [3], we only implemented shadow mapping for Phong shading. In our options, while it is possible to add shadow mapping to Gouraud shading, the practical usage of such implementation will be limited, as Gouraud shading is commonly used when accuracy can be traded for speed.

2. ALGORITHM DESCRIPTION

The shadow mapping algorithm implemented in this experiment contains two parts: the core shadow mapping logic, which is based on the z-buffering algorithm for hidden surface removal [1], and a simple resolution of the shadow acne phenomena with a fixed low-pass filter [1] [3].

Figure 1. Utah Teapot from the main camera.

2.1 Core Shadow Mapping

According to Williams, the core shadow mapping logic is analogous to the hidden surface removal process. Williams proposed that if one set up a camera from the perspective of a light, then the result is what the light "sees" or illuminates in the scene [1]. Therefore, if one sets up a renderer that has the camera set up with the same perspective of the light source, then the image that would be rendered would only contain the parts of the scene that were illuminated by the corresponding light sources, and the parts that were not illuminated, would be removed in the z-buffering process [1]. And when one renders the actual scene from the perspective of the main camera, one can project a given screen-space pixel into light space (the projection space associated with a given light source) and compare the resulting z-value and the z-value stored in the light-space framebuffer. If the light-space projected z-value is greater than the z-value stored in the light-space framebuffer, then one can be certain that the screen pixel is not being illuminated (under shadow) by the current light source, and apply processing techniques (such as changing the coefficients of the shading equation) to render the current pixel as shadow [1].

2.1.1 Changes to the Renderer Logic

The renderer, or the `GzRender` class, is where the rendering itself happens (z-buffering, LEE, shading, etc.) and it is natural for one to reuse this class for shadow mapping. Our modifications of the `GzRender` class fell under two categories: support for rendering the shadow map itself, and changes to the shading logic to take the shadow map into account.

2.1.1.1 Support for Rendering the Shadow Map

Since the shadow mapping process only cares about the z-value of the light-space renderer, one can simplify the shadow mapping process by skipping color computation completely in the light-space rendering process, and only generate a greyscale image that serves as a visualization of the z-values of each pixel. Therefore, we added a lighting interpolation mode called `GZ_SHADOWMAP`, and it serves as a signaling value to tell the renderer that no color computation is needed.

Figure 2. Shadow map for one light.

When the current renderer's interpolation mode is set to `GZ_SHADOWMAP`, it will completely skip the shading computation process, and set the color channels (R, G, and B) into the current z-value (scaled from the $0 - \text{INT\_MAX}$ of the z values $0 - 1$ of colors). However, in the process, we noted that the resulting shades of grey were too faint. Therefore, we multiplied the scaled greyscale value with an "intensity factor" of 0.75 and got blacker (and more obvious) visualizations of the shadow map.

Figure 3. Shadow map for another light.

The resulting shadow maps were never rendered to the screen but were written to disk as files along with the main frame buffer, which are then used to help with debugging other parts of the shadow mapping logic and for the purpose of demonstrating the intermediate steps of shadow mapping.

2.1.1.2 Changes to the Shading Logic

The original paper by Williams [1] did include the basics of the shadow mapping process: screen-space points that are visible need to be projected back to image space, and then re-projected to the space of each given light (light space) and have the corresponding shadow map z-values looked up. If a screen-space point has a z-value greater than the corresponding shadow-map z-value, then it falls under the

shadow for that light [1]. While Williams omitted the exact algorithm for this projection process, we were able to devise it with our understanding of projections and transformations, and it will be discussed in this section.

To implement our devised projection algorithm, we also had to modify some of the data structures of the Gz library. One of the major changes is that now each GzLight struct also contains a pointer to the light's corresponding shadow map renderer. This change allows us to easily access the shadow map of a given light, simplifying the shading process. This also demonstrates that the time and space requirements for shadow mapping scales linearly with two factors: the framebuffer resolution and the number of lights (which translates to the number of additional framebuffers that need to be rendered).

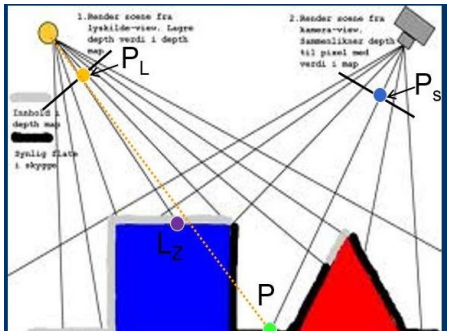

Figure 4. Z values of a point in shadow (class notes).

By allowing the access of the shadow map renderer of a given light, we can reference the light's corresponding Ximage transformation matrix (or Xlight) from the main renderer. With knowledge of both transformations (image space to screen space and image space to light space), we are then able to construct the transformation matrix that maps any given screen-space pixel to the corresponding light's space:

$$P_L = X_{LS} \times P_S \quad (1)$$

And the X_{LS} term in (1) is defined as the following:

$$X_{LS} = X_{light} \times (X_{image})^{-1} \quad (2)$$

While P_s is a point in screen space and P_L is the result of projecting the screen-space coordinates into the light's corresponding space.

By examining (2) one can conclude that we need the inverse matrix of the main renderer's Ximage to project a screen-space point into light space. We considered several approaches to this problem, including maintaining a separate "inverse transformation" stack, decomposing and inverting individual elements of the top of the Ximage stack, and generic matrix inversion algorithms. Eventually, after consulting the practices of other 3D libraries, we adapted a determinant-based matrix subroutine from an old version of the Mesa 3D library [4] for this purpose, as it is cleaner than maintaining a separate "inverse matrix" stack and is more efficient than a row-reduction based inversion subroutine.

During the color computation step, for each non-ambient light, the coordinate of each pixel is being projected into the light's corresponding space, using (1). Then the projected light-space coordinates are used to query for the z-buffering information of the light's shadow map with GzGet(). By giving the (x, y) coordinates, GzGet() will be able to return the current z-value stored in the shadow map.

Figure 4 (taken from the class notes on shadow mapping) illustrates that for a point P to be in shadow, its z-value should be greater than the stored depth buffer value of the shadow map image. Therefore, we then compare the retrieved z-value from the shadow map with the transformed light-space z-value. If the transformed z-value is greater than the z-value from the depth buffer, then we render the current pixel as in the shadow casted by the current light by decreasing the contribution of the current light by multiplying the current diffuse and specular components with a pre-defined coefficient that is less than 1.0.

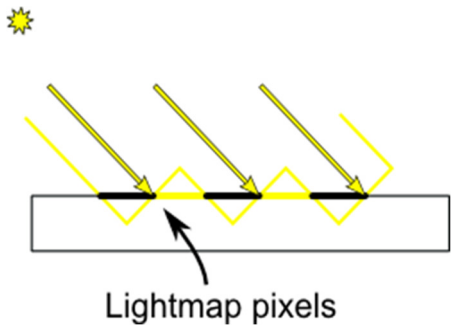

Figure 5. Shadow acne [3].

Intuitively, one might notice that (1) does not guarantee integer outputs, and the GzGet() method of a renderer only accepts integer inputs. We solved this problem by introducing a Euclidean-distance-based nearest-neighbor interpolation by implementing the algorithm described in the help documents of MATLAB [5]. The resulting (x, y) values are then used to retrieve the needed z-value with GzGet().

Figure 6. Shadow acne in our implementation.

2.2 SHADOW ACNE

A major problem that Williams noted in the original shadow mapping paper is what he terms as "shadow moiré" [1] or "erroneous self-shadowing due to precision issues" [6] that is caused by insufficient sampling resolutions and also problems with floating point precision [1] [6], or more commonly known in modern literatures as "shadow acne" [3] [7]. Figure 5 from [3] demonstrates the microscopic side of this issue.

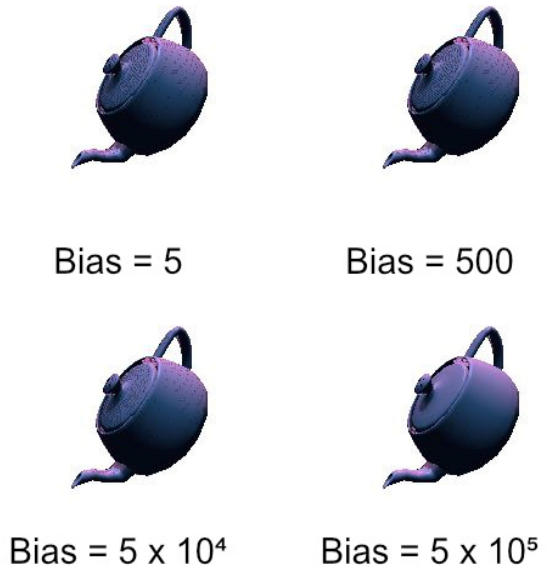

Figure 7. Different shadow biases.

As expected, we also encountered this issue when we only implemented the two-pass z-buffering shadow mapping logic. However, from Figure 5, one might notice that this kind of artifact has a high-frequency nature. Therefore, a simple low-pass filter [8] will effectively mitigate this issue. For scalar values, we introduced a "bias value" [1] which serves as the threshold difference between the pixel and shadow map z-values for determining whether the two z-values are "distant" enough to be considered in shadow:

$$z_{Pixel} - z_{ShadowMap} > bias \quad (3)$$

We started by testing a small bias value of 0.5 and incrementally went up by a factor of 10 until we got a satisfactory output image with no shadow acne. The ideal constant bias value for our predetermined camera position was 5×10^5. As noted in Figure 7, it took several attempts before reaching a shadow bias value that would result in an output with no visible acne artifacts.

Figure 8. An angle requiring a large bias.

It is worth noting that one constant shadow bias value will not work for all camera positions. In other words, a shadow bias of 5×10^5 might not be enough to eliminate most if not all visible shadow acne if the camera position was set to another location. See Figure 8 for another angle of our teapot where shadow bias of 8×10^5 was needed to generate an image with proper shadows and relatively little to no visible shadow acne.

Figure 9. Artifacts when the bias is too large.

An interesting artifact that was noted while testing out different shadow bias values is that if the bias is too large, the shadows will start moving away from the teapot and thus creating an effect that some sources refer to as "Peter Panning" [3] [7]. It is also plausible that if the bias is too large, the shadows will be eliminated entirely from the scene (Figure 9). Therefore, it is crucial to experiment with the shadow bias value until one determines a number that eliminates all the shadow acne while keeping all the shadows in place. This can be done by incrementally testing out different bias values. There have been both academic [6] and empirical solutions [3] for self-adaptive biases, but due to constraints on the project timeframe we were not able to explore them thoroughly.

3. CAVEATS AND DISCOVERIES

In the process of implementing shadow mapping, we also discovered several caveats and limitations that are worth discussing.

3.1 Light Space Coordinates

Intuitively, one might think that just like normal vectors, RGB colors, and texture indices, the light space coordinates can also be directly interpolated by computing a plane equation and linearly interpolate the light-space coordinates from the light-space values of the vertices of a triangle.

We initially attempted this approach, but the interpolated light-space values did not yield any shadow, and examining the interpolated values revealed that they are very different from the actual light-space values of pixels (especially in terms of the z-values). Switching to the inverse projection method that involved inverting the `Ximage` matrix, however, yielded correct results.

We are not sure about exactly why the light space coordinates cannot be directly interpolated. However, we

speculate that since perspective projections contain a non-linear element, a linear interpolation would produce incorrect results if the non-linearity were not accounted for, and in this case, the simplest approach is not to attempt to compensate for the non-linearity, but to just construct the needed inverse projections. However, at least one documentation from Microsoft [7] mentioned that even when using inverse-projection matrices, the non-linear element in projections can still cause artifacts. Therefore, we believe that the non-linearity of the projections is the reason behind this observation, like how it can cause artifacts even with inverse-projection matrices.

3.2 View Frustum for Lights

A major limitation of our implementation of shadow mapping is that the view frustum parameters for each light needs to be manually crafted. To make this process more tedious and complex, there is no generic transformation formula that can easily construct the required view frustum from the parameters of a given light. In our experiment, we hand-crafted the view frustum parameters via trial-and-error and had to fine-tune the parameters for each individual light. We believe that while this is doable, this process could take up considerable effort and burden for the modelers if shadow mapping is being applied at scale.

4. CONCLUSION

By extending the Gz library with shadow mapping capabilities, we have proven that the shadow mapping algorithm can be an attractive option for quickly adding shadows to a traditional object-based renderer. Instead of introducing a set of completely new logic and algorithms, shadow mapping can be easily added via creating additional framebuffers and re-using the z-buffering hidden surface removal process. Our implementation also shows that the computational time and space requirements scale linearly with the framebuffer resolution and the number of lights. Our experiment also revealed that there are additional caveats that one might come across when implementing shadow mapping. Though there are various academic and empirical solutions to these caveats, one still must pay close attention to details and conduct tedious fine-tuning to implement shadow mapping successfully.

5. ACKNOWLEDGMENTS

Our thanks to the CSCI 580 course staff for providing us with excellent course notes, base code for the Gz library, and guides for carrying out this experiment. We would also like to thank for the Mesa 3D project, MathWorks, and the OpenGL-Tutorials project for detailed implementation of various subroutines and algorithms, which we either adapted or referred to when implementing the shadow mapping function for the Gz library.

6. References

[1] L. Williams, "Casting curved shadows on curved surfaces," 1978.

[2] "Project 7 - Shadow Mapping," The University of Utah, [Online]. Available: https://graphics.cs.utah.edu/courses/cs6610/spring2020/?prj=7. [Accessed 24 4 2021].

[3] "Tutorial 16 : Shadow mapping," [Online]. Available: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/. [Accessed 24 4 2021].

[4] "new ___gluInvertMatrix() function (Mesa bug 6748)," 27 08 2007. [Online]. Available: https://gitlab.freedesktop.org/eric/mesa/commit/3069f3484186df09b671c44efdb221f50e5a6a88#57de6276c46a97f5e8629ed0c426cdf64d6a4103. [Accessed 24 04 2021].

[5] "Warp: Apply projective or affine transformation to an image," The MathWorks, Inc., [Online]. Available: https://www.mathworks.com/help/vision/ref/warp.html;jsessionid=096ebcfe483c693c8d89c12c3495#bun4186-1_sep_mw_e47bf23e-68c7-4ef1-bf1b-18ae877bd90c. [Accessed 26 04 2021].

[6] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackerts and J. Kautz, "Exponential shadow maps," *GI '08: Proceedings of Graphics Interface 2008,* pp. 155-161, 2008.

[7] Microsoft, "Common Techniques to Improve Shadow Depth Maps," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps?redirectedfrom=MSDN. [Accessed 27 04 2021].

[8] "Filtering an Image," [Online]. Available: https://northstar-www.dartmouth.edu/doc/idl/html_6.2/Filtering_an_Imagehvr.html. [Accessed 26 04 2021].