**Project Report DH2323**

**Rasterizer**

Axel Weinz - [aweinz@kth.se](mailto:aweinz@kth.se)

Oscar Ström - [osst@kth.se](mailto:osst@kth.se)

**Specification**

<https://docs.google.com/document/d/1ZDgEaeHzXG0k69vbbc-dn8YVGHzlkUB_928M3JZKJKI/edit?usp=sharing>

**Introduction**

Our project for this course was to expand the rasterizer from track 1 in the labs. Throughout the project we have worked both together and separately. To make this work smoothly, we have used Github to store our work. We have also created a blog that showcases the process. The blog was created in the end of the project, which means that the dates of the posts are not representative for the time period of the process.

<https://github.com/Sayaja/Rasterizer>

<https://rasterizer633638350.wordpress.com/>

**State-of-the-art**

There are a lot of different techniques and previous work on both rasterizers and implementing shadows. Rasterizers are the most commonly used technique for making 3d games because of its speed. Rasterizers converts vector shapes in to a raster image. [1] An alternative to rasterizers is ray tracers. Ray tracers however are a lot slower in practice. One advantage with ray tracers in our case is that it makes it very easy to implement shadows. There are plenty of techniques for implementing shadows when using rasterizers; we have decided to look in to a technique called shadow mapping. Shadow Mapping consists of three parts:

* Depth rendering from the lights perspective
* Render the regular camera perspective
* Compare the camera render with the light render

The technique was originally introduced by [Lance Williams](https://en.wikipedia.org/wiki/Lance_Williams_(graphics_researcher)) in 1978 and has been used in plenty of games. [2]

An alternative to shadow mapping is to use shadow volume. The main advantage to shadow volume over shadow mapping is that it is accurate to the pixel, whereas shadow mapping’s accuracy depends on the amount of used texture memory. [3]

We decided to go with shadow mapping for out project since its faster as well as it being more practical for us since there are a lot of tutorials on how to implement it.

**Background**

Working with a cornell box provides the opportunity for great amount of learning. [4] Once you have rendered it, for example with the rasterization method, there will be room for improvements and expansion. Understanding the theory behind each method of a rasterizer and then implementing it is a seemingly good way to learn about C++ and computer graphics. The possible improvements of a basic rasterizer are huge, where each method is on different levels, which provide choices to gradually increase your skill and knowledge. Adding shadows and clipping to a rasterizer are significant improvements to a rasterizer’s quality. The shadows will give the scene a much more realistic look, while the clipping will make it possible to move the camera seamlessly through the scene.

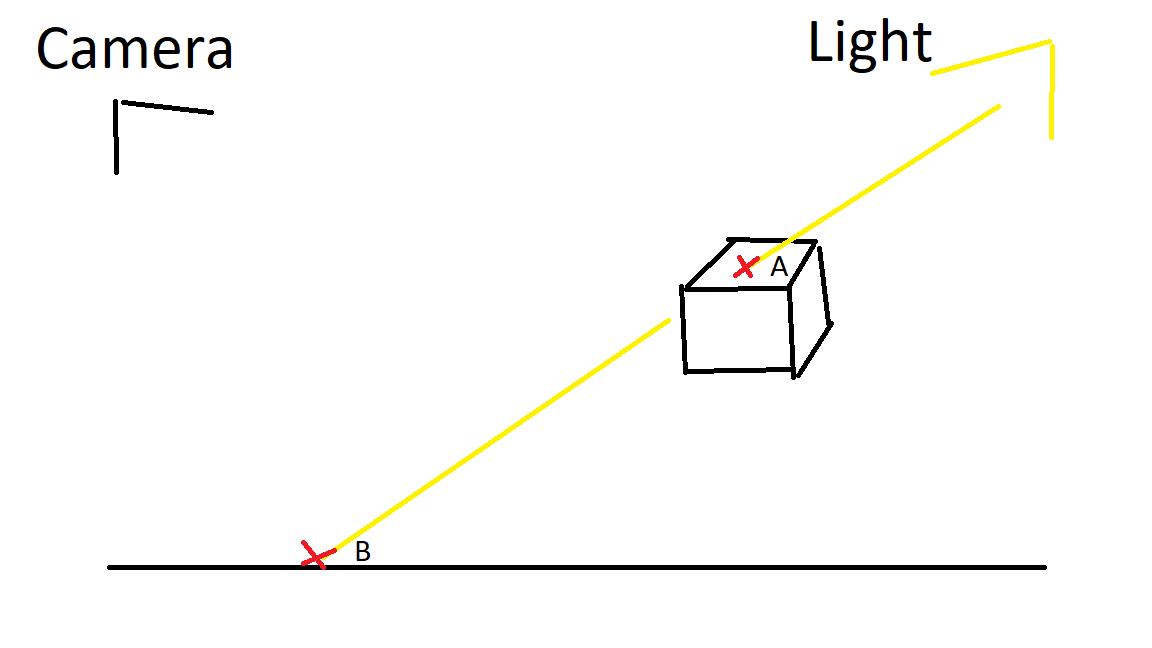
The motivation behind our project is that adding shadows and clipping to a rasterizer will give us a good understanding of how a rasterizer works. It will also provide knowledge about problems you may encounter while programming computer graphics.

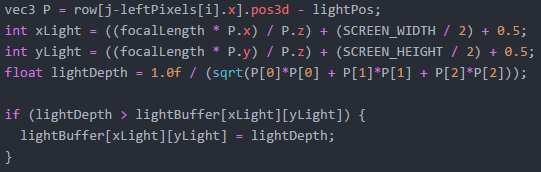
**Method**

Shadow Mapping

The method we chose for adding shadows to our rasterizer is shadow mapping. Shadow mapping is based on rendering the scene from the light source’s point of view and storing the distance for each pixel’s 3d-point to the light source. This way, when rendering from the camera’s point of view, you can compare the distances for the pixels and see if another 3d-point if closer to the light, meaning that the pixel should be shadowed. Here are the steps we used for implementing shadow mapping.

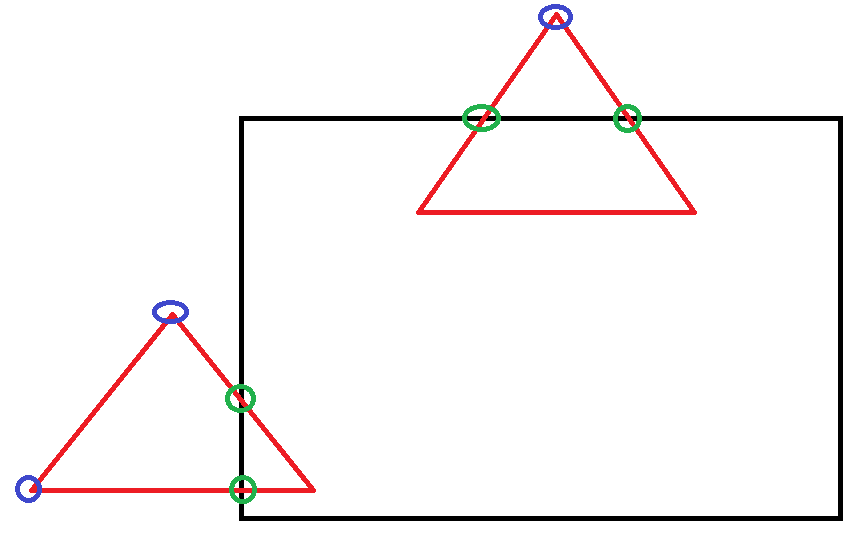
1. Create a matrix that stores the distance from the 3d-position to the light source for each pixel.
2. Render the scene from the light’s point of view and store the depth for each pixel in the matrix.
3. Render the scene from the camera’s point of view. For each pixel that is drawn, check if the distance from its 3d-position is equal or lower than the distance stored in the matrix. If so, color it. Else, it means that there is something in front of it from the light’s point of view, and therefore it should be shadowed.

For example, here is a scene with a camera and a light source. When rendering the scene from the light’s point of view, it would map point A and B to the same pixel. However, A is closer, and therefore its distance to the light source is saved in the light depth matrix. The next step would be to render the scene from the camera’s point of view. Point B is visible from the camera. Before coloring it, its distance to the light is compared with the corresponding pixel in the light depth matrix. The comparison shows that there is a point closer to the light, which is also mapped to the same pixel from the light’s point of view. Therefore, point B is to be shadowed.  


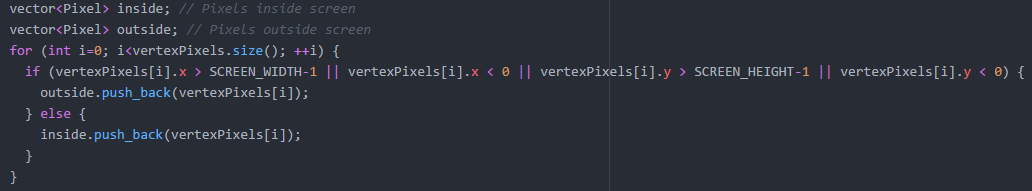
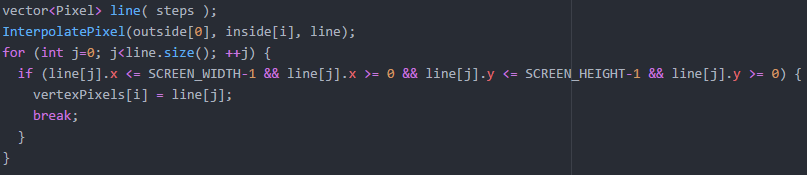
*Figure 1*  
  
Following the first step, we create a matrix with the size of the screen’s pixels. This matrix is similar to how we store the depth of the scene to draw the correct triangles in front of other triangles.  
  
  
Next, we created a method to render the scene from the light’s point of view. This method works similarly to how we render the scene from the camera’s point of view. The difference is that ShadowMapping only deals with the 3d-points and their distances to the light source.  
  
The method then maps the distance to the matrix.  
  
When drawing the polygons, the values in the matrix are then used to check if the pixel is to be shadowed or not.

Clipping  
Clipping is a straightforward method. The problem clipping solves, is that when you move the camera around, you may have 3d-points in the triangles that are mapped to pixels outside the screen size. This will cause the program to crash. The solution is to limit, or clip, the triangles so that they do not go outside the screen. Here are the steps we used to implement clipping.

1. Make the VertexShader take all vertices of a triangle at the same time. Check which vertices are outside the screen and which are inside.
2. Depending on how many vertices that are outside the screen, replace the vertices that are outside the screen by interpolating between a vertex outside the screen and a vertex inside. Then, loop through the interpolated pixels and create a new vertex on a pixel inside the screen.

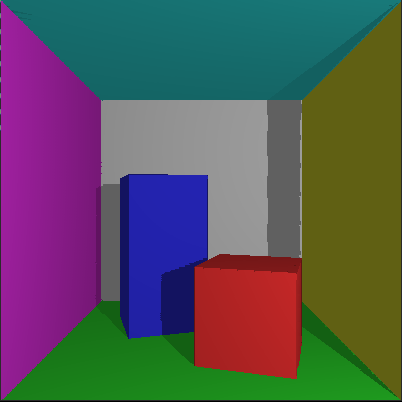
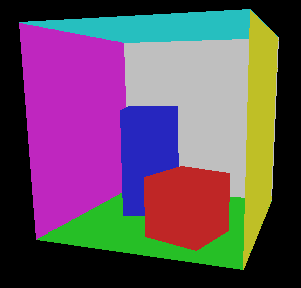
The theory behind clipping is that there are four different cases for a triangle concerning if it is inside or outside the screen. Those are the following: all vertices inside the screen, one vertex outside the screen, two vertices outside the screen, all vertices outside the screen. For each case, we need to perform different actions. The most interesting ones are when one or two vertices are outside the screen. The image describes what needs to be done to perform clipping for those two cases. We need to create new vertices at the green circles, while removing the vertices at the blue circles.  


*Figure 2*

Following step one, the first thing we do in the code is change VertexShader so that it can take all vertices for a triangle at the same time. Then, we check which vertices are mapped outside the screen, and which are mapped inside.  
  
Next, we go into the different cases depending on the size of the vector “outside”. If we find that there are one or two vertices outside the screen, we then interpolate between the vertices outside the screen and the vertices inside to get a line between them. Next, we loop through this line starting at the vertex outside the screen, until we reach a 3d-point that is mapped inside the screen. That is where we create our new vertex.  
  
Lastly, we make sure to keep the vertices that were already inside the screen.  

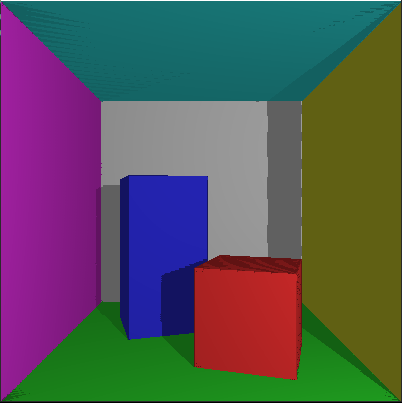
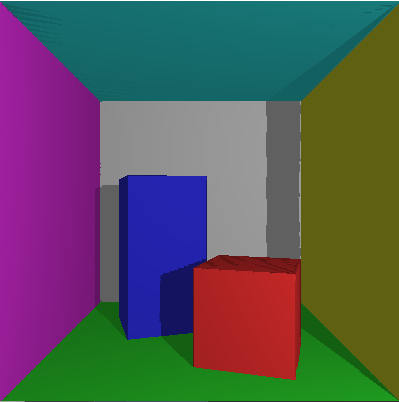

**Result**

The result is based on our implementation of shadow mapping and clipping. For shadow mapping, we managed to get shadows to work. Here is an image from the light’s point of view and its corresponding shadows from the camera’s point of view. The light is placed to the right, behind the camera.



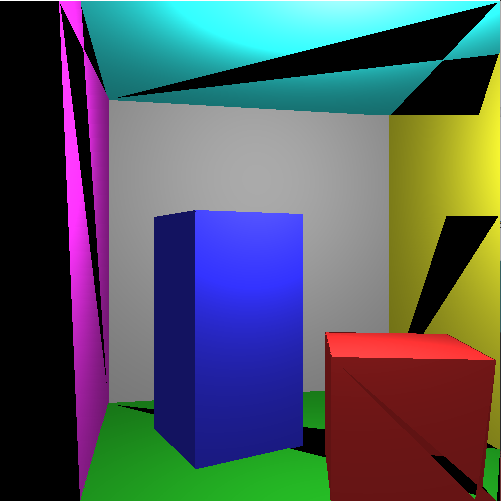
*Figure 3 and 4*

Here are images from the same point of view from the camera, but with a lower threshold for self-shadowing. Self-shadowing can be seen on top of the red cube.



*Figure 5 and 6*

For the clipping, we managed to create clipping so that the program does not crash as soon as a triangle’s 3d-point is mapped to a pixel outside the size of the screen. However, there is a problem with gaps in the polygons appearing after the clipping, as can be seen in figure 7.



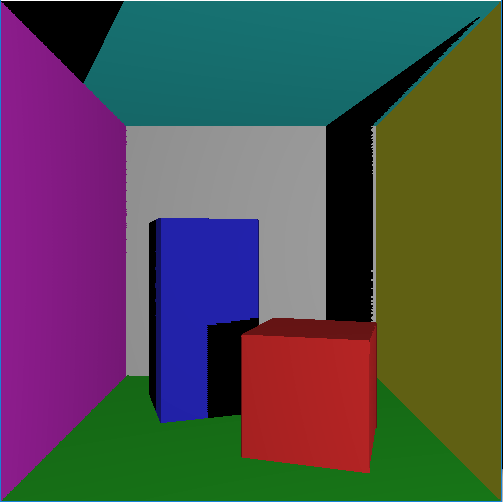
*Figure 7*

**Discussion**

During our project we learned the theory behind the algorithms clipping and shadow mapping for a rasterized 3d scene, while also implementing it. It has been an enjoyable project, as it felt like we understood the technique and theory behind both shadow mapping and clipping. However, it has also been frustrating to work with, as even though we felt like we understood the process, we were stuck on certain problems for a long time, especially with shadow mapping. We still have not got clipping to work perfectly, but we did not put as much time into solving that problem as we did with shadow mapping.

In this project, we have worked with the light source placed outside the room with a full view of the scene. This is because we are using a matrix with the size screen height times screen width to store the distances to the light source. This would not work if a polygon’s 3d-point was mapped outside the view of the light. We were planning to make the program work with the light source placed inside the room with clipping, but we did not have time to complete this.

Earlier in the process, we had trouble with shadows appearing at places they should not. This was not just self-shadowing but something else. The shadow in the upper left corner should not be there.



*Figure 8*

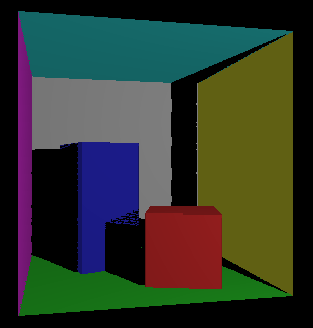
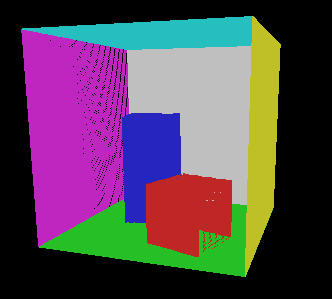
The problem was caused by the rotation of the light source. When we had this problem, we had no rotation matrix for the light source, which meant the light was pointing in the z-direction, not having the whole scene in its point of view. This meant that the parts of the scene not in the light’s point of view got shadowed, when they should not. We solved this by applying a rotation matrix to the light, so it had the whole scene in its point of view.

During the implementation of shadow mapping, we encountered a problem called shadow acne, shown in figure 5 and 6. [5] Shadow acne is caused by a triangle incorrectly shadowing itself. This is caused by 3d-points close to each other mapping to the same pixel. The solution to this is fairly simple. While comparing the distance from a 3d-point to the light, you can add a small buffer to stop the triangle from shadowing itself (0.005 in this case). However, this causes problem with missing pieces of shadows, as seen in figure 4 at the intersection between the gray and the yellow wall. We did not find a fully working solution for this, as we either got shadow acne, or partly missing shadows.



The biggest problem we had with shadow mapping was the fact that some shadows appeared correctly where they should, while some shadows did not, as can be seen in figure 8. We were stuck on this problem for a long time, and we thought the problem was located in the way we transformed the different coordinate systems between each other. However, in the end, the solution turned out to be much simpler than that. The problem was the order in which we looked for shadows and drew each polygon. Before, we checked if a triangle was shadowed, and then drew it; triangle by triangle. The issue here is that if we, for example start by drawing the floor, it will not be shadowed, as nothing else have been drawn yet and is therefore not shadowing the floor. This may sound simple, but it is a relief that we solved it, since we had been stuck on this problem for quite some time. The solution was to simply draw the entire scene from the light’s point of view and filling up the lightBuffer matrix before drawing anything from the camera’s point of view.

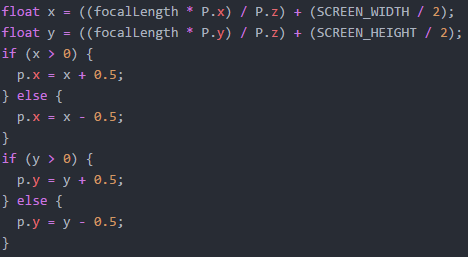
Something interesting we ran into during the project was treating the light source as another camera. You can easily get to see how its point of view looks like by removing PutPixelSDL from the DrawPolygonRows, and instead adding it to the shadow mapping. When we first did this, the light’s point of view looked like figure 9, and the resulting shadows like figure 10. The shadow on the blue cube is missing black pixels.



*Figure 9 and 10*

This is because when we created the shadow mapping, we interpolated the rows in the polygons by the amount of pixels seen from the camera’s point of view. This caused rows that were almost orthogonal to the camera to have very few pixels in them, causing them to look like they were missing pixels from the light’s point of view. An example of this is the right side of the red cube in figure 9. We solved this by having the pixel struct also store the pixel mapping from the light’s point of view and interpolating between them instead, causing it to look like figure 3. With this fix, the shadows no longer had missing pixels in them.

A general problem we encountered was with C++ and how converting from a float to an integer worked. Initially, we thought that when doing this, the float was rounded to the closest integer. However, this is not the case, as converting a float to an integer only strips the decimals. This is for example a problem when interpolating pixels. We solved this by adding if-clauses that checked if the float was positive or negative, and then added or subtracted 0.5.



**Further Improvements**

There are many ways of perfecting and improving a rasterizer. Some are connected to the methods we have implemented in this project, shadow mapping and clipping, while other are different methods that enhance the rasterizer in other ways. For shadow mapping, further improvement would be to not have any artifacts caused by self shadowing while at the same time having no gaps in any shadow. For the clipping, the first step would be to get it to work correctly. As discussed before, it would also be a great improvement if we could use the clipping to have the light be placed inside the room. There are also improvements to be made in the optimization part. As we are working with a program that would be used to render images quickly, speed is a great factor. During the project we have made optimizations that have reduced the time it takes to render the scene significantly, but there are still more that can be made.

In the beginning of the project, we discussed different things you could implement in a rasterizer. One of the most interesting things we came up with was adding textures. Adding textures would increase the possibilities of what you can use a rasterizer for drastically.

**References**

1. <https://en.wikipedia.org/wiki/Rasterisation>
2. <https://en.wikipedia.org/wiki/Shadow_mapping>
3. <https://en.wikipedia.org/wiki/Shadow_volume>
4. Niedenthal, Simon. Volume 35, Issue 3, p.249-254, 2002. *Learning From the Cornell Box*
5. <http://digitalrune.github.io/DigitalRune-Documentation/html/3f4d959e-9c98-4a97-8d85-7a73c26145d7.htm>