Soft Shadowing Comparison

Report

Owain Davies

Abstract

This report covers the design, implementation and testing of a workbench and of two image-based soft-shadowing algorithms implemented within that workbench, namely Percentage Closer Soft Shadows and Backprojected Soft Shadows in order to discover which is the best with regards to their speed and visual quality.

Acknowledgements

I would like to thank my family and friends for having to put up with my unavailability over Easter, and to Professor Ralph Martin for having to put up with me turning work in late.

Contents

1. Introduction/Background

This report covers an investigation into real time shadowing techniques, in which two specific algorithms are implemented and compared against each other with regards to their rendering speed and shadow quality.

a) Summary of primary aims and goals

These were my goals:

- Create a suitable real-time workspace with functionality to allow me to:
    - Decide upon, implement and test two soft-shadowing algorithms using appropriate test criteria and test cases,
    - Be able to implement an arbitrary number of the above,
    - Evaluate any produced results in many different ways e.g. using frames per second, or with precise camera controls that allow closer inspection of shadows.
- Create a series of test cases in order to be able to evaluate both algorithms' efficacy using the criteria I decided above
- Run and record the results from these test cases.
- Write a report containing my results and thoughts.

b) What I wanted to achieve

At the beginning of the project, I hoped to show which of the two algorithms was best with regards to quality of shadow, and their rendering efficiency.

I hoped to have certain deliverables available during the project's course in order to keep me on track, namely having the workbench up and running by the time the Interim Report was due at the halfway point, and to have decided upon the algorithms I was to implement in order to be able to do research into their implementations. The final deliverables were to finish amending the workbench given the fruits of my earlier research into algorithm implementation requirements, and the construction of a working testing system.

c) What was reported on before

At the halfway mark of the project, I produced an interim report, in which I detailed the majority of my early research and design for the later implementations in this report. The interim report covered all of my exploration into general real-time shadowing, as well as several different methods for creating soft shadows. The report also covered a few implementation details of the workbench, but no code was discussed.

The two algorithms I have decided to implement are Backprojection Soft Shadows (BPSS) and Percentage Closer Soft Shadows (PCSS). This is because PCSS is the current best solution for high quality soft shadowing in real-time rendering and Backprojection Soft Shadows is considered the cutting edge in the field.[1]

1. The Workbench

This section discusses the workbench that was created in order to facilitate the creation, running and testing of the algorithms.

a) Design

It’s able to render any number of scenes and correctly deals with using the number keys to swap between them quickly. It has a fully functional camera that allows the user to move about in the scene using WASD controls, and by moving the mouse. As well as that, it has the ability to print arbitrary text to the screen – a feature I will use for seeing an on-screen framerate, which I can also output to a text file.

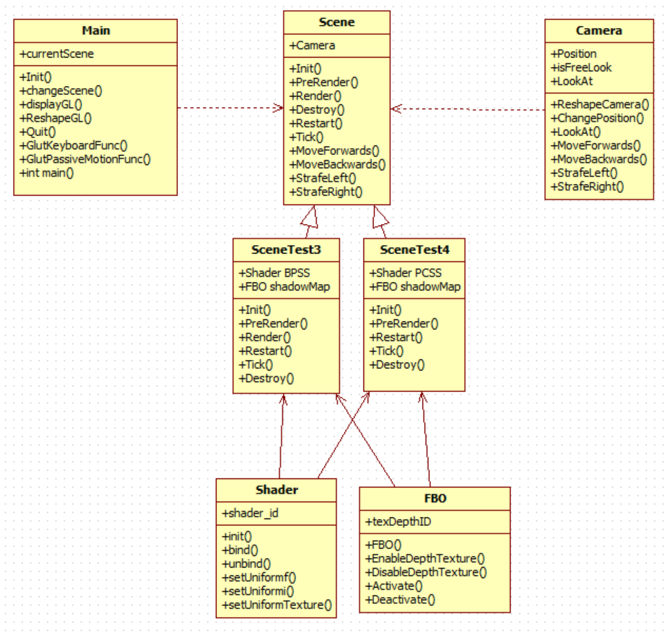
Finally, I had implemented a few test scenes. The following section is a representation of the format each test scene has.

b) Implementation

Each scene was a subclass of a class called Scene, which held a global camera object. This would allow the camera to keep its position, even after the scene changed unless otherwise specified. Each subclass scene had two important functions, Init() and Render().

Init() set up the variables that each scene would have to deal with. I also here move the camera using the two functions I created to do this, changePosition() and lookAt(). Render held all the code to be drawn to the scene.

The main class held a pointer to the current scene, currentScene, the value of which would be swapped when moving between them using changeScene() which created the new scene and called its Init() function, had a method that dealt with keyboard input called glutKeyboardFunc() which was passed to the camera, and otherwise had an initialisation function which created the window and began the OpenGL loop, glutMainLoop(). This loop then called displayGL(), a function that took the current scene and called its Render() function.



Abbreviated Class Diagram with only important features and functions

2. The Algorithms

a) Percentage Closer Soft Shadows

i. Theory

This section discusses the ideas behind the design of my first algorithm, Percentage Closer Soft Shadows. It's a technique that builds upon basic real-time shadowing theory, which is discussed in greater detail in the interim report. Percentage Closer Soft Shadows (PCSS), is an image-based shadowing technique, meaning nothing is required to be generated in order to be able to render the shadows. It was first proposed by NVidia in 2008[2].

It's an improvement upon the Percentage Closer Filtering (PCF) method [2] which itself is built upon the most basic shadowing algorithm I discussed, shadow mapping. Shadow mapping utilizes the depth buffer used in rendering as an analogy to what each camera or light can see from that position. The depth values of each point in the scene can be compared in order to find depth values that are further than those away from the light than those the light can see, and as such should logically be in shadow, as in the area labelled **A** in Figure 1.

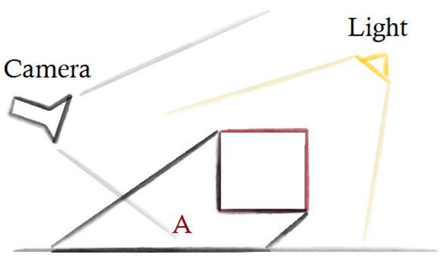


Figure 1

The method in which the values are compared is done by rendering the scene from the light's point of view into a texture, known as a shadow or depth map, which to which values are compared to during the second pass.

PCF builds on this by taking that depth texture, and after comparing the respective values as with traditional shadow mapping, and if the point is deemed to be in shadow, the algorithm blurs the values around the point to be shaded using an n x n kernel window. This gives the shadows a softer edge, but is quite computationally expensive, and so whilst enlarging the kernel window gives better results, the downsides are a substantial drop in speed.

The problem with this result is that it does not mimic reality very closely. Whilst it works in shadows that are distant in the scene, up close the shadows begin to look false under certain circumstances. As discussed in the interim report, this could cause problems with how a scene might read, as shadows give one of the most important visual cues as to the location and size of objects in the scene relative to one another[3].

The reason PCF is not capable in certain circumstances is because this technique does not allow for any contact hardening to occur. Contact hardening is when an object gets very close to a surface, its shadow at that point is very distinct and not blurred, whereas if an object is distant to a surface or a light casting on an object is far away, its shadows are blurred proportionally.

$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

PCSS is a solution to this problem. Using a variable PCF kernel size dependent on this proportion, calculated by the equation above, shadow edges should become far blurrier as distances to shadow-receiving surfaces increase, or distances to casting lights increase. This should also be as fast, if not a faster equation than an equivalent-quality PCF solution, as only certain parts of the scene are calculated using a high-value PCF kernel window.

Note that this algorithm evaluates the distance of a blocker to a point by taking the average of the depth values around a point, rather than for every point[4], meaning there's a distinct loss of precision in the values returned by this function. This means the shadow itself is only an approximation of actual geometry in the scene.

Now that the theory behind the algorithm is understood, it's time to undertake the problem of how one might design a system to run it.

ii. Design of PCSS

This section discusses what exactly the algorithm required, and what OpenGL functionality satisfied the design criteria.

The method of comparing the value in the shadow map against the second value is done using matrix multiplication. This is required because OpenGL uses multiple matrices to translate 3D points into 2D ones for rendering for each object. These matrices are usually referred to as the View (or ModelView), and Projection matrices.

There are also multiple spaces for objects within a scene to be in. Every object on the screen uses its own relative, affine coordinate system, such that usually the origin of each object is at its centre. In order to get into that object's space from global world space, which in this case might be done in order to find out the location of each vertex in that object, you must multiply by its ModelView matrix. If we were to want to move from a space back into world space, we would have to multiply by its inverse matrix, and so on.

The projection matrix is used during the OpenGL function gluLookAt(), and thereafter for rendering a scene from a camera, and it is used to translate each 3D point in the scene from its perspective into 2D points on its near plane, which is then passed further down in the rendering chain to be rasterized and displayed on the screen.

A camera's View matrix is usually referred to as its Eye matrix, and getting into its space is something that this algorithm requires. The algorithm also needs to know this matrix for the light's pass. The biggest problem during the comparison of these values comes from the actual comparison of the values. This is because of the projection matrix modifying the values that the View matrix would give, in order for the points to be in the correct 2D position. Whilst ordinarily useful, in this case it makes it more difficult to compare the values correctly. The solution is to multiply each value in the camera's space by the camera's inverse projection matrix, in order to get the points into camera View space, then again by the camera's inverse view matrix in order to get it into global world space, and then multiplying them by the light's ModelView and Projection matrices to finally have the values in a location where they are able to be compared.

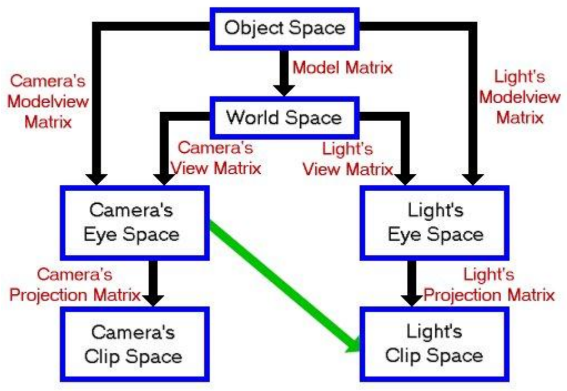


Figure 2

Rendering from a point, such as a camera is done by the function gluLookAt(), which takes three values: the position you wish the rendering object to be at in world space, the position you wish the rendering object to be looking at, and which direction is considered to be up. This function creates the required ModelView matrix for the camera. The perspective matrix is created using gluPerspective(), which takes in the field of view, the resolution and the near and far clipping planes of the camera. The resolution is tied to the field of view, and so having a smaller field of view gives a lower resolution higher fidelity. If an algorithm is particularly complex, this would probably be a useful feature to use.

Once the matrices have been created, I then required the access and storing of them into variables. This is done by glGetFloatv(), which takes in two arguments – the OpenGL feature you would like to extract information from, and the location where you would like this information to be stored. In this case, I'd like to take in GL_MODELVIEW_MATRIX and GL_PROJECTION_MATRIX.

As rendering to a texture was required, OpenGL's texture functionality was used. OpenGL's texturing system also has the ability to save just the depth component into a texture, which saves on space and in-shader complexity. When setting up a texture, the function glTexImage2D() is the main feature. It takes in information like what sort of texture format you wish to use, in this case a GL_TEXTURE_2D. It also takes in information like the width, height and depth, as well as the internal format – which in this case should be GL_DEPTH_COMPONENET over the usual RGB.

FrameBufferObjects are the other necessary feature to any implementation of this algorithm. They facilitate rendering to an off-screen target (like our previous depth texture, for example). This is done by simply creating one using the glFramebufferTexture2DEXT() function and passing in the ID of the texture.

I am using shaders because they are useful due to the control they give to users in how exactly each pixel is dealt, and allow for far faster iteration times are once set up. They do, however, need to be read in, compiled and linked in order to check for errors. This requires an ASCII file reader and further complexity. They are also not easily accessible OpenGL features, identifying themselves in the code with horrifying strings of characters which a programmer must bind to a slightly more usefully named function, and then use those. As most programmers don't want to do that, the features require additional libraries to run like GLEW (the OpenGL Extension Wrangler) or GLEE, which do most of the hard work for you.

A big problem with shaders over fixed-pipeline implementation is that because they're only available in more recent versions of OpenGL, the program is then much less compatible with older graphics cards, and due to the nature of OpenGL, some functions may not be compatible with recent other operating systems. In order to ensure that the program runs correctly, further checks are required for compatibility of each extension.

iii. Implementation of PCSS

This section discusses how I implemented the design discussed above.

The first thing I implemented on top of what I had was a shader system. This required the creation of an ASCII file reader which takes in files of my creation with a particular extension for distinction. It then takes the files read in, and compiles and links them into what are known as programs. I then implemented a Frame Buffer Object (FBO) class which created and held a pointer to the texture.

Finally, the last new class that I had to create was that of the light itself, which was essentially nothing more than a container for the light's position, size and its projection and view matrices. I added similar functionality to the camera's class so that it too could hold its matrix values within easy reach.

I amended the scene class to have a PreRender() function, to separate true rendering from the camera with the filling of the FBO, and made sure Render called it first. I made sure that front face culling was on during this stage, so that there were no self-shadowing errors on the objects in the scene when it came to render them properly.

Within Init(), I created a light, a shadow FBO with a resolution the same size as the screen resolution for high fidelity shadows, and a shader, which loaded two files: *PCSS.vert* and *PCSS.frag*. I used another feature of Uniform variables in shaders, which is to grab the location of where a future variable would be, to know where the shader would expect the shadow map texture to be, to which I passed the shadow map later on.

Lastly, the most important two functions, PreRender() and Render().

Within pre-render I activated the FBO to ensure that the texture is bound and will now be rendered to, and make sure to unbind the shader so as not to tamper with the values being rendered to it. I set backface culling to be off, to only render the back half of all objects in the scene, so that only these depth values are in the map, in order to reduce self-shadowing on the objects which can cause ugly artifacts, and then I rendered the scene as normal. After rendering for the first time, I passed both the light's projection and view matrix to the Light object, to ensure correct processing in the next function, and deactivate the FBO to unbind the texture.

Moving back to the main Render function, I clear the buffer to get rid of the depth values now in it, and bind my shader. Now is when I use that uniform location I made sure to grab before for the shadow map and pass it into the shader into the 7th texture location OpenGL allows for. I re-enable the FBO in order to re-bind that shader, only now it will be to read the values. I set the perspective to what the camera will now be seeing, and render from that perspective. I now pass both the camera's view and projection matrix into its object, for processing in the next step.

The final step is to pass in the matrices I've stored into the shader, set the culling face to be the back-half again, and render the scene.

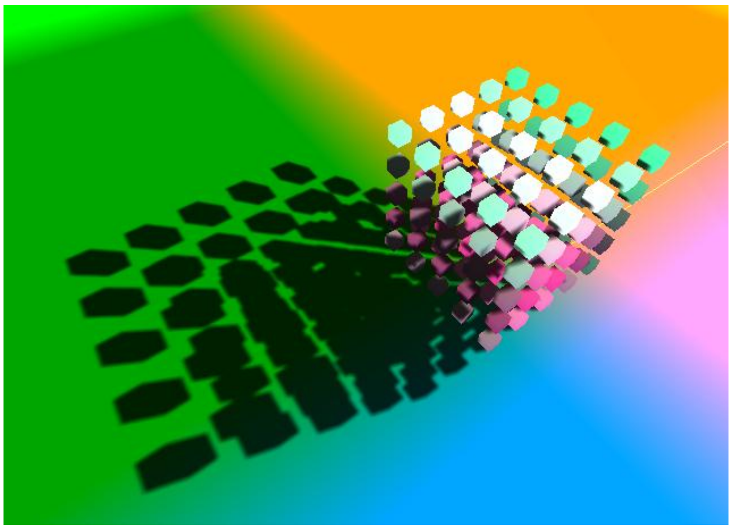
The vertex shader does four things. Firstly it uses the GLSL function ftransform() to store the position value of the camera within the range [-1,-1] to [1,1]. This means in order to get the light space coordinates of this value, we must multiply it by the light's projection and view matrix, and the inverse of the camera's view and projection matrix. This value will be used in the fragment shader for comparison.

In order to make the scene more easily readable, it would be useful to have a range of colours throughout, and this is done easiest by varying the colour per pixel by its value in space. I pass this by using the other GLSL-only variable type: Varying values. These are values which may be passed between shaders, but are otherwise read-only. Lastly I set the colour of each vertex to be the colour I set within OpenGL itself.

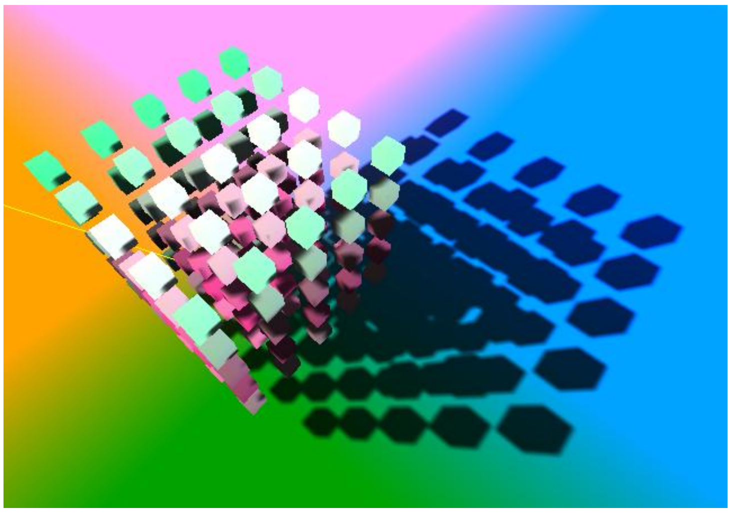
The fragment shader is more complex, but has four main features. Firstly, it searches for any potential occluders in the map from the point passed by the Vertex shader by searching around the map from that point for depth values that are closer to the light than they are to the receiving point. It sums up the values in this search and returns an average depth for this occluding surface, and is done by the function findBlockers(). Then, if an occluder is discovered, the shader calculates how big the penumbra should be using the PCSS distance equation discussed in the design section, in the function penumbraSize().

This value is then passed into a PCF filter with a variable kernel size, which returns how shaded the point in question should be. The shadow is calculated by shadowCalc(), which is an exact implementation of shadow mapping. This value is multiplied by the varying I passed earlier, Pos, in order to calculate the final colour of each pixel.

These are the results of the PCSS algorithm:



And, as a comparison, this is PCF:



b) Backprojected Soft Shadowing

i. Theory

Backprojected Soft Shadowing (BPSS) is another image-based soft shadowing algorithm, first proposed in 2006 by Guennerbaud[5], and as such is a far more recent solution to the soft-shadowing problem. It's quite different to PCF-based blurring of the final image, but quality-wise PCSS is the closest comparable algorithm. The basic idea is this: as with PCSS we render from the light's perspective off-screen into a texture, and compare both values against one another to find a case where the light's depth value is closer than the camera's depth value, again as before. Once we have found a point where this is the case, the algorithm changes quite drastically.

The assumption is now made that each of the values in the light's depth map – that is, the values in the depth texture– should be considered to be texel-wide quads floating in space at their respective depth values, parallel to the light's plane. Essentially, the depth map becomes a one-dimensional discretised approximation of the scene. Then, at least in the original algorithm, every value in the depth map is projected, or pushed away from the point to be shaded, until the quad is on the light's surface. The area that is hit is analogous to the percentage that individual depth texel is shading the point we wish to shadow. This is done for every single value around each pixel deemed to be in shadow, or can be restricted to a search kernel shown as green in figure 3, and the total occluded area of all backprojected texels is summed up and becomes the new shaded value of that point.

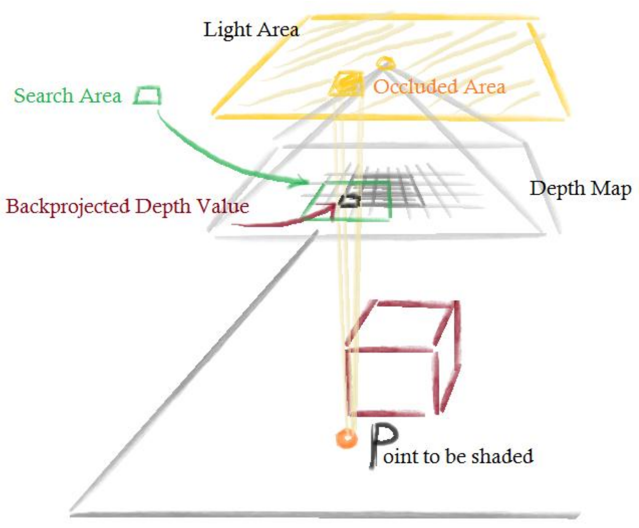


Figure 3

Clearly, this is very computationally expensive – each texel within the depth map does thousands of calculations, and if the scene is complex and thus there are many points deemed to be in shadow, this could be prohibitively complex to the point where the program would crash and reset the driver.

The solution proposed within the whitepaper is to generate what is known as a mipmapped, min-max depth texture, or Hierarchical Shadow Map (HSM) – a secondary texture is created which contains values a blurred version of the original high-resolution depth texture. This blurring is done very slightly, using a 2x2 or 3x3 blur kernel. The highest and lowest values of this blur function are then saved for each value, essentially propagating depth values throughout the map. This can then be read first, in order to see if its value requires shading, essentially reducing the size of the map to be searched dramatically.

In further detail, the projection algorithm itself is as follows:

$$B = \begin{bmatrix} b_{left} \\ b_{right} \\ b_{bottom} \\ b_{top} \end{bmatrix} = \begin{matrix} \left(u_s - u - \frac{1}{2}\right) \\ \left(u_s - u + \frac{1}{2}\right) \\ \left(v_s - v - \frac{1}{2}\right) \\ \left(v_s - v + \frac{1}{2}\right) \end{matrix} \frac{w}{n^*r} z_s \frac{z}{z - z_s} \frac{1}{l}$$

This equation calculates the bounds of each projected quad **B** using the depth values **u** and **v** are the **x** and **y** coordinates of the point to be shaded, and u_s and v_s are the coordinates of the depth sample from the shadow map. Similarly, z and z_s are the depth values of the point to be shaded and the sample point respectively. The light's near plane is also taken into account, which the algorithm assumes the shadow map to be. The shadow map's distance from the light's center is **n**, and its width is **w**. Finally, the shadow map's resolution, **r**, is taken into account, and is recommended to be a power of two in order to create the HSM easily.

The first half of the equation is what estimates the location of the projected point, where the ½ added on and subtracted is to blow the shape to be a unit quad. The following terms are, according to the whitepaper, used to normalise the value of the area of the quad such that it returns a percentage. It does this by scaling the value of the light such that it's considered to be a unit light, meaning that any values that are subtracted from this area would naturally be between [0,1] and thus the percentage covered.

Many improvements can be made to the algorithm to increase the speed and the correctness of the shadows it produces. Two of the problems the whitepaper talks about that this algorithm causes is under-occlusion. This is when two adjacent pixel values are a great distance in the z direction from each other, which could occur at a very steep incline in an object, for example. Points searching from underneath these depth values may see the gap created as a non-occlusion point, meaning the shadow at that point would be lighter than it should be.

The second is over-occlusion. This is when a depth value is incorrectly projected onto the light twice, causing the shadow to be darker than it need be. They and many others since have proposed many different ways of improving upon these problems, by adjusting the widths of the projected texels to gap-fill between texels to prevent under-occlusion. A proposed solution for over-occlusion is by adding each of the areas up for each point onto the light, but by representing the light's area as an array or mask of bits, known as a bitmask, which these projected texels would then switch on when hit. The key change would be that each texel will be logical-ORed into the array, meaning it would be impossible for over-occlusion to happen.

It's worth noting that I spent some time during the interim report describing the version of this implementation that used bitmasks, not at that time having spent enough time researching the base-implementation. I have since come to the conclusion that as this will be my first implementation of anything this complex, I think it would be wise of me to not attempt every fix before I see the basic implementation itself. Then, if I had time after getting this to function properly was so egregiously ugly, it might be worth me spending time implementing further recommendations.

ii. Design

No changes are required to my previous implementation, besides passing in the near and far clipping planes as uniform variables

iii. Implementation

The majority of the setup behind creating and rendering the scene was essentially identical to the PCSS implementation, using PreRender() and Render() to create, render to, and compare values against the FBO with the values in Render(). The only addition I had to make was inputting the width of the light, and its resolution (which would equate to the resolution of the shadow map texture). The shadow map size had to be significantly reduced from several orders of magnitude greater than the resolution of the window, to the resolution of the window or less, as I knew that traversing every single pixel thousands of times would be pretty computationally expensive.

The shader takes in the depth value of the coordinate to be shaded, searches for depth values less than it, as with PCSS. It then calculates the total area these potential occluding patches would cover if they were projected back onto the light's surface. In my implementation, this is done by a function called computeBackProjectedBounds(), which calculates the size of each of the sides of the rectangle of the backprojected pixel.

ComputeBackProjectedBounds() also uses a scaling variable, the function of which the whitepaper mentions is to scale the area of the light such that it is considered as a unit light (of size 1 in world space), meaning that any area taken from it would be returned as a percentage.

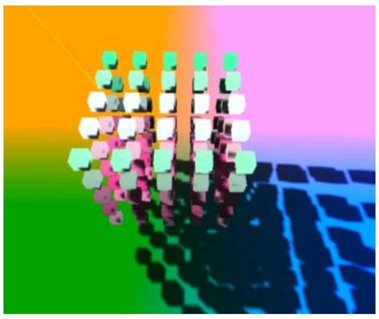
Evaluating the area of this rectangle, done in my implementation by a function called computeBounds() which returns a float for the area in world units, equates to the amount of light that would be blocked by this depth value.

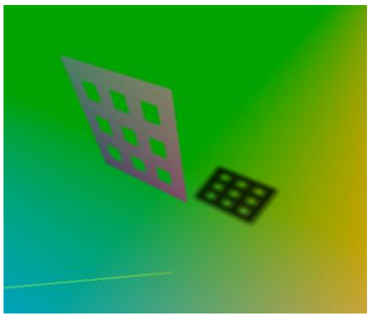
This is done for every other value in the search kernel, and the total area is summed up and removed from a float value called visibility – which in turn is multiplied by the colour of the pixel to be shaded. This gives a far, far better representation of the shadowing on the scene and some very soft shadows.

3. Implementation of Test Scenes

a) Test Scenes

I implemented two main test scenes in my testing, one mainly for speed, and another for quality only. The first was a cube made of smaller cubes, in an n x n x n formation, in which I could vary both the size of n and had the ability to add in cubes one at a time, which I did using time as the varying factor. This was implemented with a varying for loop, the size of which changed depending on a variable numCubes which changed with time.





The other test scene I used was that of a waffle-shaped object – that of a quad with many holes in it – in order to see soft shadowing artefacts easier. This was implemented simply with glQuads() and three for loops to calculate the positions of each of the points.

4. Testing for Algorithm Speed

a) Test 1 – Algorithm Scalability

i. Goal

The goal of this experiment is to discover how both algorithms scale with the complexity of the scene. It should tell me which was the best generalist algorithm that should be employed, or perhaps give particular situations as to where an algorithm would perform best – outdoor scenes with not very many large objects, or indoor scenes with lots of small objects.

I used cubes as the object the measure with, as they were quick to implement.

ii. Execution

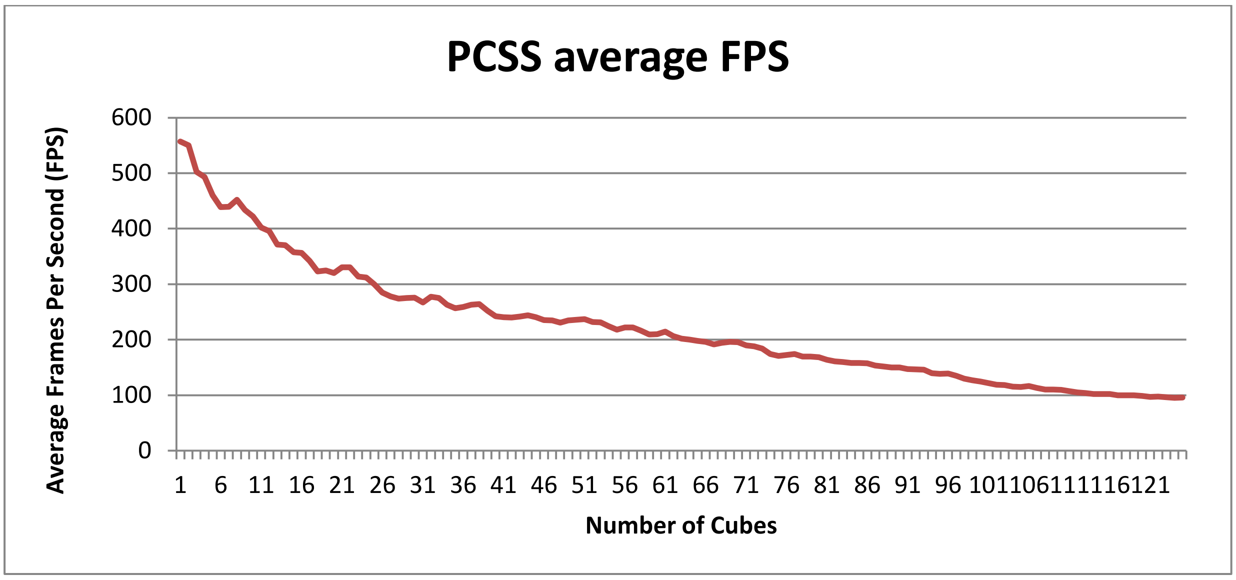
I began the test with no cubes in the scene, and then steadily increased the number of cubes in the scene with time until I reached a 5^3 cube of cubes, each an equal distance from each other, and each of the same size. I recorded the frames per second from the beginning to end, and averaged all the values for each cube in order to see a normalised value for each level.

The light in the scene is projecting into the center of the cubes from the right, meaning the shadows are to the left, and does not move for the duration of the test.

b) Results

i) Results for PCSS

Before any cubes were in the scene, the average FPS was 556.9. This trended sharply downwards as the first layer of cubes was added, at and 16 cubes began to decrease at a slower rate. At 125 cubes, the average FPS was 94.0 FPS, giving the dataset a range of 462.9, which is a big difference in frames.

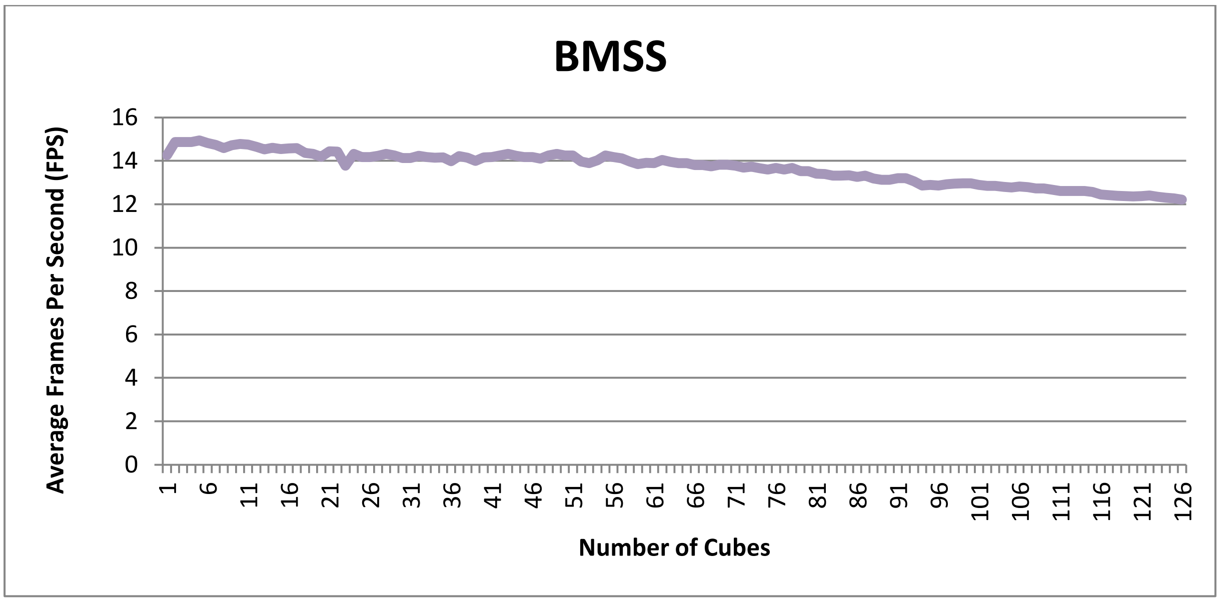


ii) Conclusion for PCSS

This is a very fast algorithm, but appears to be severely affected by the number of objects in the scene that search extra areas of the depth map. It's possible to see the point on the graph where new cubes are added on top of other ones, which is at 16 cubes. This is because each additional cube beyond that point are not searching the shadow map as extensively, meaning less work has to be done by each individual cube thereon in.

iii) Results for BPSS

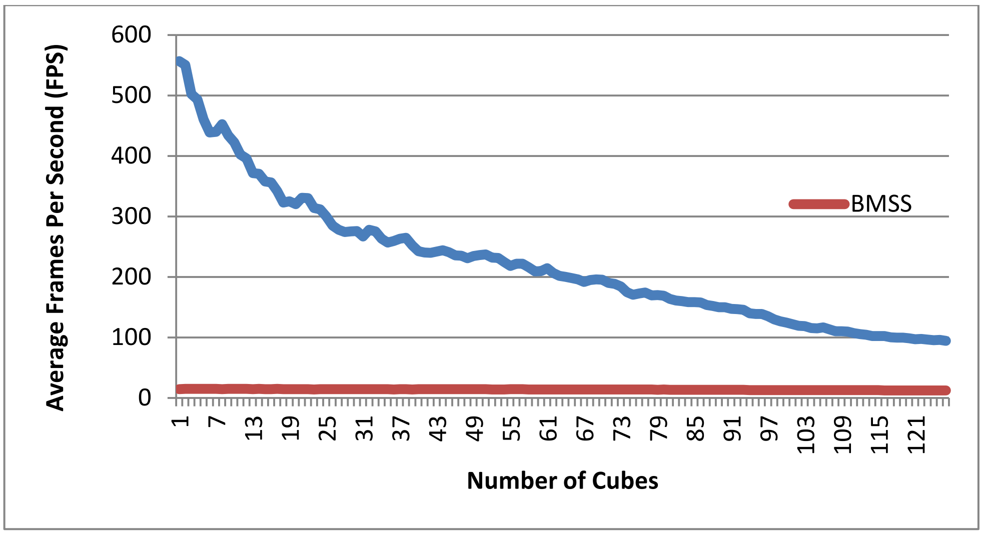
Before any cubes were added, the average FPS was 14.26FPS, shrinking down to 12.2FPS at 125 cubes. The rate of decrease seems to be consistent throughout, and the range is only 2.06FPS.



iv) Conclusion for BPSS

The first thing to note is that this algorithm runs far, far slower than PCSS. The other interesting feature is this algorithm appears to be much more scalable, with a range of only 2.06FPS in the dataset.

v) Comparison of Both



vi) Conclusion of Comparison of PCSS with BPSS

There are two conclusions that can be drawn from this test.

Firstly, PCSS is by far and away the faster shadowing algorithm, by several orders of magnitude. Secondly, and perhaps more interestingly, it's the worse algorithm for scalability. The range of FPS values from 1 cube to 125 cubes is far greater with PCSS than it is with BPSS, meaning if the overall speed of BPSS was increased to that of PCSS, BPSS might well be the best overall algorithm on both counts. However, as no improvements have been made to the base algorithms, it's fair to conclude that PCSS is the best.

b) Test 2 – Comparison of PCSS with PCF

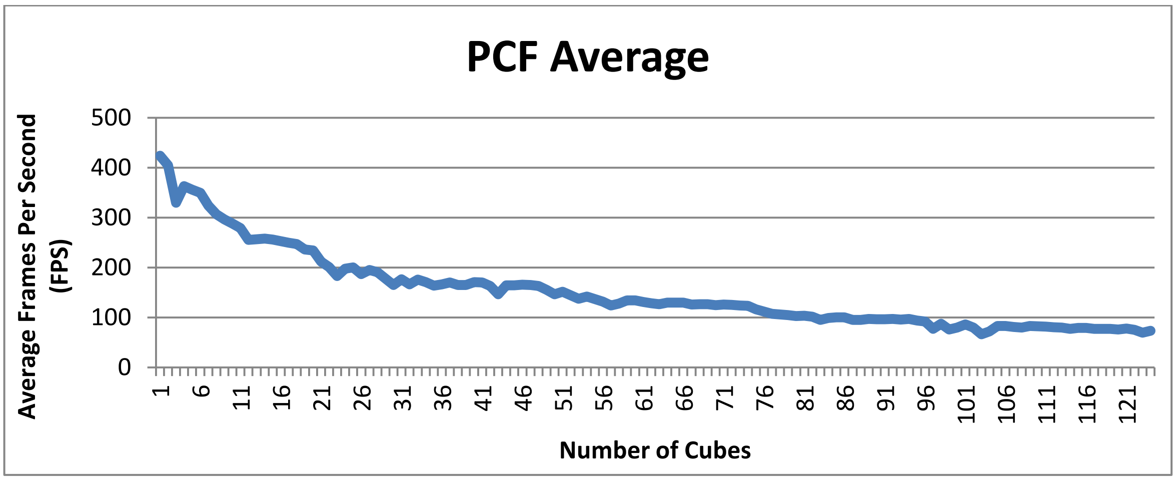
i) Goal

This test should show whether it is worth implementing PCSS algorithm over PCF. As the algorithm is supposedly an improvement over the PCF algorithm, it would be interesting to compare their outputs to that of a PCF filter with comparable quality.

ii) Execution

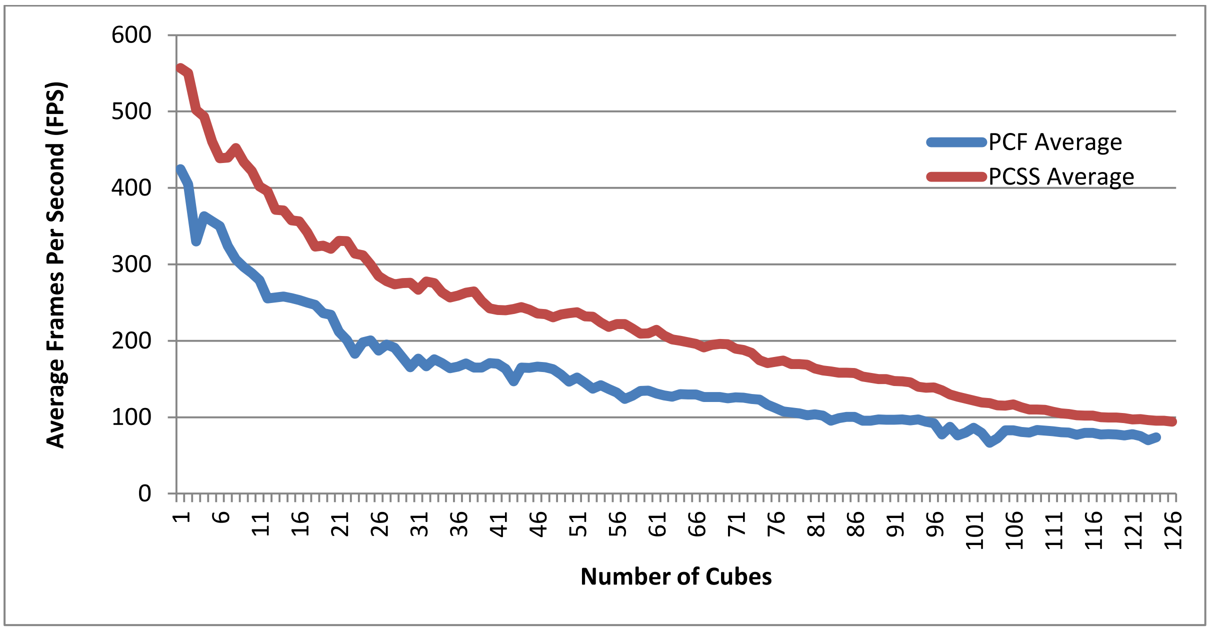
I am using the same test scene as before, and I am comparing the values I attained in the earlier test with that of the test scene running with the PCF algorithm with a kernel window of 15.

iii) Results for PCF



The results for the PCF algorithm on its own seem very similar to that of the PCSS in terms of shape and features. However, the highest and lowest FPS differ – 424.41 and 75.18 respectively. The range differs too, at a smaller 349.23 FPS.

iv) Comparison Graph



v) Conclusion of Comparison of PCF with PCSS

It's interesting to note that PCSS is faster in all cases, but seems to be affected more by each additional cube. This may well be due to the fact that additional processing is required to evaluate the PCF kernel size for each pixel. Given the data set I've produced, I think it's conclusive that PCSS is better in this situation.

vi) Conclusion of Speed Tests

Given the previous two tests had PCSS as the overall better algorithm, it seems that PCSS is the best solution for soft shadowing if it is required, even over the current industry standard of PCF.

5. Testing For Algorithm Quality

a) Test 3 – Simple Shape Quality

i) Goal

The goal of this experiment is to discover which algorithm looked the best in different circumstances. The best shadowing algorithm should be able to look plausible in as many circumstances as possible, so the algorithm that will succeed this test will be that which looks the best in the most situations.

ii) Execution

I use a simple scene with a single cube in it, and compare the outputs. The light object in the scene will not move.

iii) Results

a) PCSS

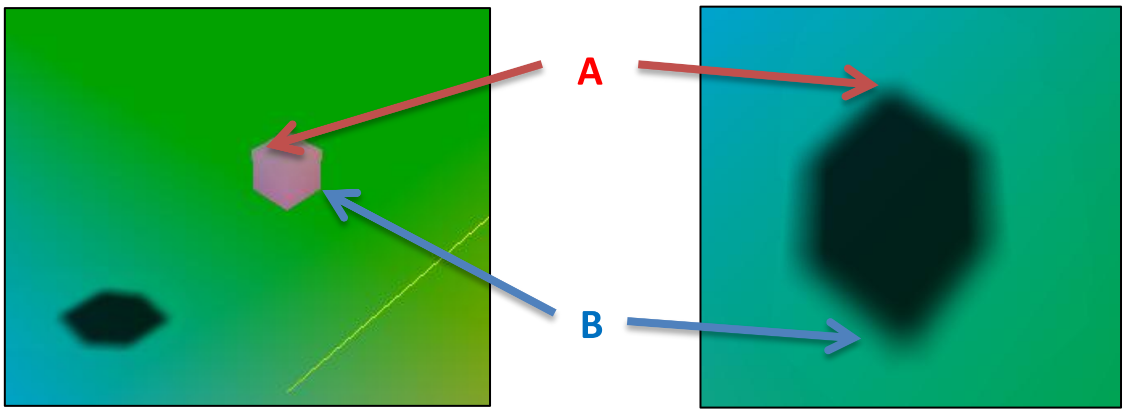
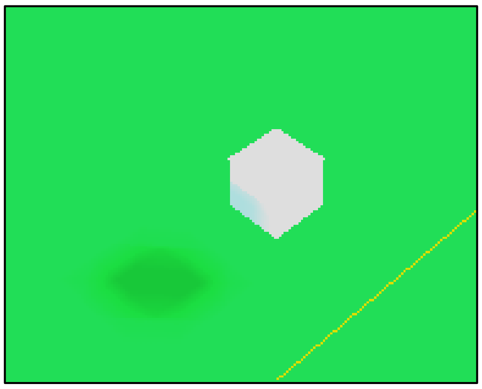


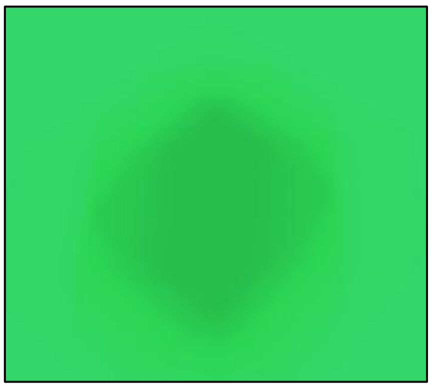
Figure 3

Figure 4

The main body of the shadow is dark, and the edges have a PCF-like blur to them. It's worth noting, however, that the top edge has a less soft blur on it than that of the bottom of the shadow. This is due to the fact that the point being shaded is closer to the receiver at the top point of the cube, labelled A on figure 3, than point B.

b) BPSS





BPSS soft shadows seem appear far softer than PCSS, even to the point of appearing to be very indistinct and almost unnoticeable. They appear to be more for the very soft shadows that occur in scenes with lots of light and very few cases of actual darkness. It's also very difficult to see any contact hardening occurring in this scene whatsoever.

iv) Conclusion

Both algorithms produce attractive results, with no jagged edges or any artefacting whatsoever. The key difference between the two is the softness and the contact hardening that occurs. Within PCSS it's far more apparent that the shadow blurriness changes as the distance of the shape changes, and as such PCSS seems to be the closer shadow to reality. Both should be entirely applicable within different scenes, however – BPSS should simply be used in scenes where there is very little true darkness for a very high scene quality.

b) Test 4 – Complex Shape Quality

i) Goal

The goal of this experiment is to discover which algorithm looked the best with more complex scene objects, like the waffle shape I created. The best shadowing algorithm should be able to look plausible in as many circumstances as possible, so the algorithm that will succeed this test will be that which looks the best in the most situations.

ii) Execution

Here I used the quality test scene I created with the complex waffle shaped object, one that has many facets that should show details not possible to show with a simple cube. I ran and checked visual quality of both to compare.

iii) Results

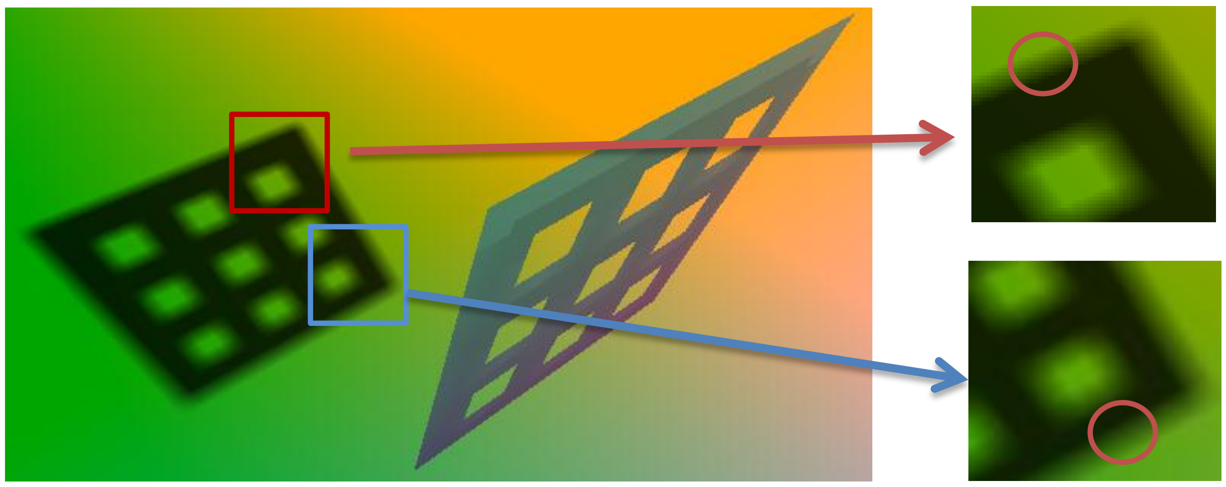


Figure 5 – PCSS with Waffle

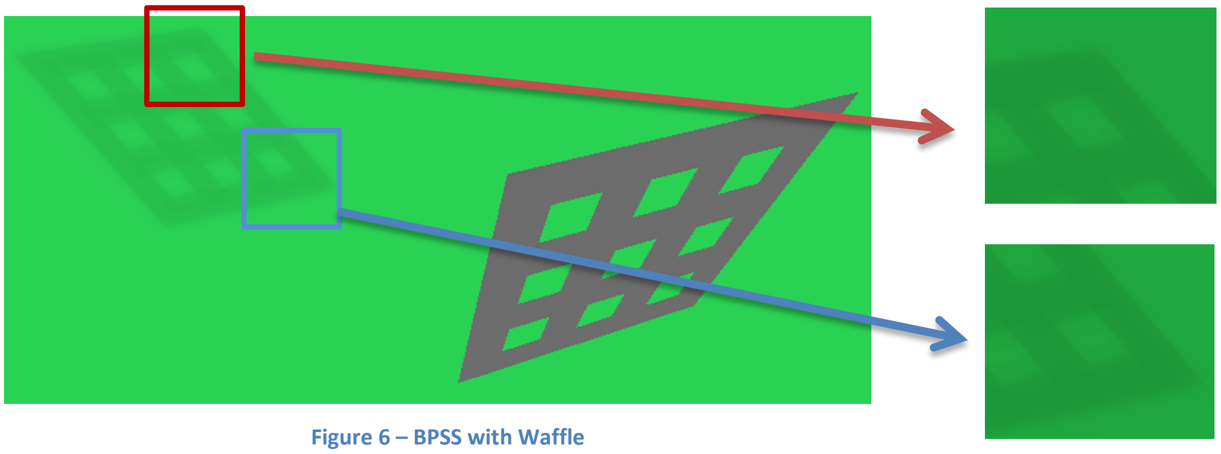


Figure 6 – BPSS with Waffle

It's quite obvious here the differences between the two algorithms. Whilst BPSS looks soft, it doesn't appear to vary the softness as much as PCSS does – compare the width of the penumbrae at the shadow edges in Figure 5 at the top with the bottom. BPSS does not appear to vary at all. Otherwise, most of the results seem the same as Test 3.

iv) Conclusion

After comparing the softness of the edges at the tops and bottoms of the complex waffle shape, it's clear that PCSS varies the penumbra edge in a much more significant way than BPSS does. As mentioned earlier in the theory section for PCSS, this gives visual clues as to the position and rotation of the object in space. For this test, I would have to conclude that PCSS has the higher quality shadow.

v) Conclusion of the Quality Tests

The quality tests are naturally more subjective than the speed test, as the data is all entirely visual. It's interesting to note how different both algorithms emulate the softness of shadows, and how stark the contrast is between them. Given that in test 3 it was shown that PCSS has overall darker shadows and test 4 showed the blurring was more realistic and appealing to the eye, it has to be said that PCSS is the better quality-wise.

6. Final Conclusion

Looking at both testing categories, it's remarkable how clear cut the better algorithm is. Not only is it the fastest algorithm overall, it's also the better looking one, and can be used in both simple and complex scenes to add a large amount of quality. The only thing it isn't competent at is scalability, which is worth noting for scenes with very complex depth maps.

Also given how BPSS requires further implementation efforts to be able to properly compete, it's also the simplest algorithm to implement, which gives several benefits to any programmer wishing to implement an attractive, well-functioning and fast algorithm quickly.

7. Evaluation

a) PCSS Design and Implementation

The idea behind shadow maps didn't come across as intuitive to me to begin with. The fact that the depth value is stored in each fragment's z value, but that in order to access the value you must divide by its w value is obvious to me now, but definitely not as I was attempting to understand.

The stage in the implementation that I had the biggest trouble with, both conceptually and with the number of bugs was trying to implement shaders and framebufferobjects. I think the reason behind this was due to the sheer number of extra functions there were behind them – framebufferobjects require knowledge of how textures work, as well as shaders, and shaders require you to know a whole new language, as well as the OpenGL pipeline. Individually, they aren't a huge amount of knowledge, but implementing the algorithms I wished to implement meant that I had to understand each aspect to the point where I knew how to trace through each function, and keep track of the states of my program.

Another huge problem I had with shaders is the fact that debugging is entirely visual – given that the program is expected to be run every frame and as quickly as possible, the programmer's debugging crutch printf() is not available for use. It took me a substantial amount of time to get used to the new style of programming raised by comparing colour values over debug output, and spending hours tweaking individual values in order to get seemingly simple implementations to look better. The results of such tweaking are entirely evident in both of my implementations of the algorithms, with values here and there being multiplied by seemingly random magic numbers.

b) BPSS Implementation

Attempting to linearise the depth values as was mentioned in the BPSS whitepaper was an ultimately fruitless endeavour on my part, and in turn wasted a week of otherwise valuable time. This made the prospect of creating Hierarchical Shadow Maps for the BPSS and bitmasks off the table entirely. This also meant that testing was less fair in terms of which algorithm produced the nicest results, as I had spent less time tweaking the BPSS implementation over PCSS.

The biggest problem this implementation had was that it was extraordinarily slow. According to the whitepaper, it would be possible to early-out most of the values in the depth map. The problem now was that I had spent so much time trying to linearise the depth values that I had run out of spare time to properly implement it, meaning that the tests that would compare the speed of both algorithms would come out with a pretty clear winner already.

c) Test Scene Implementation

Implementation of test scenes turned out to be a bigger challenge than I'd expected – as I'd assumed output would be an incredibly easy problem to solve, I'd left it until it was almost too late to be able to thoroughly structure the chain of output such that I could use it in the way that I had hoped to. In the end the implementation boiled down to a simple setup with me passing around a pointer to a file and using fprintf() to print to that file.

For the speed-based test scenes, I decided to record the number of cubes currently in the scene, and the current FPS of the scene, as this would be the simplest way of measuring how the algorithms react when I change the complexity of the scene. I would take the average of all values of the FPS for the same value of cubes in the scene in order to get a better representation of the effect each addition had. I did consider recording the milliseconds per scene, but discovered that frames per second were far more controllable a value to be able to generate, I assume due to floating point accuracy problems.

Due to some now pretty serious time constraints, my original plan for having the tests run all automatically one after another (a feature that was half implemented to see how possible it was earlier in development) was skipped for a more hands-on approach, along with my plan for the workbench to automatically move the camera to points of interest in the scene and dump the current frames automatically into a file for comparison was changed to me using a manual camera and taking screenshots. My reasoning for this was that firstly, I hadn't the time to work out where each shadow would be in order to be able to do this and have it in sync with the timing system,

d) Project

I do feel that, looking back over the contents of this report that there was less content than perhaps it felt like before or during the creation, and some of the shaders seem perhaps even a little simplistic now. I think this is more a sign of understanding over complexity. It's only broadened my interest in the field and made me want to implement more complex ideas and concepts.

After having done the project, I feel the scope of the project itself was very reasonable to complete. I did not, however, know at the time truly how long it would take me to implement certain features, as I knew so little about certain parts of what I was asking myself to do. I don't, however, think the plan I had was realistic. I have to admit to having taken wild guesstimates as to the length of time taking to learn and implement something this new and sprawling would take. Because of the plan being unrealistic, the timescale individual parts took was radically different to what I had planned for. There was an unhappy period when nothing was happening because none of my implementation was working due to some key misunderstanding, meaning weeks went by without me being able to begin on shaders. Shader coding only began a mere five weeks before the deadline of this report, meaning that I almost had to cut Backprojected Soft Shadows altogether and go for a simpler to implement algorithm.

Another problem for the timescale is a problem I've had as a person for a few years now – motivation. I find motivation is especially difficult to come by either if the topic is utterly insipid, or if I am utterly lost as to where to begin. Thankfully the former never happened during the course of this particular project, but unfortunately the latter did almost constantly. Because of this, all the programming done for this was done pretty sporadically as and when I began understanding things, meaning a lot of what was produced is are not due to any particularly spectacular design choices.

Finally I think the part of the project I wasn't expecting was the shadows didn't ever look quite how I had expected them, even after hours of tweaking. The field of real-time rendering seems like somewhat of a black hole in this regard as to endless, endless tweaking.

e) Tools and Code

While I started out unsure of visual studio, having only had experience with Eclipse as an IDE, I grew to very much love using it for its debugger. I am still not so fond of C++ as a programming language, and I'm not sure it's very fond of me.

8. Future Work

I think there are far more test cases that can be accounted for, as well as improvements to Backprojection Shadows like Bitmasks and hierarchical shadow maps. This would make the tests more fair in terms of having the algorithms start out on equal footing. Testing in the workbench could also be far easier, with the output being generated automatically averaged in some kind of CSV format.

9. Conclusions

In conclusion, the algorithm that performs the best in speed and has the best visual quality is Percentage Closer Soft Shadows. It's a simple modification to the already simple shadow mapping algorithm, and produces very attractive results.

This isn't to say that Backprojection Soft Shadows should be ignored. With improvements to the base algorithm, it has the capability to perform as well, and gives similarly pleasing results for different lighting situations.

10. Reflection on Learning

Overall I feel good about what I've learned and felt I have achieved. It's broadened my interest in the field and made me want to implement more complex ideas and concepts.

The most important fact about myself I learned during this process is that unstructured learning, that is to say self-driven learning, is incredibly chaotic and incredibly difficult to focus towards an individual topic. Every single resource I seemed to use in order to learn everything that I needed to learn seemed to leave enormous gaps in my knowledge, big enough to stop me from being able to continue. This would mean I would have to spend far more time researching than I had expected, which lead to a slippery slope of deadline missing over the course of the project, even though I had built in time for weeks of slip.

The benefit of having now done this project is I now know my bounds with regards to entirely new things far better, and it has given me the tools to at least be able to give a better ballpark figure as to the length of time it would take me to implement something.

11. References

1. Schwarz, M. & Wimmer, M. Casting Shadows in Real Time. *Computer* (2009).

2. Fernando, R. Percentage-Closer Soft Shadows Percentage-Closer Soft Shadows (Supplementary Material). *Computer* (2003).

3. Mamassian, P., Knill, D.C. & Kersten, D. The perception of cast shadows. *Trends in cognitive sciences* **2**, 288-95 (1998).

4. Myers, K. & Fernando, R.R. Integrating Realistic Soft Shadows into Your Game Engine Integrating Realistic Soft Shadows Into Your Game Engine Why Soft Shadows are Important. 0-10 (2008).

5. Guennebaud, G., Barthe, L. & Paulin, M. Real-time soft shadow mapping by backprojection. (2006).