Soft Shadow Comparison Interim Report

Owain Davies
Prof Ralph Martin
Dr Kirill Sidorov

Abstract

This is a report exploring the merits of various soft-shadowing techniques, and then a discussion of the implementation of two of them in order to compare their results. Their comparison will come about via a program which will test the algorithms in both computational prowess and shadow quality, the design of which is also discussed within.

Acknowledgements

I would like to thank both Professor Ralph Martin and Doctor Kirill Sidorov for giving me sage advice at exactly the time I needed it, and for putting up with some of my more vacuous moments.

Table of Contents

Introduction

Overview of the Problem

Rendering at close-to-realistic levels of quality is difficult, and doubly so at real-time speeds. This report will be a discussion of two particular soft-shadowing algorithms and comparing their efficacy in both accuracy and computational speed. I will do this by first creating a workbench to implement my tests, and then introduce both algorithms, and a series of automated tests which I will use to generate the results for my final report.

The quality tests will be of a criteria I set myself, using scenes I construct myself that each display a particular facet of shadowing, or are examples of complex scenes for shadows to be rendered within. This report marks the halfway point, consisting of the research and preliminary results I have uncovered. This problem is a difficult one, even for current computing, and so I hope the conclusion I come to will be of use to both myself and others.

Aims/Goals

My aims and goals are:

1. A working workbench in order to first construct and then test my two algorithms
2. Having done research into shadowing and pick two algorithms for testing
3. Have a finished workbench, with a complement of meaningful tests
    a. In order to do so, I must understand what it is I am testing for
    b. Decide upon what the tests will be
4. A set of results and a conclusion
5. A preliminary report detailing where I am halfway through the process
6. A final report detailing everything I have done, and presenting my results and conclusions in.

Scope

This report covers the implementation and comparison of two soft-shadowing algorithms, comparing both their speed and shadow accuracy against each other. The algorithms are industry standard, both used in several different contexts for different reasons, and are chosen from a selection of different solutions to the soft-shadowing problem. I will discuss how I will decide upon and test my scenes in the two methods I have specified.

The report will not cover the implementation of any other algorithms, or explicitly use them in conjunction with any other real-time feature such as fog or particle effects.

Background

Real-time rendering

Real-time rendering is the study of creating images within a computer that are a true representation of life, at a realistic frame rate. As computing power is still not at the level which would make it comfortable to run a ray tracing algorithm, in order to solve most of the problems this report will encounter, certain approximations are going to have to be made in order to emulate reality.

Studies have shown[1] that the sense of presence, i.e. that of feeling as if you were in a space you are not, such as a computer game or virtual reality space, is most aided by the addition of shadowing over other improvements like higher resolution models and better textures.

Shadows give important visual cues as to where an object is in space the shape and material of both the object in question and the background the object's shadow is being projected on, especially when the object is in movement[2].

It shows then that shadows are important, and as such must be as approximate to reality as best as possible. As before however, approximate representations of reality are also the most difficult to calculate. It is useful, then, to know exactly what we will be recreating, in order to see if there is any way any algorithm we create can be sped up by discarding unneeded information. I will then explore different solutions to the criteria I suggest.

What do we want to emulate?

Light, and by extension shadow, are possibly the most complex and computationally expensive task in realistic rendering to get right. This is due to the inherent complexity of light. It is well known within the field of graphics that replicating light is incredibly difficult, mostly down to the subtle interactions within reality that diffusing light creates. That is to say, diffusion is the fact that certain objects will absorb certain wavelengths of light as light strikes them, both reflecting a certain proportion of the light away from the object and changing the colour of the reflected light to that of the object.

Thankfully, with shadows being the absence of light and thus of colour, we do not have to deal with this particular interaction. Sadly, it seem that monochromatic light have other complex properties that we will have to emulate, namely the following: all shadows have three distinct parts, the umbra, penumbra and antumbra. These names are all astronomical terms, originally used to describe features like eclipses, and phases of the moon. Shadows that are cast cast by a point source, i.e. that of an infinitesimal point, only cast the umbra - this is the hardest part of a shadow, and the part we are most familiar with. Every other non-point light source cast all three, and as real life cannot have point sources, we're going to need to deal with the emulation of both a penumbra and antumbra.

The emulation of the second facet of realistic shadows, the penumbra, will be the main focus of this report - the blurry area of gradient between the umbra and a fully-lit space. The area of this region is dependent on a number of factors, and may well constitute the entirety of a shadow. As such, it is difficult to calculate quickly and accurately.

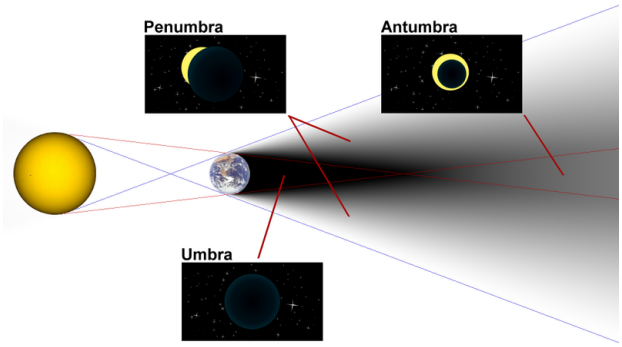


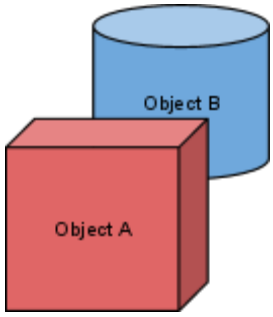
Diagram 1.1, showing the shadows in their astronomical contexts

Finally, the rarest feature of a shadow, the antumbra, is the point at which when stood between an occluding object and the light shadowing it, the occluding object appears to be entirely contained withing the disc or area of the light source, for example a an annular eclipse when the Moon is far enough away from the Earth that it does not perfectly cover the sun, creating a halo of light. This particular shadowing facet is almost entirely related to the astronomical background all shadowing terminology comes from, and as such is not a shadow that we are likely to ever encounter in this context. Thankfully, this means it is an aspect of soft-shadowing that we can ignore.

Real-time shadowing

Naturally, the first steps made into real-time shadowing were into what are known as hard shadows, or shadows that emulate the umbra only, or as if the entirety of any shadow was umbra. With techniques like Shadow Mapping, which creates a so-called shadow map by rendering the scene not just from the camera's perspective, but from the light's perspective also in order to be able to tell what exactly the light cannot "see" by using a feature in a graphics pipeline called the depth buffer.

The depth buffer is used to render objects in the correct order, so that closer objects are drawn in front of objects that may be behind them. It does this by saving the value of the closest object to the screen in the depth buffer. If a pixel being displayed on the screen is of an object that is currently closer to any other pixels at that value, that particular pixel will be superseded by the value currently there. If there are no nearest objects, the values are set to a particular far value.



For example, if OpenGL is attempting to render pixels from Object B, but this object happens to be behind Object A, then Object A's pixels would still be the closest value to the camera, and as such would be drawn first. Similarly, if the same buffer values are calculated from the perspective of the light, the buffer would calculate the nearest objects to the light. Everything behind these points that aren't the farthest value, and such are objects that the light cannot see, will be rendered as within shadow, and thus the depth buffer can be interchangeably used to mean shadow buffer, at least in this instance.

Shadow volumes were an improvement upon this technique, becoming a more visually pleasing solution by projecting actual 3D geometry outwards from the light, creating a volume behind each occluding shape edge. Whatever this volume hits, would become shadowed.

The penumbrae of a shadow are altogether more difficult to calculate, as the area of a shadow that is penumbra can range from very small, if the light is far away or the occluding object is small, to the shadow consisting entirely of penumbra. This would happen if the light was far wider, and close to, the occluding object.

Moving to soft-shadows

Image-Based Filtering

First steps were made by adapting current algorithms, like Percentage Closer Filtering (PCF)[3] , which takes the shadow map from an earlier algorithm, and filters it using a particular algorithm, called PCF. PCF averages the local values of an area of the shadow map, causing a blurring effect at the edges of the umbra, emulating a penumbra.

Whilst the above method creates more realistic-looking shadows with blurred edges, it does not emulate how a shadow becomes less distinct the closer the occluding object is to the light, or if the light is bigger than the occluding object itself. Conversely, it also doesn't emulate how if the light is small, or the occluding object is close to the receiving one, the shadow is mostly umbra. Percentage Closer Soft Shadows[4] fixes this.

PCSS proposes a method where the shadow map is created by searching through a list of the points on the receiving surface, and checking the near-plane for points that are closer to the light than this i.e. if an object is in occluding this point. An average is taken of the points found within a certain range, and the assumption is made that this is a single occluder, parallel to the light and receiving surface is made. The width of the penumbra at that point is then evaluated using the following formula:

$$Width_{Penumbra} = \frac{(Distance_{Receiver} - Distance_{Occluder\ Avg})}{Distance_{Occluder\ Avg}} \cdot Width_{Light}$$

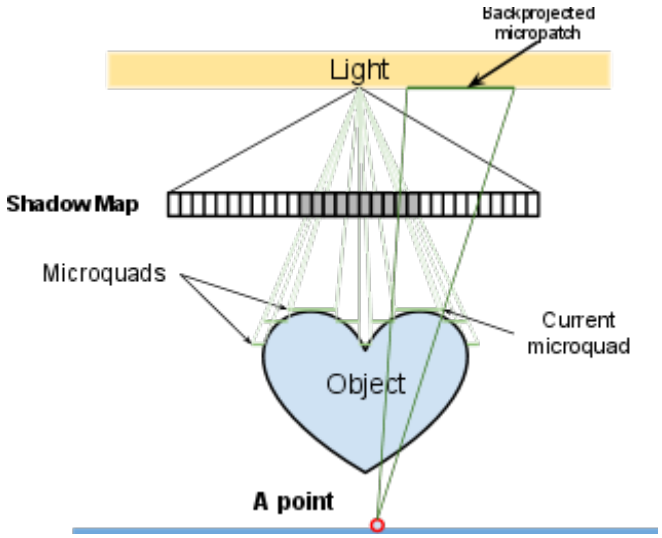
After this step, the original PCF averaging is used on a subsection of the shadow map, calculated by the following formula:

$$PCF_{window} = \frac{Width_{Penumbra}}{Distance_{Occluder\ Avg}} \cdot Distance_{Nearest\ Occluder}$$

This method, whilst simple, is also not very accurate. It's possible for it to create erroneous shadows in several different ways. The first stage of the algorithm searches for objects that block by simply checking the point on shadow map being closer the receiving object. to the due to the fact that the occluder is assumed to be perpendicular. This is an assumption that is not a good approximation in certain complex scenes and as such, it is both possible to erroneously calculate a receiving section which has shadow that should not, and indeed the reverse.

Backprojection

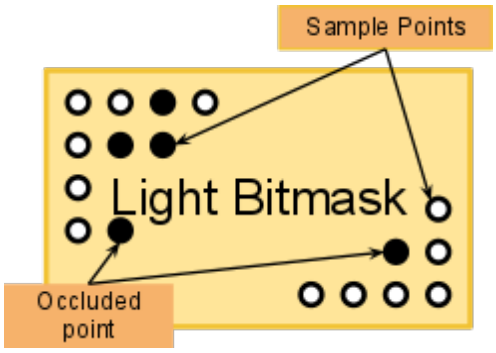
As PCSS approximates a fair deal of information about the occluding object and as such fails in certain situations, another class of soft-shadowing algorithms attempts to deal with the problem by approximating the occluder in a more precise way. Known as backprojection[5], this technique searches the shadow map that is created as before, by projecting it from the light source, and searches through each value on the map for a potential occluder. If an occluder is hit, a rectangular surface, or micropatch, is generated at that point, parallel to the light, representing an approximation of that particular point on the occluder.



As the algorithm traverses the list of points a possible shadow could be at, seen in the diagam to the left as *a point*, it searches for *micropatches* that are closer to the light than the point, meaning there is an occluding surface somewhere above this point in the shadow map. This micropatch is backprojected onto the light surface via the current location in the shadow map to determine how large the area of occlusion is. Each of these occlusion areas are summed to create an estimation of the total occluded area, and this total occlusion area is then used to calculate how shaded a

particular point will be.

An example algorithm of backprojection would be Occlusion Bitmasks, which takes the basic idea as described above, but instead of simply calculating the total backprojected area that the occluders have on the light, each backprojection travels through a bitmask - an array of points that are either one or zero, depending on whether they have an occluding object beneath them or not. Each backprojection is additively added to this bitmask, as to build up a shadow “image” of the object below, and by extension this means the total area of the occluded section is easily calculable given a tally of all set bits in the mask.



This particular solution again has its downsides, however. The micropatches by defintion are all parallel to the light, meaning that on an object with lots of sudden changes in height, the patches could have large gaps between them, allowing light to leak through. Improvements to this would be to create a quad mesh using the micropatch points, rather than flat, perpendicular quads, but this only adds to the rendering time.

Other Solutions

There are several other current solutions that do many different things, such as using multiply-scaled shadow maps that use several different level of shadow quality or penumbra width , which are applied to the outsides of a shadow umbra depending on a a particular distance algorithm. This is a speed boost on the backprojection solution, as parts of the shadowing is precalculated, but uses far too much memory to do so for any reasonable real-time application.

Another group of algorithms use the idea of generating geometry-based constructions, that in one example, projects primitive shapes backwards from the occluding objects and calculating the areas to the edges of these shadow volumes that would contain penumbrae. This particular algorithm, known as Sample Based Visibility, is a relatively complex, accurate and very computationally expensive algorithm, and are best used for shadowing large scenes where image-based shadowing is less able to plausibly fake an image.

None of these solutions take into account the indirect lighting aspect of reality, and there is another slew of algorithms that deal with this particular aspect, known as environmental lighting, using techniques even newer than the ones discussed above like ambient occlusion to plausibly fake the effect.

Specification

Methodology

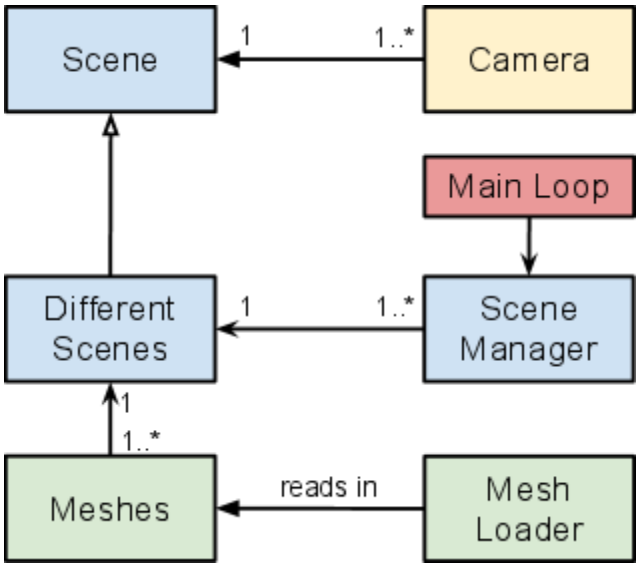
In order to create a suitable workbench which I can modify fairly easily, and to render at real-time framerates, I will be using a combination of C++ and OpenGL using SDL. Visual C++ Express 2010 is my IDE of choice, and I will be developing using an Agile methodology, with quick turnover times between iterations. This means I should be able to implement several features in a short period of time, and get to the meat of the project - the testing, as quickly as possible.

My System Design

My workbench will have to be modular in order for me to be able to quickly test and add features, and be able to debug them in a relatively short period of time. So that this is possible, I've identified particular subsystems that I'm going to require.

I will use OpenGL's default, event-driven framework for the the main loop of the program, and implement certain manager modules and classes to abstract away the complexity, and make the loop more readable and modifiable. I will create a scene manager within my initialisation to take care of each scene I will need to implement. A timer module will also need to be implemented, separate but accessible from this manager, in order to keep track of how long each frame takes to render in milliseconds. This will not only make it possible for me to play/pause the scene, but perhaps even be able to rewind the current test to an earlier state for debugging.

The main program will also deal with all, if any, GUI code, as I am unlikely to require many keyboard accessible features with the automation, besides the moving of the camera around the scene and pausing/playing/restarting of scenes.



The scene manager itself will be in charge of different scenes, each of which will contain within them the required information about lights, where meshes are etc. These meshes are dealt with and loaded by by a mesh loader class, which reads in the location of the mesh in my format of choice, and then returns the mesh to the scene.

In order to be able to swap shadowing algorithms on the fly, the scene manager will also be in charge of shadow rendering, as it is simpler to change the value within that than within each individual scene.

Each scene will inherit common information from a superclass, for example each scene would have a camera object within it, that deals with input from the user in changing its state, and the scenes are all in charge of rendering the items within themselves. The superclass also deals with tidying up the scene, by removing the pointers

to objects and turning off/removing lights.

The states of the scene will be changed via the scene manager, which will dynamically swap between different scenes (for the automated test system).

Tests themselves will be detailed within each scene, and run from the manager. They should inherit common information, like a FPS counter for example, from the superclass. It would be difficult to standardise the tests, if they are all to test a different quality, however, so that part of the testing will be dealt with within each scene themselves, and overseen by the scene manager.

I hope to have the output from the tests dealt with and partly analysed, too. This is the section I have the least clear idea about, but will probably be dealt with by first outputting raw data into a file, then having that file processed before a tidy report is produced.

Which algorithms?

The algorithms I have decided to use are Percentage Closer Soft Shadows and Occlusion Bitmasks . I will be using these on the basis that PCSS is currently a very popular algorithm and is fairly simple to implement, given an already working shadow map. Occlusion Bitmasks are an improvement in the quality of the shadows produced, but also come with the downside of being slower to calculate. The reason I have chosen these two instread of the more complex geometry-based algorithms, which are more accurate, is that I fear I will have no time to both implement and then use the algorithms I have created to test meaningfully. I have chosen two apparently simple to implement algorithms to hopefully give me as much time as possible in the final stage of the project. This also means the algorithms should produce fairly similar results, and it will be interesting to see what the computation speed will add when I test.

Design

UI justification

The GUI will be as simplistic as possible, and display information like the current CPU speed, and a graph of speed over time, as well as a frames per second/milliseconds per frame value. It would be useful to be able to split away certain parts of the automated test schedule, so that it is possible to break halfway through an operation if something were to go wrong, and so I think it would be important to be able to initiate the tests manually also, most likely from keyboard input with a representation of what is currently happening on the GUI.

Experimental Design and Metrics

First and foremost, any and all tests that I do will have to be repeatable any number of times, and have the same, or similar results given the same input circumstances. This would reduce the testable variables to those of which shadowing algorithm I am using, and which scene I am using to test the algorithms on, meaning, given the correct input and algorithmic implementations, the results will hold up to scientific scrutiny.

The tests themselves will be thus: firstly, to deduce which algorithm performs the best computationally, I will calculate the time it takes to render the shadow in milliseconds, coupled with the frames per second I am rendering at, and the CPU usage over time. I would also make a note of the complexity of each algorithm as a factor in this, looking at the number of operations per render.
Secondly, to deduce the quality of the rendered shadow, I will compare the number of artifacts in the rendered image of both shadowing algorithms over each scene, several times. I could also compare these results to the ideal shadowing solution, using a ray-tracing algorithm, but I am unsure that I will be able to complete that task in the time I have allotted.

As speed of an algorithm is directly related to complexity of the inputted data (for any non constant-time algorithm), the computational prowess of an algorithm would be tested by comparing how well each did with a basic scene with its results on a complex scene. The simplest scene would then of course be a single light shining on a single object, both static. The tests from then on would have moving objects, which would cause shadows to be cast on different parts of the scene, and over other shadows. Other objects casting multiple shadows would also test an algorithm's computational ability, and so they will also be included.

On the quality side of measurement, the testing criteria are more nebulous, but I have identified certain criteria that a shadow would need in order to be of realistic quality:

- Do they have a smooth edge?
- Do they interact with one another properly - ie do they look like one homogeneous shadow?
- Do moving shadows move smoothly?
- Do multiple lights create multiple shadows? Do these have all of the above?

These specific tests are detailed in the implementation section.

The tests would be run automatically, either via a hard-coded set of instructions from within my program, or read in from a file. This will also depend on the time I have remaining.

Implementation

Workbench Implementation

The current state of the workbench is that I can create and swap between scenes using the scene manager. I have not yet implemented any shadowing algorithms, but this will soon be remedied. A timing system is also on the cusp of finishing, and with this I can build half of my testing system.

Test Implementation

Every test that I implement will record algorithm speed and frame per millisecond, and so what I shall be testing specifically for are the quality features. I should, as a result of using more complex test scenes, also then be able to see the algorithm's capability range.

The tests I will be implementing will be:

1. A singular light source on a simple object e.g. a cube
   This is the simplest test that's possible, and will check only for the quality of the shadow edge.
2. A single light source on a complex object
   To ensure that the shadow works on all objects, this is the next logical step, and the penumbrae on this object won't be as distinct.
3. A single light with two objects
   To test shadow self-interaction
4. single light source moving around one object
   This will test the smoothness of a moving shadow
5. A single moving light with numerous objects
   To test the above three qualities at once, and see how well both algorithms do speed-wise
6. Several lights around a single object
   This will ensure that a single object can project multiple, blended shadows.
7. A scene with multiple moving lights, multiple moving objects
   Simply to see if I can hit a limit on the smoothness of either algorithm

Each of these tests will be contained within a set bounding box, in order to catch all shadows.

Achievements to date

This section discusses what changes have been made and what's been achieved from the original plan to where I'm at.

What I intended

By the time this report was due, I intended to have the following completed:

- Design and create an up-and-running workbench system that I can add modules to
- Begun the implementation of my first shadowing algorithm
- Done a research report on the shadowing algorithms I wished to use

What I achieved

The things I actually have achieved are:

- Design and create most of the subsystems I wished to have up-and-running by this point
    - Meaning I haven't begun implementation on any shadowing subsystem yet.
- Researched into the shadowing algorithms, but not created the report
    - After having done the research, I decided against creating a separate report, thinking I would just consider the background section of this report adequate

Does the plan need changing?

As things are progressing right now, I'm fairly certain I'll be able to achieve everything that I wished to achieve on the timescale that I gave myself. However, given that I have not yet begun implementing the shadowing algorithms, this may well change.

Conclusions

This report is an investigation into current soft-shadowing algorithms, and covers the design and the beginning of an implementation of software which will compare two soft-shadowing algorithms, namely Percentage Closer Soft Shadows and Occlusion Bitmasks against each other in both computational speed and quality of image using various automated tests. These tests will grade the shadows in one of the four quality factors which I identified.

References:

[1] Vinayagamoorthy, V. et al., An Investigation of Presence Response across Variations in Visual Realism. *UCL*.
[2] Mamassian, P., Knill, D.C. & Kersten, D., 1998. The perception of cast shadows. *Trends in cognitive sciences*, 2(8), pp.288-95. Available at: http://www.ncbi.nlm.nih.gov/pubmed/21227211.
[3] Reeves, W.T., Salesin, D. & Cook, R.L., 1987. Computer Graphics, Volume 21, Number 4, July 1987. *Computer*, 21(4), pp.283-291.
[4] Fernando, R., 2003. Percentage-Closer Soft Shadows Percentage-Closer Soft Shadows (Supplementary Material). *Computer*, (August 1978).
[5] Schwarz, M. & Wimmer, M., 2009. Casting Shadows in Real Time. *Computer*. pp.77-82.