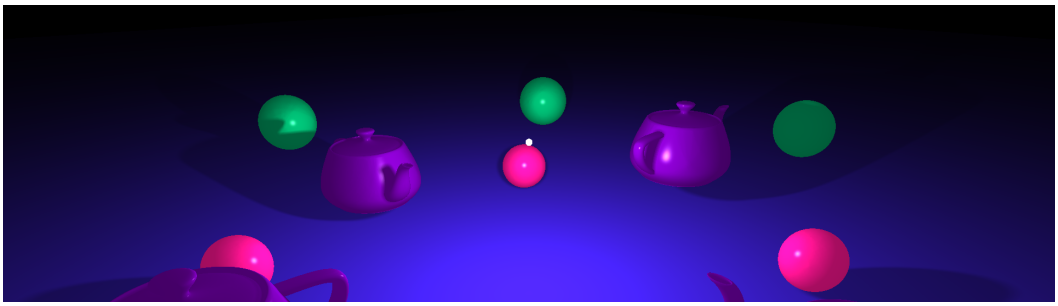
A Brief History of Rasterized Shadow Algorithms

VINCENT MARIAS, Colorado School of Mines, USA



I present an overview of three of the most popular shadowing algorithms for real-time rasterized applications, in chronological order as they were introduced/peaked in popularity/viability. I also give implementation details and techniques to improve the results from each algorithm, along with considerations for the benefits and drawbacks of each. The final result is a simple implementation of shadow mapping using percentage-closer filtering, suitable for simple scenes. The introduction additionally contains a brief overview of the broad categories of computer-generated shadows and the rendering equation.

1 INTRODUCTION

When rendering a scene, shadows are one of the most important effects that can be added if the goal for the image is realism. They are an integral part of approximating real-world lighting, as much as radiance itself. However, they have long been a sticking point for computer graphics programmers, as the way light is "faked" under rasterization does not facilitate the creation of shadows. To understand why, we'll briefly define shadows and their various computer-generated forms.

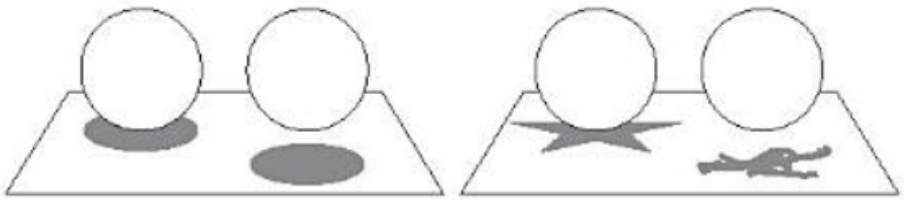


Fig. 1. Shadows have an important influence on the interpretation of spatial relationships in a scene (left). Nevertheless, even course approximations can achieve the same effect (right). [4]

What is a shadow? The authors of "Real-Time Shadows" propose a general definition:

"An area that is not or is only partially irradiated or illuminated because of the interception of radiation by an opaque object between the area and the source of radiation." [4]

Author's address: Vincent Marias, vmarias@mines.edu, Colorado School of Mines, Golden, Colorado, USA.

The most important part of this definition, and the aspect that makes rendering realistic shadows so difficult is the possibility that shadowed areas may be "only **partially** irradiated". This leads to the two categories of shadows that we can render.

1.1 Soft Shadows

In order to fulfill the definition above, we need to render what are called **soft shadows** in our scene. The basic idea is quite intuitive, given our inate understanding of how light works in the real world. Some shadows are sharp and well-defined, almost a colorless reflection of the object casting them, and some are soft, with blurred edges with difficult-to-define borders. The general difference can be seen in figure 2.

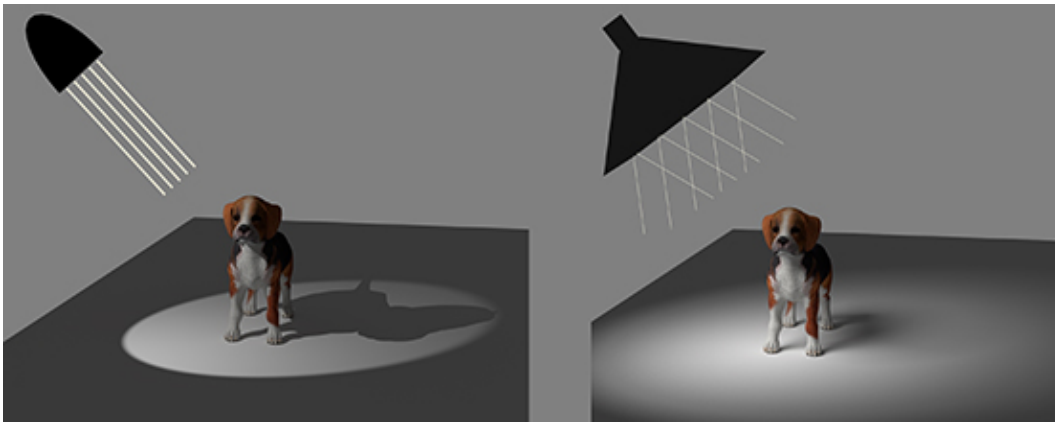


Fig. 2. Hard shadows produced by a spotlight (left) vs. soft shadows produced by an area light (right).

The darkest part in the middle of a soft shadow is called the **umbra**, while the softer part around the edges of the umbra is called the **penumbra**. We'll see why these different parts occur shortly. (There is also the **antumbra**, but we'll ignore that for our purposes here.)

You'll notice another important difference in the image: the type of light in the scene. Hard shadows are produced by point lights, while soft shadows are produced by area lights.

Consider the scene as a collection of points, either part of an object, **view samples**, or part of a light source, **light samples**. We'll call the objects that block light to cast shadows **casters**, **occluders**, or **blockers**, and the objects that have shadows cast onto their surfaces **receivers**. With this setup, and considering the rendering equation[1], we can start to see how soft shadows can be realised.

$$L_o\left(\mathbf{p},\omega\right) = L_e(\mathbf{p},\omega) + \int_{\Omega_+} f_r\left(\mathbf{p},\omega,\hat{\omega}\right) L_i\left(\mathbf{p},\hat{\omega}\right) \cos\left(\mathbf{n}_p,\hat{\omega}\right) d\hat{\omega} \tag{1}$$

Without going into detail, all we really need to understand about this equation is that for each view sample $\mathbf{p}$, we integrate all incoming light L_i from every incoming direction $\hat{\omega}$ in the hemisphere above that point Ω_+. One way to think about this is that view samples are lit based on whether or not they "see" one or more light samples. If it sees all the light samples, that point is lit completely by direct light. If it sees none, it is lit only indirectly, and is part of the shadow's umbra. If it sees only some of the light samples, it is paritally lit, and is part of the shadow's penumbra. This integral can be approximated in a rendering application as a summation of some number of sampled view directions.

1.2 Hard Shadows

It quickly becomes apparent that approximating soft shadows is extremely difficult due to the expensive calculations that must be performed. Since every light in the real world is an area light - point lights don't really exist - every shadow in the real world is a soft shadow. So in order to render shadows that in any way approach realism, we need soft shadows. But if you remember back to figure 1, we don't actually need realistic shadows to get most of the benefits that shadows in general bring. Even bad shadows are better than no shadows.

This brings us to **hard shadows**, which are produced by point lights and consist only of the shadow's umbra. These are much easier to compute, and allow us to take advantage of rasterization tricks to generate them in real time.

2 RELATED WORK

Many, many researchers have written about shadows and proposed many, many algorithms to render them. In fact, the sheer number of different algorithms and techniques is overwhelming to the point where entire books have been written, such as "Real-Time Shadows"[4], whose sole purpose is to gather and consolidate as many of the useful ones as possible. The most popular ones are described below in chronological order.

2.1 Planar Projection Shadows

In the beginning, there were projection matrices. The perspective projection is one of the fundamental building blocks of rasterization. It projects 3D points from camera space to view space - actual pixel locations on the screen - in a way similar to how the human eye or a pinhole camera works. Another way to think of this is that we project a 3D object onto a 2D image plane. In 1988, Jim Blinn realized that if we consider the scenario of projecting a shadow onto a flat plane, this is very similar to the projection of objects onto the flat image plane[2]. He proposed a technique using a special projection matrix M, given a plane defined by the equation $\mathbf{n} \cdot \mathbf{x} + d = 0$ and a point light at position l:

$$M = \begin{bmatrix} \mathbf{n} \cdot \mathbf{l} + d - \mathbf{n}_x\mathbf{l}_x & -\mathbf{n}_y\mathbf{l}_x & -\mathbf{n}_z\mathbf{l}_x & -d\mathbf{l}_x \\ -\mathbf{n}_x\mathbf{l}_y & \mathbf{n} \cdot \mathbf{l} + d - \mathbf{n}_y\mathbf{l}_y & -\mathbf{n}_z\mathbf{l}_y & -d\mathbf{l}_y \\ -\mathbf{n}_x\mathbf{l}_z & -\mathbf{n}_y\mathbf{l}_z & \mathbf{n} \cdot \mathbf{l} + d - \mathbf{n}_z\mathbf{l}_z & -d\mathbf{l}_z \\ -\mathbf{n}_x & -\mathbf{n}_y & -\mathbf{n}_z & \mathbf{n} \cdot \mathbf{l} \end{bmatrix} \tag{2}$$

This generally allows us to project a flat image of any blocker object onto a flat plane on a receiver object. If we then draw this image in black, we have a perfectly sharp hard shadow!

There are some caveats, of course. Blockers and receivers must be separated - no self-shadowing is possible. Receivers must be perfect planes (or parts of a plane). Blockers must be between the light source and the plane, or strange artifacts called anti-shadows will be produced. All blocker geometry is drawn twice, as the shadows are literally flat geometry. But it's simple and fast, which made this an extremely popular technique in real-time applications like games for many years[1].

2.2 Shadow Textures

An extension of projection shadows[1], shadow textures start by generating the projections in two passes. Let's think about the process differently, though. If we imagine a camera located at the light source and facing in the direction that light is emitted, we could render the scene from this perspective. In this first pass, render the blockers into a texture. Now, in the second pass, render the receivers while sampling from the texture to determine which fragments are lit and which are in shadow.

This simple method allows for more flexibility: Now we can project shadows onto curved surfaces. However, blockers and receivers are still separated, and now we have to deal with all the problems that come along with textures. Shadow quality becomes an exercise in choosing the best resolution and filtering method for your textures. We'll see what this looks like later, but it's safe to say it was massively over**shadowed** by the following technique.

2.3 Shadow Mapping

By far the most popular and long-lived technique for real-time shadows[1], shadow mapping is a further extension of the shadow textures method. With that method, the texture produced is binary - any fragment is either shadowed or not. The reason this prevents self-shadowing is because if we were to render the blockers again in the second pass, we'd end up deciding that the whole object is in shadow (shadowed by itself). Textures can hold lots more information, so what information are we missing? This technique proposes that we include the depth of each fragment in our texture[5].

Now the process becomes slightly different. In the first pass, we render our texture, but instead of just the blockers, we render every fragment in the scene, recording it's depth. In the second pass, we sample from this texture again. However, we must now calculate the fragment's distance to the light source, and compare this to the stored depth in the texture. If the fragment is closer than that in the texture, it is lit. Otherwise, it is in shadow. This involved transforming each fragment to light space again in the second pass.

Now we have a general, per-fragment solution to hard shadows. Self-shadowing is a given, and point lights can be modeled with ease. However, there are many artifacts that will be discussed later, which arise from the fact that we are now effectively performing a signal reconstruction from a relatively low-resolution signal. Many shadow techniques are simply approaches to mitigating these artifacts[4].

3 PROBLEM STATEMENT

My code implements all three of the above techniques, along with various solutions to each of their shortcomings. I encountered and (at least somewhat) solved all of the artifacting and issues described above. We'll see this in more detail in a moment.

One of the biggest challenges when it comes to both shadow textures and shadow mapping is that of omnidirectional point lights. Of course, a point light is just that: a point. It emits light uniformly in all directions. However, you'll notice that the above techniques only seem to account for what is effecitvely a spot light - the light shining in a single direction (that being the direction our "camera" faces in the first pass). There are a couple of solutions to this, which will also be discussed below.

4 PROBLEM SOLUTION

I wrote an OpenGL program capable of rendering spheres, quads, and Utah Teapots. I then constructed a scene with a number of teapots and spheres hovering in a circle above a ground quad, along with a single white point light in the middle of the circle, so that it might cast shadow outwards. The program allows for testing of all three of the following algorithms, as well as real-time adjustment of their parameters.

4.1 Planar Projection Shadows

This technique was relatively easy to implement, especially in it's naive form. I send the normal of the plane I want to project onto from the CPU to the GPU via a uniform. In the vertex shader, I apply the above projection matrix (2) to the vertex positions of the blocker objects between the model and projection matrices. Then in the fragment shader, I draw those fragments in black with no

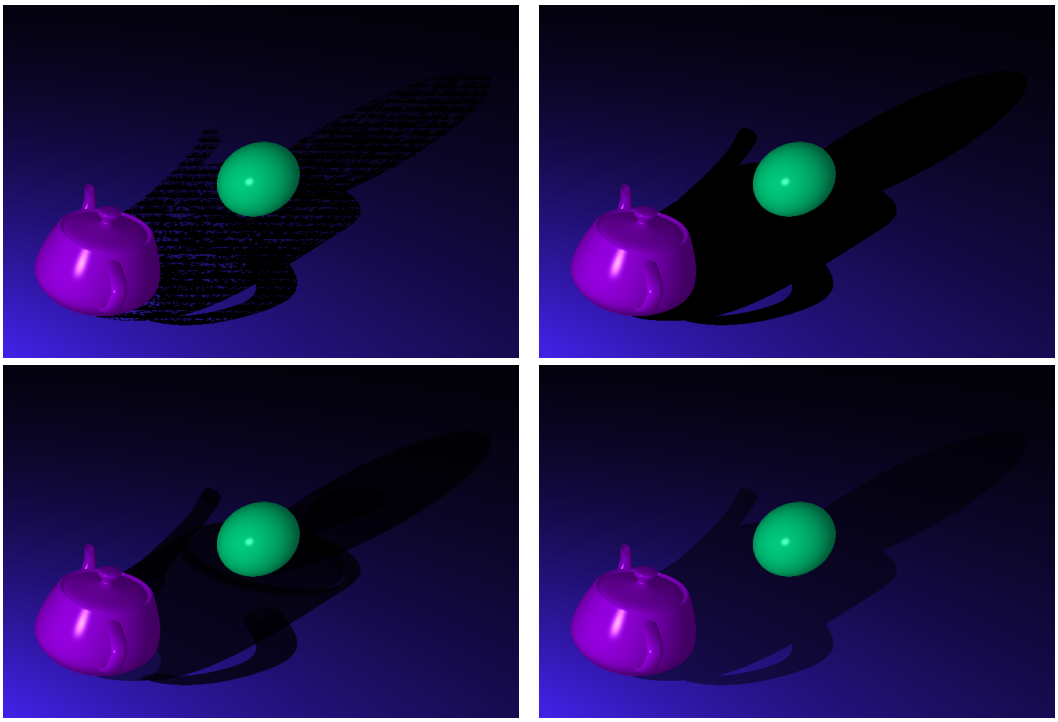


Fig. 3. Naive planar projection shadows displaying z-fighting (top left), with no blending (top right), with naive blending displaying layering (bottom left), and with proper blending via the stencil buffer (bottom right).

transparency. The initial result can be seen in the first image of figure 3. There are some z-fighting issues thanks to drawing directly on top of the ground plane, which are fixed in the second image by turning off the depth test when rendering the shadows. These are still completely black, but if we simply turn on blending, there are layering issues that can be seen in image 3. This is because the shadow itself is literally duplicated geometry, and multiple vertices get projected to the same points. This is more difficult to fix. My solution was to first render the ground plane into the stencil buffer, setting the covered values to 1. Then when drawing the shadows, we can increment any overlaps in the stencil buffer and set up the stencil test so that further overlaps will fail the test - this means shadow fragments are only ever drawn once. Before this addition, the technique could be done in one pass, but with it place, I had to use two passes.

Notice that all the curved objects cast shadow, but only the ground plane can ever receive shadow.

```
// when drawing the receivers (ground plane)
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 0, 0xffff);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);

// when drawing the shadows
glDisable(GL_DEPTH_TEST);
glStencilFunc(GL_EQUAL, 1, 0xffff);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
```

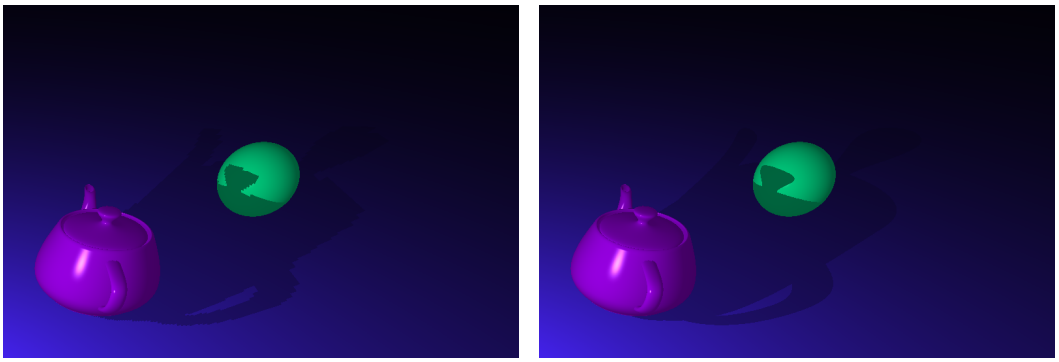


Fig. 4. Low-resolution shadow textures (left) vs. high-resolution (right).

4.2 Shadow Textures

For this technique, I needed to render the scene from the light's perspective. There was the aforementioned issue with rendering in all directions, though. From my research, there are generally two ways to do this. What we need is not a single 2D texture, but a cubemap. So you could render the scene in 6 extra passes, one facing in each direction from the light's perspective and with a square aspect ratio. Or, you construct the cubemap on the fly in the geometry shader to do everything in one pass. It sounds like the second option would be much more efficient, but it turns out that this actually makes the geometry shader very heavy [1, 3], so the performance tradeoff is not so clear. I chose the more simple option of creating the cubemap on the CPU and rendering in seven passes total.

As you can see in the first image, with a low resolution texture (even one-to-one with the final image), the result is not very appealing. Increasing the texture resolution improves things quite a bit, but has a heavy performance cost.

You'll notice that now we can cast shadows onto curved surfaces as well as planes, though to do this we still have to choose which objects are blockers and which are receivers (the green spheres are now receivers and do not cast shadows).

4.3 Shadow Mapping

With the cubemap and texture write/read framework in place from the previous method, implementing shadow maps was trivial. I simply modified the fragment shader that produces the cubemap (and the texture setup on the CPU) to write the depth component instead of just black. Then in the fragment shader that draws the second pass, I calculate the linear distance[3] from the fragment to the light source, and use the vector from the light source to the fragment to sample from the cubemap. I then compare the depth value stored there to the calculated distance to determine if the fragment is in shadow or not.

The result can be seen in the first image of figure 5. The artifacts seen there are called **shadow acne**, and are caused, in a similar manner to z-fighting, by sampling from a discrete texture with a discrete resolution. To fix this, I simply add a small bias to the depth I sample from the cubemap, so the shadows are slightly offset from where they should be but it resolves the acne quite nicely (image 2).

Notice now that all objects are casters and recievers, and objects like the teapot are able to self-shadow. The texture resolution still causes aliasing issues around the edges of the shadows, but we'll resolve that in a moment.

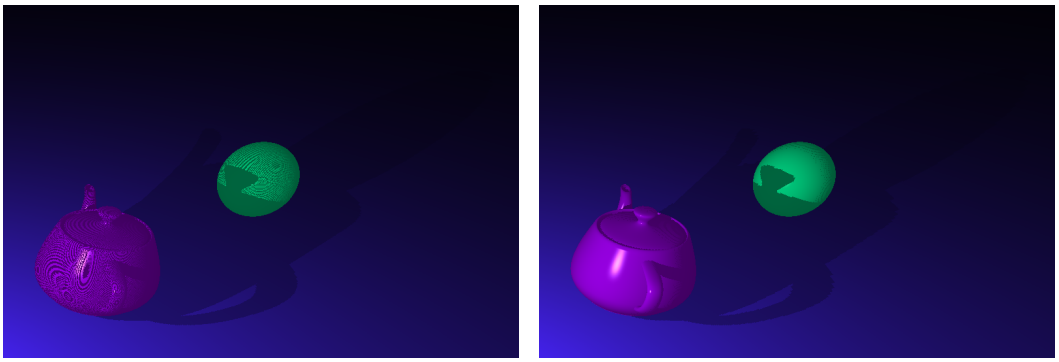


Fig. 5. Naive shadow maps exhibiting acne (left) vs. using a slight depth bias to resolve the acne (right).

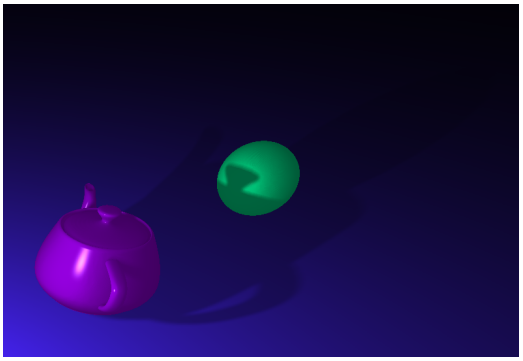


Fig. 6. Using percentage-closer filtering to reduce aliasing and soften shadow maps.

4.4 Percentage-Closer Filtering

The easiest way to resolve the aliasing issues exhibited above is to increase the texture resolution, but this gets expensive very quickly. Instead, let's use some clever filtering to smooth out the result. We can apply a basic filtering kernel to the texture which in it's most simple form just averages the depths of surrounding fragments. By adjusting both the number of samples and the texture resolution, we can obtain quite a nice looking image with very little performance overhead.

There are more advanced kernels you could apply, such as Poisson-Disk Sampling, but for this simple scene their added complexity would not be worth the negligble benefits.

5 RESULTS

All of the results can be seen above, but figure 7 contains a wider shot of the scene with the highest-quality shadows I was able to produce by adjusting all the different parameters.

6 CONCLUSION

Through the above results, we can see a progression of shadowing algorithms from very basic and limited to (almost) completely general. Each method brings it's own set of challenges and drawbacks, but none that can't be at least somewhat overcome with rasterization trickery. The most general algorithm - shadow mapping - comes with with largest set of artifacts and issues. Many of these can be alleviated with simple approaches, but to shadow very complex scenes in a

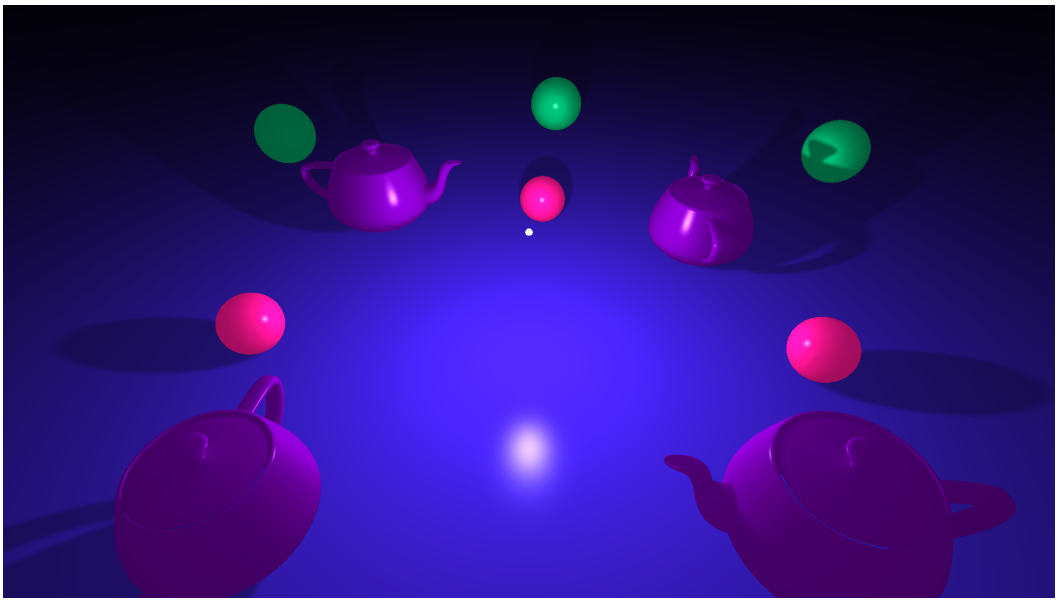


Fig. 7. Nice view of the whole scene using shadow maps and PCF.

convincing and visually appealing way requires much more work. There is endless research on this topic, and an endless number of approaches[4]. Luckily, for simple scenes like the one I constructed for this demonstration, a simple filtering kernel and some careful adjustments are all that's needed for a relatively clean image.

There are still many more techniques not explored here, such as shadow volumes, which also take into account per-object information rather than simply per-fragment. There are also methods to simulate more advanced shadow scenarios, such as shadows cast by semi-transparent objects, shadows cast through media such as fog, or by multi-colored lights. These are outside the scope of this project, however.

I had much grander plans for this project, but writing a polished program to show off the above techniques proved to be more work than I expected. Regardless, I think the final results look quite good.

REFERENCES

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2019. *Real-time rendering*. Crc Press.
[2] Jim Blinn. 1988. Me and my (fake) shadow. *IEEE Computer Graphics and Applications* 8 (1988).
[3] Joey De Vries. 2015. Learn opengl. *Licensed under CC BY* 4 (2015).
[4] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. 2011. *Real-time shadows*. CRC Press.
[5] L. Williams. 1978. Casting curved shadows on curved surfaces. *Computer Graphics* 12 (1978).