Real-Time Soft Shadows

CS 248A Final Project

Jared Watrous

1 Introduction

In this project, I implement five methods of rendering and filtering real-time shadows from a shadow map, two of which support soft shadows for light sources of varying radii. I focus on directional (sun-like) light sources, as they only require a single shadow map per light source. I also explore an original ray-marching method for soft shadows, which had originally inspired this project.

1.1 Starter Code

I build my implementation on top of a previous project of mine from CS 348K[1]. The starter code written in C++17 and uses OpenGL 4.3+. It supports object/texture importing, scene graphs, and PBR materials with directional and point light sources. For simplicity, only the Forward render pipeline is supported, and light culling is not enabled for directional light sources. Notably, the lighting in the starter code did not support shadows of any kind; shadow mapping for directional lights was implemented from scratch for this project.

All code for this project is available at `https://github.com/thetruejard/cs248a_project`.

2 Methods

2.1 Basic Implementation

The basic implementation uses the light source's world-to-local matrix (e.g. inverse of its model matrix) as the "view" matrix when rendering its shadow map. The bounds of the shadow map are determined using the scale of the light source in the scene graph - a scale of 1 indicates the bounds are 1 unit from the light position in each direction. The Z-axis scale is taken as the near/far clipping planes for the shadow map. Note that this means the "up" direction may not be aligned with the world, if the light has a "roll" component to its rotation. Shadow maps are allocated and rendered in `graphics/pipeline/rp_forward_opengl.cpp`.

Up to 16 shadow-mapped light sources are supported, and any additional light sources will not have shadows (the first 16 are chosen in arbitrary order.) The shaders used to compute lighting and shadows are `shaders/opengl/forward.vert` and `shaders/opengl/forward.frag`, the former of which transforms vertices into the space of each shadow map, and the latter of which computes lighting for each fragment. **All shadow implementations are defined in this latter file.**

The most basic shadow implementation simply queries the shadow map, applies an additive bias based on light/normal angle, and compares it to the depth of the fragment. This gives aliased result as shown in Section 3.

2.2 Percentage-Closer Filtering

Percentage-Closer Filtering (PCF) extends the basic implementation by applying a simple uniform filter to accumulate nearby samples and reduce aliasing. I follow [2] and use a simple 3x3 box blur. The result still appears blocky/pixelated, but reduces perceived harshness of the shadows.

[1] `https://github.com/thetruejard/cs348k_project`

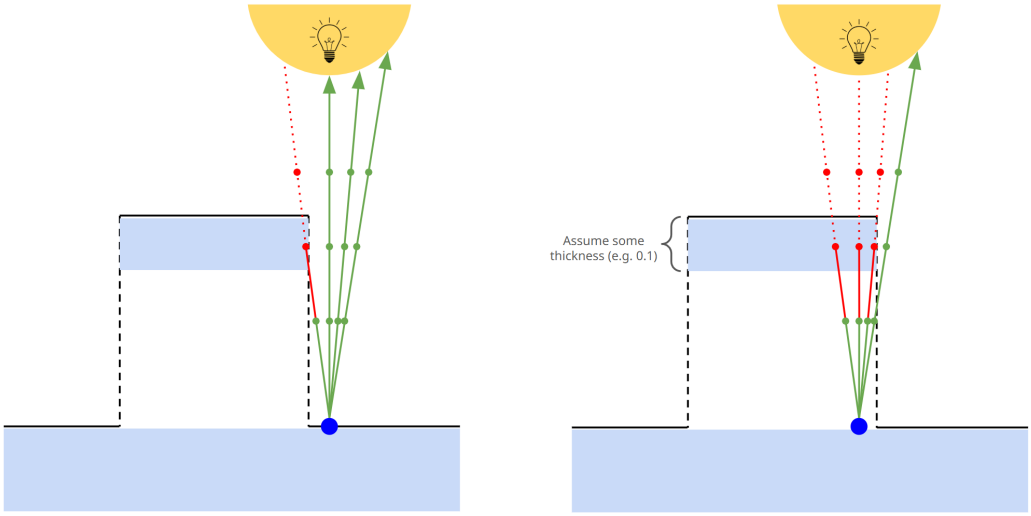

Figure 1: The Ray-Marching method. The left shows a general case, where rays are marched upwards towards some random point on the light source. Points along the way are considered obstructed if they are further from the light than the depth value recorded in the shadow map. The ratio of unobstructed to obstructed lights gives an estimate of the light's relative contribution to the fragment. The right shows a more complicated case, in which the fragment would already be considered occluded according to the depth map. We address this by defining some maximum "thickness" to the shadow map, and any points further from the recorded depth than this thickness is still considered "valid." This is a suboptimal assumption, but significantly improves qualitative results.

2.3 Jittered PCF

I implement an alternative to the box blur filter outlined in [4], which instead transforms randomly jittered points into a Poisson disk pattern for area-weighted sampling. This generally reduces aliasing further (though not completely) at the expense of some noise. **For this method and all other randomized methods,** are randomized per fragment by default. To make each fragment share the same seed, set `DETERMINISTIC_SEEDS` to `true` in `shaders/opengl/forward.frag`.

2.4 Percentage-Closer Soft Shadows

Percentage-Closer Soft Shadows (PCSS) [3] attempts to render perceptually plausible soft shadows. In essence, it scans nearby depth samples in the shadow map for potential "blocker" objects - depth values closer to the light that may obscure some part of the light source for the current pixel. It averages the depth of all the candidates it finds, and treats this is as the "blocker" depth. The penumbra size is then computed as a function of blocker distance and light radius, and this penumbra size is used as the filter size (following the same filtering method as Jittered PCF.) See the original paper [3] for more details on the precise algorithm.

Notably, PCSS assumes the light source, blocker, and shaded fragment are all parallel planes. I make no effort to relax this assumption, though I do loosely fine tune the light source radius, since the directional light size is defined in radians rather than world-space units.

2.5 Original Ray-Marching Method

In my project proposal, I suggested a novel approach to soft shadows inspired by Parallax Occlusion Mapping [1]. Upon implementation, this effectively resolves to a simple ray-marching algorithm that attempts to estimate relative occlusion of a light source using a Monte-Carlo approximation. Considering how straightforward this method is, I do not necessarily claim it to be "novel" – however, I was unable to pinpoint any

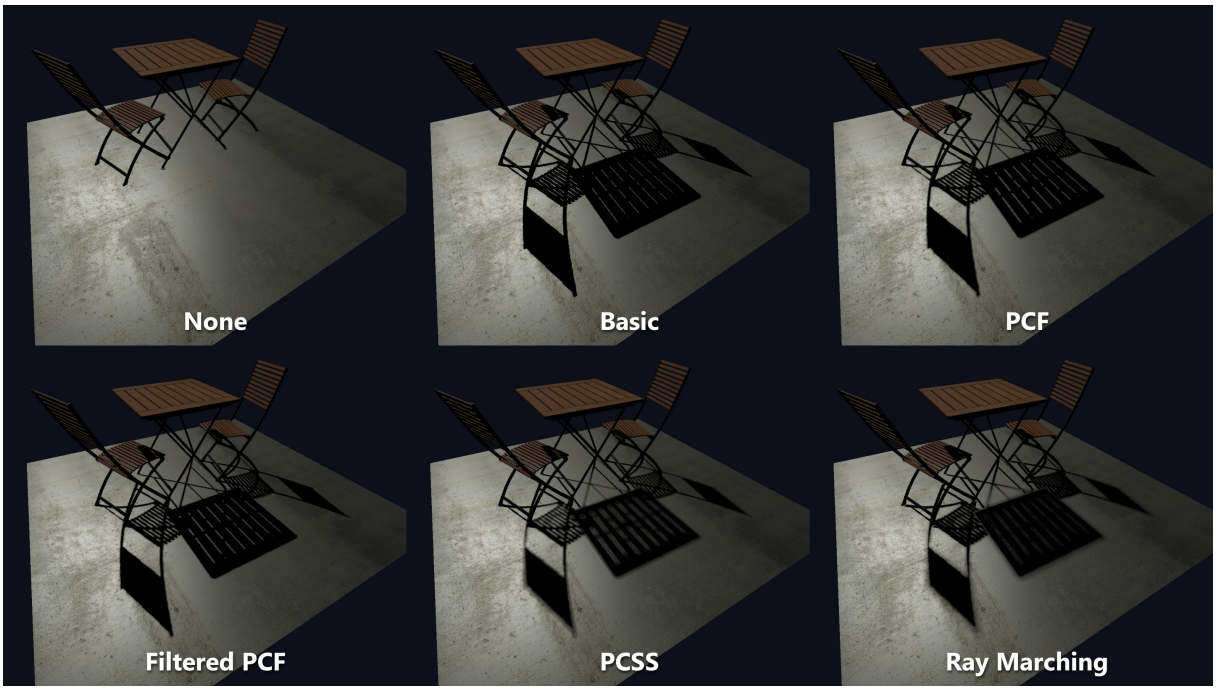

Figure 2: A side-by-side comparison of methods on a complex scene with sparse geometry.

references describing a similar algorithm for this application, so here I decide to refer to it as "original."

The general idea is outlined in Figure 1. We march rays away from the current point in a feasible direction towards the light source, searching for points in the depth map at which the ray is obstructed. If we find such a point, the ray does not contribute to the point's light; if the ray is not obstructed, then it does. Note that, if the initial point is already occluded (i.e. it would be in shadow using a "basic" shadow implementation), then some early points on the ray may be considered obstructed when it shouldn't. To mitigate this, we can assume some maximal "thickness" to the depth map, and allow rays to pass unobstructed if they do not intersect within this thickness threshold. This assumption may break down in cases where object surfaces are (nearly) parallel to the light's direction. Note that multiple rays are only used for shadow estimation, not lighting; the PBR lighting computation still only uses a single incident light direction.

This method is imperfect and introduces some artifacts that previous methods such as PCSS don't, including aliasing near objects and "Peter-Panning" (objects appearing detached from their shadows). For demo renders, I use 36 sample rays (defined using the same jitter pattern as PCSS) with 64 ray marching steps per ray. I find that for interactive applications, this is far too slow, and I reduce it to 16 sample rays at 16 steps each.

3 Results

Figures 2 through 9 include sample images rendered by these implementations. If I were to choose a go-to from these methods for real-time applications such as games, I would likely select PCSS due to its relative speed, stability, and overall quality.

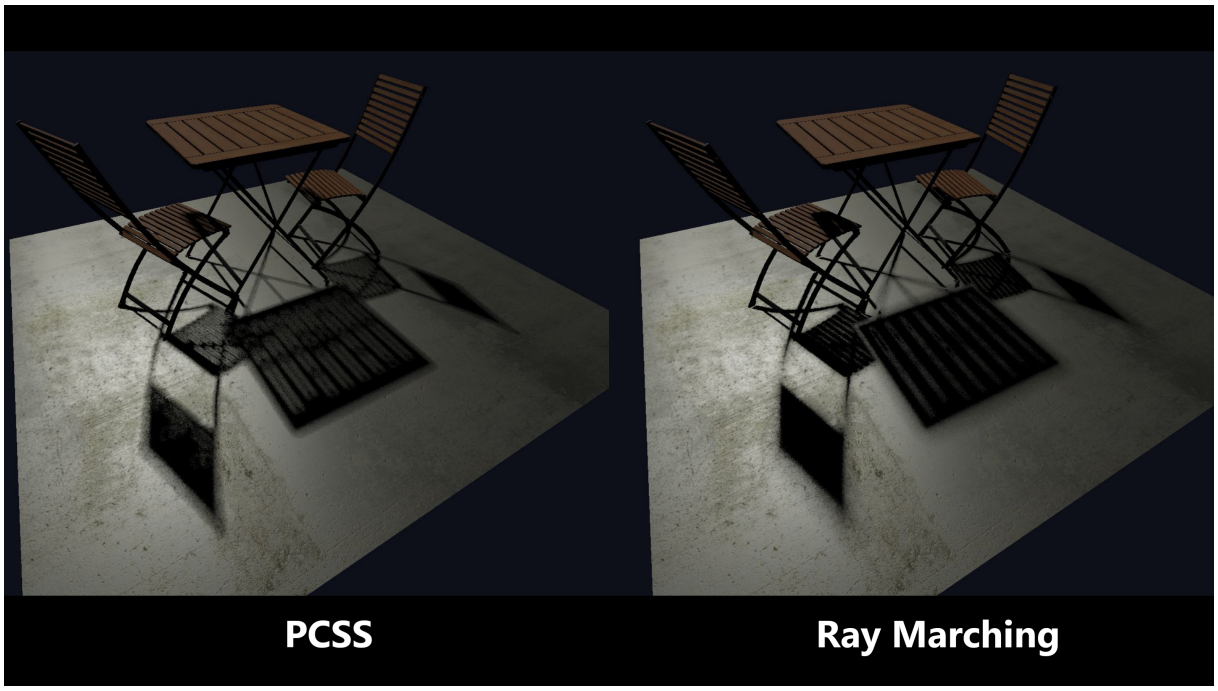

Figure 3: PCSS (left) and the Ray-Marching method (right) using a light source with a large radius. The Ray-Marching method develops artifacts, especially near the edges of thin objects.

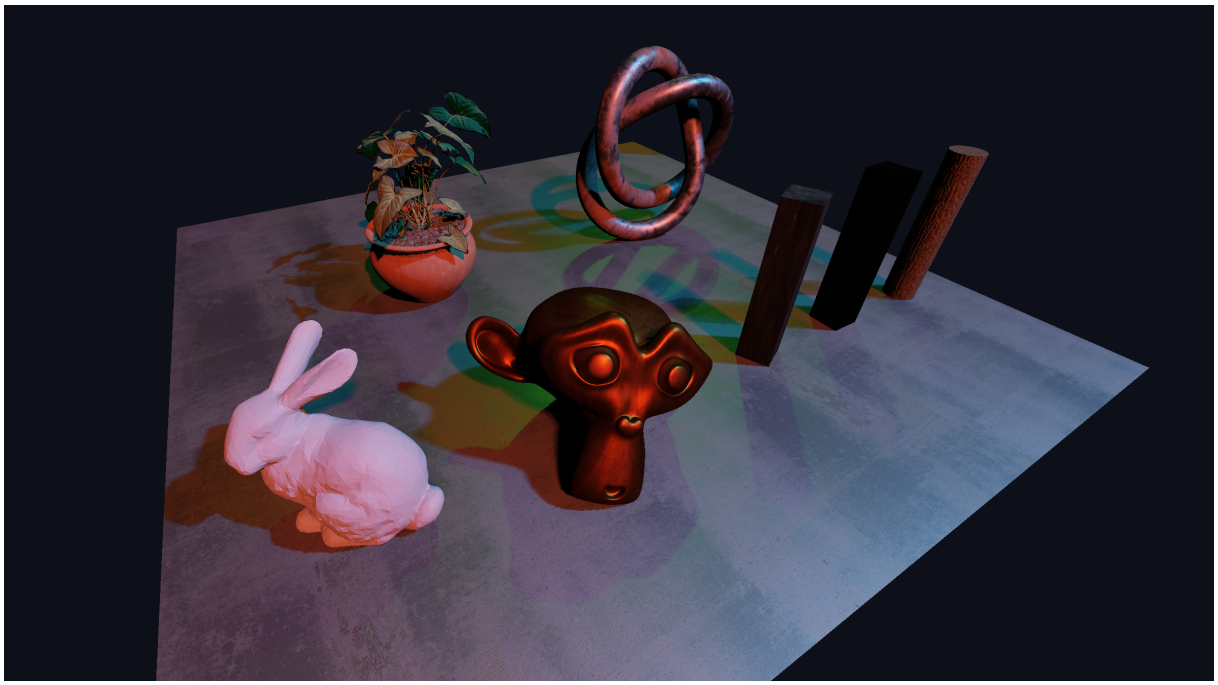

Figure 4: A sample scene with three different directional light sources with different colors, all using PCSS for soft shadows.

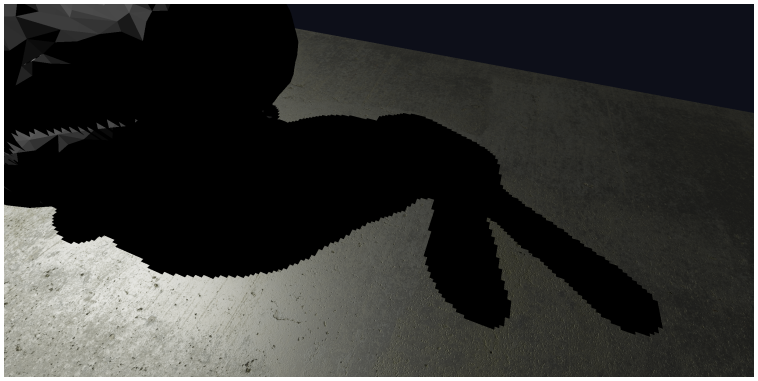

Figure 5: Basic shadow mapping

Figure 6: Percentage-Closer Filtering (PCF)

Figure 7: Jittered PCF

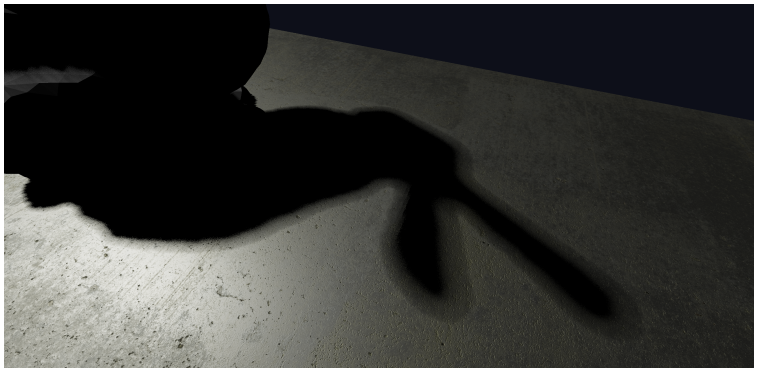

Figure 8: Percentage-Closer Soft Shadows (PCSS)

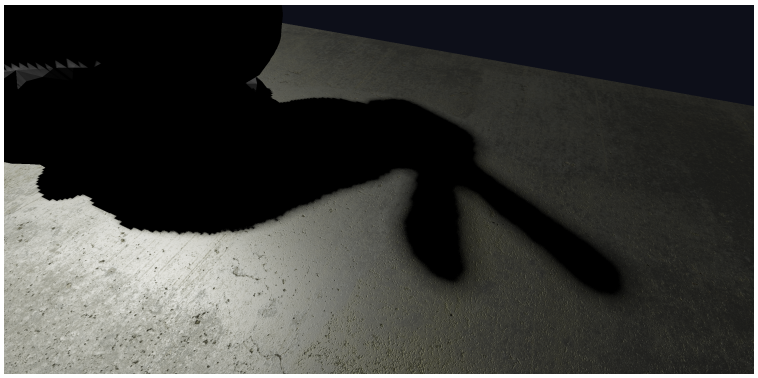

Figure 9: Ray-Marching

References

[1] Joey de Vries. LearnOpenGL - Parallax Mapping — learnopengl.com. https://learnopengl.com/Advanced-Lighting/Parallax-Mapping.

[2] Joey de Vries. LearnOpenGL - Shadow Mapping — learnopengl.com. https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

[3] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, page 35–es, New York, NY, USA, 2005. Association for Computing Machinery.

[4] Yury Uralsky. Chapter 17. Efficient Soft-Edged Shadows Using Pixel Shader Branching — developer.nvidia.com. https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-17-efficient-soft-edged-shadows-using.