CS283 HW4: Deep Shadow Maps

Richard Lin

Program Usage

The program is invoked as follows:

```
cs283-hw3.exe <input scene> [options]
```

<input scene> is the xml scene descriptor to load. For this project, the focus was on getting shadowParticles.xml to work.
[options] are as follows:
    --res, -r <x resolution> <y resolution>: sets the rendering resolution

The DLLs required are freeglut.dll, FreeImage.dll, glew32.dll, and glew32mx.dll.

You must execute the program in the included `assets` folder; in particular, it depends on the shaders and art materials located there.

Controls

Mouse movement: camera rotation
`w a s d r f`: camera movement, hold `(shift)` to slow movement
`z x` zoom control
`Esc` exit

Additional debugging commands to visualize the deep shadow maps are as follows:
`y h`: go up / down a shadow map layer
`u`: toggle debug (shadowmap visualization mode)
`j`: toggle visualizing depth component and luminance component

Part I. Introduction

The last project concluded with fancy shadow mapping, where shadows against an opaque object are calculated by taking into account semi-transparent objects between the light source and surface. However, this can only produce nice effects where the semi-transparent objects aren't affected by lighting. While it works well for simple effects (like a glass block), it doesn't work well for complex scenes.

Motivation

The motivating scene for this project will be from this clichéd quote:

> ***'It was a dark and stormy night'***

The idea is to simulate a single spotlight in rolling fog (imagine a streetlight on a foggy night). Realistically, we would expect the lamp to light up the fog, and the fog to cast a shadow against the ground. Extending HW3 to play nicely with particles, this works:

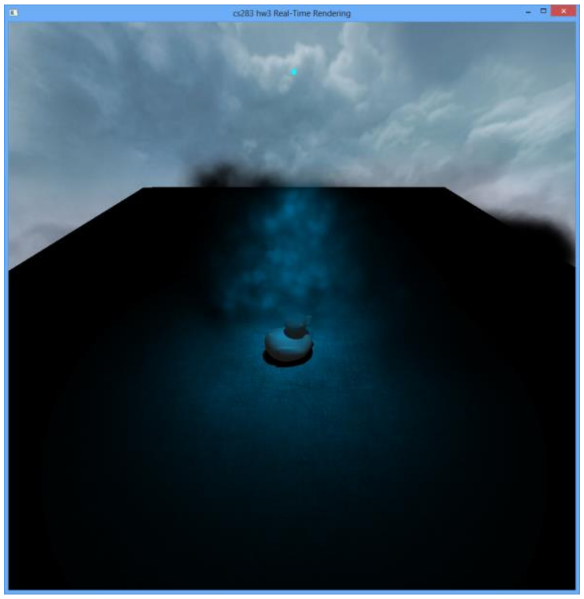

Figure 1: shadows with particles: the blue dot in the sky is the light source, and there is a particle emitter sending smoke particles from the left side towards the right side.

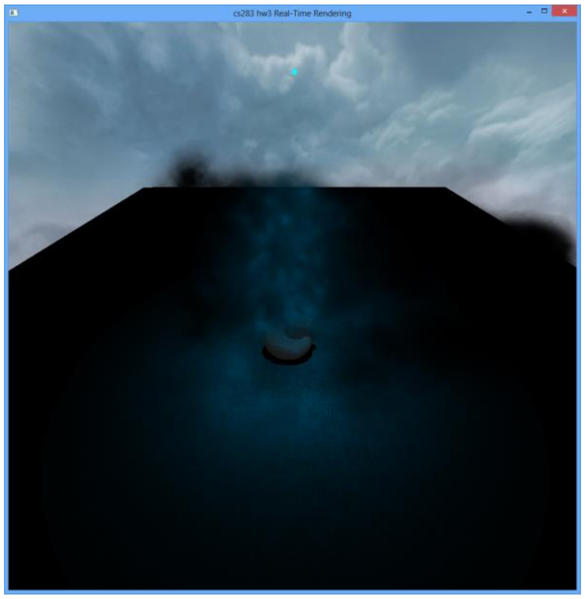

Figure 2: the same scene, with lower transmittance through the particles. Note the shadow of the particles cast onto the floor.

However, we also expect the fog to absorb light – a self-shadowing effect. Therefore, the fog should be brighter on the top than the bottom, something that doesn't happen here.

As seen in the comparison between the two images, there is self-shadowing, just incorrect. With the current way shadow maps work, anything illuminated by the light source is given the luminance seem at the far wall. Therefore, the light at any point in the fog is the light at the furthest point in the fog. This obviously is not current.

Deep shadow mapping should solve that problem – where instead of one shadow map for the light, representing all depth values from the further opaque surface, multiple shadow maps are kept per light, allowing different illuminances at different depths, giving proper self-shadowing behavior.

Part II. Basic Particle Lighting

As the first step, the particle system was updated to play nice with shadowing and lighting. In particular, the old lighting model (which does diffuse shading based on normal and dot products) did not work well with particles, since a particle's normal (with respect to lighting) means nothing. Although a particle is rendered as a 2d plane, the effect is to simulate a cloud, so it essentially should by omnidirectional. Therefore, when rendering particles, the lighting shader assigns a 1.0 to the dot product – so the fragment is facing the light source regardless of the particle plane's orientation.

With this done, the output looks like Figures 1 & 2 in Part I. Looks fair, but not super-good.

(before this part, particles and diffuse lighting looked awful – particles were ambient lighting only)

Part III. Deep Shadow Mapping

In the second step, deep shadow mapping was implemented, where separate luminance values are kept for each semi-transparent surface the light passes through. This simple version keeps luminance in discrete steps, with a discontinuity in the luminance function for each surface it passes.

The results with the scene from Part 1 (light and fog) are as follows:

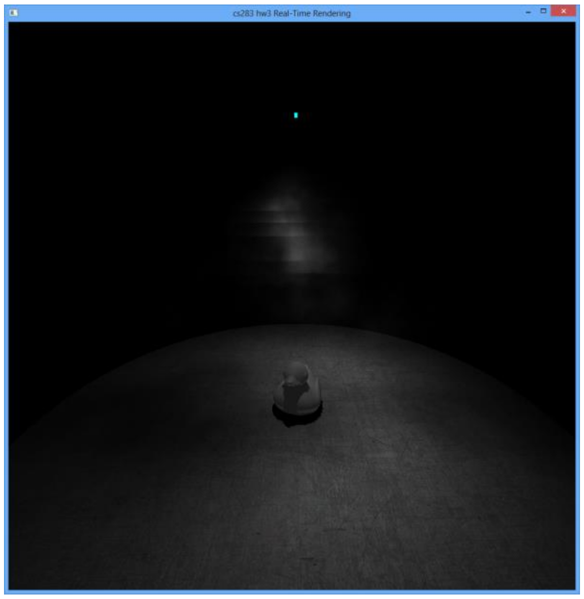

Figure 3. Simple deep shadow maps with a handful of particles and 16 shadowmap layers. The effect isn't clear, though you can see the self-shadowing effects which form the lightbeam.

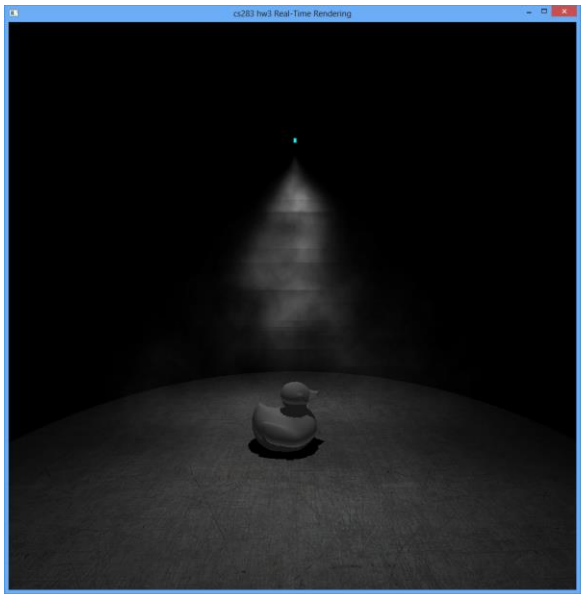

Figure 4. Same rendering engine with more particles and 64 shadowmap layers. Particle parameters like light occlusion have been tweaked accordingly. Self-shadowing here is more obvious, with brighter fog on top.

Implementation Details

Deep shadow mapping was implemented by using depth peeling to render layers of the shadowmap to an array of 2d textures. The rendering engine is OpenGL based, built on top of previous cs184 and cs283 projects.

The deep shadow map itself is a texture2DArray (array of 2d textures, kind of like (but not exactly) a 3d texture). Each texel entry is two floats: one keeping the depth of this slice, and one tracking luminance reaching this slice.

The map is calculated in the fragment shader using an iterative depth peeling process, starting from the light source and expanding outwards. On each iteration, the position is checked against the previous position and the furthest (opaque) position; fragments with positions not in this range are discarded. The native OpenGL Z-buffer test takes care of the rest, outputting the fragment of the surface that is the next closest to the previous. The iterations repeat until the number of specified layers. Empty layers are possible, and even likely, since not having enough layers will cause graphical artifacts.

The map is then read in the lighting phase of the final render pass. Here, the map is first checked against the opaque depth, to check if the fragment is completely shadowed. If not, it iterates through the deep shadowmap layer-by-layer, until it finds a layer where the shadowmap depth is greater than the projected fragment depth. It then uses the corresponding luminance value in the lighting calculation.

As for the fog effect, the particles are rendered facing the camera during each render pass. Therefore, during shadowmapping, the particles face the lights, and during the render pass, they face the eye. This results in the (very noticeable) discontinuity in the middle of each particle, where the shadowmap plane is perpendicular to the eye and rendered particle plane.

Part IV. Fine-tuning

Discrete layers do well for some applications, where the actual luminance function is discontinuous (say, passing through transparencies). However, applying this method on fog produces visual artifacts, where the discrete steps between layers are clear. Interpolating the luminance values based on the fragment depth can help alleviate this problem:

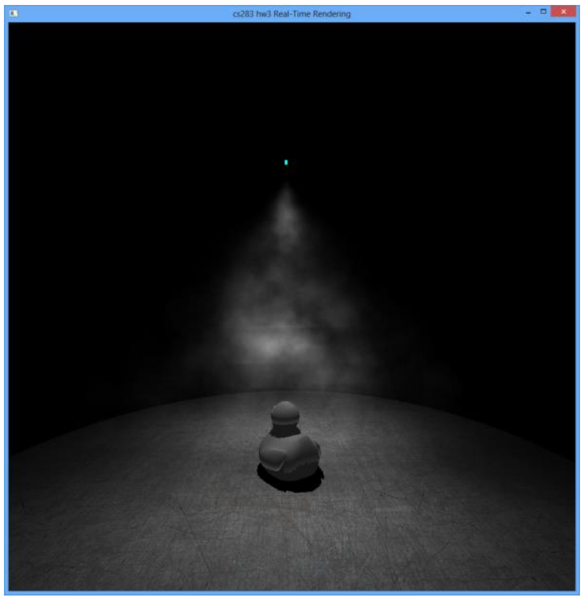

Figure 5. Interpolated deep shadow maps. The layer boundaries are not as visible, but some artifacting can still be seen. 64 shadowmap layers.

Part V. Conclusion

Particles are faking it, which means trade-offs. Deep shadow maps are one way to provide a more realistic effect, where particles can cast shadows other particles. In the videos, it's possible to see effects where particles cast beams of light and significantly occlude each other.

However, faking it is still faking it, and there are issues. One is the space consumption: a 2d texture array will eat memory. Furthermore, they are expensive to calculate (in the images and videos, each shadowmap layer was 512x512, giving it a coarse look – yet the framerates were still barely interactive, though on integrated graphics), and at the end, particles are still flat surfaces. A significant amount of tweaking was necessary to generate the above pictures – for example, particles had to be spawned so that they were in the light source's cone when rendered perpendicular to it. In general, the final rendered particle has to be "aligned" to significant features in the shadowmap to create a realistic effect – something which doesn't always happen. Overall, the effect isn't as good as I hoped it would be, but it's still an improvement over basic particle lighting, and is arguably more correct. A more realistic effect would be volumetric based, but probably be much more computationally expensive.

The End

Part VI. Rant

After working with around 6 OpenGL projects during cs184 and cs283, it’s time to rant. OpenGL makes for some of the most frustrating, difficult to debug, and unsafe masterpieces of spaghetticode. Why? Let me vent:

For one, OpenGL is very beginner unfriendly. While there are resources online, they are often outdated (new style OpenGL, old style OpenGL, even various versions of GLSL, and all kinds of extensions) and don’t work as expected. Furthermore, OpenGL silently drops errors – if you do something wrong, OpenGL will happily chug along with broken code. It doesn’t lead to immediately obvious effects, but when the bugs materialize, they are difficult to trace, and have some of the most unintuitive (and also unintentional) fixes.

And speaking of spaghetticode, the state-machine nature is guaranteed to code into spaghetti, unless strict coding discipline is followed. Of course, for a student just learning graphics and OpenGL, these issues are unanticipated until it is too late, and a leaving the wrong framebuffer bound in the middle of a render call can royally screw up the operations of a method in a wholly different class. And let’s not even get into properly setting up a framebuffer object, especially when the compiler / runtime doesn’t tell you that you messed something up (and where/what you messed up – you’re only given a screen that obviously isn’t what you want it to look like – which is unhelpful, unlike modern runtimes, such as STL, where compiling it in debug mode causes it to execute runtime asserts that can help you pinpoint bugs). It’s amazing that the deep shadowmap code works at all, though it’s probably fairly brittle.

Anyways, that’s it. At the end of the day, OpenGL is probably a very flexible option, providing you with performance low-level control while also giving you enough rope to hang yourself. It’s also the only major cross-platform 3d graphics library.