UM EECS 487
Lab 8: Projected (Soft) Shadows with Framebuffer Object

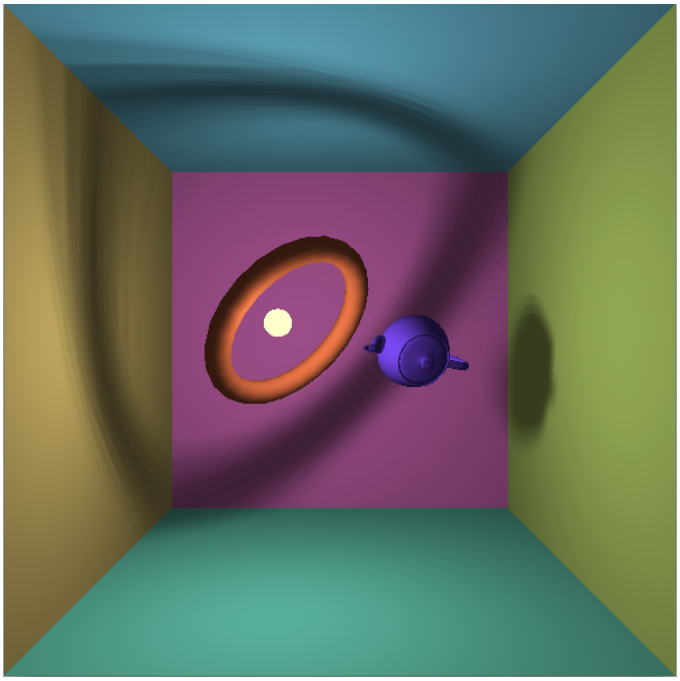

FIGURE 1. A scene with a torus and a teapot. A light floats inside the torus, casting eerie, soft shadows.

This lab is divided into four parts: the first is to add hard shadows to a scene, the second to use the stencil test to clip the shadows on the walls, the third to add soft-shadows using the accumulation buffer, and the fourth to use a framebuffer object as accumulation buffer, all to be done in OpenGL.

1. HARD SHADOWS

To produce a hard shadow on the planar walls, populate the shadow projection matrix `shadowMat` in `multShadowMat()` using the light position `lightPos` and the implicit function of the receiver wall (see the lecture notes on *Interactive Visual Effects: Projected Shadows*). The implicit function of a wall $Ax + By + Cz + D = 0$ is defined by four elements (A, B, C, D). Inside `multShadowMat`, the receiver wall is given to you and all you have to do is fill in the elements of `shadowMat[]`. After this is done you should have hard shadows visible in the scene.

2. CLIPPED PROJECTED SHADOWS

In the provided program, press the '`x`' key. The walls and ceiling will disappear, leaving only the floor. You'd immediately notice that the shadows extend beyond the edges of the floor. Continue pressing the '`x`' key to cycle through each of the other walls (or '`X`' to cycle through the walls in reverse order). You may want to also move the light, or rotate ('`r`') the torus and teapot, around to cast shadows on the visible wall. The shadows continue to extend beyond the edges of each wall. Your task in this part of the lab is to clip off the shadows that extend beyond the edges of the wall using the stencil test (see the lecture notes on *Interactive Visual Effects: Stencil Buffer*).

In the function `drawScene()`, as each wall is to be drawn, you initialize the stencil buffer in three steps: First set the reference value of the stencil test to 1 and set the stencil test to always accept a fragment. Next replace the contents of the stencil buffer with the reference value (1) wherever a fragment of the wall is drawn. Then call `drawWall()` to draw each wall (provided). As the wall is being drawn, the stencil buffer will be populated as a side effect, transparent to you. After drawing a wall, your stencil buffer is now initialized. Next reset the stencil test to accept a fragment only if the stencil buffer content is 1 (i.e., only if the fragment is within the boundaries of the wall). Then draw the shadows by calling `drawShadows()` (provided). The shadows will only be drawn when a fragment passes the stencil test you've set up in the previous step, using the contents of the stencil buffer you've set up in the first three steps. Each of the three steps that you're asked to do consists of a single line of OpenGL call. Your task is to figure out which functions to call and what arguments to give each function.

Now when you hit '`x`' or '`X`', you should be able to hide the walls and observe the shadow clamped to the edges of the walls. Table 1 lists the available key bindings. Moving the light around the scene should work out-of-the-box and the effect of the light on the walls, torus, and teapot should be immediately visible. The '`f`' key toggles soft and hard shadows but this will have no affect until the two are implemented. By default, the walls are each tessellated as sixteen quads. If your computer is not that powerful, you may want to try a smaller number ('`t`'). When tessellation is too low however, lighting may become unintuitive (since lighting is computed per-vertex).

TABLE 1. Key Bindings

ESC or q	Quit the application
h	Move the light left
j	Move the light down
k	Move the light up
l	Move the light right
w	Move the light back
s	Move the light forward
r	Rotate the sphere and the torus
f	Toggle soft/hard shadows
-	Decrease the number of light jitters
+	Increase the number of light jitters
d	Decrease the amount of light jitter
a	Increase the amount of light jitter
t	Decrease wall tessellation
T	Increase wall tessellation
c	Toggle stencil-based clipping of shadows against receiver geometry
x	Show/hide the walls and ceiling, cycling through them
X	Show/hide the walls and ceiling, cycling in reverse order

3. SOFT SHADOWS

To produce soft shadows, render the scene multiple times. Each time, jitter the location of the light slightly (a function of `jitterAmount`). Then average all the renderings. Since the geometry does not move, it will be crisp. The shadows will be slightly different each render, so when they are averaged a softness, or blurriness, appears. You can accomplish this in three steps inside the `display()` callback function. First, clear the accumulation buffer and jitter the position of the light before render starts. For each new light position, call `drawScene()` to render the scene. Second, after each render, copy the contents of the frame buffer into the accumulation buffer using `glAccum()` (see the lecture notes on *Interactive Visual Effects: Accumulation Buffer* and the section on *Accumulation Buffer* in the Red Book, Ch. 10). This should be a weighted copy, such that the renders sum up correctly. For instance, if you perform two separate render passes, each render should be copied into the accumulation buffer with a weight of 0.5. The number of passes, which is also the number of light jitters, is controlled by the global variable `numJitters`. Finally, after the final render pass has been accumulated, move the accumulated contents from the accumulation buffer over to the frame buffer, with full weight.

4. FRAMEBUFFER OBJECT

The accumulation buffer has been deprecated in OpenGL since version 3.1. In this part of the lab, you're asked to implement the functionalities of the accumulation buffer using a frame buffer object. We have provided three skeletal functions `fbo_accum_init()`, `fbo_accum()`, and `fbo_accum_return()` that are designed to be one-to-one replacements for the OpenGL accumulation buffer's APIs. In `drawScene()`, replace your OpenGL accumulation buffer calls with these functions. Then, fill out each of the functions following the instructions in the code and the lecture notes on *Interactive Effects: Using FBO as Accumulation Buffer*.

5. BANDING IN A LIMITED BUFFER

Graphics cards have varying capabilities when it comes to buffer sizes and pixel formats. For instance, a given system may support a 12-bit accumulation buffer, 8-bit, or none at all (in which case its functionality is performed in software). Thus when `glutInitDisplayMode(GLUT_ACCUM)` is called, a minimum buffer (or none) may be allocated. Fig. 2(a) shows the result of running the code from this lab on a system that has an 8-bit accumulation buffer (and frame buffer). (The function `glutInitDisplayString()` allows one to specify the size of the buffers explicitly, but not all operating systems support this function. One also cannot set pixel format from GLUT.)

Imagine rendering a quad with color ranging from red at the top to black at the bottom. The red component of the color gradation would have these values:

$$255 \quad 254 \quad 253 \quad \ldots \quad 3 \quad 2 \quad 1 \quad 0.$$

If the accumulation buffer has only 8 bits for each color component, and the quad is rendered eight times into the accumulation buffer with 1/8th weight each, pixels with red component 240-255 will all be mapped to the final color 240. This loss of precision produces banding (the image will be 16 vertical bands ranging from red to black). When there is not enough precision, the final colors are discretized. You can see this effect for yourself by replacing `GL_RGB32F` with `GL_RGB` when you

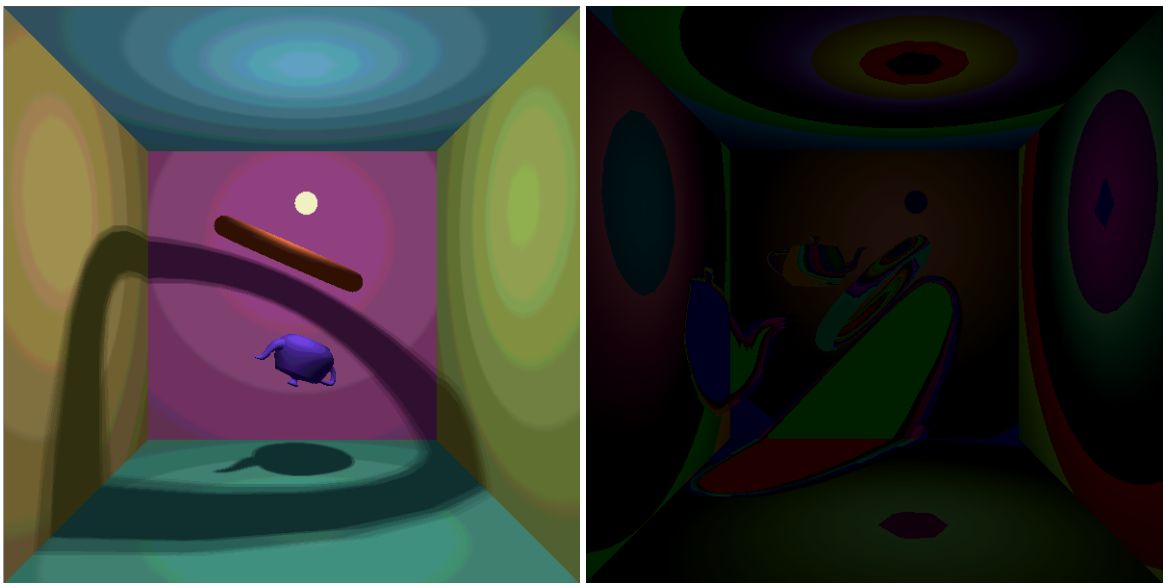

FIGURE 2. Results from (a) 8-bit accumulation buffer showing banding and (b) averaging the sum of all renders at the end.

create the renderbuffer storage to be used as your accumulation buffer. Even with `GL_RGB16F`, one starts to see banding at `numJitters` ≥ 4.

One might think that it would be better to add up the individual renders and *then* divide the total. Unfortunately, the accumulation buffer is bounded in [-1,1] and adding up the individual renders will cause overflow into the negative numbers, which at the end will be clamped to the [0,1] range. Thus the image will be completely incorrect and mostly in the dark region, as shown in Fig 2(b).