UM EECS 487

Lab 9 OpenGL (Soft) Shadows

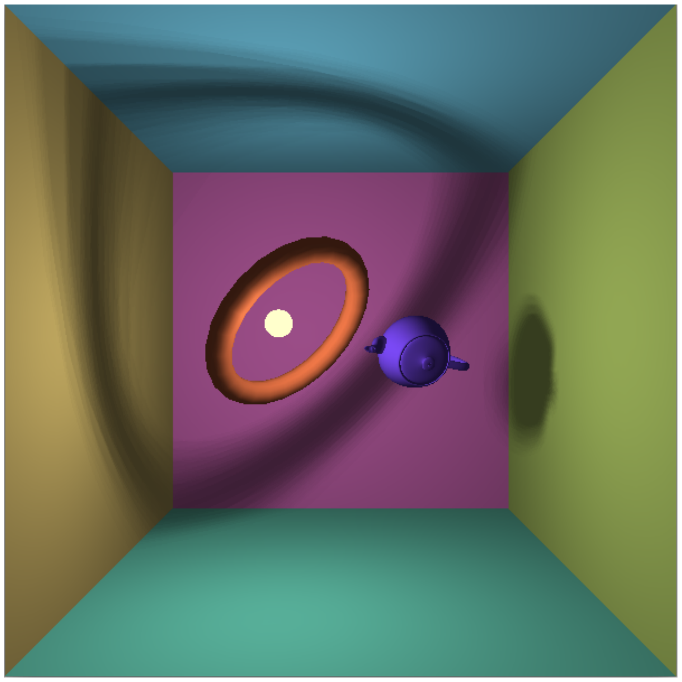


Figure 1. A scene with a torus and a teapot. A light floats inside the torus, casting eerie, soft shadows.

1. Hard Shadows

This lab is divided into three: the first part is to add hard shadows to scene, the second part to add soft-shadows, and the third to use the stencil test to clip the shadows to the walls, all to be done in OpenGL. The provided display callback to produce hard shadows functions as follows:

- Clear the buffers
- Position the OpenGL light and draw the glowing sphere to represent it
- For each wall
  - Draw the wall (an OpenGL quad)
  - Set the current shadow projection plane to be that wall
  - Multiply a shadow matrix onto the ModelView Matrix
  - Draw the teapot and the torus with black (the "shadows")
- Draw the teapot and the torus

All of this is already taken care of except the step that projects the shadow matrix. This is done with a call to `multShadowMat()`. As provided, this function just uses the identity matrix. Fill in the 16 elements of the matrix `shadowMat[16]` using `float posLight[4]` and the global variable `float flatteningPlane[4]`, the current plane to project against (see the lecture notes on *Projected Shadows*). The plane $ax + by + cz + d = 0$ is defined by four elements (a, b, c, d). When inside this function, the current plane has already been set 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. Soft Shadows

To produce soft shadows, the scene will be rendered multiple times. Each time, the location of the light will be jittered slightly. Then all the renderings will be averaged. 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, will appear. You can accomplish this in three steps, each of which is inside `void display()`. The first task it to jitter the position of the light before the render starts. Then, after the scene has been rendered, copy the contents of the frame buffer into the accumulation buffer using `glAccum()` (see the section on *Accumulation Buffer* in the Red Book, Ch. 10). This should be a weighted copy, such that the renders total correctly. For instance, if two separate render passes are to be performed, each render should be copied into the accumulation buffer with weight 1/2. The number of passes, which is also the number of light jitters, is given by the global variable `numJitters`.

The last step is, after copying the last render pass into the accumulation buffer, move its contents over into the frame buffer, and with full weight. At this point, right-clicking inside the window will bring up a menu for you to choose between soft and hard shadows. (Aside: the accumulation buffer has been deprecated in OpenGL since version 3.1. In its place, we would use a frame buffer object with floating-point pixel format instead. The basic technique will be the same in both cases. However, the accumulation buffer is still currently more widely available than frame buffer objects. It would be acceptable if you choose to use a frame buffer object instead of the accumulation buffer to implement this lab.)

Table 1. Key Bindings

ESC	Quit the application
h	Move the light left
j	Move the light down
k	Move the light up
l	Move the light right
b	Move the light back
f	Move the light forward
r	Rotate the sphere and the torus
s	Toggle soft/hard shadows
x	Show/hide the walls and the ceiling
c	Toggle stencil-based clipping of shadows against receiver geometry
[	Decrease the number of light jitters
]	Increase the number of light jitters
;	Decrease the amount of light jitter
'	Increase the amount of light jitter
-	Decrease the wall tessellation
+	Increase the wall tessellation

Make the lab and then try these keys. 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. You can also right-click inside the window to toggle between hard and soft shadows but this will have no affect until the two are implemented. By default, the walls are each drawn as sixteen quads. If your computer is not that powerful, you may want to try a smaller number. When tessellation is too low however, the lighting may become unintuitive (since the lighting is computed per-vertex). Pressing the '+' and '-' keys will increase or decrease the number of quads used to draw each wall. Move the light around the scene and observe the behavior with different wall tessellations.

3. Shadow Clipping with the Stencil Test

The third part of this lab is to use the stencil buffer to clip shadows to area (on screen) covered by the floor. Press the 'x' key. The walls and ceiling will disappear and the light will grow a sun-like orb. If you move the light (or the torus/teapot) you will notice that the shadows extend beyond the edges of the floor. Your task is to clip off the shadows that extend beyond the edges of the floor using the stencil test.

In the function `drawScene()` the drawing of the walls, floor, ceiling, teapot, and torus is performed as explained above. Before drawing each wall, the stencil buffer is cleared and stencil testing is enabled. After drawing the wall the stencil test is disabled. The general procedure to use here is:

- Clear the stencil buffer, so it contains all zeroes.
  `glClear(GL_STENCIL_BUFFER_BIT);`
- Enable stencil testing.
  `glEnable(GL_STENCIL_TEST);`
- Set the stencil buffer's function to *always accept a fragment and prepare 1 as the new value to store.*
  `glStencilFunc(?, ?, 0xffffffff);`
  (The stencil buffer can be used to do much more than what we have done here. Part of its functionality involves a bitwise AND with a bitmask. Using `0xffffffff` simplifies the behavior of the buffer since we don't use the bitmask.)
- Set the stencil buffer's operation to only replace the contents of the stencil buffer if a fragment passes the stencil and the depth test. In all other cases, keep the current value that is already in the buffer.
  `glStencilOp(?, ?, ?);`
- Draw the reciever.
- Set the stencil buffer's function to *only accept a fragment when the associated stencil value is 1*, which is wherever the reciever is.
  `glStencilFunc(?, ?, 0xffffffff);`
- Multiply a shadow matrix onto the ModelView Matrix.
- Draw the occluder.
- Disable stencil testing.
  `glDisable(GL_STENCIL_TEST);`

After implementing this in `drawScene()` you should be able to hide the walls and observe the shadow clamped to the edges of the floor.

4. Banding in a Limited Buffer

It is important to remember that GLUT is about the most lightweight, yet functional, windowing toolkit that there is. It provides platform-independence while remaining simple and easy to use. All GLUT applications will contain a line similar to this:

```
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE | GLUT_ACCUM);
```

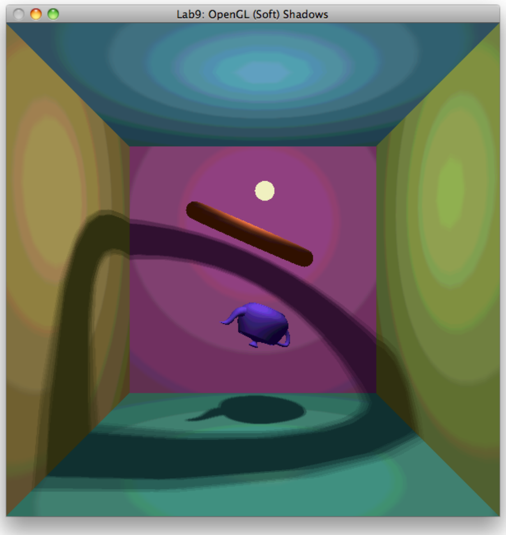
This line is responsible for the OpenGL context (which stores the texture objects, the shader objects, drawing attributes, transformation matrices, pixel format, etc.) that is created and the pixel format (number of bits per pixel, number of color components, etc.) with which it is created. In this example, the pixel format will be to store red, green, blue, and alpha color components; a depth value; a second, double buffer; and an accumulation buffer. Similar arguments, such as `GLUT_STENCIL`, can allocate/request other buffers, or specifications for the pixel format.

Graphics cards support varying capabilities when it comes to buffers 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 using an initializer such as `GLUT_ACCUM`, a minimum buffer (or worse none at all) may be allocated. All Operating systems have methods to create pixel formats very specifically, but this is not quite available in GLUT[1].

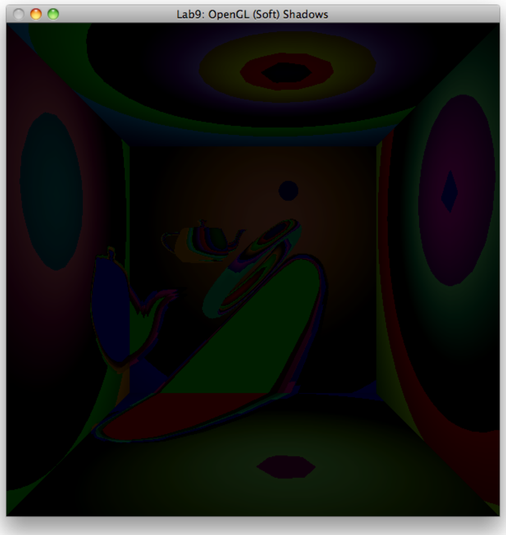
What happens when a buffer is allocated with limited size? Imagine rendering a scene with one quad filling the screen that is red on one side and black on the other. The buffer for the red color component would look something like this (for each row):
255 254 253 ... 3 2 1 0 Now imagine that an accumulation buffer with 8-bits (per component) is allocated. The scene described above will be rendered eight times into the accumulation buffer with 12.5% weight each and then the contents will be copied into the frame buffer.

Since the accumulation buffer has the same amount of precision as the frame buffer, there will be a loss of precision when multiplying each by 1/8 and then adding that to itself 8 times. For instance, the pixels with red component 240-255 will all be mapped to the final color 240. This loss of precision will produce banding (the image will be 16 vertical bands ranging from red to black). When there is not enough precision, the final colors are discretized. Here is an image of the completed lab9 on a system that only has 8 accumulation buffer bits (the same number as the frame buffer).

[1]There is a function `glutInitDisplayString()` that allows one to specify the size of the buffers explicitly, but this GLUT function is not supported on most systems.



One might think that it would be better to add up all the individual renderings and *then* divide by the total, thus averaging after instead of during. Unfortunately, the accumulation buffer is bounded in [-1,1] and this will cause overflow into the negative numbers, which at the end will be clamped up to the range [0,1]. All the numbers that overflowed into [-1,0] will be become zero. Thus the image will be completely incorrect and mostly in the dark region.


Lab9: OpenGL (Soft) Shadows