CSE 167 (FA 2021) Shadow Mapping. Final project expectations, guide and hints

In this final project topic, we implement shadows in OpenGL using the aid of texture. In essential idea is presented in the slides of 10/29. Bonus credits will be given if you also implement the perspective warping of the light space to rearrange the light space resolution to reduce aliasing.

Expectation

You will present a scene (like HW3) with shadows from at least one light source. The light source can be a distant light (light at infinity) or a point light.

While bonus credits will be given as long as you have excellent exposition and demonstration, one possible direction to get bonus is to implement one of the methods of *Perspective Shadow Maps* (PSM)[1], *Light Space Perspective Shadow Maps* (LiPSM)[2][3], *Trapezoidal Shadow Maps* (TSM)[4]. All of these methods take advantage of a perspective transformation to rearrange the density of the light rays so that we have less pixelated effect on the edge of the shadow.

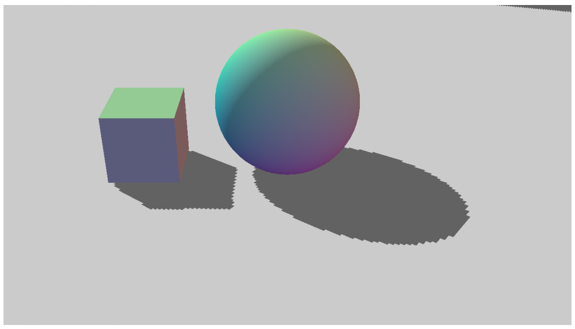


Figure 1 An example of demonstrating having shadows in the scene.

Basics

The shadow mapping technique requires two rendering passes. That is, in the display callback function, we will call draw scene twice. However, the two rendering are set with two different cameras and different shaders.

- In the first rendering, the camera is at the light responsible for casting shadows.
- In the second rendering, the camera is placed at the actual camera, forming the final image.

Each of the two cameras has a camera matrix $\mathbf{C}$, view matrix $\mathbf{V}$ (inverse of the camera matrix) and a projection matrix $\mathbf{P}$. Let us them $\mathbf{C}_{\text{light}}$, $\mathbf{V}_{\text{light}}$, $\mathbf{P}_{\text{light}}$ and $\mathbf{C}_{\text{cam}}$, $\mathbf{V}_{\text{cam}}$, $\mathbf{P}_{\text{cam}}$ respectively.

In the first pass of rendering (using $\mathbf{V}_{\text{light}}$, $\mathbf{P}_{\text{light}}$), we product an image whose pixel value is the depth of the scene. This depth can be the Z value after the multiplication by $\mathbf{P}_{\text{light}}$ (and dehomogenized (divided by the w coordinate)). That is, the Z value in light's normalized device coordinate. The depth value can also simply be the physical distance between the light (eye of light) and the fragment position.

[1]https://www-sop.inria.fr/reves/Basilic/2002/SD02/PerspectiveShadowMaps.pdf

[2]Original paper `https://www.cg.tuwien.ac.at/research/vr/lispsm/shadows_egsr2004_revised.pdf`

[3]Sec 3.2.2 of the course note `http://research.michael-schwarz.com/publ/files/shadowcourse-eg10.pdf`

[4]`https://www.comp.nus.edu.sg/~tants/tsm/tsm.pdf`

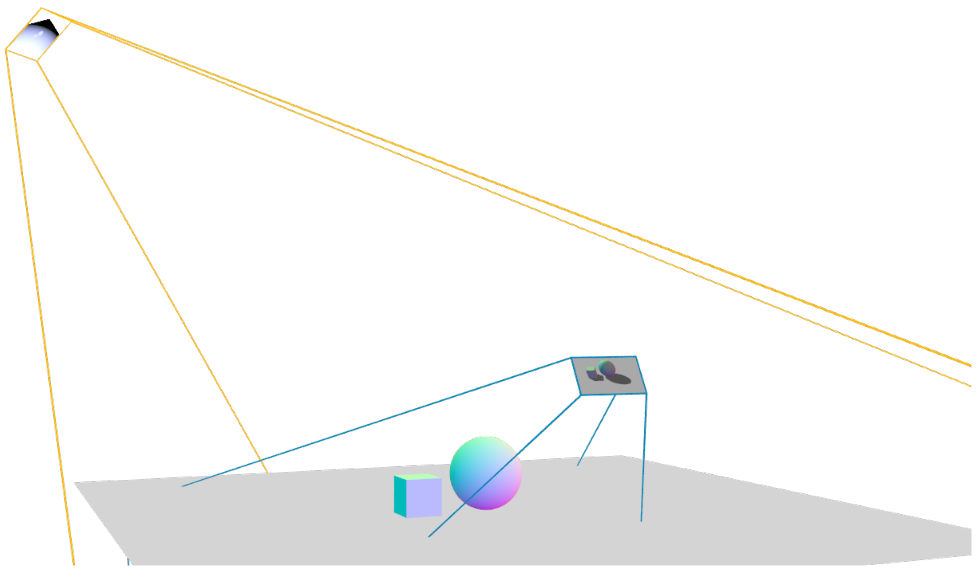


Figure 2 Two pass rendering. The first rendering from the light records the distance from the geometry to the light. The second rendering is the final image whose color is evaluated using information sampled from the first image.

Now, store this first pass rendering result in a texture.[5] Of course, as you develop the program, you may want to visualize this first pass rendering. In that case you can show it on the screen for the visualization purpose.

In the second pass of rendering (using $\mathbf{V}_{\text{cam}}$, $\mathbf{P}_{\text{cam}}$), render it as usual (like in HW3). The only difference from HW3 is in the fragment shader where we shade the color.

- If $\mathbf{n} \cdot \mathbf{l} < 0$, the surface is already facing away from the light, so we don't need to invoke the shadow technique.
- If $\mathbf{n} \cdot \mathbf{l} > 0$, then we check whether we are in the shadow or not:
  - Transform the fragment position to the light's normalized device coordinate (multiply by $\mathbf{P}_{\text{light}}\mathbf{V}_{\text{light}}$, dehomogenize). This is the coordinate of the fragment in the 1st pass image.
  - Note that the texture coordinates range in $[0, 1] \times [0, 1]$, whereas the XY coordinate in the normalized device coordinate has the range of $[-1, 1] \times [-1, 1]$. A simple rescaling is needed.
  - Sample the depth that was recorded in the texture.
  - Also compute the depth value of the current fragment. If the depth value was computed by the physical distance between the light and the fragment, then do the same calculation for this fragment.
  - If the sampled depth is (much) shorter than the new depth value, then we have a shadow.
- Add contribution of the light only when it is not under the shadow. An exception is the ambient component of the lighting. Without a bit of ambient shading, the shadow would look pitch black.

[5] See http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/ for direct rendering into a texture buffer.

Implementation Tips

We recommend you to implement this project by extending your HW3 code. Here's a short list of necessary modifications needed to be taken care of:

- In `main.cpp`:
  - Add a depth rendering pass to `display()`.
- In `include/`:
  - Derive a `DepthShader` structure from `Shader`.
- In `include/light.h`:
  - Augment the `Light` structure to store shadow map texture / buffer object.
- In `Scene.cpp`:
  - Augment or add a `draw()` method to handle shader swapping on demand.
- In `Scene.inl`:
  - Add a ground plane to the scene so that the shadow effect could be better observed.
- In `shaders/`:
  - Add a `lightspace.vert` vertex shader, which transforms vertices to light space.
  - Add a `depth.frag` fragment shader, which writes depth information to fragments.
  - Augment `lighting.frag` to incorporate shadow computations. Specifically, you'll append a `uniform sampler2D shadowMap;` texture sampler to sample the depth map, and pass in an additional `fragCoord` in light space.

Next is how to setup the depth map and its corresponding frame buffer object. Here's a quick rundown of the procedure:

1. Generate a frame buffer object.

```
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

2. Create a 2D texture for the depth map.

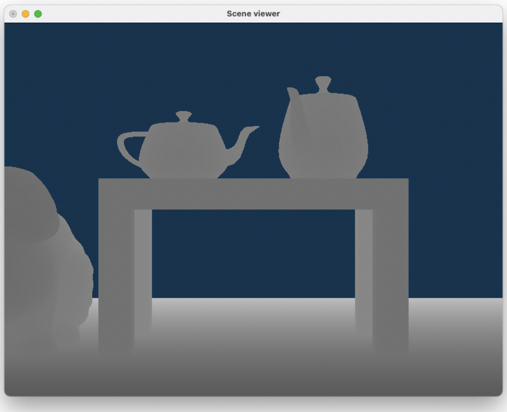
```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;

GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
             SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT,
             GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

3. Attach `depthMap` to `depthMapFBO`'s depth buffer:

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);  // Omitting color data
glReadBuffer(GL_NONE);  // Omitting color data
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

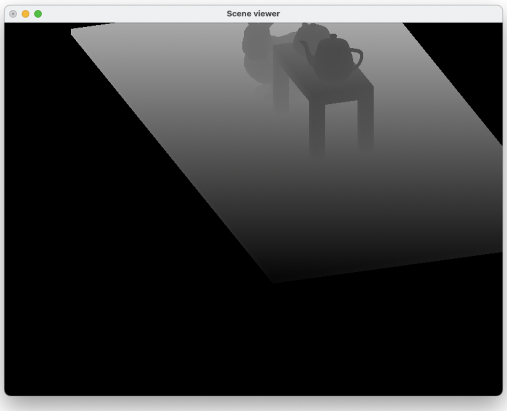
Implementing a project of this size can easily get you lost, so here's a few checkpoints we've curated to help you stay on the right track:

1. Get your depth shader ready. Once you've finished the additional shader implementation. Compile them and use their shader program to render the scene from main camera. It should produce something like this:



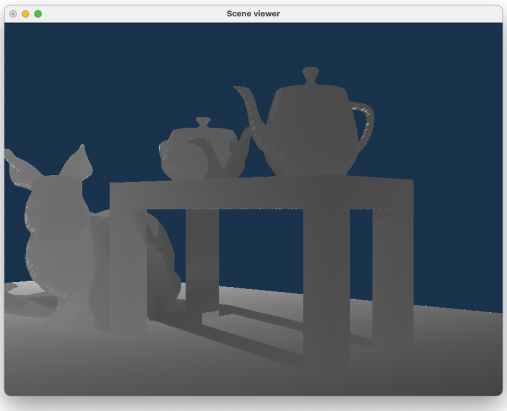
Tip: If yours looks all white, scale the depth down by 0.1, or remap it using a nonlinear function.

2. Check your light space transformations are all good. Hotwire your render pipeline in `main.cpp` to output whatever your light is seeing.



Tip: Viewport is expected to look clamped if your shadow map has higher resolution than screen.

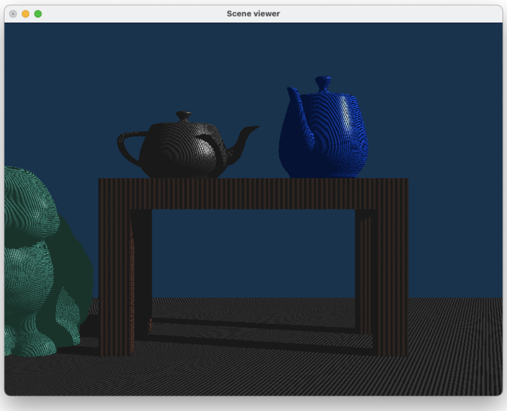
3. Before final lighting computation, check if depth from the depth map is correctly passed-in. A common trick to debug shader is to convey information in colors. You can view this from your main camera, simply let `fragColor = vec4(vec3(depthSampled), 1.f);`. You should see something like this:



Tip: Artifacts at greasing angles from light's perspective are results of z-fighting.

One more thing to mention,

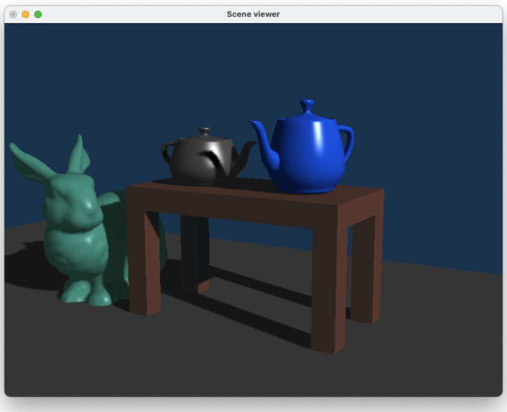
- Shadow acne: if you're getting something like this:



  This is a common shadow mapping artifact called shadow acne. It is a type of aliasing artifact due to limited resolution of the shadow map, such that discretized depth values undersamples actual depth information and causing disagreement between the shadow map and lightspace fragment coordinates during occlusion test. This can be resolved by introducing a slight bias to the sampled depth from the shadow map:

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
float shadow = LSDepth - bias > sampledDepth ? 1.f : 0.f;
```

Lastly, here's a reference "solution" for you to comapre with:



Advanced

As you can see in Figure 1, the edges of the cast shadows have lego/pixelated boundary. This is due to the finite resolution of the image of the first pass rendering. You are encouraged to work on a solution to this problem using a projective transformation. (The principle of shadows is invariant under collinear transformation. As long we map straight lines to straight lines like in projective transformation, the resulting light-visibility will be the same, but with different distribution of resolution!)

There are two ways one can perform projective transformation without messing up the light directions:

- Work in the normalized device coordinate (NDR) of the camera. (Perspective shadow maps, or PSM)
- Work in the normalized device coordinate of the light. (Light space perspective shadow maps, or LiSPSM)

In these post-perspective coordinates, the light rays from either the camera or the light will be parallel to the Z axis. This somewhat makes analysis easier.

In PSM, we can just set up the light (*i.e.* its view matrix and projection) in camera's NDR. In this coordinate system, objects closer or further from the camera will be of the same size, and thus we have a more evenly spread resolutions. Pretending the world space is NDR, we need to work out the eye position, target, *etc.* of the light. The rest is the same as the basic shadow map.

In LiSPSM, as shown in Figure 6, one performs a perspective transformation in light's space. One must choose this additional transformation so that the rasterizer's Z direction remain the same. The remaining X and Y can be tapered and non-uniformly rescaled to gain more resolution closer to the camera.

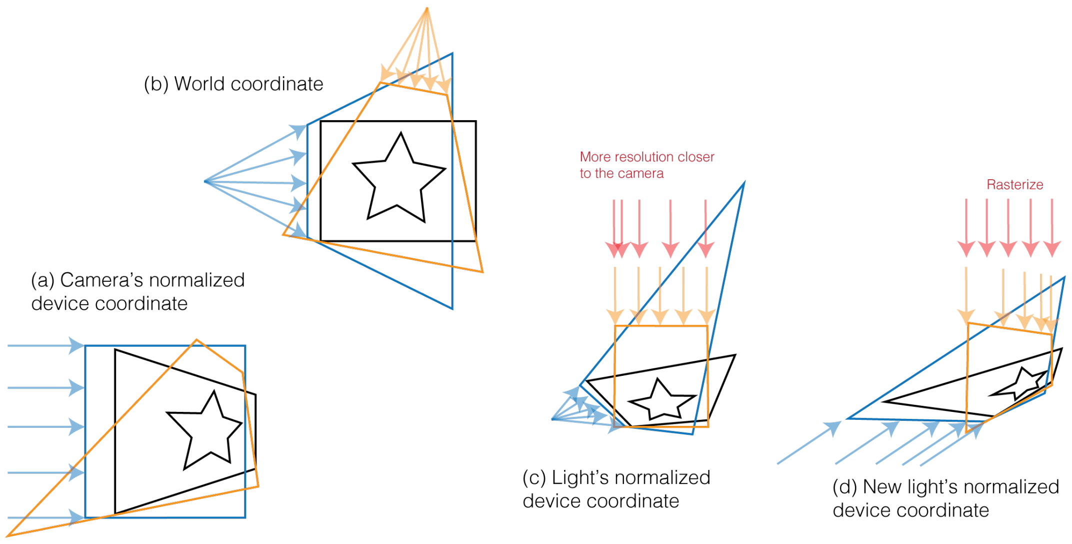

(b) World coordinate
(a) Camera's normalized device coordinate
More resolution closer to the camera
(c) Light's normalized device coordinate
Rasterize
(d) New light's normalized device coordinate

Figure 6 In Light Space Perspective Shadow Map (LiSPSM), one performs an additional perspective transformation on (c) light's normalized device coordinate (NDR) to get (d) a new light's NDR. In the new light's NDR, the Z axis remains the same so that the rasterization visibilities are equivalent. However, the density of pixel is rearranged (red vs orange) so that we have more light-space pixel resolution closer to the camera.