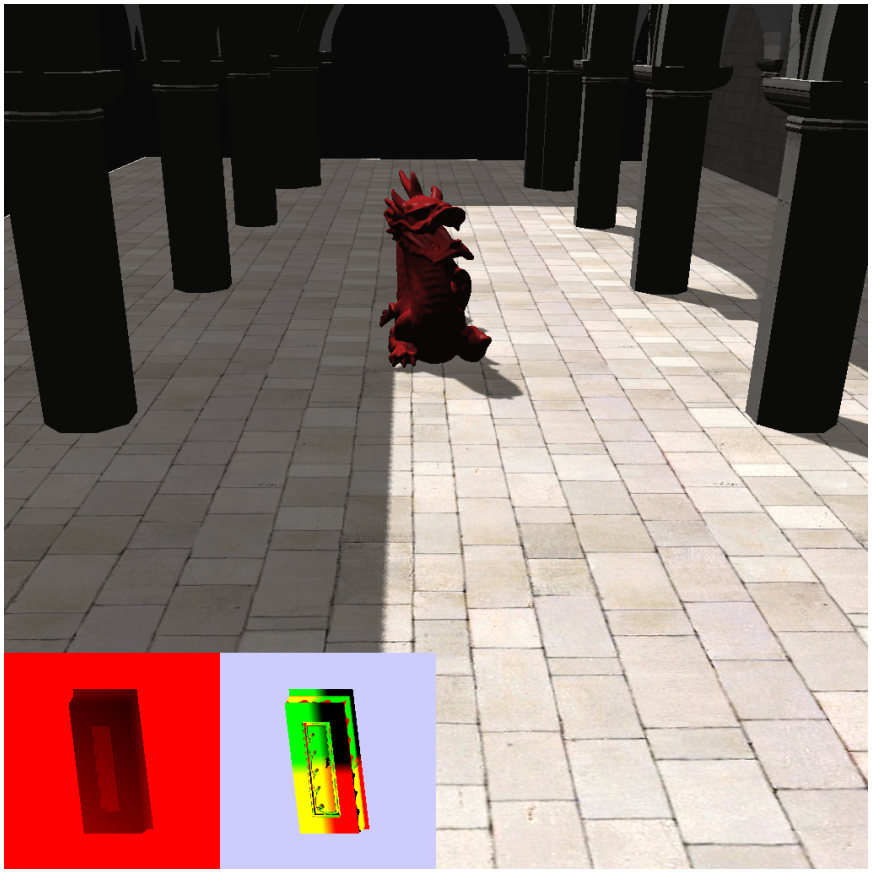
6.837: Computer Graphics Fall 2019

Programming Assignment 5: Shadow Mapping

Due Wednesday, November 20 at 8:00 pm.

In goal in this assignment is to implement Shadow Mapping[1], a popular algorithm for rendering shadows in real-time. The assignment is implemented in OpenGL. This handout contains step-by-step instructions for how to draw a mesh, and how to apply textures in OpenGL. Most of the assignment is implemented in C++, with a few parts written in GLSL shading language.

Please also note that your **final project proposal** is due on **November 15 at 8pm**.



[1] https://people.eecs.berkeley.edu/~ravir/6160/papers/p270-williams.pdf

1 Getting Started

Run the sample solution as `$ sample_solution/athena/a5`

By default, the starter code will try to load meshes and shaders relative to the current working directory. To run the binary from an arbitrary path, e.g. the build folder, pass the base directory as first command line argument: `build$ ./a5 ../`

The starter code will load a simplified version of the sponza scene from the `data/sponza_low` directory. To make things more interesting, we added a dragon mesh.

2 Summary of Requirements

Vertex Data in OpenGL

As part of the starter code, we load a relatively large model file (about 200k vertices) for you. Your first task is to draw the mesh using the familiar `VertexRecorder` class.

Texture Mapping

So far, we only used objects with uniform appearance (e.g. appearance that does not vary over the surface of the object). Spatially-varying materials are more interesting, so in this part of the assignment, you will

- Create OpenGL texture objects.
- Specify which texture to use when drawing.
- Modify the shader code to sample from the texture using texture (U-V) coordinates.

Shadow Mapping

Until now, objects in the scene do not cast shadows. The most prevalent technique to render shadows in real-time is *Shadow Mapping*. You will implement shadow mapping in this step. Along the way, you will

- Learn how to render to a texture, rather than straight to the screen.
- Direct OpenGL to read from this texture in the next draw call.
- Implement the shadow mapping algorithm as part of the fragment shader.

3 Drawing the Mesh

Draw the mesh in the `drawScene(GLint program, Matrix4f View, Matrix4f Projection)` function. Later, we will render the same scene from the point of view of the light source, so we pass in the GLSLprogram, as well as light and camera matrices.

We implemented a simple .obj parser in the `objparser.cpp` file. It reads the Sponza scene into vertex and index arrays.

In `objparser`, we store several `draw_batch`es. Each batch should be called with a separate `VertexRecorder::draw()` call[2] Before the draw call, make sure to update the material uniforms to the material properties specified by the `draw_batch`. You can use the `updateMaterialUniforms()` function to update the material of the current program.

The pseudo code for drawing the scene is

```
drawScene(GLint program, Matrix4f V, Matrix4f P)
  M = identity()
  updateTransformationUniforms(program, M, V, P)

  FOREACH draw_batch b:
    FOREACH index in indices[b.start_index : b.start_index + b.nindices]:
      recorder.record(position[index], normal[index], texcoord[index])
    updateMaterialUniforms(... b.mat ...)
    recorder.draw()
```

Beware: index-to-index: The `draw_batch` structure holds a `start_index` of the first vertex in the batch. This is not an index into the `position` arrays. It is an index into the indices array. To get the first vertex position of a draw batch, you call `scene.position[scene.indices[batch.start_index]]`.[3]

We again provide the `VertexRecorder` class. Use the three-argument record function

```
void VertexRecorder::record(Vector3f pos,
                            Vector3f normal,
                            Vector3f color);
```

and pass the UV coordinates in the first two elemements of the color attribute (`color.z` can be zero).

You should see a texture-less mesh with mostly lambertian reflectance as in Figure **??**.

Visualizing Texture Coordinates

Now, make sure that the texture coordinates are passed through to the fragment shader correctly. In CPU code we could just add print statements and see whether the texture coordinates are $\neq 0$. Unfortunately,

[2] **Aside:** We are still drawing in *immediate mode*, where each vertex is specified explicitly each frame. This is inefficient; a more performance-minded application would upload vertex data once at application startup, and then draw from the same vertex buffer object at each frame.

[3] Indirection and index-to-index logic like this often useful in realtime graphics. In the case of vertex indices, it allows us to store indices of multiple batches in a single array, which helps reduce fragmentation of graphics memory.

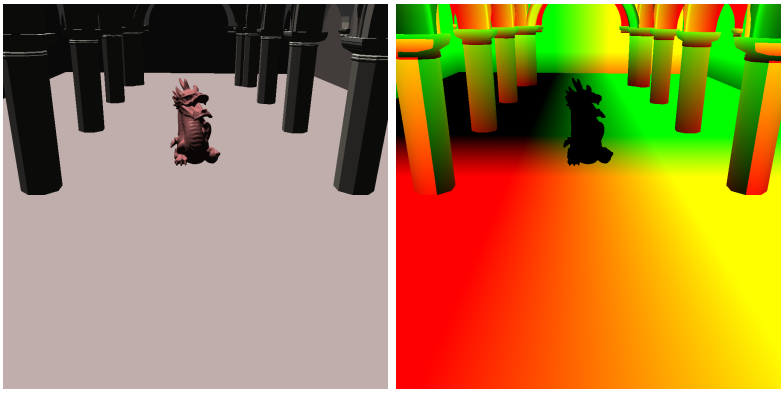


Figure 1: **Left:** Texture-less rendering. **Right:** Debug output of texture coordinates (sometimes called UV coordinates)

.

there are no print statements in OpenGL shaders. Thus, the preferred way of debugging shaders is by *drawing* debug information.

Modify the `fragmentshader_dirlight.glsl` file (you can modify the file while the application is running) so that it colors the mesh with the texture coordinates stored in `var_Color.xy`. The resulting image should look like the one in Figure **??** on the right.

4 Textures

In the previous item, you verified that texture coordinates are correctly passed to the fragment shader. Now you will create OpenGL texture objects, and read the diffuse material color from the texture, rather than a per-object uniform.

You should create two functions, `loadTextures()` and `freeTextures()`. Call `loadTextures()` from `main()` before the main loop begins, and `freeTextures()` after the main loop ends.

In `objparser`, we already loaded the JPEG files from disk. They are stored in a `std::map<string, rgbimage>` map. You can iterate through the map like this:

```
    for (auto it = scene.textures.begin(); it != scene.textures.end(); ++it) {
        std::string name = it->first;
        rgbimage& im = it->second;
        // Create OpenGL Texture
        // GLuint gltexture;
        // ...
        gltextures.insert(std::make_pair(name, gltexture));
}
```

You will create an OpenGL object for each texture, and insert the texture object into a global `std::map` object.

OpenGL Textures

Creating a texture in OpenGL requires several steps.

1. Generate a new *handle* using `glGenTextures(1, &gltexture)`
2. *Bind* the new handle to make it active `glBindTexture(GL_TEXTURE_2D, gltexture)`
3. Allocate storage for the texture currently bound as `GL_TEXTURE_2D` and upload pixel data using `glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, <width>, <height>, <border>, <format>, <type>, <data>)`. The OpenGL online reference explains all these parameters in detail. You almost always want to set `<border>` to 0. The pixel data in `rgbimage.data` is in `<format> = GL_RGB, <type> = GL_UNSIGNED_BYTE` format. The `<data>` parameter must be a pointer to an array of pixel values. `rgbimage` stores pixel data inside an `std::vector`. Use the `rgbimage.data.data()` function to get the raw pointer.
4. Configure the texture minification filter using `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)`. This enables basic bilinear filtering.

After these steps store the `GLint` texture handle in the `std::map` object for later use.

Texture memory can be a precious resource. When you no longer need a texture, it is good practice to delete it with a call to `glDeleteTextures(1, &gltexture)`. In `freeTextures()`, write a loop like the one in `loadTextures()` and call `glDeleteTextures()` on each texture handle.

Binding textures before drawing

Back in the `drawScene` function, you must make sure to bind the right texture before calling `recorder.draw()`. Just before you draw (1) look up the needed texture handle from the `std::map`, and (2) `glBindTexture()` it as `GL_TEXTURE_2D`.

Sampling from a texture in the fragment shader

We access textures in the fragment shader using a special type of uniform variable called *sampler*. It is defined with other uniform variables as

```
// in fragmentshader_dirlight.glsl
uniform sampler2D diffuseTex;
```

In the shader main function, use the GLSL `texture()` function to read the diffuse reflectance from the texture. The sampler uniform goes into the first argument, the 2D texture coordinates go second.

If everything went well, the scene will now draw as a textured mesh.

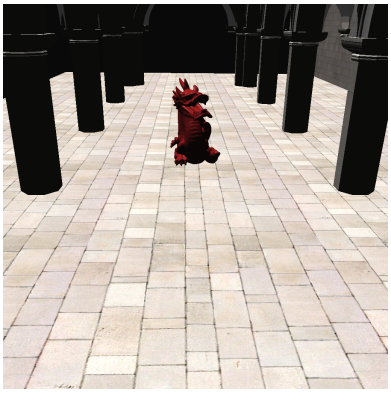


Figure 2: Drawing with textures.

5 Shadow Mapping

Shadow mapping is an example of a multi-pass algorithm in computer graphics: In the first pass, we render to a depth texture from the position of the light source. In the second pass, we use the depth value stored in this texture to determine light source visibility of scene points.

By default, OpenGL renders directly to the screen. With bit of additional configuration, we can make OpenGL render to a texture instead. To be precise, we will create two textures, one for color[4] and one for depth, and will combine both into a *framebuffer object*.

The shadow mapping algorithm has multiple steps, and if something goes wrong, it is often not obvious where the bug was introduced. To increase our chances of success, we will implement the first step of the algorithm (render to texture), and then visualize this intermediate result. Once we are sure step one is correct, we will move on to the actual shadow computations in step 2.

OpenGL Framebuffer Objects

You will create functions `loadFramebuffer()` and `freeFramebuffer()` that are called before the main loop enters, and after the main loop exits, respectively. Create three `GLint` OpenGL handles as global variables:

```
GLuint fb;          // framebuffer handle
GLuint fb_depthtex; // framebuffer depth texture handle
GLuint fb_colortex; // framebuffer color texture handle
```

Inside `loadFramebuffer()`, request a valid handle for each texture with calls to `glGenTextures()`. Bind the color texture to `GL_TEXTURE_2D`. Use `glTexImage2D` to allocate space for it. This time, you don't have to pass a data pointer, since the texture will be filled by rendering into it.

For the color texture, use `GL_RGBA8` as internal format, `width == height == 4096` as size, `GL_RGB` as format and `GL_UNSIGNED_BYTE` as data type. Pass `nullptr` as data pointer. [5]. After `glTexImage2D()`,

[4] The color texture isn't strictly necessary for the algorithm, but it's useful for debugging.

[5] That's a pretty large shadow map. Using a large shadow map helps reduce aliasing. As extra credit, you can implement more memory-efficient ways to reduce aliasing.

you need to configure the texture interpolation for this texture, so that it can be rendered into. Texture interpolation settings are set with glTexParameter() like this

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

`GL_NEAREST` means "nearest-neighbor interpolation"

Now, bind the depth texture to `GL_TEXTURE_2D`. Depth textures have a special internal format: `GL_DEPTH_COMPONENT32F`. As `<format>` parameter use `GL_DEPTH_COMPONENT` (without `32F`) and `GL_FLOAT` as data type. Make sure to set the minification mode to `GL_NEAREST` for the depth texture.

Next, request a handle for the framebuffer using `glGenFramebuffers()`. Like textures, the current framebuffer object must be *bound* before you can configure it further. Once bound, you will specify the two textures as the data stores for color and depth buffer respectively:

```
glGenFramebuffers(1, &fb);
glBindFramebuffer(GL_FRAMEBUFFER, fb);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, fb_colortex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, fb_depthtex, 0);
```

Since the code we wrote in this section is quite intricate, we double-check that our framebuffer is configured correctly:

```
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if (status != GL_FRAMEBUFFER_COMPLETE) {
    printf("ERROR, incomplete framebuffer\n");
    exit(-1);
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

In the last line, we set the active framebuffer back to the default framebuffer 0.

freeFramebuffer: Use `glDeleteTextures()` and `glDeleteFramebuffers` to release the memory that was allocated in **loadFramebuffer()**.

Light View and Light Projection matrices

Implement two functions `Matrix4f getLightView()` and `Matrix4f getLightProjection()`.

The light projection matrix is constant throughout our program (although optimized implementations of shadow mapping change the projection matrix dynamically, see extra credit). You can use `Matrix4f::orthographicProjection()` to get an orthographic projection matrix. The parameters of the orthographic projection (x-scale, y-scale, near-clip plae, far-clip plane) must be chosen based on the scene geometry and the expected view frustum of the virtual camera.

The light view matrix is set to look along the light direction `light_dir` onto the scene. The distance from the scene is arbitrary, but must be coordinated with the setting for near and far clip plane of the projection matrix.

It is convenient to use `Matrix4f::lookAt(eye, center, up)`.

- `center` should be the center of the sponze scene (i.e. position of the dragon statue)
- `eye` should be above the scene (remember, the unit-Y vector points up). The line of sight from `eye` to `center` should be parallel to `light_dir`
- up must be orthogonal to light dir.

Adding a depth render pass.

When we implemented `drawScene()`, we made sure that the function can be called with a different GLSL program and difference camera matrices. This comes in handy now, as we can call the function once for the depth pass, and then a second time for the *light pass*.

In `draw()`, first bind the shadow frame buffer, clear it, and make sure the OpenGL viewport is set to the correct state.

```
glBindFramebuffer(GL_FRAMEBUFFER, fb);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glUseProgram(program_color);
```

In the depth pass, we don't really care about the color output of our program. Thus, it is best to use a very simple shader, for example the shader that just assigns a fixed color to each vertex.

All that's left is to call `drawScene()` and pass it the active GLSL program, the `LightView` matrix, and the `LightProjection` matrix. After this, the scene will have been rendered into the shadow texture.

So far the depth texture is still invisible; we can draw it to the screen using the `drawTexturedQuad(GLint texture)` helper function: After the light pass, with the current framebuffer set to 0 (drawing to the screen), call the following to draw the depth texture to the lower-left corner of the screen.

```
// set viewport to lower-left corner of screen
glViewport(0, 0, 256, 256);
drawTexturedQuad(fb_depthtex).
```

Now, set the viewport to another region of the screen and call `drawTexturedQuad()` again, this time passing it the color buffer of the depth render pass. The color buffer is not used later on, but it's good to double check its content.

In case the field of view of the orthographic projection is too wide or too narrow, tune the parameters of the orthographic projection until the scene just fills the depth texture. Also, you should tune the near clip plane and far clip plane of the orthographic projection, so that the scene's depth values are well distributed in the 0 to 1 range.

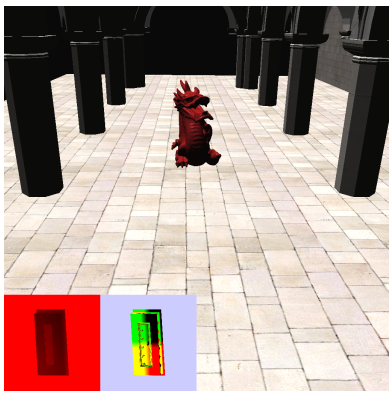


Figure 3: An intermediate step for shadow mapping: We have rendered the scene from two viewpoints. In the lower-left, we see scene as seen from the light source. Make sure that the near-clip and far-clip plane of the orthographic projection fit tightly, so that the depth texture is used in the full range from 0 to 1.

Using multiple textures

Until now, we only use a single texture. For shadow mapping, we will need to use two textures simultaneously: One texture for the diffuse color, and another for the depth texture. In OpenGL, we can achieve this by using multiple *texture units*.

The currently active texture unit is selected with the `glActiveTexture()`. By default, have been using texture unit 0 with the diffuse texture. For shadow mapping, after setting up the diffuse texture in `drawScene`, we switch to texture unit 1, bind the depth texture, and then switch back to texture unit 0 to make sure we don't affect other code that expects unit 0 (the default) to be active.

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, fb_depthtex);
glActiveTexture(GL_TEXTURE0);
```

In `fragmentshader_dirlight.glsl`, add another sampler2D uniform named `shadowTex`. What's left is to "bind the sampler to a texture unit". To do this, update the sampler object like any other uniform, passing the texture unit number as an integer:

```
int loc = glGetUniformLocation(program, "shadowTex");
glUniform1i(loc, 1); // bind sample to texture unit 1
```

Samplers are by default bound to texture unit 0, so there is no need to configure anything for the diffuse texture.

Shadow Mapping

There is one last step to do in the C++ code: Before drawing in `drawScene()`, look up the location of an additional 4x4 matrix uniform named `light_VP` and pass the *Light View Projection* matrix. Make sure you get the order of view and projection right! That's it for the C++ part of the assignment, you will implement the remaining steps in the fragment shader.

In `fragmentshader_dirlight.glsl` define the corresponding `uniform mat4 light_VP`. Now, use the `Position` varying, the `light_VP` matrix, and the depth values stored in the red channel of the `shadowTex`ture to implement shadows as discribed in class.

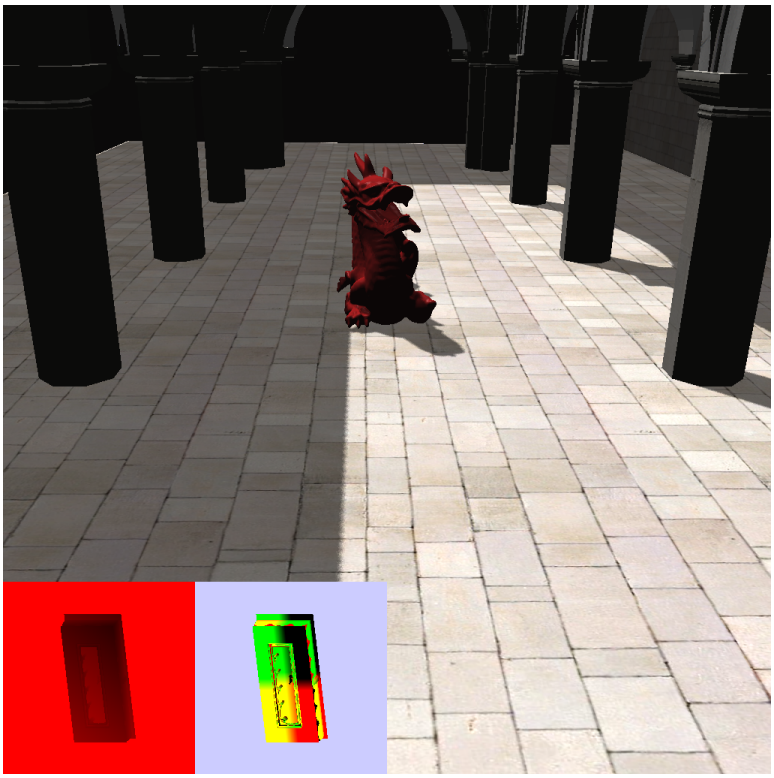


Figure 4: The final result of this assignment (screenshot includes PCF extra credit).

6 Extra Credit

Easy

- Implement Percentage-Closer Filtering[6]. You can do so with a few lines of code in the fragment shader.
- Dynamically compute a tightly fitting orthographic projection matrix for the shadow camera
- We left all texture interpolation settings to their default values. As an extension, add bilinear texture filtering with mip-mapping. Can you get OpenGL to do anisotropic filtering for you?
- Add some form of Anti-Aliasing, e.g. SSAA[7] or MSAA[8].
- Add a spot light, also with shadow.

Medium

- Implement Cascaded Shadow Maps[9]
- Render the scene with Ambient Occlusion. There is a variety of *screen-space* ambient occlusion algorithms out there, such as Alchemy[10].

Hard

- Implement stencil shadow volumes
- Use a modified version of the ray tracer from the previous assignment to render out light maps (textures that store global illumination for static illumination and diffuse objects).

Submit your assignment on Stellar. Please submit a single archive (`.zip` or `.tar.gz`) containing:

- Your source code and instructions on how to build it on Athena.
- A compiled executable named `a5`.
- Any additional files (not already in the starter archive) that are necessary.
- The `README` file.

[6] http://graphics.pixar.com/library/ShadowMaps/paper.pdf
[7] https://en.wikipedia.org/wiki/Supersampling
[8] https://en.wikipedia.org/wiki/Multisample_anti-aliasing
[9] http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf
[10] http://graphics.cs.williams.edu/papers/AlchemyHPG11/VV11AlchemyAO.pdf