CPSC 314
Assignment 4: Textures and Shadows

Due 11:59PM, Nov 22, 2017

1 Introduction

In this assignment, you will learn how to use texture mapping. You will implement a skybox and a brick floor. The floor terrain will use basic texture mapping (for color), normal and ambient occlusion mapping (for detailed bumps), and shadow mapping.

1.1 Template

In the assignment, you are provided with an armadillo on a plane and template shader files for your terrain and skybox. We have defined lighting uniforms for you, as well as texture images you will need for your terrain and skybox.

2 Work to be done (100 pts)

1. **Part 1** (70 pts)

   (a) **Basic Texturing** (15 pts)

   A modeled object often has many very small details and facets of the surface (e.g. the grain of wood on a box, scratches on metal, freckles on skin). These are very difficult if not impossible to model as a single material lit by a Phong-like model. In order to efficiently simulate these materials we usually resort to texturing. In basic texturing, UV coordinates are taken from the vertex buffer (three.js provides them in the `vec2 uv` attribute on default geometries such as planes and spheres). UV coordinates are a 2D index into an image file whose values can be used to color the mesh.

   You are provided with `images/color.jpg` and `images/ambient_occlusion.jpg` that you have to apply to the terrain. Textures in GLSL are specified as `sampler2D` uniforms, and the values can be looked up using the `texture2D` function.

   In our context, you can think of the color map as the diffuse reflectance of the surface while the ambient occlusion maps is used to model the visibility of the indirect lighting, represented by the ambient term.

(b) **Bump mapping** (25 pts)

In computer graphics, we often want to give the illusion of detail with as simple geometry as possible. Although our terrain is a simple plane, we can perturb surface normals with a texture and use standard lighting methods to simulate the effect of bumps along the surface. This is called bump mapping, and the most common way of doing this is with a **normal map** that directly stores modified normals on each point in the surface.

We've provided you with a normal map in `images/normal.png`, which you can map to normals in your terrain shaders (RGB colours store XYZ coordinates of the normal respectively). Then, use standard Phong lighting to light the terrain, where your color information (from part a) should be multiplied into the diffuse term. The sampled normal is already provided for you in `N_1` as a small adjustement is required. The normals provided are defined with respect to the tangent frame on the point. You need to make sure that all calculations are done in the same frame. For example, you may transform your **view** and **light** vector to the same space as the new normal in order for the operations to be meaningful.

(c) **Skybox** (10 pts)

A skybox is a simple way to create backgrounds for your scene with textures. We've provided six textures in `images/cubemap/cube_.png`. You will implement cube mapping, which is a skybox where you map these textures on to a large cube surrounding the scene. The six textures are already loaded into `skyboxCubemap`, you can pass this as a `samplerCube` uniform to your skybox shader. Sampling is done using `textureCube()` function; this requires a `vec3` as texture coordinate input, corresponding to the viewing direction for the specific fragment. In order not to break the immersion, as the character moves, the skybox **should not** get closer or farther away and should **always** be in the background.

(d) **Shadow mapping** (20 pts)

Shadows are a tricky part of computer graphics, since it's not easy to figure out which parts of a scene should be in shadow. There are many techniques to create shadows (raytracing, shadow volumes, etc.) but we will be implementing shadow mapping for this assignment[1].

Shadow mapping is all about exploiting the z-buffer. A shadow map is rendered in an off screen frame buffer by projecting the scene from the perspective of a light source, giving us the depth at each fragment along the lookAt direction of the light source. When we render the scene, we check the current fragment's depth (in the clip coordinates of the light) against the depth we stored in the shadow map. If the current fragment is further from the light than the shadow map, then there must be an object closer to the light, which means we should this fragment is in the shadow!

We've done the hard part for you by computing the shadow map in the first pass: that's this line: `renderer.render(depthScene, lightCamera, shadowMap)`.

[1] See http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

Now it's up to you to add the shader code that will convert the current fragment to shadow coordinates (clip coordinates of the light source), check against the shadowMap with the `getShadowMapDepth` function provided, and make that fragment darker if it's in the shadow.

Hint: When comparing fragment depth with the shadow map depth, it will be useful to add a small margin of error to prevent artifacts.

Figure 1: Completed Part 1 Scene

2. **Part 2** (30 pts) Creative License.

   This is it. The final creative license section. Go all out and show us what you've learned!

   Here are some suggestions:

   - Experiment with other graphics techniques, such as anti-aliasing, cascading shadow maps, procedural methods, raytracing.
   - Make an interactive video game.

- Make a short film/animation and tell a tear-jerking story with sounds.
- Experiment with WebVR. We've included some WebVR code if you're interested in seeing your scene in VR (you can use anything from a smartphone to a VR headset).

Bonus marks may be given at the discretion of the marker for particularly noteworthy explorations.

2.1 Hand-in Instructions

You do not have to hand in any printed code. Create a README.txt file that includes your name, student number, and login ID, and any information you would like to pass on to the marker. Create a folder called "a4" under your "cs314" directory. Within this directory have two subdirectories named "part1," and "part2", and put all the source files, your makefile, and your README.txt file for each part in the respective folder. Do not use further sub-directories. The assignment should be handed in with the exact command:

```
handin cs314 a4
```