CPSC 314
Assignment 4: Textures and Shadows

Due 11:59PM, Mar 20 2019

Figure 1: Completed Part 1 Scene

1 Introduction

In this assignment, you will be learning all about texture. You will implement a skybox and a brick floor, as well as make your favourite armadillo reflective or cast a shadow. The floor terrain will use basic texture mapping (for color), normal and ambient occlusion mapping (for detailed bumps), and shadow mapping.

1.1 Getting the Code

Assignment code is hosted on UBC STASH BitBucket, to retrieve it onto your local machine first ensure that you have logged in at least once onto BitBucket:

`https://stash.ugrad.cs.ubc.ca:8443`.

Then navigate to the folder on your machine where you intend to keep your assignment code, and run the following command from the terminal or command line:

```
git clone https://stash.ugrad.cs.ubc.ca:8443/scm/cpsc314-2018wt2_students/a4.git
```

1.2 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** (10 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 RGB 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 `texture()` 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** (20 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 axes 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 `Nt` as a small adjustement is required (0..1 range) to use it directly as normal. The normals provided are in **tangent space**, that is defined with respect to the tangent frame on the point. You need to make sure that all the calculations are done in the correct space. As shading is usually done in eye coordinates, you will need to transform your **view** and **light** vector to the same space as the new normal in order for the operations to be meaningful.

Hint: Depending on how you setup the transformation to tangent space you might require a transpose(), right-multiplying is equivalent.

(c) **Skybox & Reflection** (20 pts)

A skybox is a simple way of creating backgrounds to your scene with textures. We've provided six textures in `images/pos_.png & images/neg_.png`. You will implement cube mapping, which is a skybox where you map these textures on to a large cube surrounding the scene. You need to load the six textures to `skyboxCubemap` with some order, you can pass this as a `samplerCube` uniform to your skybox shader. Sampling is done using the overloaded `texture()` function which requires a `vec3` in this case as texture coordinate input, corresponding to the viewing direction for the specific fragment.

In order to not break the immersion, as the character moves, the skybox **should not** get closer or farther away and **should always** be in the background. Note that while the material and shaders are already loaded, it's your task to create the correct geometry and add it to the scene as you did in the previous assignments.

Reflection can be achieved by **environment mapping**. Complete `envmap.vs.glsl` and `envmap.fs.glsl` that implement a basic reflective environment map shader. This is used to shade the second Armadillo and the sphere (originally colored white). You can use the same cube texture `skyboxCubemap` in your shaders which is passed as a `samplerCube` uniform like before, as well as the same `texture()` function, but pay attention to use the correct `vec3` for retrieving the texture color.

(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 cast 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.

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

When rendering an object, we want to check whether the very same vertex position is occluded by some other object when looked at from the light point of view. This is done by comparing the depth value as if projected by the light camera with the one stored in the shadowmap, which is the scene's depthbuffer viewed from the light camera. 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 this fragment is in the shadow!

You can use the `DepthTexture` and `WebGLRenderTarget` provided by three.js to set up the off-screen scene. And use this line: `renderer.render(depthScene, shadowMap_Camera, shadowMap)` to render it. Most of this has been setup for you, please set up the `shadowMap_Camera`'s Orthographic projection parameters and its `lookAt` matrix in the `A4.js` code.

Think about it: Why are we using orthographic projection? Feel free to try out other perspectives.

After getting the shadow map, you need to add the shader code that will convert the current fragment to shadow coordinates (clip coordinates of the light source), check against the shadowMap 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.

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! We've also included WebVR code if you're interested in seeing your scene in VR (you can use anything from a smartphone to a VR headset). To test your scene on your smartphone, use python to serve your site locally with `python -m SimpleHTTPServer 8000`, allow public connections from your firewall, and then access your computer's `IP:8000` on your phone!

   Here are some suggestions:

   - Experiment with multi-pass rendering to add static 2D toolbox to your 3D scene.
   - Experiment with other graphics techniques, like procedural mapping (eg. Perlin noise), subsurface scattering ("Guassion blur"), ambient occlusion, raytracing
   - Make a short film/animation and tell a tear-jerking story with sounds.
   - Make an interactive video game.

   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, login ID, creative components 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
```