CS5625 Programming Assignment 2 (PA2): Advanced Texturing

Due: Program - Feb 28 (Thursdays), 11:59 pm.
Work in groups of 2.

Instructor: Kavita Bala

Overview

In this assignment you will use texturing techniques we learned in class to implement new effects and add visual realism to your scenes. These techniques will be useful in your final project when you are creating your interactive game engine.

1 Copy Over PA1 Code

Find all the "PA1 TODO" sections and copy over your solutions from PA1. Note that you cannot simply copy over the entire files as we have made major changes to some parts of the framework that you will overwrite. In particular, there was a small bug in the compressed vector decode method that only became apparent on our dynamic cube map.

2 Normal/Bump Mapping

Implement two different normal modifying shaders. Both shaders should produce Blinn-Phong material information. In both shaders, you should remap the given normals using "normal $=$ normalFromTexture $-$ vec3(0.5)", which will bring back the negative components in the normals.

- Classic normal mapping (world space). We have supplied a normal map texture that encodes the correct world space normals for a plane perpendicular to the z-axis. Add code in the classic normal map shaders to correctly send the necessary information to the ubershader.
- Normal/Bump mapping. In this method, assume that the normal map represents the normal in the object's tangent space. Add code in the normal mapping shaders to do what is needed including transforming the normals from the normal map into the correct space.

3 Static Cube Maps for Reflections

You will add support for environment lighting, using cube maps. We ask you to implement a new material **ReflectionMaterial**, that simulates a perfect mirror. Similar to the previous assignment, you will have to do a small amount of work in the Java file **ReflectionMaterial.java**, such as setting uniforms, but most of the work will be inside the GLSL shaders. You must fill in the function

Figure 1: Final scene (two different view points), showing (1) the static cube map shader (the small sphere); (2) reflection material with dynamic cube map (the big sphere); (3) classical normal mapping (the floor); (4) normal mapping/bump mapping (the cube); (5) Cook-Torrance with static cube map (the monkey).

shadeReflective in the ubershader to compute the reflection direction and the corresponding texture coordinates, then pass this into sampleCubeMap along with the cube map index. This index tells the ubershader which cube map to sample from.

- 0 = no cube map
- 1 = StaticCubeMapTexture
- 2 = DynamicCubeMapTexture0
- 3 = DynamicCubeMapTexture1
- 4 = DynamicCubeMapTexture2

We support one static cube map and up to three dynamic cube maps. There is no particular reason for this limitation, and you can decide to support more cube maps for your final projects.

Notice that the reflected vector that we use to index the cube maps should be given in world space, thus, you will have to transform vectors from object space into world space. You can do that by using the inverse of the camera rotation matrix, given as a uniform in the ubershader (CameraInverseRotation). You also have to supply this matrix to the ubershader, which is done from within **Renderer.java**.

3.1 Cook-Torrance with environment lighting

Next, we ask you to extend the **Cook-Torrance** lighting model by adding contribution from an environment map. You must implement the requirements of the function **mixEnvMapWithBaseColor**, which should return a linear mix between a base color (B, from the regular shading equation for the surface) and a reflected color (R, from the environment map), based on the Fresnel coefficient (F): $B * (1 - F) + R * F$. The Fresnel coefficient gives you what part of the energy gets refracted vs. reflected. In this function, you should compute the Fresnel factor based on the cosine of the angle between the fragment normal and the view vector, and you can use Schlick's approximation to do that.

4 Dynamic Cube Maps for Reflections

You will extend your rendering code to support dynamic cube maps. First, you will render a cube map of the surroundings of one object. You will have to correctly position the camera to render each cube face, as well as tell the FBO to render the final scene to the cube map texture.

Once you have the texture created, combining it with the static cube mapping shader from above will give the effect of dynamic cube maps for reflections.

You must perform this each frame so that reflected objects move dynamically. Notice that the current implementation would require 6 full passes for every dynamic cube map in the scene, which can be very expensive for larger scenes. However, as we want to support OpenGL 2.0 in our framework, this is the only option for dynamic cube mapping. In later versions of the GL specification, you can use the geometry shader to do this more efficiently. The main idea is to reduce the CPU/GPU communication by letting the GPU do the rendering of the 6 cube map faces with 1 CPU call. Furthermore, in production systems, people would implement various optimizations, such as not re-generating the dynamic cube maps at every frame, and rendering dynamic cube maps in lower resolution. In our framework, we use dynamic cube maps with a fixed resolution of 6x512x512 pixels.

4.1 Glossy reflections using cube maps

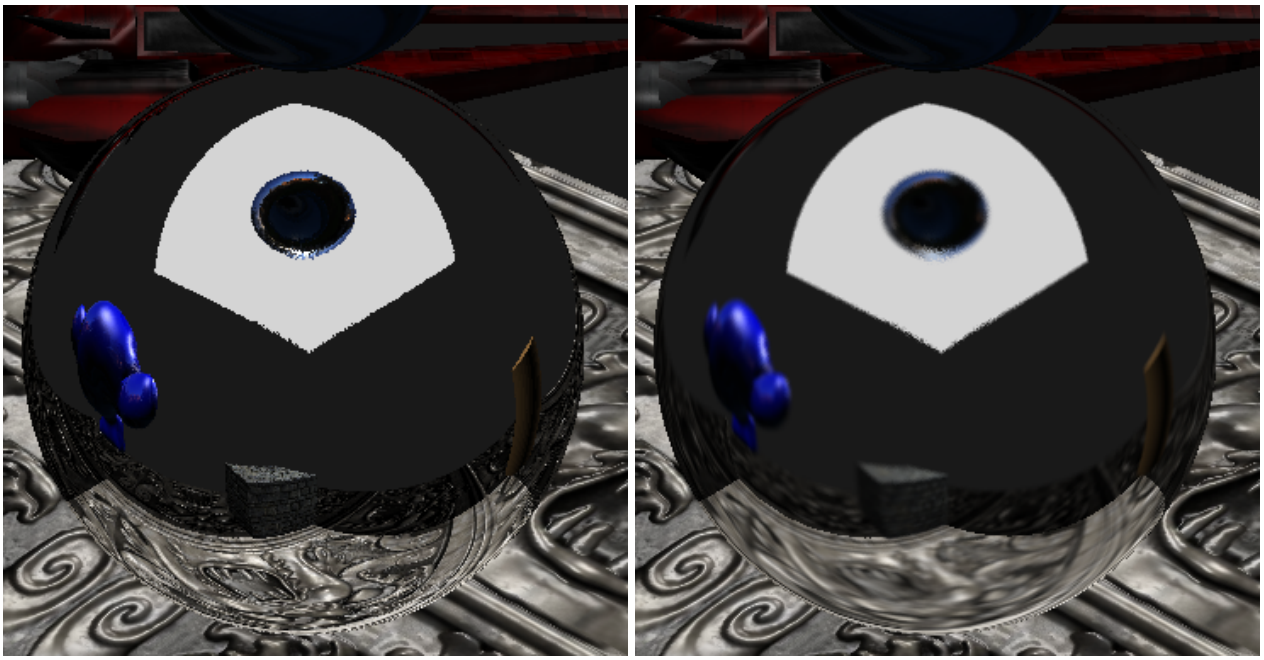

Figure 2: The big sphere is a perfect mirror (aka. reflection material) with dynamic cube mapping. On the left we show the result with no blurring applied, and on the right the result after dynamic cube map Gaussian blur has been enabled (by pressing 'g').

After you have cube mapping working, write a program to apply different Gaussian blurs to the

cube map; each blur width changes the perceived glossiness of the object.

We use the fact that the 2D Gaussian blur kernel can be separated into two 1D Gaussian kernels: horizontal and vertical. This leads to a more efficient implementation of the 2D Gaussian, especially for kernels with large widths. Your job is to fill the fragment shader, **gaussian_blur.fp**, so that it performs 1D Gaussian blur (horizontal or vertical) on a given (square) texture with a given width and variance.

Note that there are five uniform parameters given to this shader. You must use all of them as indicated below:

- $SourceTexture$: the input texture image, you should be reading pixel values from this texture.
- $Axis$: 0 indicates that you should compute a 1D Gaussian in the horizontal direction, otherwise you should compute it in the vertical direction.
- $TextureSize$: the size of the texture dimension. This will be the width of the texture, when the shader is called for horizontal blur, and the height of the texture, when called for vertical blur.
- $KernelVariance$: the variance of the 2D Gaussian function.
- $KernelWidth$: the half-width of the 2D Gaussian function. You should run your 1D Gaussian filter over a sequence of $2 * KernelWidth + 1$ pixels centered on the target fragment.

Use the following 1D weighting function: $weight(d) = exp(-d^2/(2.0 * KernelVariance))$, where d is the distance (in pixels) between the given pixel and its queried neighbor. Hint: you can let GLSL take care of texture coordinates that are outside the valid range for squared textures, [0, 1].

You can use the 'g' key on your keyboard to toggle the blur option for all dynamic cube maps in the scene.

5 Bloom Postprocess Shading

Figure 3: The scene with bloom turned off (left) and on (right). The effect might be difficult to see on a laptop screen. Hit 'b' to toggle bloom. You can control the Gaussian variance with 'v' and 'V', the Gaussian width with 'x' and 'X', and the brightness threshold with 'c' and 'C'.

Bloom post-processing fakes the bright blur that occurs around the edges of opaque objects when they are very brightly lit from behind. Many games use this technique to make outdoor scenes look more realistic.

In this part you will implement a basic single-pass bloom shader in **bloom.fp**. It must do the following:

- Threshold the brightness of each pixel (a brightness is a single float).
- Apply a Gaussian blur to the thresholded brightness values. Since we do bloom in a single pass you must use a 2D weight function: $weight(x, y) = exp(-(x^2 + y^2)/(2.0 * KernelVariance))$. Note that x and y are measured in pixels relative to the center pixel $(0, 0)$.
- Output the sum of the original color and the blurred brightness.

Note that there are three uniform parameters given to this shader. You must use all of them as indicated below:

- $KernelVariance$: the variance of the 2D Gaussian function.
- $KernelWidth$: the half-width of the 2D Gaussian function. You should run your 2D Gaussian filter over a square of $2 * KernelWidth + 1$ by $2 * KernelWidth + 1$ pixels centered on the target fragment.
- $Threshold$: below this value all brightnesses should be clamped to 0, otherwise they should be clamped to 1.

Hint: remember that **FinalSceneBuffer** is a rectangular texture that maps directly to each of our output fragments. This means that it should be very easy to look up neighboring pixels using **texture2DRect** and **gl_FragCoord.xy**. See the OpenGL docs for more details.

To use bloom, hit 'b' on your keyboard. See Figure 3 for a rendering of the default scene with bloom enabled.

6 Silhouette Detection

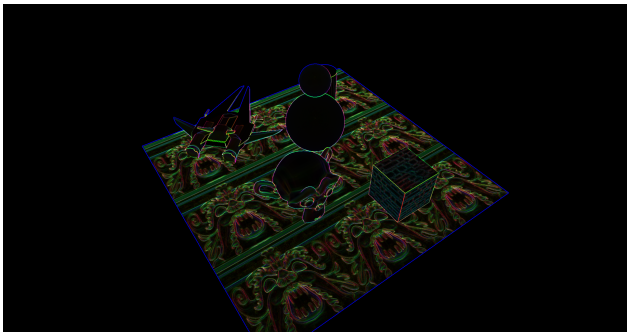

Figure 4: The silhouette buffer; you can view it by hitting '5' on your keyboard.

Complete the **silhouette** shader to do edge and crease detection as described in the first paragraph of Section 2.3.1 of this paper:

Decaudin, Philippe, "Cartoon-Looking Rendering of 3D-Scenes," Technical Report INRIA 2919, Université de Technologie de Compiègne, France, June 1996.

You can find a copy of the paper on his website: `http://www.antisphere.com/Research/Publis/RR-2919-en.pdf`

As far as your work in the **silhouette** shader is concerned, you must simply implement the contour filter g described on the 6th page of the paper for our normals and depth values. You should store the gradient of the normal in the RGB channels, and the gradient of the depth in the alpha channel.

In the next section we will use these values to implement toon shading. Since we are using eye-space depth (in the range [-zNear, -zFar]) instead of depth-buffer depth (in the range [0, 1]), the value of our k_p coefficient will be much larger than that described in the paper.

7 Toon Shading

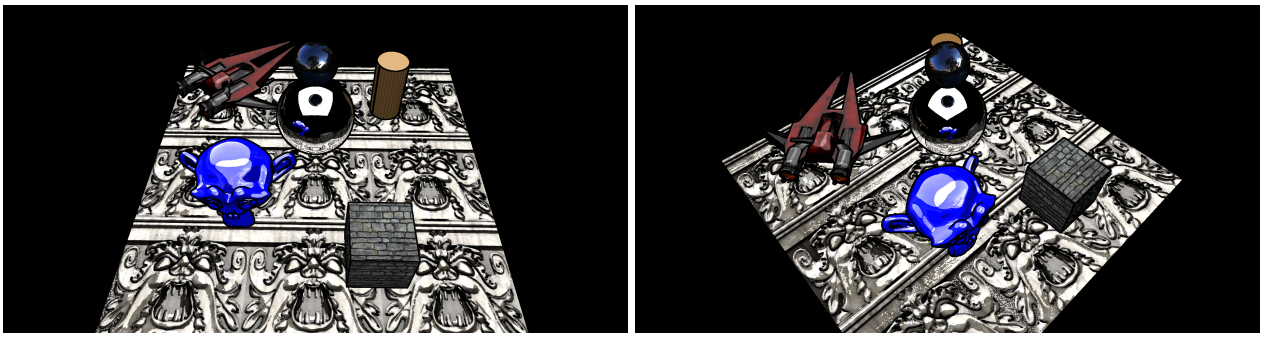

Figure 5: Final scene (two different view points), showing the toon shader effect. The 't' key toggles the toon shader.

Using the silhouette buffer from the previous section, toon shading can be implemented quite easily.

Threshold the lighting coefficients in each of the ubershader's lighting shading functions to achieve the classic cartoon cel-shaded appearance if the uniform **EnableToonShading** is true. The $\vec{n} \cdot \vec{l}$ coefficient should be thresholded at 0.1, i.e. replaced with 0.0 if it's less than 0.1 and 1.0 if it's greater than or equal to 0.1. The $\vec{n} \cdot \vec{h}$ specular coefficient should be thresholded at 0.9 using the same technique. GLSL has a `step()` function which provides this thresholding functionality.

To get the thick black outlines to render you must fill out the ubershader's **silhouetteStrength** function. This requires you to compute the local silhouette strength p using the formula from the top of page 7 of [Decaudin96]. You should find g_{min} and g_{max}, the minimum and maximum values in the **SilhouetteBuffer**, within a 3x3 region. Make sure you find the min and max for each channel in the **SilhouetteBuffer**. Then apply the formula for p as seen in the paper with one modification: take the sum of the squared terms across all four channels before you clamp the silhouette strength to one or greater. Note that we use 0.3 for k_p.

8 What To Submit

You should submit your completed deferred shading framework, along with a readme file describing any implementation choices you made or difficulties you encountered. If you make any fancy new scenes, feel free to include those (and screenshots) as well.