CS5625 Programming Assignment 1: Shading Models

Due: February 13 (Wednesday), 11:59 pm.
Work in groups of 2.

Instructor: Kavita Bala

Overview

PA1 is built on a framework designed for interactive applications. You are tasked with completing bits of Java and GLSL code throughout the framework, linking together the mostly-finished program we have provided.

You must complete portions of the Lambertian, Blinn-Phong, Cook-Torrance, Isotropic and Anisotropic Ward Java classes and GLSL shaders. Since lighting is deferred to the ubershader, you will have to complete portions of "ubershader.fp" as well.

You must also implement a data visualization shader which displays the normals, tangents, and bitangents hidden away in the g-buffer. These display modes should be helpful in debugging your materials, particularly the Ward model.

1 Overview of Deferred Shading

In 4620, all real-time assignments used a rendering technique known as "forward" shading, where the lighting and shading of each fragment is computed immediately once the fragment is rasterized from the geometry. This is the obvious thing to do, but it has two main drawbacks:

1. If the scene has many overlapping objects, expensive lighting calculations are performed for all fragments, even those that will be overwritten by an object closer to the viewer.
2. Since the scene is rendered directly to the screen, the application never has an image of the scene in an easy-to-manipulate form for applying post-processing effects.

Deferred shading addresses both of these shortcomings of forward shading:

1. Instead of lighting each fragment as it is generated, the scene is first rendered into an off-screen buffer (the "g-buffer") using simple shaders which just output material properties. Since no lighting or other computation has been done yet, overlapping objects are handled efficiently.
2. Since the g-buffer already contains an image-space representation of the scene, post-processing is natural and easy in this situation.
3. Run an "ubershader" on the g-buffer to compute lighting. This shader is usually called an "ubershader' ("supershader") since it contains lighting code for all types of lights and materials.

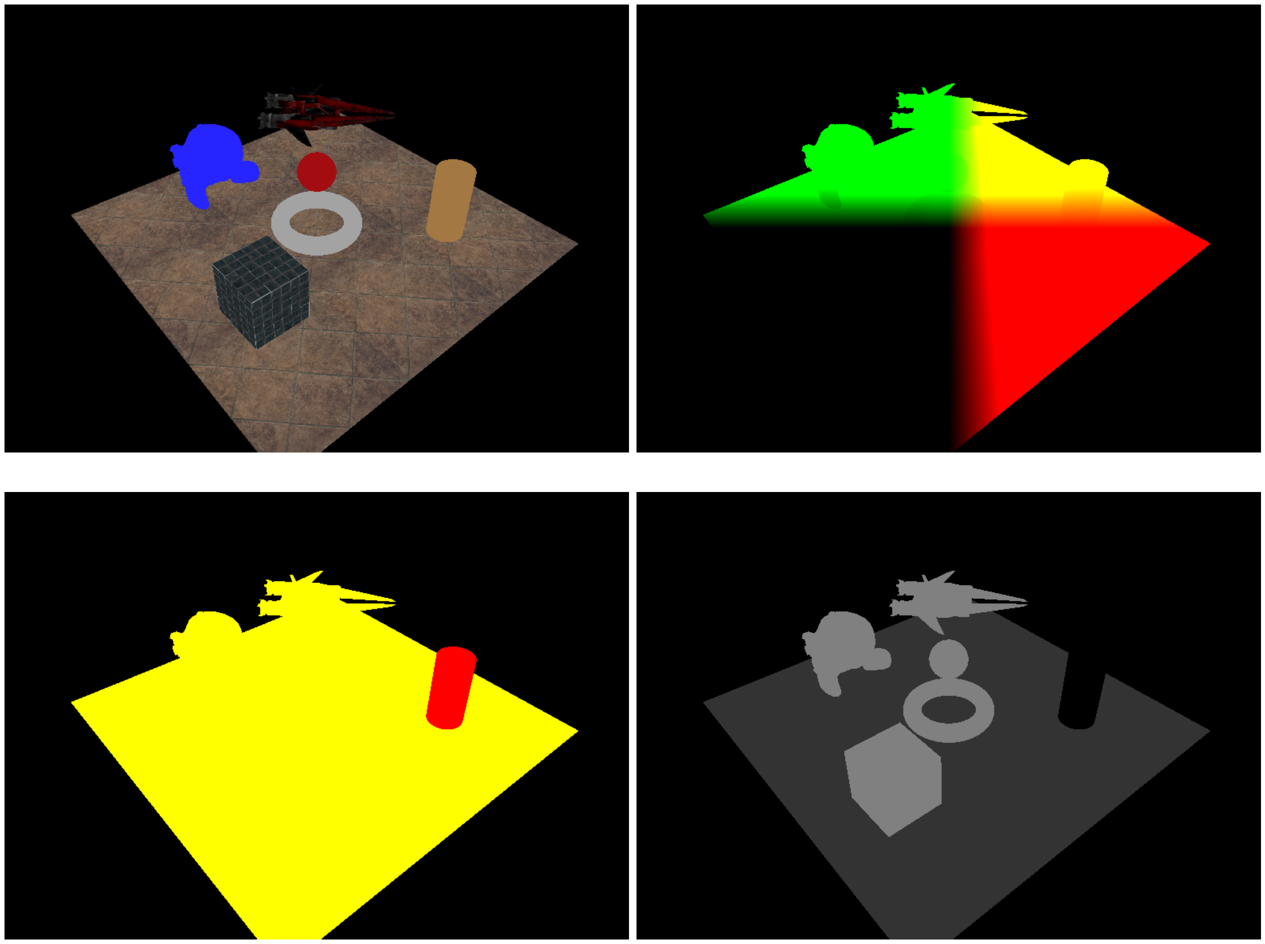

Figure 1: The g-buffer of the deferred renderer. From top-left: diffuse buffer, position buffer, the first material buffer, and the second material buffer. Depending on how you implement your shaders you may get different results for your material buffers.

Our deferred shader uses a g-buffer composed of six textures. The first four textures are the targets of the first rendering pass and are used as inputs by the ubershader (among other things). The remaining two textures in the g-buffer are the silhouette buffer and the final scene buffer, each of which are targets of their own rendering passes. In this assignment the silhouette buffer is unused; you will use it to implement toon shading in PA2.

The first four textures are (in texture index order) the diffuse buffer, the position buffer, and two buffers for material properties. The diffuse buffer and position buffer store the diffuse color and eye-space position in their R, G, and B components; the alpha components of these textures are the x and y values of the compressed eye-space normal (more on this later). The material property textures contain material specific data and their organization is up to you to decide with one exception: the R component of the first material buffer must be the material ID. When organizing the content of the two material g-buffers, you should think of them as arrays, where you can save material specific parameters, which will be needed for the second pass, inside **ubershader.fp**, that will do the actual per-pixel shading. For example, looking at the equation for the Cook-Torrance model (lecture 2), in order to compute the specular term for every pixel you will have to save: (1) the specular color (3 float values), (2) the parameter m (1 float), that controls the apparent smoothness of the surface, and (3) the parameter n (1 float), that defines the index of refraction at this pixel. Notice that in this case you will not be using all the memory in the material g-buffers, which is perfectly fine. However, for the Anisotropic Ward model, you will need to save more information, such as the (compressed) tangent vector and the sign of the tangent-bitangnet-normal basis, which will utilize all the memory in the two material buffers.

Whenever possible, you must compress any normalized vectors before they are added to the g-buffer. This way we can pack more information into the g-buffer. We provide "encode" and "decode" functions in our shader files to compress and expand normalized vectors for you.

2 Deferred Shading Assignment

All areas which need your attention are marked with "TODO" comments. Here is a complete list:

2.1 Material shaders

Complete the **LambertianMaterial**, **BlinnPhongMaterial**, **CookTorranceMaterial**, **IsotropicWardMaterial**, and **AnisotropicWardMaterial** Java classes and their GLSL shaders to output the required fragment properties to the g-buffer.

Before you begin, take a look at the **UnshadedMaterial** class and its shaders "material_unshaded.vp" and "material_unshaded.fp". Make sure you understand how this material encodes its normal and how it stores its data into the g-buffer. See the previous section for more information about the contents of the g-buffer.

All of these materials can be assigned textures in place of or in addition to their normal uniform parameters. When binding a material's Java class, you should set the "Has____Texture" uniform for each texture in the shader based on whether or not the Java texture is null. If the texture in question is not null you should go on to bind it to the appropriate sampler and unbind it in the unbind method.

As a general rule, if a shader's "Has____Texture" uniform is set to true it should use the value from that texture instead of the corresponding uniform value. There are two exceptions to this rule: diffuse and specular texture values should be multiplied by their uniform values as this allows us to tint a color texture without create a brand new texture to achieve this relatively simple effect.

Start with the Lambertian material as it is the most simple, and only supports texturing for the diffuse color. All other shaders will require you to pick some way to store material data into the g-buffer's third and fourth textures. You can organize this in any way you think is appropriate as long as you compress all normalized vectors (normals and tangents). There is an additional trick for anisotropic Ward that is mentioned below.

Here are a few more shader hints. Your Blinn-Phong shader and Cook-Torrance shader should multiply the exponent texture value and 'n' texture value by 255 to get a more reasonable value.

Before the Anisotropic Ward material will work properly, you must complete the **computeAndAccumulateTangentVectors** and **averageAndNormalizeAllTangentVectors** methods in the **Mesh** class. The method we use was outlined in the 4th lecture.

Your anisotropic Ward shader won't have enough room to store the encoded bitangent vector. Instead you should compute the cross product of the normal and the tangent. Then take the dot product of this vector with the bitangent to determine the sign (-1 or +1) of this dot product with respect to the bitangent. Later when you need to reconstruct the bitangent using the normal and tangent, you will need this sign. Since the entire g-buffer will be full, you can store the sign of the dot product by multiplying it against the material ID float.

You can view the contents of your g-buffer using the keyboard keys '1' through '6'. Note that the 5th g-buffer texture is the unused silhouette buffer and will always be black in this assignment.

2.2 Ubershader

Complete the **ubershader** shader to support your new materials.

Since you could write down the Lambertian and Blinn-Phong lighting equations in your sleep by now, we have provided functions which evaluate them for a single light. The corresponding functions for Cook-Torrance and both flavors of Ward are left for you to implement.

You also have to write uber-shader code which calls the right lighting function for the fragment in question (extracting data from the g-buffer), and handles multiple lights.

The user is free to resize the window, so we have no guarantee that its size will be a power of two. Because of this, the g-buffer FBO textures are created as "rectangular" textures. This type of texture uses a different texture target from Java (`GL_TEXTURE_RECTANGLE`), and a different sampler type and sampling function in GLSL (`sampler2DRect` and `texture2DRect()`).

When using rectangular textures, texture coordinates are pixel coordinates, not normalized [0, 1] coordinates. This plays to our advantage here, though: since the ubershader needs to access the pixels "under" the current fragment in the g-buffer, we can use the current fragment position (`gl_FragCoord.xy`) to sample the g-buffer textures. This code is already written in **ubershader.fp**, but you should understand why it works.

The top left image of Figure 2 shows the expected output for the default scene.

2.3 Visualization

Complete the "visualize.fp" shader so that it produces a useful visualization of the normals, tangents, and bitangents stored in the g-buffer. This shader is invoked by the '7', '8', and '9' keyboard keys, where '7' displays normals, '8' displays tangents, and '9' displays bitangents.

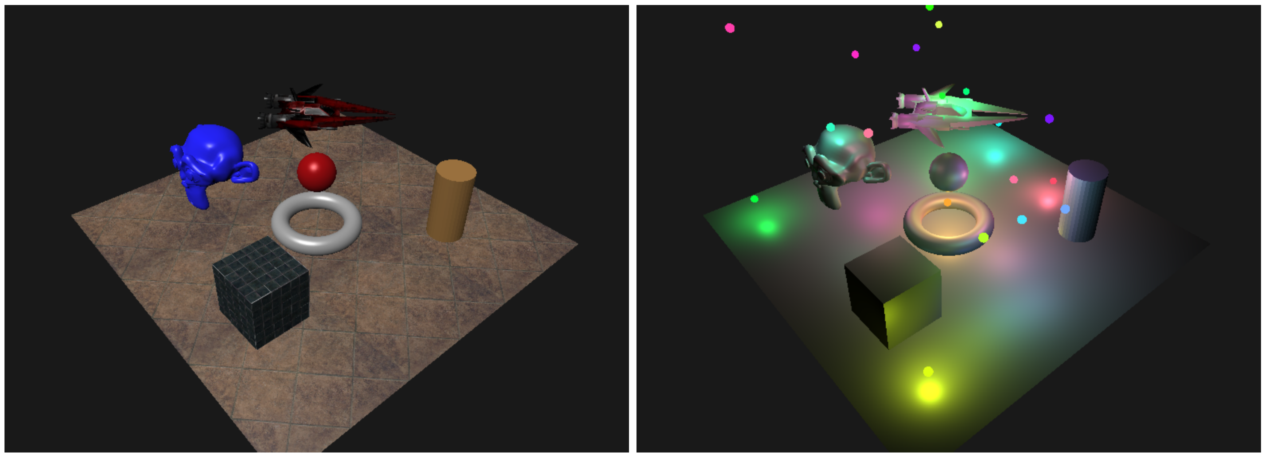

Figure 2: Final scenes rendered with the deferred renderer. Left: the scene corresponding to the g-buffers in Figure 1. Right: a similar scene with white materials and many small colored lights.

What exactly the visualization looks like is up to you. Just make sure it visualizes all possible values that the inputs can take and presents something that is useful to the user.

3 Provided Example Scenes

We provide three scenes to test your code. You can choose which scene to use by editing **SceneController.main()**.

The **DefaultSceneController** loads the contents of "default-scene.obj" and adds a single light. You can spin the camera by dragging the mouse, and zoom with the scroll wheel.

The **ManyLightsSceneController** loads the contents of "default-scene.obj" with only white materials and creates a randomized cloud of point lights. You can spin the camera by dragging left mouse, spin the light cloud by dragging right mouse, and zoom with the scroll wheel.

The **MaterialTestSceneController** loads a number of objects and assigns them materials that test all of the features you will implement. You can spin the camera by dragging the mouse, and zoom with the scroll wheel.

All scenes respect the following keyboard control scheme. The number keys 1-6 preview the corresponding g-buffer textures from the deferred renderer. The number keys 7-9 display visualizations of mesh normals, tangents, and bitangents. Note that tangents and bitangents are only used by Ward materials. The 0 key turns off the preview, displaying the final scene. The 'w' key turns on wireframe rendering.

4 Hardware Requirements

This framework might not run on older (or netbook) hardware. Known requirements, and the error to expect if your hardware or drivers don't meet the requirements, are:

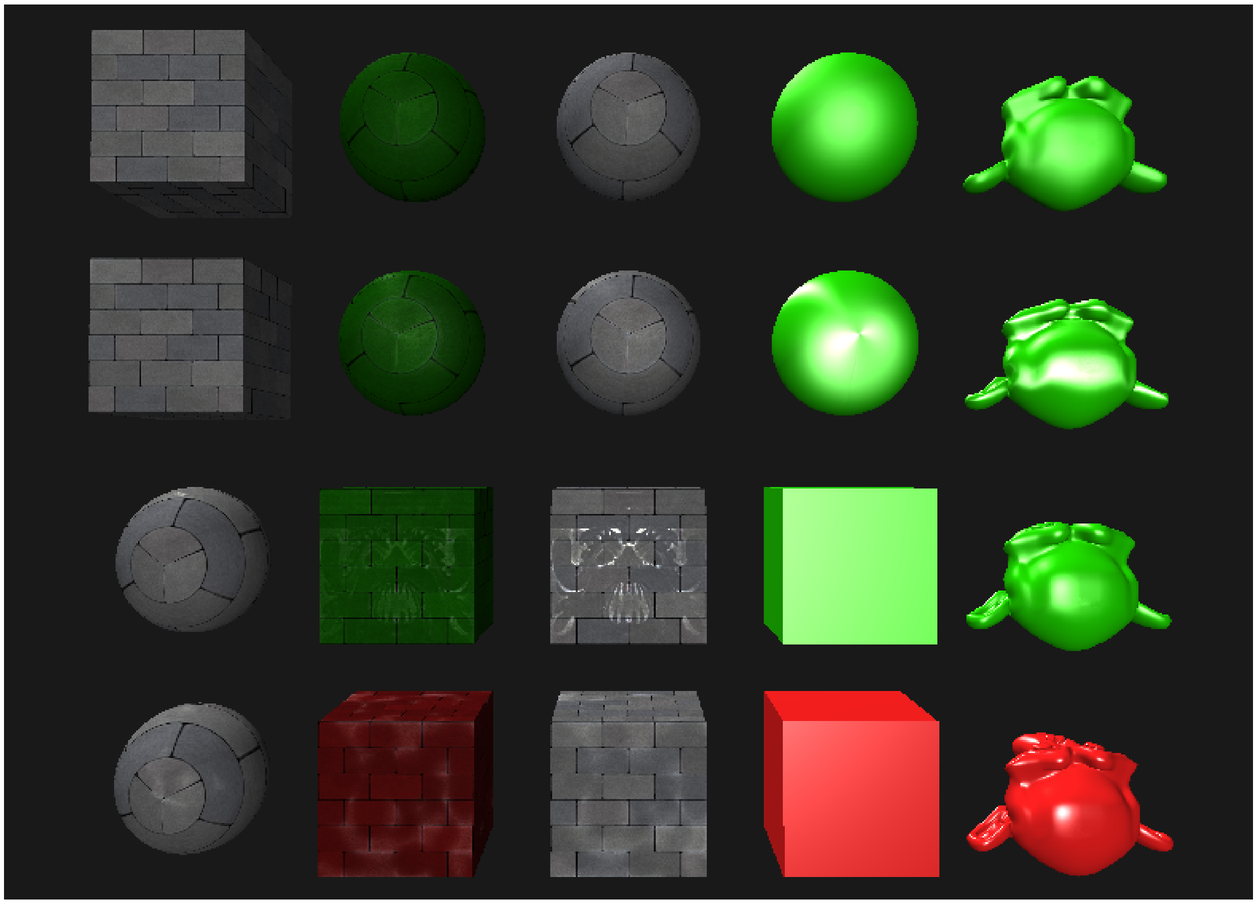

Figure 3: The MaterialTestSceneController rendered with our solution code.

1. Must support at least 4 color attachments on an FBO. The `FramebufferObject` class will throw an exception if this isn't the case.
2. Must support at least 5 texture targets which can be sampled from a shader. The `Texture` class will throw an exception if this isn't the case.
3. Must support dynamic branches and loops in fragment shaders. On the hardware tested which didn't support this, it rendered black instead of the correct output from the ubershader, and didn't generate any errors. If you see this problem and you absolutely can't use a newer graphics card, you can try temporarily replacing the `NumLights` uniform in the ubershader with a constant. Remember to change it back (and test that it works) before submitting.

Some of the CSUG lab computers meet these requirements:

1. **Optiplex 760**: does not support dynamic looping. You'd have to use the `NumLights` workaround described above, or just use one of the other machines!
2. **Optiplex 980**: everything works.
3. **Fancy new white computers**: everything works.

5 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.