CPI 411 Graphics for Games

Lab 3 (First 3D Model)

This lab is to introduce you to the basic lighting algorithms and the way to implement the lighting models in HLSL. We are going to display a 3D object and implement mouse-controls to change the views to check the shading effects from different angles.

A. Importing FBX file into the MonoGame pipeline

In using MonoGame API, we have to compile all resource files through the MonoGame's Pipeline tool. The FBX file is the common format of 3D object now. However, there are several versions of FBX format, and they have ASCII and Binary types. MonoGame prefers the binary type of FBX 2011. When the FBX file uses textures, another issue may happen. The FBX file stores the full path to the image file directory, and the MonoGame tool fails if there is no image file in the stored folder. The following is the recommended way to import FBX files.

1. Add the FBX file to the Content folder (Copy the file into it)
2. Add the other image files used in the FBX file
3. In this condition, the FBX file is pointing to the original image files, so we have to change the file path to the copied ones.
4. Open the FBX file in 3ds max, and make a new material using the image files located in the copied folder. Then, assign the material to the FBX file.
5. Export /Override the FBX file as the same file name.

The old XNA supported the directX format (.x), but the MonoGame does not. If you use the .x files, convert them to the FBX files using the online tool. (Google the .X to .FBX file).

Let's import the bunny.fbx file. It does not have any texture. Once it is added to the Visual Studio, you can check the data in the Visual Studio 3D Viewer.

B. First 3D Object

In the precious exercises, we studied 3D graphics pipeline with a single triangle. Now it is time to change the program to support regular 3D object. In XNA/MonoGame framework, a 3D object is treated as an instance of Model. The first step is to make the global variable and load the data from the content files.

```
Model model;

-- LoadContent() method
model = Content.Load<Model>("name of model");
```

The last thing is to update the Draw method. The Model class has a method to draw the object with basic built-in shader. Same as the pipeline we studied, it requires three matrices, world, view and projection.

```
-- Draw() method
model.Draw(world, view, projection);
```

However, this is not our goal. We have to use the custom shader/effect to implement lighting models. In order to use the custom shader, we ask directly to the graphics device. Last time, we used the `GraphicsDevice.DrawUserPrimitives<VertexPositionColor/Texture>(...),` because we did not have to use the indices for polygons for a single triangle. This time the data has a list of vertex and a list of index for each triangle, so the `GraphicsDevice.`**`DrawIndexedPrimitives`** `(...)` method is used instead.

In XNA/MonoGame, each model has a list of `ModelMeshes`, named `Meshes`. Also each mesh has a list of `MeshPart(s)`, named `MeshParts`. We have to access to each MeshPart to get the vertex and index data for a 3D object. Therefore, the code should be like below. Usually one FBX file has several 3D objects. A mesh is a 3D object among them. On the other hand, the part is a part of mesh using a same material. When an object has multiple materials, it has multiple parts.

```
foreach (ModelMesh mesh in model.Meshes){
   foreach (ModelMeshPart part in mesh.MeshParts) {

         GraphicsDevice.SetVertexBuffer(part.VertexBuffer);
         GraphicsDevice.Indices = part.IndexBuffer;

         GraphicsDevice.DrawIndexedPrimitives(
            PrimitiveType.TriangleList, part.VertexOffset, 0,
            part.NumVertices, part.StartIndex, part.PrimitiveCount);

    }
}
```

The first two lines in the block are to set the vertex and index data as a buffer. The last statement is to draw the data. Look at the details in the MS document. Try to run the program. It will not show anything because the shader has not been applied to the graphics device. Add the following block out of the rendering block above. These statements are to choose one shader pass from the effect and apply to the graphics device. The next section explains the first lighting model in HLSL.

```
effect.CurrentTechnique = effect.Techniques[0];
foreach (EffectPass pass in effect.CurrentTechnique.Passes){

      foreach (ModelMesh mesh in model.Meshes) {
          ...
            pass.Apply();
            // set buffers and draw mesh model
      }
}
```

C. First Lighting Model

Let's start the basic lighting model. The per-vertex lighting is to calculate the color for each vertex in the vertex shader, and the other fragments are interpolated (linear) in the pixel shader. Look at the Chapter 5 of Cg Tutorial.

In the vertex shader, we calculate the ambient, diffuse, and specular colors in addition to the position for screen space. Let's declare several variables for the colors.

```
float4x4  World;
```

```
float4x4  View;
float4x4  Projection;
float4x4  WorldInverseTranspose;
float4    AmbientColor;
float     AmbientIntensity;
float3    DiffuseLightDirection;
float4    DiffuseColor;
float     DiffuseIntensity;
```

The world inverse transpose is a matrix to use in calculating the normal vector. The other additional variables are for the colors.

The lighting model uses normal vectors of input vertex data to add shading effects so that new data structures are required to support the input data. The following are the minimum requirement.

```
struct VertexInput{
      float4 Position: POSITION;
      float4 Normal: NORMAL;
};

struct VertexOutput{
      float4 Position: POSITION;
      float4 Color: COLOR;
};
```

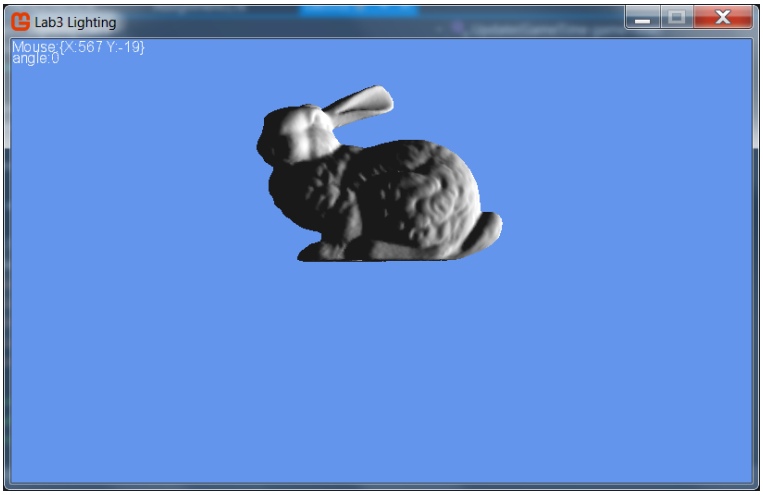
As explained above, the per-vertex lighting is to calculate the shaded color. The following example is to calculate the diffuse color in the vertex shader, and the ambient color is added in the pixel shader. You can modify these in any way.

```
-- in the vertex shader
VertexOutput output;
float4 worldPos = mul(input.Position, World);
float4 viewPos = mul(worldPos, View);
output.Position = mul(viewPos, Projection);

float4 normal = mul(input.Normal, WorldInverseTranspose);
float lightIntensity = dot(normal, DiffuseLightDirection);
output.Color = saturate(DiffuseColor * DiffuseIntensity *
                                                    lightIntensity);
return output;


-- in the fragment shader
return saturate(input.Color + AmbientColor * AmbientIntensity);
```

Once the shader is created, add statements to pass the data from MonoGame side to Shader side. There are two important points here. One is to set the World matrix for each mesh. When you save a 3D object in 3ds max, the position (x, y, z) of each vertex is not in the world coordinate but the local coordinate. In order to allocate the vertices in the proper position, they should apply the transform matrix, which is also exported in the FBX file. The Mesh object has a parent bone with a transform matrix in MonoGame.
The second is more complicated. The normal vector may not be calculated when some scaling is applied. (Look at the lecture slide) To fix the normal vector direction, the inverse transpose matrix should be

applied to the current/imported data. The last two statements are to calculate the matrix and send it to the shader side.

```
-- in MonoGame program
effect.Parameters["World"].SetValue(mesh.ParentBone.Transform);
effect.Parameters["View"].SetValue(view);
effect.Parameters["Projection"].SetValue(projection);
effect.Parameters["AmbientColor"].SetValue(ambient);
effect.Parameters["AmbientIntensity"].SetValue(ambientIntensity);
effect.Parameters["DiffuseColor"].SetValue(diffuseColor);
....
Matrix worldInverseTranspose =
            Matrix.Transpose(Matrix.Invert(mesh.ParentBone.Transform));
effect.Parameters["WorldInverseTranspose"].SetValue(worldInverseTranspose);
```



You can see the object in the screen. The exercise is based on http://rbwhitaker.wikidot.com/diffuse-lighting-shader

D. Mouse Controls to change the view (Main Exercise)

Before starting today's main exercise, look at the Assignment#1. You are asked to implement several lighting models and user interfaces there. An example of key controls was introduced in the last exercise. Today we develop a mouse event to rotate the camera. The goal is to rotate the camera position in two directions during mouse dragging. To check the left mouse status, the following condition is used in MonoGame.

```
if (Mouse.GetState().LeftButton == ButtonState.Pressed)
```

First, test it by adding the angle value incrementally (`angle += 0.01f;`) similar to the last Lab. Once it works, go to the next step.

In order to implement mouse-drag controls, keep in mind that the offset of mouse position between the last Update() and current Update() is used. Therefore, we need to keep the mouse position in the previous update.
Create a global variable of `MouseState` to keep the last mouse condition. The MouseState has X and Y properties, so we can access to the last position as `previousMouseState.X (or Y)`. Since this previousMouseState is not a built-in variable, so it does not update automatically. Don't forget to update

at the end of Update() method.

```
MouseState previousMouseState;

-- Update method
previousMouseState = Mouse.GetState();
```

Only when both previous and current mouse status are pressed (this is equivalent to "dragging"), calculate the offset (x, y) between the status. The x-offset is used to update the `angle` of camera position, and the y-offset is used to update the `angle2` of camera position. Once these angles are updated, update the view matrix as following.

```
Vector3 camera = Vector3.Transform(
           new Vector3(0, 0, 20),
          Matrix.CreateRotationX(angle2)* Matrix.CreateRotationY(angle)
          );
view = Matrix.CreateLookAt(camera, Vector3.Zero, Vector3.UnitY);
```

*) Options: add two more mouse controls, which is for Assignment#1.

- Change the zoom (cameraPosition.Z) by using Mouse Right drags.
- Translate the camera (cameraPosition.X and Y) by using Mouse Middle drags.

***) Debugging:**
In 3D programming, it is much harder to find and kill the bugs rather than regular programming. The best way is to display all variables on the screen to check the value change visually. Use the DrawString(font, string, position, color) methods between the Begin and End statements in the Draw() method. The following is an example to show the angle value on the screen.

```
-- Draw() method
spriteBatch.Begin();
spriteBatch.DrawString(font, "angle:" + angle, Vector2.UnitX +
Vector2.UnitY*12, Color.White);
spriteBatch.End();
```

***** IMPORTANT *****
Complete the exercise in D section, and submit a zipped file including the solution (.sln) file and the project folders to course online site. The submission item is located in the "**Quiz and Lab**" section. Each lab has **10 points**. If you complete the exercise in class time, the full points will be assigned. The late submission is accepted just before the next class with 2 points reductions, because the solution is demonstrated in the next class.