CMSC 23700 **Introduction to Computer Graphics** **Project 3**
Autumn 2015 **October 20, 2015**

Advanced Lighting Techniques

Due: Monday November 2 at 10pm

1 Introduction

This assignment is the third and final part of the first project. In this part, you will add both support for richer modeling and for more advanced rendering modes. This assignment builds on your Project 2 code by adding the following features to your viewer:

- Orientation for objects — objects in the scene description now come with a frame that defines their orientation in world space.
- Heightfields — these are a compact way of representing terrain meshes (and which figure prominently in the final project).
- Normal mapping — this is a technique for creating the illusion of an uneven surface by perturbing the surface normals.
- Shadow mapping — this is a technique for adding shadows to the scene.

You should start by fixing any issues with your Project 2 code.

For extra credit, you can implement a rendering mode that supports both normal mapping and shadowing.

2 Description

2.1 Scenes

We extend the scene format to support new features. The extensions include adding a shadow factor, a ground objects, and object frames to the `scene.json` file, and adding normal maps to the object materials. We will provide a enhanced version of the `Scene` class that handles the new features. As before, your code will use the scene object to initialize your view state and object representations.

The lighting information contains an extra field, `shadow`, that specifies the darkness of the shadows. This value is called the shadow factor; see Section 2.5 for details.

Ground objects are represented as height fields (described below in Section 2.3). Each scene has at most one ground object, which will be specified as a JSON object with the following fields:

```
{
  "lighting" : {
      "direction" : { "x" : 0, "y" : -1, "z" : 0},
      "intensity" : { "r" : 0.8, "b" : 0.8, "g" : 0.8},
      "ambient" : { "r" : 0.2, "b" : 0.2, "g" : 0.2},
      "shadow" : 0.25
    },
  "camera" : {
      "size" : { "wid" : 1024, "ht" : 768 },
      "fov" : 120,
      "pos" : { "x" : 0, "y" : 3, "z" : -8 },
      "look-at" : { "x" : 0, "y" : 3, "z" : 0 },
      "up" : { "x" : 0, "y" : 1, "z" : 0 }
    },
  "ground" : {
      "size" : { "wid" : 50, "ht" : 50 },
      "v-scale" : 0.1,
      "height-field" : "ground-hf.png",
      "color" : { "r" : 0.5, "b" : 0.5, "g" : 0.5 },
      "color-map" : "ground-cmap.png",
      "normal-map" : "ground-nmap.png"
    },
  "objects" : [
      { "file" : "box.obj",
        "color" : { "r" : 0, "b" : 1, "g" : 0 },
        "pos" : { "x" : 0, "y" : 0, "z" : 0 },
        "frame" : {
            "x-axis" : { "x" : 1, "y" : 0, "z" : -1},
            "y-axis" : { "x" : 0, "y" : 1, "z" : 0},
            "z-axis" : { "x" : 1, "y" : 0, "z" : 1}
          }
      }
    ]
}
```

Figure 1: An example `scene.json` file

1. The `size` field specifies the width and height of the ground object in world-space units. By convention, the ground object is centered on the origin.

2. The `v-scale` field specifies the vertical scale in world-space units.

3. The `height-field` field specifies the name of a gray-scale PNG file, which defines the offsets from the XZ plane.

4. The `color` field specifies the color used to render the ground object in wireframe, flat-shading, and diffuse modes.

5. The `color-map` field specifies a texture image used to render the ground object in texturing and normal-mapping modes.

6. The `normal-map` field specifies a texture image used to render the ground object in normal-mapping model. Note that this file is **not** the definition of the vertex normals for the ground

mesh.

The other change to the format of the `scene.json` file is the addition of object frames. A frame is a JSON object with three fields `x-axis`, `y-axis`, and `z-axis`, that represent the object space's basis in world coordinates.

2.2 Object frames

Associated with each object instance in the scene is a frame that defines how the X, Y, and Z axes are oriented and scaled in world space. For example, the frame $\langle 1, 0, 0\rangle$, $\langle 0, 1, 0\rangle$, $\langle 0, 0, 1\rangle$ specifies the identity transformation, since the object-space axes coincide with the world-space axes. If we wanted to rotate an object by $90°$ around the Y axis (counter-clockwise) and double its height, we would specify the frame $\langle 0, 0, -1\rangle$, $\langle 0, 2, 0\rangle$, $\langle 1, 0, 0\rangle$. Taking the frame vectors as the columns of a 3×3 matrix and combining it with the object's position, gives an affine transformation for mapping points in object space to world space. Note that the transformation is not restricted to be isotropic.

2.3 Heightfields

Heightfields are a special case of mesh data structure, in which only one number, the height, is stored per vertex (height field data is often stored as a greyscale image). The other two coordinates are implicit in the grid position. If the world-space size of the ground object is $wid \times ht$ and the height field image size is $w \times h$, then the horizontal scaling factors in the X ansd Z dimensions are given by

$$s_x = \frac{wid}{w-1} \qquad \text{and} \qquad s_z = \frac{ht}{h-1}$$

If s_v is the vertical scale and $\mathbf{H}$ is the height field, then the 3D coordinate of the vertex in row i and column j is $\mathbf{h}_{i,j} = \langle x_0 + s_x j, s_v \mathbf{H}[i, j], z_0 + s_z i\rangle$, where the upper-left corner of the height field has X and Z coordinates of x_0 and z_0 (respectively). By convention, the top of the height field is north; thus, assuming a right-handed coordinate system, the positive X-axis points east, the positive Y axis points up, and the positive Z-axis points south.

Because of their regular structure, height fields are trivial to triangulate; for example, Figure 2 gives two possible triangulations of a 5×5 grid. The triangulation on the left lends itself to rendering as triangle fans, while the triangulation on the right can be rendered as a triangle strip. Rendering with these primitives reduces the number of vertices that are submitted. For example, rendering the left triangulation using fans of eight triangles each will reduce the number of vertices by about 45%[1] But the space savings will be much less when using `glElements`, since the indices only account for about 17% of the VAO data. Therefore, we recommend rendering the ground using `GL_TRIANGLES`.

Unlike an object specified in an OBJ file, a height field does not come with normal vectors, so you will need to compute these as part of your initialization. There are a couple of different ways to compute these normals, but most approaches involve averaging the sum of the normals for surrounding faces or edges.

[1]Each fan of eight triangles will take 10 indices plus a restart index (see the documentation for `glPrimitiveRestartIndex`).

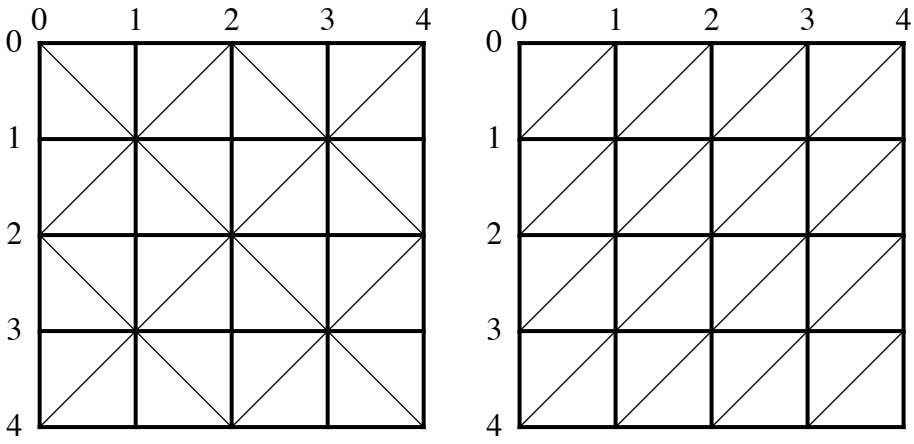
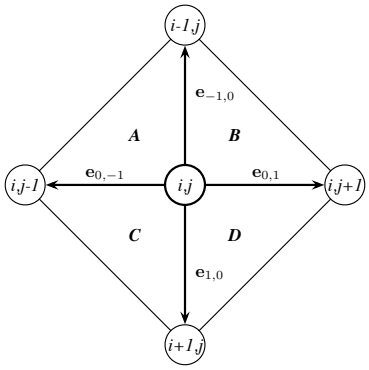


Figure 2: Possible height field triangulations

For example, consider the vertex $\mathbf{h}_{i,j}$. It has four neighboring right triangles (A, B, C, and D) for which it is the apex as illustrated in the following diagram:[2]



To compute the vertex normal $\mathbf{n}i, j$ for $\mathbf{h}_{i,j}$, we start by computing the four edge vectors radiating out from $\mathbf{h}_{i,j}$:

$$\begin{aligned}
\mathbf{e}_{-1,0} &= \mathbf{h}_{i-1,j} - \mathbf{h}_{i,j} \\
\mathbf{e}_{0,-1} &= \mathbf{h}_{i,j-1} - \mathbf{h}_{i,j} \\
\mathbf{e}_{0,1} &= \mathbf{h}_{i,j+1} - \mathbf{h}_{i,j} \\
\mathbf{e}_{1,0} &= \mathbf{h}_{i+1,j} - \mathbf{h}_{i,j}
\end{aligned}$$

[2]Note that these triangles are not the ones in the triangulation.

Then we can compute the face normals of the triangles

$$\begin{aligned}
\mathbf{n}_A &= \frac{\mathbf{e}_{0,-1} \times \mathbf{e}_{-1,0}}{\|\mathbf{e}_{0,-1} \times \mathbf{e}_{-1,0}\|} \\
\mathbf{n}_B &= \frac{\mathbf{e}_{0,1} \times \mathbf{e}_{-1,0}}{\|\mathbf{e}_{0,1} \times \mathbf{e}_{-1,0}\|} \\
\mathbf{n}_C &= \frac{\mathbf{e}_{0,-1} \times \mathbf{e}_{1,0}}{\|\mathbf{e}_{0,-1} \times \mathbf{e}_{1,0}\|} \\
\mathbf{n}_D &= \frac{\mathbf{e}_{1,0} \times \mathbf{e}_{0,1}}{\|\mathbf{e}_{1,0} \times \mathbf{e}_{0,1}\|}
\end{aligned}$$

Then the vertex normal is just the average of the triangle normals:

$$\mathbf{n}_{i,j} = \frac{1}{4}\left(\mathbf{n}_A + \mathbf{n}_B + \mathbf{n}_C + \mathbf{n}_D\right)$$

The remaining detail is to remember that for vertices at the corners and edges, we have fewer adjacent faces.

2.4 Normal mapping

Normal mapping is a technique to simulate the lighting effects of small-scale geometry without directly modeling the geometry.[3] A *normal map* is a three-channel texture that holds normal vectors in a surface's tangent space. To use the normal map, we map the light vector to the tangent space and then dot it with the normal vector at the fragment location. Figure 3 illustrates this process.

2.4.1 Normal maps

We use a three-channel texture to represent the normal vectors on the surface. These normals are stored in the tangent space of the surface, so to compute the illumination of a fragment, one has to map the light vector(s) and view vector to the surface's tangent space.[4] Specifically, the red and green channels hold the X and Y components of the normal vector, which should coincide with the S and T axes of the texture. The blue channel holds the Z component of the normal. Note that the normal vectors stored in the normal texture are not unit vectors.

2.4.2 Tangent space

For each vertex $\mathbf{p}_0$ in the mesh, we want to compute the tangent space at that vertex. Assume that we have a triangle $\langle \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2 \rangle$ with texture coordinates $\langle s_0, t_0 \rangle$, $\langle s_1, t_1 \rangle$, and $\langle s_2, t_2 \rangle$ (respectively), and that $\mathbf{n}$ is the vertex normal at $\mathbf{p}_0$. As mentioned above, we need to map the light vector into the *tangent space* of the triangle. The tangent space will be defined by three vectors: the tangent vector $\mathbf{t}$, the bi-tangent vector $\mathbf{b}$, and the normal vector $\mathbf{n}$. These vectors (taken as columns) define the mapping from tangent space into object space. When implementing a feature like normal mapping, it is important to keep the coordinate systems straight. The following picture helps:

[3] Normal mapping is often called *bump mapping*, but they are different techniques.

[4] Note that since we are only implementing diffuse and ambient lighting in this project, the view vector is not required.

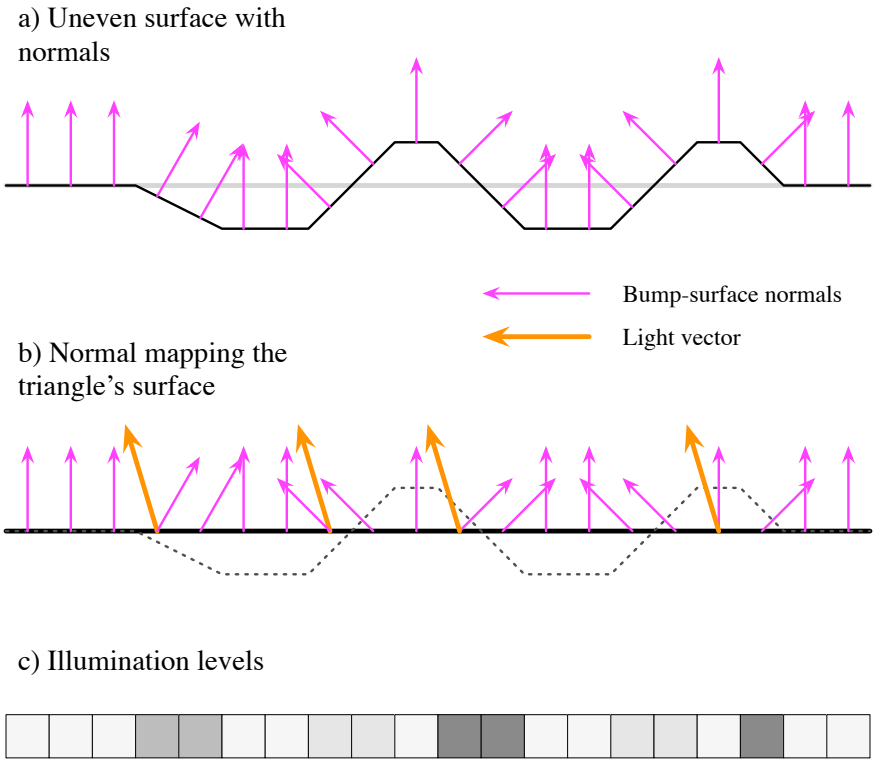
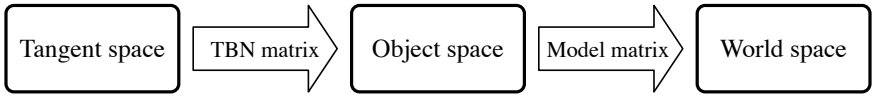


Figure 3: Lighting a surface with a normal map. Part (a) shows an uneven surface and its surface normals; (b) shows the resulting vectors in the triangle's tangent space, and (c) shows the resulting illumination levels.

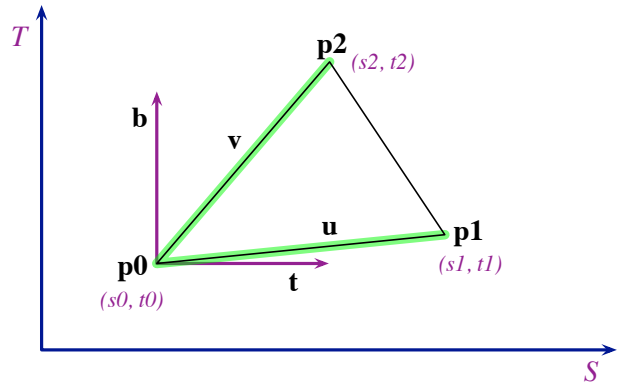


Lights are usually specified in world coordinates, so they need to be mapped to model space and then to tangent space.

The tangent space should be chosen such that $\mathbf{p}_i$ has the coordinates $\langle s_i - s_0, t_i - t_0, 0\rangle$ (*i.e.*, $\mathbf{p}_0$ is the origin of the tangent space) Let

$$\begin{array}{rcll} \mathbf{u} & = & \mathbf{p}_1 - \mathbf{p}_0 & \text{object-space vector} \\ \mathbf{v} & = & \mathbf{p}_2 - \mathbf{p}_0 & \text{object-space vector} \\ \langle u_1, v_1\rangle & = & \langle s_1 - s_0, t_1 - t_0\rangle & \text{tangent-plane coords of } \mathbf{p}_1 \\ \langle u_2, v_2\rangle & = & \langle s_2 - s_0, t_2 - t_0\rangle & \text{tangent-plane coords of } \mathbf{p}_2 \end{array}$$

as illustrated by this picture



Since $\mathbf{t}$ and $\mathbf{b}$ form a basis for the texture's coordinate system (*i.e.* tangent plane), we have

$$\begin{aligned} \mathbf{u} &= u_1\mathbf{t} + v_1\mathbf{b} \\ \mathbf{v} &= u_2\mathbf{t} + v_2\mathbf{b} \end{aligned}$$

Converting these equations to matrix form gives us:

$$\begin{bmatrix} \mathbf{u}^{\mathrm{T}} \\ \mathbf{v}^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \begin{bmatrix} \mathbf{t}^{\mathrm{T}} \\ \mathbf{b}^{\mathrm{T}} \end{bmatrix}$$

Multiplying both sides by the inverse of the $\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix}$ matrix gives us the definition of the $\mathbf{t}$ and $\mathbf{b}$ vectors:

$$\frac{1}{u_1v_2 - u_2v_1} \begin{bmatrix} v_2 & -v_1 \\ -u_2 & u_1 \end{bmatrix} \begin{bmatrix} \mathbf{u}^{\mathrm{T}} \\ \mathbf{v}^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} \mathbf{t}^{\mathrm{T}} \\ \mathbf{b}^{\mathrm{T}} \end{bmatrix}$$

The matrix $[\mathbf{t}\ \ \mathbf{b}\ \ \mathbf{n}]$ maps from tangent space to object space, where $\mathbf{n}$ is the normal vector assigned to $\mathbf{p}_0$. If this matrix were guaranteed to be orthogonal, then we could take its transpose to get the mapping from object space to tangent space that we need. In practice, however, it is likely to be close, but not orthogonal. We can use a technique called *Gram-Schmidt Orthogonalization* to transform it into a orthogonal matrix. Using this technique, we get

$$\begin{aligned} \mathbf{t}' &= \text{normalize}\left(\mathbf{t} - \text{proj}_{\mathbf{n}}\mathbf{t}\right) \\ &= \text{normalize}\left(\mathbf{t} - (\mathbf{t}\cdot\mathbf{n})\mathbf{n}\right) \\ \mathbf{b}' &= \text{normalize}\left(\mathbf{b} - (\mathbf{b}\cdot\mathbf{n})\mathbf{n} - (\mathbf{b}\cdot\mathbf{t}')\mathbf{t}'\right) \end{aligned}$$

Finally, we can define the mapping from object space to tangent space to be $\begin{bmatrix} \mathbf{t}'^{\mathrm{T}} \\ \mathbf{b}'^{\mathrm{T}} \\ \mathbf{n}^{\mathrm{T}} \end{bmatrix}$. Applying this transform to the light vector $\mathbf{l}$ gives us $\langle \mathbf{t}'\cdot\mathbf{l}, \mathbf{b}'\cdot\mathbf{l}, \mathbf{n}\cdot\mathbf{l}\rangle$, which we can dot with the normal from the normal map to compute the lighting.

We only need compute the transformed light vector at vertices and then let the rasterizer interpolate it across the triangle's surface.

Also, note that since $\mathbf{b}' = \pm 1(\mathbf{n}\times\mathbf{t}')$, we only need to store four floats per vertex to define the mapping from object space into tangent space. In our vertex shader, we set $\mathbf{b}' = \mathbf{t}'_w(\mathbf{n}\times\mathbf{t}'_{xyz})$, where $\mathbf{t}'_w \in \{-1, 1\}$.

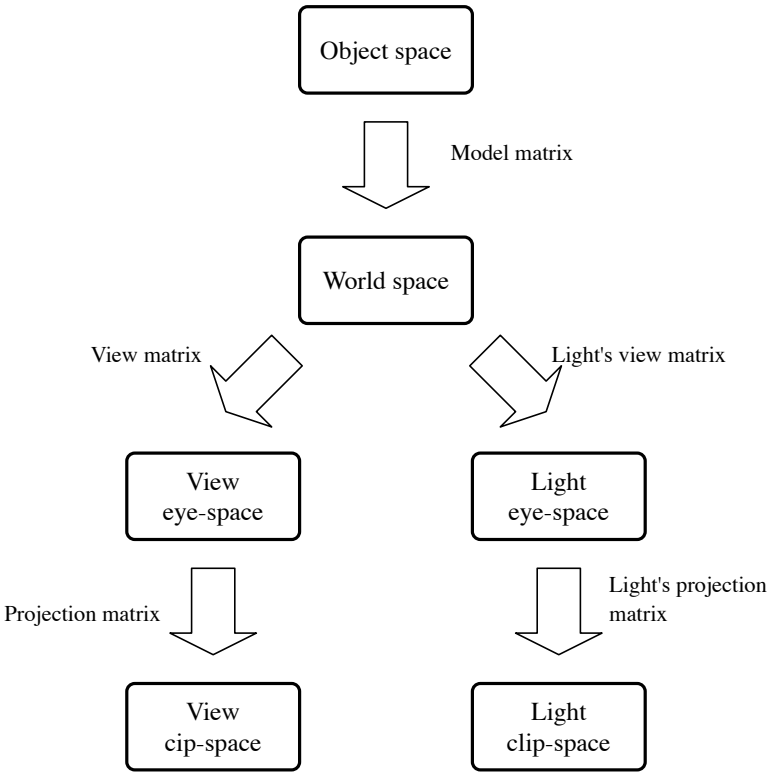


Figure 4: The coordinate-space transforms required for shadow mapping

To support normal mapping, you will need to compute the vector $\langle \mathbf{t}'_{xyz}, \mathbf{t}'_w \rangle$ for each vertex in the mesh.

2.5 Shadows

Shadows are one of the most important visual cues for understanding the relationship of objects in a 3D scene. As discussed in class, there are a number of techniques that can be used to render shadows using OpenGL. For this project, we will use shadow mapping.

The idea is to compute a map (*i.e.*, texture) that identifies which screen locations are shadowed with respect to a given light. We do this by rendering the scene from the light's point of view into the depth buffer. Then we copy the buffer into a *depth texture* that is used when rendering the scene. When rendering the scene, we compute a $\langle s, t, r \rangle$ texture coordinate for a point $\mathbf{p}$, which corresponds to the coordinates of that point in the light's clipping space. The r value represents the depth of $\mathbf{p}$ in the light's view, which is compared against the value stored at $\langle s, t \rangle$ in the depth texture. If r is greater than the texture value, then $\mathbf{p}$ is shadowed. To implement this technique, we must construct a transformation that maps eye-space coordinates to the light's clip-space (see Figure 4). Let

- $\mathbf{M}_{model}$ be the model matrix
- $\mathbf{M}_{view}$ be the eye's view matrix

- $\mathbf{M}_{light}$ be the light's view matrix, and
- $\mathbf{P}_{light}$ be the light's projection matrix.

To map a vertex $\mathbf{p}$ to the light's homogeneous clip space, we first apply the model's model matrix ($\mathbf{M}_{model}$), then the light's view matrix ($\mathbf{M}_{light}$), and finally the light's projection matrix ($\mathbf{P}_{light}$). After the perspective division, the coordinates will be in the range $[-1 \ldots 1]$, but we need values in the range $[0 \ldots 1]$ to index the depth texture, so we add a scaling transformation $(\mathrm{S}(0.5))$ and a translation $(\mathrm{T}(0.5, 0.5, 0.5))$.

You will need to compute the diffuse (D_l) intensity vectors using the techniques of Project 2. You should also compute the shadow factor S_l, which is given by the scene when the pixel is in shadow and 1.0 otherwise. Then, assuming that the input color is C_{in} and $\mathcal{L}$ is the set of enabled lights, the resulting color should be

$$C_{\text{out}} = \mathbf{clamp}\left(\sum_{l \in \mathcal{L}} S_l D_l\right) C_{\text{in}}$$

2.6 User interface

You will add support for two new commands to the Project 1 user interface:

`n N`	switch to normal-mapping mode
`s S`	switch to shadowed mode
`x X`	switch to eXtreme mode, which includes both normal mapping and shadows (extra credit).

In addition, your viewer should provide the camera controls that you implemented in Project 1.

Your viewer will support six different rendering modes (seven if you do the extra credit). To help you keep track of which features are enabled in which mode, consult the following table:

Rendering mode	**Polygon mode**	**Color**	**Diffuse Lighting**	**Texturing**	**Normal mapping**	**Shadow mapping**
Wireframe (w)	`GL_LINE`	✓				
Flat-shading (f)	`GL_FILL`	✓				
Diffuse (d)	`GL_FILL`		✓			
Textured (t)	`GL_FILL`		✓	✓		
Normal-mapping (n)	`GL_FILL`		✓	✓	✓	
Shadowed (s)	`GL_FILL`		✓	✓		✓
Extreme (x)	`GL_FILL`		✓	✓	✓	✓

3 Sample code

Once the Project 2 deadline has passed, we will seed a `proj3` directory with a copy of your source code and shaders from Project 2. We will update this code with a new implementation of the `Scene` class. The seed code will also include a new `Makefile` in the build directory and new scenes.

4 Summary

For this project, you will have to do the following:

- Fix any issues with your Project 2 code.
- Modify your Project 2 data structures to support object frames and normal mapping.
- Add support for loading and rendering the ground.
- Write a shader that renders objects using normal maps.
- Write a shader that renders objects with shadows.
- Add support for the new rendering modes to the UI.
- For extra credit, implement an *extreme mode* that supports both normal mapping and shadows.

5 Submission

We have set up an **svn** repository for each student on the `phoenixforge.cs.uchicago.edu` server and we will populate your repository with the sample code. You should commit the final version of your project by 10:00pm on Monday November 2. Remember to make sure that your final version **compiles** before committing!

History

2015-10-22 Simplified description of bump mapping.

2015-10-16 Original version.