Computer Graphics – Lab Assignment
Texture Mapping

November 23, 2010

1 Introduction

This assignment will familiarize you with different aspects of texture mapping.

The provided framework loads texture images from PPM files, a number of which are provided. The framework also sets up a lot of the OpenGL 2D texturing state for you. It does not apply texture coordinates for you, which you will be asked to do that yourself in the assignments below.

A number of keys are available in the framework application. The `t` toggles texturing on/off. The key `l` renders the scene in lines/wireframe, `p` renders the scene in "normal" polygons. The camera position can be controlled using the mouse (left button), including zoom (right button). The `o` key can be used to switch between different centers of camera rotation, which roughly correspond to some of the objects in the scene.

The coordinate system used in the framework is as follows: the Y-axis points upwards. Imagine the X-axis pointing to the right and the Y-axis pointing up then the Z-axis points towards you. This is a so-called "right-handed" coordinate system.

1.1 Object files

A number of 3D objects is provided in text files, with extension `.obj`. These files use a simple format, suitable for editing in a standard text editor. The

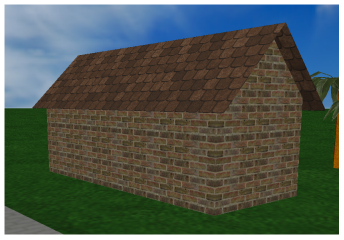

Figure 1: The textured house, ground plane, part of the road and skydome

format is line-oriented and any line starting with a `#` character is considered a comment line. The files declare one or more vertices and one or more polygons using these vertices.

Vertices are declared by a single line of the format `"v <x> <y> <z>"`, giving the coordinates of the vertex. Each declared vertex is given an integer index, starting at 0.

Polygons are declared using multiple lines. The first line for each polygon is of the form `"p <n> <t>"` and lists the number of vertices n in the polygon and the texture identifier t. Both of these are integers. Note that the framework only supports polygons with 3 or 4 vertices (i.e. triangles and quads), as OpenGL does not really cope well with polygons with more vertices.

Then follows a line describing the color of the polygon, which is used as the object color for non-textured displaying. For textured display this color is combined with the color from the texture to calculate the final color. This line is of the form `"<r> <g> <b>"`, with three values in the range $[0, 1]$.

Finally, n lines are given, one for each of the vertices in the polygon. These lines are of the form `"<vi> <s> <t>"`. Here, vi is the index of a vertex previously declared and s and t are the texture coordinates to be used for this vertex. See for example, the file `ground.obj`, which creates a plane defined by 4 vertices. Note that an assumption is made that all polygons are planar, so that a polygon normal can be easily calculated by the framework.

The texture identifier is used as an index in the array of OpenGL texture names, `texture_names`. This array contains the texture names generated at run-time for each of the loaded texture images. Take a look at function `InitGL` in *main.c* to see how this is handled.

2 Assignments

2.1 Part 1 - Basic texturing

Complete the following tasks:

- In `DrawPolylist` add a call to `glTexCoord2f` in the correct place so that it applies texture coordinates for polygon vertices. See the lecture sheets on texture mapping for hints and code examples.

- Open the road texture *textures/road.ppm* in an image viewer, e.g. **cd textures; eog road.ppm**. This texture should be mapped on the road object (the grey strip running straight through the scene) in such a way that the full height of the image is mapped along the smallest side of the strip. Set the correct texture coordinates in *road.obj* in such a way that the texture image does not get stretched but is repeated.

  You will notice that the texture seems to be drawn only once, i.e. it isn't repeated. Alter the relevant OpenGL call(s) to enable texture repeating. Hint: look at function `InitGL` and the lecture sheets.

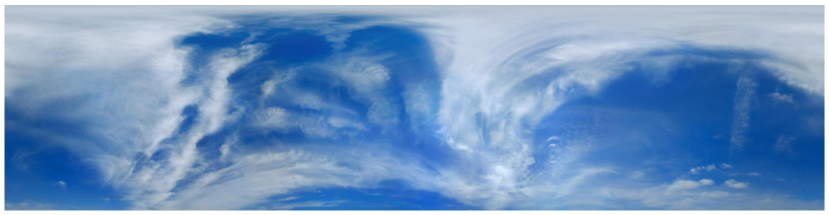

Figure 2: The sky texture, to be mapped onto the skydome

- The green ground plane actually consist of two copies of the polygonal model defined in *ground.obj*, separated by the road. Add texture coordinates to the ground plane model so that it is covered with the grass texture 20 times in one direction and 10 times in the other one, thereby again not stretching the texture image.

- The house model in the scene (in `house.obj`) is made up of wall polygons and roof polygons. Add texture coordinates to the roof polygons so that the roof texture image is repeated 3 times along the longest side and once along the other side. The tiles in the texture should be oriented the way you would expect for a roof, see Figure 1.

  There's two types of wall polygons used in the house: rectangular ones and 5-sided ones. Add textures coordinates to the walls of the house model. Do this in such a way such that texture coordinates match between different wall polygons, so the texture should continue fluently from one wall to the other. The texture image should appear a total of 7 times, when counting horizontally along the four walls. The texture image should not change its aspect ratio (the ratio between width and height of the image). This means the applied texture should not appear to be stretched horizontally or vertically, compared to the original texture image.

  Hint: making a little drawing of the house geometry together with its dimensions might help here.

- The scene also includes what is known as a *skydome*. This is a hemisphere surrounding the scene, which acts as a sort of sky. The skydome is not stored in a file but is procedurally generated, by function `createHemisphere()` in *geometry.c*. In the framework displaying of the skydome is commented out in function `DrawGLScene()`, to give you a clear view of the ground plane while texturing it with the grass. Remove the commenting so the skydome is drawn.

  A texture is provided (`sky.ppm`) that should be mapped on the hemisphere, see Figure 2. If you imagine this image to be rolled up from left to right into a cylinder, followed by folding the top of the cylinder to come together in a single point, then you end up with the way the image should be mapped onto the hemisphere. I.e., the bottom of the image should run once around the bottom of the hemisphere (the horizon). The top line of the image should converge in the top of the hemisphere.

The texture coordinates for the skydome's vertices are currently all set to zero. Alter the relevant function(s) in `geometry.c` so that the correct coordinates are applied.

If you've successfully set the hemisphere texture coordinates, it should be relatively straightforward to also alter function `createCylinder()`, which is used to draw the stem of the tree. Alter this function as well.

2.2 Part 2 - Mip-mapping

You might notice Moiré patterns in the different textured polygons, especially in the "sidewalk" part of the road, but also on the house walls at certain distances.

These are the places where OpenGL needs to combine several texture pixels (texels) to obtain the color of a single screen pixel, so-called texture minification. The default method for this is to use linear interpolation. A better way is to use mip-mapping, as shown during the lecture.

Enable mip-mapping, by replace the `glTexImage2D` call with a call to `gluBuild2DMipmaps` and setting/changing the relevant texturing parameters. See the man-pages of `gluBuild2DMipmaps` and `glTexParameteri` for details. Try out the different mip-mapping minification filters that OpenGL provides to notice the difference and select the one you think works best.

2.3 Part 3 - Using textures as objects

The tree objects so far looks pretty simplistic, i.e. just cylinders with spheres on top. In `DrawGLScene()` the trees are drawn with a simple for-loop that draws the same polygon lists multiple times, each time with slightly different transformations to put tree objects in different locations.

We haven't done any texturing yet on the sphere, mostly because it is very similar to the skydome texturing. We are going to replace the sphere with something a bit more "leaf-like". For this, a texture image is provided of a large leaf from a banana palm, see Figure 3. To mark transparent pixels in this image

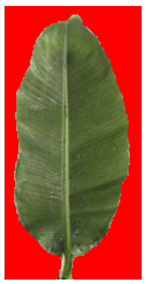

Figure 3: A texture of a single leaf from a banana palm

the color red is used. When the image is loaded by the framework these red pixels are replaced by fully transparent pixels.

For the final part of this assignment you are asked to replace the tree spheres with simple polygonal objects that are textured using the leaf texture. For this you will have to both create some geometry and assign the relevant texture coordinates. This is most conveniently done using a separate `.obj` file. You can then read in this file and use the geometry from there. The palm texture is already loaded by the framework.

Use at least 8 vertices for a leaf and display 5 to 10 leafs at the location of the sphere (which you can remove), similar to the way a real tree's leafs would be organized. If you like to use random numbers for this you can use function `rand_float()`, which will return a random floating-point value in the range $[0, 1]$.

See Figure 4 for a possible use of the leaf texture.

3 Tips

You can look at the provided texture images using an image editing/viewing package like The GIMP (`gimp`) or Eye of Gnome (`eog`). With the GIMP you could even create a special test texture image, to aid you in setting texture coordinates.

4 Grading

Don't forget to include the relevant `.obj` *files when sending in your submission!*

You can receive up to 6 points for part 1 (setting of texture coordinates 0.5, road texture 0.5, ground texture 0.5, repeating textures 0.5, house texturing 2, skydome texturing 1.5, cylindrical texturing 0.5).

You can receive up to 1 point for adding and enabling mip-mapping.

You can receive up to 2 points for the leafs and their texturing.

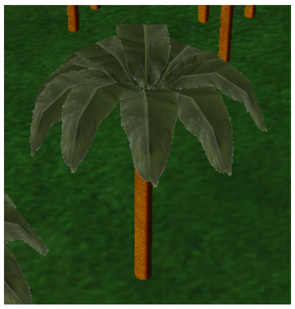

Figure 4: Multiple leafs on a palm tree

You can get one additional point for writing clean, well-structured and well-commented code (on the opposite side, unreadable or overly complex code might cost you a point).