6.837 Introduction to Computer Graphics
Assignment 5: OpenGL and Solid Textures
Due Wednesday October 22, 2003 at 11:59pm

In this assignment, you will add an interactive preview of the scene and solid textures. For interactive display, you will use the OpenGL API that uses graphics hardware for fast rendering of 3D polygons[1]. You will be able to interactively pre-visualize your scene and change the viewpoint, before using ray-tracing for higher-quality rendering. Most of the infrastructure is provided to you, and you will mostly need to add functions that send the appropriate triangle-rendering commands to the API to render or "paint" each kind of `Object3D` primitive. Finally, you will add new Material effects where the color of the material varies spatially using procedural solid texturing. This will allow you to render checkerboard planes and truly satisfy the historical rules of ray-tracing.

The two parts of this assignment are mostly independent.

1 OpenGL Rendering

In OpenGL, you display primitives by sending commands to the API. The API takes care of the perspective projection and the various other transformations, and also "rasterizes" polygons,. i.e., it draws the appropriate pixels for each polygon. How it does this is the subject of the following series of lectures on the *rendering pipeline*. In addition, the infrastructure we provide takes care of the user interface and how the mouse controls the camera.

Using OpenGL

To use OpenGL on Athena, you will first need to obtain access to the OpenGL libraries and header files. To do this, from an Athena prompt, type:

```
add mesa
```

If you are using Windows, then you may need to download the OpenGL libraries yourself from `http://www.opengl.org`.

To add an OpenGL rendering interface to your application, you will use the class `GLCanvas` provided in `glCanvas.h` and `glCanvas.C`. They rely on an updated version of `light.C` and `light.h`; on a new member method, `paint`, of

[1]On some configuration, software emulation might be used, resulting in slower rendering.

`Object3D`; and on new methods that will be added to your `Camera` class. Most of the code for these routines is provided. You will need to write the `paint()` routine for each `Object3D` subclass to render the primitives within the canvas.

To use the GLCanvas as the real-time front-end for your application, you will need to create a GLCanvas object in your main routine and call the following function:

```
  glCanvas.initialize(SceneParser *_scene,
                      void (*_renderFunction)(void));
```

The `initialize` routine takes two parameters: The first is a pointer to the global scene. The second is the function that will perform the raytracing. The GLCanvas class is set up so that the renderFunction takes no parameters and has a `void` return type. From within the real-time interface (with the mouse cursor within the frame of the GL display window), you can call the render function by pressing 'r'.

Once the initialize routine is called, the GLCanvas will take over control of the application and will monitor all mouse and keyboard events. This routine will not return, although the application can be terminated by closing the window or calling `exit(0)`.

All files implementing OpenGL code should include the OpenGL header files:

```
// Included files for OpenGL Rendering
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
```

1.1 Camera and Object3D

First you will need to modify your camera implementation to control the interactive camera. Copy-paste the code provided in `camera_additions.txt` into your camera files, and update it to re-normalize an re-orthogonalize the up, direction, and horizontal vectors (similar to the camera constructor).

Use the left mouse button to rotate the camera around the center of the scene, the middle mouse button to translate the scene center (truck), and the right mouse button to move the camera closer to or further from the scene (dolly). To prevent weird rotations of the camera, it is necessary to store the original up vector of the camera and define a new "screen up" vector that is the normalized orthogonal up vector for the current direction vector.

Next, you will add a virtual function `void paint ()` to your `Object3D` class. To test your code incrementally, do not make this function pure virtual `= 0;`, but instead implement this method in `Object3D` with an empty function. This way, you will be able to test your code even before you have implemented the function for all subclasses.

You should now be able to test your viewer. No primitive will be displayed because you have not yet implemented the appropriate `paint` methods, but the canvas renders three axes depicting the world-space frame.

1.2 Group

Similar to the intersection method, a group implements `paint` by iterating over all its children and calling their `paint` methods.

1.3 Triangle

OpenGL is based on polygons. You tell the API to render a polygon by first telling it that you start a polygon, then describing all the vertices and their properties, and finally closing the polygon. The code to specify just the positions of a single triangle looks like this:

```
glBegin(GL_TRIANGLES);
   glVertex3f(x0, y0, z0);
   glVertex3f(x1, y1, z1);
   glVertex3f(x2, y2, z2);
 glEnd();
```

Alternatively, you can directly specify an array of floats for each vertex using `glVertex3fv(float *arr)`. Implement the `paint` method for your `Triangle` class.

1.4 Normals and materials

OpenGL can also compute local illumination. However, it cannot easily compute cast shadows. To take local illumination into account, you must provide the material and normal information for the triangle.

First, you must update your `Material` and `PhongMaterial` implementation and add a `virtual void Material::glSetMaterial() const` member function. This function will send the appropriate OpenGL commands to specify the local shading model. Copy the code from `material_additions.txt` for the Phong model.

For the `paint` method of each primitive, before sending any OpenGL geometric command, you will need to call the `void Material::glSetMaterial() const` member function to set up the GL material parameters. Do so in the `Object3D` parent class, and call `Object3D::paint()` at the beginning of the `paint` of each subclass.

To set the triangle normal, use one of the following commands before specifying the vertices:

```
glNormal3f(float x, float y, float z);     // List of floats
glNormal3fv(float *arr);                   // Array of floats
```

Remember that you can compute the normal of a triangle using a cross product. Test your triangle `paint` routine with the simple test scenes provided.

1.5 Plane

OpenGL does not have an infinite plane primitive. To pre-visualize planes, you will simply use very big rectangles. Project the world origin (0,0,0) onto the plane, and compute two basis vectors for the plane that are orthogonal to the normal $\vec{n}$. The first basis vector may be obtained by taking the cross product between the normal and another vector $\vec{v}$. Any vector $\vec{v}$ will do the trick, as long as it is not parallel to $\vec{n}$. So you can always use $\vec{v} = (1, 0, 0)$ except when $\vec{n}$ is along the x axis, in which case you can use $\vec{v} = (0, 1, 0)$. Then the first basis vector, $\vec{b_1}$, is $\vec{v} \times \vec{n}$ and the second basis vector, $\vec{b_2}$, is $\vec{v} \times \vec{b_1}$ Display a rectangle from (-BIG, -BIG) to (BIG, BIG) in this 2D basis (Caution: OpenGL does not like rendering points at INFINITY). In OpenGL, you may include $3n$ vertex positions within the `glBegin` and `glEnd` commands to draw n triangles:

```
 glBegin(GL_TRIANGLES);
    glNormal3f(nx, ny, nz);

    glVertex3f(x0, y0, z0);
    glVertex3f(x1, y1, z1);
    glVertex3f(x2, y2, z2);

    glVertex3f(x2, y2, z2);
    glVertex3f(x3, y3, z3);
    glVertex3f(x0, y0, z0);
  glEnd();
```

1.6 Sphere

OpenGL does not have a sphere primitive, so spheres must be transformed into triangles, a process known as *tessellation* [2]. You will implement the classic sphere tessellation using angular parameters θ and φ. The number of steps in θ and φ will be controlled by a command line argument `-tessellation <theta> <phi>`. Deduce the corresponding angle increments, and use two nested loops on the angles to generate the appropriate triangles. Note that θ should vary between 0 and 360°, while φ must vary between $-90°$ and 90°. You can use a single `glBegin(GL_TRIANGLES)`, `glEnd()` pair for the entire sphere:

```
glBegin(GL_TRIANGLES);
    for (iPhi=...; iPhi<...; iPhi+=...)
        for (int iTheta=...; iTheta=...; iTheta+=...)
        {
            //compute appropriate coordinates

            //send gl vertex commands
```

[2] Actually, `glu` does have a sphere primitive, but you are not allowed to use this shortcut for the assignment

```
            glVertex3f(x0, y0, z0);
            glVertex3f(x1, y1, z1);
            glVertex3f(x2, y2, z2);

            glVertex3f(x2, y2, z2);
            glVertex3f(x3, y3, z3);
            glVertex3f(x0, y0, z0);
        }
glEnd();
```

You will implement two versions of sphere normals: flat shading (visible facets) and *Gouraud interpolation*. For flat shading, you will use the normal of each triangle, as you would for polygon rendering. For the Gouraud interpolation version, you will use the true normal of the sphere for each vertex (set the vertex normal before specifying each vertex position). Note how this improves the appearance of the sphere and makes it smoother. OpenGL performs bilinear interpolation between the color values computed at each vertex. Note that this is not as good as the *Phong interpolation* described in class (which interpolates the surface normals then performs the lighting calculation per pixel).

1.7 Transformation

Finally, you must handle transformations. OpenGL will do most of the work for you. You only need to specify that you want to change the current object-space-to-world-space 4x4 matrix. To do this, you first need to save the current matrix on a matrix stack using `glPushMatrix()`. Then change the matrix using `glMultMatrix(GLfloat *fd)`. We have added a `glGet()` routine to the `Matrix` class to construct a matrix with the appropriate structure. OpenGL matrices created with this routine should be deleted when they are no longer needed:

```
glPushMatrix();
GLfloat *glMatrix = matrix.glGet();
glMultMatrixf(glMatrix);
delete[] glMatrix;
```

Then, recursively call the `paint` method of the child object. After this, you must restore the previous matrix from the stack using:

```
glPopMatrix();
```

If you do not save and restore the matrix, your transformation will be applied to all the following primitives!

2 Solid textures

Solid textures are a simple way to obtain material variation over an object. Depending on the spatial location of the shaded point, different material properties (color, specular coefficients, etc.) are chosen. In this assignment, you will implement a simple axis-aligned checkerboard texture that selects between two Materials.

Checkerboard class

You will derive a class `Checkerboard` from `Material`. SceneParser has been extended to parse `CheckerBoard` materials. Your `CheckerBoard` class will store pointers to two materials, and is parameterized by the cell size (a float). The prototype for the constructor should be:

```
CheckerBoard(Material *mat1, Material *mat2, float sizeCell);
```

First, handle this new material type in the interactive viewer by implementing `CheckerBoard::glSetMaterial()`. Because OpenGL does not implement general procedural texturing, you will simply call the corresponding `glSetMaterial()` method of the first material.

Next, implement the `CheckerBoard::shade` routine used in your ray tracer. For this, you simply need to decide which of the two materials' `shade` function will be called. Using a float-to-integer conversion function and the function `odd`, devise a boolean function that corresponds to a 3D checkerboard. As a hint, start with the 1D case, just alternated segments of size `sizeCell`, then generalize to 2 and 3 dimensions.

Your checkerboard material can be used for fun recursive material definitions. You can have a checkerboard where cells have different refraction and reflection characteristics, or a nested checkerboard containing different checkerboards. This notion of recursive shaders is central to production rendering.

3 What to Turn In

As usual, turn in your executable and modified source files.

Include a README.txt describing how long it took you to complete the assignment, the major problems you encountered, and the description of any extra-credit work that you did.

4 Hints

As usual, debug your code as you write it. Run intermediate examples to make sure that sub-parts are sane.

Performances

In your code, you might want to extract the values that do not need to be recomputed for each frame and cache them. For example, you might want to store triangle normals or the tessellation coordinates of a sphere. As usual, there is a trade off between speed and memory.

5 Ideas for Extra Credit

Perlin noise and turbulence; Wood; Marble; add matrices to solid textures; Supersampling; Texture mapping (need for parameterization); Phong normal interpolation; distribution ray tracing; subsurface scattering.

6 Other

New command line arguments

`-tessellation` nTheta, nPhi
(tessellation of the sphere)

`-gouraud`
(gouraud smooth shading for spheres)