6.837 Intro to Computer Graphics
Assignment 3: OpenGL & Phong Shading

In this assignment, you will add an interactive preview of the scene, and implement Phong Shading in your ray tracer. For interactive display, you will use the OpenGL API that uses graphics hardware for fast rendering of 3D polygons. Note: with some configurations, software emulation might be used, resulting in slower rendering. You will be able to interactively pre-visualize your scene and change the viewpoint, then use your ray-tracer for higher-quality rendering. Most of the infrastructure is provided to you, and you will just need to add functions that send the appropriate triangle-rendering commands to the API to render or *paint* each kind of `Object3D` primitive. 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. (In lecture, we will talk about how this is done using the *rendering pipeline*). In addition, the infrastructure we provide takes care of the user interface and how the mouse controls the camera.

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_v501
```

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>
```

We provide an updated `Makefile` to link with the OpenGL libraries on Athena Linux. If you are using Windows, then you may need to download the OpenGL libraries yourself from http://www.opengl.org.

Tasks

- Add an OpenGL pre-visualization interface to your ray tracer, using the `GLCanvas` class. You will need to create a `GLCanvas` instance in your main routine and call the following function:

  ```
  void 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 pressing `'q'`, by closing the window, or calling `exit()`.

  - glCanvas.h
  - glCanvas.C

- 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 and re-orthogonalize the up, direction, and horizontal vectors (similar to the camera constructor). Make sure you read the comments provided in `camera_additions.txt`, as these will help you avoid some common pitfalls. Also pay close attention to the variable names in the provided code, as these may not correspond to the variable names in your own code.

  - camera_additions.txt

  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. You can test your implementation at this point. In the `GLCanvas::display()` function is a call to the provided `drawAxes()` function, which will allow you to debug your camera implementation.

  Once you start working on your `Object3D` paint methods (described below), you should comment out the call to `drawAxes`. Once your paint methods are complete, verify that your camera manipulation code is correct by moving the camera, rendering the scene & then comparing the pre-visualization to the raytraced result.

- Derive a new `PhongMaterial` class from `Material` that adds a specular component (highlight). The constructor expected by the scene parser is:

  ```
  PhongMaterial::PhongMaterial(const Vec3f &diffuseColor,
     const Vec3f &specularColor, float exponent);
  ```

  Make the original `Material` class pure virtual by adding a pure virtual method `Shade` that computes the local interaction of light and the material:

  ```
  virtual Vec3f Material::Shade
    (const Ray &ray, const Hit &hit, const Vec3f &dirToLight,
     const Vec3f &lightColor) const = 0;
  ```

  It takes as input the viewing ray, the Hit data structure, and light information and returns the color for that pixel. Implement the the Blinn-Torrance version of the Phong model taught in lecture. We suggest this version simply because it is what OpenGL uses, and the pre-visualization will thus be more similar.

  Note: We've only included one type of light source in our scenes, directional light sources, which have no falloff. That is, the distance to the light source has no impact on the intensity of light received at a particular point in space. So you may ignore the r^2 term in the Phong lighting model for this assignment. Next week we will add a second type of light source, a point light source, and the distance from the surface to the light will be important.

- OpenGL can also compute local illumination. You'll need to add the `virtual void Material::glSetMaterial() const` member function to `Material` (we have provided the code). This function will send the appropriate OpenGL commands to specify the local shading model.

- material_additions.txt

In the examples below we illustrate an artifact that occurs at grazing angles for wide specular lobes (small exponents). To solve this problem, the specular component can be multiplied by the dot product of the normal and direction to the light instead of simply clamping it to zero when this dot product is negative. You may implement either method in your ray tracer. To enable the specular lobe fix in OpenGL, a 3-pass rendering has been provided in the code (by default only the single pass rendering is performed). It is not required that you use or understand this code.

- Add a pure virtual `void paint()` method to `Object3D` and implement it for each subclass. This function executes the appropriate OpenGL calls to render each object in the pre-visualization interface. Before sending any OpenGL geometric commands, you will need to call the `void Material::glSetMaterial() const` member function to set up the OpenGL material parameters. Some useful code is provided in `object3d_additions.txt`.

  - **Group -** Similar to its intersection method, a group implements `paint()` by iterating over all its children and calling their `paint()` methods.
  - **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 *array)`. 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.

  - **Plane -** OpenGL does not have an infinite plane primitive. To pre-visualize planes, you will simply draw 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 plane normal **n**. The first basis vector may be obtained by taking the cross product between **n** and another vector **v**. Any vector **v** will do the trick, as long as it is not parallel to **n**. So you can always use **v**=(1,0,0) except when **n** is along the x axis, in which case you can use **v**=(0,1,0). Then the first basis vector, $\mathbf{b_1}$, is **v** x **n** and the second basis vector, $\mathbf{b_2}$, is **n** x $\mathbf{b_1}$. Display a rectangle from (-*big*, -*big*) to (*big*, *big*) in this 2D basis, for some big number *big*. (Caution: OpenGL does not like rendering points at INFINITY. *big* should probably be < 10^6)

    ```
    glBegin(GL_QUADS);
    glNormal3f(nx, ny, nz);
    glVertex3f(x0, y0, z0);
    glVertex3f(x1, y1, z1);
    glVertex3f(x2, y2, z2);
    glVertex3f(x3, y3, z3);
    glEnd();
    ```

  - **Sphere -** OpenGL does not have a sphere primitive, so spheres must be transformed into polygons, a process known as *tessellation*. (Actually, `glu` does have a sphere primitive, but you are not allowed to use this shortcut for the assignment). You will implement the classic sphere tessellation using angular parameters *theta* and *phi*. The number of steps in *theta* and *phi* will be controlled by the command line argument `-tessellation <theta-steps> <phi-steps>`. Deduce the corresponding angle increments, and use two nested loops on the angles to generate the appropriate polygons. Note that *theta* should vary between 0 and 360 degrees, while *phi* must vary between -90 and 90 degrees.

    In OpenGL, you may include 3*n* vertex positions within the `glBegin` and `glEnd` commands to draw *n* triangles (or 4*n* vertices to draw *n* quads):

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

          // send gl vertex commands
          glVertex3f(x0, y0, z0);
          glVertex3f(x1, y1, z1);
          glVertex3f(x2, y2, z2);
          glVertex3f(x3, y3, z3);
        }
      }
    glEnd();
    ```

    You will implement two versions of sphere normals: flat shading (visible facets) and *Gouraud interpolation*. By default your previsualization should use flat shading; that is, simply the normal of each triangle, as you would for polygon rendering. When the `-gouraud` option is specified, your pre-visualization should render with Gouraud interpolation, and 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 shaded color values computed at each vertex. Remember that this is not as good as the *Phong interpolation* described in class (which interpolates the surface normals and then performs the lighting calculation per pixel).

  - **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)`. Use the `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!

Hints

- As usual, debug your code as you write it. Initially implement empty `paint()` functions for the Object3D subclasses. Run intermediate examples to make sure that sub-parts are sane.
- To improve performance, you may 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.

Ideas for Extra Credit

- Implement the original Phong model (in addition to the Blinn-Torrance variation) and compare, add an approximation of the fresnel effect, implement a fish eye camera, etc.

Updated Files:

- Makefile
- light.h
- light.C
- scene_parser.h
- scene_parser.C

If you're interested, here's the scene description file grammar used in this assignment.

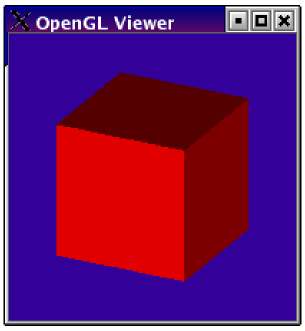
If you're interested, here's a list of command line arguments used in this assignment.

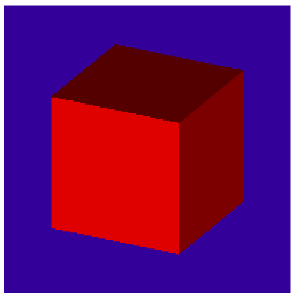
Input Files

- scene3_01_cube_orthographic.txt
- scene3_02_cube_perspective.txt
- scene3_03_bunny_mesh_200.txt
- scene3_04_bunny_mesh_1k.txt
- scene3_05_axes_cube.txt
- scene3_06_crazy_transforms.txt
- scene3_07_plane.txt
- scene3_08_sphere.txt
- scene3_09_exponent_variations.txt
- scene3_10_exponent_variations_back.txt
- scene3_11_weird_lighting_diffuse.txt
- scene3_12_weird_lighting_specular.txt

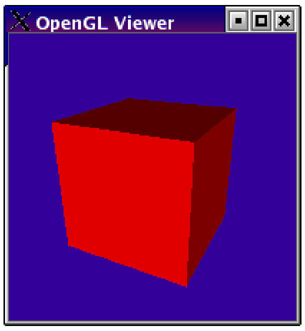
Sample Results

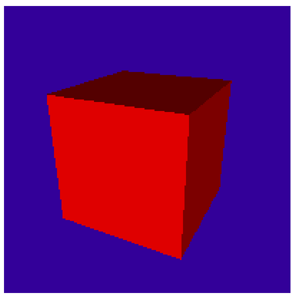
```
raytracer -input scene3_01_cube_orthographic.txt -size 200 200 -output output3_01.tga -gui
```



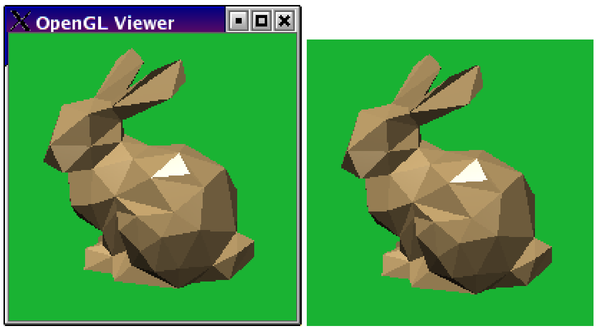


```
raytracer -input scene3_02_cube_perspective.txt -size 200 200 -output output3_02.tga -gui
```

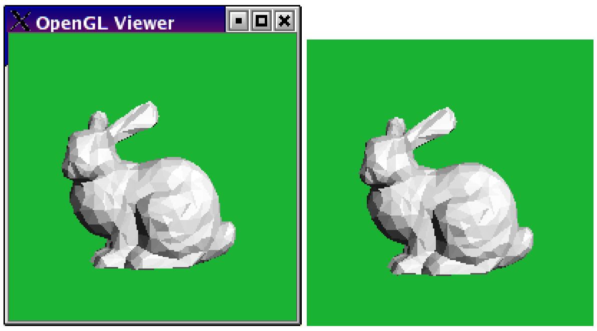




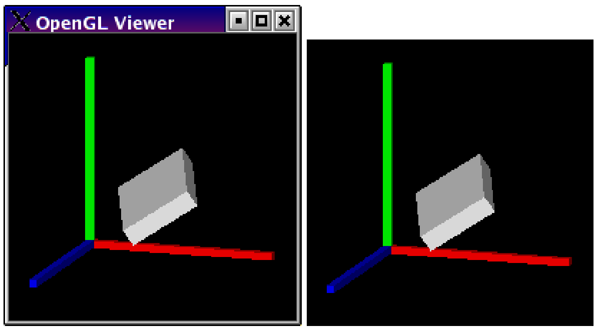
```
raytracer -input scene3_03_bunny_mesh_200.txt -size 200 200 -output output3_03.tga -gui
```



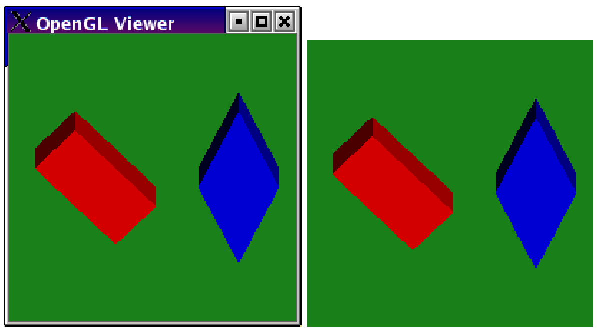
```
raytracer -input scene3_04_bunny_mesh_1k.txt -size 200 200 -output output3_04.tga -gui
```



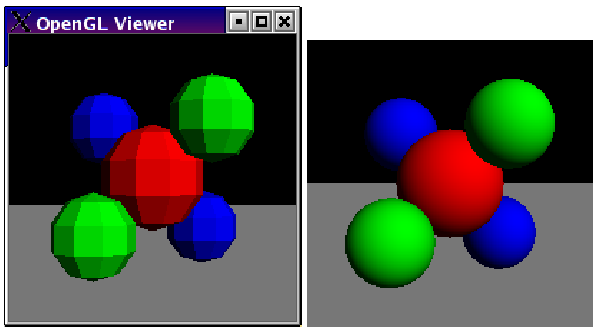
```
raytracer -input scene3_05_axes_cube.txt -size 200 200 -output output3_05.tga -gui
```



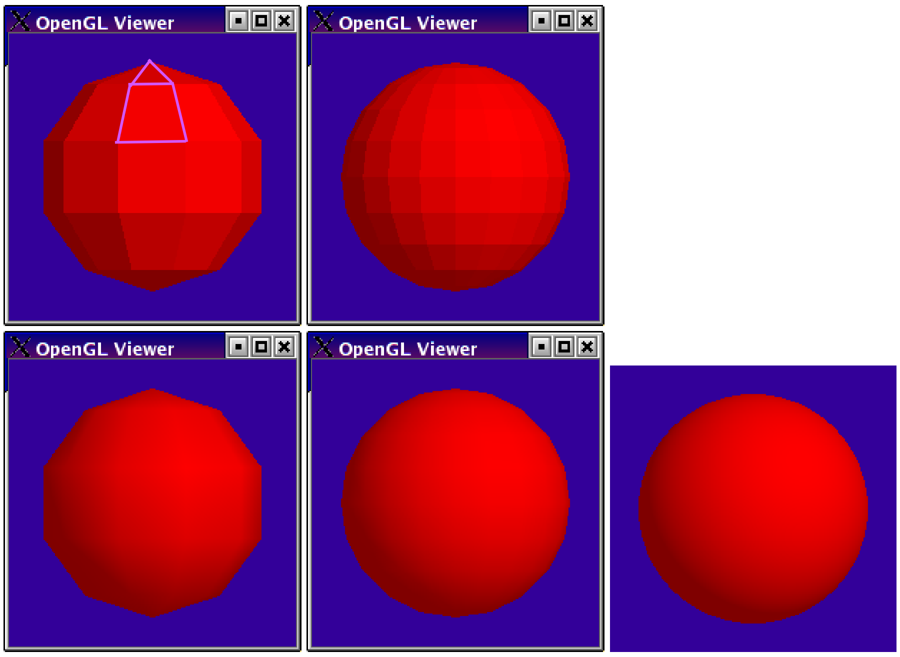
```
raytracer -input scene3_06_crazy_transforms.txt -size 200 200 -output output3_06.tga -gui
```



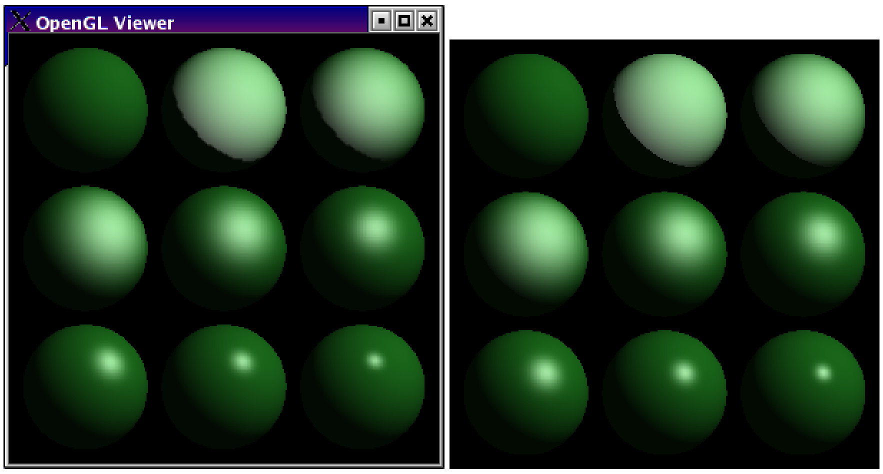
```
raytracer -input scene3_07_plane.txt -size 200 200 -output output3_07.tga -gui -tessellation 10 5
```



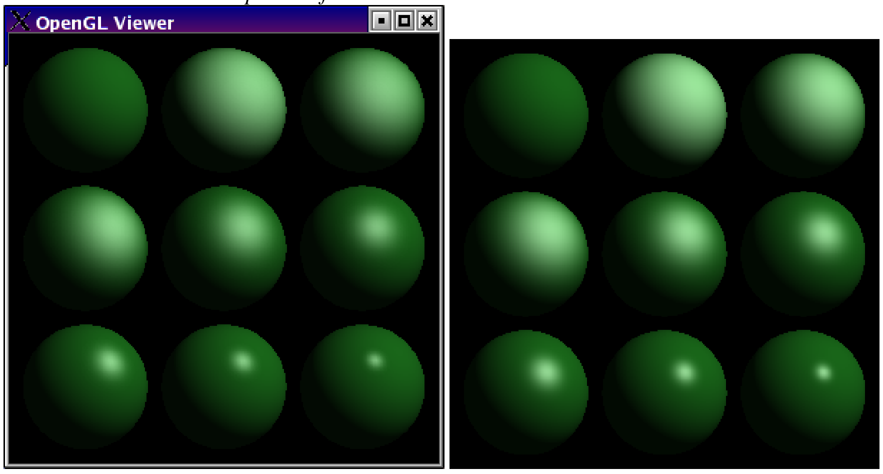
```
raytracer -input scene3_08_sphere.txt -size 200 200 -output output3_08.tga -gui -tessellation 10 5
raytracer -input scene3_08_sphere.txt -size 200 200 -output output3_08.tga -gui -tessellation 20 10
raytracer -input scene3_08_sphere.txt -size 200 200 -output output3_08.tga -gui -tessellation 10 5 -gouraud
raytracer -input scene3_08_sphere.txt -size 200 200 -output output3_08.tga -gui -tessellation 20 10 -gouraud
```



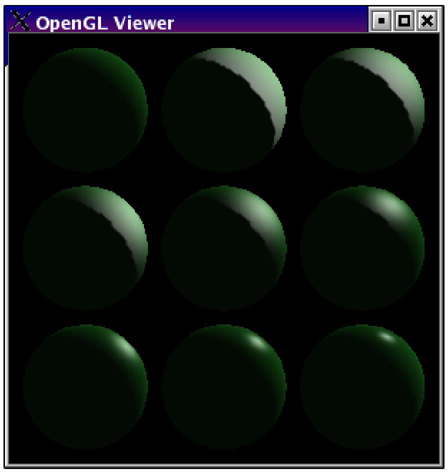
```
raytracer -input scene3_09_exponent_variations.txt -size 300 300 -output output3_09.tga -gui -tessellation 100 50 -gouraud
```

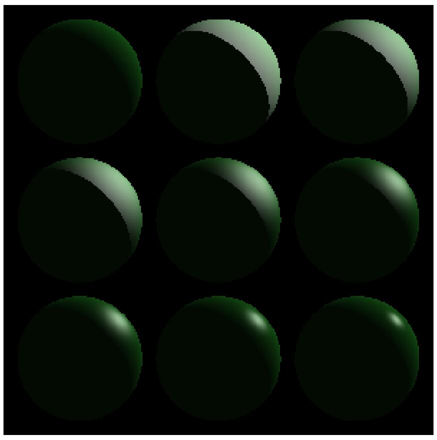


and with the OPTIONAL specular fix:

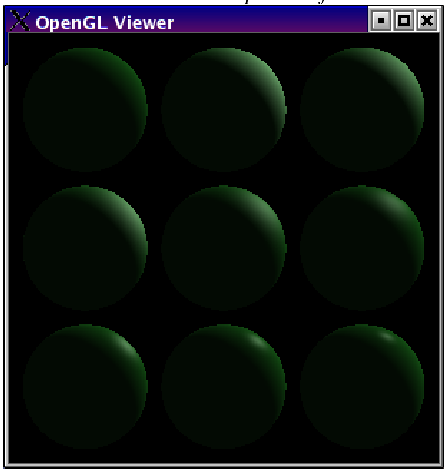


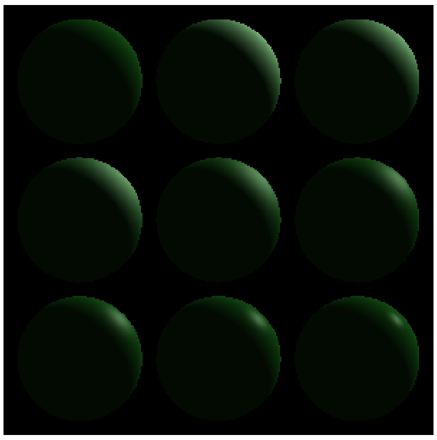
```
raytracer -input scene3_10_exponent_variations_back.txt -size 300 300 -output output3_10.tga -gui -tessellation 100 50 -gouraud
```



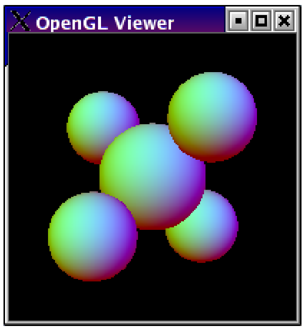


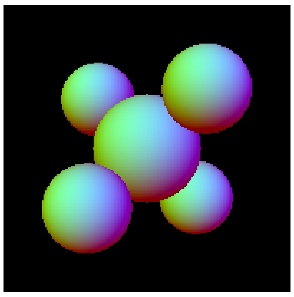
and with the OPTIONAL specular fix:



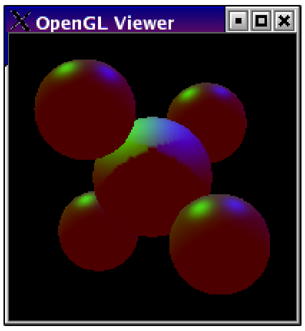


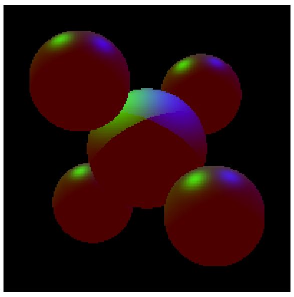
```
raytracer -input scene3_11_weird_lighting_diffuse.txt -size 200 200 -output output3_11.tga -gui -tessellation 100 50 -gouraud
```



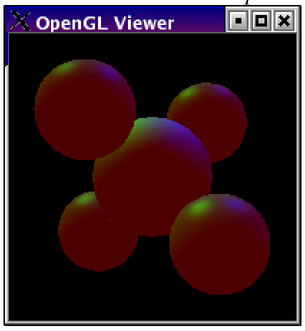


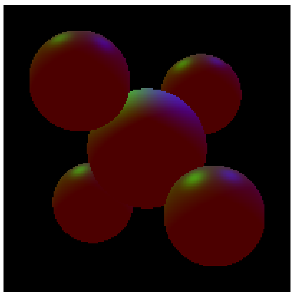
```
raytracer -input scene3_12_weird_lighting_specular.txt -size 200 200 -output output3_12.tga -gui -tessellation 100 50 -gouraud
```





and with the OPTIONAL specular fix:





See the main [Assignments Page](#) for submission information.
