CPSC 314
Assignment 4: Ray Tracer and Post-Processing

Due 4PM, Nov 29, 2013

This assignment will be done in pairs. The motivation is for you to experience pair programming (http://en.wikipedia.org/wiki/Pair_programming) and also reduce the individual workload/stress. The assignment consists of two parts. In part 1 you will implement a simple raytracer that supports spheres, planes, triangle meshes, and optionally other types of surfaces. In part 2 you will write shaders to apply basic post-processing effects to the images output from part 1. We recommend that each pair works together on both parts and not split it down the middle.

Template: The template code is found in the main assignment directory. Part 1 has you making additions to three of the template code files: `object.cpp`, `raytracer.cpp`, and `mesh.cpp`. In part 2 you will be modifying `filter.frag`, `filter.vert`, and `filter.cpp`. You do not need to make any changes to the other source files (though you can if you wish when implementing optional extra features).
There are two subdirectories, to be used for the raytracing in part 1: `scenes` and `meshes`. The `scenes` directory contains scene descriptions in the `.ray` format, describing the following scene parameters: Dimensions, Perspective, LookAt, Material, PushMatrix, PopMatrix, Translate, Rotate, Scale, Sphere, Plane, Mesh, and PointLight. The comments in those files describe the format. A few triangle meshes in OBJ format are provided in the `meshes` directory, and most of the scene files depend on one or more of these.

Execution: The `README` contains instructions for compiling and running your raytracer and post processor. The `raytracer` binary takes two optional arguments: the name of the scene description, and the name of the output PPM image files. The defaults are `scenes/basic.ray` and `output.ppm`. Furthermore, the binary accepts a flag prior to these inputs that will be stored in an internal integer in the filter. This value will be used in part 2 to allow you to swap between different filter functionality. In general the input will look like:
`"raytracer.exe [-#] [scene_desc_file] [output_file]"`

Part 1 of the assignment (the raytracer) outputs 2 image files: a color image, and a black-and-white depth map image which will be useful when debugging the program. As mentioned

above, the name of the output file is given by the specified input, and the name of the depth map image file is `filename_depth.ppm`, where `filename.ppm` is the specified output image file.
Part 2 of the assignment takes as input the two output files from part 1, and outputs two new images which are the input images with post-processing applied. The names of the processed images are `modified_filename.ppm`, where filename is the filename of the image that has been processed.
Reference solution executables `raytracer_sol` and `raytracer_sol.exe` are provided for comparison. Use them on the provided scenes to generate reference images.

1 RayTracer (70)

The raytracer should cast primary rays into the scene, which spawn shadow rays and secondary reflection/refraction rays. The goal of this part is to experiment with advanced rendering tools and to get hands-on experience with both lighting and geometry manipulation.
Extra credit points are available for extending your program to support additional features.

NOTE: Rendering very complicated scenes with many primitives (eg: the provided teapot mesh has thousands of triangles) can take a long time! Make sure that whenever possible you test and debug on simple scenes that only take a few seconds to render, rather than minutes or hours. One of the optional components of the assignment is to implement acceleration algorithms to speed up rendering. The reference solution `raytracer_sol` doesn't use any complicated acceleration structures, so rendering complex scenes like teapot.ray may take a while.

Steps:

- **12 pts** Implement the missing parts of `Raytracer::render` and `Raytracer::trace` for basic ray casting for all pixels in the image, using the camera location and the coordinates of each pixel. You can test this code by re-computing the pixel as the intersection of the ray and the view plane and testing that you obtain the same coordinates back.

- **5 pts** Implement `Sphere::localIntersect`. For this part you are required to calculate if a ray has intersected your sphere. Be sure to cover all the possible intersection scenarios(zero, one, and two points of intersection). Test your result by comparing the output of your depth algorithm with the example solution's results on the provided scenes. You can also render the spheres using the diffuse coefficients provided.

- **5 pts** Implement `Plane::localIntersect`. The implementation of this part is similar to the previous part in that you are calculating if a line has intersected your plane. Test this function in a similar way to the previous part (note that as you do new objects will appear).

- **6 pts** Implement `Mesh::intersectTriangle`. This function calculates the point of intersection of a ray with a triangle. The difference of this part when compared to the plane intersection is in the bounds you must check. Think back through the course and try to decide what equations might help you decide on which side of the bounding lines of the triangle the ray intersects. Test this part just like the last two parts; when triangle intersection is working properly, you should be able to see full meshes appear in your scenes.

- **17 pts** Implement the missing part of `Raytracer::shade` that does a lighting calculation to find the color at a point. You should calculate the ambient, diffuse, and specular terms. You should think of this part in terms of determining the color at the point where the ray intersects the scene. Test your results by comparing to the ground truth ones.

- **12 pts** Implement the shadow ray calculation in `Raytracer::shade` and update the lighting computation accordingly. You can think of this part as casting a second ray from a point of intersection where your original ray has intersected to determine which lights are contributing to the lighting at that point.

- **13 pts** Implement the secondary ray recursion for reflection in `Raytracer::shade`. Use the `rayDepth` recursion depth variable to stop the recursion process. (The default used in the solution is 10.) Update the lighting computation at each step to account for the secondary component. You can think of this part as an extended shadow ray calculation, recursively iterating to determine contributing light (and weighting newly determined light sources into the original pixel)

For those of you who want to explore, bonus marks will be given for implementing any of the below ideas or something of similar complexity at the discretion of the marker.

- Implementing `Conic::localIntersect` to enable intersections between the rays and generalized conical surfaces (`http://en.wikipedia.org/wiki/Conical_surface`) . Note that this requires detecting the bounding circles of the conics and accurately handling those (to get finite cylinders/cones/ellipsoid parts).

- Implementing a secondary ray recursion for refraction rays. Use the same recursion depth variable `rayDepth` as for reflection to stop the recursion process. Update the lighting computation at each step to account for the secondary component.

- Texturing - use the provided `Image` class to import textures and access the texture during ray-tracing to get a local diffuse color.

- Speed - consider speeding up your method using any of the space-partitioning methods discussed in class. The template provides a timer which you can use to compare your result to those of others and the ones in the solution.

- Gloss - use randomized direction estimation to account not only for specular but also glossy surfaces.

The comments in the template code above each section where are you required to add code contain the details of the specification. They also contain many hints. The recommended order of implementation is exactly the order we list the items above.

2 Post-Processing: Shaders (30)

In this part you will write shaders that should take the output images of the ray tracer as an input and apply two filters: first the negative of the image, and second a vertical blur filter. It is important to note that shaders errors are printed out to the terminal when the program is run. When you first run the program there are errors compiling because the shader files are empty. Once you have properly filled in the body the errors will go away (or be more meaningful).

Negative Filter: 15 pts

- Complete `Filter::processImage`. This section requires you to bind the variables that you want to use within the shaders. It should be noted that your original image is bound to GL_TEXTURE0. Look into the concept of shader *uniforms*:
  `http://www.opengl.org/wiki/Uniform_%28GLSL%29`
  `http://www.lighthouse3d.com/tutorials/glsl-tutorial/uniform-variables/`

- Complete `filter.vert` by writing a pass-through vertex shader, where a pass-through vertex shader takes as input the vertex position, and outputs the vertex position (it does nothing). You are doing this because the fragment/pixel shader cannot work unless there is geometry rendered for it to operate on. For further reading on vertex and fragment shaders see:
  `http://www.opengl.org/wiki/Rendering_Pipeline_Overview`.
  For this assignment you can ignore all shaders other than vertex and fragment shader.

  When programming a shader you use a programming language called GLSL (GL Shading Language). For this part you should use GLSL version 1.20, which is done by adding the line "#version 120" to the top of your shaders. Make sure that any code you find while trying to learn GLSL is the correct version (or lower) as there are substantial differences between the versions. In particular, this version will let you access the build in shader variables: gl_Vertex (in), gl_Position (out)(in the vertex shader) and gl_FragColor(out) (in the fragment shader). For an introduction to GLSL see:
  `http://www.davidcornette.com/glsl/glsl.html` and
  `http://nehe.gamedev.net/article/glsl_an_introduction/25007/`
  (this is also a good reference for vertex and fragment shaders)

- Complete `filter.frag` by writing a fragment (pixel) shader that computes the negative of the input image. $(r, b, g) \rightarrow (1 - r, 1 - g, 1 - b)$

Vertical Blur Filter: 15 pts

- Complete `filter.frag` by writing a fragment (pixel) shader that performs a vertical blur, which is a weighted sum of adjacent pixels in the y direction. If *(x, y)* is your current pixel, this requires you to sample the pixels of your input texture from *(x, y-3)* to *(x, y+3)* (ie you use 7 samples) and compute their weighted sum with following weights:
  (1/64, 6/64, 15/64, 20/64, 15/64, 6/64, 1/64)
  For more information on blur filters:
  `http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/`

- Your shader file should have an integer uniform variable to define which filter is applied. (based on filter member variable 'type' which is already set for you based on input flags '-#', where # is any number.)

For those of you who want to explore, bonus marks will be given, at the discretion of the marker, for implementing any of the below ideas:

- Upgrade your blur filter to an efficient gaussian blur. This is done by modifying the shader to perform different operations given some input uniform. A gaussian blur is a multi-directional blur that can be considered to be a horizontal blur composed with a vertical blur. `http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/`

- Create your own filter to apply to the image. It must be of similar or greater complexity as compared to the blur filter. If in doubt please ask one of the TAs or instructor.

Hand-in Instructions: You do not have to hand in any printed code. Create a README.txt file that includes your name, student number, and login ID for yourself, and any information you would like to pass on the marker. Create a folder called "assn4" under your "cs314" directory and put all the source files, your makefile, and your README.txt file there. **You MUST submit the images made by your program for the example scenes provided.** If you design extra-credit scenes, also submit the .ray file for them. Include any images that you used as texture maps. Do not use further sub-directories. The assignment should be handed in with the exact command:

```
handin cs314 assn4
```