Computing Science CMPT 361 **Spring 2016**
Instructor: Tom Shermer **Simon Fraser University**

Assignment #3 (30 marks)

Programming part due: April 6, Wednesday, at 11:59 pm.

Problem 1 (30 marks): Build a simple ray tracer

You will be building a ray tracer. It starts with a simple version that works on scenes consisting only of spheres, and gets expanded to include shadows, reflection, refaction, texture mapping, and supersampling. For bonus marks, its features can be expanded to handle arbitrary polygonal environments in an efficient manner.

In all cases, you may work with just a single light source. You can add any number of additional light sources if you wish, but it is not required.

You can begin with the skeleton code provided on our website.

Read the code and the comments carefully so that you would know exactly where the changes are to be made. Use "*make raycast*" to create an executable called raycast. The program requires between 2 to 8 **command line arguments**:

```
./raycast [–u | –d] step_max <options>
```

The option –u (user) should be used if you want to render a scene you created (to be defined in the function set_up_user_scene() in scene.cpp). To render the default scene, defined in the function set_up_default_scene() in scene.cpp, use –d. The next argument is an integer that specifies the number of recursion levels. For example, when step_max = 0, then no reflected or refracted rays are to be cast. The remaining arguments can be used to turn on different features of your raytracer. The default, if no flag is provided, is to turn the feature off. There are a total of six features:

- +s: inclusion of shadows
- +l: inclusion of reflection
- +r: inclusion of refraction
- +c: inclusion of chess board pattern
- +f: enabling diffuse rendering using stochastic ray generations
- +p: enabling super-sampling

For example, the following command produces a ray-traced image, with 5 levels of recursion, of the default scene with a chess board and it includes shadows, reflections, and refractions. No supersampling or stochastic ray generation are provided.

```
./raycast –d 5 +s +r +l +c +r
```

Note that the option arguments do not have to be specified in any particular order. Ways to specify the material properties of the spheres, the light source, and other scene parameters should be clear from the skeleton code sphere.h, sphere.cpp, raycast.cpp, and scene.cpp. You should define the sphere geometry. Many global variables are used in the program to simplify things for teaching purposes, but this is poor programming practice. Also, this is code from a previous instructor, and there is no guarantee that it will be bug-free. Please report any problems you experience to me, the TA, or the course mail list.

Your tasks: You should complete the following tasks in order.

1. [8 marks] Ray-sphere intersection and local reflectance: Complete functions (defined in the files sphere.h and sphere.cpp) which determine the first sphere that the ray intersects for visibility. Also, implement the ray tracer to support visible surface ray tracing using Phong's local illumination model, shown below, in trace.cpp. You should use the following model to color a pixel:

$$I = I_a k_a + \sum_{m=1}^{L} \frac{I_m}{a + b\Delta + c\Delta^2} (k_d(n \cdot l_m) + k_s(r_m \cdot v)^N)$$

This is the simplified Phong model where the light falloff factor is taken into consideration. Δ is the distance between the light source and the point on the object. As stated above, you may use L=1. Do not change the eye position or the placement of the image plane. The image size and its resolution can be altered as you wish.

You should complete this part of your program first and test it using:

```
./raycast [–u | –d] 0
```

2. [4 marks] Shadows: Next, implement *shadows* in your ray tracer. Add a shadow multiplier $S_m$for each light source *m* you have; let $S_m$be 0 if the object is in shadow from light *m*, and 1 if it is not.

$$I = I_a k_a + \sum_{m=1}^{L} \frac{S_m I_m}{a + b\Delta + c\Delta^2} (k_d(n \cdot l_m) + k_s(r_m \cdot v)^N)$$

This part can be tested using the command:

```
./raycast [–u | –d] 0 +s
```

3. [4 marks] Reflections: Augment your ray tracer to take into consideration of *directly reflected light*. The material property "reflectance" of a sphere (see sphere.h) specifies how much reflected light should contribute to the color of a pixel.

$$I = I_{Phong} + k_r * I_r$$

where I_{Phong} is the pixel color computed using the shadowed Phong model (in part 2 above), k_r is the object's reflectance, and I_r is the reflected light accumulated through recursive ray tracing. This part of your program can be tested using the command:

```
./raycast [–u | –d] 5 +s +l
```

for 5 levels of recursion, for example.

4. [4 marks] Chess board: Think about how to add a *shiny* planar 8×8 chess board to the default scene with spheres. (**Hint**: It is probably not necessary to create 64 polygons for this). The size of the board should be finite to save computational time. Place the chess board appropriately with respect to the spheres so that in the ray-traced image, one is able to see multiple interactions among them, e.g., an image of the chess board in the spheres, shadows on the board, etc. This part may be tested, e.g., using

```
./raycast [–u | –d] step_max +s +l +c
```

5. [4 marks]: Refraction: Make the three spheres in your default scene *refractive.* To implement refraction, you will probably need to modify some existing data structures given in the skeleton code. So make sure that you have done the previous parts correctly before attempting this option. You are free to choose material properties for the transparent spheres. Here is the lighting model, with I_t being the light hitting the surface due to refraction, and k_t being the *transmissivity* of the surface.

$$I = I_{Phong} + k_r * I_r + k_t * I_t$$

Place the scene description in the function void set_up_user_scene(). This part may be tested, e.g., using

```
./raycast –u step_max +s +l +r +c
```

6. [4 marks]: Diffuse reflections using stochastic ray generation: Generate better diffuse effects by tracing a number, say 5, of randomly generated reflected rays to simulate the diffuse->diffuse inter-reflection. You may declare a constant in global.h for the number of rays. Here, the average intensity of the diffuse reflected rays (I_d) is multiplied by the diffuse reflection coefficient (k_d) for the surface:

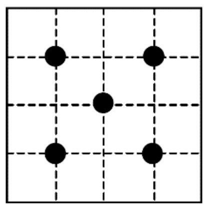
$$I = I_{Phong} + k_r * I_r + k_t * I_t + k_d * I_d$$

(This is the same k_d as in the basic Phong model.) This part can be tested, e.g., using the following command

```
./raycast [–u | –d] step_max +s +l +r +c +f
```

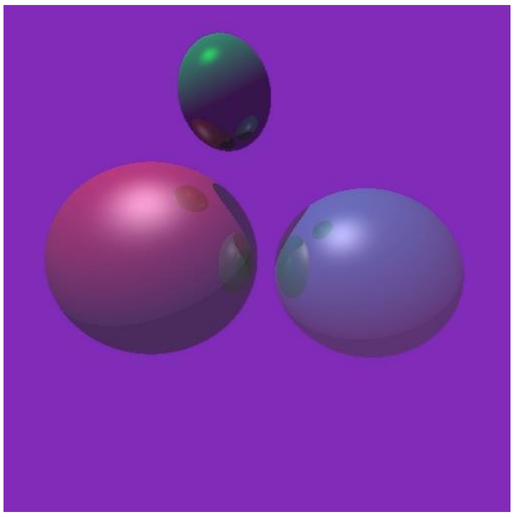
7. [2 marks]: Supersampling to reduce aliasing: Generate 5 rays per square pixel, arranged in a way as shown in the figure below. Compare the result you can obtain against those from the previous parts, where a single ray through the pixel center is generated per pixel. This part can be tested, e.g., using the following command

```
./raycast –u step_max +s +l +r +f +c +p
```



What to submit: All required C/C++ and header files, along with the Makefile, a README file in which you clearly state which features your have implemented, and two images (512 × 512) in PNG format. One image (default.png) is a screen capture of your rendering of the default scene, and the other one (mine.png) is for the scene you created (with the checker board and/or transparent spheres).

Your default.png should resemble the image below (no transparency or chess board):



Bonus problem [10 marks]:

(a) [5 marks] You are to ray trace a scene with two glass chess pieces along with a glass chess board extended to infinity (you can use the way you modeled the chess board from Problem 5). Each chess piece is given by a triangle mesh with 800 faces. Four mesh files are given in the chess_pieces directory.

The format SMF is used to represent the triangle meshes. In the SMF (Simple Mesh Format) format, a mesh is given by *a vertex list* followed by *a face list*. Each line in the vertex list starts with the character '*v*', followed by the *x*, *y*, and *z* vertex coordinates. Each line in the face list starts with the character '*f*', and followed by three integers indexing into the vertex list. The vertex indexes start with 1 and are given in counterclockwise order, viewed from the tip of the triangle's outward pointing normal. Any comments start with the character '#'. The very first line of the SMF file is of the form "# *n m*", where *n* is the number of vertices and *m* the number of faces in the mesh.

You are free to choose material properties to model your objects. You may even use appropriate textures that you can find. Note that this would involve a texture mapping process, which you may not want to delve into. The arrangement of the chess pieces should be somewhat nontrivial so that **inter-surface reflections and refractions** can be seen. You are free to choose specific configurations of your scene.

(b) [5 marks] The next key ingredient of your ray tracer that I am looking for is a way to **reduce unnecessary ray-polygon intersections or to speed up the process via other means**. You are free to choose the technique to implement, e.g., **octree or other bounding volume hierarchy**. In order to judge the effectiveness of your technique, your program should be able to report the total number of ray-polygon intersections performed by your program. You should report this number for a scene with a single chess piece standing on the chessboard, for both the naïve ray tracer without speedups and one that uses an octree and/or bounding boxes.

What to submit: All required source files, along with the Makefile, and a README. Place all the source files required for the bonus problem in their own directory called RayChess. In your README file, you should explain briefly the speedup technique you employed and report the improvements it is able to achieve. Submit a final, most impressive image of your ray-traced scene called chess_scene.png. Your Makefile should produce an executable called raychess, by executing make raychess. Any command line arguments needed should be explained in the README.