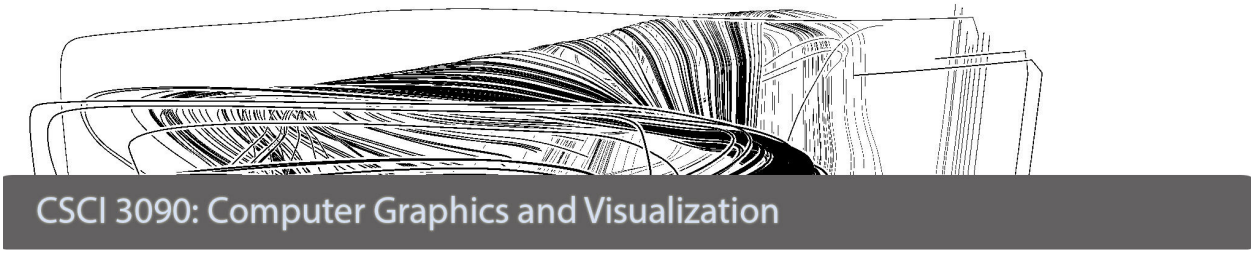


CSCI 3090U Assignment Three
Ray Tracing (value: 10%)
Due: March 20, 201 (11:59pm)

Introduction

This assignment gives you some experience with building your own rendering engine using ray tracing. In this assignment you will not produce a complete ray tracer, but will gain some experience with both basic and advanced techniques through augmenting supplied code. The ray tracer will be built in phases, each expanding the framework code kindly provided to us by Dr. Tobias Isenberg, University of Groningen. The expected outputs for each phase of the development are indicated below. You need only submit your final code and final output. However, your report should refer to the functions which generate the necessary calculations for each section in your report.

While we are building this project on framework code, please submit your complete working code (not only the classes you change) so that your raytracer can be easily compiled and tested. You are free to disregard the framework code and build your own raytracer in your preferred language, as long as it satisfies the requirements laid out in the following sections.

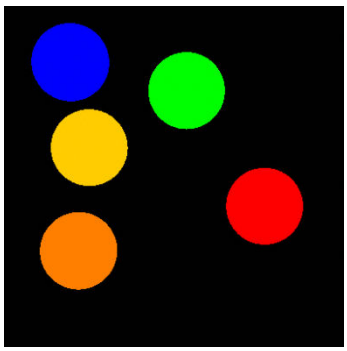
Framework Code

The framework code for our ray tracer can be found on Blackboard as raytracer-visualstudio.zip or raytracer-eclipse.zip, depending on your IDE of choice. These include either a Visual Studio Solution file or Eclipse project file, but you may port it to any IDE you prefer. See the readme.txt file for a description of each file within the archive.

Our raytracer will take as input a YAML file (a simplified form of XML) which describes a scene. It will output a .png graphic. If you compile the framework code and run it with:

```
Raytracer.exe scene01.yaml
```

You should get the `scene01.png` output, which is shown at right:

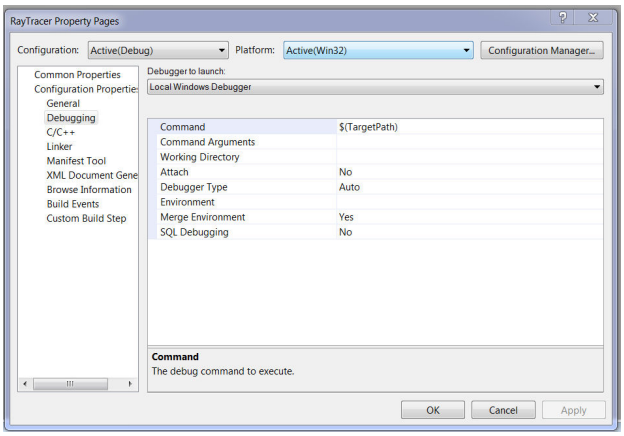


The command line syntax for our raytracer is:

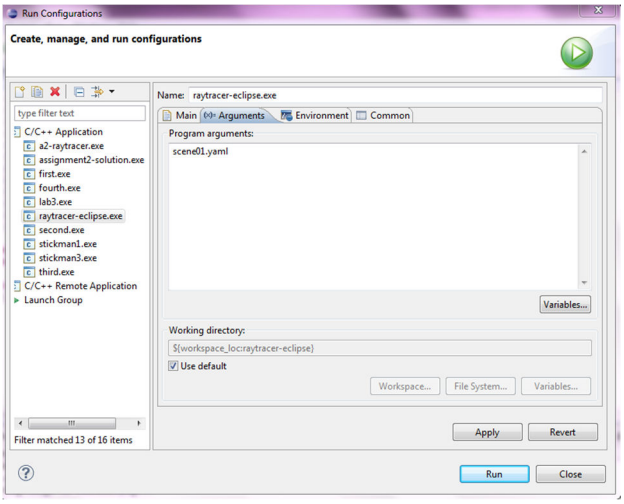
```
Raytracer.exe <yaml input file> [png output file] [width height]
```

Where both [png output file] and [width height] are optional, but the output file must be specified if width and height are specified. Width and height are optional (a default size will be rendered if they are omitted).

Recall that you can specify command line arguments from within your IDE or using the command line. In Visual Studio:



In Eclipse:



Look at the source code of the classes and try to understand the program. Of particular importance is the file `triple.h` which defines mathematic operators on vectors, points, and colors. The actual raytracing algorithm is implemented in `scene.cpp`. The YAML based scene files are parsed in `raytracer.cpp`. Look at the included `readme.txt` file for a description of the source files.

YAML Input Files

YAML input files describe a scene: the objects, their material properties (colour, reflectance, index of refraction) and the relative geometry. The end-of-line characters in the Visual Studio distribution are in Windows Format, in the Eclipse version they are in UNIX format. When using Eclipse on Windows, the UNIX end-of-line character is expected. If you edit the YAML files, be sure to use an editor which preserves the end-of-line encoding, such as Notepad++ (freeware).

Intersections and Normals [15 marks]

In this part your program will produce a first image of a 3D scene using a basic ray tracing algorithm, for now without illumination, shadows, or reflection. The intersection calculation together with normal calculation will be the groundwork for the illumination of the next section.

Your raytracer only needs to support spheres. Each sphere is given by its midpoint, its radius, and its surface parameters (color and parameters for Phong shading, to be implemented in the next section).

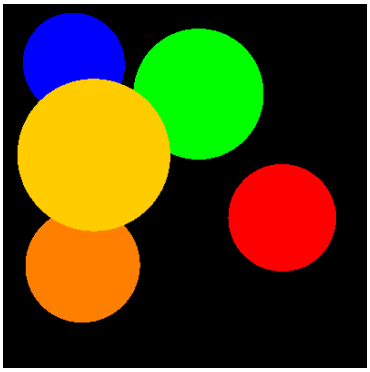
A white **point** light source is given by its position (x,y,z) and color. In the example `scene01.yaml` a single white light source is defined.

The viewpoint is given by its position (x,y,z). To keep things simple the other view parameters are static: the image plane is at z=0 and the viewing direction is along the negative z-axis.

The scene description is read from a YAML file.

Tasks:

1. [10 points] Implement the intersection calculation for the sphere. Extend the function `Sphere::intersect()` in the file `sphere.cpp`. The resulting image should be similar to the following image:



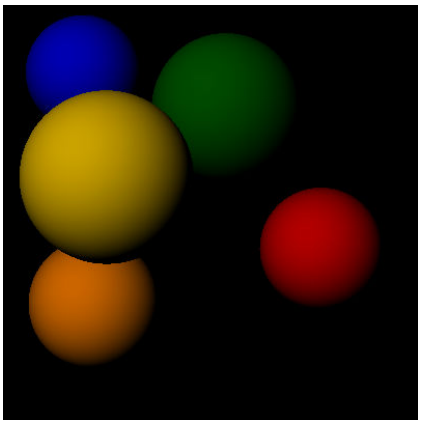
2. [5 points] Implement the normal calculation for the sphere. To this end, complete the function `Sphere::intersect()` in the file `sphere.cpp`. Because the normal is not used yet, the resulting image will not change. For a sphere the normal computation is quite simple, it is just the vector from the center of the sphere to the point on the sphere. This vector need to be normalized.

Phong Illumination [20 marks]

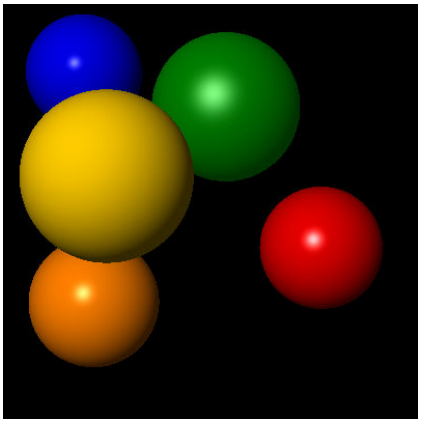
In this section you will calculate the Phong illumination at points on the surfaces of the scene. Note that we do not yet have a *recursive* ray tracer (i.e. the Whitted terms I_r and I_t for reflection and refraction are not included).

Tasks:

1. [10 marks] Implement the diffuse term of Phong's lighting model to obtain simple shading. Modify the function `Scene::trace(Ray)` in the file `scene.cpp`. This step requires a working normal calculation. The resulting image should be similar to the following image:



2. [10 marks] Extend the lighting calculations with the ambient and specular parts of the Phong model. This should yield the following result:



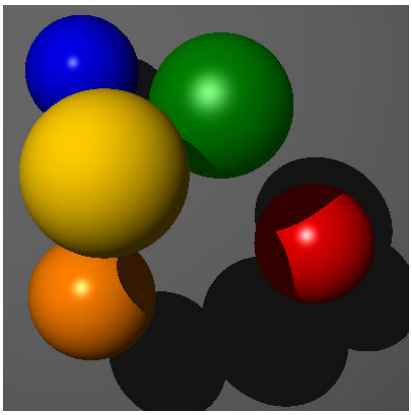
You can also test your implementation with `scene02.yaml`.

Shadows [20 marks]

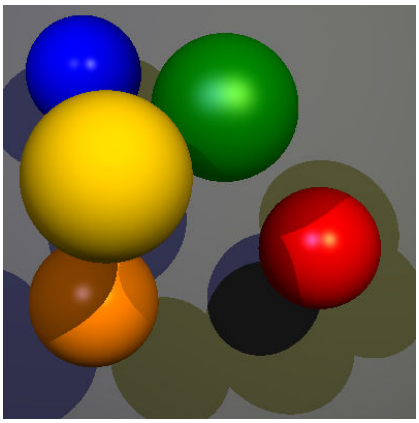
In this part you will implement a global lighting simulation. Using recursive ray tracing the interaction of the lights with the objects is determined. The program should be able to handle multiple coloured light sources and shadows.

Tasks:

1. [15 marks] Extend the lighting calculation in `Scene::trace(Ray)` such that it produces hard shadows. First make it configurable whether shadows should be produced (e.g., `Shadows: true`). Recall that the general approach for producing shadows is to test whether a ray from the light source to the object intersects other objects. Only when this not the case, the light source contributes to the lighting. **For the following result `scene03.yaml` was used**, which is just `scene01.yaml` with a large background sphere added to the scene so that shadows will be visible:



2. [5 points] Now loop over all light sources (if you didn't do that already) and use their color in the calculation. For the following result image two light sources were used: bluish light (.4,.4,.8) at [-200,600,1500], a yellowish light (.8,.8,.4) at [600,600,1500]. **Use `scene04.yaml` to test with both lights:**

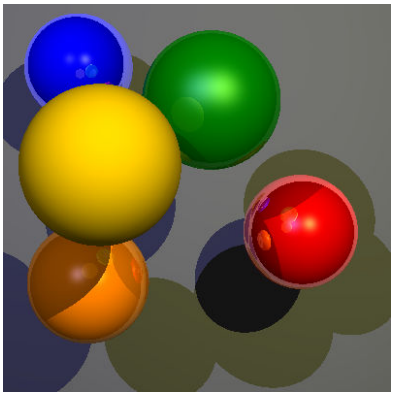


Reflection [25 Marks]

In this step you will add propagation of rays through reflection. To do this, we need to know the reflectivity of the materials. You'll also parse refraction material parameters so you are prepared to do the bonus section.

Tasks:

1. [5 marks] Extend the `Material` class with three parameters: `reflect`, `refract`, and `eta`. Also extend the function for parsing material descriptions in the `Raytracer` class. **Test your changes with the scene file `scene05.yaml`.**
2. [20 marks] Implement reflections, by recursively continuing rays in the direction of the reflection vector, using the `reflect` parameter. You may decide on a stopping condition. The easiest is to stop after a fixed number of reflection steps or if the ray leaves the scene. Stopping when intensity change falls below a threshold is also possible. Here is an example result. Note the yellow and red colours picked up on the orange sphere, and that the yellow sphere is not reflective:



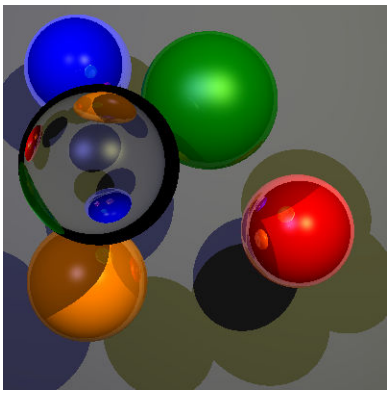
Extensions [20 marks + up to 20 bonus]

There are several options to extend your raytracer. Each option is worth 20 marks. **You must do one**, and you may do a second one for up to 20 bonus marks.

Options:

1. Implement refraction, by recursively continuing rays in the direction of the transmission vector, using the parameters `refract` and eta, where eta is the index of refraction of the material. You may assume the material surrounding the spheres is air (eta=1.00**). Use `scene06.yaml` to test (yellow sphere no longer reflects light itself).** Don't forget that the ray will enter the sphere, then (possibly) exit it – so, unless total internal reflection occurs, each refraction ray has two direction changes due to refraction.

Example output:



2. Add **2** additional types of geometry to your scene. Examples include quads, cones, triangles, cylinders. Submit your new YAML scene file as well as the output image. You will have to create new classes similar to Sphere.cpp and implement the intersection detection.
3. Implement soft shadows. To do this you will have to give your light source an area (instead of a point source). Then, for each intersection, generate a number of rays that strike the light source in random positions. In areas of penumbra, some of these rays will reach the light, others will be blocked (in shadow). Average the values returned by the rays to determine a value of the shadow. This should be a number between 0 and 1 which can be used to weight the diffuse and specular components of the object's colour.

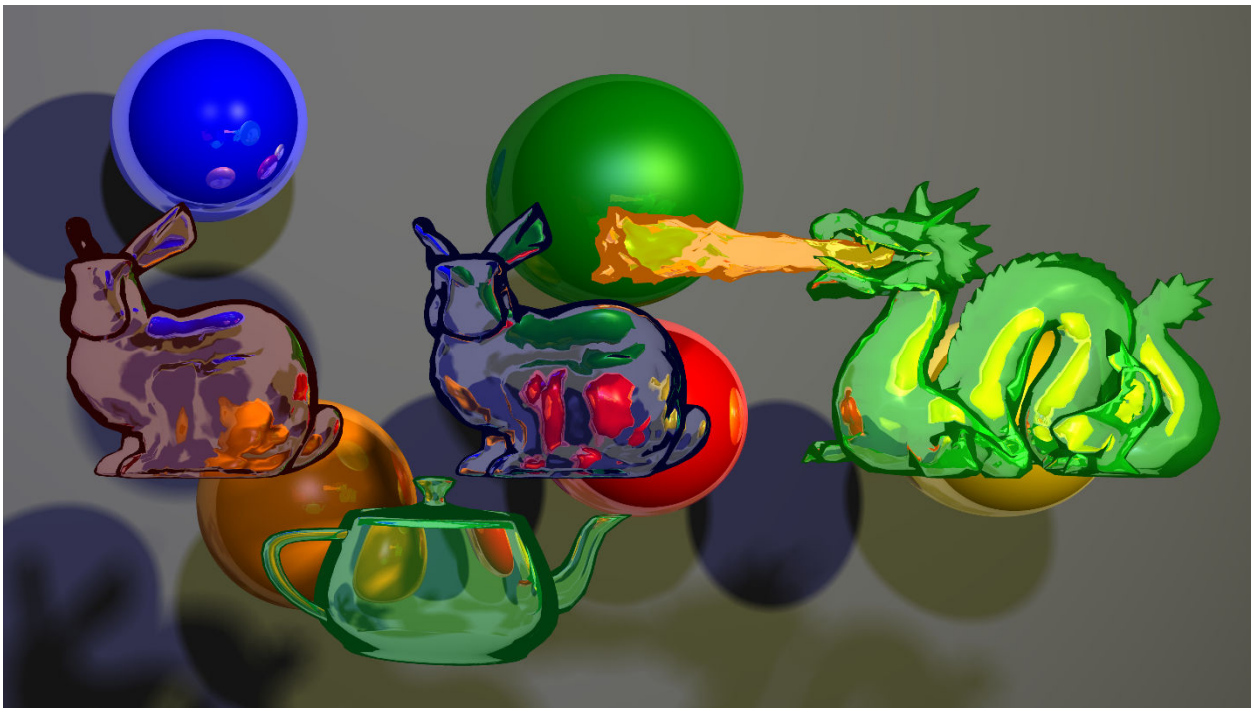
Rendering in UHD

The J123 Lab now has a Samsung 4K display. You may attempt to render your images for this resolution (3840x2160). Your laptop may not be able to drive this large display (we are testing this), but you can make your image available via a URL (e.g. a Dropbox link) and display it to the UHD display. You will be provided access to this display for testing during lab time and also during open hours of J123.

Hall of Fame

To demonstrate the skill of UOIT students, the best raytracing projects (going beyond the minimum assignment requirements) will be invited to be displayed (with student permission) on the course hall of fame website (which is in development). Having your work featured here would be a great footnote to your CV!

Here is the 2014 winner, by Wesley Taylor (full image is 4k):



Tip on Intersection Detection for Recursive Ray Tracing

Due to the intersection calculation precision, sometimes when a ray intersects with a sphere (say, Sphere_A), the resulting position can be just slightly inside the sphere. If you use this as the origin of a refraction, reflection, or shadow ray, you are likely to intersect with Sphere_A again (receiving a very small parameter value is a clue that this has happened). This can result in a speckled appearance in your ray tracing as some rays will not propagate properly.

There are two things to do to ensure correct ray intersections.

First: don't allow a negative parameter value - this means the intersection is behind the origin!

Second: Jiggle the rays! To ensure your propagated ray actually exits Sphere_A, you need to move the actual origin point of the new rays to be outside. To do this, we recommend "jiggling" the origin point. This can be done with the ray.at(t) function, which gives location along a ray "at" the given parameter value.

Thus ray.at(2^-32) is the location at t=2^-32 along the ray.

By finding ray.at(2^-32) *then using this as the adjusted origin for the ray*, the rounding errors can be avoided. Here is a code snippet for reflection:

```
//add reflected ray
Vector Ref = /* YOUR calculation for reflection direction here */
Ref.normalize();
Ray reflected_ray = Ray(hit, -Ref);

//jiggle the hit point to make sure it is outside of the sphere
Point hit_jiggle = reflected_ray.at(pow(2,-32));
// reset the reflected ray to start at the jiggled origin
reflected_ray = Ray(hit_jiggle,-Ref);

// continue with reflection intersections etc.
```

General Assignment Guidelines

This is an individual assignment. While you may discuss your progress with your classmates, each member of the class is expected to hand in their own work. Note that we do have copies of assignment submissions from previous course offerings, and reserve the right to compare your work for originality.

Please include this statement on the cover page of your report:

"I, <insert name>, certify that this work is my own, submitted for CSCI 3090U in compliance with the UOIT Academic Integrity Policy."

In this course, you MAY NOT review, copy or otherwise use any graded academic assignments from prior semesters or other sections of this course, in written, electronic, or verbal form, used in whole or part, including formatting of any assignment. While code re-use is a useful part of efficient programming, for this course the goal is for you to gain personal experience solving programming problems. Therefore, you may not directly use source code copied from the web. **Reference any sources you use to create your programs (in the comments) and report (in the report).**

Submission Guidelines

Submit (on Blackboard) your appropriately documented source code, any scene files which you may have modified, and a clearly written report.

If your final output is not produced with scene05.yaml, please specify which scene file you used so that we may reproduce it.

Please hand in the following with every assignment in this course:

- A written report (Word or PDF) describing your solution, highlighting any features of your source code which you would like to bring to the instructor's attention, including

images produced by your raytracer, answering any questions posed in the assignment handout, and confirming the academic integrity statement. This report should be well-formatted, clearly organized, and use correct grammar and spelling.

- If you know there are errors in your calculations, please describe the problems to receive partial marks (e.g. "The surface normals are not calculated correctly because the specular highlight is shown in the wrong position."). Provide correct pseudocode.
- Source code which can be compiled in Windows or Linux. C++ or Java ONLY are acceptable. If your code has special compilation requirements, include them in a `readme.txt` file.
- Any high-resolution images you produce may be submitted separately. A low-resolution version should be embedded in your report.