T-111.4310: Interactive Computer Graphics Fall 2012

Programming Assignment 4: Ray Casting and Tracing

Due December 9th at 23:59.

In this assignment, you will first implement a basic ray caster. As seen in class, a ray caster sends a ray for each pixel and intersects it with all the objects in the scene. Your ray caster will support orthographic and perspective cameras as well as several primitives (spheres, planes, and triangles). You will also have to support several shading modes and visualizations (constant and Phong shading, depth and normal visualization). Then you will extend the implementation with recursive ray tracing with shadows and reflective materials.

The remainder of this document is organized as follows:

1 Getting Started

Note that this assignment is a lot of work. Please start as early as possible.

Run the sample solution `example.exe` in the `exe` folder as follows:

```
example.exe -input scenes1/scene1_01.txt -size 200 200 -output output1_01.png
```

This will generate an image named `output1_01.png`. We'll describe the rest of the command-line parameters later. When your program is complete, you will be able to render this scene as well as the other test cases given below.

2 Summary of Requirements

Here is a list of the core requirements for this assignment. They will be explained in more detail later in this document.

- Generate rays from an orthographic and a perspective camera.
- Mapping of pixel coordinates to the canonic image space $[-1, 1]^2$.
- Intersecting ray with
  - plane
  - sphere
  - group
  - triangle
- Shading
  - distant light
  - point light
- Shadows
- Phong material
  - diffuse, specular
  - ambient
- Mirror reflection
  - Recursive ray tracing
  - Computation of reflection direction
- Supersampling inside pixel
  - uniform random
  - jittered
  - regular

The different shape primitives are designed in an object hierarchy. A generic `Object3D` class serves as the parent class for all 3D primitives. The individual primitives (such as `Sphere`, `Plane`, `Triangle`, `Group`, and `Transform`) are subclassed from the generic `Object3d`. You should provide the implementations for intersecting a ray with each individual primitive. To generate rays from pixel coordinates you have to implement the appropriate methods in `Camera.h`.

You will implement a Phong shading model in `PhongMaterial::shade`. It may be best to start simple and only provide the diffuse component first. When you have that working, continue refining the model.

Your code will be tested using a script on all the test cases below.

3 Starter Code and Debugging

Compile the project in debug mode and run it. We have configured the project to run in the `exe` folder. The input parameters for the executable have been set to "–input scenes1/scene1_01.txt -size 200 200 -output out1/output1_01.png". The -input parameter specifies the scene description, -output specifies the output image name.

You can modify the input parameters for the debugger by right-clicking on the Visual Studio solution `four` in the solution explorer. Navigate to `Properties->Configuration Properties->Debugging`. The `Command Arguments` field specifies the input parameters for the program. The `Working Directory` field specifies the working directory.

The starter code can take a number of command line arguments to specify the input file, output image size and output file. Make sure the examples below work, as this is how we will test your program.

4 Implementation Notes

4.1 Quick start guide

The `render` function in `Foundation/main.cpp` is the main driver routine for your application. Some of the function has been marked with `YOUR CODE HERE` comments to indicate what should be implemented.

To get started with the project, you need to implement the methods

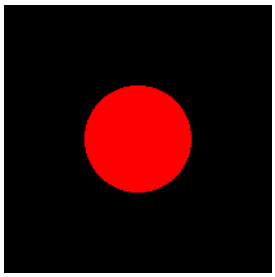
- `Camera::normalizedImageCoordinateFromPixelCoordinate`
- `OrthographicCamera::generateRay`
- `RayTracer::traceRay`

for generating rays. Notice that the scene `scenes1/scene1_01.txt` uses an orthographic camera. When you want to render scenes with a perspective camera, you must implement the generateRay method for `PerspectiveCamera`. Furthermore, you must implement the intersection routines

- `Group::intersect`
- `Sphere::intersect`

The class `Group` is simply a container of `Object3d` instances, each of which is quaranteed to implement the `intersect` routine. Remember to set the pixel color for the image with `Image::setVec4f` once you have computed the sample color in the `render` function.

Now you should have produced an image out1/output1_01.png which looks like this



We provide you with a `Ray` class and a `Hit` class to manipulate camera rays and their intersection points, and an abstract `Material` class. A `Ray` is represented by its origin and direction vectors. The `Hit` class stores information about the closest intersection point and normal, the value of the ray parameter t and a pointer to the `Material` of the object at the intersection. The `Hit` data structure must be initialized with a very large t value (try `FLT_MAX`). It is modified by the intersection computation to store the new closest t and the `Material` of intersected object. We provide a skeleton `PhongMaterial` class, which you should fill in with your implementations.

4.1.1 Object3d hierarchy

The `FULLY IMPLEMENTED` virtual `Object3D` class (a virtual class in C++ is like an abstract class in Java). It only provides the specification for 3D primitives, and in particular the ability to be intersected with a ray via the virtual method: `virtual bool intersect( const Ray& r, Hit& h, float tmin ) = 0;`

Since this method is pure virtual for the `Object3D` class, the prototype in the header file includes '`= 0`'. This '`= 0`' tells the compiler that `Object3D` won't implement the method, but that subclasses derived from `Object3D` must implement this routine. An `Object3D` stores a pointer to its `Material` type. For this assignment, materials are very simple and consist of a single color. The `Object3D` class has:

- a default constructor and destructor
- a pointer to a `Material` instance, and get and set methods
- a pure virtual intersection method

The derived class `Sphere`, a subclass of `Object3D`, additionally stores a center point and a radius. The `Sphere` constructor will be given a center, a radius, and a pointer to a `Material` instance. The `Sphere` class implements the virtual `intersect` method mentioned above (but without the '`= 0`'): `virtual bool intersect(const Ray& r, Hit& h, float tmin);`. You have to provide the implementation for the `intersect` method.

With the `intersect` routine, we are looking for the closest intersection along a `Ray`, parameterized by t. `tmin` is used to restrict the range of intersection. If an intersection is found such that $t >$ `tmin` and t is less than the value of the intersection currently stored in the `Hit` data structure, `Hit` is updated as necessary. Note that if the new intersection is closer than the previous one, both t and `Material` must be modified. It is important that your intersection routine verifies that $t >=$ `tmin`. `tmin` depends on the type of camera (see below) and is not modified by the intersection routine.

The derived class `Group`, also a subclass of `Object3D`, stores an array of pointers to `Object3D` instances. For example, it will be used to store the entire 3D scene. You'll need to implement the `intersect` method of `Group` which loops through all these instances, calling their intersection methods. The `Group` constructor takes as input the number of objects under the group. The group includes a method to add objects: `void addObject(int index, Object3D* obj);`

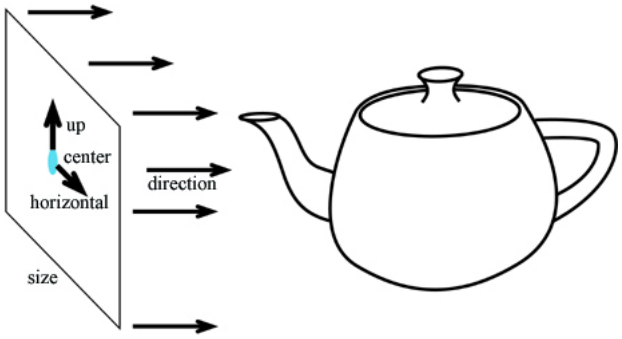
4.1.2 Orthographic and perspective camera

The virtual `Camera` class is derived by `OrthographicCamera` and `PerspectiveCamera`. The `Camera` class has two pure virtual methods:

- `virtual Ray generateRay( const Vec2f& point ) = 0;`
- `virtual float getTMin() const = 0;`

The first is used to generate rays for each screen-space coordinate, described as a `Vec2f`. The direction of the rays generated by an orthographic camera is always the same, but the origin varies. The `getTMin()`

method will be useful when tracing rays through the scene. For an orthographic camera, rays always start at infinity, so `tmin` will be a large negative value (try `FLT_MIN`). However, you will also implement a perspective camera and the value of `tmin` will be zero to correctly clip objects behind the viewpoint. You should provide the implementations for `generateRay` in both the orthographic and the perspective camera. In addition, you have to implement the `normalizedImageCoordinateFromPixelCoordinate` method in the base class `Camera`.



An orthographic camera is described by an orthonormal basis (one point and three vectors) and an image size (one float). The constructor takes as input the center of the image, the direction vector, an up vector, and the image size. The input direction might not be a unit vector and must be normalized. The input up vector might not be a unit vector or perpendicular to the direction. It must be modified to be orthonormal to the direction. The third basis vector, the horizontal vector of the image plane, is deduced from the direction and the up vector (hint: remember your linear algebra and cross products). The origin of the rays generated by the camera for the screen coordinates, which vary from $(-1, -1) \rightarrow (1, 1)$, should vary from: $\textbf{center} - (size * \textbf{up})/2 - (size * \textbf{horizontal})/2 \rightarrow \textbf{center} + (size * \textbf{up})/2 + (size * \textbf{horizontal})/2$

The camera does not know about screen resolution. Image resolution is handled in your main loop. For non-square image ratios, just crop the screen coordinates accordingly.

4.2 Depth and normal vector rendering

Implement a second rendering style to visualize the depth t of objects in the scene. Two input depth values specify the range of depth values which should be mapped to shades of gray in the visualization. Depth values outside this range should be clamped. Most of the implementation is already given in the `render` function. Add another input argument for the executable `-depth 9 10 depth_file.png` to specify the output file for this visualization. See the scripts `test_cases1.cmd` and `test_cases2.cmd` for good depth values for each scene.

Update your sphere intersection routine to pass the correct normal to the `Hit`.

Implement the new rendering mode, normal visualization. Try the visualization of surface normals by adding another input argument for the executable `-normals <normal_file.png>` to specify the output file for this visualization (see examples below).

4.3 Plane and triangle intersection

At this point it might be a good idea to implement the `intersection` methods for `planes` and `triangles`. To test the plane implementation you can use the scene description `scenes1/scene2_06_plane.txt` and `scenes1/scene2_07_sphere_triangles.txt` for the triangle implementation.

Use the method of your choice to implement the ray-triangle intersection: general polygon with in-polygon test, barycentric coordinates, etc. We can compute the normal by taking the cross-product of two edges, but note that the normal direction for a triangle is ambiguous. We'll use the usual convention that counter-clockwise vertex ordering indicates the outward-facing side. If your renderings look incorrect, just flip the cross product to match the convention.

4.4 Shading

4.4.1 Diffuse shading and distant light

- Provide an implementation for `DirectionalLight::getIncidentIllumination` to get directional lights.
- Implement diffuse shading in `PhongMaterial::shade`. You will extend the implementation in the next section.
- Extend `RayTracer::traceRay` to take the new implementations into use. The class variable `m_scene` (in `RayTracer`) is a pointer to a `SceneParser`. Use the `SceneParser` to loop through the light sources in the scene. For each light source, ask for the incident illumination with `Light::getIncidentIllumination`.

Diffuse shading is our first step toward modeling the interaction of light and materials. Given the direction to the light $\mathbf{L}$ and the normal $\mathbf{N}$ we can compute the diffuse shading as a clamped dot product:

$$d = \begin{cases} \mathbf{L} \cdot \mathbf{N} & \text{if } \mathbf{L} \cdot \mathbf{N} > 0 \\ 0 & \text{otherwise} \end{cases}$$

If the visible object has color $c_{object} = (r, g, b)$, and the light source has color $c_{light} = (L_r, L_g, L_b)$, then the pixel color is $c_{pixel} = (rL_rd, gL_gd, bL_bd)$. Multiple light sources are handled by simply summing their contributions. We can also include an ambient light with color $c_{ambient}$, which can be very helpful for debugging. Without it, parts facing away from the light source appear completely black. Putting this all together, the formula is:

$$c_{pixel} = c_{ambient} * c_{object} + \sum_i \left[clamp(\mathbf{L}_i \cdot \mathbf{N}) * c_{light} * c_{object} \right]$$

Color vectors are multiplied term by term. Note that if the ambient light color is $(1, 1, 1)$ and the light source color is $(0, 0, 0)$, then you have constant shading.

You will also implement two visualization modes. One mode will display the distance t of each pixel to the camera. The other mode is a visualization of the surface normal. For the normal visualization, you will simply display the absolute value of the coordinates of the normal vector as an (r, g, b) color. For example,

a normal pointing in the positive or negative z direction will be displayed as pure blue (0, 0, 1). You should use black as the color for the background (undefined normal).

4.4.2 Phong shading and point lights

- Implement Phong shading in `PhongShader::shade`
- Implement `PointLight::getIncidentIllumination`

Previously, scenes only included one type of light source, directional lights, 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. Now, you have point light sources, so the distance from the surface to the light source will be important. The `getIllumination` method in `PointLight`, which you should implement, will return the scaled light color with this distance factored in.

The shading equation in the previous section for diffuse shading can be written

$$I = A + \sum_i D_i,$$

where

$$A = c_{ambient} * c_{object\_diffuse}$$
$$D_i = c_{light_i} * clamp(\mathbf{L}_i \cdot \mathbf{N}) * c_{object\_diffuse}$$

i.e., the computed intensity was the sum of an ambient and diffuse term.

Now, for Phong shading, you will have

$$I = A + \sum_i \left[D_i + S_i\right]$$

where S_i is the specular term for the ith light source:

$$S_i = c_{light_i} * k_s * (clamp(\mathbf{H}_i \cdot \mathbf{N}))^e$$

Here, k_s is the specular coefficient, $\mathbf{H}_i$ is the half-angle vector, $\mathbf{N}$ is the surface normal, and e is the specular exponent. k_s is the `specularColor` parameter in the `PhongMaterial` constructor, and e is the `exponent` parameter. The half-angle vector $\mathbf{H}_i$ is the vector that bisects the vector pointing towards the light and the vector pointing towards the camera.

4.5 Recursive Rays

Next, you will add some global illumination effects to your ray caster. Because you cast secondary rays to account for shadows, reflection (and refraction for extra credit), you can now call it a ray tracer. You should extend the implementation of `RayTracer::traceRay` to account for shadows and reflections.

To compute cast shadows, you will send rays from the visible point to each light source. If an intersection is reported, the visible point is in shadow and the contribution from that light source is ignored. Note that shadow rays must be sent to all light sources. To add reflection (and refraction) effects, you need to

send secondary rays in the mirror (and transmitted) directions, as explained in lecture. The computation is recursive to account for multiple reflections (and or refractions).

The method `RayTracer::traceRay`, given a ray, computes the color seen from the origin along the direction. This computation is recursive for reflected (or transparent) materials. We therefore need a stopping criterion to prevent infinite recursion. The parameter `bounces` is checked by the starter code for this purpose. You only need to remember to decrement its value for a recursive call. The parameter `refr_index` is the index of refraction for a material.

```
Vec3f traceRay( Ray& ray, float tmin, int bounces, float refr_index, Hit& hit ) const;
```

Add support for the new command line arguments: `-shadows`, which indicates that shadow rays are to be cast, and `-bounces`, which controls the depth of recursion in your ray tracer. The arguments are parsed by the starter code.

Implement cast shadows by sending rays toward each light source to test whether the line segment joining the intersection point and the light source intersects an object. If there is an intersection, then discard the contribution of that light source. Recall that you must displace the ray origin slightly away from the surface, or equivalently set `tmin` to some ϵ.

Implement mirror reflections by sending a ray from the current intersection point in the mirror direction. For this, you should implement the function:

```
Vec3f mirrorDirection( const Vec3f& normal, const Vec3f& incoming );
```

Trace the secondary ray with a recursive call to `traceRay` using a decremented value for the recursion depth. Modulate the the returned color with the reflective color of the material at point.

4.6 Anti-aliasing

Next, you will add simple anti-aliasing to your ray tracer. You will use supersampling and filtering to alleviate jaggies and Moire patterns.

For each pixel, instead of directly storing the colors computed with `RayTracer::traceRay` into the `Image` class, you'll compute lots of color samples (computed with `RayTracer::traceRay`) and average them.

You are required to implement simple box filtering with uniform, regular, and jittered sampling. To use a sampler provide one of the following as additional command line arguments:

- `-uniform_samples <num_samples>`
- `-regular_samples <num_samples>`
- `-jittered_samples <num_samples>`

The box filter has already been implemented. You should implement the sampling in

- `UniformSampler::getSamplePosition`

- `RegularSampler::getSamplePosition`
- `JitteredSampler::getSamplePosition`

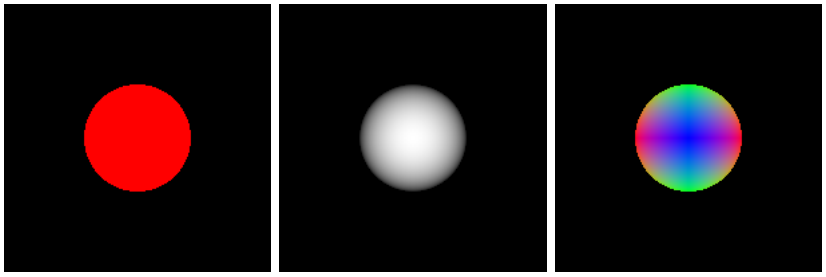
In your rendering loop (`render` in `main.cpp`), cast multiple rays for each pixel as specifed by the `<num_samples>` argument (the innermost for loop iterates as many times as the `num_samples` specifies). If you are sampling uniformly, the sample rays should be distributed in a uniform grid pattern in the pixel. If your jittering samples, you should add a random offset (such that the sample stays within the appropriate grid location) to the uniform position. To get the final color for the pixel, simply average the resulting samples.

At this point you should be able to successfully produce all of the test case images below. Try it out, then backup your code (or tag it in your source control repository).

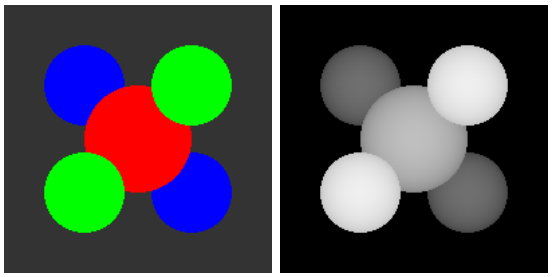
5 Test Cases

Your assignment will be graded by running a script that runs these examples below. Make sure your ray caster produces the same output if you want to receive full credit.

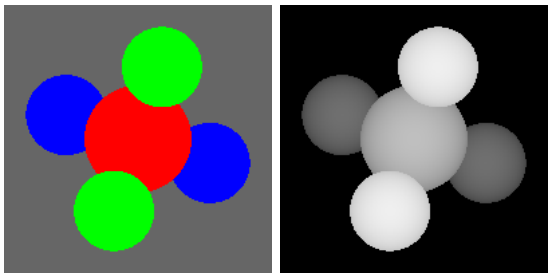
```
example.exe -input scene1_01.txt -size 200 200 -output output1_01.png -depth 9 10 depth1_01.png
-normals normal1_01.png
```



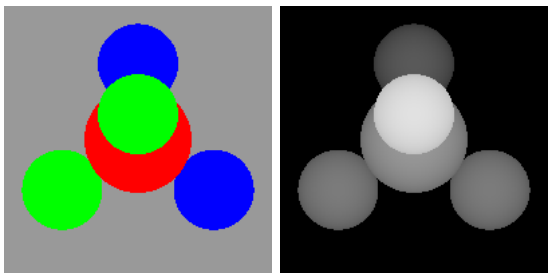
```
example.exe -input scene1_02.txt -size 200 200 -output output1_02.png -depth 8 12 depth1_02.png
```



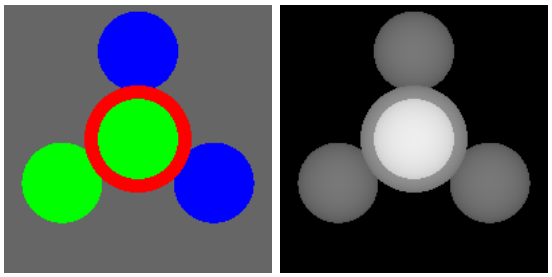
```
example.exe -input scene1_03.txt -size 200 200 -output output1_03.png -depth 8 12 depth1_03.png
```



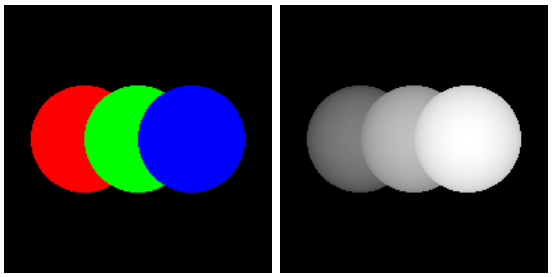
```
example.exe -input scene1_04.txt -size 200 200 -output output1_04.png -depth 12 17 depth1_04.png
```



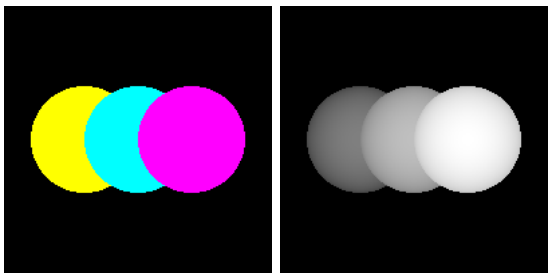
```
example.exe -input scene1_05.txt -size 200 200 -output output1_05.png \
-depth 14.5 19.5 depth1_05.png
```



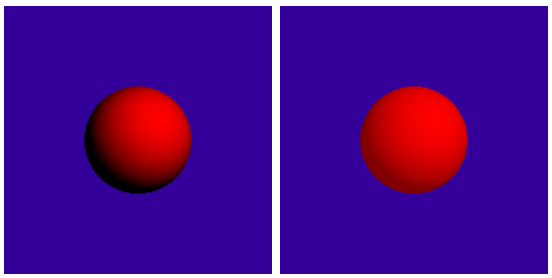
```
example.exe -input scene1_06.txt -size 200 200 -output output1_06.png -depth 3 7 depth1_06.png
```



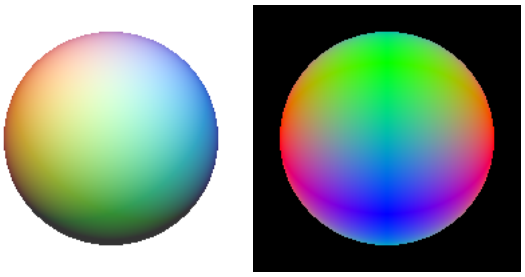
```
example.exe -input scene1_07.txt -size 200 200 -output output1_07.png \
-depth -2 2 depth1_07.png
```



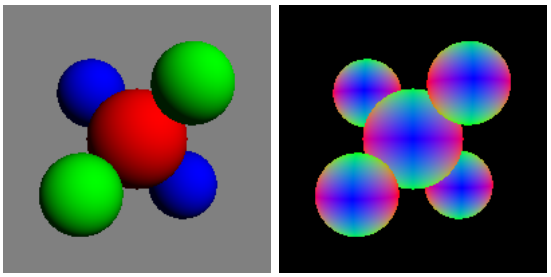
```
example.exe -input scene2_01_diffuse.txt -size 200 200 -output output2_01.png
example.exe -input scene2_02_ambient.txt -size 200 200 -output output2_02.png
```



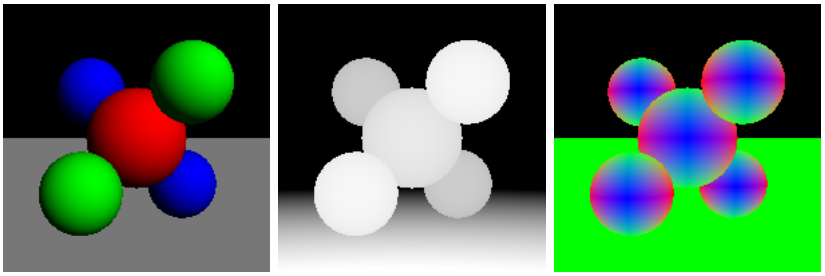
```
example.exe -input scene2_03_colored_lights.txt -size 200 200 -output output2_03.png \
-normals normals2_03.png
```



```
example.exe -input scene2_04_perspective.txt -size 200 200 -output output2_04.png \
-normals normals2_04.png
```

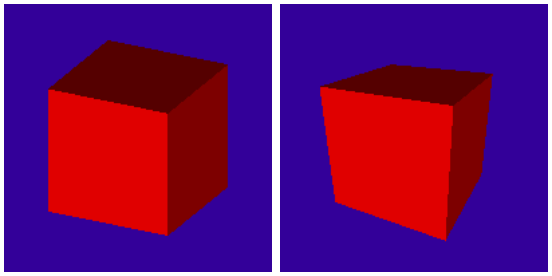


```
example.exe -input scene2_06_plane.txt -size 200 200 -output output2_06.png \
-depth 8 20 depth2_06.png -normals normals2_06.png
```



```
example.exe -input scene2_08_cube.txt -size 200 200 -output output2_08.png
example.exe -input scene2_09_bunny_200.txt -size 200 200 -output output2_09.png
example.exe -input scene2_10_bunny_1k.txt -size 200 200 -output output2_10.png
```

```
example.exe -input scene3_01_cube_orthographic.txt -size 200 200 -output output3_01s.png
example.exe -input scene3_02_cube_perspective.txt -size 200 200 -output output3_02s.png
```

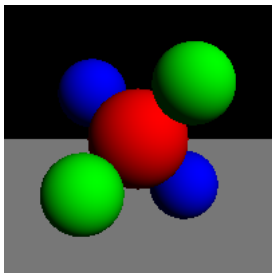


```
example.exe -input scene3_03_bunny_mesh_200.txt -size 200 200 -output output3_03s.png
```

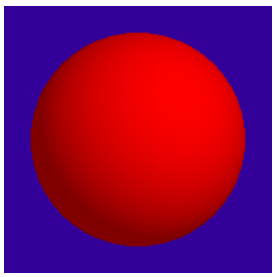
```
example.exe -input scene3_03_bunny_mesh_200.txt -size 200 200 -output output3_03super.png
-jittered_samples 16
```

```
example.exe -input scene3_04_bunny_mesh_1k.txt -size 200 200 -output output3_04s.png
```

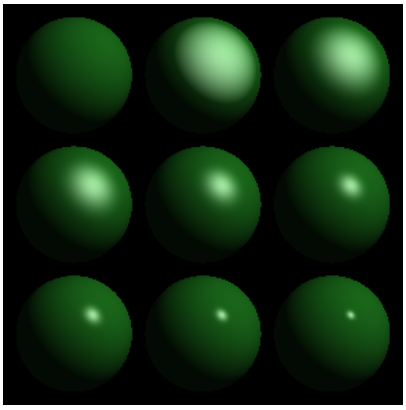
```
example.exe -input scene3_07_plane.txt -size 200 200 -output output3_07s.png
```



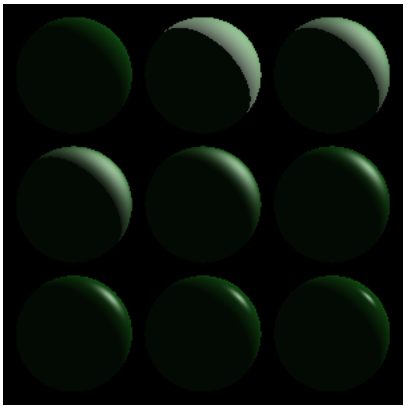
```
example.exe -input scene3_08_sphere.txt -size 200 200 -output output3_08s.png
```



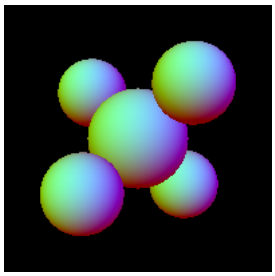
```
example.exe -input scene3_09_exponent_variations.txt -size 300 300 -output output3_09s.png
```



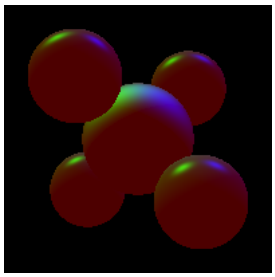
```
example.exe -input scene3_10_exponent_variations_back.txt -size 300 300 -output output3_10s.png
```



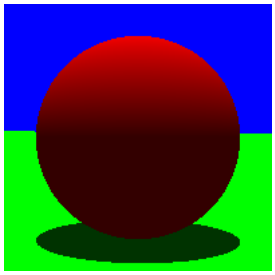
```
example.exe -input scene3_11_weird_lighting_diffuse.txt -size 200 200 -output output3_11s.png
```



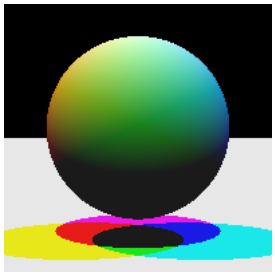
```
example.exe -input scene3_12_weird_lighting_specular.txt -size 200 200 -output output3_12s.png
```



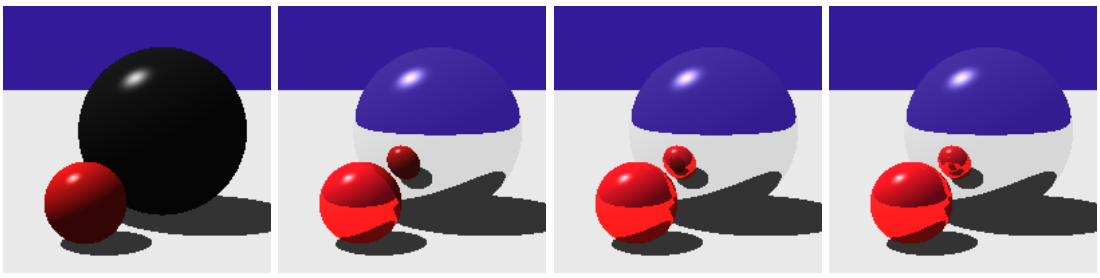
```
example.exe -input scene4_01_sphere_shadow.txt -size 200 200 -output output4_01s.png -shadows
```



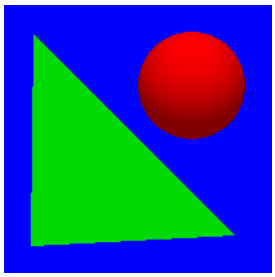
```
example.exe -input scene4_02_colored_shadows.txt -size 200 200 -output output4_02s.png -shadows
```



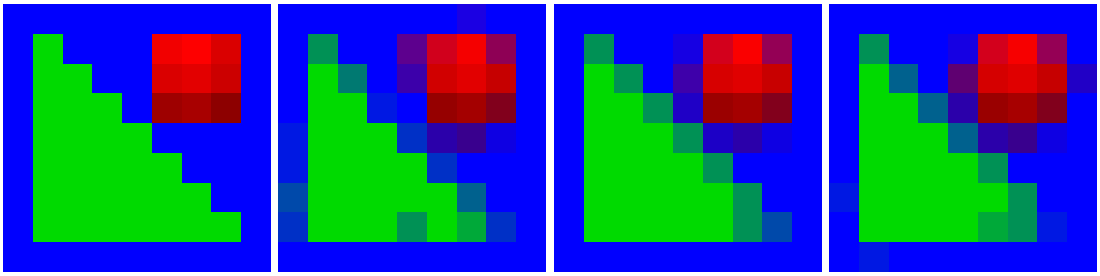
```
   example.exe -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04as.png
\
-shadows -bounces 0 -weight 0.01
   example.exe -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04bs.png
\
-shadows -bounces 1 -weight 0.01
   example.exe -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04cs.png
\
-shadows -bounces 2 -weight 0.01
   example.exe -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04ds.png
\
-shadows -bounces 3 -weight 0.01
```



```
   example.exe -input scene7_01_sphere_triangle.txt -size 200 200 -output output7_01.png
```



```
   example.exe -input scene7_01_sphere_triangle.txt -size 9 9 output7_01_nosampling.png
   example.exe -input scene7_01_sphere_triangle.txt -size 9 9 output7_01_uniform_9.png \
-uniform_samples 9
   example.exe -input scene7_01_sphere_triangle.txt -size 9 9 output7_01_regular_9.png \
-regular_samples 9
   example.exe -input scene7_01_sphere_triangle.txt -size 9 9 output7_01_jittered_9.png \
-jittered_samples 9
```



6 Hints

- Use a small image size for faster debugging. 64×64 pixels is usually enough to realize that something might be wrong.
- As usual, don't hesitate to print as much information as needed for debugging, such as the direction vector of the rays, the hit values, etc.
- Use `assert()` to check function preconditions, array indices, etc. See `cassert`.
- The "very large" negative and positive values for t used in the `Hit` class and the intersect routine can simply be initialized with large values relative to the camera position and scene dimensions. However, to be more correct, you can use the positive and negative values for infinity from the IEEE floating point standard.
- Use the various rendering modes (normal, diffuse, distance) to debug your code.

7 Extra Credit

Some of these extensions require that you modify the parser to take into account the extra specification required by your technique. Make sure that you create (and turn in) appropriate input scenes to show off your extension.

- Implement refraction. (Easy)
- Add simple fog to your ray tracer by attenuating rays according to their length. Allow the color of the fog to be specified by the user in the scene file. (Easy)
- Add other types of simple primitives to your ray tracer, and extend the file format and parser accordingly. For instance, you could provide implementations for `Transform` and `Box`, or how about a cylinder or cone? These can make your scenes much more interesting. (Easy)
- Add a new oblique camera type (or some other weird camera). In a standard camera, the projection window is centered on the z-axis of the camera. By sliding this projection window around, you can get some cool effects. (Easy)
- Implement a torus or higher order implicit surfaces by solving for t with a numerical root finder. (Medium)

7.1 Transparency

Given that you have a recursive ray tracer, it is now fairly easy to include transparent objects in your scene. The parser already handles material definitions that have an index of refraction and transparency color. Also, there are some scene files that have transparent objects that you can render with the sample solution to test against your implementation.

- Add transparency and refracted rays. (Easy)
- Dealing with transparent shadows by attenuating light according to the traversal length. We suggest using an exponential on that length. (Easy)
- Add the Fresnel term to reflection and refraction. (Very Easy).

7.2 Advanced Texturing

So far you've only played around with procedural texturing techniques but there are many more ways to incorporate textures into your scene. For example, you can use a texture map to define the normal for your surface or render an image on your surface.

- Image textures: render an image on a triangle mesh based on per-vertex texture coordinate and barycentric interpolation. You need to modify the parser to add textures and coordinates. Allow tiling (normal tiling and with mirroring) and bilinear interpolation of the texture. (Medium)
- Bump and Normal mapping: perturb (bump mapping) or look up (normal mapping) the normals for your surface in a texture map. This needs the above texture coordinate computation and derivation of a tangent frame, which is relatively easy. The hardest part is to come up with a good normal map image. To get credit, you must implement both bump mapping and normal mapping and produce maps for both techniques. (Medium)
- Isotropic texture filtering for anti-aliasing using summed-area tables or mip maps. Make sure you compute the appropriate footprint (kernel size). This isn't too hard, but of course, requires texture mapping). (Medium)
- Adding anisotropic texture filtering using EWA or FELINE on top of mip-mapping (a little tricky to understand, easy to program). (Easy)

7.3 Advanced Modeling

Your scenes have very simple geometric primitives so far. Add some new `Object3D` subclasses and the corresponding ray intersection code.

- Combine simple primitives into more interesting shapes using constructive solid geometry (CSG) with union and difference operators. Make sure to update the parser. Make sure you do the appropriate things for materials (this should enable different materials for the parts belonging to each primitive). (Easy)

- Raytrace implicit surfaces for blobby modeling. Implement Blinn's blobs and their intersection with rays. Use regula falsi solving (binary search), compute the appropriate normal and create an interesting blobby object (debugging can be tricky). Be careful for the beginning of the search, there can be multiple intersections. (Hard)

7.4 Advanced Shading

Phong shading is a little boring. I mean, come on, they can do it in hardware. Go above and beyond Phong. Check http://people.csail.mit.edu/addy/research/ngan05_brdf_supplemental_doc.pdf for cool parameters.

- Cook-Torrance or other BRDF (Easy).
- Bidirectional Texture Functions (BTFs): make your texture lookups depend on the viewing angle. There are datasets available for this http://btf.cs.uni-bonn.de/. (Medium)
- Write a wood shader that uses Perlin Noise. (Easy)
- Add more interesting lights to your scenes, e.g. a spotlight with angular falloff. (Easy)
- Replace RGB colors by spectral representations (just tabulate with something like one bin per 10nm). Find interesting light sources and material spectra and show how your spectral representation does better than RGB. (Medium)
- Simulate dispersion (and rainbows). The rainbow is difficult, as is the Newton prism demo. (Medium)

7.5 Global Illumination and Integration

Photons have a complicated life and travel a lot. Simulate interesting parts of their voyage.

- Add area light sources and Monte-Carlo integration of soft shadows. (Medium)
- Add motion blur. This requires a representation of motion. 10 points if only the camera moves (not too difficult), 10 more points if scene objects can have independent motion (more work, more code design). We advise that you add a time variable to the `Ray` class and update transformation nodes to include a description of linear motion. Then all you need is transform a ray according to its time value.
- Depth of field from a finite aperture. (Easy)
- Photon mapping (very difficult). (Hard).
- Distribution ray tracing of indirect lighting (very slow). Cast tons of random secondary rays to sample the hemisphere around the visible point. It is advised to stop after one bounce. Sample uniform or according to the cosine term (careful, it's not trivial to sample the hemisphere uniformly). (Easy)
- Irradiance caching (Hard).
- Path tracing with importance sampling, path termination with Russian Roulette, etc. (Hard)
- Metropolis Light Transport. Probably the toughest of all. Very difficult to debug, took a graduate student multiple months full time. (Very Hard)

- Raytracing through a volume. Given a regular grid encoding the density of a participating medium such as fog, step through the grid to simulate attenuation due to fog. Send rays towards the light source and take into account shadowing by other objects as well as attenuation due to the medium. This will give you nice shafts of light. (Hard)

7.6 Interactive Preview

A good tool to have when writing a ray tracer is some sort of interactive preview. Preview your scenes using OpenGL.

- Allow the user to manipulate the camera, preview all the primitives in the scene and render from that view. Takes some work, fair amount of code design and code writing. (Medium)
- Allow the user to interactively model the scene using direct manipulation. The basic tool you need is a picking procedure to find which object is under the mouse when you click. Some coding is required to get a decent UI. But once you have the mouse click, just trace a ray to find the object. Then, using this picker for translating objects (requires good coding), and for scaling and rotation. Allow the user to edit the radius and center of a sphere, and manipulate triangle vertices. All those are easy once you've figured out a good architecture but requires a significant amount of programming. (Medium)

7.7 Nonlinear Ray Tracing

We've had enough of linearity in T-111.4310! Let's get rid of the limitation of linear rays.

- Mirages and other non-linear ray propagation effects: Given a description of a spatially-varying index of refraction, simulate the non-linear propagation of rays. Trace the ray step by step, pretty much an Euler integration of the corresponding differential equation. Use an analytical or discretized representation of the index of refraction function. Add Perlin Noise to make the index of refraction more interesting. (Hard)
- Simulate the geometry of special relativity. You need to assign each object a velocity and to take into account the Lorentz metric. I suggest you recycle your transformation code and adapt it to create a Lorentz node that encodes velocity and applies the appropriate Lorentz transformation to the ray. Then intersection proceeds as usual. Surprisingly, this is not too difficult; that is, once you remember how special relativity works. In case you're wondering, there does exist a symplectic raytracer
  http://yokoya.naist.jp/paper/datas/267/skapps_0132.pdf
  that simulates light transport near the event horizon of a black hole. (Medium)

7.8 Multithreaded and Distributed Ray Tracing

Raytracing complicated scenes takes a long time. Fortunately, it is easy to parallelize since each camera ray is independent. Breaking your image up into subimages rendered by multiple threads will help render times even on one machine (especially if the machine has more than one processor). See `base/thread.hpp` for the threading facilities of the `framework` library.

- A multithreaded ray tracer with good load balancing. Requires serious programming skills and familiarity with multithreading. (Medium)
- Distribute the render job to multiple machines. Doing this in a brute force manner (split the image into one sub-region per machine) (Easy). With Load balancing (Medium). Good programming skills required.

7.9 Fancier Acceleration Techniques

Using spheres is ok but you can get much fancier when trying to accelerate your raytracer.

- Use kd-trees to accelerate your raytracer. (Hard).
- Uniform Grids: the sample solution uses grid acceleration (you can enable it with `-grid gx gy gz`). Essentially, you create a 3D grid and "rasterize" your object into it. Then, you march each ray through the grid stopping only when you hit an occupied voxel. Difficult to debug. (Medium).

7.10 More anti-aliasing

The assignment only asks for box filtering, but we all know it's limited and has bad Fourier characteristics..

- Make it possible to use arbitrary filters. We recommend you build on the `Film` data structure as an intermediate representation of the subsamples, which you will filter after they are all computed. Demonstrate your filtering approach with tent and Gaussian filters. (Easy)
- Add blue-noise or Poisson-disk distributed random sampling and discuss in your `README` the differences you observe from random sampling and jittered sampling. (Medium).

8 Submission Instructions

You are to write a `README.txt` (or optionally a PDF) that answers the following questions:

- Your name and student number at the beginning of the `README.txt`.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.

- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

Submit your assignment on Optima by **December 9th by 23:59**. Please submit a single archive (`.zip`) containing:

- Your Visual Studio project including the source code.
- Any additional files that are necessary.
- The `README.txt` file.