6.837: Computer Graphics Fall 2010

Programming Assignment 5: Ray Tracing

Due December 3rd at 8:00pm.

In this assignment, you will greatly improve the rendering capabilities of your ray caster by adding several new features. First, you will improve the shading model by adding Phong shading with point light sources. Second, you'll increase realism by recursively generating rays to create reflections and shadows. You'll add procedural solid texturing and implement supersampling to fix aliasing problems. Finally, you'll increase performance by adding a bounding sphere hierarchy to your scene.

The remainder of this document is organized as follows:

1 Getting Started

This assignment used to be several one week assignments. Please start as early as possible.

An updated parser, sample solution, and new scene files for this project are included in the assignment distribution. Run the sample solution `a5soln` as follows:

```
./a5soln -input scene7_03_marble_vase.txt -size 200 200 -output output7_03a.tga -shadows
```

This will generate an image named `output7_03a.tga`. You may view this file using `eog`. 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 well as the other test cases given below.

2 Summary of Requirements

As mentioned above you'll be adding several new features to your ray caster to improve its realism and performance.

New features

- Phong shading for specular objects
- Recursive ray tracing for shadowing and reflections
- Procedural solid textures to show off...
- Supersampling for antialiasing
- Bounding sphere hierarchy to speed up intersection queries

We go into more detail about these features below in the implementation notes.

Your code will be tested using a script on all the test cases below. Make sure that your program handles the exact same arguments as the examples below.

3 Starter Code

For this assignment, the starter code is your own. You'll start from the code you developed for the previous assignment. In addition, we'll provide you with a new sample solution and some updated files. There's a new scene parser to handle new material types, light specifications, and bounding spheres. If you had to modify `SceneParser.cpp` for assignment four, remember to make the same modifications to the new scene parser we provide for you. Also, there is updated light code that includes a new point light source. To help build procedural shaders, we provide you with some procedural noise routines. Finally, we give you some code to help with the antialiasing portion of the assignment (described in detail below).

4 Implementation Notes

4.1 Phong Shading

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

```
PhongMaterial::PhongMaterial( const Vector3f& diffuseColor,
   const Vector3f& specularColor, float exponent, const Vector3f& reflectiveColor,
   const Vector3f& transparentColor, float indexOfRefraction,
   const char* texture );
```

Note that the last three arguments are there to help you with some of the possible extra credit. For the base requirements, however, you can ignore them.

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 Vector3f Material::shade( const Ray& ray, const Hit& hit,
   const Vector3f& dirToLight, const Vector3f& lightColor ) const = 0;
```

It takes as input the viewing ray, the `Hit` data structure, and light information and returns the color for that pixel. In assignment four, the class `Material` represented a diffuse material. Now, we'll use it as a base class for the other types of materials you'll implement in this assignment.

4.2 Lighting for Phong Materials

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 has been provided for you, will return the scaled light color with this distance factored in.

In the last assignment the shading equation looked like:

$$I = A + \sum_i D_i$$

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, 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.

After you implement Phong shading, test your code against some of the simpler examples below.

4.3 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 will encapsulate the high-level computation in a `RayTracer` class that will be responsible for sending rays and recursively computing colors along them.

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).

Make sure your `Material` classes store the reflective multipliers, which are necessary for recursive ray tracing.

Create a new class `RayTracer` that computes the radiance (color) along a ray. Update your main function to use this class for the rays through each pixel. This class encapsulates the computation of radiance (color) along rays. It stores a pointer to an instance of `SceneParser` for access to the geometry and light sources. Your constructor should have these arguments (and maybe others, depending on how you handle command line arguments):

```
RayTracer( SceneParser* s, int max_bounces, float cutoff_weight, bool shadows, ...  );
```

The main method of this class is `traceRay` that, 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. `traceRay` takes as additional parameters the current number of bounces (recursion depth) and a ray weight that indicates the percent contribution of this ray to the final pixel color. The corresponding maximum recursion depth and the cutoff ray weight are fields of `RayTracer`, which are passed as command line arguments to the program. Note that weight is a scalar that corresponds to the magnitude of the color vector.

```
Vector3f traceRay( Ray& ray, float tmin, int bounces, float weight, Hit& hit ) const;
```

Now is a good time to make sure you did not break any prior functionality by testing against scenes from the last assignment.

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

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 for reflective materials (`getReflectiveColor().absSquared() > 0`) by sending a ray from the current intersection point in the mirror direction. For this, we suggest you write a function:

```
Vector3f mirrorDirection( const Vector3f& normal, const Vector3f& incoming );
```

Trace the secondary ray with a recursive call to `traceRay` using modified values for the recursion depth and ray weight. The ray weight is simply multiplied by the magnitude of the reflected color. Make sure that `traceRay` checks the appropriate stopping conditions. Add the color seen by the reflected ray times the reflection color to the color computed for the current ray.

4.4 Procedural Textures

4.4.1 Checkerboard

Next, you will add new material effects where the color of the material varies spatially using procedural solid texturing. This will allow you to render checkerboard planes and truly satisfy the historical rules of ray tracing. Solid textures are a simple way to obtain material variation over an object. Different material properties (color, specular coefficients, etc.) vary as a function of spatial location.

Derive a class `Checkerboard` from `Material`. `SceneParser` has been extended to parse the new material types. The `Checkerboard` class will store pointers to two `Materials`, and a pointer to a `Matrix4f` which describes how the world-space coordinates are mapped to the 3D solid texture space. Your solid texture implementation will describe a unit-length axis-aligned checkerboard between the two materials and the matrix will control the size and orientation of checkerboard cells. The prototype for the constructor is:

```
Checkerboard( const Matrix4f& matrix, Material* mat1, Material* mat2 );
```

Implement the `Checkerboard::shade` routine for ray tracing. You simply need to delegate the work to the shade function of the appropriate `Material` of the checkerboard. Using a float-to-integer conversion function `floor` and the function `odd`, devise a boolean function that corresponds to a 3D checkerboard. As a hint, start with the 1D case, just alternated segments of unit length, then generalize to 2 and 3 dimensions. Remember to multiply the intersection point by the matrix to properly transform the materials.

4.4.2 Perlin Noise

Next you'll build materials using Perlin Noise, which lets you to add controllable irregularities to your procedural textures. Here's what Ken Perlin says about his invention (`http://www.noisemachine.com/talk1/`):

Noise appears random, but isn't really. If it were really random, then you'd get a different result every time you call it. Instead, it's "pseudo-random"—it gives the appearance of randomness. Its appearance is similar to what you'd get if you took a big block of random values and blurred it (ie: convolved with a Gaussian kernel). Although that would be quite expensive to compute.

So we'll use his more efficient (and recently improved) implemenation of noise (`http://mrl.nyu.edu/~perlin/noise/`) translated to C++ in `PerlinNoise.{h,cpp}`.

First derive the class `Noise` from `Material` which computes the function:

$$N(x, y, z) = noise(x, y, z) + noise(2x, 2y, 2z)/2 + noise(4x, 4y, 4z)/4 + \ldots$$

for the specified number of octaves. Note that we provide this function for you in `PerlinNoise::octaveNoise`. With just the first octave, the `Noise` function is very smooth and blobby. Including additional octaves of higher frequencies adds finer details to the texture. The value of $N(x, y, z)$ will be a floating point number that you should clamp and use to interpolate between the two contained `Materials`. Here's the constructor for `Noise`:

```
Noise( const Matrix4f& m, Material* mat1, Material* mat2, int octaves );
```

You'll need to interpolate the values returned by your `Material::shade` computation. In addition, to correctly handle the ambient term and the reflective and transmissive components of materials, you'll need to modify the prototype of some `Material` accessor functions to also take the world-space coordinate as an argument.

Ken Perlin's original paper and his online notes have many cool examples of procedural textures that can be built from the noise function. You will implement a simple `Marble` material, also a subclass of `Material`. This shader uses the `sin` function to get bands of color that represent the veins of marble. These bands are perturbed by the noise function as follows:

$$M(x, y, z) = sin(\omega x + aN(x, y, z))$$

Try different parameter settings to understand the variety of appearances you can get. Remember that this is not a physical simulation of marble, but a procedural texture attempting to imitate our observations. The constructor for `Marble` is:

```
Marble( const Matrix4f& matrix, Material* mat1, Material* mat2,
   int octaves, float frequency, float amplitude );
```

where frequency is w and amplitude is a in the equation above.

Procedural materials can be used for fun recursive material definitions. You can have a checkerboard where cells have different reflection characteristics, or a nested checkerboard containing different materials. The notion of recursive shaders is central to production rendering.

4.5 Anti-aliasing

Next, you will add some 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. We also provide a `SampleDebugger` and `Sample` classes in `sample.h` and `SampleDebugger.{h,cpp}` to help you debug your code and implement some extra credit.

You are required to implement simple box filtering with uniform and jittered sampling. Modify your program to handle some new command line arguments (although some of them are only relevant to extra credit):

- `-uniform_samples <num_samples>`

- `-jittered_samples <num_samples>`
- `-box_filter <radius>`

In your rendering loop, cast multiple rays for each pixel as specifed by the `<num_samples>` argument. 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.

Note that for the base requirements the third parameter can be ignored since you are simply averaging the samples within the pixel being rendered. We make you handle the command line argument to be compatible with the sample solution which handles the more general case of filtering where samples can be averaged across pixels. This is a relatively easy extension described at the end of the assignment.

To visualize your sampling pattern add the command line option:

`-render_samples <filename> <zoom_factor>`.

If this option is specified, you should call the provided functions:

```
void SampleDebugger::setSample( int x, int y, int i,
   const Vector2f& offset, const Vector3f& color );
```

for each sample you generate in your rendering loop. Then, after all samples have been created call:

```
void SampleDebugger::renderSamples( char* samplesFile, int sampleZoom );
```

This function creates a zoomed-in version of the image plane with a pixel at each sample location. Verify that your jittering creates a nice distribution across the pixels. Do not try it with a large image resolution, as the zoom factor will result in a huge image. Now you can create interesting "pointillist" versions of your image. Verify that with a small zoom factor this visualization looks like the scene.

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

4.6 Acceleration

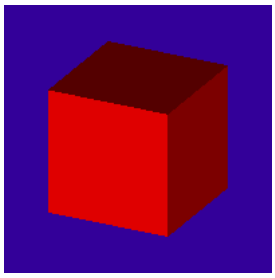
Now, you are going to make a simple change that will help improve perfomance quite a bit. You will have to handle a new `Object3D` class called `BoundingSphere`. The new `SceneParser` already handles parsing these objects. The constructor for `BoundingSphere` takes a `center` and `radius`, just like a `Sphere`, and also takes a child object like the `Transform` node. The `intersect` method should only call `intersect` on the `BoundingSphere`'s child if the given ray intersects the sphere. You already have sphere intersection code so you should refactor it in such a way that both `BoundingSphere` and `Sphere` can share the same code.

In the last test scene, the bunny mesh has been carved up into k bounding spheres using a special preprocessor (more details on that in the extra credit section). On the sample solution this bounded scene renders about seven times faster than the original bunny scene. Your code should see a similar improvement.

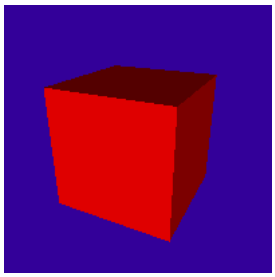
5 Test Cases

Your assignment will be graded by running a script that runs these new examples below (and any examples from the last assignment). Make sure your ray tracer produces the same output if you want to receive full credit.

```
./a5 -input scene3_01_cube_orthographic.txt -size 200 200 -output output3_01.tga
```



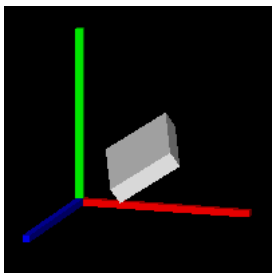
```
./a5 -input scene3_02_cube_perspective.txt -size 200 200 -output output3_02.tga
```



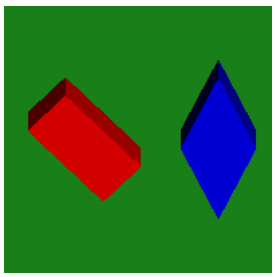
```
./a5 -input scene3_03_bunny_mesh_200.txt -size 200 200 -output output3_03.tga
```

```
./a5 -input scene3_04_bunny_mesh_1k.txt -size 200 200 -output output3_04.tga
```

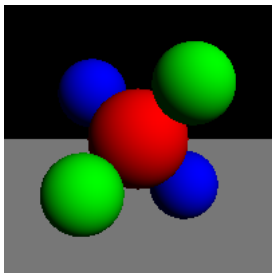
```
./a5 -input scene3_05_axes_cube.txt -size 200 200 -output output3_05.tga
```



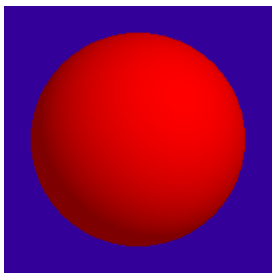
```
./a5 -input scene3_06_crazy_transforms.txt -size 200 200 -output output3_06.tga
```



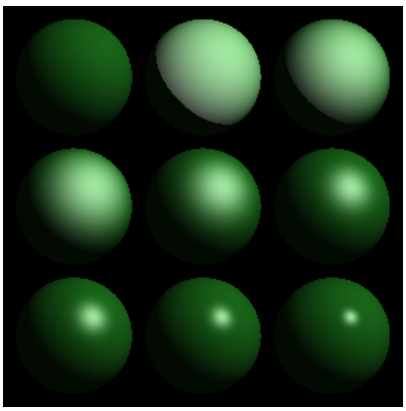
```
./a5 -input scene3_07_plane.txt -size 200 200 -output output3_07.tga
```



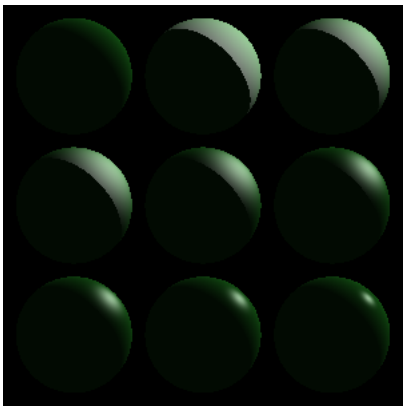
```
./a5 -input scene3_08_sphere.txt -size 200 200 -output output3_08.tga
```



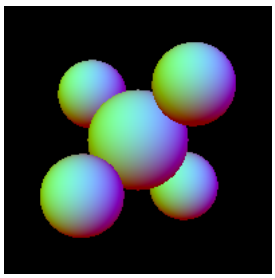
```
./a5 -input scene3_09_exponent_variations.txt -size 300 300 -output output3_09.tga
```



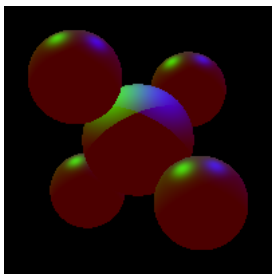
```
./a5 -input scene3_10_exponent_variations_back.txt -size 300 300 -output output3_10.tga
```



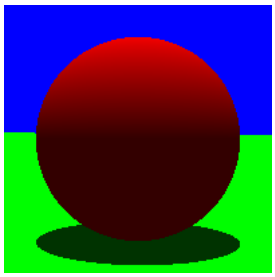
```
./a5 -input scene3_11_weird_lighting_diffuse.txt -size 200 200 -output output3_11.tga
```



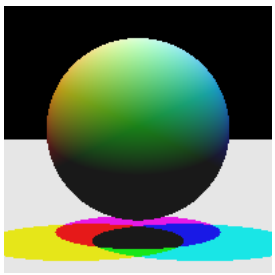
```
./a5 -input scene3_12_weird_lighting_specular.txt -size 200 200 -output output3_12.tga
```



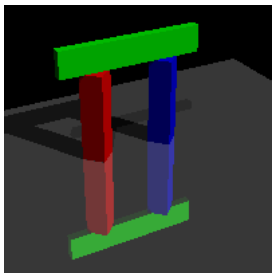
```
./a5 -input scene4_01_sphere_shadow.txt -size 200 200 -output output4_01.tga -shadows
```



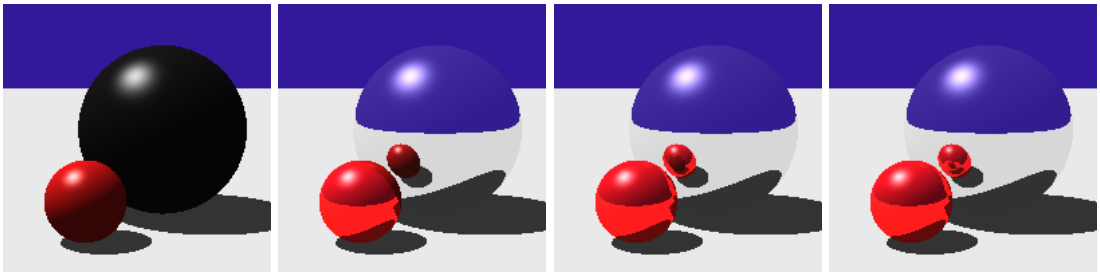
```
./a5 -input scene4_02_colored_shadows.txt -size 200 200 -output output4_02.tga -shadows
```



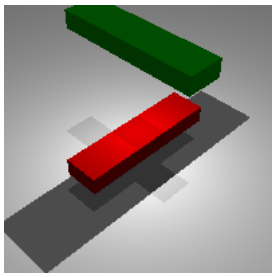
```
./a5 -input scene4_03_mirrored_floor.txt -size 200 200 -output output4_03.tga -shadows \
-bounces 1 -weight 0.01
```



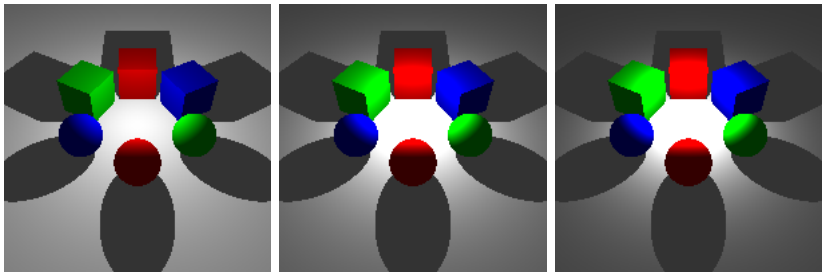
```
   ./a5 -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04a.tga \
-shadows -bounces 0 -weight 0.01
   ./a5 -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04b.tga \
-shadows -bounces 1 -weight 0.01
   ./a5 -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04c.tga \
-shadows -bounces 2 -weight 0.01
   ./a5 -input scene4_04_reflective_sphere.txt -size 200 200 -output output4_04d.tga \
-shadows -bounces 3 -weight 0.01
```



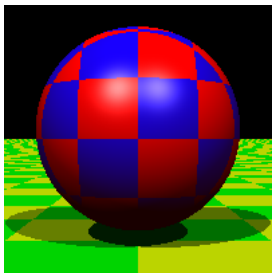
```
   ./a5 -input scene4_10_point_light_distance.txt -size 200 200 -output output4_10.tga \
-shadows
```



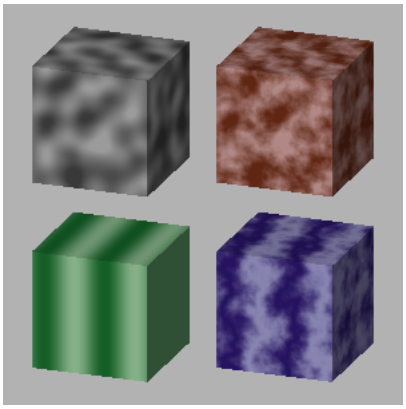
```
   ./a5 -input scene4_11_point_light_circle.txt -size 200 200 -output output4_11.tga \
-shadows
   ./a5 -input scene4_12_point_light_circle_d_attenuation.txt -size 200 200 -output \
output4_12.tga -shadows
   ./a5 -input scene4_13_point_light_circle_d2_attenuation.txt -size 200 200 -output \
output4_13.tga -shadows
```



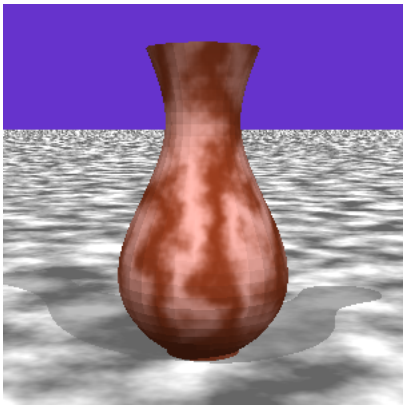
```
./a5 -input scene6_13_checkerboard.txt -size 200 200 -output output6_13.tga -shadows
```



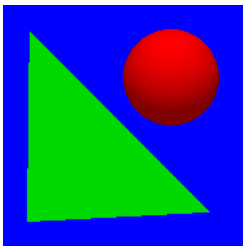
```
./a5 -input scene6_15_marble_cubes.txt -size 300 300 -output output6_15.tga
```



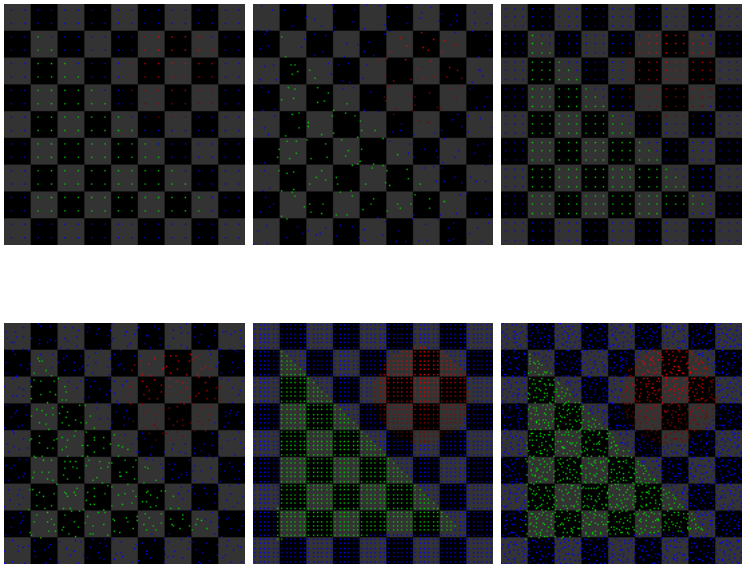
```
./a5 -input scene6_17_marble_vase.txt -size 300 300 -output output6_17a.tga \
-bounces 1 -shadows
```



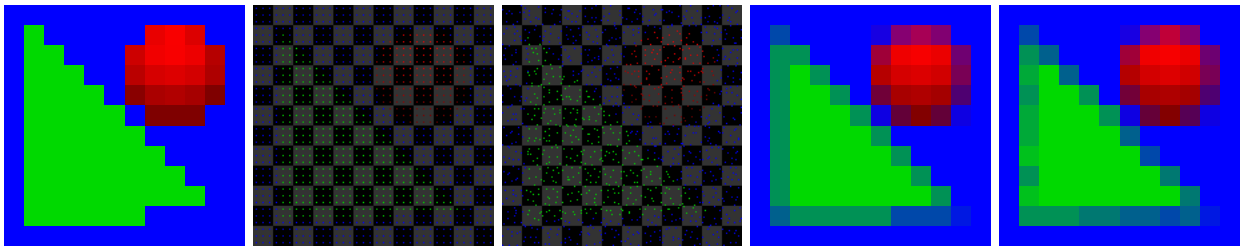
```
./a5 -input scene7_01_sphere_triangle.txt -size 180 180 -output output7_01.tga
```



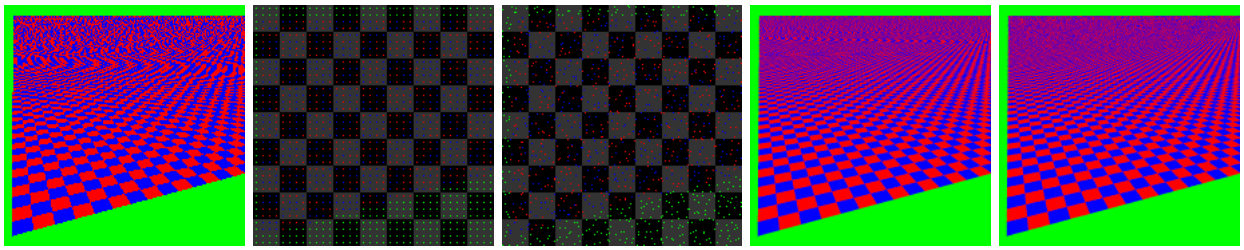
```
./a5 -input scene7_01_sphere_triangle.txt -size 9 9 -render_samples samples7_01b.tga 20 \
-uniform_samples 4
./a5 -input scene7_01_sphere_triangle.txt -size 9 9 -render_samples samples7_01c.tga 20 \
-jittered_samples 4
./a5 -input scene7_01_sphere_triangle.txt -size 9 9 -render_samples samples7_01e.tga 20 \
-uniform_samples 9
./a5 -input scene7_01_sphere_triangle.txt -size 9 9 -render_samples samples7_01f.tga 20 \
-jittered_samples 9
./a5 -input scene7_01_sphere_triangle.txt -size 9 9 -render_samples samples7_01h.tga 20 \
-uniform_samples 36
./a5 -input scene7_01_sphere_triangle.txt -size 9 9 -render_samples samples7_01i.tga 20 \
-jittered_samples 36
```



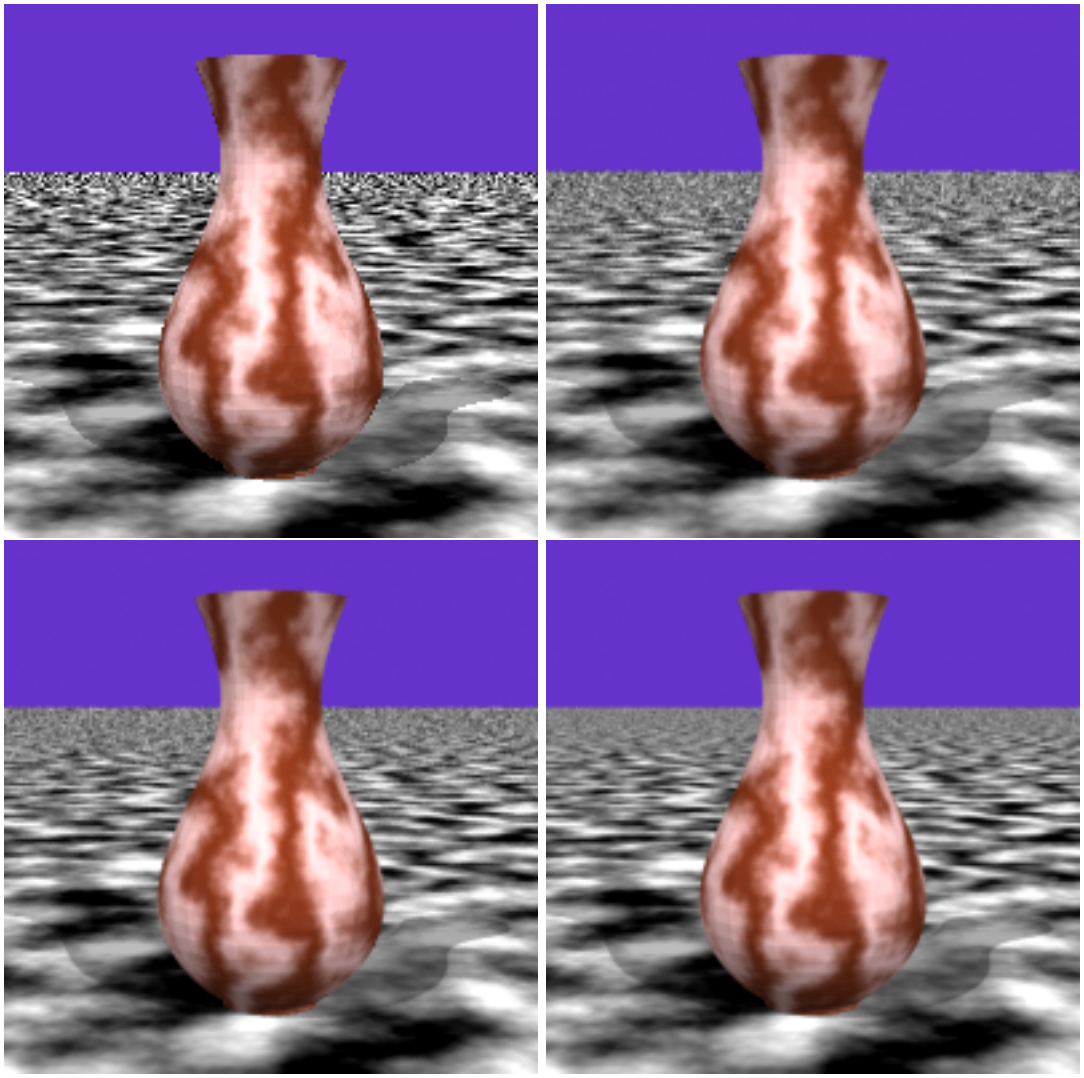
```
   ./a5 -input scene7_01_sphere_triangle.txt -size 12 12 -output output7_01_low_res.tga
   ./a5 -input scene7_01_sphere_triangle.txt -size 12 12 -render_samples \
samples7_01b_low_res.tga 15 -uniform_samples 9
   ./a5 -input scene7_01_sphere_triangle.txt -size 12 12 -render_samples \
samples7_01c_low_res.tga 15 -jittered_samples 9
   ./a5 -input scene7_01_sphere_triangle.txt -size 12 12 -output output7_01d_low_res.tga \
-uniform_samples 9 -box_filter 0.5
   ./a5 -input scene7_01_sphere_triangle.txt -size 12 12 -output output7_01g_low_res.tga \
-jittered_samples 9 -box_filter 0.5
```



```
   ./a5 -input scene7_02_checkerboard.txt -size 180 180 -output output7_02.tga
   ./a5 -input scene7_02_checkerboard.txt -size 9 9 -render_samples samples7_02b.tga 20 \
-uniform_samples 16
   ./a5 -input scene7_02_checkerboard.txt -size 9 9 -render_samples samples7_02c.tga 20 \
-jittered_samples 16
   ./a5 -input scene7_02_checkerboard.txt -size 180 180 -output output7_02d.tga \
-uniform_samples 16 -box_filter 0.5
   ./a5 -input scene7_02_checkerboard.txt -size 180 180 -output output7_02g.tga \
-jittered_samples 16 -box_filter 0.5
```



```
   ./a5 -input scene7_03_marble_vase.txt -size 200 200 -output output7_03a.tga -shadows
   ./a5 -input scene7_03_marble_vase.txt -size 200 200 -output output7_03b.tga -shadows \
-jittered_samples 4 -box_filter 0.5
   ./a5 -input scene7_03_marble_vase.txt -size 200 200 -output output7_03c.tga -shadows \
-jittered_samples 9 -box_filter 0.5
   ./a5 -input scene7_03_marble_vase.txt -size 200 200 -output output7_03d.tga -shadows \
-jittered_samples 36 -box_filter 0.5
```



```
./a5 -input boundBunny.txt -size 200 200 -output output3_04_BOUND.tga
```

6 Hints

- 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.
- You do not need to declare all methods in a class virtual, only the ones which subclasses will override.
- Print as much information as you need for debugging. When you get weird results, don't hesitate to use simple cases, and do the calculations manually to verify your results. Perhaps instead of casting all the rays needed to create an image, just cast a single ray (and its corresponding ray tree).
- Modify the test scenes to reduce complexity for debugging: remove objects, remove light sources, change the parameters of the materials so that you can view the contributions of the different components, etc.
- To avoid a segmentation fault, make sure you don't try to access samples in pixels beyond the image width and height. Pixels on the boundary will have a cropped support area.

7 Extra Credit

Most 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.

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.
- 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 here 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 online. (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 (we recommend using Qt, which is cross-platform and supported on Athena). 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 6.837! 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 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). On Linux you can use the `pthreads` library or Qt's threading classes.

- 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:

- How do you compile your code? Provide instructions for Athena Linux. You will not need to provide instructions on how to run your code, because it must run with the exact command line given earlier in this document.
- 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 Stellar by **December 3rd by 8:00pm**. Please submit a single archive (`.zip` or `.tar.gz`) containing:

- Your source code.
- A compiled executable named `a5`.
- Any additional files that are necessary.
- The `README` file.