This handout was compiled on 2024-11-18 20:37.

CS-C3100 Computer Graphics, Fall 2024

Lehtinen / Härkönen, Kemppinen, Kynkäänniemi, Kozlukov, Timonen

Assignment 5: Ray Tracing

Due December 8, 2024 at 23:59.

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

Requirements (maximum 15p) ***on top of which you can do extra credit***

0. **Displaying image coordinates (0.5p)**
1. **Generating rays; ambient lighting (1.5p)**
2. **Visualizing depth (1p)**
3. **Perspective camera (1.5p)**
4. **Phong shading; directional and point lights (3p)**
5. **Plane intersection (1p)**
6. **Triangle intersection (1.5p)**
7. **Shadows (1.5p)**
8. **Mirror reflection (1.5p)**
9. **Antialiasing (2p)**

1 Getting Started

Again, the build environment is unchanged; start by pulling and merging the changes from our upstream repository at Aalto GitLab.

This time in addition to being a graphical OpenGL application, the assignment can be run as a console application that takes arguments on the command line, reads a scene file and outputs a PNG image. There's an interactive preview mode that lets you fly around the scene and ray trace an image from at any point, but for any test you want to repeat, it's more convenient to use a test script. We supply such scripts in the assignment main folder that you can copy and modify as you need. More on them below.

As usual, you also get an `reference.exe` binary whose results you can (and should!) compare to yours. The scene files are plain text, so you can easily open and read them to know exactly what a particular scene is supposed to be and what features it uses. You won't have to write any loading code. The interactive preview lets you move around using the `WASD` keys and look around with the mouse by dragging with the right mouse button down. The mouse wheel controls the speed of camera movement.

There are buttons and sliders for most of the important ray tracer and camera settings. Pressing Enter renders an image using the ray tracer from the current viewpoint and using all of the current settings. Pressing Space toggles between the ray traced image and interactive preview. If you press E, a debug visualisation ray is traced and its path (and intersection normals) appears as line segments in the preview – this is especially useful when debugging geometry intersections, reflections and refractions.

To use the scripts, start the familiar x64 command prompt and navigate to the root directory of your Assignment 5 code. The scripts there work as follows:

- `render_all.bat` renders all requirements' scene files with manually tweaked settings for some scenes and stores the results in the `out` subdirectory. You can add or change some of the calls in it to render a specific scene in higher resolution, or add a test scene of your own, for example.
- `render_options.bat` renders a single scene and takes a set of arguments so you can change the rendering settings. The -depth argument must be the last one given. See the contents of `render_all.bat` for example usage. There are lots more arguments that you can set; consult the `Args` class for more info.
- By default, the render scripts use the executable from your `build\Debug` folder. To render the scenes using the reference renderer instead, run `set_renderer_reference.bat` prior to running any rendering script. To use your own release build, run `set_renderer_release.bat`. Note that the scripts do not check whether the executables exist or if they've been built with the latest changes to your sources. Be sure to build the executables and check from the script's output that the correct one is being called.

We apologize for the lack of scripts for platforms other than Windows; we will likely move to Python-based scripting in the future to avoid the complexities of the various types of shells and command prompts out there. However, it should not be difficult to port the batch files to Unix-like environments, including MacOS.

If you plan on doing extra credit, it's a very good idea to create a test scene where your new feature is clearly visible (and maybe have two render calls to it; one with and one without the feature). These already exist for some of the recommended extras. Also add the scenes to your `render_all.bat`.

The batch files assume that you have an exe file compiled in x64 release mode. Switch that on in visual studio, or edit the exe path accordingly (.bats are just text files).

Note that this assignment is more work than the previous ones. Please start as early as possible.

2 Application structure

The `render` function in `main.cpp` is the main driver routine for your application.

The code is designed to render individual scanlines in parallel on multiple processor cores by using OpenMP (link). It will speed up the rendering by a lot and we recommend you use it. *However, it is disabled by default in the starter code's GUI to prevent any surprises to and to make debugging easier.* There is a checkbox in the GUI for enabling parallelization[1], and the batch files and the reference executable have it enabled by default, too. If you choose to enable multithreading, you have to keep your own code thread-safe! All the requirement code should be thread-safe if you write it in the most natural way, but with some extras like Film/Filter, special care will be needed. (There are pointers in the comments.) If you get strange bugs, first try disabling multithreading to rule it out as a cause.

The different shape primitives are designed in an object hierarchy. A generic `ObjectBase` 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 `ObjectBase`.

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

[1] feel free to change the default behavior by editing the initializer for the `parallelize_` member of `class App` in `app.h`

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 (such as `FLT_MAX`). It is modified by the intersection computation to store the new closest t and the `Material` of intersected object.

3 Detailed instructions

R0 Displaying image coordinates (0.5p)

As an intro, we'll render a simple combination of color gradients to introduce the structure of the renderer in this assignment. Instead of preparing a list of attributes to be sent to the GPU for realtime rendering, we'll be forming images pixel by pixel on the CPU.

The image we'll be generating is this:

The color value of each pixel in the image is a linear function of its coordinates in the image. If the coordinate system of the image is such that the top left corner is at $(0, 0)$ and bottom right corner is at $(1, 1)$, we map the x coordinate to the red channel, the y coordinate to the green channel and keep the blue channel at 1. You should implement this mapping in the function `render` in `main.cpp`. Familiarize yourself with the loop structure in `render` and set the `sample_color` to match the definition of this image if `args.display_uv` is true.

To test your solution, you should open the scene file `r0_empty.txt` and press Enter or the Raytrace button, or run the command line version with the flag `uv`. You can also enable the UV rendering via a button in the GUI.

Note that your solution might appear tiled; this is due to the default Downscale Factor of 8. We expect each pixel of our images to take some time to compute so we render a lower resolution image to get a quicker preview. You can drag the Downscale Factor slider on the right down to 1 to see a smooth end result, but even this simple image will take a bit to render.

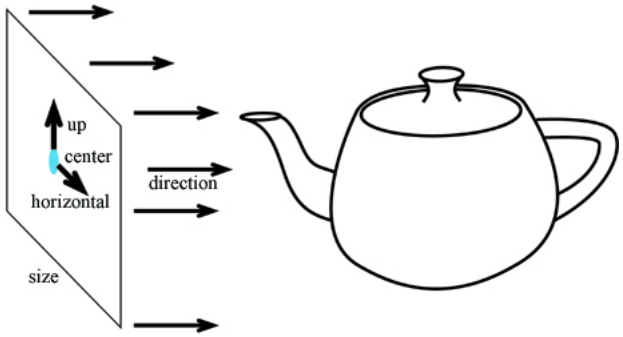
R1 Generating rays; ambient lighting (1.5p)

Our first task related to ray tracing is to cast some rays into a simple scene with a red sphere and an orthographic camera (`r1+r2_01_single_sphere`).

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

- `virtual Ray generateRay(const Vector2f& point, float fAspect) = 0;`
- `virtual float getTMin() const = 0;`

The first is used to generate rays for each screen-space coordinate, described as a `Vector2f` and the given image aspect ratio. 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.



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 has a range of the whole image plane.

The screen coordinates of the plane vary from $(-1,-1) \rightarrow (1,1)$. The corresponding world coordinates (where the origin lives) vary from $\mathbf{center} - (size * \mathbf{up})/2 - (size * fAspect * \mathbf{horizontal})/2$ to $\mathbf{center} + (size * \mathbf{up})/2 + (size * fAspect * \mathbf{horizontal})/2$.

The camera does not know about image resolution; image resolution is handled in your main loop.

Implement `normalizedImageCoordinateFromPixelCoordinate` method in `Camera` as well as `OrthographicCamera::generateRay`. Because the base code already lets you intersect rays with `Group` and `Sphere` objects, you should now see the sphere as a flat white circle. To complete the requirement, head to `RayTracer::traceRay` and add one line of code that creates ambient lighting for the object using the ambient light of the scene and the diffuse color of the object. After this is done, you should see the sphere in its actual color:

There is also another test scene with five spheres that end up overlapping each others in the picture frame:

R2 Visualizing depth (1p)

In the `render` function, implement a second rendering style to visualize the depth t of objects in the scene. Depth arguments to the application are given as `-depth 9 10 depth_file.png`, where the two numbers 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) and the filename specifies the output file. The depth rendering can be performed simultaneously with normal output image rendering.

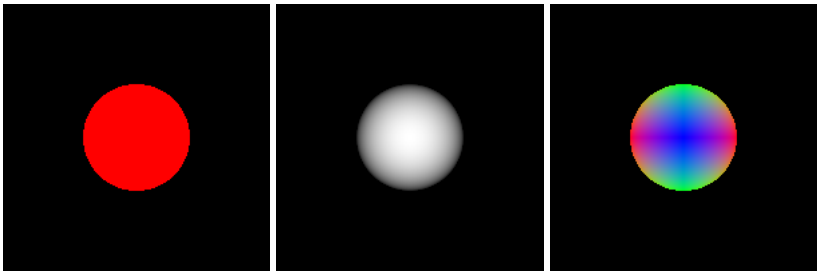


Figure 1: `r1+r2_01_single_sphere`: colors, depth, and normals

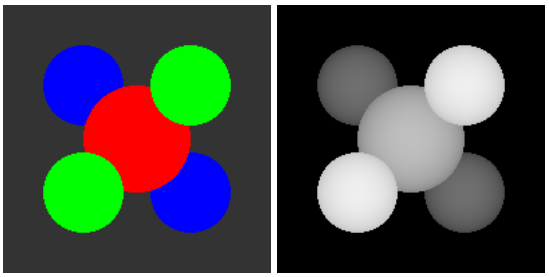
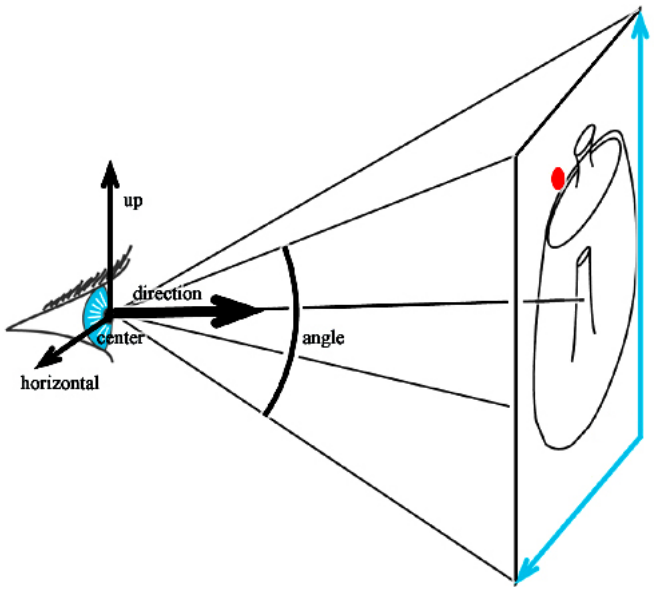


Figure 2: `r1+r2_02_five_spheres`: colors, depth

See the ready-made batch files for details and good depth values for a few scenes. Feel free to fill your own.

Note that the base code already supports normal vector 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. This may prove useful when debugging shading and intersection code.

R3 Perspective camera (1.5p)



To complete this requirement, implement the `generateRay` method for `PerspectiveCamera`. This happens

pretty much exactly as detailed on the lecture slides, starting with the computation of a convenient distance for the image plane based on the field-of-view.

Note that in a perspective camera, the value of `tmin` has to be zero to correctly clip objects behind the viewpoint.

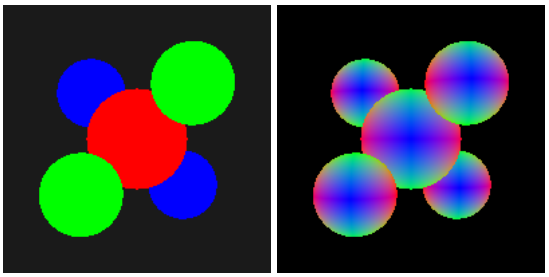


Figure 3: `r3_spheres_perspective`: a familiar scene again with a perspective camera (colors and normals)

R4 Phong shading; directional and point lights (3p)

- Provide an implementation for `DirectionalLight::getIncidentIllumination` to get directional lights.
- Implement diffuse shading in `PhongMaterial::shade`.
- Extend `RayTracer::traceRay` to take the new implementations into use. The class variable `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::getIncident-Illumination`.

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. In Eigen, this is implemented through `Vector3f::cwiseProduct`. Note that you have to explicitly cast the result back to a `Vector3f` because the type returned is an expression object instead of a vector: `Vector3f b, c; Vector3f a = Vector3f(b.cwiseProduct(c))`. Note, also, that if the ambient light color is $(1, 1, 1)$ and the light source color is $(0, 0, 0)$, then you have constant shading.

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

Directional lights 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. With point light sources, 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

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

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

$$S_i = c_{light_i} * k_s * (clamp(\mathbf{v} \cdot \mathbf{r}_i))^q$$

Here, k_s is the specular coefficient, $\mathbf{r}_i$ is the ideal reflection vector of light i, $\mathbf{v}$ is the viewer direction (direction to camera), and q is the specular reflection exponent. k_s is the `specularColor` parameter in the `PhongMaterial` constructor, and q is the `exponent` parameter. Refer to the lecture notes for obtaining the ideal reflection vector.

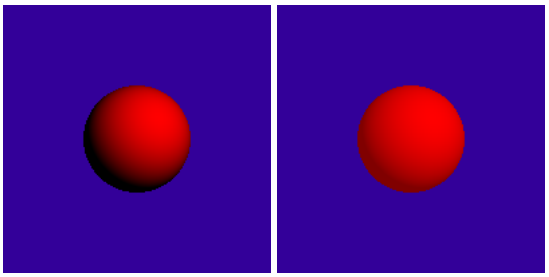


Figure 4: `r4_diffuse_ball, r4_diffuse+ambient_ball`: only diffuse, and both diffuse and ambient shading, respectively

Figure 5: `r4_colored_lights`: three different directional lights shading a white sphere

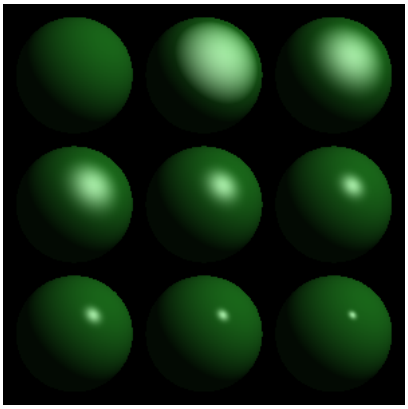


Figure 6: `r4_exponent_variations`: spheres with different specular exponents. The light is coming from somewhere in the top right.

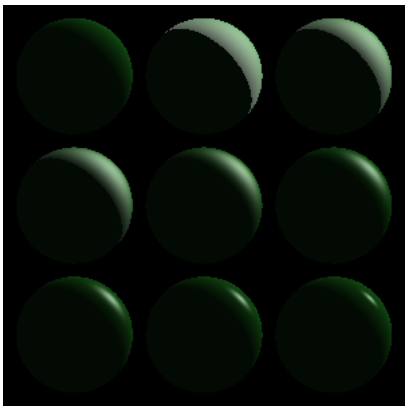


Figure 7: `r4_exponent_variations_back`: spheres with different specular exponents, back side. Note that you'll have to disable shadows if rendering from GL window to get this look.

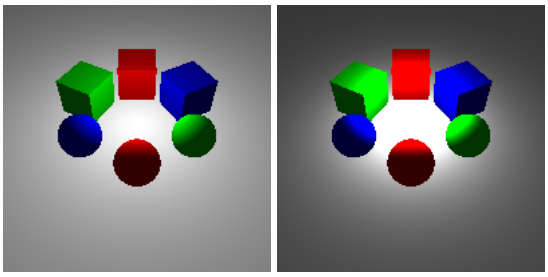


Figure 8: `r4_point_light_circle, r4_point_light_circle_d2`: constant and quadratic attenuation, respectively. Note that this scene requires transforms as well as sphere, plane and triangle intersections. The light on the plane will look the same without transforms as well, so pay attention to that.

R5 Plane intersection (1p)

Implement the `intersect` method for `Plane`.

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.

You can look at `Group` and `Sphere` intersection code to see how those are implemented.

Then implement the `intersect` method for `Plane`. Test with `r5_spheres_plane`. The R4 point light circle scene includes also a plane. Note that the preview approximates the plane with a finite square; it will look different than the ray traced plane.

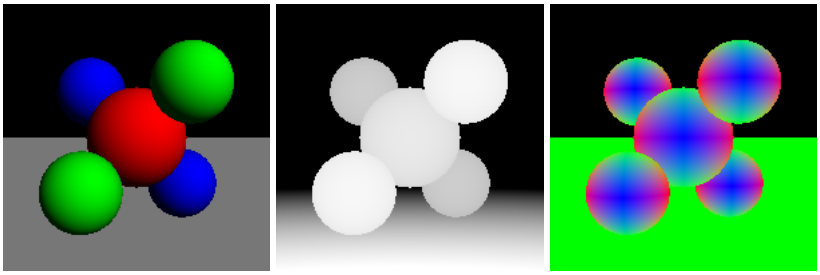


Figure 9: `r5_spheres_plane`: a familiar scene again with a plane as a floor. Colors, depth, and normals.

R6 Triangle intersection (1.5p)

Implement the `intersect` method for `Triangle`. Simple test scenes are e.g. the plain cube scenes, and more complicated ones (many many triangles) are the bunnies. The R4 point light circle includes also boxes that are composed by triangles, so you can try it too; it verifies also the shading result.

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.

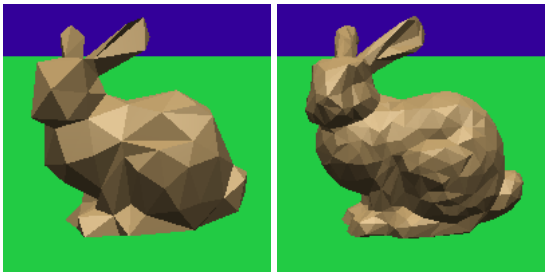


Figure 10: `r6_bunny_mesh_200 and ..._1000`: Bunnies consisting of 200 and 1000 triangles, respectively.

R7 Shadows (1.5p)

Next, you will add some global illumination effects to your ray caster. Once you cast secondary rays to account for shadows and reflection (plus refraction for extra credit), you can call your application a ray tracer. For this requirement, extend the implementation of `RayTracer::traceRay` to account for shadows. A new command line argument `-shadows` will indicate that shadow rays are to be cast.

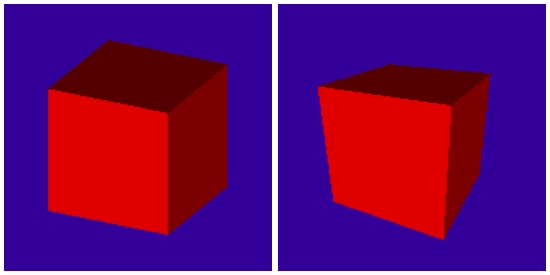


Figure 11: `r6_cube_orthographic, r6_cube_perspective`: a cube (consisting of just triangles) with the two different cameras

To implement cast shadows, send rays from the hit point toward each light source and test whether the line segment joining the intersection point and the light source intersects an object. If so, the hit point is in shadow and you should 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. You might also want to add the shadow ray to the debug visualisation vector to make it visible among the other rays.

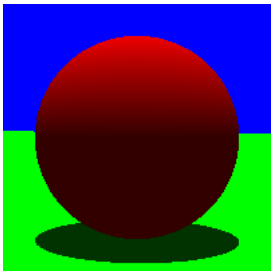


Figure 12: `r7_simple_shadow`: light coming from the above

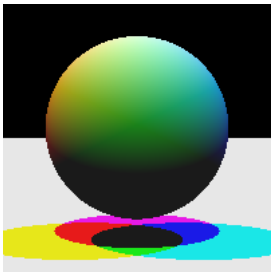


Figure 13: `r7_colored_shadows`: three different lights casting somewhat intersecting shadows

R8 Mirror reflection (1.5p)

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.

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

```
Vector3f mirrorDirection(const Vector3f& normal, const Vector3f& 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.

We need a stopping criterion to prevent infinite recursion — the maximum number of bounces the ray will make. This argument is set like so: `-bounces 5`. When you make a recursive `traceRay` call, you need to remember to decrement the bounce value.

The ray visualisation might be very useful for debugging reflections. In the preview application, click on the rendered result and fly around the scene to see how the rays have bounced.

The parameter `refr_index` in `traceRay` is the index of refraction for a material, needed for extra credit.

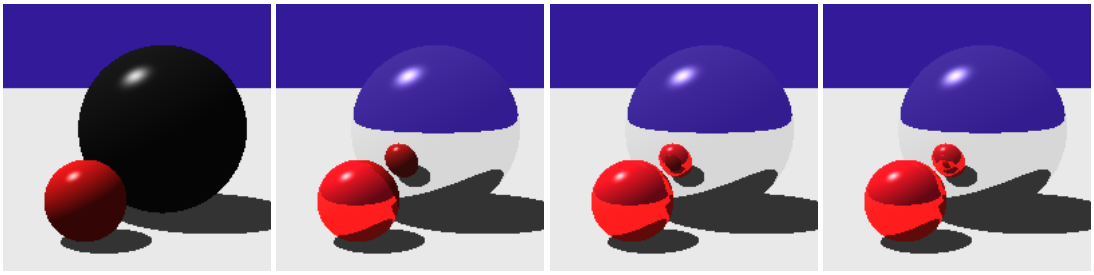


Figure 14: `r8_reflective_sphere`: shown here four different levels of reflections (0, 1, 2, and 3 bounces) with weight 0.01

R9 Antialiasing (2p)

Finally, you will add simple supersample anti-aliasing to your ray tracer to alleviate jagged edges and Moiré patterns.

For each pixel, instead of directly storing the colors computed with `RayTracer::traceRay` into the `Image` class, you'll compute many color samples using `RayTracer::traceRay` at different positions in each pixel — not just the center! — and average them. You'll also implement two methods for generating the sample positions inside the pixels.

In signal processing terms, the simple method of aggregating the information from the samples by plain average is called "box filtering". It is also the only filter you are required to implement. Implementing better averages ("prefilters") is the topic of easy extra credit. The reference executable implements two additional filters, the tent and Gaussian.

To generate samples within the pixels, we use three different techniques:

- (Already implemented in starter code) `UniformSampler::getSamplePosition`: returns a uniformly distributed random 2D point within the unit square $[0,1] \times [0,1]$ using `Sampler::random_Vector2f`.
- `RegularSampler::getSamplePosition`: no randomization; the sampler should conceptually divide the unit square $[0,1] \times [0,1]$ into $\sqrt{N} * \sqrt{N}$ sub-pixels (with `num_samples`=N) and return the center point of the nth subpixel when requested. So, when $N = 1$, it should always return $(0.5, 0.5)$; when $N = 2$, it should return $(0.25, 0.25)$, $(0.75, 0.25)$, $(0.25, 0.75)$, and $(0.75, 0.75)$ for $n = 0 \ldots 3$, and so on.
- `JitteredSampler::getSamplePosition`: the same as `RegularSampler::getSamplePosition`, except the samples at the sub-pixels should be uniformly distributed. For example, when $N = 2$, the call `JitteredSampler::getSamplePosition(3)` should result in a random `Vector2f` with coordinates uniformly distributed in $[0.5, 1] \times [0.5, 1]$, etc. (It makes sense to draw a picture of this with pen and grid paper first.) When $N = 1$, the points it returns are distributed as with the uniform random sampler.

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>`

In your rendering loop (`render` in `main.cpp`), the innermost loop already generates multiple rays for each pixel as specifed by `args.samples_per_pixel`. To fulfill the box filtering requirement, you simply need to average the colors of all the samples taken from each pixel before writing it to the image.

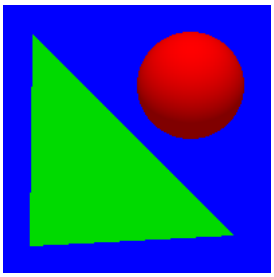


Figure 15: `r9_sphere_triangle` 200x200

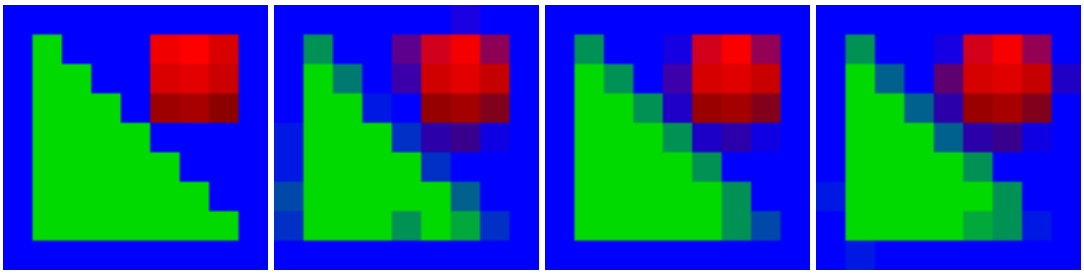


Figure 16: `r9_sphere_triangle`, just 9x9 resolution: none, uniform, regular, and jittered sampling (sample count 9)

4 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.

4.1 Recommended

- Implement refraction through transparent objects. See handout and code comments around where reflection is implemented. For full points, make sure you handle total internal reflection situations. (1-2p)
- 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. (1p)
- Add other types of simple primitives to your ray tracer, and extend the file format and parser if necessary. At least provide implementations for `Transform` and `Box`; there are skeletons for them in the base code. (3p)
- Make it possible to use arbitrary filters by filling in the `addSample` function of the class `Film`. Its function is to center a filter function at each incoming sample, see which pixel centers lie within the filter's support, and add the color, modulated by the filter weight, into all those pixels. Further, accumulate the filter weight in the 4th color channel of the image to make it easy to divide the sum of weights out once the image is done. Demonstrate your filtering approach with tent and Gaussian filters. Be aware that the trivial Filter/Film implementation is not thread-safe, as many threads may want to update the same pixel simultaneously. See the comments in `main.cpp` for hints on how to use `std::mutex` for serializing updates to the image to guarantee correctness. (1-3p)
- Implement stereo cubemap rendering for your ray tracer. Stereo cubemaps can be used to display a 3d view you can freely look around in using a VR headset. Your task is to write code that generates a cubemap texture file that can be rendered using a third-party application, such as vizor.io, or you can create your own stereo renderer for further extra credit. Note that you won't necessarily need a VR headset, as you can view the cubemaps with a non-stereo display as well.

  Your stereo cubemap should consist of two cubemaps rendered from slightly offset viewpoints matching the offset between human eyes, to enable the viewer to experience depth perception in the scene. The produced cubemap should be a single png file with 12 square tiles, each describing a single face of the cube map. The axis layout of a single cubemap can be seen in the image below. For stereo cubemaps your image should repeat the axes twice, first for the left eye, then the right. There are other layout standards as well, but at least vizor.io uses the one described here. For more information on cubemaps, you can read the Wikipedia article, and you can find example stereo cubemaps from https://render.otoy.com/vr_gallery.php/.

  You can use this scene, created with one of the example cubemaps, as your starting point for viewing your cubemaps. You can just upload your own cubemap into the stereo cubemap object, which you can find in the scene tree under the Program tab in the editing window. You should include a link to your own scene in your readme file, or include the produced cubemap texture in your submission folder. (4+p)

4.2 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.

- Enable transparent shadows by attenuating light according to the traversal length through transparent objects. We suggest using an exponential on that length. (1.5p)
- Add the Fresnel term to reflection and refraction. (1p)

4.3 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. Some features you might want to support are tiling (normal tiling and with mirroring) and bilinear interpolation of the texture. (2-4p)
- Bump and Normal mapping: perturb (bump map) or look up (normal map) 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. Produce a scene demonstrating your work. (2-3p)
- 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)

4.4 Advanced Modeling

Your scenes have very simple geometric primitives so far. Add some new `ObjectBase` 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). (4-5p)
- Implement a torus or higher order implicit surfaces by solving for t with a numerical root finder. (2-3p)
- 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. (4-6p)

4.5 Advanced Shading

Phong shading is a little boring. I mean, come on, they can do it in hardware. Go above and beyond Phong. Check this for cool parameters.

- Cook-Torrance or other BRDF (2p).
- Bidirectional Texture Functions (BTFs): make your texture lookups depend on the viewing angle. There are datasets available for this, for example here. (3p)

- Write a wood shader that uses Perlin Noise. (2p)
- Add more interesting lights to your scenes, e.g. a spotlight with angular falloff. (1p)
- 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. (3-4p)
- Simulate dispersion (and rainbows). The rainbow is difficult, as is the Newton prism demo. (3-4p)

4.6 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. (4-5p)
- Add motion blur. This requires a representation of motion. 3 points if only the camera moves (not too difficult), 3 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. (2-3p)
- Photon mapping. (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). (3-5p)
- 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)

4.7 Interactive Editing

- 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, 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. (up to 7p)

4.8 Nonlinear Ray Tracing

We've had enough of linearity already! 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. (Hard)

4.9 Multithreaded and Distributed Ray Tracing

Raytracing complicated scenes takes a long time. Fortunately, it is easy to parallelize since each camera ray is independent. We already provide an easy OpenMP implementation for distributing the load on local processor cores, but you can take it further.

- Create a raytracer running on the GPU. Since replicating all requirement features would be a huge amount of work, you can make your GPU raytracer separate and give it only a smaller amount of functionality. You can use your choice of API - CUDA, OpenCL, GLSL shaders. We make an exception here and allow you to use technology that is not supported on the classroom computers; if necessary, we'll call you in to demonstrate the code you submitted. (Medium/Hard)
- Distribute the render job to multiple computers in a brute force manner. Split the image into one sub-region per machine, send them off to individual machines for rendering and collect the results when done. (Hard)

4.10 Acceleration Techniques

- Use a Bounding Volume Hierarchy to accelerate your raytracer. (Hard)

4.11 More anti-aliasing

- Add blue-noise or Poisson-disk distributed random sampling and discuss in your `README` the differences you observe from random sampling and jittered sampling. (2p)

5 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Windows with Visual Studio 2022, preferably in the VDI VMs. You may develop on your own platform, but the TAs will grade the solutions on Windows.
- Comment out any functionality that is so buggy it would prevent us seeing the good parts. Use the `reference.exe` to your advantage: does your solution look the same? Remember that crashing code in Debug mode is a sign of problems.

- Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code. Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, project and CMake+vcpkg configuration files, required to build your submission, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. *Do not* include your `build` subdirectory or the reference executable. Beware of large hidden `.git`-folders if you use version control - those should not be included.
- Sanity check: look inside your ZIP archive. Are the files there? Test and see that it actually works: unpack the archive into another folder, run the CMake configuration, and see that you can still compile and run the code.
- If you experience a system failure that leaves you unable to submit your work to MyCourses on time, prepare the submission package as usual, then compute its MD5 hash, and email the hash to us at `cs-c3100@aalto.fi` before the deadline. Then make the package that matches the MD5 hash available to us e.g. using `filesender.funet.fi`. *This is only to be done when truly necessary; it's extra work for the TAs.*

Submit your archive in MyCourses folder "Assignment 5: Ray tracing".