CS 488/688 (.): Introduction to Computer Graphics

Fall 2023 Assignment 4: Ray Tracer

Due: Friday, November 10th, 2023 at 5:30 PM

Summary

The goal of this assignment is to write a simple, mostly non-recursive ray tracer that can do (at least) the following:

1. Cast primary rays from the eye position through every pixel in an image plane;
2. Intersect primary rays with scene geometry;
3. Cast shadow rays from intersection points to each light source; and
4. Use the gathered information to perform illumination calculations and choose a colour for the pixels associated with the primary rays.

For geometric primitives you must support spheres, cubes and triangle meshes. Those primitives can be assembled into a modelling hierarchy, meaning that your ray-object intersection code must be able to traverse that hierarchy recursively. You must support a standard Phong illumination model (remember that this is different from Phong shading of mesh surfaces), lit by point light sources and a global ambient light. You must support ray tracing acceleration via hierarchical bounding volumes (which can be spheres, boxes, or some other, fancier data structure), and be capable of demonstrating the resulting improvement in performance. Finally, you must implement one additional feature of your own choosing, and create a novel scene and rendered image that demonstrate that feature.

A few additional notes on these features:

- The ray tracer does not need to have a graphical user interface. It can read the input script, produce an output image, and stop.
- You should use face normals to shade cubes and triangle meshes. In particular, you do not need to support vertex normals, and you do not need to implement Phong shading (i.e., interpolation of normals across faces). By default, your meshes will look faceted. You can implement Phong shading as your extra feature, as long as you maintain backwards compatibility with the provided test scripts. (In that case, you should consider implementing a second mesh type, with a new Lua command, that can store vertex normals and use them during shading.)
- You are required to implement only primary and shadow rays. You are *not* required to implement recursive reflection or refraction rays, though of course these can form part of your additional feature. (Reminder: attenuation applies to shadow rays only, not primary rays.)
- When a primary ray doesn't hit any scene object, you should assign it a background colour. You must implement an interesting background that does not interfere with the objects in the scene. The background should not be too bright, or the patterns too busy (a sunset might be a reasonable background, for example...). A constant-coloured background is not acceptable. You can parameterize the background by either pixel position or ray direction. Ray direction could be used to create a sky background with dark blue at the zenith, white at the horizon, and brown below the horizon, or a starfield for an outer space scene. Use your imagination. All your raytraced images should have a background.
- Remember that you're not implementing the forward graphics pipeline. You can write an entire ray tracer without ever constructing our familiar V and P matrices. Instead, construct primary rays directly in world coordinates. Then, in a recursive tree traversal, you'll end up re-expressing primary rays in local modelling coordinate systems in order to perform object intersection. When you find an intersection,

you'll have to transform the information about (location, normal vector) back into world coordinates on the way back to the root of the tree. To some extent you could flatten the tree, baking the hierarchy of transformations down into the primitives or a single level of transformation above them; note that this can't count as an extra objective.

- You should place your test scene in a file called `sample.lua` in the `A4` directory. The test scene should contain at least one of each required primitive type, at least one point light source, at least one shiny surface (with obvious specular reflection), a non-trivial background (see below) and at least one demonstration of your additional feature. The script should produce a rendered image in `sample.png` and exit.

Modelling

As with Assignment 3, input scenes are described using a simple modelling language built on top of a Lua interpreter. The following function behaves similarly to Assignment 3, but isn't quite identical:

- `gr.mesh( name, objfilepath )` : Load a mesh from an external OBJ file. Returns a reference to a newly created `GeometryNode` that points to the mesh. Internally, meshes are cached so that multiple invocations of `gr.mesh()` with the same file name will share the same underlying `Mesh` instance.

Assignment 4 also includes the following new functions:

- `gr.nh_box( name, {x,y,z}, r )` : Create a non-hierarchical box of the given name. The box should be aligned with the axes of its Model coordinates, with one corner at (x, y, z) and the diagonally opposite corner at ($x + r$, $y + r$, $z + r$).
- `gr.nh_sphere( name, {x,y,z}, r )` : Create a non-hierarchical sphere of the given name, with centre (x, y, z) and radius r.
- `gr.cube( name )` : Create an axis-aligned cube with one corner at (0,0,0) and the other at (1,1,1). (It's expected that this cube would be placed under transformations in a tree.)
- `gr.sphere( name )` : Create a sphere with centre (0,0,0) and radius 1.
- `gr.light( {x,y,z}, {r,g,b}, {c0,c1,c2} )` : Create a point light source located at (x, y, z) with intensity (r, g, b) and quadratic attenuation parameters c_0, c_1 and c_2).
- `gr.render( node, filename, w, h, eye, view, up, fov, ambient, lights )` : Raytrace an image of size $w \times h$ and store the image in a PNG file with the given filename. The parameters *eye*, *view*, *up* and *fov* control the camera, with *fov* being the field of view in the y-direction (up direction). The last two parameters are the global ambient lighting and a list of point light sources in the scene.

The functions `gr.nh_box` and `gr.nh_sphere` allow you to implement a non-hierarchical version of the ray tracer, since you can place spheres and boxes in arbitrary locations in world coordinates. Thus you will be able to test aspects of your code such as shading and shadows without necessarily having hierarchical transformations working. We provide a few non-hierarchical test scenes to facilitate this process. Later you may find it easier to build scenes using `gr.sphere` , `gr.cube` , and hierarchical transformations.

Tips and cautions

You are free to develop your code as you like. However, you will probably find it easiest if you first write a non-hierarchical ray tracer (Objectives 1–6). Once you have that part of the code debugged, add the hierarchical part (Objectives 8 and 9, affine transformations, a general hierarchy, and bounding volumes for meshes). Depending on what you implement for your extra feature, you may or may not want to complete Objective 10 before starting on hierarchical transformations. Finally, you need to create a unique scene (Objective 7). While you can write the unique scene earlier in the development cycle, you may want to wait until you know if you will finish hierarchical models, since the power of hierarchical modelling will let you build a more interesting scene.

For this assignment, you are allowed to produce some console output. For instance, you may want to output your render parameters and have some indication of how much progress your ray tracer is making (i.e., 10%, 20% done etc). Printing out your entire hierarchy tree is probably too much output, though.

For debugging purposes, it can be useful to have a way to trace a single primary ray. That way, if you know that a particular part of your scene is producing incorrect output you can follow your program's execution at a single pixel and examine its behaviour in detail. Another approach is to render a scene in false colours that provide some insight into the behaviour of the program at every pixel. You might even consider embedding the ray tracer in an interactive UI like that of A3, where you can compare an OpenGL rendering of your scene with the corresponding raytraced image (but this could require a lot of extra effort).

Depending upon which extra feature you are implementing, you may find that rendering times start to increase. To improve performance, you can add the `-O2` flag to the buildOptions variable in the `premake4.lua` file. Then simply run `premake4 gmake && make` again. To use the gdb debugger, you can replace the `-O2` with `-g`.

Be aware of numerical issues that arise in any ray tracing algorithm. In particular, consider the following tips from past students:

- Try to minimise the number of times that you normalize vectors and normals. Each time you normalize, you introduce a small amount of error that can cause major problems.
- The intersection of a ray and an object may end up slightly inside the object due to limited numerical precision. In such a case, any secondary rays cast from the intersection location will hit the same object before anything out in the rest of the world. To avoid this problem, discard all intersections that occur too close to the ray origin.
- Use "epsilon" checks in your intersection routines, particularly the ray-intersect-triangle routine.

In hierarchical ray tracing, on "the way down" you should transform the point and vector forming the ray with a node's *inverse* transform. On "the way back up" you should transform the intersection point by the original transformation, and (assuming you represent the normal as a column vector) you should transform the normal with the transpose of the upper-3×3 part of the inverse transform.

Extensions

You are required to implement an additional non-trivial feature. There are many possibilities, such as an efficiency improvement or an addition of functionality. Several ideas are given below. They are purely a list of suggestions for your consideration, and not intended to be exhaustive; a list of papers on ray tracing is given in the course bibliography (https://www.student.cs.uwaterloo.ca/~cs488/r.pdf). These papers contain a lot more possibilities (some of them might even include code.) A good overall reference for possible extensions is the book *Physically Based Rendering* by Pharr and Humphreys.

New features

- **Mirror reflections**: This involves (recursively) issuing secondary reflection rays from the point of intersection. This would apply to objects that have been assigned a material with a non-zero "reflectivity" (which is best kept distinct from a material's specular colour).

  Since purely reflective objects are rare, blend the colour accumulated along the reflected ray with the colour produced by local illumination, using the reflectivity as a coefficient. Note that you will need to specify a means of halting recursion.

- **Refraction**: This involves (recursively) issuing secondary refracted rays for objects that have a transparency coefficient and an index of refraction. Implementation is very similar to reflection rays. Use Snell's law to compute the direction of the refracted ray. [Whitted]

  Aside: if you want to get fancy, the ratio between reflection and refraction is given by a function that produces more reflection at glancing angles. This is called the *Fresnel effect* and is a consequence of Maxwell's Laws; see the text or the references. For the purposes of this assignment, you can use

constant coefficients for the amount of reflection and refraction. However, watch out for total internal reflection.

- **Supersampling**: This involves generating multiple rays for each pixel and using some averaging function to combine the colours returned (e.g., sample on a 3×3 grid over the area of the pixel and average the nine colour values that are returned). This helps to reduce the "jaggies" and is the most basic form of **antialiasing**.

- **Other antialiasing methods**: Generate more rays only where the scene changes (adaptive sampling), use random (stochastic) sampling techniques (jitter the sample positions in the subpixel grid), etc. There are many, many antialiasing techniques. [Cook : Stochastic] [Dippe : Stochastic] [Lee Uselton] [Mitchell] [Painter : Adaptive-Progressive Refinement]

  Note: If you choose to implement a supersampling objective, make sure you include at least two comparison images of the same scene, one with supersampling turned on and the other without.

- **Fisheye/Omnimax Projection**: Ray trace a 180 degree view, using a hemispherical "screen".

- **Spherical Lens Systems**: Simulate a real lens system; use stochastic sampling across the aperture to simulate depth of field, and refractive intersection with sphere surfaces to create a lens system. (Even one lens is interesting; more would make a good project).

- **Additional primitives**: Extend the modelling language and add primitives for (truncated) cylinders and cones. Note that these primitives are basically particular quadrics intersected with a pair of halfspaces, and (truncated) paraboloids and hyperbolids are in the same category. Quadric-based primitives are very easy to implement, especially since you have already been provided with a stable quadratic solver.

  There are other possible primitive objects, such as superquadrics or tori, although these are harder than quadric-based solvers. For instance, a torus is generated by a quartic equation, which can be solved analytically, but it's hard to write a numerically stable quartic solver. Some kind of numerical root-solver is often required; regula falsi is recommended, but isolate the roots first using a geometric approach (for polynmials, binary search using Sturm sequences might be a better choice). You might also try recent 3D fractal primitives like the Mandelbulb (https://en.wikipedia.org/wiki/Mandelbulb), using some kind of ray marching approach to converge on the fractal surface.

- **Texture mapping**: Determine the diffuse reflectance of objects based on stochastic or deterministic functions. This requires a function that computes, for each intersection point on the surface of a primitive, a set of texture coordinates. For instance, for a sphere, you can use the elevation and azimuth of the intersection point to index the texture image. Alternatively, you can use a projective transformation of the modelling coordinates of the intersection point. Computing the diffuse reflectance procedurally as a function of the spatial position (x, y, z) in modelling coordinates can be used to generate solid textures like wood grain or marble [Perlin,Hart].

  See textures.guinet.com (http://textures.guinet.com/) for some sample textures.

- **Bump mapping**: Similar to texture mapping, but modulate an object's surface normal instead of its diffuse colour [Blinn].

- **Lighting Models**: Implement a decent "physical" lighting model, or some other alternative lighting model (like the Blinn-Phong model). Implement Phong shading (interpolation of vertex normals). Note that catN.obj is the same model as cat.obj but with normals added. However, to load this model, you will need to extend the mesh reader to read (and store) normals.

- **Constructive Solid Geometry (CSG)**: Extend the ray tracer to provide CSG operations at model nodes (intersect, union, diff, etc.) and extend the intersection routines to return 1D intersection intervals along the ray rather than a single intersection distance. Then perform suitable merges of lists of intersection

intervals at internal CSG nodes. These computations can be performed using a simple count-up-at-entry and count-down-at-exit algorithm. Once the final set of intervals has been computed, take the first point of the first interval as the intersection point.

Improved efficiency

The key to improving efficiency is the avoidance of unnecessary work, or rather, only applying work were it will make a difference. Some ideas for improving efficiency are:

- **Intensity thresholds**: Check if the accumulated product of mirror reflectances is less than some epsilon, then return black without checking for further ray hits (obviously goes along with generating secondary rays for mirror reflection).
- **Spatial partitioning**: Modify your intersection routine to use a space partitioning scheme; e.g., BSP trees, uniform spatial subdivision, or octrees. Uniform spatial subdivision is particularly easy to implement and performs well in practice; a hierarchical version can be used as an extension of this in a ray-tracing project.

  Note: simply flattening the scene graph and caching transformation matrices, as suggested earlier, is not enough to get you credit for this objective.

If you choose to implement some kind of optimization for your extra feature, you must offer an effective demonstration that it actually works. If the improvement can not be demonstrated visually, you might note differences in timing in your `README`, for example.

There's no harm in implementing multiple extensions in the scope of this assignment. We'll give you the credit for whichever one you want to associate "officially" with this assignment. If you plan to extend your ray tracer for the project, you'll still be able to count all those additional extensions as objectives later.

Provided files

As usual, you should be able to use `premake4 gmake; make` in the `A4/` subdirectory to build the skeleton code. When run, the skeleton program ignores the scene graph read in from the Lua script and produces a fixed background as a demonstration of writing colours to pixels.

A number of sample scripts are provided in the `Assets` directory. The `.lua` files are the scene files themselves, which you pass as command-line arguments to the `A4` executable; the `.obj` files are meshes that are loaded by the scene files. Some of these meshes have non-triangular faces, and at least one mesh has normals included. You are not required to handle non-triangular faces or normals, but may choose to do so if you want, although at a minimum, you will have to extend the mesh reading code to handle these extensions.

Most of these scripts will work only when invoked from within the `Assets` directory, since they load OBJ files in the same directory. For some of the test scenes, we provide sample renderings in the course gallery (https://www.student.cs.uwaterloo.ca/~cs488/testscenes.html).

Note the files `polyroots.hpp` and `polyroots.cpp`, which provide stable and robust solvers for quadratic, cubic and quartic polynomials (*caution*: there is some concern that polyroots doesn't always find the roots of quartic polynomials). We recommend you use these solvers to handle intersections with most algebraic surfaces.

Deliverables

Prepare and upload a ZIP file of the `A4/` directory, omitting unnecessary files (executables, build files, etc.). Make sure to include at least the following files:

- **A `README` file**, as usual. Your `README` needs to contain a description of your extra feature and of your unique scene(s). If you implement an acceleration feature, provide a switch to turn it on and off (this can

be a compile-time switch) and provide comparative timings. If you use external models, please credit where you got them from.

- **A screenshot**, in the file `screenshot.png`. For this assignment, the screenshot should not be of the running program, but your best rendered image. This is the image considered for bonus marks. **Your assignment will not qualify for bonus marks if you do not submit a file named screenshot.png**
- **A signed copy of a4declare.pdf.** If you fail to submit this file, then you will receive a 0 on this assignment.
- **A sample script**. In the `A4/` directory, include a test script called `sample.lua` for your unique scene, and the rendered image `sample.png` that it produces when run. Ideally, this script should demonstrate your extra feature. This image should have a resolution of at least 500×500. You may submit additional images if you wish; mention them in your README.
- **Additional sample images**. In the `Assets/` directory, include renderings of *at least* the files `nonhier.lua`, `macho-cows.lua` and `simple-cows.lua`. The images should be in PNG format and stored in `nonhier.png`, `macho-cows.png` and `simple-cows.png`. The scripts `macho-cows.lua` and `simple-cows.lua` will fully test all the features of your raytracer except your additional feature. If you do not complete the entire assignment, of course, you will not be able to render all three scenes.

  In addition, to demonstrate that you've implemented bounding volumes, you should make a *special renderering* of either `nonhier.lua` or `macho-cows.lua` in which you draw the bounding volumes instead of the meshes. This image should be stored in `nonhier-bb.png` or `macho-cows-bb.png`.
- **Bounding Volume Code**. Be sure to submit the code that renders the bounding volumes. You can `#ifdef` it out using the pre-processor for normal rendering, but it should be submitted. We have provided a `#define` called `RENDER_BOUNDING_VOLUMES` in `Mesh.hpp` for this purpose.

There will be a bit of subjective grading of your raytraced images. If the image is extremely good, we may award extra credit. If the image is extremely simple, but tests all features, the TA may subtract up to one half a mark. If the image does not test all features, more marks may be deducted, since the TAs will not be able to verify all the features.

If you can not get hierarchical transformations working, submit an image made without them, and mention that you are missing hierarchical transformations in your `README`. You will be severely penalized if we discover that you misrepresented yourself, of course.

Objective list

Every assignment includes a list of objectives. Your mark in the assignment will be based primarily on the achievement of these objectives, with possible deductions outside the list if necessary.

Assignment 4 objectives
▢ 1. Objects are visible in rendered images. This implies that you can generate primary rays, intersect them with spheres, and generate PNG output.
▢ 2. Cubes and triangle meshes are properly rendered.
▢ 3. Hidden surfaces are not visible: objects are correctly ordered from back to front.
▢ 4. There is a function that generates a non-trivial background for the scene without obscuring the view of any objects in the scene. The background appears in all generated images.
▢ 5. A Phong illumination model is implemented, including diffuse, specular, and ambient illumination.

▢ 6. Objects are able to cast shadows on other objects.
▢ 7. A script has been supplied that defines and renders a novel scene.
▢ 8. Hierarchical transformations perform correctly. Spheres and cubes can be transformed via arbitrary affine transformations.
▢ 9. Bounding volumes (spheres or boxes) have been implemented for mesh objects, as demonstrated by a special rendering as described in the assignment text.
▢ 10. The assignment includes an extra feature or improvement beyond the core capabilities defined in the assignment text.