Chapter 1

CS488/688 W12

A4: Introduction

> *"If you want to do a ray tracer for your project, you* **really** *need to complete Assignment 4 first."*

This assignment is due **Friday, March 9th [Week 9]**.

If you still need the provided code for this assignment run `/u/gr/cs488/bin/setup A4` from your CSCF account.

1.1 Topics

- Ray tracing.
- Inverse transformations and ray transformation.
- Ray intersection with spheres, cubes, and meshes of convex planar polygons.
- The Phong lighting model.

1.2 Statement

A ray tracer comprises a few major pieces of code to

1. Cast rays from the eye point through each pixel,
2. Intersect the rays with the objects in the scene,
3. Cast shadow rays from the point of intersection to each light source, and
4. Shade the rays, summing the contribution from each visible light source, and assign a colour to the pixel.

You are to write such a ray tracer. This ray tracer needs to support spheres, cubes, and meshes of convex, planar polygons as its basic geometric primitives. The ray tracer modeling language will be implemented as a set of Lua callbacks, so scenes—including procedurally generated scenes—are specified as Lua scripts.

If you are planning to render a large scene (as opposed to a small test scene, small being less than 32x32 rays) please use the machines in MC6055: raytracing takes a lot of sustained CPU time, which CFCF frowns upon for timeshared machines. On shared CFCF machines, if you render small test images, please "nice" your rendering jobs. Note also that many shells put a limit on how much CPU time a single program can use. To avoid running into this limit, run `unlimit cputime` before starting your ray tracer.

You are required to implement the following ray tracing functionality:

- Hierarchical transformations and models.
- The sphere, cube, and (convex, planar) polygonal mesh geometry types.
- Bounding volumes (sphere or boxes) for polygonal mesh types.
- Point light sources, with a Phong lighting model.
- Output of the resulting image to a PNG file.
- One additional feature of your own choosing, and a scene file (and resulting image) that demonstrates this feature.

Note: you may assume that all faces of a polygonal meshes are convex and planar. It is *suggested* that an intersection-of-halfspaces approach be used in 2D to test if an intersection point with the plane of a polygon lies within that (convex) polygon's area. Project the 3D plane/ray intersection point into 2D by dropping the coordinate corresponding to the largest value in the polygon's plane normal.

Use face normals for the cube and polygonal mesh model types to support shading. You do *not* have to support vertex normals. Note that Phong shading, i.e. normal interpolation, is *not* required—you don't have vertex normals anyway on the polygonal mesh type provided. However, you can implement Phong shading as your extra feature, as long as you can still read the existing test scripts. It is suggested that if you do this you add a new polygonal mesh primitive with a new Python command, rather than modifying the interface to the one provided here. In general, implement the interface to your new features in a backward-compatible way.

You are also required to generate a test script named `sample.lua` that demonstrates the following: all of the primitive types required, the point light sources, at least one "shiny" surface (using the specular Phong reflectance model), and your additional features. This script should just write the resulting image to a `sample.png` file, and exit. You can look at the sample files in `/u/gr/cs488/data/A4` to get some ideas. You can also obtain some images corresponding to the test scripts from the course web page. These can be found in the A4 test scene link, under the gallery/exhibition section. When you submit the images rendered from these given scripts, please use the ORIGINAL scripts without any modifications. If you want to use these scripts to demonstrate your extra objective(s), such as transparency, reflectivity, etc., be sure to include the images generated from the unmodified scripts as part of you sample images.

You are only required to implement primary and shadow rays. Recursive secondary rays, needed to support reflections and transparency, are *not* required. Technical note: attenuation applies only to shadow rays, not primary rays!

You are also required to provide an interesting background that does not interfere with the objects in the scene. This means that colours should not be too bright, or the patterns too varied.

Sunset effects could be tried, just as a suggestion. A constant background is not acceptable. You can parameterize the background either by pixel position or ray direction. The latter parameterization can be used, for instance, to make a sky background with dark blue at the zenith, white at the horizon, and brown below the horizon, or to implement a starfield for an outer space scene. Use your imagination. Your background should be in all your raytraced pictures.

The easiest way to implement a hierarchical raytracer is to transform each *ray* from the Viewing Coordinate System (VCS) to the Modeling Coordinate System (MCS) of each model. The transformed ray is then intersected with models in a canonical position, so you can take advantage of special forms of equations. To search the entire scene, for each ray you can do a recursive tree traversal, generating transformation matrices as you go. Alternatively, you can precalcuate all VCS-to-MCS transformations, flattening the tree; this will be somewhat faster (flattening cannot count as an extra objective, though). Keep in mind that in either case light sources should be specified in WCS, so you will have to transform the intersection from MCS back to WCS to do shading.

To compute the matrix used to transform the rays, you should extend the model callbacks to maintain an *inverse* transformation in each model node. Then compose these in reverse order as you descend the tree to find the necessary VCS to MCS transformation for each primitive. Object intersections then can be made more efficient (and simpler) by using a canonical form of an object in MCS.

1.3 Suggested Development

You are free to develop your code as you like. However, you will probably find it easiest if you first write a non-hierarchical raytracer (Objectives 1-6). Once you have that part of the code debugged, add the hierarchical part (Objectives 8,9; affine transformations, a general hierarchy, and bounding volumes for polyhedral objects). Depending on what you implement for your extra improvement, you may or may not want to complete Objective 10 before starting on hierarchical transformations.

Finally, you need to make a unique scene (Objective 7). While you can write a 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 have some console output. For instance, you may want to output your render parameters and have some indication of how much progress your raytracer is making (i.e., 10%, 20% done etc). Printing out your entire hierarchy tree is probably too much output, though.

1.4 Cautions

This assignment is a lot of work. Although things look easy in the course notes, the details are a bit tricky to work out. What follows are a few hints from students who have worked in this assignment in the past.

- Numerical problems abound. In particular, watch for the following:
  - 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 the ray and an object may actually be slightly inside the object. When casting secondary rays (such as shadow rays), the first object the secondary rays will hit will be the same object. 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-polygon routine.

- In hierarchical ray tracing, on "the way down" you should transform the point and vector forming the ray with the *inverse transform.* On "the way back up" you should transform the intersection point by the transformation, and (assuming you represent the normal as a column vector) you should transform the normal with the *transpose of the inverse transform.* Potentially, this transformation of the normal will result in a non-zero 4th coordinate, in which case you should set the 4th coordinate to 0 (or, write a special matrix-vector product that ignores the fourth coordinate, and uses the transpose).

  To speed things up you way want to precompute and cache all the VCS-MCS transformations. Expand the DAG into a strict hierarchy first if you use caching.

1.5 Possible Raytracer 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; a list of papers on ray tracing is given in the course bibliography. These papers contain a lot more possibilities (some of them might even include code.)

1.5.1 Extra Functionality

Mirror reflections: This involves (recursively) issuing secondary reflection rays from the point of intersection. This would apply to objects that have been assigned a surface property with a non-zero "mirror coefficient".

Since purely reflective objects are rare, blend the colour found recursively with the colour used for shading a semi-reflective object using shadow rays, using the mirror coefficient. Note that you will have to specify a maximum recursion depth. If you use mirror coefficients $r < 1$, then the magnitude of r^n for the maximum r in the scene will give you an idea of how much error will be made when truncating the recursion in scenes with multiple reflective objects. Even a recursion depth of 1 can generate interesting pictures, though.

Refraction: This involves generating (recursively) 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 gives 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; reguli-falsi is recommended, but isolate the roots first using a geometric approach. [Many available references. Take a look.]

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 MCS coordinates of intersection point. Computing the diffuse reflectance procedurally as a function of MCS spatial position (x, y, z) can be used to generating solid textures, like wood grain or marble. [Perlin,Hart]

See `http://textures.guinet.com/` for some sample textures.

Bump-mapping: Involves techniques similar to texture-mapping, but the texture functions perturb the object's surface normal, rather than diffuse reflectance, at a given point. [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).

CSG: Extend the raytracer 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 at union nodes merge lists of intersection intervals passed back from the children, at intersection nodes compute the 1D intersection the intervals passed up from the children, etc. 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.

1.5.2 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 epsilon, then return black without checking for further ray hits (obviously goes along with doing 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 DAG and caching transformation matrices, as suggested earlier, is *not* enough to get you credit for this objective.

1.6 The Interface

We have provided a set of stubbed Lua callback functions in C++. They implement a superset of the modeling language used in Assignment 3; you are expected to extend your code from that assignment.

What we list here are the additional functionality you need to add to the Assignment 3 language to get full credit in Assignment 4. You'll find that the Assignment 3 source code already provides stubs for this functionality.

- `gr.nh_box(`*`name`*`, (`*`x, y, z`*`), `*`r`*`)` — Return a non-hierarchical box with name *name*. The box should be aligned with the axes of its MCS, 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 with name *name* of radius r centered at (x, y, z).

- `gr.cube(`*`name`*`)` — Return an hierarchical box with name *name*. The box should be aligned with the axes of its MCS, with one corner at $(0, 0, 0)$ and the diagonally opposite corner at $(1, 1, 1)$.

- ```
  gr.mesh(name, {
          {v1x, v1y, v1z},
          {v2x, v2y, v2z},
  ```
  ```

```
        ...
         {vnx, vny, vnz},
}, {
         {p11, p12, p13, ...  p1m},
         {p21, p22, p23, ...  p2m},
        ...
         {pn1, pn2, pn3, ...  pnm}
})
```

Create a polygonal mesh named `name` with the listed vertices and faces. The first list is a list of vertex coordinates, and the second list is a list of polygons. Each vertex is given as an (x, y, z) triple, and each polygon is a list of integer indices into the vertex list. Vertices are indexed starting at 0.

It may be assumed that polygons are convex and planar. However, polygons may have an arbitrary number of vertices.

- `gr.light({x, y, z}, {r, g, b}, {c0, c1, c2})` — Create a point light source at (x, y, z) of intensity (r, g, b). The attenuation parameters *c0, c1, c2* specify the attenuation for the particular light source according to the formula $1/(c0 + c1 * r + c2 * r^2)$.

- `gr.render(node, filename, w, h, eye, view, up, fov, ambient, lights)` — Raytrace an image of $w \times h$ pixels to *filename*. The camera is to be located at position *eye*, looking in direction *view* with *up* pointing up (all of these quantities are three-vectors). A field-of-view of *fov* degrees is to be used. The ambient light should have an intensity of *ambient* (also a three-vector). All lights to be used in raytracing are listed in *lights*.

The `gr.nh_box` and `gr.nh_sphere` callbacks will allow you to implement a non-hierarchical version of the raytracer, since you can place spheres and boxes in arbitrary locations. Thus you will be able to test aspects of your code such as shading and shadows without necessarily having hierarchical transformations working. Later, you may find it easier to build scenes using `gr.sphere` from Assignment 3 and, say, `gr.cube` for building a unit cube at the origin.

1.7 Donated Code

The Assignment 4 code is similar to that of Assignment 3. There are a few new files in the source directory:

- `a4.cpp` – Contains the `render()` stub you need to implement.
- `image.cpp` – A class storing rectangular images, supporting PNG saving and loading.
- `light.cpp` – A very simple light class.
- `mesh.cpp` – A simple mesh class.
- `polyroots.cpp` – Robust polynomial root solver. You may optionally use this in your ray-intersection functions.

Furthermore, a number of sample scripts are available in the `data/` directory. Some of these require that you run them in that directory, as they depend on other files found there. A simple Lua reader for the Alias/Wavefront OBJ mesh format is also provided (`readobj.lua`) as well as a sample OBJ file (`cow.obj`).

You may want to merge in some of the changes you have made to Assignment 3. For this assignment, however, you do not need to implement a user interface. Your program will be a command-line program.

In Assignment 3, `scene_lua.cpp` provided a function `import_lua` which returned a scene node after parsing. In this assignment, a function called `run_lua` is provided instead. It will take care of parsing the scene, and call `render()` (from `a4.cpp`) as necessary. Once `run_lua` is done, the program will finish executing.

1.8 Deliverables

Executable: `rt` – This should take a scene file as its argument.

Sample Images: For a complete assignment, generate three image files using the following Lua scripts (which can be found in `/u/gr/cs488/data/A4`):

- `nonhier.lua` – a non-hierarchical scene
- `macho-cows.lua` – a hierarchical scene with instances
- `simple-cows.lua` – a simplified version of `macho-cows.lua`.

Only one of `simple-cows.lua` and `macho-cows.lua` needs to be submitted (although you're welcome to submit both). The image files should be in the 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` where you draw the bounding volumes instead of the polygonal meshes. This image should be stored in `nonhier-bb.png` or `macho-cows-bb.png`.

These image files should be in your `cs488/handin/A4/data` directory.

You will find sample renderings of all three scenes in the image gallery on the course web site.

Sample Script: You need to generate a test script called `sample.lua` and render an image from it called `sample.png`. You should place `sample.lua` and all of your images in your `cs488/handin/A4/data` directory. You do not need to submit a special rendering of your sample scene to demonstrate bounding volumes.

This script or another one should demonstrate your "additional feature".

Additional Documentation: Your `README` needs to contain a description of your extra feature and 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: provide two executables) and provide comparative timings. If you use external models, please credit where you got them from.

You need to submit at least one image file: `sample.png`. This image should have a resolution of at least 500x500. You may submit additional images if you wish; mention them in your `README`.

If you could 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.

There will be a bit of subjective grading of the image created from the data file you create. If the image is extremely good, the instructor may award up to 1 point extra credit. If the image is extremely simple, but tests all features, the TAs 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 that feature.

1.9 Objectives: Assignment 4

Due: Friday, March 9th [Week 9].

Name: _____

UserID: _____

Student ID: _____

___ **1:** Objects are visible on the image. This implies that you can generate primary rays, can intersect them with spheres, and can generate a PNG output file.

___ **2:** Cubes and polygonal meshes are properly rendered.

___ **3:** Objects are correctly ordered from back to front.

___ **4:** There is a function that generates a background for the scene without obscuring the view of any of the objects in the scene. This background is on all the generated images.

___ **5:** Diffuse and specular (Phong) lighting has been accomplished.

___ **6:** Shadows have been accomplished.

___ **7:** A script has been supplied that defines and renders a scene different from anyone else's.

___ **8:** Hierarchical transformations operate properly. Spheres and cubes can be transformed with affine transformations.

___ **9:** Bounding volumes (spheres or boxes) have been implemented for polygonal objects as demonstrated by a special renderering as described above.

___ **10:** Some extra improvement has been made to the ray tracer. This may be quite small and range from extra efficiency to extra functionality.[1]

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

[1] State clearly in your `README` what has been done. If you implement a feature that improves the performance of the ray tracer, you must state in your manual execution times for your ray tracer running both with and without the feature.