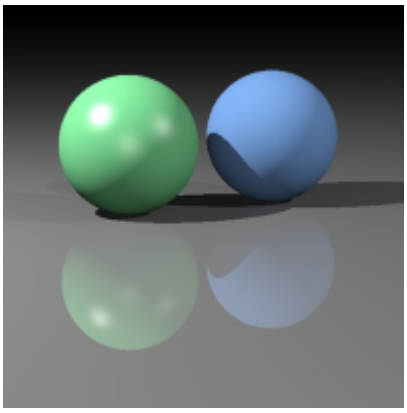
COMP 557 - Fall 2022 - Assignment 4
Ray Tracing

Competition image due before last class

Getting Started



In this assignment, you will write a raytracer. The sample code will get you started with a json scene file parser and code to view the result and write a PNG image file. You are free to **make any modifications** and extensions that you please, to both the JSON format, parser code, and the ray tracing code; however, your code **should remain compatible with the simple examples provided**, and likewise, any changes you make must be well documented in your readme file.

The provided code does not require OpenGL bindings, but requires GLM.

json scene description

The json file is organized as sequence of named materials, lights, cameras, nodes, and geometry. The main scene is defined in the top-level node. In general you will only need to have one node defined, but nodes can also be referred to within the scene graph hierarchy as an instance (i.e., to help you reuse parts of the scene hierarchy multiple times). It is also possible to define the scene with geometry directly, without using the node structure.

The scene nodes each have an associated transformation. The node definition can contain transformation attributes: translation, rotation, and scale (and others if you choose to add them). These transformations are applied, in this order, to build the node transformation (see *Parser.createNodeTree* and **note how the transforms are accumulated** into the matrix *hierarchicalShape.M*). *If you want a different order, consider making a subtree with multiple nodes chained together*. Scene nodes can also be different kinds of geometry (sphere, box, mesh, instance). Finally, each node can also contain a list of child nodes, allowing a hierarchy of transformations and geometry to be built.

Look at the provided examples to get a better idea of how the scene description files are organized. You may wish to develop new test scenes and share them on the discussion board. However, note that you **may need to implement additional tags and attributes** as you proceed through the objectives.

Provided Code

The scene to render must be provided as a command line argument, e.g.,

```
C:/Users/me/COMP557-L04/build> ./Debug/L04.exe ../resources/scenes/TorusMesh.json ../out/TorusMesh.png
```

and note that mesh files referenced within the json files must be specified relative to the build directory.

The main function will initialize the scene and call the scene render method to write the specified image file. You should put your images into a directory `COMP557-L04/out` directory (i.e., to submit with your assignment). This render method is a good place to start making changes to the code, but you will need to make lots of changes to many classes, and make new classes on your own.

You've been provided with basic classes for defining a materials, lights, and nodes, but they do nothing except hold loaded data. A *Ray* class and an *IntersectionData* have been defined for your convenience. You may wish (or need) to change them. They are defined to allow the *Shape* interface to be defined. The sphere, box, plane, mesh, or any other geometry (or node) that can be intersected will implement this interface. Here follows a brief description of each file.

- ***main***

  This is where the application starts and calls the render method.

- ***Shape***

This is the base class for objects in your scene that the rays intersect with. Each shape object has a material, and an intersect method to check for ray-object intersection.

- ***HierachicalShape***

  This is subclass of Shapes, and works as a group node to contain a collection of shapes. It also applies a homogeneous transformation *M* to the ray before intersecting with all of its children.

- ***AABB / Sphere / Mesh / Plane***

  These are subclasses of Shapes. Each have extra information relevant to the object type, such as radius and center for Sphere. The class supports an arbitrary number of materials to be attached to a given shape, however, for this assignment we assume the Plane holds two materials, and all others hold one. You are free to play around with the number of materials you use for custom scenes. ***You need to implement the ray intersection code in each!***

- ***Ray***

  This is a class for a Ray being cast into the scene. It is composed of a eye position and ray direction.

- ***IntersectData***

  This object is passed alongside the ray and is used to store the extra information needed to color and shade the pixel once an intersection is found. It stores information such as the material of the object hit, the intersection point, and the normal.

- ***Scene***

  The scene class contains information about all the intersectable objects, the lights, and extra scene information such as the ambient light. ***This is where the rendering nested-for-loop lives .***

- ***Parser***

  This class is where the json file is parsed into a Scene object. This is already coded for you and will parse the objects (spheres, boxes, planes, etc.), the scene information, the camera position, and material data. ***If you want to add extra objects or parameters, for convenience or for bonus objectives, you will need make changes in this file***.

Ray Tracing Competition: best in show / le lapin d'or

There will be optional competition for images created with your raytracer. This is an opportunity to show off all the features (e.g., extra features) of your ray tracer in an aesthetically pleasing novel scene that you design! To participate, your assignment submission should have a **ID-name-competition.json** scene file for which you also create the image **ID-name-competition.png**, where ID is your student ID and name is your name. Also include a file **ID-name-competition.txt** containing a title and short description of the technical achievements and artistic motivations for your entry. A small jury will judge submissions based on **technical merit, creativity, and aesthetics**. Results will be announced on the last day of class. Note that you must submit these files to a **different assignment box on MyCourses** the night before the last class.

Steps and Objectives

1. ***Generate Rays (1 mark)***

   The sample code takes you up to the point where you need to compute the rays to intersect with the scene. Use the camera definition to build the rays you need to cast into the scene.

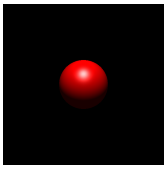
2. ***Sphere Intersection (1 mark)***

   The simplest scene, *Sphere.json*, includes a single sphere at the origin. Write the code to perform the sphere intersection and set the colour to be either black or white depending on the result (i.e., don't worry about lighting in this first step).

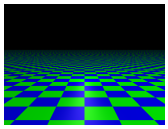
3. ***Lighting and Shading (2 marks)***

   Modify your code so that you're always keeping track of the closest intersection, the material, and the normal. Use this information to compute the colour of each pixel by summing the contribution of each of the

lights in the json file. You should implement ambient, diffuse Lambertian, and Blinn-Phong specular illumination models as discussed in class (note that the specular exponent, or shinyness, is called hardness in the json file).

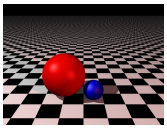


4. ***Plane (1 mark)***



Add code to create an intersectable Plane object at y = 0 (i.e., a ground plane). Planes may have two materials. In the case of a second material being defined, the plane should be tiled with a checker board pattern. Each square in the checkerboard should have dimension 1, and it should be centered at the origin. The first material should be used in squares in the +x +z and -x -z quadrants, while material2 should be used in the +x -z and -x +z quadrants. The *Plane.json* (result shown at right) and *Plane2.json* demo scenes may serve as a useful test at this point.

5. ***Shadows (1 mark)***

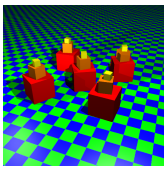


Modify your lighting code to compute a shadow ray, and test that the light is not occluded before computing and adding the light contribution in the previous step. Make sure your shadows work with multiple lights. The *TwoSpheresPlane.json* demo scene may serve as a useful test at this point (see result shown at right).

6. ***Box (1 mark)***

Add code to create an intersectable Box object. The box should be an axis aligned rectangular solid, defined by the min and max corners of the box. You will find the *BoxRGBLights.json* scene a useful test as it sets up different coloured lights in different axis directions.

7. ***Hierarchy and Instances (1 marks)***

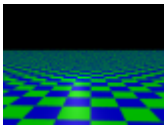


Each scene node has a transform matrix to allow you to re-position and re-orientate objects within your scene. The transformations defined in the scene nodes should transform the rays before intersecting the geometry and child nodes, then transform the normal of the intersection result returned to the caller.

The code in the *HierachicalShape* class implements the *Shape* interface and performs the intersection test on all of its child nodes. If the material of the intersection result is null, then the material of the scene node should be assigned to the result.

The *BoxStacks.json* scene is a useful test of your code when scenes are defined with a heirarchy and instances.

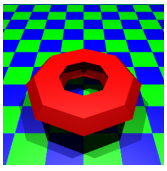
8. ***Anti-aliasing and Super-sampling (1 mark)***



Perform anti-aliasing of your scene by sampling each pixel more than once. The super-sampling technique you use is up to you-- uniform grid, stochastic pattern, or even adaptive. Test your technique with the checkerboard plane in *AACheckerPlane.json*, and note that the high frequency changes near the horizon are difficult to treat.

Note in addition to the `samples` member in scene, there is a boolean member to specify if you should *jitter* the samples. Per pixel jittering (i.e., small amount of sub-pixel displacement of the ray direction) can help replace aliasing with noise, which is helpful even in the absence of super-sampling!

9. ***Triangle Meshes (1 mark)***



The provided code includes the polygon soup loader within the mesh class similar to that used in the previous assignment. You can use this loader, or extend it, or write something new to load the obj file specified in the mesh json nodes. Note that you do not have vertex normals by default, so flat shaded triangles are fine (though Phong shading (interpolated normals) would be nice). The scene file *TorusMesh.json* provides a simple example that you may find useful for testing. Larger meshes, such as the bunny, will probably require some acceleration techniques for practical rendering time. You can also try to run the code in Release mode rather than Debug.

10. ***Create a Novel Scene (1 mark)***

Create a unique scene of your own. Be creative. Try to have some amount of complexity to make it interesting (i.e., different shapes and different materials). Your scene should demonstrate all features of your ray tracer. Be sure to include your name and student number in the filename as described above in the competition information so that it is unique in the class. That is presuming your novel scene is likewise the scene that you will want to submit to the competition.

11. ***Remaining Marks and Bonus (4+ Marks to a max of 9)***

    Implement extra feature in your ray tracer to receive the remaining marks and bonus marks. A combination of several additional features will be necessary to complete and then max out the bonus marks. To receive full marks, your features must be clearly demonstrated to be correct with a test scenes, result image, and a description in your read me file. For anything beyond the list below, the TA and professor will evaluate the difficulty and assign additional marks accordingly. Be sure to document all your additional features in your readme file.

    - Sampling and Recursion
        - Mirror reflection and or Fresnel Reflection (0.5 marks)
        - Refraction (0.5 marks)
        - Motion blur (0.5 marks simple motion, 1 mark complex motion)
        - Depth of field blur (1 mark)
        - Area lights, i.e., soft shadows (1 mark)
        - Path tracing (requires Area lights, 2 marks)
    - Geometry
        - Quadrics, easy! (0.5 marks)
        - Metaballs or other implicits (1.5 marks)
        - Bezier surface patches (2 marks)
        - Boolean operations for Constructive Solid Geometry (2 marks)
    - Textures
        - Environment maps, i.e., use a cube map or sphere map (1 mark)
        - Textured mapped surfaces or meshes (1 mark) with adaptive sampling and or mipmaps (1 more mark)
        - Perlin or simplex noise for bump maps, or procedural volume textures (1 mark)
    - Other
        - Multi-threaded parallelization (0.5 marks)
        - Acceleration techniques with hierarchical bounding volumes for big meshes, e.g., 100K triangles or more (2 marks)
        - Acceleration techniques with spatial hashing and ray marching for big meshes (2 marks)
        - Something else totally awesome (discuss on boards, justify how many marks you want in the readme)

Final Submission Format (read carefully)

Note that there is a different submission box in MyCourses for submitting your novel scene to the raytracing competition!

Your submission should consist of your **code**, a detailed **readme** file, and a the set of **json files** and **png files**. While you do need to have at least one novel scene, and you could probably create one image to show all of your features at once, you (and the TAs) may find it useful if you create a variety of test scenes and associated images that demonstrate any novel (and bonus) features of your raytracer. Your **readme file must include a description of each image/json pair**. Use a zip file (only use zip) matching the directory structure of the provided code.

Note that the TA will also be running your code, but may not have the time to render all your tests (depending on the efficiency or inefficiency of your implementation). Thus, a part of the evaluation of your assignment will be by inspection of images in the submitted documents. Submitting an image that was not generated by your code is considered cheating. It is largely on your honor that the images you show are yours (do not violate this trust).

Finished?

Great! Submit the requested files as a **zip file** (do not use a different type of archive) via MyCourses. Be sure to include a readme file as requested. DOUBLE CHECK your submitted files by downloading them from MyCourses. You can not receive any marks for assignments with missing or corrupt files!

Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code must be your own.