COMP 557, Assignment 4: Ray Tracing

In this assignment, you will be tasked with writing a ray tracer to render scenes described in json files. You are free to modify **any** part of the starter code. However, it should still be compatible with all the provided json files.

Libraries

In this assignment, you will need some new modules: `PyGLM` for vector and matrix manipulation, `libigl` for obj file loading, and `matplotlib` for displaying results. Additionally, the code uses the `tqdm` module to show how much time you have left for your render to complete.

```
python -m pip install -r requirements.txt
```

Json scenes

Each json file defines a scene with key values pairs to specify the image resolution, camera, lights, and a list of `Geometry` objects. The objects list may contain planes, spheres, and cubes, and hierarchies that use the `node` type. See `BoxStacks.json` for an example hierarchy, and for an example of instances where the `ref` key is provided a named geometry as value (i.e., to help reuse parts of the scene hierarchy multiple times).

Scene nodes each have an associated transformation and a list of `children` geometry objects. The node definition can contain transformation attributes: translation, XYZ Euler rotation, and scale (and others if you choose to add them). These transformations are composed with one another, in this order, to build the node transformation (see the `make_matrix` function in `scene_parser.py`). Note that instances are simply nodes where the referenced node is the only child.

Look at the provided examples and the parser to get a better idea of how the scene description files are organized. You may need to implement and define additional attributes and values as you proceed through the objectives. You may likewise wish to develop new test scenes. Moreover, you may consider sharing your test scenes on the discussion board.

Provided Code

The `main.py` file will load the specified input json file(s), and call the render method of `scene.py`. The image returned will be written to a png file with the same file name root as the json file, and by default will be written to the `out` directory in the current directory. You can use the `-o` command line parameter to change the output dir.

When testing or debugging, you will probably want to speed up your results by using the `-f` command parameter to do the render at a factor of the specified resolution, e.g., `-f 0.1`. The default the program does not show the output, but `-s` will show the final render in a matplot window. See likewise that you can provide multiple input files, for instance to render all scenes you might do the following at a linux command prompt:

```
python main.py -i *.json
```

The equivalent command in windows powershell is the following:

```
python main.py -i (Get-ChildItem scenes/*.json).FullName
```

The `helperclasses.py` file provides you with basic classes that define Rays, Intersections, Lights, and Materials. You should make any changes you require for your implementaiton.

The `geometry.py` file defines all the shapes that make up scenes, which all extend the Geometry class. They all implement the intersect method, which should return an `Intersection` object (although this choice is ultimately up to you). A Geometry 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 encouraged to make any modifications or extensions necessary for your custom scenes.

The `scene-parser.py` file reads the provided json file and initializes the described scene. If you want to describe new scene elements, you need to modify the parser and the associated scene initialization. Any changes you make need to remain compatible with the **unmodified** json files provided to you.

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 include 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. Your image should have a resolution of 1280 pixels horizontally by 720 pixels vertically. 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. You'll need to submit your assignment on time to guarantee that your submission is considered.

Objectives

1. Generate Rays (1 mark)

You are provided code that takes you up to the point where you need to compute the rays to intersect with the scene. Use the elements of the scene class related to camera parameters to build the rays you need to cast into the scene.

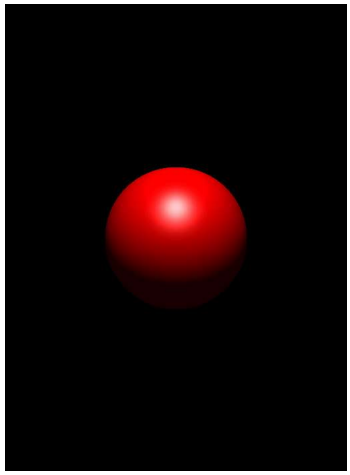
2. Sphere Intersection (1 mark)

The `Sphere.json` scene includes a single sphere at the origin. Write the code to perform the sphere intersection and output black if there was no intersection, and white if there was one.

3. Lighting and Shading (2 marks)

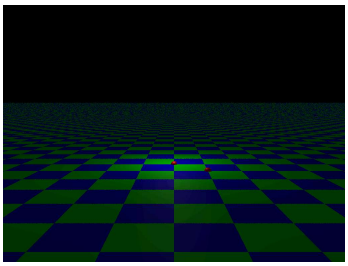
Modify your code so that you're always keeping track of the closest intersection, the material, and the normal. Use this information to compute the colour of each pixel by summing the contribution of each of the lights in the json file. You should implement ambient, diffuse Lambertian, and Blinn-Phong specular illumination models as discussed in class. For that the specular surfaces, use `shininess` as the specular exponent. Lambertian surfaces can be defined by setting the `specular` colour to the zero vector. Also note that the light definition in teh json file has a `power`

value, which simply modulates the colour and is an easy way to dim or birghten lights without changing the colour. Finally, note that point lights should use the specified quadratic function `attenuation` coefficients.



At right you can see the output of `Sphere.json` after properly implementing sphere intersections, lighting, and shading.

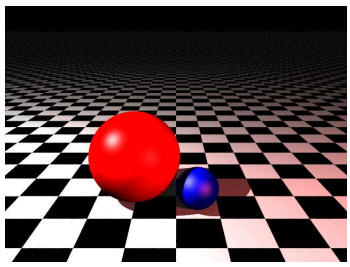
4. Plane Intersection (1 mark)



Add code that will find intersections with a Plane. Planes are allowed to use two materials. If two materials are defined, then they create a checkerboard pattern, where each checker is a 1 x 1 square. Assuming that the plane contains the origin, we expect the point (0, 0, 0) to lie at the corner of 4 squares, where the squares in the +x +z and -x -z quadrants use the first material, while the squares in the +x -z and -x +z quadrants use the second material. You can test your code using `Plane.json` and `Plane2.json`.

At right, you can see the output that should be generated for `Plane2.json`.

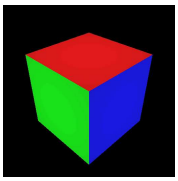
5. Shadow Rays (1 mark)



Modify your lighting code to compute a shadow ray in order to ensure that the light is not occluded before computing and adding this light's contribution. Make sure your shadows work with multiple lights. You can test this step using `TwoSpheresPlane.json`.

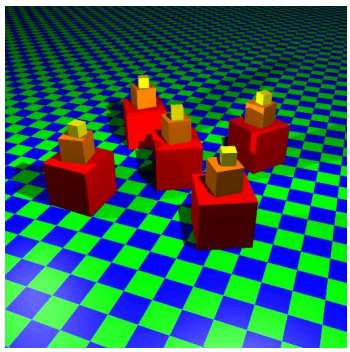
At right you can see the output that should be generated for `TwoSpheresPlane.json`.

6. Box Intersection (1 mark)



Add code in the AABB class that will calculate intersections for axis-aligned bounding boxes, which are defined in our code by the position of two diagonal corners (the min and max corners). You can test this step using `BoxRGBLights.json` and at the right margin you can see the output you should generate for this scene (the box has white reflectance properites, but is illuminated by a red, green, and blue light placed on x y and z axes).

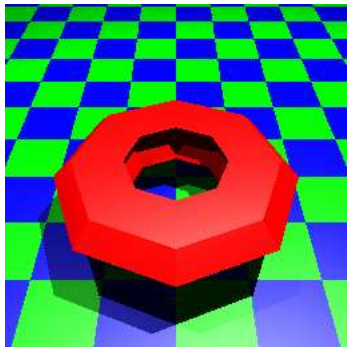
7. Hierarchy Intersection and Instances (1 mark)



Each scene node and has a transform matrix to allow you to re-position and re-orientate objects within your scene. Instances are likewise simply nodes that have the referenced subtree set as a child, and thus have a transformation matrix too. 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.

Add code to the Hierarchy class that will calculate an intersection for 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. You can test this step using `BoxStacks.json`, and you can see the output that should be generated at the right margin.

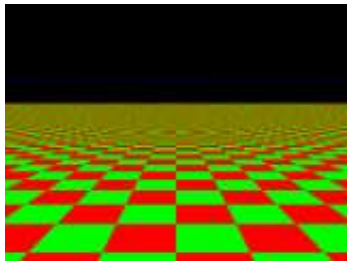
8. Triangle Mesh Intersection (1 mark)



Implement code that will calculate intersections with a Mesh. Note that you do not have vertex normals by default, so flat shaded triangles are fine (though Phong shading using interpolated normals across triangles would be nice if you load meshes with per vertex normals).

You can test this step using `TorusMesh.json`, for which you can see sample output at right. Larger meshes, such as the bunny, will certainly require some acceleration techniques for practical rendering time.

9. Anti-Aliasing and Super-Sampling (1 mark)



Perform antialiasing of your scene by sampling each pixel more than once. The `AA_samples` parameter should be interpreted as the dimension of a uniform subgrid of samples within the pixel, i.e., if 8 is specified, then an 8-by-8 subgrid of 64 samples within the pixel should be collected. You may wish to consider implementing differnt super-sampling techniques, such as adaptive methods. Test your technique with the checkerboard plane in `AACheckerPlane.json` and note that the high frequency changes near the horizon are difficult to treat! Does the red and green checker plane become uniform yellow on the horizon?

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! Only a small amount is needed. Can you identify a good value?

10. A Novel Scene (1 mark)

Create a unique scene of your own. Be creative. Aim to have some amount of complexity to make it interesting (i.e., different shapes and different materials). Your scene should try to demonstrate all features of your ray tracer (see objective 11 below). Be sure to include your name and student number in the filename as described above in the competition information. That said, your novel scene need not be the same scene that you submit to the competition.

11. Go Wild! (4 marks + at most 5 bonus marks)

Implement extra features 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 test scenes, result images, and a description in your readme file. For anything beyond the list below, you may try to justify the marks to be assigned, but ultimately the TAs 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, 1 mark)
- Geometry
    - Quadrics, easy! (0.5 marks)
    - Metaballs or other complicated implicits (1.5 marks)
    - Bezier surface patches, e.g., a teapot! (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
    - Parallelization (e.g., taichi or warp) (1 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)

Finished?

Great! Submit your source code files and your renders of the test scenes as a zip file. Be sure the top of each file has your name and student number, and should there be any other information you wish to include put it in a readme file (e.g., request to use your late penalty waiver).

Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own. Please see the course outline and the fine print in the slides of the first lecture.