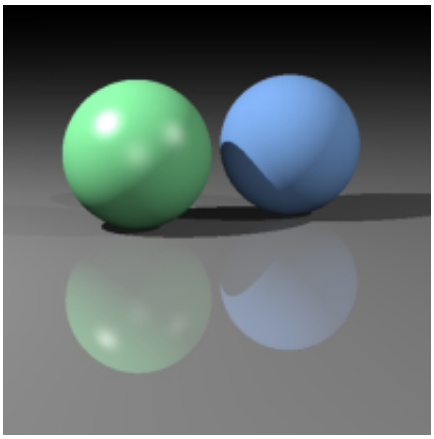
COMP 557 - Fall 2020 - Assignment 4
Ray Tracing

Available Tuesday 9 November

Assignment due 23:30 Monday 30 November

Competition image due 23:59 Monday 30 November

Getting Started



In this assignment, you will write a raytracer. The sample code will get you started with an XML 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 XML 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 the vecmath jar.

XML scene description

The XML 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).

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.createSceneNode* and **note how the transforms are accumulated** into the matrix *sceneNode.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 will definitely want to make new and simple test scenes to help you debug. You **may also need to implement additional tags and attributes** as you proceed through the objectives.

Provided Code

If A4App is run without arguments, it will open a window and display all the xml files in the a4data folder. When you select one of the scenes, it will open a new window to display your image (providing runtime arguments to A4App will skip this interface). It will open the specified xml document, and call the Scene constructor to load the scene definition from the xml file. It then calls the scene render method to produce an image file. 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 *IntersectionResult* have been defined for your convenience. You may wish (or need) to change them. They are defined to allow the *Intersectable* 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.

- ***A4App***

  This is where the application starts. It finds all the scenes in your data folder and populates a scene loader. You can render a scene by doubleclicking on the file name.

- ***Intersectable***

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

- ***SceneNode***

  This is subclass of Intersectable, but contains a collection of Intersectables. It also applies a homogeneous transformation *M* to the ray before intersecting with all of its children.

- ***Box / Sphere / Mesh / Plane***

  These are subclasses of Intersectable. Each have extra information relevant to the object type, such as radius and center for Sphere. ***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.

- ***IntersectResult***

  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 which 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 XML 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-competition.xml** scene file that creates an image labeled **ID-competition.png**, where ID is your student ID. Also include a file **ID-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.

Please submit **exactly** the three named files with **exactly** the specified file names! **Do not zip these 3 files together**.

Gallery of previous winning images

Steps and Objectives

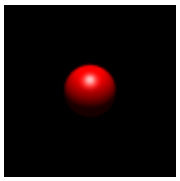
1. ***Generate Rays (1 mark)***

   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. Use the *bgcolor* defined in the xml file for the default background (i.e., when rays do not intersect any objects). You may optionally want to create new xml node to override the background colour with a varying colour based on the ray direction (e.g., to model a sunset, or otherwise reveal information about the viewing direction) as this may help you with debugging.

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

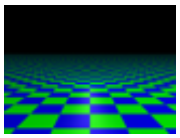
   The simplest scene, *Sphere.xml*, includes a single sphere at the origin. Write the code to perform the sphere intersection and set the colour to be either black or white depending on the result (i.e., don't worry about lighting in this first step).

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



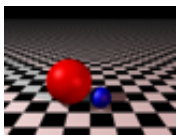
   Modify your code so that you're always keeping track of the closest intersection, the material, and the normal. Use this information to compute the colour of each pixel by summing the contribution of each of the lights in the xml file. You should implement ambient, diffuse Lambertian, and Blinn-Phong specular illumination models as discussed in class (note that the specular exponent, or shininess, is called hardness in the xml 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.xml* (result shown at right) and *Plane2.xml* demo scenes may serve as a useful test at this point.

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

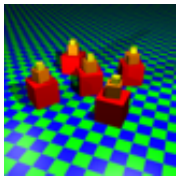


   Modify your lighting code to compute a shadow ray, and test that the light is not occluded before applying computing and adding the light contribution in the previous step. Make sure your shadows work with multiple lights. The *TwoSpheresPlane.xml* demo scene may serve as a useful test at this point (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.xml* scene a useful test as it sets up different coloured lights in different axis directions.

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

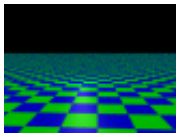


   Each scene node has a transform matrix to allow you to re-position and re-orientate objects within your scene. The transformations defined in the scene nodes should transform the rays before intersecting the geometry and child nodes, then transform the normal of the intersection result returned to the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

   The code in the *SceneNode* class implements the *Intersectable* 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.xml* scene is a useful test of your code when scenes are defined with a hierarchy and instances.

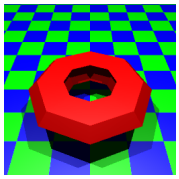
8. ***Anti-aliasing and Super-sampling (1 mark)***



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. The number of samples per pixel is stored as a variable in stored in the *Render* class. It is OK to assume that the number of samples will be a power of 2, or do anything reasonable. If you do adaptive sampling, then note that the samples per pixel could instead be interpreted as a maximum number of samples per pixel (feel free to define additional or alternative parameters as necessary).

Test your technique with the checkerboard plane in *AACheckerPlane.xml*, and note that the high frequency changes near the horizon are difficult to treat. You should also add a boolean member to the Render class to store a attribute **jitter** (modify the parser to set the member on loading a scene). Per pixel jittering can help replace aliasing with noise if there is a regular sampling pattern applied for super sampling at each pixel. Even without super sampling, where one normally would cast a sample in the middle of the pixel, a small amount of sub-pixel jittering is useful to deal with aliasing.

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



The provided code includes a simple polygon soup loader. You can use this loader, or extend it, or write something new to load the obj file specified in the mesh XML nodes. Note that you do not have vertex normals by default, so flat shaded triangles are fine (though Phong shading (interpolated normals) would also be nice). The scene file *TorusMesh.xml* 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.

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. Can you recreate the character you designed in xml? You may want to create a new xml tag for axis angle rotation before you try! Be sure to include your name in the filename as described above in the competition information so that it is unique in the class, and presumably your novel scene is likewise the scene that you will want to submit to the competition.

11. ***Other objectives (4 Marks) and Bonus (2 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 a test scene, 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)
  - Implicit blobby objects (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 **xml 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 may find it useful to 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/xml pair**.

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.

Remember the competition has separate submission instructions and a separate submission box. Review the submission instructions above!