Program 1: Ray Tracing 1.1 CS 5465: Computer Graphics
Updated: January 19, 2024 R. M. Parry

Read Chapter 4: "Ray Tracing" in the textbook before attempting.

This assignment is to be completed individually and the work you submit must be your own. You may edit any of the classes in the framework code and you may create new classes to complete the requirements. However, you do not need and should not use any other libraries.

In this programming assignment, you will build a simple ray tracer. We will start simple with scenes that include only planes and discs. These shapes will have a constant color regardless of lighting. You will also implement a way to visualize the surface normal vectors $\mathbf{n}$ using red, green, and blue to represent x_n, y_n, and z_n values, respectively, using a "normal" light. That is, a "light" that reveals the normal vectors instead of the actual color of a surface.

The framework code provides a way to read the scene information from an XML file, populating your program with a "scene" object with corresponding attributes. Using this information you will implement the ray tracer described in Chapter 4 of the textbook that will:

- construct the camera coordinate orthonormal basis.
- use the basis and pixel coordinates to determine where to cast a ray for each pixel
- intersect that ray with the surfaces in the scene
- color the corresponding pixel with the surface color or normal vector.

Framework code: `https://cs.appstate.edu/rmp/cs5465/ray1.zip`

1 XML Scene Files

For example, the XML file might look like this:

```xml
<?xml version="1.0" ?>
<scene>
  <camera>
    <viewPoint>0 0 0</viewPoint>
    <projDistance>1</projDistance>
    <viewWidth>2</viewWidth>
    <viewHeight>2</viewHeight>
    <viewDir>0 1 0</viewDir>
    <projNormal>0 -1 0</projNormal>
    <viewUp>1 0 0</viewUp>
  </camera>
  <image>200 200</image>
  <light type="ConstantLight"/>
  <material name="white" type="Lambertian">
    <color>1 1 1</color>
  </material>
  <surface type="Plane">
    <material ref="white"/>
    <point>0 1 0</point>
    <normal>0 -1 0</normal>
  </surface>
</scene>
```

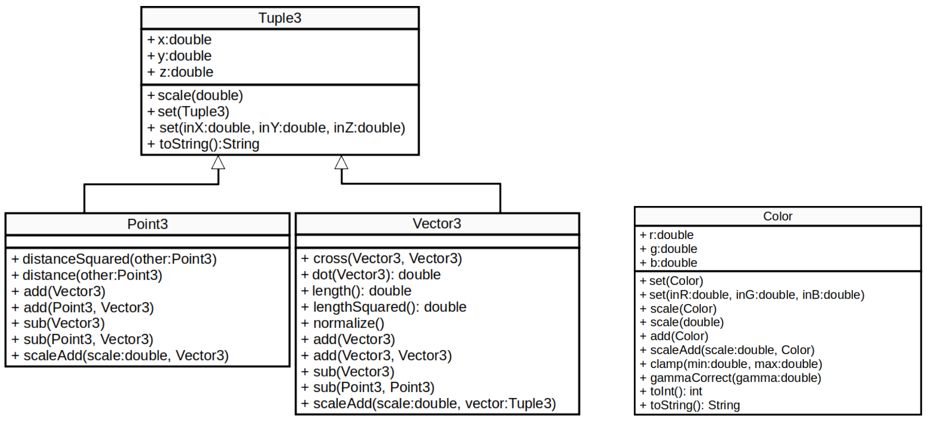
You do not need to worry about reading or writing files. The `Parser` handles creating the `Scene` from the provided XML file. `RayTracer.main()` handles reading XML files or whole folders containing XML files to produce output images from the command line arguments. The output PNG files have the same name as the XML with a PNG extension. In between, it calls the `RayTracer.renderImage()` method that does all the rendering. You will need to edit the `renderImage` method to store the correct colors in the image.

2 Framework Classes

The framework code parses XML scene files and creates the `Scene` object which contains everything you need to render the scene to produce the output image. The `RayTracer.renderImage` method takes the scene as the input argument and fills in the output image with the appropriate colors.

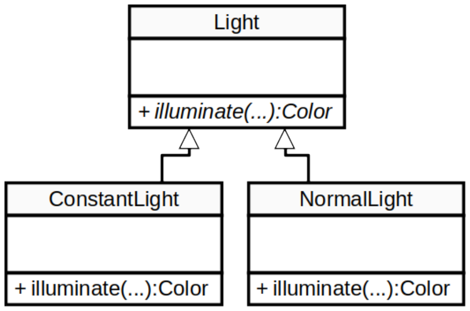
2.1 Three-dimensional Data

In order to render the scenes you will be using `Point3`, `Vector3`, and `Color` classes. `Point3` objects represent points in three-dimensional space, such as vertices in a triangle, the center of a sphere, a point on a plane, etc. `Vector3` objects represent vectors in three-dimensional space, such as surface normal vectors, the displacement between two points, a basis vector, etc.



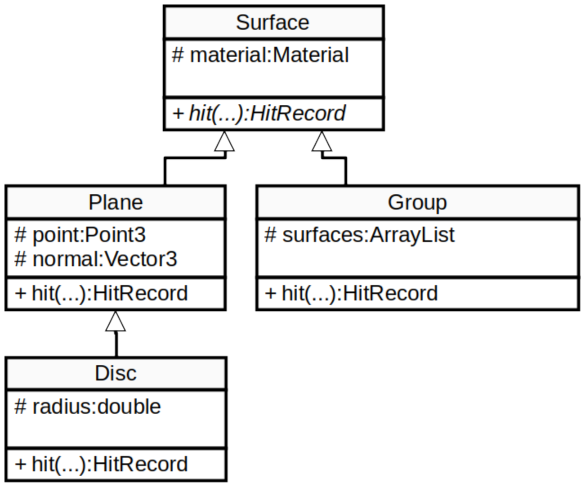
2.2 Lights

Lights in the scene are represented by concrete subclasses of the `Light` class:



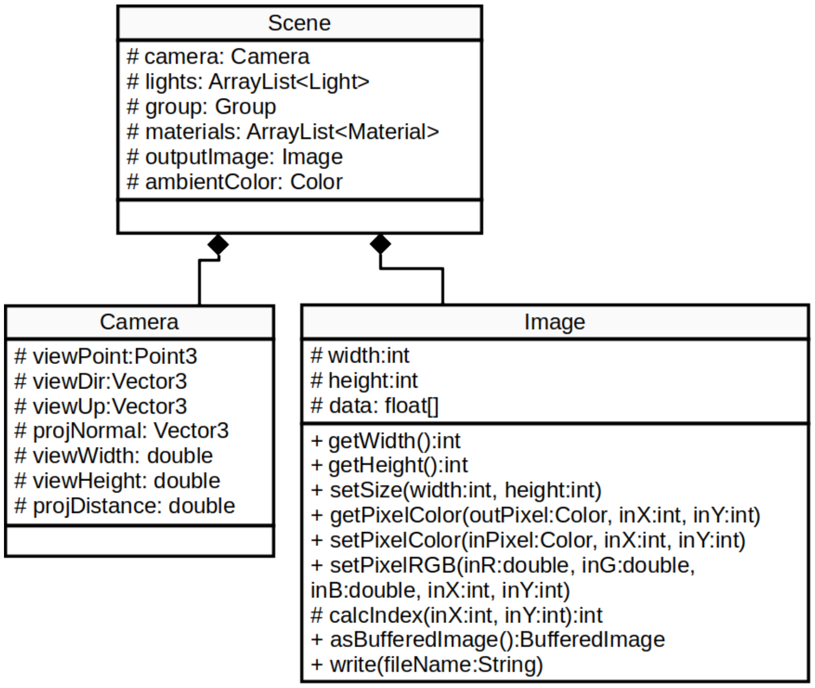
2.3 Surfaces

Surfaces in the scene are represented by concrete subclasses of the `Surface` class:



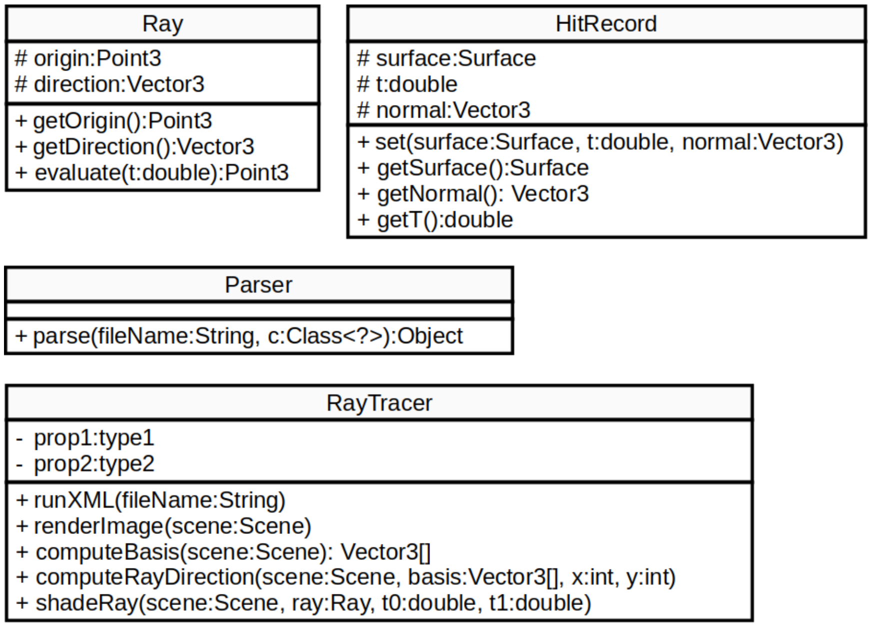
2.4 Scene

The scene contains in a `Camera`, a list of `Lights`, a `Group` of surfaces, a list of `Materials`, an output `Image`, and a `Color` for the ambient light source:


Scene
# camera: Camera
# lights: ArrayList<Light>
# group: Group
# materials: ArrayList<Material>
# outputImage: Image
# ambientColor: Color
Camera
# viewPoint:Point3
# viewDir:Vector3
# viewUp:Vector3
# projNormal: Vector3
# viewWidth: double
# viewHeight: double
# projDistance: double
Image
# width:int
# height:int
# data: float[]
+ getWidth():int
+ getHeight():int
+ setSize(width:int, height:int)
+ getPixelColor(outPixel:Color, inX:int, inY:int)
+ setPixelColor(inPixel:Color, inX:int, inY:int)
+ setPixelRGB(inR:double, inG:double, inB:double, inX:int, inY:int)
# calcIndex(inX:int, inY:int):int
+ asBufferedImage():BufferedImage
+ write(fileName:String)

2.5 Miscellaneous Classes

It is convenient to represent a ray as a `Ray` object and the result of a ray-surface intersection as a `HitRecord`. The `Parser` parses XML and constructs the objects in the scene. The `RayTracer` renders the scene.


Ray
# origin:Point3
# direction:Vector3
+ getOrigin():Point3
+ getDirection():Vector3
+ evaluate(t:double):Point3
HitRecord
# surface:Surface
# t:double
# normal:Vector3
+ set(surface:Surface, t:double, normal:Vector3)
+ getSurface():Surface
+ getNormal(): Vector3
+ getT():double
Parser
+ parse(fileName:String, c:Class<?>):Object
RayTracer
- prop1:type1
- prop2:type2
+ runXML(fileName:String)
+ renderImage(scene:Scene)
+ computeBasis(scene:Scene): Vector3[]
+ computeRayDirection(scene:Scene, basis:Vector3[], x:int, y:int)
+ shadeRay(scene:Scene, ray:Ray, t0:double, t1:double)

3 Casting Rays

3.1 Computing the Camera Basis

In `RayTracer.computeBasis`, use the camera information to construct the **u**, **v**, **w** vectors and return them as an array: $[\mathbf{u}, \mathbf{v}, \mathbf{w}]$. The **w** vector should point in the same direction as the projection normal which is *usually* the opposite direction as the view direction.

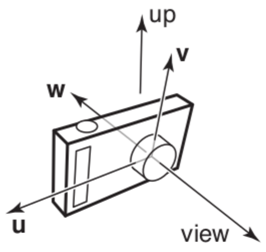
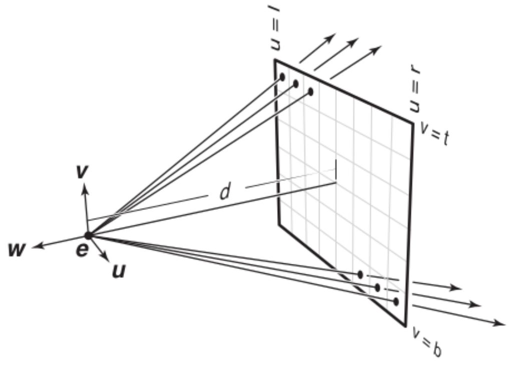


Figure 4.8. The vectors of the camera frame, together with the view direction and up direction. The **w** vector is opposite the view direction, and the **v** vector is coplanar with **w** and the up vector.

3.2 Computing the Ray Direction

In `RayTracer.computeRayDirection`, use the camera, output image, and camera basis to determine the ray direction for the given row (y) and column (x) of the output image. The **w** vector is perpendicular to the image plane. d is the view distance along the view direction. The view width and view height of the camera determine how big the rectangle is in world coordinates. The image width and image height determine how many squares are in the grid. The **u** vector and **v** vector indicate the horizontal and vertical direction of the image plane, respectively. Look at chapter 4 of the textbook for details.



Perspective projection
same origin, different directions

3.3 Rendering the Scene

In `RayTracer.renderImage`, use the scene to fill in the color for each pixel in the image. You will compute the camera basis, cycle through the rows and columns of the output image, produce a ray for each pixel, intersect it with the scene, and color the pixel according to the material properties and light's illumination.

4 Intersection

Every surface needs its own ray-intersection method. For this assignment, you will implement intersections for the `Plane`, `Disc`, and `Group`.

4.1 Ray-Plane Intersection

In `Plane.hit`, implement ray-plane intersection. To intersect a ray, $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, with a plane defined by a point $\mathbf{p}$ and a surface normal $\mathbf{n}$, you can use the implicit surface of a plane:

$$f(\mathbf{r}) = (\mathbf{r} - \mathbf{p}) \cdot \mathbf{n}$$

Points for which $f(\mathbf{r}) = 0$ are on the plane. Given $\mathbf{o}$, $\mathbf{d}$, $\mathbf{p}$, $\mathbf{n}$, you can solve for t using a similar approach to Ray-Polygon intersection discussed in the textbook. Plug in the solution t into the ray equation to get the intersection point.

4.2 Ray-Disc Intersection

In `Disc.hit`, implement ray-disc intersection. The ray hits a disc if the intersection point with its plane is within the radius of the center of the disc.

4.3 Ray-Group Intersection

In `Group.hit`, implement ray-group intersection. A `Group` represents a group of surfaces contained in a list. The hit method cycles through each of the surfaces in the group and intersects the ray with each of them, returning the closest hit among them.

5 Lights

For this assignment, you will implement two "light" sources: `Constant Light` and `Normal Light`. The `Light.illuminate` method returns the color that this light source produces for the surface at the intersection point. For now, that is either the diffuse color of the material or the normal vector at the intersection point.

5.1 Constant Illumination

In `ConstantLight.illuminate`, return the diffuse color of the material for the surface the ray intersected.

5.2 Normal Vector Illumination

In `NormalLight.illuminate` return a `Color` representing the surface normal at the intersection point to help with debugging. To encode the unit-length surface normal, $\mathbf{n} = (n_x, n_y, n_z)$, as a red/green/blue color use the following:

$$R = \frac{n_x + 1}{2}$$
$$G = \frac{n_y + 1}{2}$$
$$B = \frac{n_z + 1}{2}$$

6 Java Development

To work on this program you can use your favorite Java IDE such as Eclipse or VS Code. We will be using **Java 8** with **JUnit4**. Later versions of Java will not work for some of the assignments. The working directory for your project should be the `ray1` folder. The source location should be the `ray1/src` folder.

You can install the JDK for Java 8 here:
`https://adoptium.net/temurin/releases/?version=8`

On linux, you may install OpenJDK version:
`https://openjdk.org/install/`

You can get JUnit4 here:
`https://github.com/junit-team/junit4/wiki/Download-and-Install`

6.1 Running the Program

To run the program on a particular XML file, you can run `ray.RayTracer` as a Java Program with one or more XML files as arguments. Or, you can provide a path to a folder containing XML files as the argument.

The resulting image(s) will be created in the same directory as the XML file with a ".png" extension.

For example, when I run the program in VS Code I get:

```
parryrm@parryrm-OptiPlex-7050:~/Documents/cs5465/vscode/ray1$  cd
↪  /home/parryrm/Documents/cs5465/vscode/ray1 ; /usr/bin/env
↪  /usr/lib/jvm/java-8-openjdk-amd64/bin/java -cp
↪  /home/parryrm/.config/Code/User/workspaceStorage/6d62e398596cafc081fb78d350c154d0/redhat.java/jd
↪  ray.RayTracer scenes/plane-constant/rubiks-blue-plane-constant.xml
scenes/plane-constant/rubiks-blue-plane-constant.xml
Done.  Total rendering time: 0.026 seconds
Writing scenes/plane-constant/rubiks-blue-plane-constant.xml.png
```

6.2 Running the Tests

Use the standard way to run JUnit4 tests in your IDE to run the tests in `tests.Ray1Test1`.

(Program 2 will use `tests.Ray1Test2`.)

For example, when I run the tests in VS Code I get:

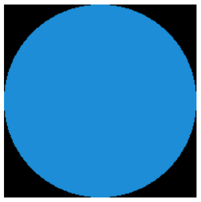
```
Debug Console:


scenes/mario/mario-red.xml
Done.  Total rendering time: 0.161 seconds
scenes/mario/mario-red.xml : rmse = 0.0
.
.
.
scenes/multiple-discs-normal/rubiks-blue-white-orange-disc-normal.xml
Done.  Total rendering time: 0.004 seconds
scenes/multiple-discs-normal/rubiks-blue-white-orange-disc-normal.xml : rmse = 0.0
```

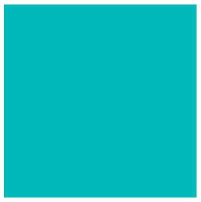
Again, the resulting image(s) will be created in the same directory as the XML file with a ".png" extension.

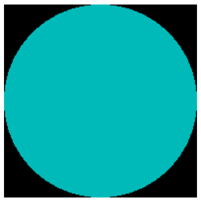
7 Examples

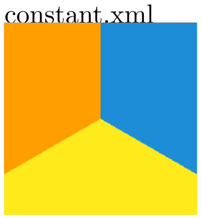
Here are some of the test cases provided in the `ray1/scenes` folder. You can click on the image to download the full resolution image file or click on the XML file to download it:

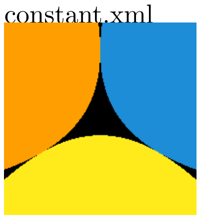
plane-constant/rubiks-blue-plane-constant.xml

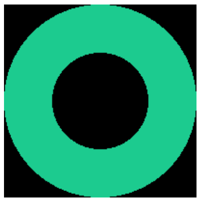

disc-constant/rubiks-blue-disc-constant.xml

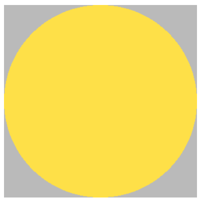

plane-normal/rubiks-blue-plane-normal.xml


disc-normal/rubiks-blue-disc-normal.xml


multiple-planes-constant/rubiks-blue-yellow-orange-plane-constant.xml


multiple-discs-constant/rubiks-blue-yellow-orange-disc-constant.xml


multiple-objects/rubiks-green-ring.xml


multiple-objects/plane-disc.xml

mario/mario-rgb.xml

mario/mario-rgb-precise.xml

8 Web-CAT Submission

Your assignment will be graded by Web-CAT. ZIP your `src` directory and upload it here:
`http://webcatvm.cs.appstate.edu:8080/Web-CAT`