Theory and Practical Tips

Contents

Geometry

Vectors (Vec class)

In ex2 you've implemented this class. You should import it to your project under ex3.math.

Points

You should implement a Point3D class which represents a point. As you know, although both vector and point can be represented by the same three coordinates, there is a big difference between them. Point3D class should be located in ex3.math.

Your Point3D class should hold 3 coordinates (x, y, z). In addition your point class should be able to perform basic point calculations. ***Note: your Point3D Class should not extend your Vec class!*** *It is ok to make them both inherit the same abstract class.*

Geometric primitives

You need to support 3 primitives (spheres, discs and convex polygons). From these 3 primitives you can create a great variety of objects. Intersection calculations for spheres and convex polygons are described in the lecture notes and recitations. Discs are similar, figure it out on your own.

- ### Sphere

  A sphere is defined by a center point and scalar radius. The normal of each point on the sphere's surface is easily computed as the normalized subtraction of the point and the center.

- ### Disc

  A disc is just a flat filled circle in 3d space. It is defined by a center point, scalar radius and a normal vector to the plane that it lies in. A disc should be visible only from its front side (the side where the normal points at).

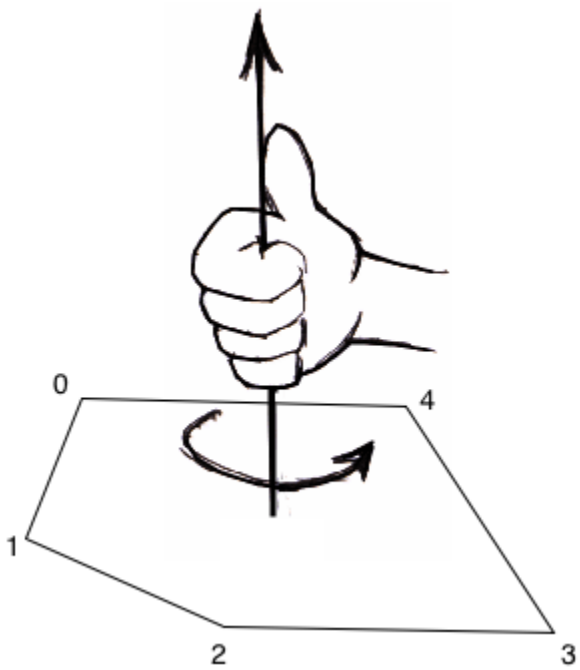
- ### Convex Polygon

  A closed shape whose boundary consists of a set of straight edges is called a *polygon*.

  Every 2 points inside a **convex polygon** define a straight line which is inside the polygon. Notice that for a convex polygon all interior angles are less than 180 degrees.

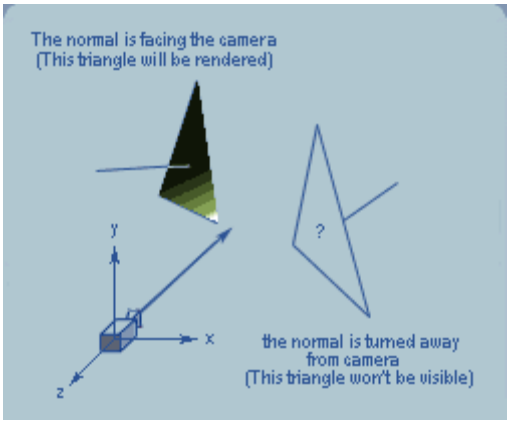
  The polygon is defined by its points and their order. For example $p_0p_1p_2$ means that there are edges p_0p_1 p_1p_2 p_2p_0.

  The normal of the polygon will be directed according to the right hand rule: curl your fingers in the direction of the ordered points and your thumb will point in the direction of the normal:

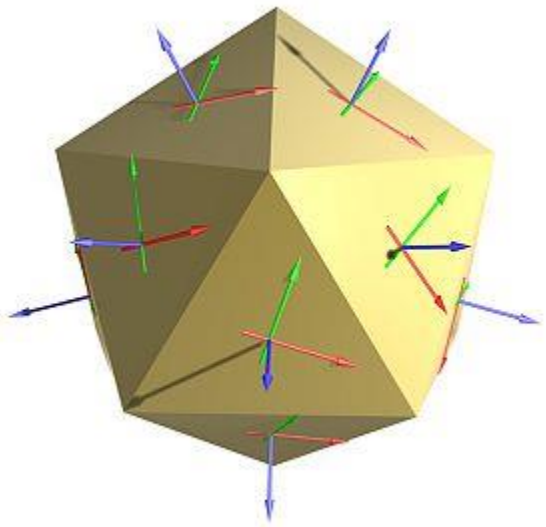


You can calculate it through cross product between 2 of the edges, be careful when you calculate their direction!

A polygon should be visible only from its **front** side (the side where the normal points at). The other side is called **back face**



Note: The order geometrically is defined in a **counter-clockwise** way, $(p_0p_1p_2p_3p_4)$ in the example, this is extremely important when you deal with closed shapes, that way the normal will point outwards of the shape and the shape will be visible in the rendered image- the faces will be front faces!



- ## Your scene out of Convex Polygons

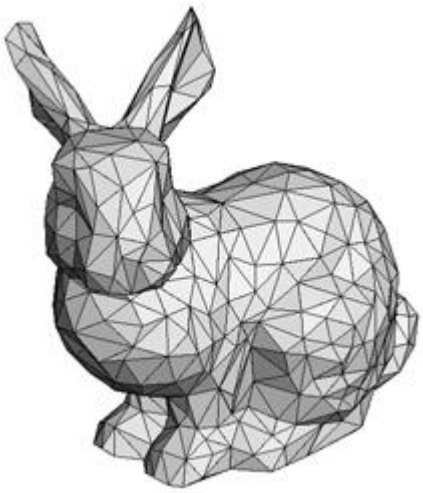
You can design any scene you like, including **3d volumetric shapes**, from convex polygons. Remember ordering the vertices in a counter clockwise way so that normal will point outwards of the shape as described above.

For instance you can render a box, a pyramid, a parallelepiped or anything you want.

- ## Triangular Mesh from obj file – (Bonus)

A triangle is a convex polygon defined by 3 points. A triangular mesh is just a collection of triangles, which can be interpreted as the surface of a 3d volumetric shape. A mesh class includes a list of triangles.

Read how to load a triangular mesh from an obj file and render cool shapes that you can download from the internet. First render shapes with few triangles (like a pyramid), the more triangles you render the more you wait..



Background

The background of the rendered scene is either a flat color or an image scaled to the dimensions of the canvas.

Physics

Camera

The camera is a simple pinhole camera as described in the lecture slides.

A few notes you should remember about the camera:

- You need to either specify a direction or a look-at point but not both. Specifying a look-at point implicitly sets the direction and if you set the direction there is no need to specify a look-at point. The direction can be calculated using the eye and look-at points. you don't have to check that programmatically, just be aware.
- The direction and the up vector of the camera need to be orthogonal to each other (90 degrees between them) however the vectors specified in the file may not be orthogonal. You need to pre-process these vectors to make them orthogonal. This can always be done if they are not co-linear. (use a cross product to find the right vector and another cross product to find the real up direction, like we did in rec4).
- Only the screen width is specified. The screen height needs to be deduced according to the aspect-ratio of the canvas.

Lighting

Basic lights and shadows

For basic lighting you need to implement:

1. Ambient lighting – a color that reflects from all surfaces with equal intensity.
2. Directional – A light source like the sun, which lies at infinity and has a direction. **This kind of light shouldn't cast shadows**.
3. Omni-direction point light – A point light source that illuminates in all directions, which intensity fades as the distance from the light source increases using the parameters I_0, k_c, k_l, k_q as was explained in class (and its slides).
4. Spot light – a point light source that illuminates in a direction given by a vector D, which intensity fades as the distance from the light source increases using the parameters I_0, k_c, k_l, k_q as was explained in class (and its slides)
5. Self-emittance – each surface can also have a color component that is emitted from within the surface.

Every light source has its own intensity (color) and there can be multiple light sources in a scene. Shadows appear where objects obscure a light source. In the equation they are expressed in the S_i term. To know if a point p in space (usually on a surface) lies in a shadow of a light source, you need to shoot a ray from p in the direction of the light and check if it hits something. If it does, make sure that it really hits it before reaching the light source and that the object hit is not actually a close intersection with the object the ray emanated from. Some common mistakes may cause spurious shadows to appear. Make sure you understand the vector math involved and all the edge-cases.

Materials

You need to implement the lighting formula (Phong) from the lecture slides. The material of a surface should be flat with ambient, diffuse, specular reflections, emission parameter and a shininess parameter (the power of $V \cdot R$). The first 4 parameters are RGB colors.

Reflection

This is where ray-casting becomes ray-tracing. If a material has reflectance (K_S) greater than 0, then a recursive ray needs to be shot from its surface in the direction of the reflected ray. Say the ray from the camera arrives at point p of the surface at direction v. Using the normal n at point p, you need to calculate vector R_v which is the reflected vector of v. This can be done using simple vector math which was seen in rec2. Using R_v, you recursively shoot a ray again, this time from point p. Once the calculation of this ray returns you multiply the color returned by the reflectance factor K_S and add it the color sum of this ray as explained in class.

Refraction (bonus)

Look into Snell's law: http://en.wikipedia.org/wiki/Snell's_law
Refractive index table: http://en.wikipedia.org/wiki/List_of_refractive_indices

Practical Tips

- There is always an issue with 'double' calculation! Once implementing the exercise, you will get some artifacts that the returning ray will hit the object it already hit multiple time – to overcome this you can use a tolerance value or move the ray just a little ahead before continuing to the next intersection.
- A color can be seen as a vector that points from black (0,0,0) to a point in the RGB space. Use the Vec class to handle vectors and RGB.
- One of the first things you need to do is map the canvas coordinates you receive from the renderer to the scene coordinate system. You do this using the camera parameters.
- Before plotting a pixel, make sure that its color does not exceed the range of 0-255 for every color channel. The scale for each value of a color is from 0 (no color) to 1 (most color) for display it needs to be stretched back to 0-255.

Algorithmic enhancements (bonus)

Super Sampling (bonus)

If you only shoot one ray from each pixel of the canvas the image you get will contain aliasing artifacts (jagged edges). A simple way to avoid this is with super sampling. With super sampling you shoot several rays through each pixel. For each such ray you receive the color it's supposed to show. Then you average all these colors to receive the final color of the pixel. In your

implementation you will divide every pixel to grids of 2x2 or 3x3 etc' of sub pixels and shoot a ray from the center of every such sub pixel. In fact, the parameter that will control super sampling will be an integer k that tells how many vertical and horizontal rays are casted per pixel ($k = 1$ means no super sampling, $k = 2$ means 2x2=4 rays per pixel etc.)

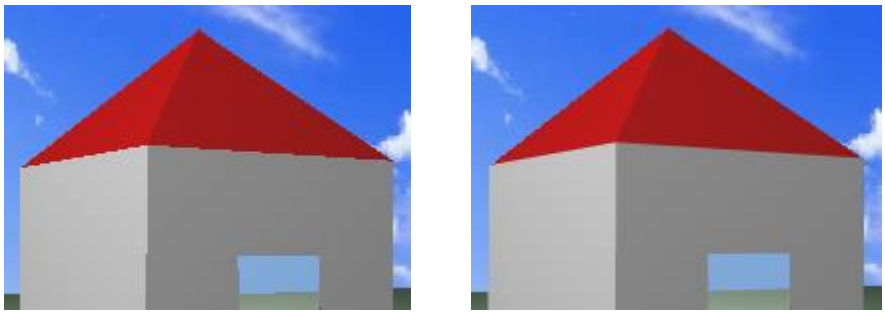


Figure 1 On the left an image rendered without super sampling, on the right with super sampling.

Acceleration (bonus)

When having many objects in a scene, ray-tracing consumes a great deal of computation time on intersection calculations. Acceleration methods amend this by ruling out intersections prior to the actual intersection test. This prior computation takes a small overhead but on average greatly increases performance (as most intersections are ruled out). One acceleration method that you may implement is the hierarchical bounding boxes method which is addressed in the lecture slides (see also next section). A bounding box for an object is defined as the smallest axis aligned box that contains the object. When rendering, before testing intersection between ray and object, one checks intersection between ray and object's bounding box. If there is no intersection with the bounding box then we know for a fact that there is no intersection with the object. Otherwise we proceed to test for intersection with the object. Note that ray intersection with an axis aligned box can be computed very efficiently. To use this acceleration method one must compute every object's bounding box in preprocessing. Being an axis aligned box, the bounding box can be defined by two points; One point is obtained by taking the minimum over all x,y,z coefficients of the surface points, and the other point is obtained by taking maximum.

Hierarchal Bounding Boxes

Primitives in this ray-tracer are very simple and their intersection computation is not much more expansive than calculating bounding box intersection. Thus a more suitable way for acceleration is to merge groups of bounding boxes together, forming larger bounding boxes and creating a hierarchy. Ruling out intersection with bounding boxes at the top of the hierarchy will be very beneficial. There are many heuristics for forming bounding box hierarchies. Some can be based on fixed space partitioning (quad-tree-like approach) others on clustering methods.

One such method is Hierarchical clustering: For each level in the hierarchy, merge pairs of bounding boxes having the smallest merging error. Merge error can be measured by the amount of empty space introduced by the merge (that is the volume of the parent bounding box minus the volumes of the two children). You should discuss your choice and implementation of the acceleration method shortly in the doc accompanying your submission.

Appendix – FAQ

Q: What should I do if a ray (light/viewer's eye) hits the back side of a triangle?

A: You just ignore this intersection. The ray continues as if there was no intersection. In both cases – if you defined your object meshes properly, then there should be a front facing triangle intersecting the ray closer, and therefore you can discard these intersections and move on to the next ones! For an example, open the scenes folder, go to primitives, and see the rendered tri_front and tri_back scenes.

Q: What should I do if a ray (light/viewer's eye) hits the back side of a disc?

A: The front side of the disc is defined by the disc's normal. If a ray hits the back side, it should be ignored.

Q: How do I make java find the texture files? There is no way to change the current directory.

A: Use the path from the scene text file you loaded. For example `String path = new File(filename).getParent() + File.separator;`

Q: What other library classes can I use?

A: You shouldn't need to be using anything other than basic math and vector calculations. All of the external dependencies are taken care of by the supplied wrapper.

Q: How do I calculate R, the reflection vector?

A: Have a look at the last slide of rec2. Also, this page: http://www.cs.umbc.edu/~rheingan/435/pages/res/gen-11.Illum-single-page-0.html contains (among other things) a nice explanation of this.

Q: How can I debug my raytracer? I keep staring at the code but I can't figure out what's wrong.

A: An easy way to start debugging is to find a pixel where you know something is wrong and try to debug the calculation of that specific pixel. Say you found there is a problem at (344, 205), You can start by writing something like this:

```
if (x == 344 && y == 205)
      System.out.print("YO!"); // set breakpoint here.
```

in your main loop, and then setting a breakpoint at that print line. From this point in the execution you can follow what exactly leads to this pixel being the color it is.

Q: What can I use to edit XML files?

A: Any text editor (e.g. notepad) is sufficient for the task. However some editors also support syntax highlighting and validation (e.g. notepad++) which might be useful. Some versions of Eclipse also have a fancy built-in XML editor.

Q: Do I need to constantly reload the scene to see changes?

A: No. You should load the scene once and every time you render the scene will be automatically reloaded.