CSci 490, Spring 2005, Assignment 1

This assignment, worth 40 points, is due at 3pm, Friday, January 28. *Submit it by attaching your modified files to an e-mail to* `cburch@cburch.com`.

This assignment involves writing two important pieces to a rather sophisticated Java program for walking through a building. This program employs the *ray casting* technique for rendering the walls; this is the basic technique pioneered by Wolfenstein 3D, which was enhanced in Doom and several subsequent game engines.

Like Wolfenstein 3D, our engine is based on some important limitations: We require the floor to be level (no stairs); all surfaces must be vertical; all surfaces must have the same height; and the user must always look straight ahead.[1] These assumptions render the vertical dimension irrelevant, flattening the 3D space into a 2D space.

You can download the supporting files from the class Web page. The files include:

- several Java source code files, with some key methods in the View and Cylinder classes left undefined. You'll implement these methods for this assignment.
- documentation for these classes, as generated by *javadoc*.
- two "world" data files, `simple.map` and `pillars.map`, describing worlds you can use for testing your program. You can of course create your own worlds.
- several texture files, beginning with "`txt-`" and ending in "`.gif`" or "`.jpg`." These files are referenced in the world files.
- a JAR file (`tour.jar`) containing a compiled version of the completed program. You can use this to see how your program will end up working. To run it, you will need to include the world file as a command line argument.

  ```
  java -jar tour.jar pillars.map
  ```

  The main window displays a first-person view from your current location. The inset in the upper right maps the world including your current position. The numbers in the lower left indicate your current position and orientation.

  Note that you can move using the arrow keys. The left and right arrow keys rotate your view, and the up and down keys move you forward and backward.

The program works as distributed, but the first-person view will be absent. To compile and run your program, type:

```
java Main simple.map
```

Part I (25 pts)

Complete the View class's `fillPixels` method, which implements the fundamental ray casting technique. You'll find the classes of Table 1 useful in completing this. With the method complete, you should be able to see a first-person view of `simple.map`. (The `pillars.map` file is intended for testing Part II.)

The `fillPixels` method takes four parameters. The first, a User object, gives information about the user's context. The second, an `int[]`, is an array of pixels that should be filled with an image of the user's view. The third and fourth parameters give the dimensions of this image. The `pixels` parameter is one-dimensional array that includes a single `int` for each pixel of the two-dimensional image, with entry `pixels[0]` being the top left pixel, `pixels[width]` being the pixel just below it, and

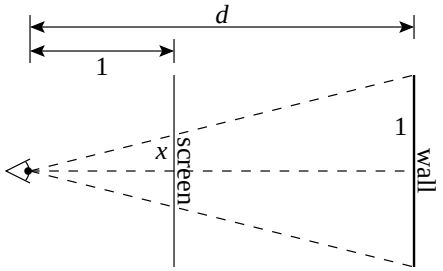
[1] The Wolfenstein 3D engine is even more limited: It also requires all surfaces to be unit cubes. Also, Wolfenstein 3D lights all surfaces uniformly, whereas your engine will be much more sophisticated in how it lights surfaces.

User	Represents the user's location, including a reference to the World in which the user is located.
World	Represents all the information about the world map.
Surface	Represents a single vertical surface in the world. At present, there are two complete implementations of this interface, LatWall (east-west walls) and LongWall (north-south walls). You'll complete the Cylinder implementation in Part II.
Ray	Represents a two-dimensional ray, with a starting point (x, y) and a direction (dx, dy).
Intersection	Represents information about where a ray strikes a surface in the world.
TextureColumn	Represents a vertical strip of a texture.

Table 1: Classes useful for Part I.

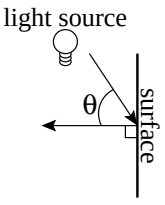
`pixels[width * height - 1]` being the bottom right pixel.[2] (In case you're wondering, though you don't need the fact for this assignment, each `int` represents a color by packing four 8-bit values together: The highest 8 bits are the pixel's opaqueness value (you'd always use 0xFF to be entirely opaque), then the next 8 bits are the pixel's red value, then its green value, and finally its blue value.)

The ray-casting algorithm works as follows. Each x-coordinate in the image corresponds to a direction in the 2D plane. The algorithm determines the closest surface in that direction. If it is a distance d away, then we draw the wall $2/d$ pixels tall, centered vertically. This way, surfaces that are far away appear smaller than surfaces that are closer. That this is the correct height to draw the wall can be deduced from the following diagram.



Because the triangles are similar, $x/1$ must equal $1/d$. Thus the length of the wall's image as projected onto the screen is $2x = 2/d$.

For lighting, assume that the only light in the world is a torch that the user holds. This assumption allows us to simulate real lighting while avoiding shadows (which complicate the process considerably). We will assume that the walls reflect light diffusely. (That's a reasonable assumption for rough-hewn rocks, but not for polished surfaces.) To compute the diffuse reflecion, suppose we have the following.



If the intensity of light from the source strikes the surface with intensity i, and if θ is the angle between the light ray striking one point on a surface and the surface's normal at that point, then the intensity of light

[2] A two-dimensional array of Color objects would be more intuitive. But the Java API for working with images directly uses this one-dimensional array of `int`s instead, presumably for performance reasons. We stick with it because doing any conversion would decrease performance. In other words, don't blame me for this non-intuitive representation.

reflected at that surface point is

$$i \cos \theta$$

Essentially, this expression says that a surface reflects the most light when the light strikes it head-on. When it strikes the surface at an angle, then a square foot of light is spread over more than a square foot of surface.[3] Computing $\cos \theta$ is simple: Assuming that that both rays' directions have length 1, then the the cosine of the angle between the rays is their directions' dot product.

You will also want to account for the fact that a torch will light objects farther away more dimly than objects close up. You can compute this directly imagining the torch at the center of a sphere of radius r. The torch's total intensity i will be spread over the sphere's surface of $4\pi r^2$. Thus, a point on the surface receives $\frac{i}{4\pi r^2}$ intensity. This model, though, fails to take into account secondary reflections of the light from other points. In practice, a decay of $1/r$ rather than $1/r^2$ generates better images.

Putting these two facts together, we get that the intensity at a surface at distance d, with an angle θ between the surface normal and the light (which is of intensity i) is

$$\frac{i \cos \theta}{d} .$$

Remember that $\cos \theta$ is simply a matter of finding the dot product of the two rays' directions.

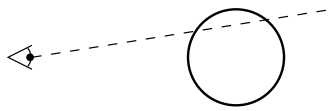
Incidentally, using the same brightness level for the entire vertical stripe of the surface is technically incorrect, since as we go up and down the wall both θ and d will change. But we will not take this into account, because using the same intensity simplies computation considerably without causing any perceptual problems.

Part II (15 pts)

Note: *I recommend proceeding to Part II only after you have completed tested Part I.*

Implement support for vertical cylinders. In particular, complete the blank methods of the Cylinder class, `getDistanceFrom` and `getNormal`.

Of course, since ray casting allows us to compute in a 2D world, you really only have to worry about simple circles. For the `getDistanceFrom` method, imagine a world like the following.



Suppose the eye is at (u_x, u_y) and is looking in the direction (Δ_x, Δ_y). Thus, a point at distance d along the ray has the coordinates

$$(u_x + d\Delta_x, u_y + d\Delta_y) .$$

Suppose, moreover, that the cylinder has its center at (c_x, c_y), and has a radius of r; thus, the points on its circumference satisfy the equation

$$(x - c_x)^2 + (y - c_y)^2 = r^2 .$$

You want to find the smallest value of d for which the point at distance d along the ray intersects with the cylinder's circumference. This is simply a matter of solving the quadratic equation.[4]

Enjoy the assignment, and if you have any questions or problems, please don't hesitate to talk to me!

[3] Incidentally, this fact is the reason that upper latitudes are colder than lower latitudes. Also, coupled with the earth's tilt, this fact is the primary reason that winters are cold.

[4] If you've forgotten, the solution to $ax^2 + bx + c = 0$ is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$