Project Amiga Juggler

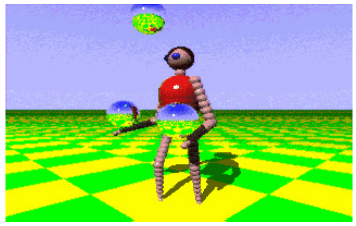
Step 0

Programming—especially when you are just starting out, learning how to code—endows you with a sense of control, a sense of power. Often, it's like playing a God game, at least when you do it recreationally. Of course, you need to spell out to extremely high precision exactly what you want the machine to do. But, then it's forced to carry out your instructions. It's forced to obey.

You can derive a similar sense of fun from mathematics. You can force math to do amazing things on your behalf. Of course, you need to figure out the right equations, but once you code them into your programs, the formulae are under your control. They too will obey.

To show you what I mean, let's do some recreational programming. Let's recreate the Amiga Juggler from scratch using core Java.

In the fall of 1986, Eric Graham discovered that the Amiga 1000 personal computer, running at a meager 7.16 MHz, was powerful enough to ray trace simple 3D scenes. His test animation consisted of an abstract, humanoid robot juggling 3 mirrored balls. It stunned the Amiga community including the executives at Commodore, who later acquired the rights to the animation for promotional purposes.



I want to share some of the pain that Eric Graham went through by coding this from first principles. To accomplish this, I'll have to develop a mini ray tracer, a program that generates images by simulating the optical properties of light interacting with an environment. I'll delve into linear algebra, geometry, trigonometry and a bit of calculus. Hopefully, you vaguely remember learning about these topics back in school. Let's review.

Step 1

A three-dimensional vector is simply a list of 3 numbers. It can represent a point in space, a direction, a velocity, an acceleration, etc. It can even represent a color in RGB color space. For coding simplicity, I'll represent a vector as an array of doubles of length 3. An object with 3 double fields is probably more efficient, but I'm trying to minimize the amount of code to write.

Many vector functions act on the individual components independently. For example, to add 2 vectors together, you simply need to add each pair of components together.

```
// a = b + c
public static void add(double[] a, double[] b, double[] c) {
  for(int i = 0; i < 3; i++) {
    a[i] = b[i] + c[i];
  }
}
```

The magnitude of a vector (the length of a vector) is computing using the 3-dimensional version of the Pythagorean Theorem.

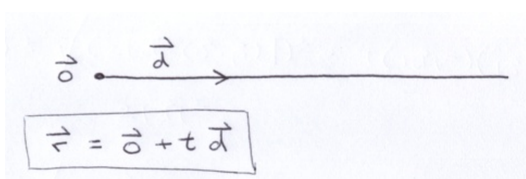
```
// |v|
public static double magnitude(double[] v) {
  return Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
}
```

You can scale a vector (change its length) by multiplying it by a constant (a.k.a. a scalar).

```
// a = b * s
public static void scale(double[] a, double[] b, double s) {
  for(int i = 0; i < 3; i++) {
    a[i] = b[i] * s;
  }
}
```

You can negate a vector by scaling by −1, reversing its direction. If you divide a vector by its magnitude (scale by 1/magnitude), you end up with a unit vector (a vector of length 1).

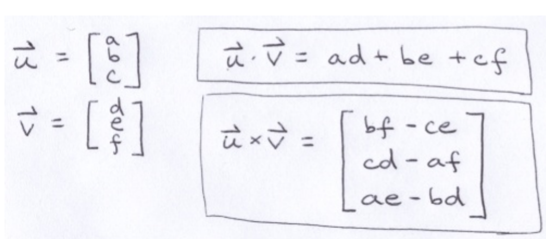
Vectors can do cools things. Consider an astronaut floating in space at some origin point **o** that fires a gun in direction **d** (a unit vector). The bullet leaves the gun and it continues to travel indefinitely in a straight line at a constant velocity (a unit of distance for every unit of time). Its position at any time is given by the ray equation.



Above, a vector is scaled by a constant and the result is added to another vector.

```
// a = b + c * s
public static void ray(double[] a, double[] b, double[] c, double s) {
  for(int i = 0; i < 3; i++) {
    a[i] = b[i] + c[i] * s;
  }
}
```

Most vector functions are intuitive, but 2 are mysterious: dot-product and cross-product.



You may recall this trick involving a determinant that is helpful for remembering how to perform cross product:

$$\begin{vmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ a & b & c \\ d & e & f \end{vmatrix} = \begin{vmatrix} b & c \\ e & f \end{vmatrix}\hat{\imath} - \begin{vmatrix} a & c \\ d & f \end{vmatrix}\hat{\jmath} + \begin{vmatrix} a & b \\ d & e \end{vmatrix}\hat{k}$$
$$= (bf-ce)\hat{\imath} + (cd-af)\hat{\jmath} + (ae-bd)\hat{k}$$

The notation used in the trick actually reveals a little bit about the historic origin of dot product and cross product. They come from a number system invented by William Hamilton in 1843 called quaternions. Hamilton attempted to extend complex numbers by introducing a new imaginary-like constant **j**, but he could not work out a self-consistent system. According to legend, while out on a walk in Dublin by the Brougham Bridge, formulae that would work popped into his head. He was so overcome by excitement in his revelation that he impulsively vandalized the bridge by scratching the equations into the stonework (WTF?).

Hamilton's solution involved introducing 2 new imaginary-like constants:

$$\hat{\imath}^2 = \hat{\jmath}^2 = \hat{k}^2 = \hat{\imath}\hat{\jmath}\hat{k} = -1$$

You can multiply pairs of those constants in 6 different ways. Below, note that the commutative property of multiplication does not work here (order matters).

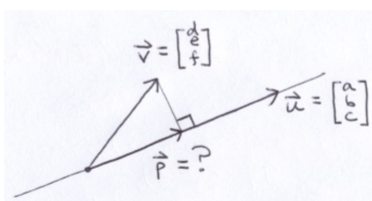
$$\begin{array}{ll}
\hat{\imath}(\hat{\imath}\hat{\jmath}\hat{k}) = -\hat{\imath} & (\hat{\imath}\hat{\jmath}\hat{k})\hat{k} = -\hat{k} \\
\hat{\imath}^2\hat{\jmath}\hat{k} = -\hat{\imath} & \boxed{\hat{\imath}\hat{\jmath} = \hat{k}} \\
\boxed{\hat{\jmath}\hat{k} = \hat{\imath}} & \hat{\imath}\hat{\jmath}^2 = \hat{k}\hat{\jmath} \\
\hat{\jmath}^2\hat{k} = \hat{\jmath}\hat{\imath} & \boxed{-\hat{\imath} = \hat{k}\hat{\jmath}} \\
\boxed{-\hat{k} = \hat{\jmath}\hat{\imath}} & \\
-\hat{k}\hat{\imath} = \hat{\jmath}\hat{\imath}^2 & \\
\boxed{\hat{k}\hat{\imath} = \hat{\jmath}} & \\
\hat{\jmath}\hat{k}^2 = \hat{\imath}\hat{k} & \\
\boxed{-\hat{\jmath} = \hat{\imath}\hat{k}} &
\end{array}$$

Using those pairs, it's possible to multiply 2 quaternions together. When the real components are both 0 (absent), you end up with this:

$$(a\hat{\imath} + b\hat{\jmath} + c\hat{k})(d\hat{\imath} + e\hat{\jmath} + f\hat{k})$$
$$= a\hat{\imath}(d\hat{\imath} + e\hat{\jmath} + f\hat{k}) + b\hat{\jmath}(d\hat{\imath} + e\hat{\jmath} + f\hat{k}) + c\hat{k}(d\hat{\imath} + e\hat{\jmath} + f\hat{k})$$
$$= -ad + ae\hat{k} - af\hat{\jmath} - bd\hat{k} - be + bf\hat{\imath} + cd\hat{\jmath} - ce\hat{\imath} - cf$$
$$= -(ad + be + cf) + (bf - ce)\hat{\imath} + (cd - af)\hat{\jmath} + (ae - bd)\hat{k}$$

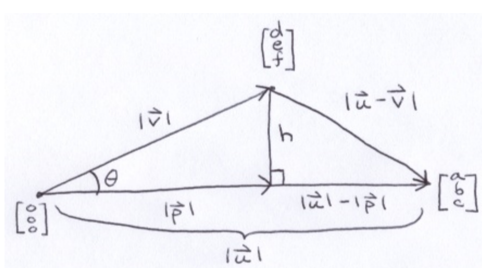
The real part on the left is the negative dot product and vector part on the right is the cross product. As mathematicians used quaternions in the decades after their discovery, they began to recognize that they were often using them to compute exactly those values. To make life easier, the vector product operators were introduced.

Dot product is an operation that takes place on the plane for which the vectors reside. The image below shows a view of that plane.



These vectors contain only direction information; you can think of their tails as touching the origin. Vector **u** defines a line and vector **v** casts a perpendicular shadow onto that line. The problem is find vector **p**.

You can redraw it as 2 right triangles that share a common side and height.



From there, it's just a matter of applying the Pythagorean Theorem twice.

$$\cos\theta = \frac{|\vec{p}|}{|\vec{v}|} \qquad |\vec{v}|\cos\theta = |\vec{p}|$$

$$|\vec{v}|^2 = d^2+e^2+f^2 \qquad \vec{u}-\vec{v} = \begin{bmatrix} a-d \\ b-e \\ c-f \end{bmatrix}$$

$$|\vec{u}|^2 = a^2+b^2+c^2 \qquad |\vec{u}-\vec{v}|^2 = (a-d)^2+(b-e)^2+(c-f)^2$$

$$= (a^2+b^2+c^2)+(d^2+e^2+f^2) - 2(ad+be+cf)$$

$$|\vec{p}|^2 + h^2 = |\vec{v}|^2$$

$$(|\vec{u}|-|\vec{p}|)^2 + h^2 = |\vec{u}-\vec{v}|^2 \qquad = |\vec{u}|^2+|\vec{v}|^2-2(ad+be+cf)$$

$$\cancel{|\vec{u}|^2} + |\vec{p}|^2 - 2|\vec{u}||\vec{p}| + h^2 = \cancel{|\vec{u}|^2} + |\vec{v}|^2 - 2(ad+be+cf)$$

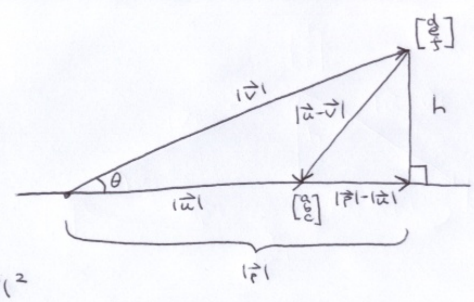
$$\cancel{|\vec{p}|^2} - \cancel{2}|\vec{u}||\vec{p}| + \cancel{|\vec{v}|^2} - \cancel{|\vec{p}|^2} = \cancel{|\vec{v}|^2} - \cancel{2}(ad+be+cf)$$

$$|\vec{u}||\vec{p}| = ad+be+cf \qquad \vec{p} = |\vec{p}|\frac{\vec{u}}{|\vec{u}|}$$

$$\boxed{|\vec{u}||\vec{v}|\cos\theta = ad+be+cf} \qquad \boxed{\vec{p} = \frac{\vec{u}(\vec{u}\cdot\vec{v})}{|\vec{u}|^2}}$$

In addition to finding **p**, I wrote down a useful equation that shows that dot product is proportional to the magnitude of the vectors involved and the cosine of the angle between them. Note if the angle is 90 degrees, then the dot product is 0. If the angle is less-than 90, then the dot product is positive. And, if the angle is greater-than 90, then the dot product is negative.

But, what if vector **u** was a lot shorter than vector **v**?



$$|\vec{p}|^2 + h^2 = |\vec{v}|^2$$

$$|\vec{v}|^2 = d^2+e^2+f^2 \qquad \vec{u}-\vec{v} = \begin{bmatrix} a-d \\ b-e \\ c-f \end{bmatrix}$$

$$(|\vec{p}|-|\vec{u}|)^2 + h^2 = |\vec{u}-\vec{v}|^2 \qquad |\vec{u}|^2 = a^2+b^2+c^2$$

$$|\vec{u}-\vec{v}|^2 = (a^2+b^2+c^2)+(d^2+e^2+f^2) - 2(ad+be+cf)$$

$$= |\vec{u}|^2+|\vec{v}|^2-2(ad+be+cf)$$

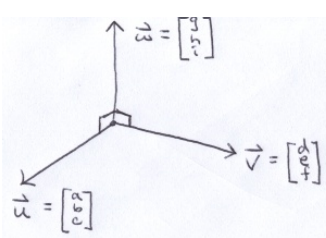
$$\cancel{|\vec{p}|^2} + \cancel{|\vec{u}|^2} - \cancel{2}|\vec{p}||\vec{u}| - \cancel{|\vec{u}|^2} - \cancel{|\vec{v}|^2} + \cancel{2}(ad+be+cf) = -\cancel{|\vec{v}|^2} + \cancel{|\vec{p}|^2}$$

$$|\vec{p}||\vec{u}| = ad+be+cf \qquad \cos\theta = \frac{|\vec{p}|}{|\vec{v}|}$$

$$\boxed{|\vec{u}||\vec{v}|\cos\theta = ad+be+cf = \vec{u}\cdot\vec{v}}$$

As you can see, you end up with exactly the same answer. If the angle is greater than 90 degrees, you can negate one of the vectors to end up with something similar to one of the pictures above. Again, you'll get the same answer.

Cross product is used to find a vector, **w**, perpendicular to the plane that **u** and **v** reside.



The angle between **u** and **v** is not necessarily 90, but the angles between **u** and **w** and **v** and **w** are both 90. From dot product, that gives us:

$$\vec{u}\cdot\vec{w} = 0$$
$$\vec{v}\cdot\vec{w} = 0$$

$$ag + bh + ci = 0$$
$$dg + eh + fi = 0$$

If **u** and **v** reside on a plane, as opposed to a line, then they cannot be parallel. But, if they were parallel, then they would only be distinguishable by some constant k.

$$\vec{u} = k\vec{v}$$

If they are not parallel, then at least one of the following relationships cannot exist.

$$a = kd \;,\; b = ke \;,\; c = kf$$

Suppose that the first or the second relationship is false. Then, we can rewrite the 2 original equations like this:

$$\begin{aligned} ag + bh &= -ci \\ dg + eh &= -fi \end{aligned}$$

Next, we solve for g and h using Cramer's Rule. Note that if the first and the second of the aforementioned relationships were both true, then we would be dividing by 0.

$$g = \frac{\begin{vmatrix} -ci & b \\ -fi & e \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}} \qquad h = \frac{\begin{vmatrix} a & -ci \\ d & -fi \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}}$$

We end up with this:

$$\boxed{g = \frac{bf - ce}{ae - bd}\,i} \quad \boxed{h = \frac{cd - af}{ae - bd}\,i}$$

Now, you have some freedom here. You can actually set i to any value of your choosing and compute g and h accordingly. The resulting vector, **w**, will be perpendicular to the **uv**-plane. But, you may happen to notice that the (non-zero) denominators of both fractions in the formulae are the same. If you set i to that value, the denominators vanish and the result is the definition of cross product.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \times \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} bf - ce \\ cd - af \\ ae - bd \end{bmatrix}$$

If the first and the second of the aforementioned relationships were true, then you could make a similar argument using the second and the third instead since at least one of them must be false assuming the vectors are not parallel. When they are parallel, the cross product is a null vector.

One important aspect of cross product (not proved here) is that it obeys the right-hand rule.

Finally, using all these vector ideas, I put together the following utility class.

```
public final class Vec {

  private Vec() {
  }

  // a += b
  public static void add(double[] a, double[] b) {
    for(int i = 0; i < 3; i++) {
      a[i] += b[i];
    }
  }

  // a = b + c
  public static void add(double[] a, double[] b, double[] c) {
    for(int i = 0; i < 3; i++) {
      a[i] = b[i] + c[i];
    }
  }

  // a -= b
  public static void subtract(double[] a, double[] b) {
    for(int i = 0; i < 3; i++) {
      a[i] -= b[i];
    }
  }

  // a = b - c
  public static void subtract(double[] a, double[] b, double[] c) {
    for(int i = 0; i < 3; i++) {
      a[i] = b[i] - c[i];
    }
  }

  // a *= s
  public static void scale(double[] a, double s) {
    for(int i = 0; i < 3; i++) {
      a[i] *= s;
    }
  }

  // a = b * s
  public static void scale(double[] a, double[] b, double s) {
    for(int i = 0; i < 3; i++) {
      a[i] = b[i] * s;
    }
  }

  // a /= s
  public static void divide(double[] a, double s) {
    double inverse = 1.0 / s;
    for(int i = 0; i < 3; i++) {
      a[i] *= inverse;
    }
  }

  // a = b / s
  public static void divide(double[] a, double[] b, double s) {
    double inverse = 1.0 / s;
```

```
    for(int i = 0; i < 3; i++) {
      a[i] = b[i] * inverse;
    }
  }

  // a = b
  public static void assign(double[] a, double[] b) {
    for(int i = 0; i < 3; i++) {
      a[i] = b[i];
    }
  }

  // a = (x, y, z)
  public static void assign(double[] a, double x, double y, double z) {
    a[0] = x;
    a[1] = y;
    a[2] = z;
  }

  // |v|
  public static double magnitude(double[] v) {
    return Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  }

  // |v|^2
  public static double magnitude2(double[] v) {
    return v[0] * v[0] + v[1] * v[1] + v[2] * v[2];
  }

  // |a - b|
  public static double distance(double[] a, double[] b) {
    double x = a[0] - b[0];
    double y = a[1] - b[1];
    double z = a[2] - b[2];
    return Math.sqrt(x * x + y * y + z * z);
  }

  // |a - b|^2
  public static double distance2(double[] a, double[] b) {
    double x = a[0] - b[0];
    double y = a[1] - b[1];
    double z = a[2] - b[2];
    return x * x + y * y + z * z;
  }

  // a = a / |a|
  public void normalize(double[] a) {
    double s = 1.0 / Math.sqrt(a[0] * a[0] + a[1] * a[1] + a[2] * a[2]);
    for(int i = 0; i < 3; i++) {
      a[i] *= s;
    }
  }

  // a = b / |b|
  public void normalize(double[] a, double[] b) {
    double s = 1.0 / Math.sqrt(b[0] * b[0] + b[1] * b[1] + b[2] * b[2]);
    for(int i = 0; i < 3; i++) {
      a[i] = b[i] * s;
    }
  }

  // a = -a
  public static void negate(double[] a) {
    for(int i = 0; i < 3; i++) {
      a[i] = -a[i];
    }
  }

  // a = -b
  public static void negate(double[] a, double[] b) {
    for(int i = 0; i < 3; i++) {
      a[i] = -b[i];
    }
  }

  // a += b * s
  public static void ray(double[] a, double[] b, double s) {
    for(int i = 0; i < 3; i++) {
      a[i] += b[i] * s;
    }
  }

  // a = b + c * s
  public static void ray(double[] a, double[] b, double[] c, double s) {
    for(int i = 0; i < 3; i++) {
      a[i] = b[i] + c[i] * s;
    }
  }

  // a . b
  public static double dot(double[] a, double[] b) {
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
  }

  // b = b x c
  public static void cross(double[] b, double[] c) {
    double x = b[1] * c[2] - b[2] * c[1];
    double y = b[2] * c[0] - b[0] * c[2];
    double z = b[0] * c[1] - b[1] * c[0];
    b[0] = x;
    b[1] = y;
    b[2] = z;
  }

  // a = b x c
  public static void cross(double[] a, double[] b, double[] c) {
    double x = b[1] * c[2] - b[2] * c[1];
    double y = b[2] * c[0] - b[0] * c[2];
    double z = b[0] * c[1] - b[1] * c[0];
    a[0] = x;
    a[1] = y;
    a[2] = z;
  }
}
```

It's unfortunate that Java does not support operator overloading. Instead, this is an all static class like the `Math` class.

Step 2

The original Amiga Juggler animation was a 1 second loop consisting of 24 frames rendered at 320×200 resolution. I am aiming for 30 frames rendered at the HD resolution of 1920×1080. However, during testing, I'll render 1 frame at a third of that resolution (640×360). To that end, I need some boilerplate code to save an image.

```
import java.awt.image.*;
import javax.imageio.*;
import java.io.*;
import java.util.*;

public class Main {

  public static final int SQRT_SAMPLES = 1;

  public static final int IMAGE_SCALE = 3;
  public static final int WIDTH = 1920 / IMAGE_SCALE;
  public static final int HEIGHT = 1080 / IMAGE_SCALE;

  public static final String IMAGE_TYPE = "jpg";
  public static final String OUTPUT_FILE = "output." + IMAGE_TYPE;

  public static final double GAMMA = 2.2;

  public static final int SAMPLES = SQRT_SAMPLES * SQRT_SAMPLES;
  public static final double INVERSE_SAMPLES = 1.0 / SAMPLES;
  public static final double INVERSE_GAMMA = 1.0 / GAMMA;

  public static final long SECOND_MILLIS = 1000L;
  public static final long MINUTE_MILLIS = 60 * SECOND_MILLIS;
  public static final long HOUR_MILLIS = 60 * MINUTE_MILLIS;

  private double[][][] pixels = new double[HEIGHT][WIDTH][3];

  public void launch() throws Throwable {
    saveImage();
  }

  private void saveImage() throws Throwable {
    int[] data = new int[WIDTH * HEIGHT];

    for(int y = 0, k = 0; y < HEIGHT; y++) {
      for(int x = 0; x < WIDTH; x++, k++) {
        int value = 0;
        for(int i = 0; i < 3; i++) {
          int intensity = (int)Math.round(255
              * Math.pow(pixels[y][x][i] * INVERSE_SAMPLES, INVERSE_GAMMA));
          if (intensity < 0) {
            intensity = 0;
          } else if (intensity > 255) {
            intensity = 255;
          }
          value <<= 8;
          value |= intensity;
        }
        data[k] = value;
      }
    }

    BufferedImage image = new BufferedImage(
        WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
    image.setRGB(0, 0, WIDTH, HEIGHT, data, 0, WIDTH);
    ImageIO.write(image, IMAGE_TYPE, new File(OUTPUT_FILE));
  }

  public static void main(String... args) throws Throwable {

    long startTime = System.currentTimeMillis();

    Main main = new Main();
    main.launch();

    long interval = System.currentTimeMillis() - startTime;
    long hours = interval / HOUR_MILLIS;
    interval %= HOUR_MILLIS;
    long minutes = interval / MINUTE_MILLIS;
    interval %= MINUTE_MILLIS;
    long seconds = interval / SECOND_MILLIS;
    interval %= SECOND_MILLIS;
    System.out.format("%d hour%s, %d minute%s, %d second%s, %d millisecond%s%n",
        hours, hours == 1 ? "" : "s",
        minutes, minutes == 1 ? "" : "s",
        seconds, seconds == 1 ? "" : "s",
        interval, interval == 1 ? "" : "s");
  }

}
```

`0 hours, 0 minutes, 0 seconds, 191 milliseconds`

source

As you can see, it stores an all black image.

The `main` method contains code that outputs the rendering time (ray tracing is a slow process).

The `pixels` array will store the RGB values of each pixel in the image as it is rendered. Each color intensity component is in the range of 0.0 (darkest) to 1.0 (lightest). However, to compute the color value of a pixel, the code will statistically sample that part of the 3D scene a specified number of times (the constant `SAMPLES`) and it will average the samples together. The `pixels` array stores the sum of all the samples for each pixel. The `saveImage` method divides by the number of samples to convert the sum into an average.

For saving, the `pixels` array is temporarily converted into a `BufferedImage`. The `BufferedImage.setRGB` method expects an integer array where color intensities are in the range of 0 to 255. Unfortunately, we cannot simply multiply the `pixels` array values by 255 because the relationship between them is not linear. The actual brightness of a monitor pixel increases exponentially with the size of that integer value by a power of a constant called gamma (typically 2.2 on PCs and 1.8 on Macs). To compensate, the `pixels` array values are raised to the inverse of gamma before multiplying.

$$255\, c^{1/\gamma}$$

Step 3

To push this machine to the max, I added code to render in parallel executing threads.

```
  private int runningCount;
  private int rowIndex;

  public void launch() throws Throwable {

    int processors = Runtime.getRuntime().availableProcessors();
    updateRunningCount(processors);
    for(int i = 0; i < processors; i++) {
      new Thread(Integer.toString(i)) {
        @Override
        public void run() {
          render();
          updateRunningCount(-1);
        }
      }.start();
    }

    synchronized(this) {
      while(runningCount != 0) {
        wait();
      }
    }

    saveImage();
  }

  private void render() {
    while(true) {

      int y = getNextRowIndex();
      if (y >= HEIGHT) {
        return;
      }

      for(int x = 0; x < WIDTH; x++) {

      }
    }
  }

  private synchronized void updateRunningCount(int dx) {
    runningCount += dx;
    if (runningCount == 0) {
      notifyAll();
    }
  }

  private synchronized int getNextRowIndex() {
    return rowIndex++;
  }
```

This is a Dell Studio XPS 435T with an Intel Core i7 CPU 920 at 2.67 GHz with 64-bit Vista. `Runtime.getRuntime().availableProcessors()` returns 8 and I confirmed that it takes 8 threads before the Task Manager reports 100% CPU usage.

Each thread will work on separate rows of the image. The `getNextRowIndex` method returns the next row of the image yet to be rendered.

Step 4

I decided it would be nice to see the output as it is slowly generated. To make that happen, I created a panel capable of displaying images.

```
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;

public class ImagePanel extends JPanel {

  private BufferedImage image;

  public ImagePanel(BufferedImage image) {
    this.image = image;
    setPreferredSize(new Dimension(image.getWidth(), image.getHeight()));
  }

  @Override
  protected void paintComponent(Graphics g) {
    g.drawImage(image, 0, 0, this);
  }
}
```

Next, I created a frame to display the panel. The constructor captures a `BufferedImage` and calling the `imageUpdated` method forces it to repaint.

```
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;

public class RenderFrame extends JFrame {

  private ImagePanel imagePanel;

  public RenderFrame(BufferedImage image) {
    setTitle("Amiga Juggler");
    setContentPane(imagePanel = new ImagePanel(image));
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setResizable(false);
    pack();
    setLocationRelativeTo(null);
    setVisible(true);
  }

  public void imageUpdated() {
    EventQueue.invokeLater(new Runnable() {
```

```
      @Override
      public void run() {
        imagePanel.repaint();
      }
    });
  }

  @Override
  public void setTitle(final String title) {
    EventQueue.invokeLater(new Runnable() {
      @Override
      public void run() {
        RenderFrame.super.setTitle(title);
      }
    });
  }
}
```

To support the frame, I had to move some stuff around. There is no longer a global `pixels` array. Instead, the `render` method allocates an `int` array representing color intensities of a single row with RGB values compatible with `BufferedImage`. The `render` method does the conversion itself from `double` intensities to `int` intensities by dividing by the number of samples to compute the average, applying gamma correction and then multiplying by 255. When it finishes rendering a row, it invokes `rowCompleted`. That method updates the `BufferedImage` and it forces the `RenderFrame` to repaint.

```
import java.awt.image.*;
import javax.imageio.*;
import java.io.*;
import java.util.*;

public class Main {

  public static final int SQRT_SAMPLES = 1;

  public static final int IMAGE_SCALE = 3;
  public static final int WIDTH = 1920 / IMAGE_SCALE;
  public static final int HEIGHT = 1080 / IMAGE_SCALE;

  public static final String IMAGE_TYPE = "jpg";
  public static final String OUTPUT_FILE = "output." + IMAGE_TYPE;
  public static final boolean RENDER_IN_WINDOW = true;

  public static final double GAMMA = 2.2;

  public static final int SAMPLES = SQRT_SAMPLES * SQRT_SAMPLES;
  public static final double INVERSE_SAMPLES = 1.0 / SAMPLES;
  public static final double INVERSE_GAMMA = 1.0 / GAMMA;

  public static final long SECOND_MILLIS = 1000L;
  public static final long MINUTE_MILLIS = 60 * SECOND_MILLIS;
  public static final long HOUR_MILLIS = 60 * MINUTE_MILLIS;

  private RenderFrame renderFrame;
  private BufferedImage image;
  private int runningCount;
  private int rowIndex;

  public void launch() throws Throwable {

    image = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
    if (RENDER_IN_WINDOW) {
      renderFrame = new RenderFrame(image);
    }

    int processors = Runtime.getRuntime().availableProcessors();
    updateRunningCount(processors);
    for(int i = 0; i < processors; i++) {
      new Thread(Integer.toString(i)) {
        @Override
        public void run() {
          render();
          updateRunningCount(-1);
        }
      }.start();
    }

    synchronized(this) {
      while(runningCount != 0) {
        wait();
      }
    }

    saveImage();

    if (RENDER_IN_WINDOW) {
      renderFrame.setTitle("Amiga Juggler [DONE]");
    }
  }

  private void render() {

    int[] pixels = new int[WIDTH];
    double[] pixel = new double[3];

    while(true) {

      int y = getNextRowIndex();
      if (y >= HEIGHT) {
        return;
      }

      for(int x = 0; x < WIDTH; x++) {

        // -- SIMULATE SLOW DRAWING ---------------------------
        pixel[0] = 1;
        try {
          Thread.sleep(1);
        } catch(InterruptedException e) {
        }
        // ---------------------------------------------------

        int value = 0;
        for(int i = 0; i < 3; i++) {
          int intensity = (int)Math.round(255
              * Math.pow(pixel[i] * INVERSE_SAMPLES, INVERSE_GAMMA));
          if (intensity < 0) {
            intensity = 0;
          } else if (intensity > 255) {
            intensity = 255;
          }
          value <<= 8;
```

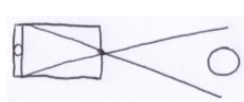
```
          value |= intensity;
        }

        pixels[x] = value;
      }

      rowCompleted(y, pixels);
    }
  }

  private synchronized void updateRunningCount(int dx) {
    runningCount += dx;
    if (runningCount == 0) {
      notifyAll();
    }
  }

  private synchronized int getNextRowIndex() {
    return rowIndex++;
  }

  private synchronized void rowCompleted(int rowIndex, int[] pixels) {
    image.setRGB(0, rowIndex, WIDTH, 1, pixels, 0, WIDTH);
    if (RENDER_IN_WINDOW) {
      renderFrame.imageUpdated();
    }
  }

  private void saveImage() throws Throwable {
    ImageIO.write(image, IMAGE_TYPE, new File(OUTPUT_FILE));
  }

  public static void main(String... args) throws Throwable {

    long startTime = System.currentTimeMillis();

    Main main = new Main();
    main.launch();

    long interval = System.currentTimeMillis() - startTime;
    long hours = interval / HOUR_MILLIS;
    interval %= HOUR_MILLIS;
    long minutes = interval / MINUTE_MILLIS;
    interval %= MINUTE_MILLIS;
    long seconds = interval / SECOND_MILLIS;
    interval %= SECOND_MILLIS;
    System.out.format("%d hour%s, %d minute%s, %d second%s, %d millisecond%s%n",
        hours, hours == 1 ? "" : "s",
        minutes, minutes == 1 ? "" : "s",
        seconds, seconds == 1 ? "" : "s",
        interval, interval == 1 ? "" : "s");
  }

}
```

To test it out, the inner loop in the `render` method contains code to slowly set pixels red.

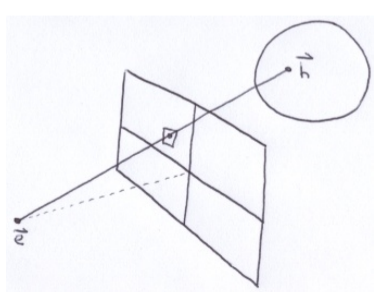
source

Step 5

The virtual camera is based on the camera obscura and the pinhole camera. Surprisingly, a real camera does not actually need a lens to function. A pinhole-sized aperture ensures that each spot on the film is only exposed to light arriving from one particular direction. Unfortunately, in the real world, the smaller the aperture, the longer the exposure time.

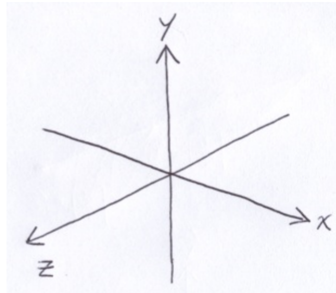


Instead of putting virtual film behind the pinhole, a virtual screen is placed in front of it. Geometrically, it works the same way. Rather than a pinhole, consider it as the pupil of a virtual eye staring at the center of the virtual screen.

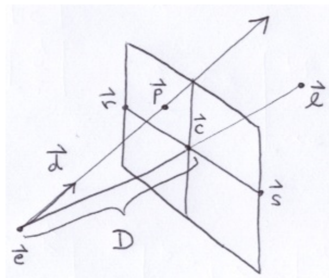


The ray tracer will follow the pathway of light backwards from the eye, through the virtual screen and out into the scene. In this way, the virtual screen acts like a window. Each pixel in the image has a corresponding point on the virtual screen in 3D space. Before it can cast a ray out into the scene, it has to figure out the direction of the ray.

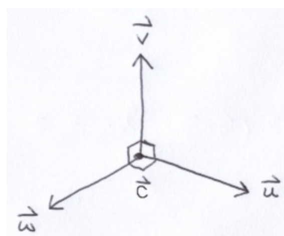
I should mention that the coordinate system used looks like this:



The position and the orientation of the virtual screen is determined by 3 things: the location of the eye, **e**; a point that the eye is looking at, **l**; and, the distance, D, between the eye and the center of the virtual screen, **c**.



We need to construct an orthonormal basis (ONB) situated at the center of the screen, **c**.



$$|\vec{u}| = |\vec{v}| = |\vec{w}| = 1$$

The unit vectors, **u**, **v** and **w**, are perpendicular to each other. Collectively, the point **c** and the ONB define a right-handed coordinate system. The point **c** acts like the origin. The vector **u** serves as the x-axis of the virtual screen. The vector **v** acts as the y-axis of the virtual screen. And, the vector **w** points toward the eye.

If you had some point **p** = (a, b, e) and you wanted to know where it was located in that coordinate system, you could position it using a variation of the ray equation.

$$\vec{p} = \begin{bmatrix} a \\ b \\ e \end{bmatrix}$$

$$\boxed{\vec{c} + a\vec{u} + b\vec{v} + e\vec{w}}$$

In our case, **p** represents a point on the virtual screen; so, e = 0.

For the general case, I added 2 new methods to the `Vec` utility class.

```
// p = o + p[0] u + p[1] v + p[2] w
public static void transform(
    double[] p, double[] o, double[] u, double[] v, double[] w) {
  double x = o[0] + p[0] * u[0] + p[1] * v[0] + p[2] * w[0];
  double y = o[1] + p[0] * u[1] + p[1] * v[1] + p[2] * w[1];
  double z = o[2] + p[0] * u[2] + p[1] * v[2] + p[2] * w[2];
  p[0] = x;
  p[1] = y;
  p[2] = z;
}

// q = o + p[0] u + p[1] v + p[2] w
public static void transform(
    double[] q, double[] p, double[] o, double[] u, double[] v, double[] w) {
  double x = o[0] + p[0] * u[0] + p[1] * v[0] + p[2] * w[0];
  double y = o[1] + p[0] * u[1] + p[1] * v[1] + p[2] * w[1];
  double z = o[2] + p[0] * u[2] + p[1] * v[2] + p[2] * w[2];
  q[0] = x;
  q[1] = y;
  q[2] = z;
}
```

The unit vector **w** is the easiest of the 3 to obtain:

$$\boxed{\vec{w} = \frac{\vec{e} - \vec{l}}{|\vec{e} - \vec{l}|}}$$

The formula inspired this addition to the `Vec` utility:

```
// v = ( head - tail ) / | head - tail |
public static void constructUnitVector(
    double[] v, double[] head, double[] tail) {
  for(int i = 0; i < 3; i++) {
    v[i] = head[i] - tail[i];
  }
  double s = 1.0 / Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  for(int i = 0; i < 3; i++) {
    v[i] = v[i] * s;
  }
}
```

There is already a `subtract` method to construct a non-unit vector from 2 point vectors.

Now that we have **w**, we can find **c** by starting at **e** and traveling a distance of −D along **w** using the ray equation (**w** points toward the **e**).

$$\boxed{\vec{c} = \vec{e} - D\vec{w}}$$

On the picture of the virtual screen above, observe points **r** and **s**. Imagine that the virtual screen were a camera that you were holding with your hands over **r** and **s**. You can rotate around (shoot the photo in any direction) and you can tilt the camera upwards and downwards. But, you would typically keep the line between **r** and **s** level. In this way, when you take the picture, the horizon will be parallel to the bottom of the photograph. Rarely do you roll the camera because it produces strange photographs.

We'll construct the remainder of the ONB with the assumption that there is no roll. If you do need to roll the virtual camera, you can rotate **u** and **v** around **w** after the ONB is fully constructed. No camera roll is used in this project.

If there is no roll, then **u** is perpendicular to the normal of the ground plane that the juggler stands on, **n** = (0, 1, 0). And, by the definition of the ONB, **u** is also perpendicular to **w**. Hence, we can find **u** using:

$$\boxed{\vec{u} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \times \vec{w}}$$

Again from the definition of the ONB, we can now solve for **v**.

$$\boxed{\vec{v} = \vec{w} \times \vec{u}}$$

You can verify that applying cross products in this sequence does produce a right-handed coordinate system.

There is only 1 catch with the above construction: If **w** is parallel to the ground normal, **n**, then taking the cross product produces a null vector. Meaning, if you are looking straight up or straight down, then it will fail. But, it's clear what **u** and **v** should be in those cases.

I added these methods to `Vec`:

```
public static final double TINY = 1e-9;

// a == b
public static boolean equals(double[] a, double[] b) {
  for(int i = 0; i < 3; i++) {
    if (Math.abs(a[i] - b[i]) > TINY) {
      return false;
    }
  }
  return true;
}

// v == (x, y, z)
public static boolean equals(double[] v, double x, double y, double z) {
  return Math.abs(v[0] - x) <= TINY
      && Math.abs(v[1] - y) <= TINY
      && Math.abs(v[2] - z) <= TINY;
}

// w -> u, v
public static void onb(double[] u, double[] v, double[] w) {
  if (equals(w, 0, 1, 0)) {
    assign(u, 1, 0, 0);
    assign(v, 0, 0, -1);
  } else if (equals(w, 0, -1, 0)) {
    assign(u, 1, 0, 0);
    assign(v, 0, 0, 1);
  } else {
    assign(u, w[2], 0, -w[0]);
    cross(v, w, u);
  }
}
```

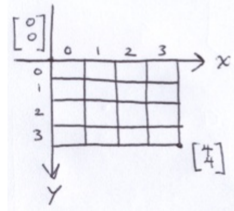
The input of the `onb` method is **w** and the output is **u** and **v**.

Putting it all together, I added this to the top of the `render` method:

```
  double[] u = new double[3];
  double[] v = new double[3];
  double[] w = new double[3];
  double[] c = new double[3];

  Vec.constructUnitVector(w, EYE, LOOK);
  Vec.ray(c, EYE, w, -DISTANCE_TO_VIRTUAL_SCREEN);
  Vec.onb(u, v, w);
```

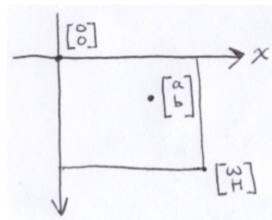
Step 6

We normally think of addressing pixels in an image or on the screen in terms of row and column index. The upper-left corner is origin and the y-axis points downwards. Consider this 4×4 image:

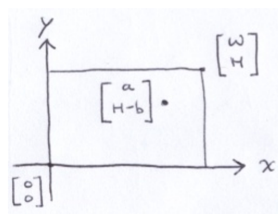


The width and height are 4, but we use the indices 0 to 3 (inclusive). Instead of row and column indices, consider the upper-left corner of each pixel as the origin of that pixel. For instance, the upper-left corner of the lower-rightmost pixel is (3, 3). The lower-right corner of that pixel is (4, 4). In this way, it is possible to refer to points anywhere on the image, even within pixels.

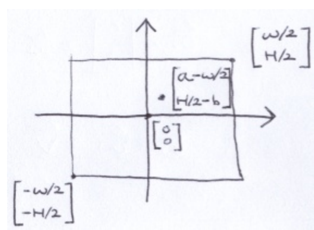
Now, consider some arbitrary point on the image that we want to find the equivalent point in 3D space on the virtual screen. The image has a width and height of W and H respectively.



First, without modifying the image, I'll move the origin to the lower-left and I'll reverse the direction of the y-axis. To address the same point on the image, the coordinates need to change to:



Again, without modifying the image, I'll move the origin to the center of the image and adjust the points coordinates to compensate.



Now, let's assume that the virtual screen has the same aspect ratio as the image. We'll specify the width of the virtual screen as a constant in the program. For this discussion, I'll refer to the virtual screen width as A and the virtual screen height as B.

$$\frac{W}{H} = \frac{A}{B}$$

That gives us a way to find the virtual screen height:

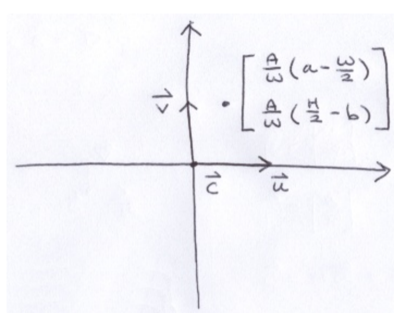
$$\boxed{B = \frac{AH}{W}}$$

If we had some arbitrary point (x, y) in the image aspect ratio and we needed to convert it to the virtual screen aspect ratio, we could accomplish the conversion with this:

$$x' = \frac{A}{W} x$$

$$y' = \frac{B}{H} y = \frac{A}{W} y$$

Combining the last 2 concepts leaves us with this:



Finally, we have a way to transform some arbitrary 2D point (a, b) on the image to a 3D point on the virtual screen.

$$\vec{p} = \vec{c} + \left[\frac{A}{W}\left(a - \frac{W}{2}\right)\right]\vec{u} + \left[\frac{A}{W}\left(\frac{H}{2} - b\right)\right]\vec{v}$$

That formula inspired these additions to `Vec`:

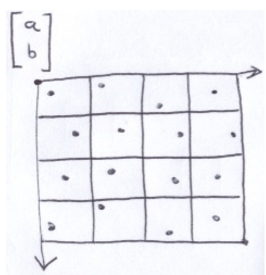
```
// p = o + p[0] u + p[1] v
public static void map(
    double[] p, double[] o, double[] u, double[] v) {
  double x = o[0] + p[0] * u[0] + p[1] * v[0];
  double y = o[1] + p[0] * u[1] + p[1] * v[1];
  double z = o[2] + p[0] * u[2] + p[1] * v[2];
  p[0] = x;
  p[1] = y;
  p[2] = z;
}

// q = o + p[0] u + p[1] v
public static void map(
    double[] q, double[] p, double[] o, double[] u, double[] v) {
  double x = o[0] + p[0] * u[0] + p[1] * v[0];
  double y = o[1] + p[0] * u[1] + p[1] * v[1];
  double z = o[2] + p[0] * u[2] + p[1] * v[2];
  q[0] = x;
  q[1] = y;
  q[2] = z;
}

// q = o + x u + y v
public static void map(
    double[] q, double[] o, double[] u, double[] v, double x, double y) {
  double i = o[0] + x * u[0] + y * v[0];
  double j = o[1] + x * u[1] + y * v[1];
  double k = o[2] + x * u[2] + y * v[2];
  q[0] = i;
  q[1] = j;
  q[2] = k;
}
```

Since the point, **p**, on the virtual screen is known, we can finally compute the direction of the ray, **d**:

$$\vec{d} = \frac{\vec{p} - \vec{e}}{|\vec{p} - \vec{e}|}$$

Step 7

As mentioned above, the ray tracer will fire several rays through each pixel at slightly different coordinates to sample the 3D scene and then it will assign that pixel to the average of the results. To accomplish this, an image pixel is uniformly divided into a grid. Within each grid cell, a random point is chosen. This is known as jittered sampling. There are superior sampling techniques, but this technique works well enough and it's easy to code. Below is an example of a sampled image pixel.



Generating pseudorandom values may be a computationally expensive operation. To guard against that possibility, I created a utility class that stores a large cache of random values.

```
public class RandomDoubles {

  private static volatile double[] values;

  private int index;

  public RandomDoubles() {
    synchronized(RandomDoubles.class) {
      if (values == null) {
        values = new double[70001];
        for(int i = values.length - 1; i >= 0; i--) {
          values[i] = Math.random();
        }
      }
      index = (int)(Math.random() * values.length);
    }
  }

  public double random() {
    if (index >= values.length) {
      index = 0;
    }
    return values[index++];
  }
}
```

The cache is initialized when the first instance of `RandomDoubles` is created. 70001 is a prime number. I chose a prime to reduce the chances of strange patterns from showing up in the output image as a consequence of cyclically using the same set of random values.

Here is the `render` method after introducing random sampling:

```
  private void render() {

    double[] u = new double[3];
    double[] v = new double[3];
    double[] w = new double[3];
    double[] c = new double[3];
    double[] p = new double[3];
    double[] d = new double[3];
```

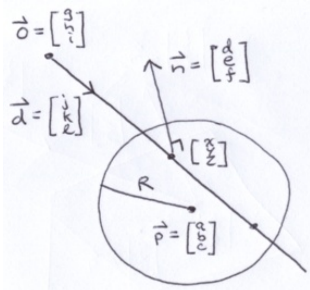
```
    Vec.constructUnitVector(w, EYE, LOOK);
    Vec.ray(c, EYE, w, -DISTANCE_TO_VIRTUAL_SCREEN);
    Vec.onb(u, v, w);

    RandomDoubles randomDoubles = new RandomDoubles();
    int[] pixels = new int[WIDTH];
    double[] pixel = new double[3];

    while(true) {

      int y = getNextRowIndex();
      if (y >= HEIGHT) {
        return;
      }

      for(int x = 0; x < WIDTH; x++) {

        for(int i = 0; i < SQRT_SAMPLES; i++) {
          double b = y + INVERSE_SQRT_SAMPLES * i;
          if (SQRT_SAMPLES == 1) {
            b += 0.5;
          } else {
            b += randomDoubles.random();
          }

          b = VIRTUAL_SCREEN_RATIO * (HALF_HEIGHT - b);

          for(int j = 0; j < SQRT_SAMPLES; j++) {
            double a = x + INVERSE_SQRT_SAMPLES * j;
            if (SQRT_SAMPLES == 1) {
              a += 0.5;
            } else {
              a += randomDoubles.random();
            }

            a = VIRTUAL_SCREEN_RATIO * (a - HALF_WIDTH);

            Vec.map(p, EYE, u, v, a, b);
            Vec.constructUnitVector(d, p, EYE);

            // ... USE d ...
          }
        }

        int value = 0;
        for(int i = 0; i < 3; i++) {
          int intensity = (int)Math.round(255
              * Math.pow(pixel[i] * INVERSE_SAMPLES, INVERSE_GAMMA));
          if (intensity < 0) {
            intensity = 0;
          } else if (intensity > 255) {
            intensity = 255;
          }
          value <<= 8;
          value |= intensity;
        }

        pixels[x] = value;
      }

      rowCompleted(y, pixels);
    }
  }
```

The square-root of the number of samples, `SQRT_SAMPLES`, can be increased to improve image quality. Notice that when it is 1, instead of using a random sample, the sample is taken in the center of the pixel.

Step 8

The juggling robot is constructed entirely out of spheres. The sky also appears to be a hemisphere. So, it is pertinent to work out how to intersect a ray and a sphere.

Below, a sphere of radius R is centered about **p**. A ray originating from **o**, fired in direction **d**, strikes the sphere at some point (x, y, z).



$$\left\|\begin{bmatrix} x-a \\ y-b \\ z-c \end{bmatrix}\right\|^2 = R^2$$

$$(x-a)^2+(y-b)^2+(z-c)^2 = R^2$$

$$\vec{o} + t\vec{d} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} g \\ h \\ i \end{bmatrix} + t\begin{bmatrix} i \\ k \\ l \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$x = g + ti \qquad (ti+g-a)^2+(tk+h-b)^2+(tl+i-c)^2 = R^2$$

$$y = h + tk \qquad m = g-a \qquad n = h-b \qquad o = i-c$$

$$z = i + tl$$

$$(it+m)^2+(kt+n)^2+(lt+o)^2 = R^2$$

$$(i^2+k^2+l^2)t^2 + 2(im+kn+lo)t + (m^2+n^2+o^2) = R^2$$

$$|\vec{d}|^2t^2 + 2\left(\vec{d}\cdot\begin{bmatrix} g-a \\ h-b \\ i-c \end{bmatrix}\right)t + \left\|\begin{bmatrix} g-a \\ h-b \\ i-c \end{bmatrix}\right\|^2 = R^2$$

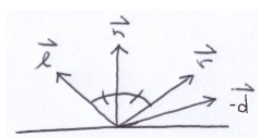
$$\boxed{|\vec{d}|^2t^2 + 2(\vec{d}\cdot(\vec{o}-\vec{p}))t + |\vec{o}-\vec{p}|^2 = R^2}$$

$$At^2 + Bt + C = 0 \qquad t = \frac{-B \pm \sqrt{B^2-4AC}}{2A}$$

Now we have the amount of time it takes for the ray to reach a sphere from the virtual eye. We will apply this formula to every sphere in the scene. If the value within the square-root is negative, then the ray does not intersect the sphere. Of the ones that do intersect, the one with the minimal time is the one that we see. Well almost. If a sphere is behind the eye, this formula will output a negative time value. Intersection only took place if time is positive.

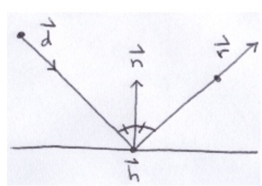
We actually need to constrain time a bit more. The eye is not the only point that we will be casting rays from. For example, when a ray bounces off one of the mirrored spheres, we will cast a new ray off its surface. To prevent the possibility of intersection between a ray and a surface that emitted the ray, we need to make sure that time is greater than a tiny positive value that we'll call epsilon. Also, we need to make sure that time is smaller than some max value. For example, to determine if a surface is illuminated, we'll cast a ray back to a light source. In that case, time cannot be greater than the time it takes to reach the light source. Any intersections that occur beyond the source of light cannot be considered.

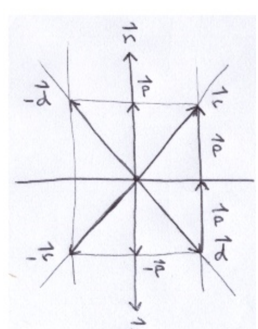
Once we locate the intersection with minimal time, we'll compute the color of the hit point using Phong Shading. Phong Shading takes into account the following vectors:



All of the vectors are unit vectors. The vector **d** is the direction of the incident ray. The negation of **d** points back to the ray's origin, back to the virtual eye. The vector **n** is the surface normal. In case of a sphere, it is the vector between the hit point and the center of the sphere, normalized. However, it is possible that we may need to negate the normal to face the viewer. The angle between **n** and −**d** must be less-than-or-equal-to 90 degrees. Meaning, if **n** dot **d** is positive, then we reverse the normal. The sky is a good example of this. It is the interior surface of a hemisphere and the normal must point inwards. The vector **l** is a unit vector pointing toward a light source. And, finally, the vector **r** is the mirror reflection of the vector **l**. It represents light originating from the light source and bouncing off of the surface.

Since Phong Shading uses a mirror reflection vector, let's consider bouncing a ray off of one of the mirror spheres. Below, a ray traveling in direction **d** strikes a surface at point **h** with normal **n** and then it reflects in direction **r**.





Vector **p** is the projection of −**d** onto **n** (vector projection was illustrated in step 1).

$$\vec{p} = \frac{\vec{n}(\vec{n} \cdot -\vec{d})}{|\vec{n}|^2} = \vec{n}(\vec{n} \cdot -\vec{d})$$

$$\vec{r} = 2\vec{p} + \vec{d} \qquad \boxed{\vec{r} = \vec{d} - 2(\vec{n} \cdot \vec{d})\vec{n}}$$

But, for Phong Shading, vector **l** actually points toward the light source; it's in the opposite direction that the light ray travels. If we negate the ray direction, we end up with:

$$\boxed{\vec{r} = -\vec{l} + 2(\vec{n} \cdot \vec{l})\vec{n}}$$

Note, if **l** dot **n** < 0 (greater than 90 degrees), then the light source is on the opposite side of the surface and none of its light reaches the side that we are viewing. Also, we need to cast a ray back to the light source to determine if there are any objects in the way. If the path to the light source is interrupted, then the surface is in shadow and only ambient light comes into play.

Finally, to shade the surface, we apply the Phong reflection model.

$$\begin{aligned} L_o = {} & k_a c_d l_s c_a \\ & + \left(\frac{k_d c_d}{\pi}\right)(l_s c_l)(\vec{n} \cdot \vec{l}) \\ & + k_s (\vec{r} \cdot -\vec{d})^e (l_s c_l)(\vec{n} \cdot \vec{l}) c_s \\ & + k_r c_r L_1 \end{aligned}$$

L represents radiance. We need to apply this formula 3 times, once for each RGB color intensity component. There is only one light source in this project. If there were multiple light sources, we would have to apply it to each of them and add them together.

There are 2 dot products in the formula. If one of them returns a negative value, change it to 0. Meaning, if the light source and the normal are on opposite sides, then the light does not contribute to the color of the surface. And, if the reflected light ray cannot be viewed seen, then it does not contribute to the color either.

The 4 terms represent ambient, diffuse, specular and reflected light. Each term is weighted by a k constant that is typically between 0 and 1. Collectively, the constants enable you to change the behavior of the surface.

- k_a: The ambient reflection constant considers light arriving at the surface from all directions as opposed to some specified light source.
- c_d: The diffuse color is the color of the surface if the light source were directly over it (**n** dot **l** = 1).
- l_s: This is a radiance scaling factor. This allows you to scale the resulting sum.
- c_a: This is the color of the ambient surrounding light. It's typically just white.
- k_d: The diffuse reflection constant considers the diffuse color of the surface and the angle between the surface normal and a light source.
- c_l: This is the color of the light source.
- k_s: The specular reflection constant determines the strength of the highlights.
- e: When the shininess constant is large, the specular highlight is small.
- c_s: This is the color of the specular highlight. If you set this to c_d, it produces a metallic-like surface.
- k_r: The reflection constant adjusts the strength of the reflected ray.
- c_r: The reflection color filters the reflected ray.
- L_1: This is the radiance of the reflected ray computed by recursively applying the overall formula once again.

I created a class to describe the material of a surface.

```
public class Material {
  public double ambientWeight;
  public double diffuseWeight;
  public double specularWeight;
  public double reflectionWeight;
  public double shininess;
  public double[] diffuseColor;
  public double[] highlightColor;
  public double[] reflectionColor;

  public Material(
      double ambientWeight,
      double diffuseWeight,
      double specularWeight,
      double reflectionWeight,
      double shininess,
      double[] diffuseColor,
      double[] highlightColor,
      double[] reflectionColor) {
    this.ambientWeight = ambientWeight;
    this.diffuseWeight = diffuseWeight;
    this.specularWeight = specularWeight;
    this.reflectionWeight = reflectionWeight;
    this.shininess = shininess;
    this.diffuseColor = diffuseColor;
    this.highlightColor = highlightColor;
    this.reflectionColor = reflectionColor;
  }
}
```

Next, I created a class to house different types of materials. Right now, there is only red plastic. It's a bright red material that has a white specular highlight.

```
public final class Materials {

  private Materials() {
  }

  public static final Material RED_PLASTIC = new Material(
    0.1,
    2.0,
    1.0,
    0.0,
    10.0,
    new double[] { 1, 0, 0 },
    new double[] { 1, 1, 1 },
    new double[] { 0, 0, 0 }
```

```
  );
}
```

For ray-object intersection, I created an object called `Intersection` that represents the result. It contains the hit time, the hit point, the surface normal and the surface material at the hit point.

```
public class Intersection {
  public double time;
  public double[] hit = new double[3];
  public double[] normal = new double[3];
  public Material material;
}
```

3D objects in the scene are represented by the following interface:

```
public interface IObject {
  public boolean intersect(
      double[] o, double[] d, boolean primaryRay, double maxTime,
      double[][] temps,
      Intersection intersection);
}
```

The `intersect` method returns `true` if the ray intersects the 3D object. If the `primaryRay` parameter is set to `true`, then additional information about the intersection will be stored in the `intersection` parameter. The parameters `o` and `d` are the origin and direction of the ray respectively. The `maxTime` parameter is the intersection time upper-bound. The epsilon lower-bound constant is defined in `Main`. Finally, the `temps` parameter are temporary vectors that can be used during the intersection calculation. The render method passes in 16 of them.

Here is an implementation using the ray-sphere intersection formula from above. Notice that if `primaryRay` is not set to true, it does not bother to compute the hit point and the surface normal. When casting shadow rays and when casting ambient occlusion rays, we only care about the return value.

```
public class Sphere implements IObject {

  public double[] center = new double[3];
  public double radius = 1;
  public Material material;

  public Sphere(
      double x, double y, double z, double radius, Material material) {
    Vec.assign(center, x, y, z);
    this.radius = radius;
    this.material = material;
  }

  public boolean intersect(
      double[] o, double[] d, boolean primaryRay, double maxTime,
      double[][] temps,
      Intersection intersection) {

    Vec.subtract(temps[0], o, center);
    double B = 2.0 * Vec.dot(d, temps[0]);
    double C = Vec.magnitude2(temps[0]) - radius * radius;
    double square = B * B - 4 * C;
    if (square >= 0) {
      double sqrt = Math.sqrt(square);
      double t1 = 0.5 * (-B - sqrt);
      double t2 = 0.5 * (-B + sqrt);
      boolean intersected = false;
      if (t1 >= Main.EPSILON && t1 <= maxTime) {
        intersected = true;
        intersection.time = t1;
      } else if (t2 >= Main.EPSILON && t2 < maxTime) {
        intersected = true;
        intersection.time = t2;
      }
      if (primaryRay && intersected) {
        Vec.ray(intersection.hit, o, d, intersection.time);
        Vec.constructUnitVector(intersection.normal, intersection.hit, center);
        intersection.material = material;
      }
      if (intersected) {
        return true;
      }
    }

    return false;
  }
}
```

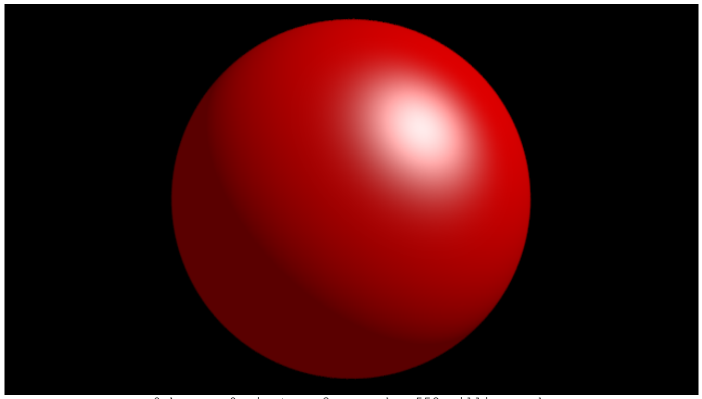
Finally, I updated the render method using the Phong Shading model and I tested it with a scene containing a single sphere.

```
  private volatile IObject[] scene
      = { new Sphere(0, 0, 0, 50, Materials.RED_PLASTIC) };

  private void render() {

    double[] u = new double[3];
    double[] v = new double[3];
    double[] w = new double[3];
    double[] c = new double[3];
    double[] p = new double[3];
    double[] d = new double[3];
    double[] l = new double[3];
    double[] r = new double[3];
    double[][] temps = new double[16][3];
    Intersection intersection = new Intersection();
    Intersection bestIntersection = new Intersection();

    Vec.constructUnitVector(w, EYE, LOOK);
    Vec.ray(c, EYE, w, -DISTANCE_TO_VIRTUAL_SCREEN);
    Vec.onb(u, v, w);

    RandomDoubles randomDoubles = new RandomDoubles();
    int[] pixels = new int[WIDTH];
    double[] pixel = new double[3];

    while(true) {

      int y = getNextRowIndex();
      if (y >= HEIGHT) {
        return;
      }

      for(int x = 0; x < WIDTH; x++) {

        Vec.assign(pixel, 0, 0, 0);

        for(int i = 0; i < SQRT_SAMPLES; i++) {
          double b = y + INVERSE_SQRT_SAMPLES * i;
```

```java
    if (SQRT_SAMPLES == 1) {
      b += 0.5;
    } else {
      b += randomDoubles.random();
    }

    b = VIRTUAL_SCREEN_RATIO * (HALF_HEIGHT - b);

    for(int j = 0; j < SQRT_SAMPLES; j++) {
      double a = x + INVERSE_SQRT_SAMPLES * j;
      if (SQRT_SAMPLES == 1) {
        a += 0.5;
      } else {
        a += randomDoubles.random();
      }

      a = VIRTUAL_SCREEN_RATIO * (a - HALF_WIDTH);

      Vec.map(p, c, u, v, a, b);
      Vec.constructUnitVector(d, p, EYE);

      boolean hit = false;
      bestIntersection.time = Double.POSITIVE_INFINITY;
      for(int k = scene.length - 1; k >= 0; k--) {
        IObject object = scene[k];
        if (object.intersect(EYE, d, true, Double.POSITIVE_INFINITY,
            temps, intersection)) {
          if (intersection.time < bestIntersection.time) {
            hit = true;
            bestIntersection.time = intersection.time;
            Vec.assign(bestIntersection.normal, intersection.normal);
            Vec.assign(bestIntersection.hit, intersection.hit);
            bestIntersection.material = intersection.material;
          }
        }
      }

      if (hit) {

        if (bestIntersection.material.ambientWeight > 0) {
          for(int k = 0; k < 3; k++) {
            pixel[k] += bestIntersection.material.ambientWeight
                * bestIntersection.material.diffuseColor[k]
                * RADIANCE_SCALE
                * AMBIENT_COLOR[k];
          }
        }

        if (Vec.dot(bestIntersection.normal, d) >= 0) {
          Vec.negate(bestIntersection.normal);
        }

        Vec.constructUnitVector(l, LIGHT, bestIntersection.hit);

        double nDotl = Vec.dot(l, bestIntersection.normal);
        if (nDotl <= 0) {
          continue;
        }

        double maxTime = Vec.distance(bestIntersection.hit, l);

        boolean illuminated = true;
        for(int k = scene.length - 1; k >= 0; k--) {
          IObject object = scene[k];
          if (object.intersect(bestIntersection.hit, l, false,
              maxTime, temps, intersection)) {
            illuminated = false;
            break;
          }
        }

        if (illuminated) {

          if (bestIntersection.material.diffuseWeight > 0) {
            for(int k = 0; k < 3; k++) {
              pixel[k] += bestIntersection.material.diffuseWeight
                  * bestIntersection.material.diffuseColor[k]
                  * INVERSE_PI
                  * RADIANCE_SCALE
                  * LIGHT_COLOR[k]
                  * nDotl;
            }
          }

          if (bestIntersection.material.specularWeight > 0) {

            Vec.scale(r, bestIntersection.normal, 2.0 * nDotl);
            Vec.subtract(r, l);
            double rDotMd = Vec.dotNegative(r, d);
            if (rDotMd > 0) {
              for(int k = 0; k < 3; k++) {
                pixel[k] += bestIntersection.material.specularWeight
                    * Math.pow(rDotMd,
                        bestIntersection.material.shininess)
                    * RADIANCE_SCALE
                    * LIGHT_COLOR[k]
                    * nDotl
                    * bestIntersection.material.highlightColor[k];
              }
            }
          }
        }
      }
    }
  }

  int value = 0;
  for(int i = 0; i < 3; i++) {
    int intensity = (int)Math.round(255
        * Math.pow(pixel[i] * INVERSE_SAMPLES, INVERSE_GAMMA));
    if (intensity < 0) {
      intensity = 0;
    } else if (intensity > 255) {
      intensity = 255;
    }
    value <<= 8;
    value |= intensity;
  }

  pixels[x] = value;
```

```
      }

      rowCompleted(y, pixels);
    }
  }
```

Here is the result with 256 samples per pixel:

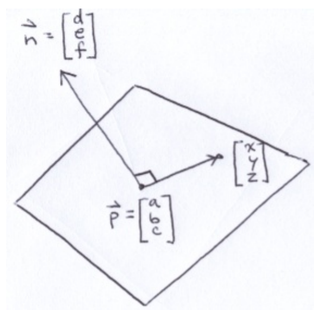


0 hours, 0 minutes, 3 seconds, 552 milliseconds
source

If you study the code, you'll notice that I did not plug in the reflection term of the Phong Shading model yet.

Step 9

The ground is a checkerboard covered plane. Below is a plane containing point **p** with a surface normal of **n**. The point (x, y, z) is some other point on the plane. Note that the vector between that point and **p** is perpendicular to **n**.



Since those vectors are perpendicular, we can take advantage of dot product to derive the equation for the plane.

$$\begin{bmatrix} x-a \\ y-b \\ z-c \end{bmatrix} \cdot \begin{bmatrix} d \\ e \\ f \end{bmatrix} = 0$$

$$d(x-a) + e(y-b) + f(z-c) = 0$$

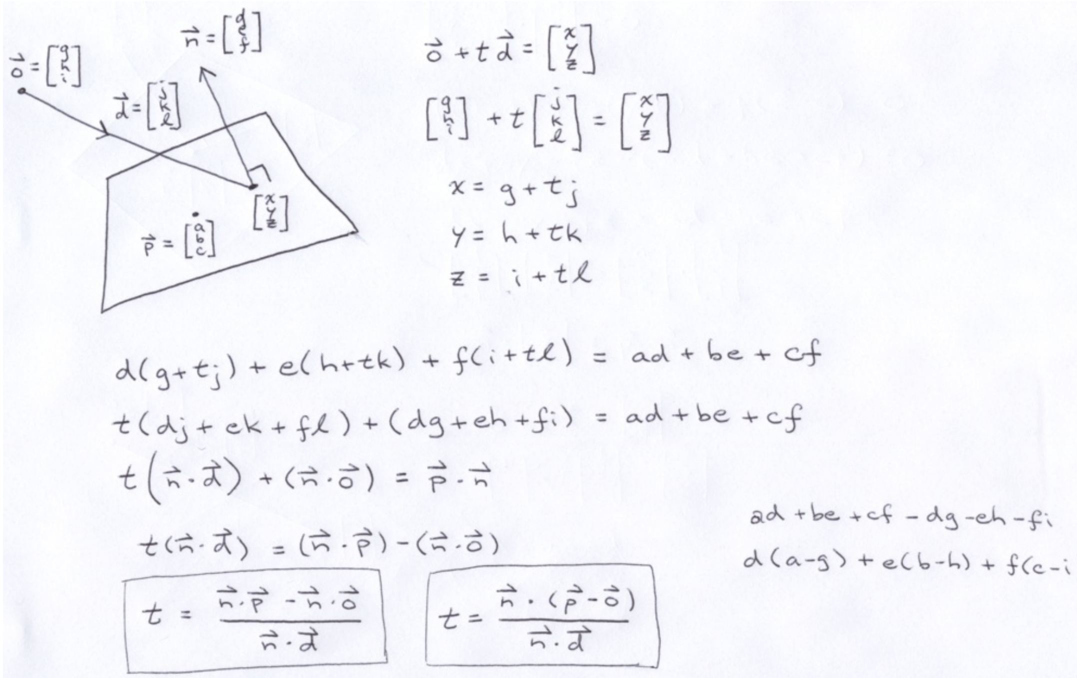
$$dx + ey + fz - (ad + be + cf) = 0$$

$$\boxed{dx + ey + fz = ad + be + cf}$$

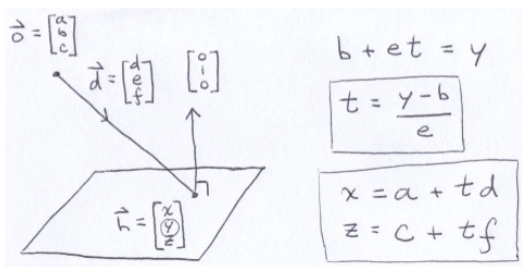
$$\begin{bmatrix} d \\ e \\ f \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

$$\vec{n} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \vec{p} \cdot \vec{n}$$

But, for this project, the ground is just the xz-plane. Meaning, **p** = (0, 0, 0) and **n** = (0, 1, 0). That leaves us with just y = 0. Nevertheless, we can still derive the generic equation for ray-plane intersection.

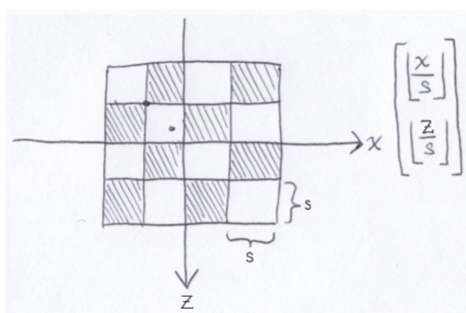


Our case is a little simpler:



Above, y = 0, simplifying it further.

Below is an aerial view of the xz-plane looking straight down the y-axis. The checkerboard consists of squares with side length s. For some point on a square, if we divide each coordinate by s and then floor it, we get a pair of integer coordinates of the upper-left corner of the square.



Notice that diagonals always have the same color. We can represent diagonals going from the lower-left to the upper-right by the following equation where t and a are integers.

$$\begin{bmatrix} t \\ t + a \end{bmatrix}$$

Above, a is the z-intercept. Adjusting it affects where the diagonal line crosses the z-axis. If we add the components together, we get:

$$\boxed{2t + a}$$

2t is an even number. Even plus even is even and even plus odd is odd. The parity (even/odd) of the sum is completely determined by a. This means we can determine the color of any point on the plane by dividing each coordinate component by s, flooring them and adding the resulting integer components together. The parity of the sum determines the color.

Based off the above discussion, I created the `Ground`:

```
public class Ground implements IObject {

  public double squareSize;
  public double inverseSquareSize;
  public Material material1;
  public Material material2;

  public Ground(double squareSide, Material material1, Material material2) {
    this.squareSize = squareSide;
    this.material1 = material1;
```