Write a Java program that attempts to model a lightsaber battle by moving two line segments in a random motion in a 3D space that is 600x600x600 pixels in size.

The program must be an animation: that is, the picture must change over time in a smooth way so that the lightsabers appear to be moving.

In a computer animation, a frame, is the static image that is copied to the display. Thus, when a computer animation is said to be running at 60 frames per second, it means that there are 60 calls to repaint() each second.

The simplest way to make an object appear to move across a monochromatic background in a computer animation is to repeat the following process:

1) Fully fill an off-screen buffer with the background color.
2) Change the value of the x and y variables used to define the location of the object.
3) Draw the object, on the off-screen buffer, in the location specified by the x variable and a y variable of the object.
4) Copy the off-screen buffer to the display (call repaint).
5) When the next timer event occurs, return to step 1.
The speed of an object is the distance it travels in a unit of time. The speed of an object is a single number.

The velocity of an object is both the speed and the direction of an object.

In this project, the speed of an object is the number of pixels it moves in one frame.

In this project, the velocity of an object is the change in its x, y and z-coordinates in one frame.

For example: If the upper-left corner of an object is \( (x=100, y=100, z=100) \) in frame 33 and is \( (x=103, y=96, z=100) \) in frame 34, then:

- The speed of the object is 5 (since \( \sqrt{3^2 + (-4)^2 + 0^2} = 5 \)) and
- The velocity of the object is the vector \( (v_x=+3, v_y=-4, v_z=0) \).

Orthographic Projection:

An orthographic projection is a parallel projection where all the projection lines are orthogonal (perpendicular) to the projection plane. Objects drawn in any type of parallel projection do not appear larger or smaller as they extend closer to or away from the viewer.

In this lab, we will be using the “top view” orthographic projection. That is a parallel projection that looks straight down the z-axis.

This is a very easy type of projection: to project, just ignore the z-coordinate.

Definition of Lightsaber to Lightsaber Collision:

In this project the lightsabers are said to collide when the following two conditions are met:

1) The orthographic projection of the lightsabers onto the x-y plane, intersect
2) The z-ranges of the two lightsabers overlap. For example, if the z-values of the endpoints of one lightsaber are -300 and -200 and the z-values of the endpoints of the other lightsaber are 100, and 101, then the two savers do not intersect even if their projections do intersect.

Note: this is not an accurate definition of a pair of line segments intersecting in 3D space. It is, however, simple and good enough for our purposes.
The goal of this project is to draw the coolest looking 3D model of a lightsaber battle using only the drawing methods of java.awt.Graphics2D. Your program must meet the following constraints:

1) [1 points]: filename must be **Lightsaber_yourfirstname_yourlastname.java**

2) [10 points]: Each lightsaber must be drawn in a different color and must be 10 to 20 pixels wide. The outer parts of the line has a different brightness and color saturation than the inner parts. This may be done as a layering of lines, thick to thin. Alternatively, Java’s Gradient paint methods (see [http://www.java2s.com/Code/Java/2D-Graphics-GUI/GradientPaintdemo.htm](http://www.java2s.com/Code/Java/2D-Graphics-GUI/GradientPaintdemo.htm)), or other techniques may be used.

3) [8 points]: The lightsabers must reverse direction when they collide (see above for the definition of collide in this system). Note: “reverse” is not precisely defined in this spec. The simplest idea of reverse will be to change the sign of the x, y and z components of the velocity of both endpoints of both lightsabers. However, if you come up with something that you think looks cooler then use that instead.

Note: The purpose of this program is to display a cool graphical effect. You may choose to always or sometimes allow the 10 to 20 pixel wide lightsabers slightly overlap before you define the event as a collision. The requirement is that the collisions appear to the user as being correct. What appears correct has some latitude for variation.

4) [4 points]: Each lightsaber must maintain a constant length of 300 pixels (±1 pixel). Note: length is defined as: \( \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \)

5) [3 points]: Each endpoint must reverse direction when it hits a boundary (0 through 600 pixels in x, y and z directions). The simplest idea of reverse in this context will be to change the sign of the x, y and z components of the velocity of the offending endpoint. As with reversal on collision, if you come up with something that you think looks cooler, then use that instead.

6) [2 points]: One endpoint on each lightsaber must move at least 2 times faster than the other.

7) [4 points]: Your program must display a JFrame (you may use the Picture class) that has a 600x600 pixel drawing area that shows an orthographic projection of the lightsaber battle straight down the z-axis. Since you will
be using a parallel projection straight down the z-axis, the transformations is simply to plot the x and y-coordinates while ignoring the z-coordinate.

8) [4 points]: The lightsabers, must not just bounce around randomly. There must be some attractive "force" that brings them together. There are many ways this can be done. The goal is to find something that looks cool.

9) [5 points]: The model must update (lines have their velocity/position change and screen repainted) at a rate no slower than 30 frames per second when run on one of the lab computers in ESPC 110 or ESPC 109.

10) [4 points]: Must play short sound clip when lightsabers collide. Suggestion: use SoundPlayer.java from the class website. You may record your own sound effect or download one from the web. If you download a sound effect, add a credit to your code showing the author/website. Either way, include the .wav file with your submission.

11) [5 points]: Must show special graphic animation when lightsabers collide. Suggestion: keep a count of the frame number. Whenever you have a "hit" set an instance variable, hitTurn to the current frame number. Then, as long as the frame number is less than 10 or 20 frames after the hitTurn, draw a frame of the graphic animation.

Note: for this lab, some useful equations to review from Algebra I are:

\[
y = mx + b, \quad m = \frac{y_2 - y_1}{x_2 - x_1}, \quad b = y_1 - mx_1
\]

Note: division by zero in slope: if \( x_1 = x_2 \), the line is vertical and the slope equation will result in a division by zero. This can be avoided by adding 1.0 to \( x_1 \) if and only if \( x_1 = x_2 \). Adding 1.0 to \( x_1 \) in this way creates a slight inaccuracy. However, inaccuracies too small for the user to see are okay.

Note: Non-integer speeds: It is perfectly reasonable for an object to move only 0.1 pixels per frame. Of course, an object cannot be moved a fraction of a pixel. However, after 10 frames, a speed of 0.1 pixels per frame will move the object one full pixel.

Note: The Pivot Point: The specifications for how the pivot point should move and how the endpoints should move are intentionally vague so that they remain open to creativity. The goal is to move the points so that the lines look like battling lightsabers. You must keep the 3D length of each line segment constant at 300 pixels, but how you move the points is mostly up to you.
Deciding a Collision in x-y plane:

Let the endpoints of line $A$ be $(x_1, y_1)$ and $(x_2, y_2)$.

Let the endpoints of line $B$ be $(x_3, y_3)$ and $(x_4, y_4)$.

Let the equation of line $A$ be $y = m_A x + b_A$

Let the equation of line $B$ be $y = m_B x + b_B$

Then, if $m_1 = m_2$, the lines are parallel and do not intersect.

If $m_1 \neq m_2$, the intersection of line $A$ with line $B$ is $(x_5, y_5)$ where:

$$x_5 = \frac{b_B - b_A}{m_A - m_B}, \quad y_5 = m_A x_5 + b_A$$

The above equations find the intersection of infinite lines $A$ and $B$. In this project, $A$ and $B$ are line segments. The line segments will NOT intersect if any one of the statements below is true:

- $x_5$ is greater than both $x_1$ and $x_2$.
- $x_5$ is greater than both $x_3$ and $x_4$.
- $x_5$ is less than both $x_1$ and $x_2$.
- $x_5$ is less than both $x_3$ and $x_4$.
- $y_5$ is greater than both $y_1$ and $y_2$.
- $y_5$ is greater than both $y_3$ and $y_4$.
- $y_5$ is less than both $y_1$ and $y_2$.
- $y_5$ is less than both $y_3$ and $y_4$.

In the case pictured above, $x_5$ is less than both $x_3$ and $x_4$. Thus, the line segments do not intersect.

If Collision in x-y plane, check for collision along z-axis:
Keeping the line a length of 300 pixels in 3D (using vector normalization)

Let \( \overline{P_1P_2} \) be a line segment in 3D space with endpoints \( P_1 \) and \( P_2 \).

Let the coordinates of \( P_1 \) be \((x_1, y_1, z_1)\).

Let the coordinates of \( P_2 \) be \((x_2, y_2, z_2)\).

Goal: Find a point \( P_3 \) on that is as distance of 300 pixels from \( P_1 \) and is in the direction of \( P_2 \).

Steps:

1) Calculate the distance \( d \) from \( P_1 \) to \( P_2 \):
\[
d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}
\]

2) The line segment \( \overline{P_1P_2} \) has the same length and the same direction as the vector \((x_2-x_1, y_2-y_2, z_2-z_1)\).

3) Normalize the vector \((x_2-x_1, y_2-y_2, z_2-z_1)\) by dividing each component by the distance from \( P_1 \) to \( P_2 \):
\[
\left( \frac{x_2-x_1}{d}, \frac{y_2-y_1}{d}, \frac{z_2-z_1}{d} \right)
\]

4) The vector from step 3 has a length of 1.0. Change the length to 300 by multiplying each coordinate by 300:
\[
\left( \frac{300 \times (x_2-x_1)}{d}, \frac{300 \times (y_2-y_1)}{d}, \frac{300 \times (z_2-z_1)}{d} \right)
\]

5) Finally, add this vector of length 300 to the coordinates of \( P_1 \) and to get the coordinates of \( P_3 \):
\[
x_3 = x_1 + \frac{300 \times (x_2-x_1)}{d}
\]
\[
y_3 = y_1 + \frac{300 \times (y_2-y_1)}{d}
\]
\[
z_3 = z_1 + \frac{300 \times (z_2-z_1)}{d}
\]

The above procedure can be used after you have decided where you want one of the end points to be. Alternatively, you could apply this method twice: once from the pivot point to one endpoint and a second time from the pivot point to the other endpoint. Say, for example, that you want your pivot point to be 100 pixels from one endpoint and 200 pixels form the other. Then, use the method above toward one endpoint by replacing all the length=300 with length=100. Finally, apply it a second time toward the other endpoint using length=200.
Note: Squaring a number in Java.
Java does have a build-in math function for raising a number to a power. However, then that power is 2 (as in $x^2$) the simplest and most efficient way to do this is $x^2$. Thus, $y = (x_1-x_2)^2$ would best be coded as:

$$y = (x_1-x_2)*(x_1-x_2);$$

or as

```java
double dx = x1-x2;
y = dx*dx;
```

Note: Square root a number in Java.
In Java, $\sqrt{101}$ can be calculated using the static method of the Math class:

```java
Math.sqrt(101);
```