Project 3 - Rectangular Frame Animation

Authors:

Julio Lopez

Audrey Nelson

## Problem Statement

We want the ability to manipulate two rectangles in respect to their scaling and rotation. In addition, we want the ability to drag a rectangle on its centroid. Rotation and scaling will occur at the vertex closest to the mouse. In addition, we want to morph one rectangle into another by transformation via the logarithmic spiral formulae.

## Solution

- Create rectangle class representative of the rectangles we want to create/manipulate

```
public class Rectangle {
    float height; float width;
    pt v0; pt v1; pt v2; pt v3; pt center;
    color c;

   public Rectangle(float width, pt center, color c) {
       this.height = 2 * width;
       this.width = width;
       this.center = center;
       center.setCenter();
       this.c = c;
       v0 = new pt((center.x - width/2), (center.y - height/2));
       v1 = new pt((center.x + width/2), (center.y - height/2));
       v2 = new pt((center.x + width/2), (center.y + height/2));
       v3 = new pt((center.x - width/2), (center.y + height/2));
   }

   void drawRectangle() {
     strokeWeight(5);
       stroke(c);
       noFill();
       beginShape();
         vertex(v0.x, v0.y);
         vertex(v1.x, v1.y);
         vertex(v2.x, v2.y);
         vertex(v3.x, v3.y);
       endShape(CLOSE);
       stroke(blue);
       strokeWeight(1);
       ellipse(v0.x, v0.y, 15, 15);
       v0.label("A", -3, 3);
       ellipse(v1.x, v1.y, 15, 15);
       v1.label("B", -3, 3);
       ellipse(v2.x, v2.y, 15, 15);
       v2.label("C", -3, 3);
       ellipse(v3.x, v3.y, 15, 15);
       v3.label("D", -3, 3);
```

- **Closest Point detection:** Create a getClosestPoint method to calculate which vertex is closest to the mouse upon mouse click. This is important because this will determine which vertex we will be using for rotating/scaling. We use the distance formula to calculate distance between the vertices to the mouse as well as the center of the rectangle to the mouse.

```
pt getClosestPoint() {

  pt curr = center;
  if (v0.getDistance(Mouse()) < curr.getDistance(Mouse())) {
      curr = v0;
  }
  if (v1.getDistance(Mouse()) < curr.getDistance(Mouse())) {
      curr = v1;
  }
  if (v2.getDistance(Mouse()) < curr.getDistance(Mouse())) {
      curr = v2;
  }
  if (v3.getDistance(Mouse()) < curr.getDistance(Mouse())) {
      curr = v3;
  }

  return curr;
}
```

```
void mousePressed() {  // executed when the mouse is pressed
  if(keyPressed && key==' ') S.empty();
  if(!keyPressed) P.pickClosest(Mouse()); // used to pick the closest vertex of C to the mouse
  change=true;

  if(rect1.getClosestPoint().getDistance(Mouse()) < rect2.getClosestPoint().getDistance(Mouse())){
     currRect = rect1; closestPt =  rect1.getClosestPoint();
  } else{
    currRect = rect2; closestPt =  rect2.getClosestPoint();
  }
  realStartingPt = Mouse();
  isRotating = false;
  }
```

- **Dragging:** Created a dragRectangle method that will translate the rectangle in any direction the user specifies upon holding and dragging the mouse click. dragRectangle will be triggered if the closest point of any rectangle is the center. Dragging is accomplished by adding the components of the mouse click shift(mouseX-pmouseX, mouseY –pmouseY) to each vertex in the rectangle, including the center.

```
void dragRectangle() {
    center.x += mouseX - pmouseX;
    v0.x += mouseX - pmouseX;
    v1.x += mouseX - pmouseX;
    v2.x += mouseX - pmouseX;
    v3.x += mouseX - pmouseX;

    center.y += mouseY - pmouseY;
    v0.y += mouseY - pmouseY;
    v1.y += mouseY - pmouseY;
    v2.y += mouseY - pmouseY;
    v3.y += mouseY - pmouseY;
}
```

- **Rotation:** Created a rotateBy method that will rotate the closest vertex by an angle theta, around the center of the currently chosen rectangle. Before calculating theta, we create two vectors representative of the mouse's previous position and the mouse's current position in relation to the center. As such, we can calculate the angle between both vectors by using the formula discussed in lecture:

atan2(det(U,V),dot(U,V))

```
void dragRectangle() {
    center.x += mouseX - pmouseX;
    v0.x += mouseX - pmouseX;
    v1.x += mouseX - pmouseX;
    v2.x += mouseX - pmouseX;
    v3.x += mouseX - pmouseX;

    center.y += mouseY - pmouseY;
    v0.y += mouseY - pmouseY;
    v1.y += mouseY - pmouseY;
    v2.y += mouseY - pmouseY;
    v3.y += mouseY - pmouseY;
}
```

- **Scaling:** Created a scale method that passes in a scale factor which scales the vertices in relation to the center of the rectangle currently selected. We created an angle "range" to determine where the currently selected rectangle can be scaled. We chose 1.2 radians (About 69 degrees) to be the area where scaling can be done for any vertex. Theta was calculated similarly as before when doing rotation. To determine the scale factor, we measured a base distance and a top distance.
    - **Base distance:** distance from the center to the previous mouse position
    - **Top distance:** distance from the center to the current mouse position

- **Scale factor:** Top distance/base distance

```
    startingPt = Pmouse();
    startVec = new vec((startingPt.x - currRect.getCenter().x), (startingPt.y - currRect.getCenter().y));
    currVec = new vec((mouseX - currRect.getCenter().x), (mouseY - currRect.getCenter().y));
    vec startVecScale = new vec(-(closestPt.x - currRect.getCenter().x), (closestPt.y - currRect.getCenter().y));
    vec currVecScale = new vec((mouseX - currRect.getCenter().x), (mouseY - currRect.getCenter().y));
  if(closestPt.isCenter()){
    currRect.dragRectangle();
  } else{
    theta = -(atan2(det(startVec, currVec), dot(startVec, currVec)));
    float thetaScale = -(atan2(det(startVecScale, currVecScale), dot(startVecScale, currVecScale)));
    text("This", closestPt.x, closestPt.y);
    if (abs(thetaScale) < 1.2 && !isRotating) {
      float baseDistance = currRect.getCenter().getDistance(startingPt);
      float topDistance = new pt(mouseX, mouseY).getDistance(currRect.getCenter());
      currRect.scale(topDistance/baseDistance);
    } else {
      currRect.rotateBy(theta);
    }
  }
```

- **Animation:** Created an animate method to set up new animating rectangle when the ‘A’ key is pressed. The animate method sets up the attributes of the animating rectangle such as time, positioning, and setting the Boolean value ‘animating’ to true.

```
void animate(){
 animationRect = new Rectangle(rect1.width, rect1.center, black);
 startVec = new vec((animationRect.v0.x - animationRect.getCenter().x), (animationRect.v0.y - animationRect.getCenter().y));
 currVec = new vec((rect1.v0.x - rect1.getCenter().x), (rect1.v0.y - rect1.getCenter().y));
 theta = -(atan2(det(startVec, currVec), dot(startVec, currVec)));
 animationRect.rotateBy(theta);
 strokeWeight(5);
 //animationRect.drawRectangle();
 time = 0;
 animating = true;
```

Once the boolean value animating is verified, we created an if structure in the draw method that handles the morphing of the new rectangle to the second rectangle. We created float values for the spiral angle and spiral scale as well as the center point of the spiral F based off of professor’s Rossignac provided methods in the morph class. Similarly, we used the provided Spiral method to transform the animating rectangle’s vertices into the second triangle by following a logarithmic spiral. Based on the equation presented in class, we figured out that the animating rectangle will be at the second rectangle’s position once time equals 1. At this point, we stop animation.

```
a =spiralAngle(rect1.v0,rect1.v1,rect2.v0,rect2.v1);
s =spiralScale(rect1.v0,rect1.v1,rect2.v0,rect2.v1);
F = spiralCenter(a, s, rect1.v0, rect2.v0);
fill(white); stroke(magenta); show(F,13); fill(black); label(F,"F");

if(animating){
    animationRect.v0 = Spiral(rect1.v0, time/30, a, s, F);
    animationRect.v1 = Spiral(rect1.v1, time/30, a, s, F);
    animationRect.v2 = Spiral(rect1.v2, time/30, a, s, F);
    animationRect.v3 = Spiral(rect1.v3, time/30, a, s, F);
    animationRect.center = Spiral(rect1.center, time/30, a, s, F);
    animationRect.drawRectangle();
   // println("Time " + time + " X: " + animationRect.v0.x + " TargetX: " + rect2.v0.x + " FX: " + F.x);
    time++;
    if (time>30){ animating = false;}
}
```

**Advantes of log spiral over LERP:** Utilizing a logarithmic spiral gives us a steady interpolation by providing a linear interpolation between two angles as well as an exponential interpolation of the magnitude. As such, it is numerically more stable than LERP. In LERP, the angle between consecutive instances is not constant. This results in a non-steady pattern or motion.