Algorithms for Game Design Exercise #5

To be completed after Session 5, before Session 6

---

1) Textbook

If you haven't already, read Chapters 0 - 5.

Then, read Chapter 10: Networking.

---

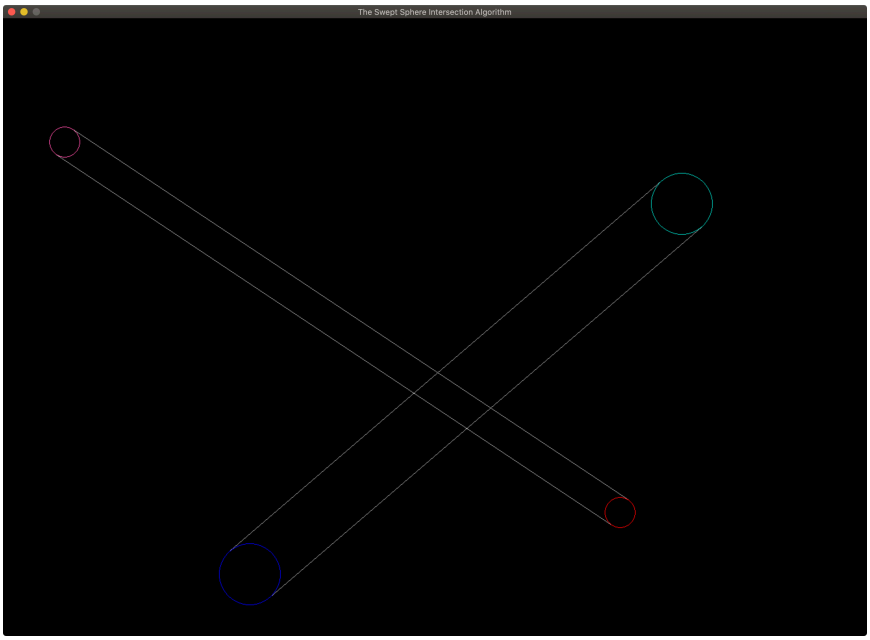
2) Swept Sphere Algorithm

THIS IS COMPLETELY OPTIONAL. Do it if you have some spare time and want a bit of a challenge.

We didn't have a chance to dig into the Swept Sphere Algorithm, which calculates collisions and precise collision point for 2- or 3-d spheres. I've given you scans of several pages from "Game Programming: Algorithms and Techniques" by Sanjay Madhav, ISBN 0-321-94015-6.

Read those pages. Depending on your degree of understanding of vector math, you may have some trouble following the derivation. Don't worry, if that's the case. Instead, focus on the idea that we can calculate a parametric equation (the one on page 144, for the variable t). The variable t will represent the "time" at which the surfaces of the spheres collide. A value of 0 means "at the current frame" and a value of 1 means "at the next frame." Any other value means either "no intersection" or "no collision that we have to worry about."

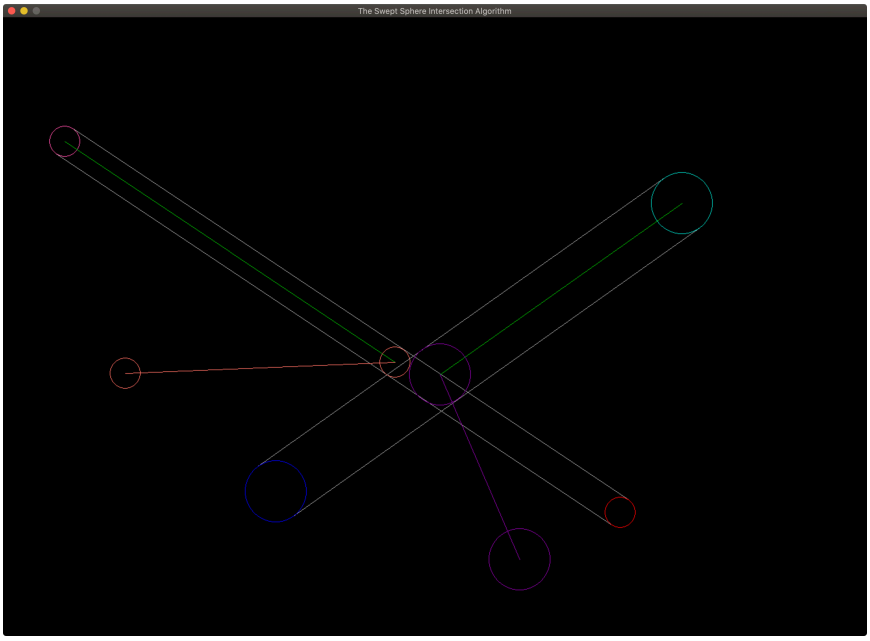
Check out the code in **`SweptSphere.py`**. This code implements the swept sphere algorithm in two dimensions. If you run it, you'll see four circles, connected by four line segments.



This represents two circles, each of which is moving along the path from a source location (pink, turquoise) to a destination location (red, blue). The idea is that this is the "now" and "next" frame locations of the circles.

You can click and drag any of the circles to set different starting and ending locations.

If a collision will occur, based on the set circle locations, you will see some additional geometry. The pink/red circle will now show a green line of the path it will follow before collision, a salmon-colored circle at the point of collision, and a line and circle showing the post-collision path and ending location. Similarly, the turquoise/blue circle will show a



purple collision point, path and ending location.

Drag the circles around a bit to see that they work.

It should be noted that the collision is modeled as perfectly elastic as if the two circles had identical mass. That is, the total distance traveled for each circle is the same distance it would have travelled if there were no collision -- no energy is lost or transferred. This model may not match how all collisions would work.

Now, take a look at the code. It isn't the prettiest code I've ever written, but I want you to see the **`calculate_collision`** function. In particular, see how using Pygame's Vector2 class made the math work in a very similar way to the math in the handout. I'm a big fan of using the Vector class whenever necessary.