CELL

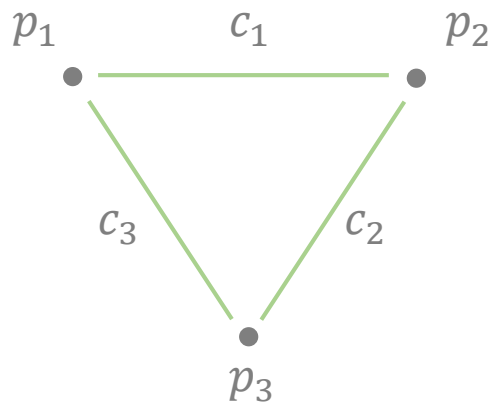
Quarter 2 Evidence Packet

Owen Martin

VERLET INTEGRATION

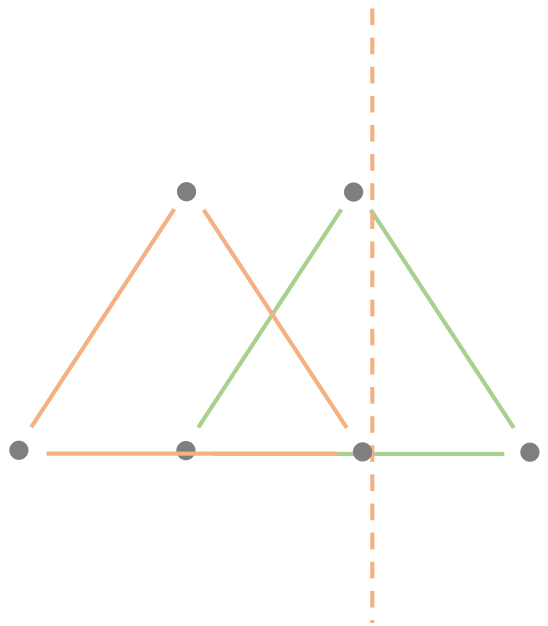
Verlet integration is a popular method of physical modeling in computer science. Any object being simulated is comprised of both particles and connectors. The alignment of the particles defines the physical shape of the object, which is maintained by the initial arrangement of its connectors.

When simulated with Verlet integration, an object's connectors enable the program to determine whether or not the object has collided with a boundary. If it has, the program will deflect the object's particles accordingly, creating a very realistic simulation of a collision.



A simple triangle object made of three points and three connectors.

OBJECT COLLISIONS



A green triangle object that passes a dotted orange boundary and is repositioned as the orange triangle.

Collisions with Verlet integration involve boundaries, typically those of objects with other objects or with applet walls. If a moving object passes a boundary, any of its offending points will be repositioned to the inside of the barrier. The object's connectors maintain its shape by moving the other points to their initial orientations.

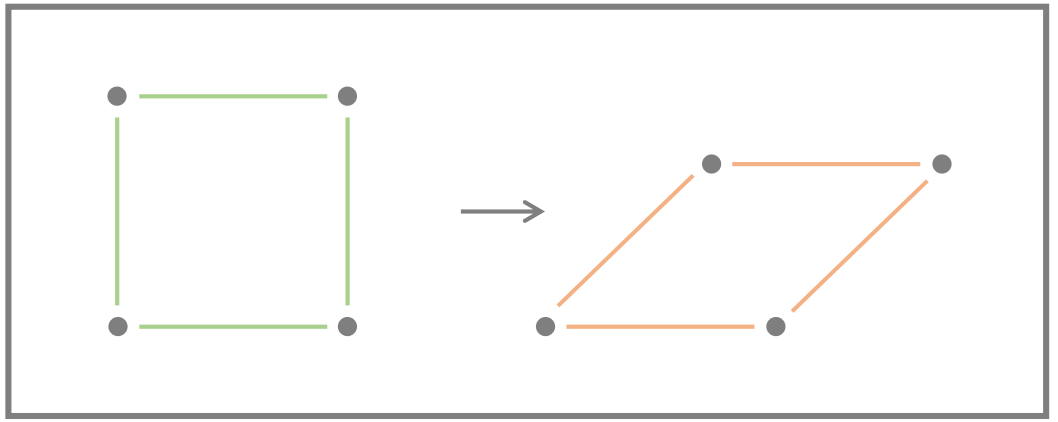
If an object is in motion, it is said to have velocity. When a particular point of an object passes a boundary, its velocity is reflected along whichever axis the collision occurs. For instance, in the situation to the left, the initial green triangle's rightward velocity would be translated to leftward motion in the orange triangle.

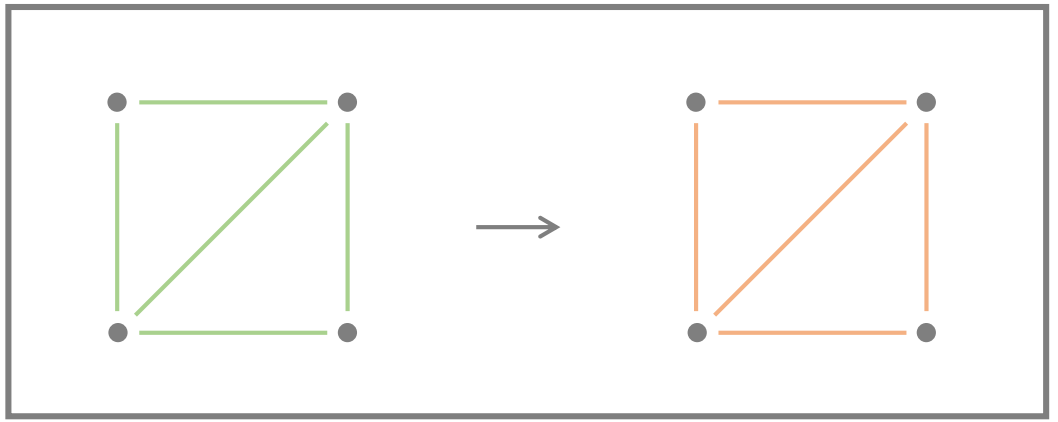
DYNAMIC OBJECTS

Objects comprised of many points will take on a dynamic shape depending on the setup of their connectors. For instance, a square object of four points can be instantiated as having either four, five, or six connectors.

If a square object has four connectors, it appears as shown in the top right box. Under the influence of a collision force, it collapses into a rhombus.

If a square object has five connectors, it appears as shown in the bottom right box. Under the influence of a collision force, it maintains its original shape. The same scenario occurs with six connectors.



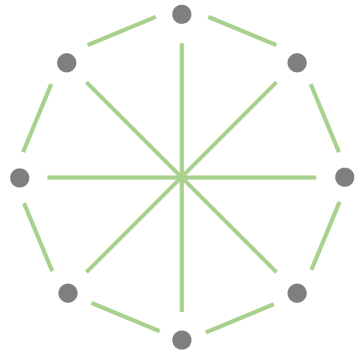


REVISED CELL STRUCTURE

Considering the dynamic structure of multi-pointal objects, it becomes possible to simulate a cell membrane with a plethora of points connected radially and circumferentially to each other.

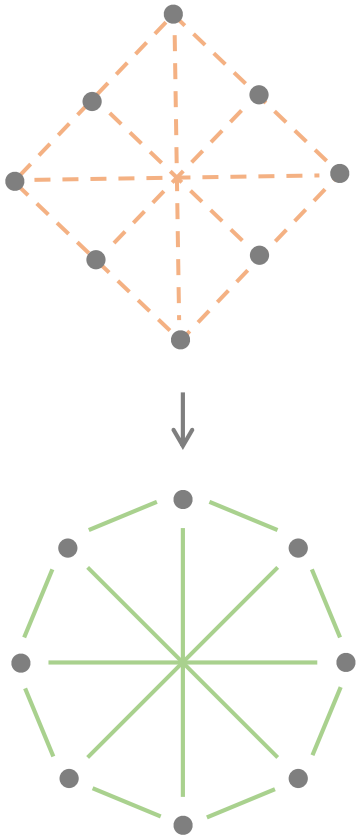
The slightly oversimplified cell to the right, when subjected to an environment of Verlet integration, has a malleable and dynamic structure that changes based on collision stimuli.

The initial lengths of its connectors are related to each other geometrically: a circumferential connector has length $\pi d/n$, where d is the length of a radial connector and n is the number of points in the cell.



A simplified representation of the current CELL with both radial and circumferential connectors.

CELL GENERATION



In order to generate a dynamic structure in Verlet integration (e.g. one that can have any arbitrary number of points), the initial placement of the points must at least slightly resemble the final product.

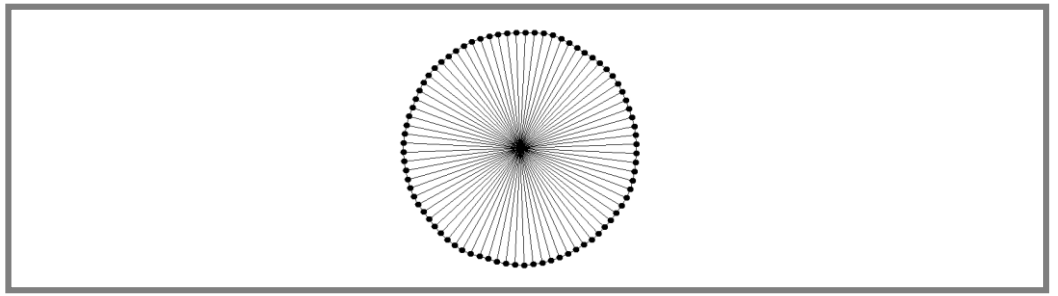
This particular program, considering that the cell is initially round, requires the formation of a rhombus from the cell's generated points at startup. These points are then pulled into the cellular circle by their connectors.

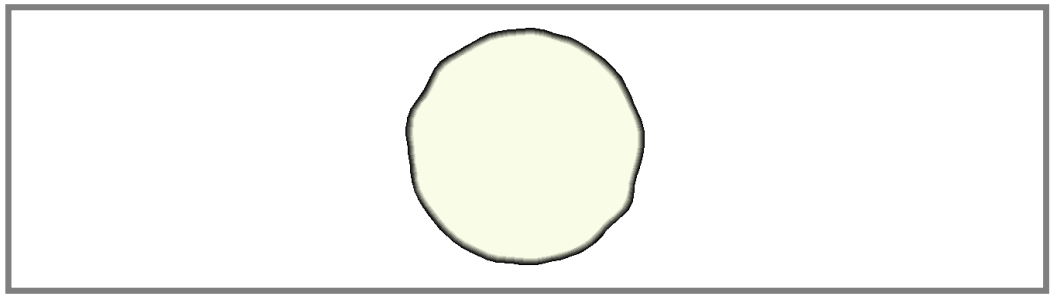
To the left is the formation of the cell membrane immediately before and after its connectors are established.

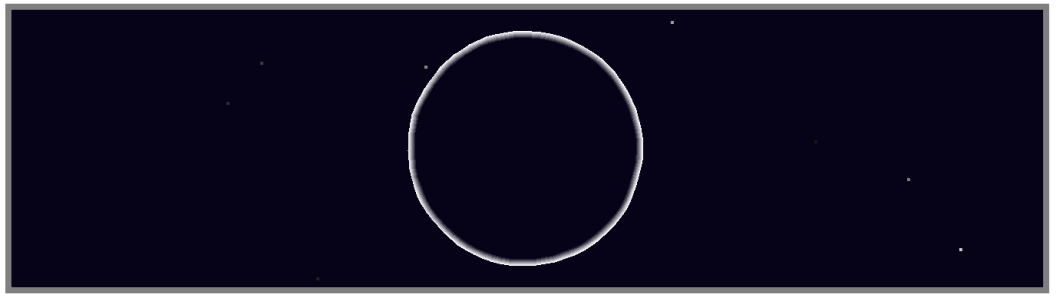
CELL DISPLAY

To add visual depth to the cell program, the initial display of points and connectors as circles and lines, respectively, was modified. In the newer version of the program, the positions of each particle in the cell are used to create a polygon object, which is filled with a particular color.

The gradient polygons to the right were created by deriving smaller polygons within the original cell and filling them with slightly different colors from their parent shapes.







CELL DISPLAY CODE

The following is an example of the code used to create colored polygons on the cell's interior.

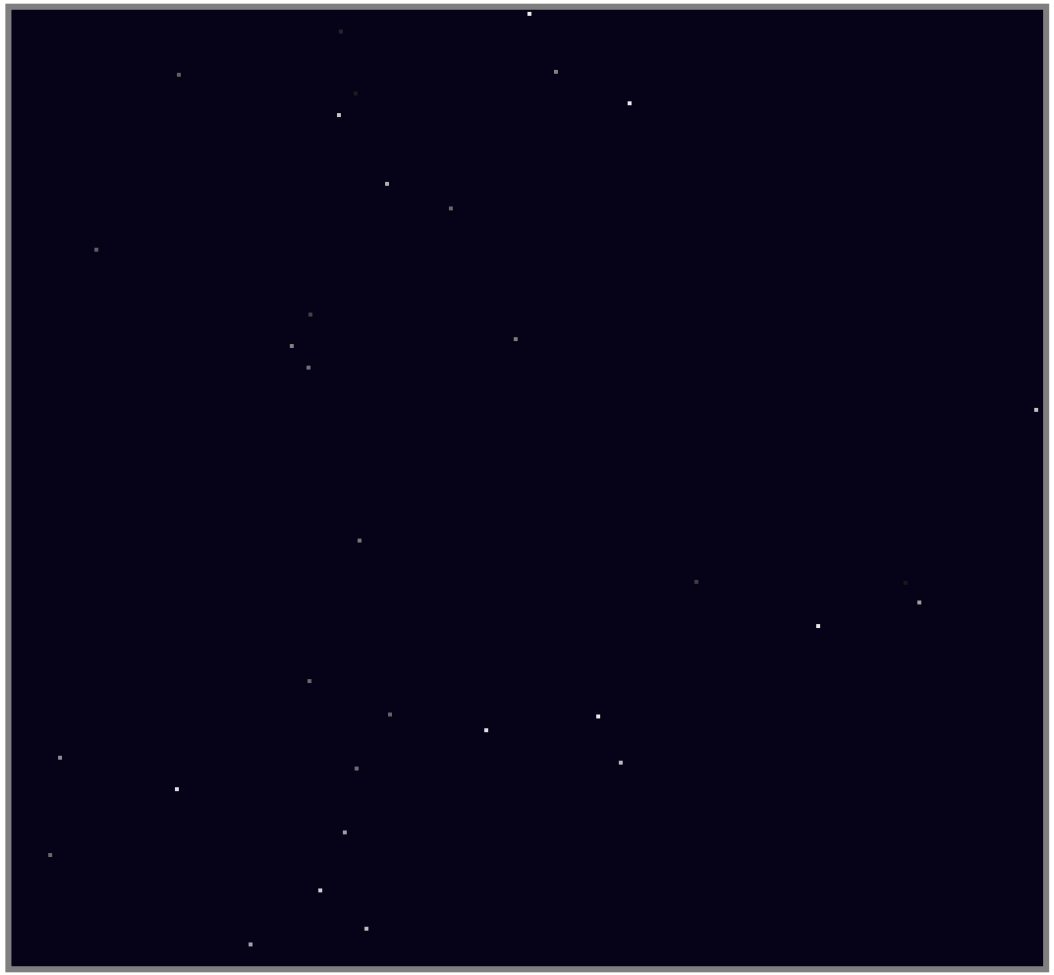
```
public void renderCell(Graphics g, double radius)
{
   double centerX = 0, centerY = 0, ratioX = 0, ratioY = 0;
   double hypot = radius / numPolygons;
   for (int p = 0; p < numPolygons; p++)
   {
      Polygon cell = new Polygon();
      for (int n = 0; n < particles.size(); n++)
      {
         if (p == 0)
         {
            centerX += particles.get(n).getX(); centerY += particles.get(n).getY();
            if (n == particles.size()-1)
            {
               centerX /= particles.size(); centerY /= particles.size();
            }
         }
         ratioX = (centerX-particles.get(n).getX())/radius; ratioY = (centerY-particles.get(n).getY())/radius;
         cell.addPoint((int)(particles.get(n).getX()+hypot*ratioX*p),(int)(particles.get(n).getY()+hypot*ratioY*p));
      }
      if (p <= numPolygons/16)
      {
         g.setColor(new Color(255-(249*p/(numPolygons/16)),255-(252*p/(numPolygons/16)),255-(231*p/(numPolygons/16))));
      }

      g.fillPolygon(cell);
      //particles.get(0).drawParticle(g,centerX,centerY);
   }
}
```

BACKGROUND DISPLAY

As seen previously, the final version of the quarter two CELL program consists of a white-bordered cell and a dark blue background. Within this background are various motes that flicker in and out of visibility behind the cell.

Resembling twinkling stars, these motes appear to reduce their opacity at regular intervals. This, however, is not the case, as the Java Paint class does not support values for opacity. Instead, the rgb values of the motes approach the color of the background and return to white after a specific time interval.



BACKGROUND DISPLAY CODE

The following is the code used to create the program's starry background.

```
for (int n = 0; n < numMotes; n++)
{
   motes.add(new Mote((int)(Math.random()*this.getWidth()),(int)(Math.random()*this.getHeight()),Math.random()));
}

public void clearCanvas(Graphics g)
{
   g.setColor(new Color(6,3,24));
   g.fillRect(0,0,this.getWidth(),this.getHeight());
   for (Mote mote : motes)
   {
      mote.incrementO();
      mote.drawMote(g,mote.getX(),mote.getY(),mote.getO());
   }
}
```

THANKS FOR WATCHING!

Stay tuned for quarter three evidence packet.