CELL

Quarter 3 Evidence Packet

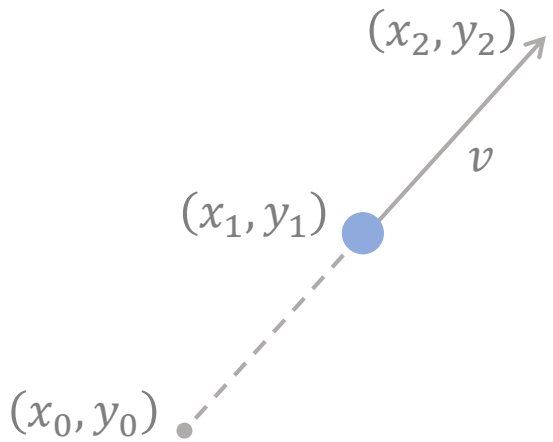
Owen Martin

THE TARGET CLASS

The Target class was created as a preliminary addition to food in the CELL program. It possesses many similarities with the Particle class, whose objects compose the cell membrane.

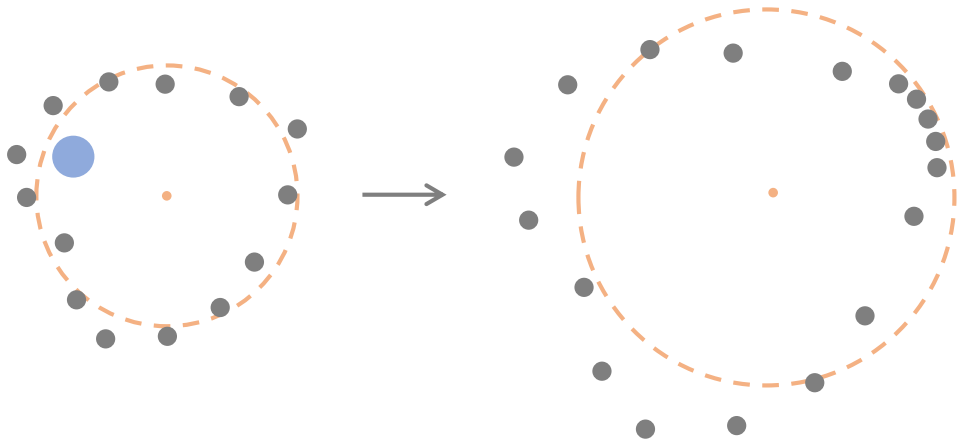
The Target objects are currently displayed as blue dots slightly larger than those of the Particle class.

Target objects have variables for current and previous x and y coordinates. Per Verlet integration, they have implicit values for x and y-component vector velocities.



An example of a Target object with depicted velocity vectors.

TARGET OBJECT CONSUMPTION



If a Target object's position is within the radii surrounding the CELL's central point, it will be converted into Particle objects.

The addition of Target objects enabled a real-time recalculation of total Particle objects within the CELL membrane. If a Target object's position is within the radial boundry of the CELL, the Target will disappear and be recreated at a random location.

When this occurs, four Particle objects are added to the membrane at the location of the last particle in the membrane's ArrayList of Particles. These particles are created with the same velocity as the final Particle in the array, and therefore have a negligible effect on the momentum of the CELL.

TARGET CLASS CODE

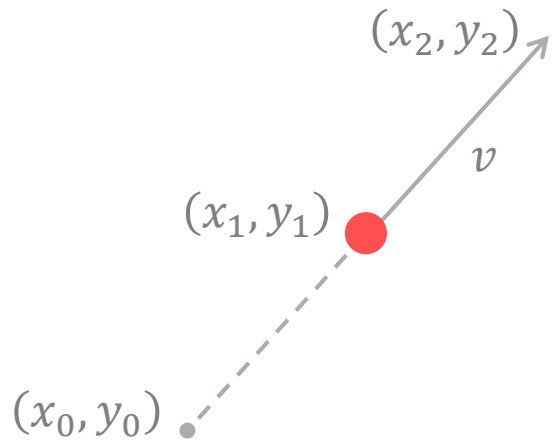
```
import java.awt.Graphics;
import java.awt.Color;

public class Target {
   double x, y, px, py;

   public Target(double x, double y, double px, double py)
   {
      this.x  = x;
      this.px = px;
      this.y  = y;
      this.py = py;
   }

   public static void drawTarget(Graphics g,double x,double y)
   {
      g.setColor(Color.blue);
      g.fillOval((int)x-6,(int)y-6,12,12);
   }

   public void addX(double x)    {   this.x += x;    }
   public void addY(double y)    {   this.y += y;    }
   public void addVx(double x)    {   this.px -= x;    }
   public void addVy(double y)    {   this.py -= y;    }

   public void setX(double x)    {   this.x = x;     }
   public void setY(double y)    {   this.y = y;     }
   public void setPx(double px)  {   this.px = px;   }
   public void setPy(double py)  {   this.py = py;   }
   public void setVx(double vx)  {   this.px = x - vx;   }
   public void setVy(double vy)  {   this.py = y - vy;   }

   public double getX()    {   return x;    }
   public double getY()    {   return y;    }
   public double getPx()   {   return px;   }
   public double getPy()   {   return py;   }
   public double getVx()   {   return x - px;   }
   public double getVy()   {   return y - py;   }
}
```

THE ENEMY CLASS

The Enemy class was created to counteract the addition of Targets to the CELL program. It also possesses many similarities with the Particle class, such as the coordinate and velocity systems.

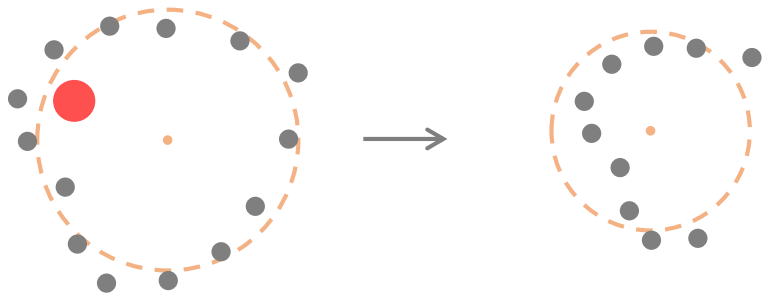
The Enemy objects are currently displayed as red dots slightly larger than those of the Particle class, equal in size to Target objects.

Enemies are single-point objects but can hunt other objects like the main CELL. This is done according to the think() method, explained later.



An example of an Enemy object with depicted velocity vectors.

ENEMY OBJECT DESTRUCTION



If an Enemy object's position is within the radii surrounding the CELL's central point, it will destroy four membrane particles.

Enemy objects operate similarly to Target objects; if one's position is within the radial boundry of the CELL, the enemy will disappear and be recreated at a random location.

When this occurs, four Particle objects are removed from the beginning of the membrane's ArrayList of Particle objects.

Because Enemy objects can actively hunt the main CELL, it is easy for the CELL to become extremely small when the total number of enemies is high. For this reason, it is necessary to maintain a small ratio of Enemy to Target objects.

ENEMY CLASS CODE

```
import java.awt.Graphics;
import java.awt.Color;

public class Enemy {
   double x, y, px, py;

   public Enemy(double x, double y, double px, double py)
   {
      this.x  = x;
      this.px = px;
      this.y  = y;
      this.py = py;
   }

   public static void drawEnemy(Graphics g,double x,double y)
   {
      g.setColor(Color.red);
      g.fillOval((int)x-6,(int)y-6,12,12);
   }

   public void addX(double x)    {   this.x += x;    }
   public void addY(double y)    {   this.y += y;    }
   public void addVx(double x)    {   this.px -= x;    }
   public void addVy(double y)    {   this.py -= y;    }

   public void setX(double x)    {   this.x = x;     }
   public void setY(double y)    {   this.y = y;     }
   public void setPx(double px)  {   this.px = px;   }
   public void setPy(double py)  {   this.py = py;   }
   public void setVx(double vx)  {   this.px = x - vx;   }
   public void setVy(double vy)  {   this.py = y - vy;   }

   public double getX()    {   return x;    }
   public double getY()    {   return y;    }
   public double getPx()   {   return px;   }
   public double getPy()   {   return py;   }
   public double getVx()   {   return x - px;   }
   public double getVy()   {   return y - py;   }
}
```

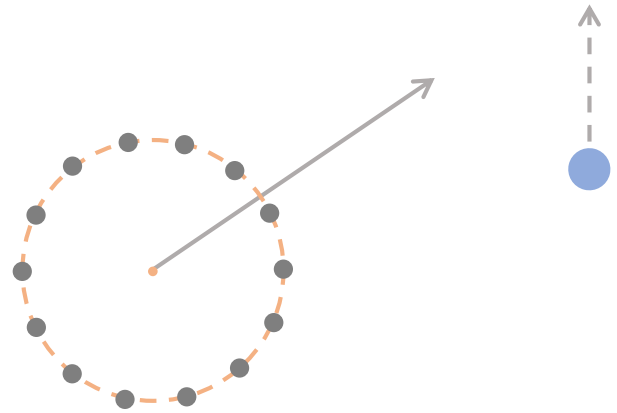
THE THINK() METHOD

The think() method is the algorithm to which every entity in CELL adheres when determining movement.

An entity will change its velocity according to the function $Ad/(d+B)$, where d is the distance between the entity and its target, A is the entity's maximum speed and B determines how quickly it will accelerate to that speed.

The direction of the velocity is determined by comparing the future x and y positions of the entity's target to the entity's center.

The entity's target is selected simply by proximity—it will hunt whichever particular iteration of its targeted class is closest.



An example of the main CELL hunting a particular Target object with depicted velocity vectors.

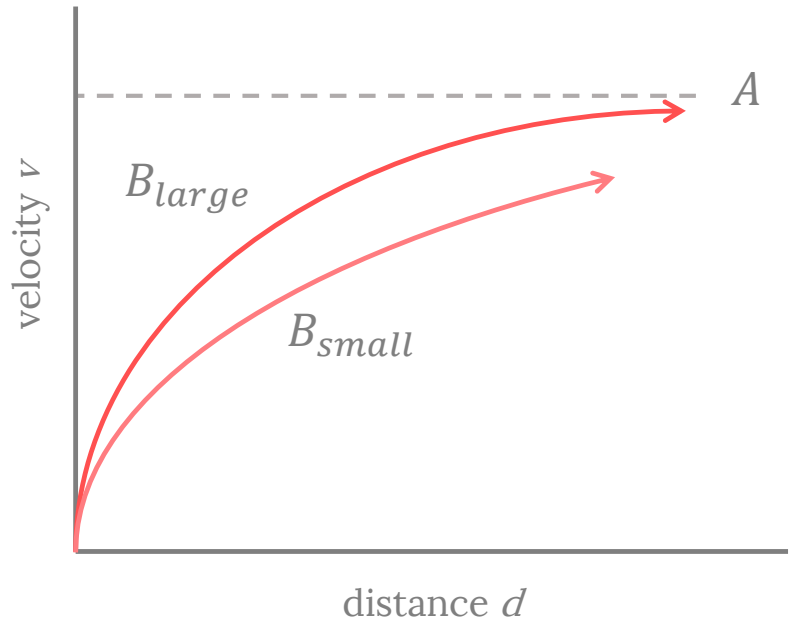
THE THINK() METHOD

The values generated by the think() method are additive, meaning that they are passed into an addVx/addVy method instead of a setVx/setVy one, emulating true physical behavior and enforcing the cellular speed limit.

The speed limit prevents an object from gaining an abnormal amount of speed if its target is located a significant distance across the screen.

The general function to the right is used to determine an entity's total speed at any given time; it is the difference between these values and the entity's current one that is added to the entity.

The general function $\frac{Ad}{B+d}$.



THINK() METHOD CODE

```
public boolean think(Graphics g, Particle center, boolean canMove, Target target)
{
   double distance = Math.hypot(target.getX()+target.getVx()-(center.getX()+center.getVx()),target.getY()+target.getVy()-(center.getY()+center.getVy()));
   if ( distance != 0 && canMove == true )
   {
      double percentX = (target.getX()+target.getVx()*mseconds/refresh-center.getX())/(Math.abs((target.getX()+target.getVx()*mseconds/refresh-center.getX()))+Math.abs((target.getY()+target.getVy()*mseconds/refresh-center.getY())));
      double percentY = (target.getY()+target.getVy()*mseconds/refresh-center.getY())/(Math.abs((target.getX()+target.getVx()*mseconds/refresh-center.getX()))+Math.abs((target.getY()+target.getVy()*mseconds/refresh-center.getY())));
      for (int n = 0; n < entities.get(n).getParticles().size(); n++)
      {
         if (cellForce + distance != 0) {
         entities.get(n).getParticles().get(n).addVx((percentX * Math.abs((speedLimit * distance) / (cellForce + distance))) - center.getVx());
         entities.get(n).getParticles().get(n).addVy((percentY * Math.abs((speedLimit * distance) / (cellForce + distance))) - center.getVy());
         }
      }
      return canMove = false;
   }
   else
   {
      return canMove;
   }
}
```

THANKS FOR WATCHING!

Stay tuned for quarter four evidence packet.