Introduction to Visual Computing

Assignment# 6

Particle System

March 23, 2020

Description

In this session, you will use a particle system to manage the obstacles in your game. The user places a villain on the board. Then, the villain repeatedly creates obstacles around it to protect itself from the ball. An obstacle disappears when the ball hits it. The goal of the game will be to defeat the villain by hitting it with the ball.

Objectives

- To understand the concept of particle systems and implement it
- To integrate a particle system into the game

Specific Challenges

- To have non-overlapping particles

Preliminary steps

If you have not completed assignment 5, please finish the missing parts.

Part I

Create a 2D Particle System

The first part is to implement a simple 2D particle system. In this part, you will learn how to create circle-shaped particles that are generated around a predetermined and fixed origin, at a random distance and direction.

Step 1 – Implementing a particle system

In order to implement a particle system, you will create a class for the particle system that manages a list of particles. The class contains the origin of the system, a method to add a new particle, and a method to update and display the system.

Try to use the code below to create a particle system that generates particles around a fixed origin. You can begin by creating the `Particle` and `ParticleSystem` classes in separate tabs. Note that you will need to complete `isDead()` and `run()` methods. Try to understand how `addParticle()` works. Run the code and see the result.

ParticleSystem Class:

```java
// A class to describe a group of Particles
class ParticleSystem {
  ArrayList<Particle> particles;
  PVector origin;
  float particleRadius = 10;

  ParticleSystem(PVector origin) {
    this.origin = origin.copy();
    particles = new ArrayList<Particle>();
    particles.add(new Particle(origin, particleRadius));
  }

  void addParticle() {
    PVector center;
    center = origin.copy();
    float angle = random(TWO_PI);
    float radius = random(min(width, height)/2);
    center.x += sin(angle) * 2*radius;
    center.y += cos(angle) * 2*radius;
    particles.add(new Particle(center, particleRadius));
  }

  // Iteratively update and display every particle,
  // and remove them from the list if their lifetime is over.
  void run() {
    //  ...
  }
}
```

Particles Class:

```
// A simple Particle class
class Particle {
  PVector center;
  float radius;
  float lifespan;

  Particle(PVector center, float radius) {
    this.center = center.copy();
    this.lifespan = 255;
    this.radius = radius;
  }

  void run() {
    update();
    display();
  }

  // Method to update the particle's remaining lifetime
  void update() {
    lifespan -= 1;
  }

  // Method to display
  void display() {
    stroke(255, lifespan);
    fill(255, lifespan);
    circle(center.x, center.y, 2*radius);
  }

  // Is the particle still useful?
  // Check if the lifetime is over.
  boolean isDead() {
    // ...
    }
}
```

Note

An example on how to use particle systems is available at `https://processing.org/examples/simpleparticlesystem.html`.

Step 2 – Avoiding overlap of particles

As you observed in the previous step, generated particles can overlap with the existing particles, which would not look nice in your game. Thus, you will try to add particles that do not overlap with the already existing particles. For this purpose, check if the position is available—free from overlaps—before adding the particle. You should modify the `addParticle()` method as follows, and complete the missing parts. In addition, allow a particle to last forever, i.e. do not change the lifespan of a particle in the `update()` method. Run the code and see the result.

```
void addParticle() {
  PVector center;
  int numAttempts = 100;

  for(int i=0; i<numAttempts; i++) {
    // Pick a cylinder and its center.
    int index = int(random(particles.size()));
    center = particles.get(index).center.copy();

    // Try to add an adjacent cylinder.
    float angle = random(TWO_PI);
    center.x += sin(angle) * 2*particleRadius;
    center.y += cos(angle) * 2*particleRadius;
    if(checkPosition(center)) {
      particles.add(new Particle(center, particleRadius));
      break;
    }
  }
}

// Check if a position is available, i.e.
// - would not overlap with particles that are already created
//   (for each particle, call checkOverlap())
// - is inside the board boundaries
boolean checkPosition(PVector center) {
  // ...
}

// Check if a particle with center c1
// and another particle with center c2 overlap.
boolean checkOverlap(PVector c1, PVector c2) {
  // ...
}
```

Step 3 – Generating particles at specified time intervals

Note that in the previous part, a particle is added at every call to `run()` of `ParticleSystem`. Now, let's adjust the rate of adding particles: Create the particles with a fixed time interval of 0.1 seconds.

Note

For timing, you can use Processing's system variables `frameCount` and `frameRate`.

Part II

Integrate Particle System into Your Game

This part lets the user decide on a location at which the origin of a particle system will be set. After it has been set, a (cylinder) obstacle as well as a 3D model representing an evil mastermind will appear at that location. At regular time intervals, the villain will generate other obstacles around itself. When the ball hits an obstacle (cylinder), the ball bounces accordingly and the obstacle disappears. The goal of the game is to hit the origin of the particle system—the villain—in order to defeat it. This will make all of the obstacles disappear and will stop the villain from generating new obstacles.

In Moodle, we have shared a video called *Week6 Goal Video*. There, you find a demonstration from your game for this week.

Step 1 – Considering the cylinders as particles of a system

Create a `ParticleSystem` class in your game to manage the list of cylinder centers as its particles. Note that in this part, you will not need to keep track of the remaining life of particles since they will disappear upon bouncing only. Hence, managing only a list of centers is sufficient.

Modify the “adding-cylinders mode” in your game so that the user can set the origin of the particle system at the click location. When a new origin is set, remove all the cylinders and add a new cylinder at the origin.

Generate new cylinders repeatedly around the origin as in Part I. Set the time interval to 0.5 seconds. The cylinders should not overlap, and be generated within the boundaries of the board.

Step 2 – Removing a cylinder upon bounce

Modify the `run()` method of the `ParticleSystem` class so that it removes a cylinder if the ball hits it. If the cylinder at the origin is hit, remove all of the cylinders.

You can modify your bounce back method so that it returns true if the ball hits the cylinder. Then, you can use this method in `run()` of the `ParticleSystem`.

Step 3 – Adding a 3D model

In order to show the origin of the particle system in the game to the user, you will add a 3D model on top of the obstacle at the origin.

In Moodle, we have shared a 3D model file `robotnik.obj`, `robotnik.mtl` that is referenced while loading the OBJ file, and a texture file `robotnik.png`[1]. Download the files, load the geometry definition (OBJ) file, apply the texture, and display the loaded shape on the top of the cylinder at the origin. Note that you may need to `scale` and `rotate` accordingly before displaying.

[1]“Eggman L” by nintendoguy21 is licensed under CC BY 4.0

Part III

Bonus: Rotate the Villain to Face the Ball

As the villain anticipates the ball coming to hunt it, it continuously faces the ball and makes the game more realistic.