P02 PARTICLE FOUNTAIN

Posted on January 17, 2019 by dahl

Submit your completed assignment by 9:59PM on Wednesday, February 6th 2019. We will accept and grade submissions made through 9:59PM on Thursday, February 7th 2019. But no credit will be granted for any work that is submitted even one second after this hard deadline.

- **Pair Programming is allowed but not required for this assignment.** Register your partnership no later than Monday, February 4th to work with a partner. If you have problems accessing this form, try following the advice here.

This assignment involved developing a graphical implementation of a particle system. Particle systems have long been used in computer graphics to render amorphous objects like: fire, clouds, and water. The idea behind this technique is to use several small images or shapes that together give the appearance of a large amorphous object that does not necessarily have a clearly defined surface. Although not necessary for this assignment, interested students are welcome to read this early paper about the use of particle systems in early movie effects including Star Trek II: The Wrath of Khan: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.5878&rep=rep1&type=pdf

OBJECTIVES AND GRADING CRITERIA

The primary goal of this assignment is to gain experience working with data that is organized within several instantiable objects. Additionally, you'll get experience working with a real-time graphical simulation, and developing your own test methods.

20	Online Tests: these automated grading test results are visible upon uploading your submission. You are allowed multiple opportunities to correct the organization and functionality of your code (if necessary).
20 points	Offline Tests: these automated grading tests are run after the assignment's deadline has passed. They check for similar functionality and organizational correctness as the Online Tests. Since you will not have opportunities to make corrections after seeing the feedback from these tests, you should consider and test the correctness of your own code as thoroughly as possible.
10 points	Code Readability: human graders will review the commenting, style, and organization of your final submission. They will be checking whether it conforms to the requirements of the CS300 Course Style Guide. Since you will not have opportunities to make corrections after seeing the feedback from these graders, you should consider and review the readability of your own code as thoroughly as possible.

STEP 1. INTRODUCTION

Start by creating a new Java Project in eclipse and adding a new class called Fountain to this project. Ensure that this new project uses Java 8, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-1.8" within the new Java Project dialog box. The Fountain.java source file that this class is defined within will be the only file that you submit for grading through gradescope.com. If you do not already have a gradescope.com account, you should receive an email about setting this up on the evening of 1/30 (or you can use your wisc.edu email with this link to update your password: https://www.gradescope.com/reset_password). Please contact your instructor, if you do not receive such an email. You can make as many submissions as you would like prior to the deadline for this assignment, and the submission marked as "active" is the one that will be used for additional grading after the deadline.

Next download this P2ParticleFountain.jar file and copy it into your project folder. If this .jar file is not immediately appearing in the Project Explorer, try right-clicking your project folder and selecting "Refresh" to fix that. To make use of the code within this jar file, you'll need to right click on it within the Project Explorer and choose "Add to Build Path" from the "Build Path" menu. To ensure that this is working, call Utility.runApplication() from the main method within your Fountain class. When you run this program, a gray window should appear as a result along with an error message in the console which we'll resolve in the next steps: "ERROR: Could not find method named setup that can take arguments [] in class Fountain." This jar file that you are using makes use of the processing library, which you will get more direct experience with in future assignments.

STEP 2. UTILITY FRAMEWORK

At the end of the last step we saw an error related to the fact that our Fountain class was missing a method called setup(). Create a public static method with this name that takes no input arguments and has no return value. This should lead to a slightly different error message being displayed: "ERROR: Could not find method named update that can take arguments [] in class Fountain." We can now fix this error in a similar way, by creating a new public static method named update(), which also takes no input arguments and has no return value.

All of the methods within the provided P2ParticleFountain.jar file are documented here. If you look at the JavaDoc documentation for the Utility.runApplication() method, see how this method is making use of the setup and update methods that we have just created. These methods should not be called directly from your code. They should only be called by the Utility class, as a result of your calling Utility.runApplication(). You can also read through some of the other methods that are available through this Utility class. Note that many of these methods take float rather double values as arguments. Any time that you need to express a float constant in Java, you should use a lower case f to distinguish its type: 1.3 is a double, but 1.3f is a float.

We're going to be utilizing a lot of randomly generated values throughout this assignment, so let's create a private static Random field named randGen within the Fountain class. If you are not familiar with this class, you can read about the nextInt() and nextFloat() methods within the JavaDocs here. Initialize this field to a new Random object, and then pass a randomly generated int value to Utility.background() within the setup method. Now every time you run this program, the background will be cleared to a different random color. You do not need to use a specific seed when instantiating this random number generator, although doing so may be helpful debugging your code.

Next try drawing some circles using the Utility.circle() method. In order to change the color of the circles that are drawn, you can call Utility.fill(). Although randomly chosen colors can be fun, let's specific some particular colors that we'll continue to use going forward. We'll use the Utility.color() method to specify the mix of red, green, and blue light that make up the colors that we'll want to use here. We'll use Utility.color(235,213,186) for the background color, and we'll draw a circle on that background with the color Utility.color(23,141,235).

STEP 3. ANIMATION

If you haven't already, look through the JavaDoc documentation for the Particle class that we'll be using next. Create a private static Particle array field named particles within your Fountain class, and then within your Fountain's setup() method: initialize this field to a new Particle array containing a single reference to a single Particle object (we'll make this array bigger, with more particles later). Move your circle drawing code to the Fountain's update method (if it's not already there), and modify this code to make use of the size, color, transparency, and position of your particle object.

Next we'll try gradually changing the position of our particle over time, by doing so from within the Fountain's update() method that is being repeatedly called. Each particle has its own velocityX and velocityY state. Set the x and y velocity of your particle to -6 within Fountian.setup(). You'll notice that this doesn't automatically update the position of your particle. Instead, you'll need to add code to your Fountain.update() method which increments the x position by the x velocity, and increments the y position by the y velocity. This should result in positive velocities causing position values to increase, and negative velocities causing position values to decrease. Running your program at this point should result in a particle moving up and to the left (because both velocities are negative). If you can see more than one particle circle at a time, try moving the Utility.background call from setup() to the beginning of your update(), to fix this.

Next, we'll add a gravitational effect. Before moving a particle in our update method, we'll adjust the y velocity of that particle to by adding 0.3 to it. This should result in what looks like a ball being tossed, as this downward acceleration causes the particle to sweep out a parabola.

STEP 4. MORE PARTICLES

Ultimately, we'd like to have several particles being created, updated, and removed from our particle fountain through every update(). To prevent this method from getting too complex, we are going to begin refactoring the code within it. Create a new private static method named updateParticle() that takes an int index as input and has no return value. This method should do all of the moving, accelerating, and drawing of a particle that we have implemented so far, but should be able to do that with whatever particle is specified through the provided index. Your update method should be left clearing the background, and then looping through the the indexes of your particles array, and calling updateParticle() for each index that does not contain a null reference. To make sure that this is working, update the size of your particles array from 1 to 800, leaving only a non-null particle reference at index 0.

Next we'll create a method that attempts to add more new particles to this array. Create a new private static method called createNewParticles() that takes an int number of new particles to create as input and has no return value. This method should begin stepping through indexes of the particles array in ascending order. As it does so, it will be looking for null references within this array that can be changed to reference newly created particles. This loop will continue to do this until either 1) it has stepped through all valid indexes and there are no more to check, or 2) it has created the specified number of new particles and has stored references to them within the particles array. Each time your Fountain's update() method is called, it should call createNewParticles() to attempt to create 10 new particles.

We'll want to vary the initial state of these particles being created within our createNewParticles() method. But first let's add some private static fields to our Fountain class. Initialize these fields to the values shown within Fountain.setup().

```
private static int positionX; // middle of the screen (left to right): 400
private static int positionY; // middle of the screen (top to bottom): 300
private static int startColor; // blue: Utility.color(23,141,235)
private static int endColor; // lighter blue: Utility.color(23,200,255)
```

Use these fields (along with the Random field that we created in Step2) to randomize the initial state of each new particle that is created within createNewParticles() as follows. For int values: both the min and max bounds specified below should be considered inclusive (particles could have an age of 0 or 40). For float values: the min should be inclusive, but the max should be exclusive (the x position of a new particle could not be Fountain.positionX+3, but could be Fountain.positionX+2.999999).

- The x position of each particle should begin within 3 pixels (+/-) of Fountain.positionX.
- The y position of each particle should begin within 3 pixels (+/-) of Fountain.positionY.
- The size of each particle should begin between 4 and 11.
- The color of each particle should begin between Fountain.startColor and Fountain.endColor (note: Utility.lerpColor() may be helpful for this).
- The x velocity of each particle should begin between -1 and 1.
- The y velocity of each particle should begin between -5 and -10.
- The age of each particle should begin between 0 and 40.
- The transparency of each particle should begin between 32 and 127.

Note that when you run this program, you should see a flurry of particles resembling a fountain. Once the 800 spaces within your array are used up, no more particles can be created or seen. In the next step, we'll remove some of the old particle references so that their positions can be reused by newly created particles. If you haven't already, feel free to remove that funny particle that we were creating in setup(), so that all of the particles that we see are created by our createNewParticles() method.

STEP 5. REMOVING AND MOVING PARTICLES

You may have noticed that we set each particle to have a randomized age between 0 and 40. A particles age should increase by one every time the updateParticle() method is called on it's index. Implement this now. Then create a new private static method named removeOldParticles() that takes a max age int as input, and has no return value. This method should be called from the end of your Fountain.update() method's definition, with a max age of 80. The job of this new method is to search through the particles array for references to particles with an age that is greater than the specified max age. The references to any such particles should be replaced with null references in your particles array by this method. Implementing this correctly should allow your particle fountain to continue running forever.

If you try clicking on the screen, you may notice an error message related to a mousePressed method that cannot be found. Let's create this public static method now, it must take two int arguments representing the x and y position of the mouse when it is pressed, and has no return value. You should implement this method to move the Fountain.positionX and Fountain.positionY to match the position of the mouse whenever the mouse button is pressed. Clicking on the screen will now allow you to move the entire fountain around which new particles are created. You can also drag it around like a sparkler.

And last but not least, we'll create a public static method called keyPressed that takes a single char argument representing the key that was pressed, and has no return value. Within this method, we'll call Utility.save("screenshot.png") whenever the key pressed happened to be the 'p' key. This is another callback method that will be called at the appropriate time by the Utility class, and so should never be called from your Fountain class.

STEP 6. TESTING AND REFLECTION

Let's now write a couple of tests to confirm the behavior of a couple methods under very specific circumstances. Here are the two test methods that you must implement within your Fountain class. Since running many of your methods require first calling Utility.runApplication(), you can call these test methods from the beginning of your setup method before any of the static fields are initialized. This will help ensure that your tests don't interfere with the subsequent execution of your program. Note that the methods below already have JavaDoc style comments, like the ones that you are required to add to all of your other methods by the course style guide.

```
/**
 * Creates a single particle at position (3,3) with velocity (-1,-2). Then checks whether calling
 * updateParticle() on this particle's index correctly results in changing its position to
 * (2,1.3).
 *
 * @return true when no defect is found, and false otherwise
 */
private static boolean testUpdateParticle() {
  return false; // TODO: implement this test
}

/**
 * Calls removeOldParticles(6) on an array with three particles (two of which have ages over six
```

```
 * and another that does not). Then checks whether the old particles were removed and the young
 * particle was left alone.
 *
 * @return true when no defect is found, and false otherwise
 */
private static boolean testRemoveOldParticles() {
  return false; // TODO: implement this test
}
```

If either of these tests fail, a message containing the word "FAILED" should be printed to System.out from your setup() method. When both tests pass, nothing should be printed. In either case, the rest of your program should execute as normal after these tests have been run.

As you reach the end of this program, consider how objects have been used to group the data of each particle. This has made it relatively easy for us to create and manage hundreds of complex particles. However, our code really only supports a single fountain. In the future, as we begin to define our own object types, our default will become organizing our code into instantiable objects (like these particles) rather than static class fields and methods (like this fountain).

SUBMISSION

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the course style guide, you should submit your final work through gradescope.com. Your score for this assignment will be based on your "active" submission made prior to the hard deadline of **9:59PM on Thursday, February 7th**.

EXTRA CHALLENGES

Here are some suggestions for interesting ways to extend this simulation, after you have completed, backed up, and submitted the graded portion of this assignment. **No extra credit will be awarded for implementing these features**, but they should provide you with some valuable practice and experience.

- Try moving the Utility.background() call from Fountain.update() back to Fountain.setup(), so that the old particles are not cleared. You could also change the color of your particles when various keys are pressed by the user. The results of these changes is a fun painting program.
- Try changing the initial state of these particles to look like different phenomena: snow, fire, clouds, fireworks, etc. can all be fun challenges to create graphically with a particle system like this.
- Another fun variation is to have your particles interact with the user. Maybe instead of moving the fountain when the mouse button is pressed, you move a position that either attracts or repels particles. This movement could be in addition to the gravity that we have simulated so far, or instead of it.