DoodleJump

Help Session: Thursday, October 29, 4:00 - 6:00pm in Barus & Holley 166

Design Checks: Sunday October 25 - Wednesday October 28

Early Due Date: Wednesday, November 4, 11:59pm
On Time Due Date: Friday, November 6, 10:00pm
Late Due Date: Sunday, November 8, 10:00pm

Run demo: cs015_runDemo DoodleJump
Run snazzy demo: cs015_runSnazzyDemo DoodleJump
Keep in mind that the demos might not work well over ssh!

To install: cs015_install DoodleJump

Silly Premise:

After seeing the amazing feats of JavaFX that were Gabriella and Troy’s Cartoon projects, Sharpay and Ryan are feeling competitive. As a result, Sharpay and Ryan are particularly enthusiastic about Bopping to the Top. Help Troy and Gabriella prevail by preventing Sharpay and Ryan from ever Bopping to the Top by creating an infinitely scrolling board!

♫♫♫
Yeah we’re gonna
Bop bop bop, bop to the top
Wipe away your NullPointerExceptions
Bug bug bug, do the debug
And strut your Eclipse skills
Bop bop bop, don’t let Doodle drop
We’re going for the calves of Andy
We’ll keep coding and we just won’t stop
Till the SunLab locks!
Bop to the top!
♫♫♫
New Concepts Covered:
- For-loops
- java.util.ArrayList
- Java Generics
- Keyboard Interaction
- More JavaFX!

Assignment Specifications:

Before you read this section, make sure you play around with the demo! Better yet, have it open while you read this and play along. Doing so will make understanding the assignment much easier.

In this program, you will write the CS15 version of the popular game Doodle Jump. This game features a “doodle” whose goal is to hop along a series of never-ending platforms without falling to the ground. The doodle will fall according to the laws of gravity (more on this later) until it hits a platform, at which point it will bounce up until gravity brings it down again. The doodle’s horizontal position can be controlled with the left and right arrow keys.

As the doodle jumps higher and higher, the game will need to scroll vertically. By scroll vertically, we mean that when the doodle hits a certain height (say the midway point of the window), rather than have the doodle move higher on the screen, all the platforms should move lower (giving the illusion that the doodle is still climbing up). See the demo for an example.

As the game scrolls upward, old platforms need to disappear off the bottom of the screen and new platforms need to be regenerated at the top of the screen so that the doodle will have more places to jump.

The game ends when the doodle falls to the bottom of the screen because it missed all the platforms as it fell. A message should be displayed to tell the user that the game is over.

Helpful Design Hints:

This program is large and complex, so it is more important than ever to design all your classes before you write any code. We mean it!

Well-chosen relationships between your classes will be very important. You have some experience with making a GUI from Cartoon, so remember what you have learned when creating the GUI components. We have provided you with an App class that takes care of the mainline, but it is your job to do a layout design using javafx.scene.layout as well as any other graphical components you need. Once you have an idea of how you want your GUI to look, you should think about the communication between the GUI, the Doodle, and the
platforms. Furthermore, you have some experience with Timelines in Java from Cartoon, which will come in handy for animation and physics simulation.

Another important design decision will be choosing how to generate your platforms. The game would be pretty boring if the same platforms were always generated, so you’ll want to incorporate some randomness into your platform generation. At the same time, your platforms should be generated in such a way that the doodle will be able to get from one platform to the next, so making platform generation completely random won’t work either. Your platforms should be generated “semi-randomly,” meaning that their location should be chosen randomly given a certain set of constraints.

So how should you keep track of your platforms? By using a java.util.ArrayList, that’s how! An ArrayList is a data structure that allows you to store and access a dynamic number of objects in a “list” of sorts, which is exactly what we’re looking for: a way to keep track of all the platforms in our game.

But wait! Why can’t we use an array for this? Since the horizontal and vertical positioning of platforms is based on a degree of randomness, varying numbers of platforms may fit on the screen at any one time. The java.util.ArrayList’s dynamic capacity lends itself very well to handling this type of dynamic storage, whereas an array of fixed capacity would not. ArrayLists also allow you to easily add and remove elements.

The Javadocs for the java.util.ArrayList data structure highlight some of the important methods that have already been defined for ArrayLists, so you should become familiar with these before starting this project.

For a review of ArrayLists, look over the later slides in the Arrays lecture. This will also help you understand how to declare and instantiate an ArrayList that holds a unique type of object.

**Physics Simulation:**

To model the effects of gravity on the doodle, we’ll need to give it some physical properties, such as position and velocity. We will need to modify these properties over time, using a couple of the handy Equations of Motion you probably learned in high school physics.
These equations may be intimidating, but what they describe is actually pretty straightforward.

We need to calculate a new y-velocity (the rate at which the Doodle is falling), which helps us calculate the new y-position for our Doodle at the end of each KeyFrame. This repeated updating simulates the effects of gravity on a falling Doodle; each successive update will need to build on the velocity and y-position that was computed in the previous update.

The duration of your KeyFrame represents $\Delta t$ in the equation above. Using the duration and acceleration (gravity), you can calculate the new velocity and position of the doodle. Here’s pseudocode for updating the velocity and position of the Doodle each time the Timeline cycles through to the end of your KeyFrame:

\[
\text{updated velocity} = \text{current velocity} + \text{ACCELERATION} \times \text{DURATION} \\
\text{updated position} = \text{current position} + \text{updated velocity} \times \text{DURATION}
\]

**Note:** We’ve converted the the original equations to discrete (non-continuous) equations because the approximation is close enough for our purposes.

The constants \text{ACCELERATION} and \text{DURATION} are already defined for you in the \text{Constants} class, though feel free to change them. Be sure to review the Math/Making Decisions lecture for more information on constants!
ACCELERATION corresponds to the gravity constant. In the real world, it's $-9.8 \text{ m/s}^2$. Because our coordinate system defines the positive y-direction as downward, our acceleration constant is positive (approximately 1000 pixels/s$^2$). The DURATION constant corresponds to the amount of time that elapses for each iteration of your KeyFrame and should be used to initialize your KeyFrame, as well as for updating position and velocity.

Be careful of the units, though—DURATION is in milliseconds! You'll want to use units of seconds in your equations for calculating the doodle’s new velocity and position.

Remember: When you're converting the DURATION to seconds you should compute $((\text{double}) \ DURATION)/1000 \text{ or } DURATION/1000.0$ instead of $DURATION/1000$ to avoid integer division truncation (see the Math/Making Decisions lecture for more information).

Once you have this set up, making your Doodle jump is more straightforward! You can detect collisions between your Doodle and a platform using javafx.scene.Node intersects method, described here:

```java
boolean intersects(double x, double y, double width, double height) {
    Consider a rectangle at (x, y) with given width and height
    If the node intersects the rectangle:
        return true
    Otherwise:
        return false
}
```

To make your doodle jump, set its y-velocity to the REBOUND VELOCITY constant whenever it hits a platform. Be careful to make sure that you're only checking for platform collisions when the doodle is falling down.

Timeline Updates:

By this point it should be clear that your Timeline is going to have to do a bit more than it did in Cartoon. To make your lives easier, we have provided for you below a list of steps that need to be taken at the end of your KeyFrame. Remember that your EventHandler doesn’t need to do all of this directly. It can (and should) delegate appropriate tasks to other methods and objects. Inside your handle(...) method (or it's helper methods!), you should:

1. Calculate a new y-velocity for the doodle
2. Calculate the change in your doodle’s height ($\Delta h$)
3. Check to see if the doodle is above the horizontal middle of the window
4. If (the doodle is above the middle):
   move all platforms down by $\Delta h$
else:

move the doodle up by $\Delta h$

5. If (the doodle intersects with a platform AND doodle is falling):
   set the doodle's y-velocity to REBOUND_VELOCITY.

Keyboard Interaction:

You may be wondering how you will make your doodle move left and right and respond to user input. The answer is keyboard interaction! There are a few ways to do keyboard interaction in Java, though we suggest the method below. (Feel free to explore Javadoc for other ways to go about it, as long as they work well and are readable).

The method we recommend makes use of the interface javafx.event.EventHandler. You need to define a private inner class which implements this interface and decide what you want the handle(Event e) method to do.

Keep in mind that you do not need separate EventHandler for moving your Doodle left and right: you can have EventHandlers take in parameters!

To get information on which key the player pressed, first your handle(Event) method should take in javafx.scene.input.KeyEvent. Then, you need to call method getCode() on the KeyEvent which will report which keyboard key (KeyCode) has been pressed. Read more about KeyEvent here (jump to Keyboard Input).

For a complete list on the types of KeyCodes, you can check the Javadoc. Once you know what keyboard key was pressed, you can make a decision (hint, hint… lecture slides) on what you want to do when a specific key is pressed.

Let's define an EventHandler to move a ball in space. We'll write it as a private inner class in the Game class:

```java
private class MoveHandler implements EventHandler<KeyEvent> {

    @Override
    public void handle(KeyEvent e) {
        KeyCode keyPressed = e.getCode();

        if (keyPressed == KeyCode.UP) {
            // Code to make the ball move up.
        } else if (keyPressed == KeyCode.DOWN) {
            // Code to make the ball move down.
        }

        e.consume(); // See explanation below.
    }

```
Remember to associate your EventHandler with the relevant Pane (hint:
setOnKeyPressed(keyEvent, handler)) or nothing will happen when you press the arrow keys!

Consuming Events

Your KeyHandler will be expected to consume the KeyEvent at the end of its handle method. In JavaFX, there may be multiple classes responding to KeyEvents (including classes that you didn’t write). A notable example of this is the Button class—the KeyEvents that you are using to move your Doodle also are used by the Button class to select different buttons. Consuming the event prevents this from happening and ensures that there are no unintended side effects of using the arrow keys to control your Doodle.

Platform Generation:

Begin by creating the bottommost platform so it is just below your doodle’s start position. Next, you’ll want to generate the platform above that one. You should decide on minimum and maximum x and y distances between two consecutive platforms so you can be sure that the Doodle is able to jump from one to the other.

Here’s an example of how to use this information to generate the platform’s location:
Assume: Platform p has x-y coordinates (300, 600)
Assume: Doodle can move up 100 pixels and left or right by 200 pixels per jump before starting to fall. (This may differ based on the constants you use, make sure your game is playable)

With the above assumptions, the maximum x-distance between consecutive platforms should be 200 and the maximum y-distance should be 100. This means the second platform’s x-position must be a random value between 100 and 500 and its y-position should be a random value between 500 and 600.

Note: Platforms should not generate outside the board. Consider the panel’s width when constraining your generation to stay within the board.

The third platform will need to be positioned relative to the second platform, and so on. Do you see a pattern?

How should you generate all the platforms? A loop! At each iteration of the loop, you are creating a new platform whose position is relative to the platform created in the previous iteration of the loop.
Here’s an idea of how you might want to implement this:

1. Create your first platform and store it as the current topmost platform. Position it directly under the Doodle’s starting location (probably somewhere near the bottom of the Pane).
2. Add that platform to your ArrayList of platforms.
3. While the current topmost platform’s y position is not above the top of the Pane:
   a. Create a new platform and add it to the ArrayList.
   b. Position that platform relative to the current topmost platform
   c. Update the topmost platform to reference this new platform

Remember: To graphically (visually) add or remove Nodes, you should add or remove them from the Scene Graph; to add or remove Nodes in your game logic, you should make sure the appropriate references to the Node (or the object that contains the Node) exist or don’t exist (in the case of removal).

Code this project incrementally!

Figuring out where to begin on this project can be a bit overwhelming. It will make your life a lot easier to get the program working a little bit at a time. Please don’t try to write a ton of code and then run it! You’ll just cause more headaches for yourself later. **TAs will turn you away from hours if you have not demonstrated that you have carefully attempted each part of the assignment before moving on to the next step.** We suggest following these steps:

1. Make a window show up.
2. Get your doodle to display.
3. Set up the Timeline that will be in charge of updating the doodle’s location and displaying the graphical changes. (To test if your Timeline is working, you can start with printlines. Afterwards, you may want to move the doodle slightly at the end of your KeyFrame.)
4. Set up the EventHandler so you can use the left and right arrows to get your doodle to move.
5. Add some physics simulation so that your doodle falls.
6. Start with creating one platform and add collision detection making sure it’s working.
7. Generate a whole screen of semi-randomly positioned platforms so that your doodle can jump its way upwards!
8. Add the vertical scrolling so that when the doodle tries to pass a certain height, it stops moving, and all the platforms move downward. (**WARNING!** This step is tricky! Really think about the best way to implement this -- careful design will simplify this step greatly.)
9. As platforms scroll down, be sure to delete them and generate new ones both graphically and logically.

**Minimum Requirements:**
There's plenty of room for creativity in this assignment, but **before you do any extra credit, make sure your game meets all of the following specifications!** (Reminder: you will not receive any extra points if you fail to meet a single minimum requirement.)

1. It has a **Doodle**! (duh...) The **Doodle** should be its own class - this makes it easier for your doodle to store and calculate relevant information. Remember: delegation!

2. You have a **Timeline**. At the end of your **KeyFrame**, the doodle’s velocity is calculated. The doodle should be constantly accelerating downwards.

3. The doodle can be moved left and right with the left and right arrow keys.

4. You have platforms (instances of a `javafx.scene.shape.Shape`, or you can write your own **Platform** class containing a **Shape**). The position of each new platform is semi-random. It should be constrained within a certain distance of the platform below it so it is reachable.

5. When the doodle is falling and intersects a platform, it bounces.

6. The game implements vertical scrolling when the doodle reaches a certain height on the screen.

7. As the doodle jumps higher and the game scrolls, new platforms are created so that the doodle can continue to climb indefinitely.

8. If the doodle falls to the bottom of the screen, a game over label appears and the doodle can no longer be moved with the arrow keys.

9. The game should display a quit button that calls `System.exit(0)` or `Platform.exit()` when clicked.

**Extra Credit:**
Once you’ve met all of the above requirements, feel free to snazz up your DoodleJump with some bells and whistles! (check out the snazzy demo for inspiration). Here are a few ideas:

- Different levels of difficulty
- A button to restart the game
- Make the Doodle a composite shape
- Keeping track of the player’s score
- Support for multiple players
- Different types of platforms - movable, breakable, high-jump
- Toys to alter the Doodle’s velocity - jetpacks, helicopter-hats, etc.
- Venomous black holes and hungry monsters

Keep in mind that most TA’s have not implemented every one of these extra credit features, so we cannot guarantee support for them (though they will try send you in the right direction).

**Installing Doodle Jump:**

Type `cs015_install DoodleJump` in a shell to install Doodle Jump support code in your course directory.
Handin Info:

Type `cs015_handin DoodleJump` in a shell from your course directory to hand in the assignment.

Some Tips:

- This project is longer than any you’ve seen in CS15 thus far. Start early, start now, start yesterday!
- Although DoodleJump is complex, careful design will simplify the project greatly. Try to design the entire program before you start to code.
- When designing, think about how you are going to implement your data structure.
- A thorough understanding of the lecture slides and this handout is critical for a successful DoodleJump. Read both carefully before beginning.
- Your code will be longer and more complex than that of any of your previous projects, and thus style will be very important to your grade! Please consult the CS15 Style guide: [http://cs.brown.edu/courses/cs015/docs/styleGuide.pdf](http://cs.brown.edu/courses/cs015/docs/styleGuide.pdf)

Stencil Code:

The only classes we’ve provided you with are an `App` class (remember that your code should begin in the `App`’s constructor) and a `Constants` class with some recommended numbers. These numbers are optional—feel free to change them, remove them, or add new constants as you see fit.