GO :: FA22 :: A3 :: RainMaker Specification [V001]

A3

Assignment 3 :: The Final Version of RainMaker

Introduction

This term we will be studying object-oriented graphics programming and design. Simple video games provide a good platform for understanding these concepts and you will be developing successively more complex variants of a simple 2D top-down video game over the course of the term.

This term our game is called **RainMaker.** RainMaker is set in the near future where endless drought is wreaking havoc in the central valley. Ponds and lakes are drying up and birdlife has virtually disappeared. With your specially equipped helicopter you will roam the skies of the central valley seeking clouds to seed using your custom designed *rainmaker* helicopter attachment. The rainmaker has several modes of operation and can either seed clouds from above or shoot at them from the sides. The seed from above approach is less effective, however, it does not harm birdlife. You will have to carefully choose your cloud seeding strategies in order to prevent harm to birdlife.

As you seed clouds, their color changes as they become more ready to bring more rain to the central valley. In this first version of the game, we will keep things simple and have just a single level with just two clouds to seed, a single pond, and only the top-down cloud seeding apparatus, and no birds or fuel concerns. The level is completed when the pond level returns to an acceptable level and the helicopter safely lands on the helipad. Future versions of the game we will ramp up the complexity with the need for mid-air refueling via fueling blimps, returning flocks of birds that increase the challenge of seeding clouds, wind that keeps the clouds moving, and more complex helicopter mechanics. As the game increases in complexity over the term you will have to avoid other objects and manage your fuel and other resources carefully in order to win the game.

The goal of this first assignment is to develop a simple version of the game and to develop the basic OOP framework using the JavaFX API. You will be building on this for the remainder of the semester and refactoring your code continuously. To that end, we are not hyper concerned about getting everything right the first time around. It's far more important that you get it working and then work on improving your working solution by continuously refactoring to cleaner code as we progress through the term.

For this version we will use the keyboard to control a single helicopter in a simplified graphical display that uses the entire screen. In later versions we will add additional screen components and controls as we learn how to build more complex GUI interfaces.

High Level Program Structure

We're going to start out with a basic structure that we will build on over the course of the remainder of the term. Some of this structure will remain in place throughout the project, other parts of it you will change and adapt. Don't be afraid of this, refactoring is an important part of good coding.

Pay close attention to the structure description below. If class names are specified specifically in this document, then you are required to use those exact class names in your project. We may refactor this structure in future assignments, do not be afraid of change.

Class GameApp

At the highest level we have the class **GameApp**. This class extends the JavaFX Application class. The purpose of this class is to manage the high-level aspects of our application and setup and show the initial Scene for your application. The GameApp class sets up all keyboard event handlers to invoke public methods in Game.

Class Game

For this first version all game logic and object construction belong in the **Game** class. All of the rules in our game are implemented in the **Game**. This class holds the state of the game and determines win/lose conditions and instantiates and links the other *Game Objects.* The **Game** does not know anything about where user input comes from or how it is generated. The Game class extends the JavaFX class Pane. This allows the Game class to be the container for all game objects. For this version of the game we will not have a separate game object collection. This may change in a future revision.

At this stage we are not overly concerned that we are purely and properly implementing any particular application pattern, e.g., MVC. We do, however, want to start thinking about separation of concerns.

The interaction of these classes is discussed further later in the document.

New for A2

In A2, we want to improve the looks of our game. You will use an Image based background for A2 to create the background. Note that we will adapt this further in A3 so you should create a class for the background. You may or may not want to extend GameObject for this. You may prefer, for example, to directly extend Pane. There advantages to both methods, you may have to refactor later depending on how the game evolves.

Either create a tan/brownish map that represents a parched small region of the central valley, or find an image online that you can use.

Game Object Classes

In addition to the classes described above you will have some additional classes that represent game objects. In this version of this project, you will build a simple hierarchy of game objects. Because we want to inherit the properties of JavaFX Node objects, our game object class will extend the JavaFX **Group** class. This alleviates us from having to setup a number of different properties that each object needs, for example, the object's location in the world.

Later in this document I will discuss the basics of object behaviors and private data, but for now, let's jump into the various classes that will represent the game objects.

Class GameObject

The abstract **GameObject** class is the base of our object hierarchy. It contains methods and fields that manage the common aspects of all game objects in our program. Any state or behavior in this class should apply to all game object this. For example, the helicopter can move, while a pond cannot. Consequently, you would not include anything regarding movement in this class.

Class Pond

This class represents a pond or lake in the Central Valley. For this first version of the project, we will abstract the pond as a simple blue circle placed at random such that it does not intersect any other ground based object.

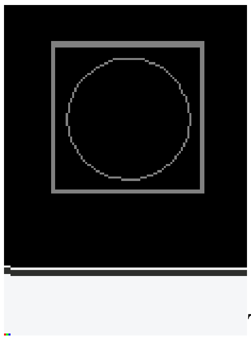
Class Cloud

This class represents a cloud in the skies of the Central Valley. For this first version of the project, we will abstract the cloud as a simple, initially white, circle placed at random anywhere other than fully directly over the helipad.

Class Helipad

A1-Spec

This class represents the starting and endling location of this first game. The helicopter will take off from the helipad and after seeding all of the clouds will have to land back on the helipad in order to end the game. For this game there is no actual notion of altitude. A helicopter is landed on the helipad whenever it is contained within the bounds of the helipad and not moving.



The helipad is represented on screen by a gray square with a gray circle centered within the square. There should be a gap between the circle's edge and the square edge. The exact relationship is not strict and you may adjust for taste. However, the circle must be centered and there must be a clear and visible gap between the circle and the square. The helipad should be centered along the width of the screen and should be roughly one half of its width above the bottom edge of the screen. Feel free to adjust slightly to make sure that your Helicopter fuel readout is clearly visible on startup.

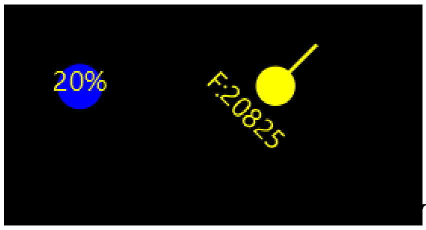
A2-Spec

The helipad in A2 can be built up from an image if you wish, however, we may adapt this in future versions.

Class Helicopter

The helicopter class is evolving with the specification. I have maintained the A1 spec so that you can focus on the changes in the A2 specification.

A1-Spec



This class is the most complex game object and represents the main *player character* of this version of the game. The helicopter is represented as a small filled yellow circle with a line emanating from the center of the circle pointing in the direction of the helicopter's heading.

In this project the heading of an object is the compass heading specified in degrees. Note: When the helicopter is initially place on the map it is placed facing a heading of zero degrees, or, due north, and a speed of zero. There are some complications relating to compass heading that are discussed later in this document. As the heading changes the line must rotate to point in the direction of the new heading. As with the helipad, you should derive the size of the helicopter object from the dimensions of the screen. More on this later. As long as it looks reasonably similar to the drawing and behaves as described herein, you will be fine.

Below the helicopter you must display the current fuel, as shown above. These move with the helicopter but remain in the same position relative to the helicopter body. The Helicopter is initially centered on the Helipad so it is a good idea to pass the necessary coordinates into the Helicopter's constructor. You must pass in the center of the helipad. You may want to think about the order in which you create the objects to avoid issues here. Note, you cannot derive the coordinates of the helipad based on its placement rules for this game. In other words, if the location of the helipad changes, no changes to the helicopter code should have to be made. You are allowed to compute any adjustments necessary to center the helicopter on the helipad based on the center of the helipad that is passed in. If this doesn't make sense at the moment, then just go ahead and use whatever you think will work and you should be able to immediately see what needs to be done. This leads us to some additional advice that you would do well to heed throughout this course:

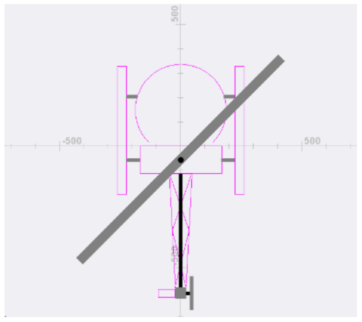
Don't be afraid to experiment and try things out. More importantly, give yourself enough time for this experimentation as it is an important part of the discovery and learning process.

The fuel properties of the helicopter are integer values. The initial fuel value is set for playability at 25000 and must be specified in the Game class. You should adjust the rest of your constants to make your game playable in a similar timeframe as the demo. The helicopter also has a speed that is initially set to zero. The helicopter speed increases and decreases with brake and acceleration commands, but, for now, there is a maximum speed of 10 and a minimum speed of -2. The helicopter will ignore requests to go faster than the maximum speed or slower than the minimum. Note that negative speeds fly the helicopter backwards and that the speed transitions smoothly from forward to reverse direction.

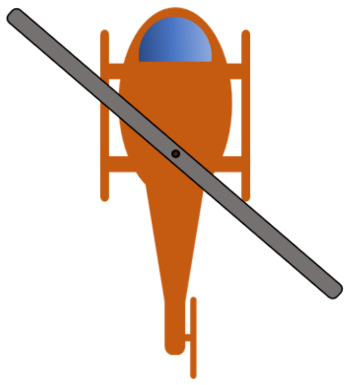
We will discuss all of these objects further in a later section on game behavior. For now, let's take a look at all of the commands that we need to play this first version of the game.

A2-Spec

For A2 we will make our helicopter look like a real helicopter. For this version you may continue to show the fuel beneath the helicopter as we did in A1, however this will go away in the A3 specification. In the A2 spec your helicopter must both look and behave more like a real helicopter. I have demonstrated in class how you can construct your helicopter from graphics primitives as shown in the first image below, or from image primitives as shown in the second picture. You may go with either approach. If you choose the first approach, you should use filled graphics and may choose to use distinct border shapes/colors to enhance the look of the helicopter against the graphics background.

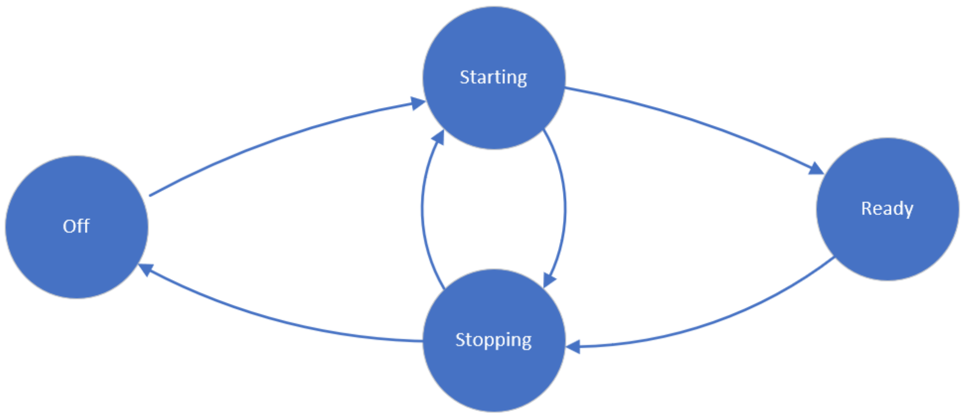


No matter which choice you use, your helicopter should now be comprised of a HeloBody class and a HeloBlade class. In A3 we will add additional components to our helicopter for more advanced cloud seeding.



There is a significant change to the helicopter in this game. In this version, in addition to having the engine started before the helicopter is able to move, the blade must now come up to speed. This means that the helicopter has four states: Off, Starting, Ready, and Stopping. You will implement this behavior as a **State Pattern.** Your helicopter class then becomes the State Context and you will delegate state dependent calls to the appropriate concrete state classes.

Initially the helicopter is in the Off state. When started the helicopter moves into the Starting state. When the engine is up to speed the Helicopter moves into the Ready state. When stopped, the helicopter moves into the Stopping state. Once the engine has stopped, the helicopter moves into the Off state. As far as changing the win/loss state of the game. The game is not won until the helicopter is parked on the helipad and is in the Off state.



The state actions for the Helicopter are somewhat complex. It's important to delegate any actions that are state dependent to the Helicopter's state objects. For example, as is shown in the table below, the helicopter can only seed clouds while in the Ready state. Hence, your seedClouds() method will simply call a method, mostly likely with the same name, within your state hierarchy. In other words, you must implement seedClouds() in your abstract state class and then override it as necessary in your concrete state classes. This way, the Helicopter will not be able to seed clouds unless it is in the Ready state.

State	**Actions**
Off	Helicopter can have engine started Helicopter ***does not*** consume fuel
Starting	Helicopter waits until blade is up to speed for Ready state. Helicopter consumes some fuel while rotating up.

	Helicopter can go into Stopping state at any time.
Stopping	Helicopter waits until blade has fully stopped before entering Off state Helicopter ***does not*** consume any fuel. Helicopter can go into Starting state at any time.
Ready	Helicopter can Move Helicopter can seed clouds

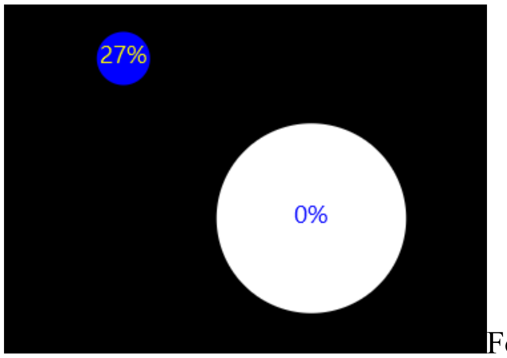
The rotational speed is tracked in a field *rotationalSpeed* of the object that represents the helicopter blade. This value is the degrees of rotation of the blade that is updated constantly while the application is running. This should have a minimum value of zero and a maximum value that is taste dependent. It should take at least a few seconds for the rotor to get up to speed and so you will need to adjust the update amount per call to the blade's update method.

The helicopter blade should be self-animating. That is, it doesn't need to have an update() method nor implement Updatable. The blade's rotation is completely dependent on the helicopter's state. There is no other interaction between the helicopter blade and the rest of the game. Consequently, it does not make sense to force a dependency. The one thing that you should think about here is that the blade's rotation speed should be independent of the frame rate. In other words, on slower machines, the blade should rotate further on each call to its own animation timer. There are several ways to solve this. I suggest that you get the basics working first.

Class Pond and Class Cloud

A1-Spec

These classes represent a single pond or cloud on the abstract map of the central valley. The Pond is represented on the screen by a blue circle and the cloud is represented by a white circle. Both display a percentage text in the center.



For the pond the percentage is of the normal radius of the pond, for the cloud it is the percentage of saturation. For this version of the game, you will only one of each place at random in the upper two thirds of the screen. Each of your objects must have some slight random variation in both size and position when placed so that each time you play the game both the position and size are slightly different. It is your job to manage this variation so that the game is fun and playable. This will be a recurring theme throughout this course and it should be among your first lessons in the following idea:

Just because the code works does not mean that the project is complete.

As long as the pond's percentage is less than 100% you must make it rain by seeding the cloud. You seed the cloud by flying over the cloud and pressing the space bar rapidly, or, holding the space bar down. Every press of the space bar increases the saturation by 1%. As the cloud becomes more saturated, the color turns to gray. Simply decrease all RGB values by the amount of saturation. A fully saturated cloud would thus have the color rgb(155,155,155). When the saturation reaches 30% the rainfall will start to fill the pond at a rate proportional to the cloud's saturation. The rainfall must increase the area, not the radius. In future versions of the game, we will adapt this further. The cloud will automatically lose saturation when it's not being seeded at a rate that allows the percentage to drop about 1%/second. For this version of the game, you will just need to adjust this experimentally. In future versions we will make this depend on frame rate so that we can be consistent.

A2-Spec

For A2 we will increase the number of clouds and ponds. There should be three ponds distributed around the game and three to five clouds present on the screen at any one time, details below. In addition, while the Ponds will remain fixed, the clouds will move in a fixed direction and speed defined by the game. For this version I suggest simply using static values in Game that define the WIND_SPEED and WIND_DIRECTION. The speed is unitless and the direction is specified in degrees. For this version of the game the direction will always be due east. This way clouds are blowing in from right to left. In reality this means that you technically don't have to deal with wind direction. Once the Y value for a cloud is set, it remains constant and only the X value changes as the cloud moves across the screen. Although wind direction is nice and real, it doesn't improve playability. In fact, it harms it because you're not guaranteed to eventually get a cloud close enough to a pond. In fact, we need to guarantee that. So let's choose the Y coordinate randomly, but, such that it is within functional distance of at least one pond. I suggest rotating through the ponds such that the first new cloud is within Y-delta of the first pond, then the next new cloud is within Y-delta of the second pond, and so on. This will guarantee that you can eventually fill all of the ponds. Strictly speaking this isn't necessary for a win, but it should be a better strategy for cloud introduction and somewhat easier than trying to follow a wind direction.

While for now you can keep the wind speed constant, I recommend giving each cloud its own random speed offset so that they all move at slightly different speeds.

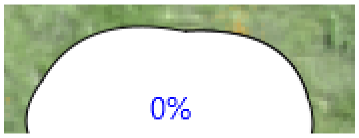
This means that your clouds are constantly moving and that their distance to the ponds is constantly changing. Consequently, you will have to compute the distance from each cloud to each pond. For this version of the game, you may optionally implement the 'd' key command to enable distance lines between each cloud and each pond. This is simply a single pixel white line from the center of the cloud to the center of the pond that shows the distance of the line next to the line.

When a cloud leaves the screen, you should decide at random whether or not to introduce a new cloud. When there are one or two clouds remaining, you will always introduce a new cloud. When there are five clouds on screen you will not introduce a new cloud. When there are three or four clouds on the screen you will flip a coin to determine whether a new cloud appears on the screen. The clouds will always appear off screen at a location randomly place along the normal to the wind heading.

The maximum distance for which a cloud can impact a pond is four times the diameter of the pond. This means that the impact of a cloud on a pond is proportional to the distance to the pond in ratio with this maximum distance. The seeding of the cloud will impact the ponds in the same way as A1, except, that the amount of increase is multiplied by this proportion. Distance is always measured center to center. So, for a pond whose center is half of the maximum distance from the cloud center, it will increase at 50% of the rate of a pond that is directly under the cloud.

Note that you will have to implement seeding only on the cloud that the helicopter is currently intersecting. I'll leave the detail of how to do this up to you. If you have questions, ask in the discussion forum.

A3-Spec



For the A3 release we will spice up the look of the clouds and ponds using Bezier curves. First replace the circle with an ellipse. For this ellipse you will want to randomize the lengths of the major and minor axis. Note that your clouds will tend to look better if the axis that is in the direction of the wind is always longer of the same length as the axis that is cross wind. This isn't required, but it's not that hard to implement. Now that you have an ellipse you will want to compute points along the perimeter of the ellipse to serve as the start and stop points for Bezier curves. In addition, you will want to compute points that lie outside of the ellipse and between the start and end points as control points. While you can get very precise computing the points on the ellipse, it's not necessary to do so. The points a sin(theta) and b cos(theta) where a, b are the radius lengths in each direction will give you a point on the ellipse. Increasing theta by some amount will give you points around the ellipse. Similarly, the control points can be generated by simply adding a constant amount to both a and b and using the new larger radii to compute points along an ellipse that has a perimeter outside of the base ellipse.

Draw quadratic curves using each of these points and control points. If you randomize the increment to theta, then you will have a variety of interesting cloud shapes. If you make the increment too large then you will end up with something that looks just like an ellipse. I suggest experimenting with this to determine the best look for your game. To do this, you will want to draw small circles at each point using a different color for start/end points and control points. Another helpful tip is to make the stroke color black so that it stands out against the cloud. Later you can soften it to a grey, or, just leave it black. If you overshoot 360 degrees in your theta increment, you will end up with a small tail going into the cloud that gives the cloud a 3d effect. Be sure to set the fill color of the QuadCurve to match that of the ellipse.

Generalize this to a class called **BezierOval** that extends Shape so that you can use it as a shape in both your pond and cloud classes. Note that you will have to override setFill() so that whenever the fill color is changed that it is changed for the ellipse as well as all quad curves within the shape. For the cloud, nothing more is necessary. You should be able to just replace the declaration of Circle with one of of BezierOval and everything should just work. The sizes of the clouds don't change so there is nothing to think about there.

Observer Pattern

The clouds must implement observer and they must observe the wind. This way when the wind speed increases the cloud speed must increase accordingly. In this context the **Wind** is the subject, or observable, and the **Cloud** is the observer.

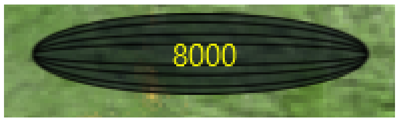
Pond

The ponds require a little bit more work, but not much. First, since the ponds do change shape, you will have to think about how to redraw the figure as the pond grows. Once choice is to just draw the figure the same as the pond size increases. This is fine. However, a more interesting choice would be to smooth out any edges as the pond size increases. This would mean carefully reducing the number of Bezier curves along the edge. You will have to experiment if you choose to put in this extra effort.

You should, in some way indicate the wind strength with waves along your pond. I suggest using either broken lines or Bezier curves, in black, to move across the pond in the direction of the wind.

We will use interfaces to define small slices of behavior and as a type for iteration purposes as needed. You will need the following interfaces. While there may be other interfaces going forward, you will want to define each class above with the following interfaces as necessary.

Class Blimp (new in A3)



For A3 we will have refueling blimps that enter the screen from the left and exit to the right. These should be very similar to the clouds in terms of operation; however, their speed should be faster. A blimp should be an image. To refuel, a helicopter flies over a blimp and matches speed with the blimp within a reasonable threshold. As long as the helicopter is matching blimp speed and over the top of the blimp refueling can begin. Blimps do not have an infinite amount of fuel. They have a finite amount from 5000 to 10000 and this will be shown on their body. It should take about ½ second to transfer 1000 fuel from blimp to helicopter. This means that you should have to match speed for five seconds in order to draw the maximum amount of fuel from a blimp.

Blimps regenerate randomly similar to the clouds. Adjust the regeneration rate for good playability.

Class TransientGameObject(new in A3)

This class extends GameObject and encapsulates all behavior for GameObjects that are Transient. Both Cloud and Blimp should extend this object. I suggest that you first implement Blimp without TransientGameObject and then refactor the common code between Cloud and Blimp. The idea is that this class manages objects that regenerate, move across the screen, and then go out of play. You must implement a state pattern. The states are relatively simple as we discussed in class.

Created -> InView -> Dead

When a transient object is first created out of view it is off screen. As soon as it is visible it switches to InView, once it has moved off screen, it is dead. If you are drawing the optional lines you only want to draw lines to transient objects that are not dead.

Organization by Panes (new in A3)

We discussed this concept in class and it will be required in A3. You must, at minimum, place your helicopter, your clouds, your blimps, and everything else in separate panes. This way your clouds will always fly over the ground and ground objects, your blimps will fly over the clouds, and your helicopter will fly over the blimps. I suggest implementing the iterable pane collection class that we showed in class. I will post some demo code for this. In any case, you are required to implement an iteratable collection. It doesn't have to do much more than leverage the backing store's iterator, but you do have to do that.

Interface Updatable

For this version of the game, we will only have this single interface. Updatable classes are dynamic and implement a callback, update() method that is invoked from the main game timer. Note that for this version of the game, not all game objects are updatable. For example, the helipad has no dynamic behavior at this time.

Sound

A3-Spec

We will be adding sound to the game for A3. Provide different sounds for helicopter takeoff, shutdown, and flying. Blimps should also have a sound of their own and there should be a somewhat constant background sound that gives a natural outdoor sound with wind blowing. If you like, you may also add background music.

Do not use any copyrighted music in your submission.

Cloud seeding should have a sound when you press the key to seed the clouds, and refueling should have a sound that plays when refueling is active.

There should be a rain sound that is active only when clouds are actively filling ponds. If you like, you can add thunder sounds that also trigger during this time.

Game Commands

For this version of the game all of the game input is via keyboard commands. Later on in the course we will talk in detail about things like command objects and event driven operation. For now, we are just going to use these features as given below. I will give you a basic explanation of the behavior and some boiler plate code to implement the commands.

The Commands

A1-Spec

Left Arrow Changes the heading of the helicopter by 15 degrees to the left.
Right Arrow Changes the heading of the helicopter by 15 degrees to the right.
Up Arrow Increases the speed of the helicopter by 0.1.
Down Arrow Decreases the speed of the helicopter by 0.1.
‘I’ Turns on the helicopter ignition.
‘b’ [optional] shows bounding boxes around objects.
‘r’ Reinitializes the game

A2-Spec

‘d’ [optional] Shows distance lines between clouds and ponds

For each command in our game, we want to add a key listener in our GameApp class. Note in JavaFX these are constants and you do not need to define them in terms of any integer or character constants. See the following URL:

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/KeyCode.html

Game Mechanics

You should have watched the game play demo video to get a basic idea of the gameplay for this version of the game. Note, we will be changing both the game play as well as the win/lose conditions throughout the term. At each stage the gameplay will reflect the learning goals for the module.

Game Play Overview

A1-Spec

The game begins with the helicopter stopped and resting on the pad. For this version of the game the engine is not running and will need to be started by pressing the ‘i’ key. Up until the helicopter is running, it does not consume fuel. As soon as you start the helicopter it consumes fuel on every cycle. The player will use the navigation keys to move the helicopter towards the cloud. At the cloud the helicopter will slow down and seed the cloud by pressing the space bar rapidly. The helicopter may leave the cloud at any time; however, the saturation level is always decreasing and the single pond will need to reach 100% capacity in order to avoid a loss.

To complete the game, the player must return to the landing pad and bring the speed of the helicopter back to zero and turn off the ignition. The game ends when the win-conditions are met or the helicopter runs out of fuel, whichever comes first. Once this happens the game will end and a dialog box will appear to report the player's score which is defined as the remaining fuel, and to give the player an opportunity to play again. If at any time during the mission the player runs out of fuel then the game is over and a dialog box will appear letting the player know that they have lost the game and, again, giving them an opportunity to play the game again.

A2-Spec

In A2, total pond capacity will need to reach 80% in order to avoid a loss. For this version of the game, the score is total capacity times the remaining fuel. So, if the remaining fuel is 100 and the ponds have reached 80% total capacity, then the score is 80. You may need to adjust the decay rate of the clouds and the fuel amount for playability.

A3-Spec

Game play in A3 is very similar to A2. We simply do not have enough time to get to all that was planned. One thing that we want to focus on in A3 is playability with respect to fuel. Make sure that your game has a challenging tradeoff between seeding clouds and refueling. You should not be able to win the game without at least one trip to a refueling blimp.

Helicopter Mechanics

The helicopter is somewhat more complex and will evolve significantly over the course of the term. For this assignment we want to focus on simple movement and cloud-seeding mechanics. The **Helicopter** object has a number of properties in addition to the location property. The helicopter's state includes fields for heading, speed, and fuel. Make sure that you review the section on object-oriented programming in this assignment before you start coding setters and getters arbitrarily.

Movement

The helicopter moves forward with increasing speed as the up arrow is pressed. You should define a maximum speed that the helicopter does not exceed. For this version of the game, increase the speed by 0.1 for each press of the up arrow and implement a max speed of 10 and a minimum speed of -2. Adjust all other timing factors to work with these choices. Pressing the down arrow reduces the speed and will begin to move the helicopter backwards as the speed becomes negative. In later versions of the game, we will adjust this behavior somewhat. In addition to more realistic flight mechanics, we will link the speed change to the device's refresh rate so that the game plays well on both fast and slow devices.

The right and left arrow keys will adjust the *heading* of the helicopter. In this first version we will keep this very simple and simply change the heading by fifteen degrees to the left or to the right based on which key was pressed. A heading of zero degrees represents due north and you will want to make sure that your code respects this convention. It is very important to realize, however, that most of calculations that you will need to execute will expect that zero degrees lies along the X axis. Moreover, the various trigonometric functions such as sine and cosine expect the argument to be in radians, and not degrees. Finally, because we are not using the standard device coordinates that causes the Y axis values to increase in a downward direction, a *turnLeft()* and *turnRight()* will work as expected in terms of direction. In order to achieve this easily, you will want to implement a Y scaling of (-1) in your main Game class (Pane).

Turning or changing speed has no immediate effect on the helicopter's movement. These actions merely change the state of the helicopter by updating the helicopter's speed and heading fields. The helicopter's position is updated when its *move()* method is invoked from the *update()* method in the Game class.

Detailed Program Structure

Before we start talking about the internal structure of this project, let's clarify some things about programming in Java related to common misconceptions as well as some unusual requirements for this assignment.

New for A3: Packages and One file per public class

For this assignment you will put each public class in its own file. This is the standard way to program in java. You may still have some non-public classes in files with other classes. For example, it might be helpful to put your state pattern classes in the same file with the class for which they manage the state.

You will also put your code in packages. Your base package will be ***rainmaker*** and you will also setup a package called ***gameobjects*** (***rainmaker.gameobjects***) and you will put all of your game objects into this package. You may have to refactor some of you code to accommodate this. I will briefly demonstrate in class how to move code into packages and what this entails in terms of referencing your code.

Constants and Static Methods

A1-Spec

Create final static constants in your Game class for anything that needs to be easily changed, for example, the GAME_HEIGHT and GAME_WIDTH which should initially be 800 and 400 respectively.

A2-Spec

For A2 we want to move to a larger screen. Ideally, we would use a screen size of 800x800. If you have a laptop with a much small screen you may need to make it somewhat smaller.

Screen Origin and Coordinate System

The origin, (0,0), of the screen is located, by default, in the upper left-hand corner with positive values increasing to the right and down. We will invert this by scaling the main pane by (-1). This will also require us to scale all text by (-1) to avoid mirroring. I suggest that you use a GameText class to achieve this easily.

A3-Spec

We won't be changing anything with respect to coordinates with A3. Dealing with JavaFX transformations is challenging enough as it is.

Game Structure

The Game class provides the model for our game. It manages the changing state of our game as we interact with it. Your Game must declare and initialize all other game objects, manage the initialization of the game, determine when the game is won or lost, and create all of the objects in the game world by placing them in the scene graph. Your Game must have an init() method that is distinct from the constructor. The init() method is invoked whenever a new game must be played. It is perfectly reasonable to simply recreate all of the game objects in this init() method. The init() method creates all of the new state of the world including the positioning of each of the game objects. Don't forget to clear all children out of the Pane before initializing new objects.

As we have seen from the Game class structure, Game has an update() method that is called to update the state of the game. In this method you need to move your helicopter and check the win/lose status of the game.

If the game as reached either a win or lose condition, the game should invoke a *modal dialog box* as shown in the demo video to offer to quit or replay.

You may create helper methods as necessary but they should not be public unless required. Later we may move these methods to a separate class.

A3-Spec

For A3 your Game class must be a singleton. You must provide a statice Game.getInstance() method that returns the Game instance. You may not pass the Game object around to other classes.

OOP concerns for this Project

Private Data and Setters and Getters

In this course, all mutable, i.e., settable, fields must be private. This means that you may need setters and getters to obtain the value of this data externally or to set the value of this data. You do not always need setters or getters, however, and it's important to try to think in terms of behaviors. As discussed in class, you should not assume that it's best to just add setters and getters for each variable. This is unnecessary and is reasonably equivalent to just making the data public. Unnecessary setters and getters indicate lower quality code and may negatively impact your grade.

Coding STandards

Many of the required coding standards for this project have been defined throughout this document. In addition, you must adhere to the following. Note, these are only starter guidelines and you should have learned these in your previous courses. You will be learning the basics of Clean Coding in this course and your grade will, in part, reflect the degree to which you adopt those standards.

1. Class names always start with an upper-case letter
2. Variable names always start with a lower-case letter
3. Non-Constant identifiers use camel case
4. Constant identifiers use upper snake case
5. All code is neat and properly indented
6. You are restricted to an **80-character width**
    a. I want you to break habits that you may have developed of writing very long lines of Java code. You must learn to limit your width to aid readability.
    b. Java allows you to break lines in places you might not have thought about. Learn to structure your code more vertically with carefully placed line breaks that do not change the semantics of the Java language.
    c. The reason that it's so important for this first project is that it makes peer review and grading easier. Lines longer than 80 characters will break in speed-grader making your code less readable.
    d. ***You WILL lose points for not adhering to this requirement.***

In the clean code discussion, the authors will warn you that commenting is a code smell. I'm quite sure that your other instructors have told you to comment excessively. We will be focusing in this course on writing clean and self-documenting code. However, this is not always possible and, where necessary, you must communicate your intent with comments.

Meta-Comments

For this submission I definitely want you to leave meta-comments. These are communications to your instructor about what your intent and choices were. Tell me about your thought process while writing your code. This should be a natural part of your coding. If you practice this while you're coding, then you will learn to develop text that can go in documentation at a later time. For future versions of the project, you will most likely have to submit a writeup that includes much of this discussion. You definitely want to practice communicating about your design process now!

Submission

Submission standards are changing for this semester. For now, just get started with your coding, I will discuss submission mechanisms in class. In short, you will be submitting your work, in progress, every week on Monday. You will be expected to show constant improvement in your code over the remainder of the term and you will be expected to meet specific milestones. This approach is intended to keep you on track while allowing you some flexibility. If you miss too many milestones then it is possible to fail the project.