COMP 452

GAME DESIGN DOCUMENT

ASSIGNMENT 2

Jody Pequin

Contents

[Game 1: 1](#_Toc123332120)

[How To Play: 1](#_Toc123332121)

[Run: 1](#_Toc123332122)

[Controls: 1](#_Toc123332123)

[Description: 1](#_Toc123332124)

[General Overview of Development: 2](#_Toc123332125)

[Description: 2](#_Toc123332126)

[Assets Borrowed: 5](#_Toc123332127)

[Algorithm: 5](#_Toc123332128)

[A\*: 5](#_Toc123332129)

[Heuristic: 7](#_Toc123332130)

[Bugs: 9](#_Toc123332131)

[Bug 1: 9](#_Toc123332132)

[Bug 2: 10](#_Toc123332133)

[Game 2: 11](#_Toc123332134)

[How To Play: 11](#_Toc123332135)

[Run: 11](#_Toc123332136)

[Controls: 11](#_Toc123332137)

[Description: 11](#_Toc123332138)

[General Overview of Development: 14](#_Toc123332139)

[Description: 14](#_Toc123332140)

[State Machine Implementation: 17](#_Toc123332141)

[StateManager.java 20](#_Toc123332142)

[AntPlayerState.java 21](#_Toc123332143)

[FindFood.java 22](#_Toc123332144)

[FindHome.java 23](#_Toc123332145)

[FindWater.java 24](#_Toc123332146)

[Death.java 24](#_Toc123332147)

[Bugs: 25](#_Toc123332148)

[Bug 1: 25](#_Toc123332149)

[Bug 2: 26](#_Toc123332150)

# Game 1:

## How To Play:

### Run:

To start the game, double-click on the jar file called “desktop-1.0.jar”. Then from the main menu, select:  
 *Assignment 2 🡪 Game 1*

To build a new “desktop-1.0.jar”, go into COMP452 folder in the terminal (I do this in Android Studio) and type the command:  
 ./gradlew desktop:dist

### Controls:

Left Mouse Click – To select tiles, textures and buttons

ESC – to go to Pause Menu

### Description:

The Ant Queen needs help training her minions to hunt for food, create different mazes for the new recruits to search through to find the quickest path to the food!

Graphical user interface, application

Description automatically generatedThe panels on the left side of the screen are the different tiles that can be selected to create the map. Tooltips will popup to help indicate how much a tile costs as well as which tile is currently selected to put on the map.

When a tile is selected, either *Click* and/or *Click’n Drag* the mouse to cover the map with the desired texture. If you’re unhappy with the map, click the *RESET* button at the top to put the map back to default. When finished, select the *RUN* button at the top of the screen to begin simulation. There must be only one Ant and one Berry on the field, if there isn’t, a warning sign will be displayed indicating what needs to be fixed before the simulation will begin. Once the changes are fixed, select *RUN* again to begin.

Text

Description automatically generated

The simulation will show the Ant move from tile to tile on it’s search for the Berry, indicated by the Ant’s current position and visited tiles shown in purple. If you don’t want to wait for the Ant, click *Show End* to see the Ant’s final path. The *Redo Simulation* button will restart the Ant’s journey again from the beginning and the *Map Style* button will send you back to the create map screen.

When the recruit is good and trained head to Game 2 to see them all in action!

## General Overview of Development:

### Description:

There are 2 main screens, *Game1.java* and *RunMap.java*, *Game1.java* starts the game for the user to create the map of their choosing, then *RunMap.java* is called to run the algorithm and display the results back to the user. For the map creation, the *TileSelector* class is called to display the blank map and the tiles the user can pick from. The tiles in this map are an array of Button objects with listeners attached to them, were used to make it easier for development and user interaction since they already have the desired behaviors I was wanting and just needed to modify them (e.g. changing the image when the user clicks on a tile, having a Tooltip popup when the mouse enters a tile, ect.). Once the user clicks the *Run* button, the map will check for 1 Ant and 1 Berry, if these are not met the simulation will not run until they are. If requirements are met, the Button array will be scanned through to create Tile objects, an integer list of adjacent Tiles for each Tile, flag the Start and End tiles (the Ant and Berry tiles) and assign the initial Tile costs(e.g. Dirt Tile = 1).

The integer list of neighbor tiles, specifically their ID numbers, is the Connection list that will be used in the Path Finding algorithm. The only tiles that are included are the tiles that the Ant can travel to, if a tile is flagged as a “Rock\_Tile” its ID number will not be added to the connection list. The ID numbers are based on their positions in the original Button array, for example:

With a 16x16 map, two for loops were used to create it and their combined values are used to make the index numbers:

for(int j = 0; j<16; j++ ){

for(int i=0; i<16; i++) {

btnTiles[(i\*16) + j] << if i= 2 and j=10 then the Tile ID will be 42

The final layout then looks like this with each cell representing its index/ID number:

Table

Description automatically generated with medium confidenceThese IDs are further used when working with calculating the heuristic value (explained below in the Algorithm section).

When the game switched to the *RunMap* screen, the simulation will begin and the A\* algorithm in *PathFindingAStar.java* will start. The figure below shows an example of the result of a simulation, the yellow path is the final path the Ant took to get to the Berry and the purple tiles are the visited tiles but not deemed the shortest path. The final results are then displayed at the top with the assumption that the cost to go from a tile to the berry is zero.

Text

Description automatically generated with medium confidence

Game 1 isn’t too extensive in design but there are some design features that could’ve been better implemented. It became clear that my game could’ve benefited from an Asset Manager, there are a couple instances where I’m loading the same Textures – once in *TileSelector* and then again in *RunMap* – an Asset Manager would help get rid of that overhead of loading these assets into memory as the game is being run. From what I’ve read online, having these loaded once when the game first loads, is better for performance. Another design flaw is how I’m working with the Tiles, first I make a Button array and then I transform that into a Tile array (technically a priority heap), although the game map isn’t large, this would become cumbersome if the map was much larger. It probably would’ve been more beneficial to make the Tiles extend a Button object and just use one array for the Tile/Button objects and another for keeping track of which were modified. Lastly, there are a couple of cosmetics that in hindsight I should’ve done differently. The tooltip that pops-up when the cursor is hovered over is neat but the ”Cost” of the tiles should’ve been just a label. Then when the simulation is running, it can be kind of hard to see where one Tile begins and the other ends, when making the Textures I should’ve put in a distinct boarder to help with this.

Graphical user interface, diagram, Teams

Description automatically generated

### Assets Borrowed:

* Okami sound bites from <https://www.sounds-resource.com/playstation_2/okami/sound/7689/>

Similar to game 1, this sound bite was modified, using OcenAudio, to give the AntInro voice the different pitches to mimic a voice.

* All assets created by me with Procreate

## Algorithm:

### A\*:

The A\* algorithm is located in *PathFindingAStar.java* under the function *findPath()*, the *findPath* function is called from *RunMap.java* in *render*. It’s design is very similar to our textbook’s example with the exception of having a specific Tile that is returned if an ‘End” can not be reached. This ‘noGoal’ Tile was just an easy way to signal *RunMap* that a goal can not be found without it disrupting the rendering process.

*/\*\*  
 \* A\* Algorithm implementation  
 \* Keeping polling the top of the priority queue until either the 'End' Tile is found or there   
 \* are no more tiles to get from the queue.  
 \** ***@return*** *Tile that was currently looked at or the 'noGoal' Tile if there are no more tiles to  
 \* look at  
 \*/*public Tile findPath(){  
  
 // Check if there are any more tiles to be searched - if not, return the 'noGoal' Tile  
 if(open.size() == 0){  
 return noGoal;  
 }  
 else {  
 // Get the Tile with the lowest F(n) cost  
 current = open.poll();  
 // Get the connections attached to the current Tile  
 List<Integer> connections = current.getConnections();  
  
 // Parse through the Tiles that are neighbors to the current tile we're at  
 for (Integer conn : connections) {  
 // Fetch the Tile thats connected to the current tile  
 Tile connectionTile = graph.get(conn);  
  
 // Get the new 'G' cost by adding the current tile 'costSoFar' to the connecting  
 // tile's cost  
 float currentCost = current.getCostSoFar() + connectionTile.getCostSoFar();  
  
 // If we found the Berry then calculate costs and return  
 if (connectionTile.getNode().equals("End")) {  
  
 connectionTile.setCostSoFar(currentCost);  
 connectionTile.setEstimatedTotalCost(currentCost);  
 // Set the connection from the current tile to the neighbor tile - this is used  
 // to back track and get the shortest path  
 connectionTile.setConnection(current);  
 // Add the connecting tile to the open queue if its not there  
 if (!open.contains(connectionTile)) {  
 open.add(connectionTile);  
 }  
 // Set current Tile's state to closed  
 current.setState(0);  
 return current;  
 }  
  
 // If Berry is not found then keep searching  
 // If neighbor tile is 'CLOSED', double check for shorter path else continue  
 if (connectionTile.getState() == Tile.Category.*CLOSED*) {  
 // If a shorter route wasn't found then skip  
 if (connectionTile.getCostSoFar() <= currentCost) {  
 continue;  
 }  
 // Else set state as open  
 connectionTile.setState(1);  
 // Calculate new heuristic and set it  
 newEndCost = connectionTile.getEstimatedTotalCost() - connectionTile.getCostSoFar();  
  
 // If tile is 'UNVISITED' then calculate heuristic and put it in the open list  
 } else if (connectionTile.getState() == Tile.Category.*UNVISITED*) {  
 // Set Tile to 'OPEN'  
 connectionTile.setState(1);  
 // calculate new heuristic  
 newEndCost = hue.estimatedCost(connectionTile, connectionTile.getCostSoFar());  
  
 } else { // Else tile is 'OPEN'  
 // Compare the 'G' cost, if the route isn't better than skip  
 if (connectionTile.getCostSoFar() <= currentCost) {  
 continue;  
 }  
 // calculate new heuristic  
 newEndCost = connectionTile.getEstimatedTotalCost() - connectionTile.getCostSoFar();  
  
 }  
  
 //Update 'G' value  
 connectionTile.setCostSoFar(currentCost);  
 // Update 'F' cost with 'H' (pass to heuristic function) and 'G'  
 connectionTile.setEstimatedTotalCost(newEndCost + currentCost);  
 // Set the Tile connected to the tile we're currently at  
 connectionTile.setConnection(current);  
  
 // If tile is not in Open tile then add it  
 if (!open.contains(connectionTile)) {  
 open.add(connectionTile);  
 }  
  
 }  
 // Set current Tile's state to 'CLOSED' then return  
 current.setState(0);  
 return current;  
  
 }  
  
}

### Heuristic:

As mentioned above in the Development Notes, the Heuristic class uses the Tile ID to help generate the heuristic value used in the A\* algorithm. When the class is created it is given the “End” Tile ID which it then extracts the X and Y coordinates. Since the tiles were drawn as 32x32, this was used as an arbitrary number to represent the X and Y plane. For example, if the Berry has a Tile ID of 205:

X = 205/16 = 12  
 Y = 205 – (12 \* 16) = 13

These X and Y values represent the Tiles that are from the start of the axis up to the Berry Tile.  
Graphical user interface, table

Description automatically generated

X and Y are then multiplied by 32, because that’s technically the size of the tiles, and 16 is added to get the center coordinate of Tile 205. The center coordinate was used to try and get the closest estimated distance from a current tile to the Berry Tile. So with a 16x16 map, we’d end up with:

X-Axis = 0 to 512  
Y-Axis = 0 to 512  
Tile 205 center coordinate = (400, 432)

I had a lot of problems with the heuristic, originally, I was using the Euclidean Distance but I was getting odd results. Eventually I found this [Reddit](https://www.reddit.com/r/roguelikedev/comments/59u44j/warning_a_and_manhattan_distance/) page that explained the Euclidean Distance is really if movement doesn’t matter. For instance, I wanted my ant to move from Tile to Tile in 8-directions, but Euclidean is used if movement isn’t restricted. The people on the Reddit page talked about the Chebyshev Distance and when I implemented that, I got much better (less wild Tile searching) results.

*/\*\*  
 \* This class will find and return the Heuristic value from a node to the set End Node  
 \* f(n) = g(n) + h(n)  
 \*  
 \* f(n) is the potential distance between start and target node - n=current  
 \* g(n) is the distance between the start and current node - n=current  
 \* h(n) is the estimated distance between the current node and the target node - n=current  
 \*  
 \* h(n) is the heuristic  
 \*  
 \* We want to minimize f(n) to find the shortest path  
 \*/*public class Heuristic {  
  
 private float endX, endY;  
  
 public Heuristic(Tile end){  
 float id = end.getID();  
 int x = (int)id/16;  
 int y = ((int)id - (x \* 16));  
  
 // Translate the extracted x and y coordinates  
 // Multiply by 32 as this is the pixel size and add 16 as to signify the center of the tile  
 endX = x\*32+16;  
 endY = y\*32+16;  
  
 }  
  
 */\*\*  
 \* This function will calculate the Euclidean distance from the current node to the target  
 \* node and return the calculated value.  
 \* Chebyshev distance reference:  
 \** *https://www.reddit.com/r/roguelikedev/comments/59u44j/warning\_a\_and\_manhattan\_distance/  
 \** ***@return*** *the distance between the current node and the target node  
 \*/* public float estimatedCost(Tile currentTile, float cost){  
 // Use the current node's ID to calculate the x and y coordinates  
 float id = currentTile.getID();  
 int currX = (int)id/16;  
 int currY = ((int)id - (currX \* 16));  
  
 // x distance from current node to target node on the x-axis  
 // y distance from current node to target node on the y-axis  
 // Translate the extracted x and y coordinates  
 // Multiply by 32 as this is the pixel size and add 16 as to signify the center of the tile  
 float x = *abs*(endX - (currX\*32 + 16));  
 float y = *abs*(endY - (currY\*32 + 16));  
  
 // Multiply by the cost of the tile because we're using a weighted graph  
 float distance = cost \* Math.*max*(x,y);  
  
 return distance;  
 }  
  
}

## Bugs:

### Bug 1:

If one tile is selected, it will show the tooltip above the mouse cursor but if the mouse is moved onto a different tile, it will show the cost of the tile it’s currently hovering over. For instance in the figure below, the Swamp tile is selected but when the mouse is over the Grass tile it’s displaying the cost for the Grass tile.



### Bug 2:

When doing the *Click’n Drag* of the tiles on the map sometimes the first tile isn’t “clicked” and needs to be clicked again to give it the desired texture.

# Game 2:

## How To Play:

### Run:

To start the game, double-click on the jar file called “desktop-1.0.jar”. Then from the main menu, select:  
 *Assignment 2 🡪 Game 2*

To build a new “desktop-1.0.jar”, go into COMP452 folder in the terminal (I do this in Android Studio) and type the command:  
 ./gradlew desktop:dist

### Controls:

Left Mouse Button – To select Text Field and buttons on the screen

Enter – Once there is a number in the Text Field, hit enter to begin

### Description:

Now that the new recruits are trained, its time for their field test. Send out a unit to look for near by berries to bring back to the colony for the Queen! The humans have also left purple goop on the ground, wonder what it is..

The player will start the simulation by inputting how many ants they would like to see on the screen.  
Shape

Description automatically generated

The simulation will load the environment with the following:

* 1 Ant Hill  
  
* 2 Berries  
  
* 2 Poison Spots  
  
* 2 Water Spots  
  

These will be randomly placed on the map and all ants will spawn from the “Ant Hill” location.

Ants can be differentiated by their color:

* Looking for Berry:  
  
* Berry found, looking for Ant Hill:  
  
* Berry dropped off at Ant Hill, looking for Water:  
  

A screenshot of a video game

Description automatically generated with medium confidence

As the simulation is running, there will be an Alive/Dead ant count at the top for the user to keep general track of the status of the ants. Graphical user interface

Description automatically generated with low confidence

At any time, the “Show Results” button will pause the simulation and show the status of all ants. The “Continue” button will resume the simulation.

Text

Description automatically generated

When finished, either close the application or press <ESC> to either exit to the main menu or run another simulation.

*Player Tip!*  
Ants have tendency to find poison easily so start with a bunch for more interesting results.

## General Overview of Development:

### Description:

When the game begins, the player will be prompted to enter the starting number of ants. *Game2.java* controls the game’s overall flow from this initial prompt to initializing and running the simulation and showing the final results. An *AssetManager* has been added in this game to help with texture loading overhead (an issue described in Assignment 2 – Game 1).

Graphical user interface, Teams

Description automatically generated

The *AntFactory.java* is responsible for creating the *AntPlayer* objects, which are then placed in an array to keep track of the ants as they’re generated and die. The game’s items (berries, poison and water) are constantly set to randomly place 2 each and 1 ant hill. After several tests, these numbers seem to be fairly good to keep the simulation running with steady numbers. It really depends on where the berries and poison spots are located and number of ants generated at the beginning (my tests were run with 10 starting ants). The closer the berries are to the ant hill, the more new ants are likely to be created and if poison spots are close to the ant hill, the ants are more likely to all die off faster.

A picture containing text, electronics

Description automatically generatedThe map was created with two different layers, one that is the Dirt background with the generated berries, poison, ect. The other layer is one that keeps track of the ants, showing their current position. Each ant has its own state machine (*StateManager.java* explained in State Machine Implementation section) and a movement class (*AntPlayerMovement.java*). The movement class uses a simple *wander* behavior where the program will get a random cell that is surrounding the ant and move to that position:

There are several lists that are used to select a cell depending on where the ant is located on the map, for instance, if an ant is located on the very left edge of the map, cells 6, 7 and 0 will not be picked from. If an ant has a berry and needs to return home, a heuristic function (*HeuristicGame2.java)* is used along with the *returnHome* function for the ant to go directly home. This is with the assumption that the ant knows the way back to the colony.

*Hud.java* keeps a simple tally of the number of alive ants vs dead ants and will also display the final results when “Show Results” button is selected. When the button is selected, the simulation will pause and the *Hud* class will go through the ant array and tally up which state the ants are in (minus the deceased ants, that is tallied as the ants touch poison).

Text

Description automatically generated When done with the simulation, hit ESC to bring up the Pause Menu to select a different Bug Wars game action.

## State Machine Implementation:

Diagram

Description automatically generated

Each ant object is initialized with a *StateManager*, *AntPlayerState, FindFood, FindHome, FindWater,* and *Death* state. As the simulation is running, the ant object will check its current state via *StateManager* and run the state’s *Update.*

public void Update(){  
 currentState = stateManager.getCurrentState();  
 currentState.Update();  
}

The *AntPlayer* class is the ant objects, it houses boolean functions to check what the ant is currently stepping on. These will be called by the states *FindFood, FindHome,* and *FindWater*.

*/\*\*  
 \* 'FindFood' state will call this function to see if the ant is stepping on a berry  
 \** ***@return*** *\*/*public boolean checkBerry(){  
 String currentCell = map.checkCell(antCurrentPos());  
 if(currentCell.equals("Berry")){  
 // Set the ant picture to the 'pink' ant to indicate it has a berry and looking for home  
 antCurrentPic = antPicBerry;  
 return true;  
  
 }  
 return false;  
  
  
}  
  
*/\*\*  
 \* 'FindFood', 'FindHome' and 'FindWater' states will call this function to see if the ant is  
 \* stepping on poison  
 \** ***@return*** *\*/*public boolean checkPoison(){  
 String currentCell = map.checkCell(antCurrentPos());  
 if(currentCell.equals("Poison")){  
 return true;  
  
 }  
  
 return false;  
}  
  
*/\*\*  
 \* 'FindHome' state will call this function to see if the ant is stepping on an AntHill  
 \** ***@return*** *\*/*public boolean checkHome(){  
  
 String currentCell = map.checkCell(antCurrentPos());  
 if(currentCell.equals("Home")){  
 game.makeNewAnt();  
 // Set the ant picture to the 'blue' ant to indicate its thirsty and looking for water  
 antCurrentPic = antPicWater;  
 return true;  
  
 }  
  
 return false;  
}  
  
*/\*\*  
 \* 'FindWater' state will call this function to see if the ant is stepping on water  
 \** ***@return*** *\*/*public boolean checkWater(){  
  
 String currentCell = map.checkCell(antCurrentPos());  
 if(currentCell.equals("Water")){  
 // Set the ant picture back to normal  
 antCurrentPic = antPic;  
 return true;  
 }  
  
 return false;  
}

The *Death* state is the only one that will call the *AntPlayer*’s function *dispose*:

*/\*\*  
 \* 'Death' state will set 'antAlive' variable to false to indicate death  
 \*/*public void dispose() {  
 antAlive = false;  
}

This is simply set as a flag and gets properly disposed of once the cycle is completed. This is done in *Game2.java* in the *update* function:

// Render ants, the time intervals are to see the movements at a steady speed  
if (currentTime >= startTime) {  
 for (AntPlayer ant : ants) {  
 if (ant.getAntAlive()) {

// Clear the ant image from its current position  
 scene.cellUpdateAntPrevPos(ant.antPreviousPos());

// Put the ant image from to its new current position  
 scene.cellUpdateAntPos(ant, ant.antCurrentPos());  
 ant.Update();  
 } else { // If an ant is dead put it in the remove list to dispose of object in update  
 deadAnts.add(ant);

// Clear the ant image from its current position  
 scene.removeDeadAnt(ant.antPreviousPos());  
 hud.setDeadAnts();  
 }  
 }  
 startTime = *millis*() + 500;  
}  
// Dispose of dead ants  
for (AntPlayer deadAnt : deadAnts) {  
 ants.remove(deadAnt);  
}  
deadAnts.clear();

### StateManager.java

The *StateManger* initializes the ant’s starting state, keeps track of the it’s current state and changes the ant’s state depending on if certain requirements are met.

*/\*\*  
 \* Holds a variable to our current state and a function to change current state  
 \*/*public class StateManager {  
  
 private AntPlayerState currentState;  
  
 */\*\*  
 \* Initialize the starting state for an ant  
 \** ***@param*** *startingState  
 \*/* public void Initialize(AntPlayerState startingState){  
 this.currentState = startingState;  
 currentState.Enter();  
 }  
  
 */\*\*  
 \* Change the state of the ant  
 \** ***@param*** *newState  
 \*/* public void ChangeState(AntPlayerState newState){  
 currentState.Exit();  
 currentState = newState;  
 currentState.Enter();  
  
 }  
  
 public AntPlayerState getCurrentState (){  
 return currentState;  
 }  
   
  
}

### AntPlayerState.java

This class is the parent class that all other states inherit their pattern from. This class technically isn’t required since there is only a few states the ants can be in but is still useful when passing state information since we don’t have to define functions to take each different state.

*/\*\*  
 \* Parent class of the ant's different states (FindFood, FindHome, FindWater and Death).  
 \*/*public class AntPlayerState {  
  
 protected AntPlayer player;  
 protected StateManager stateMachine;  
  
 public AntPlayerState(AntPlayer player, StateManager stateMachine){  
 this.player = player;  
 this.stateMachine = stateMachine;  
  
 }  
  
 */\*\*  
 \* Enter gets called when we enter a specific state  
 \*/* public void Enter(){  
 //DoCheck();  
  
 }  
  
 */\*\*  
 \* Gets called when we leave a state  
 \*/* public void Exit(){  
  
 }  
  
 */\*\*  
 \* Gets called every frame  
 \*/* public void Update(){  
 DoCheck();  
 }  
  
 */\*\*  
 \* Check criteria if states need to be changed  
 \*/* public void DoCheck(){  
  
 }  
}

### FindFood.java

Keeps the ant using the *wander* movement to find food and uses the ant object’s functions to check if berries or poison has been stepped on. If a berry is found call the state manager to change to the *FindHome* state.

*/\*\*  
 \* This class inherits from 'AntPlayerState' so it has access to everything from this class.  
 \* Part of the Ant FSM, this will use the 'wander' algorithm and look for 'berry' and 'poison' tiles  
 \*/*public class FindFood extends AntPlayerState {  
  
 public FindFood(AntPlayer player, StateManager stateMachine) {  
 super(player, stateMachine);  
  
 }  
  
 @Override  
 public void Enter() {  
 super.Enter();  
 }  
  
 @Override  
 public void Exit() {  
 super.Exit();  
 }  
  
 @Override  
 public void Update() {  
 super.Update();  
 player.movement.wander();  
  
 }  
  
 @Override  
 public void DoCheck() {  
 super.DoCheck();

// Use the ant’s checkBerry function to see if it’s on a berry  
 if(player.checkBerry()){  
 stateMachine.ChangeState(player.findHome);  
 return;  
 }

// Use the ant’s checkPoison function to see if it’s on a poison  
 if(player.checkPoison()){  
 stateMachine.ChangeState(player.foundDeath);  
 return;  
 }  
  
  
 }  
}

### FindHome.java

Similar to the *FindFood* state, this state keeps the ant using the *returnHome* movement function and uses the ant’s functions to see if it has stepped on the ant hill or poison. If the ant hill has been found, call the state manager to change states to *FindWater*.

*/\*\*  
 \* This class inherits from 'AntPlayerState' so it has access to everything from this class.  
 \* Part of the Ant FSM, this will use the 'returnHome' algorithm and look for 'ant hill' and 'poison' tiles  
 \*/*public class FindHome extends AntPlayerState {  
  
 public FindHome(AntPlayer player, StateManager stateMachine) {  
 super(player, stateMachine);  
 }  
  
 @Override  
 public void Enter() {  
 super.Enter();  
  
 }  
  
 @Override  
 public void Exit() {  
 super.Exit();  
 }  
  
 @Override  
 public void Update() {  
 super.Update();  
 player.movement.returnHome();  
 }  
  
 @Override  
 public void DoCheck() {  
 super.DoCheck();

// Use the ant’s checkHome function to see if it’s on the ant hill  
 if(player.checkHome()){  
 stateMachine.ChangeState(player.findWater);  
 return;  
 }

// Use the ant’s checkPoison function to see if it’s on a poison  
 if(player.checkPoison()){  
 stateMachine.ChangeState(player.foundDeath);  
 return;  
 }  
  
  
 }  
}

### FindWater.java

Keeps the ant doing the *wander* movement in search for water using the ant’s functions to check if its stepping on a water tile. If water is found, go back looking for food.

*/\*\*  
 \* This class inherits from 'AntPlayerState' so it has access to everything from this class.  
 \* Part of the Ant FSM, this will use the 'wander' algorithm and look for 'water' and 'poison' tiles  
 \*/*public class FindWater extends AntPlayerState {  
  
  
 public FindWater(AntPlayer player, StateManager stateMachine) {  
 super(player, stateMachine);  
 }  
  
 @Override  
 public void Enter() {  
 super.Enter();  
 }  
  
 @Override  
 public void Exit() {  
 super.Exit();  
 }  
  
 @Override  
 public void Update() {  
 super.Update();  
 player.movement.wander();  
 }  
  
 @Override  
 public void DoCheck() {  
 super.DoCheck();

// Use the ant’s checkWater function to see if it’s on water  
 if(player.checkWater()){  
 stateMachine.ChangeState(player.findFoodState);  
 return;  
 }

// Use the ant’s checkPoison function to see if it’s on a poison  
 if(player.checkPoison()){  
 stateMachine.ChangeState(player.foundDeath);  
 return;  
 }  
 }  
}

### Death.java

With each of the three states mentioned above, if at anytime an ant is found to be stepping on a poison tile, the *Death* state will be set by the state manager and simply dispose of the ant.

*/\*\*  
 \* This class inherits from 'AntPlayerState' so it has access to everything from this class.  
 \* Part of the Ant FSM, this kill the ant :(  
 \*/*public class Death extends AntPlayerState {  
  
  
 public Death(AntPlayer player, StateManager stateMachine) {  
 super(player, stateMachine);  
 }  
  
 @Override  
 public void Enter() {  
 super.Enter();  
 player.dispose();  
 Exit();  
 }  
  
 @Override  
 public void Exit() {  
 super.Exit();  
  
 }  
}

## Bugs:

### Bug 1:

When the game starts, there is no blinking indicator to show that the text box has been selected. From what I can tell, that feature isn’t apart of the TextField button objects that I used.

Shape

Description automatically generated

### Bug 2:

When ants are moving around, sometimes the ants run into each other (overlap) so it can be hard to tell how many ants are on the field. This is mainly seen when the simulation is first initialized and all the ants come out of the ant hill. The figure below shows the start of a game that was initialized with 10 ants but clearly only 7 are seen.  
A picture containing graphical user interface

Description automatically generated