COMP 452

GAME DESIGN DOCUMENT

ASSIGNMENT 3

Jody Pequin

Contents

[Game 1: 1](#_Toc128144714)

[How To Play: 1](#_Toc128144715)

[Run: 1](#_Toc128144716)

[Controls: 1](#_Toc128144717)

[Description: 1](#_Toc128144718)

[General Overview of Development: 2](#_Toc128144719)

[Description: 2](#_Toc128144720)

[AI: 4](#_Toc128144721)

[Heuristic: 6](#_Toc128144722)

[Bugs: 10](#_Toc128144723)

[Game 2: 11](#_Toc128144724)

[How To Play: 11](#_Toc128144725)

[Run: 11](#_Toc128144726)

[Controls: 11](#_Toc128144727)

[Description: 11](#_Toc128144728)

[General Overview of Development: 14](#_Toc128144729)

[Description: 14](#_Toc128144730)

[Assets Borrowed: 17](#_Toc128144731)

[AI: 18](#_Toc128144732)

[Utility Theory: 18](#_Toc128144733)

[Patrol: 19](#_Toc128144734)

[Lunge: 20](#_Toc128144735)

[TailSwipe: 21](#_Toc128144736)

[Beam: 23](#_Toc128144737)

[Bugs: 25](#_Toc128144738)

# 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 3 🡪 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:

A, D – To move left and right to select your character and move game piece

SPACE – to Select character and place game piece

ESC – to go to Pause Menu

### Description:

 *Bug Wars: Battle of the Minds* faces the Spider and Ant against each other in an epic battle of Connect 4. After the intro, select the character you wish to play as, the Ant will always go first.

The game’s rules are the same as Connect 4, a player’s goal is to connect 4 disks in a row either horizontally, vertically, or diagonally. The Spider and Ant’s game pieces are color coded to their respected colors:

Ant – Pink Piece  
 Spider – Green Piece

When one player has completed a winning move, those game pieces will be highlighted yellow to indicate where they won, along with a winning text.



Contend with your favorite character to be the victor! When completed, head to Game 2 to see the epic conclusion to *Bug Wars*…

## General Overview of Development:

### Description:

In this game I kept development straight forward but tried to immerse a little more story and tie together the first 2 assignments by making the Spider and Ant the players of the game. I also implemented a character selection screen for user engagement but to also show how the AI works when its first or second player.

Graphical user interface, application, Teams

Description automatically generated Our assignment states that the player with the ‘white’ chips go first, the Ant is considered the ‘white’ piece which is why it will always go first, to conform to this game requirement. When researching heuristics with Connect 4, I read that “the first player can force a win, on or before the 41st move by starting in the middle column,”[[1]](#footnote-1) for this reason I thought it would be neat to have the user decide if they would like to go first. So if the user knew this, they could technically make the game alittle more challenging by choosing to go second. The AI is implemented with the NegaMax algorithm with Alpha-Beta pruning as recommended in the assignment (see AI section below). I followed Chapter 9 of the textbook when designing the overall class structures of the game, this is mainly scene in the *Board* class with its functions such as *getMoves, makeMove, evaluateBoard* and *isGameOver*. The player and AI player ID’s are set to 0 for the Ant or 1 for the Spider depending on which the user picks. The 0 or 1 are then assigned to the *Player* and *AIPlayer* classes respectively.

*Game1.java* is the main hub of the game that will control the flow, so flagging which player’s turn it is and telling the *CreateScene* class what needs to be rendered. *Player* and *AIPlayer* essentially just flag the other classes when they’re done making move, *AIPlayer* does have more interaction with the *Board* class as it uses the evaluating move positions there.

The game has three end states: Ant wins, Spider wins and a tie. The tie state is difficult to achieve, to show it I put the heuristic score values to 0 and below is a picture of a completed game with it’s game over text.



Overall, the game seems to play smoothly but when selecting the playable character, there is a slight couple second delay to the game starting. This is because the game screens are being switched and the Asset Manager is loading all the game textures. Ideally, I should have the Asset Manger load with all textures when *Bug Wars* first starts.

### AI:

As mentioned, the AI is made with the NegaMax and AB pruning, located in *AIPlayer.java*. Originally, I implemented MiniMax /w AB pruning (code is in *AIPlayer.java*) but when looking for tutorials and details on it, a lot of comments said advised others to use NegaMax. After implementing MiniMax I decided to try out NegaMax and decided to go with that because the code was less lines and cleaner looking. For the NegaMax /w AB I just followed the Wikipedia page’s [pseudocode code](https://en.wikipedia.org/wiki/Negamax), it was however missing a crucial point of when to return the move (I also had a dumb moment of forgetting how recursion works :/ ), so I also used this tutorial to help debug my issue [Chess Engine in Python - Part 14 - Nega Max and Alpha Beta Pruning - Eddie Sharick](https://www.youtube.com/watch?v=5UVwksLYAKI&t=892s). This is seen in the ‘IF’ statement labeled “##Tutorial##”. This function is called in *AIPlayer.java -> playTurn()* and is initialized with a depth of 4.

*/\*\*  
 \* https://www.youtube.com/watch?v=5UVwksLYAKI&t=892s  
 \* Chess Engine in Python - Part 14 - Nega Max and Alpha Beta Pruning - Eddie Sharick  
 \* The above youtube tutorial was used to help debug negamax function.  
 \* NegaMax /w will try to find the maximum possible score for each player. It will evaluate each  
 \* terminal node, then negate the scores found and pick the maximum value from the nodes of that  
 \* branch.  
 \** ***@param*** *board Current instance of the game board  
 \** ***@param*** *depth Depth of how far the AI wants to look  
 \** ***@param*** *playerID PlayerID of what score is currently being evaluated (switches between 0 & 1)  
 \** ***@param*** *alpha lower/upper bound  
 \** ***@param*** *beta upper/lower bound  
 \** ***@return*** *best score  
 \*/*public int negamax(Board board, int depth, int playerID, int alpha, int beta){  
 // Check if we've reached the terminal node  
 if(depth == 0){  
 int multiplier = 1;  
 // If this is the opponent's score of the node, make it negative

// because they're trying to make us get the least score  
 // (Current Player) != AI's player ID  
 if(playerID!=this.playerID){  
 multiplier=-1;  
 }  
 return multiplier \* board.evaluateBoard(playerID);  
 }  
 // Get all VALID available moves - this will change as depth is decreased  
 Array<Move> availableMoves = board.getMoves();  
 // Set the negative infinity value - NegaMax looks for the highest value hence the negative infinity  
 int bestScore = -Integer.*MAX\_VALUE*;  
 // Go through all current available moves for the current board instance  
 for (int i = 0; i < availableMoves.size; i++) {  
 // Make a new 'Move' as to not permanently modify it until we know the best move  
 Move move = new Move(availableMoves.get(i));  
 // Make 'fake' move on the board to see future results  
 Board newBoard = board.makeMove(move, playerID);  
 */\*\* Invert the score bubbled up from child node below  
 \* If AI turn, find it's highest score  
 \* If its the opponents turn, whatever their best score is - is the worst score for the AI  
 \* so negate score to find it's highest value  
 \* Pass in:  
 \* - Board instance  
 \* - Next depth to search  
 \* - Player ID of opponents turn (goes back and forth from 0 or 1)  
 \* - Set the negation of alpha and beta for the next player's lower and upper bounds of  
 \* possible values that represent a window a node must score in based on it's parent node  
 \*/* int score = -negamax(newBoard, depth - 1, playerID ^1, -beta,-alpha);  
 // Delete 'fake' move that was placed on the board - do this to keep board at original  
 // when NegaMax is finished or onto the next instance  
 // state, depending on the current instance of the board  
 newBoard.deleteMove(move);  
 // If highest score found  
 if(score>bestScore){  
 // Set new highest score  
 bestScore = score;  
 // ## Tutorial ##  
 // Check if we're back at the root node - if so assign best

// move, if not keep digging.  
 if(depth == DEPTH){ // DEPTH==4  
 bestMove = move;  
 }  
 }  
 // Get new best score to check if next score is less than or equal to it - if so cut off  
 alpha = *max*(alpha, bestScore);  
 // Terminal node is outside of lower/upper bound so cut off that branch - no point to look at it  
 if(alpha >= beta){  
 break;  
 }  
 }  
 // Return highest score found  
 return bestScore;  
}

### Heuristic:

When I was first trying to develop the heuristic for evaluating the Connect 4 pieces, I was unsure what was the best way to go about it, I looked up what others did and they all seemed to have the same general idea by using a ‘window’ array to evaluate the positions. I used this tutorial to make the evaluation function [How to Program a Connect 4 AI (implementing the minimax algorithm)](https://www.youtube.com/watch?v=MMLtza3CZFM&t=4738s), I found this same logic on multiple sites but this was my main reference. The *evaluateBoard* function has the logic to scan the game board, it’ll make all combinations of the game space of size 4, build a “window” array of these sets, then pass the array to the function *windowCount* to tally up the results. I kept the heuristic values basically the same as the ones in the tutorial as it seemed to make the AI beatable but only if the player is paying attention – so its not an expert but a novice is how Id probably classify it. These functions are in *Board.java*.

*/\*\*  
 \* https://www.youtube.com/watch?v=MMLtza3CZFM&t=4738s  
 \* How to Program a Connect 4 AI (implementing the minimax algorithm) - Keith Galli  
 \* The above tutorial was used to build the heuristic for the game.  
 \* This function will take in player id it's evaluating and then go through the entire game board  
 \* and tally up the points for the pieces it finds in that board instance. 'windowCount' function  
 \* is used to help tally up these scores  
 \** ***@param*** *player Player ID function is evaluating  
 \** ***@return*** *Return board score  
 \*/*public int evaluateBoard(int player){  
 // Length of the array 'window' being used to find sections of the game board  
 int windowLength = 4;  
 // Initial score of the board  
 int score = 0;  
 // ID of the player the function is evaluating  
 int playerID = player;  
  
 // Center preference  
 // Because the center column of the board is shown to yield a greater chance of  
 // winning the game, any pieces found here are given a point boost.  
 // Initialize array to store center column - column will have 6 row values  
 Move[] centerColumn = new Move[6];  
 // Build column to scan  
 for (int j = 0; j < 6; j++) {  
 // j = row , 3 = center column  
 Move move = board[j][3];  
 centerColumn[j] = move;  
 }  
 // Counter for the evaluating player's game pieces  
 int countCenter = 0;  
 // Go through column  
 for(Move move: centerColumn) {  
 // Make sure there is a move there  
 if (move != null) {  
 int currentID = move.getPlayerID();  
 // Make sure the player ID is the evaluating player's then increment counter  
 if (playerID == currentID) {  
 countCenter++;  
 }  
 }  
 }  
 // Add counter booster the score  
 score += (countCenter \* 3);  
  
 // Horizontal  
 // Get a board row  
 for(Move[] row: board){  
 // Increment the window array starting from 0 to column 3  
 for(int i=0; i< row.length-3; i++){  
 // Slice the row for the window  
 Move[] window = Arrays.*copyOfRange*(row,i, windowLength+i);  
 // Evaluate window and get score  
 score += windowCount(window, playerID);  
  
 }  
 }  
  
 // Vertical  
 // Generate the boards columns  
 for(int g=0; g<7; g++) { // 7 columns we need to go through  
 Move[] column = new Move[6]; // column will have 6 row values  
 // Build the first column to scan  
 for (int j = 0; j < 6; j++) { // starting row  
 Move move = board[j][g];  
 column[j] = move;  
 }  
 // Scan first column with window array  
 for (int i = 0; i < 3; i++) { // column  
 Move[] window = Arrays.*copyOfRange*(column, i, windowLength+i);  
 // Evaluate window and get score  
 score += windowCount(window, playerID);  
 }  
 }  
  
  
 // Diagonal Positive - bottom left to top right  
 // Row position - 0 to 2  
 for(int i=0; i<3; i++){  
 // Column position = 0 to 3  
 for(int j=0; j<4; j++){  
 // Reset window array  
 Move[] window = new Move[4];  
 // Build the window  
 for(int index=0; index<4; index++) {  
 window[index] = board[i + index][j + index];  
 }  
 // Evaluate window and get score  
 score += windowCount(window, playerID);  
  
 }  
 }  
  
 //Diagonal negative - top left to bottom right  
 // Row position - 0 to 2  
 for(int i=0; i<3; i++){  
 // Column position = 0 to 3  
 for(int j=0; j<4; j++){  
 // Reset window array  
 Move[] window = new Move[4];  
 // Build the window  
 for(int index=0; index<4; index++) {  
 window[index] = board[i+3 - index][j + index];  
 }  
 // Evaluate window and get score  
 score += windowCount(window, playerID);  
  
 }  
 }

// Return final score of the board  
 return score;  
  
}  
  
*/\*\*  
 \* Takes in a 'window' array and the player ID to be evaluated and returns a score based on  
 \* what criteria was found in the window  
 \** ***@param*** *window Move array  
 \** ***@param*** *playerID ID to be evaluated  
 \** ***@return*** *score  
 \*/*public int windowCount(Move[] window, int playerID){  
 int count =0; // Evaluating player counter  
 int nulls =0; // Empty square counter  
 int score = 0; // Evaluating player's score  
 int enemyScore = 0; // Opponent counter  
  
 for(Move move: window){  
 if(move != null){  
 int currentID = move.getPlayerID();  
 // If move in window is not null then get the Player ID and see if it matches the  
 // current evaluating player, if so, increment 'count' else increment enemy's score count  
 if (playerID == currentID) {  
 count++;  
 }  
 else {  
 enemyScore++;  
 }  
 }  
 // No Move object, therefore, empty square - increment empty square counter  
 else{  
 nulls++;  
 }  
 }  
 // Tally up the score depending on the state of the window array found above  
 if(count==4){ // Winning move found  
 score += 100;  
 }  
 else if(count==3 && nulls==1){ // 3 player spaces found and 1 empty found  
 score += 5;  
 }  
 else if(count==2 && nulls==2){ // 2 player spaces found and 2 empty found  
 score += 1;  
 }  
  
  
 if(enemyScore==3 && nulls==1){ // 3 enemy spaces found and 1 empty found  
 score -= 4;  
 }  
 else if(enemyScore==2 && nulls==2){ // 2 enemy spaces found and 2 empty found  
 score -= 1;  
 }  
  
 // Return overall score found of window  
 return score;  
}

## Bugs:

I’ve tested the following and have yet to find any bugs!

* Skipping Intro conversation – starting as Ant and Spider
  + Both win conditions appear as expected
  + Both lose conditions appear as expected (AI’s win character displayed)
  + Hit ESC and “Retry” goes back to Character Select screen as intended
* Watching Intro conversation – starting as Ant and Spider
  + Both win conditions appear as expected
  + Both lose conditions appear as expected (AI’s win character displayed)
  + Hit ESC and “Retry” goes back to Character Select screen as intended

# 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 3 🡪 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:

A, W, S, D – To move spider left, up, down and right to move the spider

SPACE – to fire webbing

ESC – to go to Pause Menu

Mouse Click – mouse click when at either “You Died” or “You Win!” screens to

Continue.

### Description:

The Evil Centipede is back and badder than ever in this epic conclusion of *Bug Wars*! Face-off against the Centipede for the last time but be careful! It has new tricks up it’s all those sleeves.



Figure 1 Assignment 3-Game 2 Final Boss

Like in the previous game, collect the web sacs as the appear on the ground and fire your webbing at the centipede. The web sacs have three stages and only when they’re at stage 3 can you pick them up:

A picture containing text, first-aid kit, scoreboard, sign

Description automatically generated

Figure 2 Web Sac levels

You can only hold 3 webs at a time, so make sure you plan out your attacks! The Centipede has 4 main attacks:

A screenshot of a video game

Description automatically generated with medium confidence

Figure 3 Swarm Shot

**Swarm Shot**:

Like in the first game, the swarm shot will hurt the Spider for 5 pts of damage.



Figure 4 Lunge Attack

**Lunge**:

If the player gets too close the Centipede it will lunge at the player doing 10 pts of damage.

A screenshot of a video game

Description automatically generated

**Tail Swipe**:

Once the centipede’s health starts to go down, it will become more threatened and start attacking the player with it’s tail doing 10 pts of damage. Only the last body with the tail pincers will cause damage, other parts of the tail can be touched with no damage to player taken.

A red soda can

Description automatically generated with low confidence

Figure 5 Tail Swipe attack

Figure 6 Beam attack

**Beam**:

The centipede’s ultimate move! Once the centipede becomes desperate it will start firing this beam and will do 20 pts of damage :o .

To win the game, get the centipede’s health to zero and then you’ll see a special ending.

Player Tip:  
 Its helpful to let the web sacs grow to level 3 before making the centipede take major damage – the more damage he takes the harder hes going to get!

## General Overview of Development:

### Description:

\*\*\* SPECIAL NOTE \*\*\*  
 For this game I didn’t do the learning algorithms, I really did not vibe with this section of the book at all. I struggled a lot with trying to find an idea with what to implement and how. In the end I wasted too much time with trying to figure out an idea and ended up not doing the learning algorithm unfortunately. I understand that I will probably get a 0 in this section which is why I made this ‘note’ so you don’t waste time reading through it. I did want to finish my game, so instead I made a FSM and used ‘Utility Theory’ to help my boss AI make decisions. So if you’re interested in this final part of the game please continue reading, otherwise, thank you for your time and I overall really enjoyed this course.. except for this last part.

Cheers!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Development for this final game was taken from Assignment 1 and added onto (see dataflow chart *COMP452 Assignment 3 Game 2.png* for added on classes). For instance, the Spider and Web Shooters are the same but the Centipede was heavily added onto and modified. Like the first game, the game *Binding of Isaac* was the inspiration for this last one, particularly when Isaac faces off against Mega Satan (Figure 3). The Evil Centipede floats off from the platform the spider is on and shoots projectiles or lunges at the player – which is some of the basic mechanics of Mega Satan. A screenshot of a computer

Description automatically generated with low confidence

Figure 7 Left: Mega Satan boss fight from Binding of Isaac, Right: Centipede boss fight

The centipede has four main states: Patrol, Lunge, Tail Swipe and Beam. Each of these states are initialized and put in a Dictionary with a value that acts as the weight/priority of that state. For example:

*stateDictionary {*

*( Patrol : 1 )  
 ( Lunge : 0 )  
 ( TailSwipe : 0.6)  
 ( Beam : 0.2 )*

*}*

In this scenario, the Patrol state would be called because it has the highest priority, which is 1, this could be due to the centipede’s health, player position or an attack may have just played out. These priority weights are calculated with Utility Theory[[2]](#footnote-2), the theory essentially takes a snapshot of the environment and assigns a value based on these results, but these values are based off a *desire*. In the case of the centipede, the *desire* is represented as an *attack desire*, meaning that the lower the centipede’s health the harder it’s going to try to kill you. The formula used is taken from David Graham’s book *Game AI Pro* (pg 123):

A picture containing diagram

Description automatically generated

*Hp, minDmg,* and *maxDmg* are the same as Graham’s but the *a-value* is based off of the player’s current position along the x-axis and/or y-axis of the game. The formula will then have a result that is from 0 to 1. Figure (4) below shows an example of the chances a certain attack will occur (theses attacks are further discussed below).  
Timeline

Description automatically generated

Figure 8 Probability example of attack rates

For example, if the AI wanted to see whether it wants to do a Beam attack, a utility value would be calculated as such:

Evil Centipede Health: 70  
minDmg: 5  
maxDmg: 20  
Spider position: (280, 100) << Puts spider in Beam attack section

To find the a-value we take the Spider’s x-coordinate and divide it by the max length of the x-axis:  
a-value = 280/364 = 0.76923..

We then calculate the Beam utility with Graham’s formula:

Beam Utility = *max*( -1.794.. , 0) = 0

Because the Beam Utility is 0 there is a zero chance it will fire, the main contributor to this is the centipede’s health is too high. If we run the same calculation again but with the centipede’s health at 40%, we then get a value close to 0.416667. There is now a near 42% chance the Beam will be picked from the state pool.

The idea with using weights to decide attacks was to increase the difficulty of the centipede end-game boss as the player is succeeding in killing it. I wanted to create a boss AI that was challenging and force the player to devise a strategy in defeating it. I didn’t want the boss to be easy because I feel this takes away from that feeling of accomplishment when the player has finally over come a major boss.

I made a Youtube video of me playing the boss fight:

<https://youtu.be/tpFXn6EOSB0>   
(Please let me know if this doesn’t work and you want to see it, Im using audio that isn’t mine so YouTube may flag it)

My strategy is making sure I don’t drastically lower the boss’s health too fast else it will start firing the Beam attack – which does 20 pts of damage – so it’s a major attach that the player wants to avoid. I also make sure I know where the Web Sacs are spawning and don’t attack until there are more generated. The trick it to get the centipede’s health to 60%, have 3 webs following you and have the 3 Web Sacs ready for pick up before you start your final assault. At the end of the video you don’t see it but I fire off my last 3 web shots when the boss is firing his beam attack.

### Assets Borrowed:

All assets for this game are found in:  
 COMP452 > assets > Assignment3 > Game2

* In game sound effects from the game *Binding of Isaac*  
  <https://www.sounds-resource.com/pc_computer/bindingofisaac/sound/4368/>  
  Boss\_Spit\_Blob\_Barf.mp3  
  Death\_Burst\_Small\_0.mp3  
  fetus\_flyingwhoosh.mp3  
  isaacbosswin.mp3  
  leech2.mp3  
  Maggot\_Enter\_Ground\_1.mp3  
  Monster\_Roar\_2.mp3  
  plop.mp3
* Game music “Tool - Vicarious (8 Bit)” from pandoozles  
  <https://www.youtube.com/watch?v=uj_fzTGOmqw&list=PLO12u-VvvFBiwwyf2w27qPC669-USDWY4&index=5>
* OcenAudio used to make Centipede’s voice by altering these *Binding of Isaac* files  
  <https://www.sounds-resource.com/pc_computer/bindingofisaac/sound/4368/>  
  bird\_death0.mp3  
  bird\_death1.mp3  
  bird\_death2.mp3
* Utility Theory from:  
  David “Rez” Graham, *Game AI Pro - Chapter 9*, http://www.gameaipro.com/GameAIPro/GameAIPro\_Chapter09\_An\_Introduction\_to\_Utility\_Theory.pdf

## AI:

### Utility Theory:

*Called in Patrol.java*

This function is the Utility Theory explained above that is used to determine the likely probability of preforming the TailSwipe or Beam attack. The ‘If’ statements are used to gatekeep if an attack is appropriate for the time in the game (whether the boss’s health is low enough) and the boss is in the correct position. For instance, having the major beam attack going off when the centipede is only at 70% health makes the game too difficult (this would be something to adjust if there was a game difficulty option); as for position, if the boss is stopped at the very left hand of the screen and the player quickly moves the right side, this could trigger the beam attack which wouldn’t be appropriate because it would have a 0% chance of hitting the player.

*/\*\**

*\* This is the heuristic that is used to calculate which attack the boss should do next.*

*\* It uses Utility Theory to calculate the threat level that the boss is at. Example:*

*\* If the boss is low health the threat level will be high to try and defeat the player.*

*\*/*

public void Heuristic(){

float spidX = spider.getPosition().x;

float healthCheck = boss.getHealth();

// a-value used in the utility theory algorithm, both TailSwipe and

// Beam attacks will evaluate this differently.

float xValue = spidX/364;

// beam and tail utility hold the weight at which these should of should not be used

// based on the boss's health and player position

float beamUtility = Math.*max*(Math.*min*((1-((healthCheck-5)/(20-5)))\*(1-xValue)+xValue,1),0);

// We want the inverse percentage of the X-axis for the tail swipe attack because it comes from the left side of the screen – so this attack would be favored more in the area

xValue = Math.*abs*(1-xValue);

float tailUtility = Math.*max*(Math.*min*((1-((healthCheck-5)/(20-5)))\*(1-xValue)+xValue,1),0);

// Check if boss health is low enough to be considered as an attack and that the spider

// is in approximation of the beam

if (healthCheck <= healthFinal && boss.getBody().getPosition().x <= spidX+40 && boss.getBody().getPosition().x >= spidX-40){

boss.attackSelector.put(boss.beam,beamUtility);

}

else{ // If beam is not considered then set tail probability high

tailUtility = 0.85f;

}

// Check if health is low enough to be considered for attack and make sure the tail isn't

// already running

if(healthCheck <= healthMid && !boss.tailRunning){

boss.attackSelector.put(boss.tail,tailUtility);

return;

}

// else keep patrolling

boss.attackSelector.put(boss.patrol,1f);

}

### Patrol:

*Called in Patrol.java*

Patrol is the base state for the centipede and has a few functions, it calls the function to move the boss (just the arrive behavior like in Assignment 1), fire the swarm shot (same as Assignment 1) and decides the next state to move into.

@Override  
public void DoCheck() {  
 super.DoCheck();  
 // boss.getY()-100 == max Y distance the spider has to be in for lunge

// attack, and boss.getX +(-)40 == if player is in range of the boss’s

// head. If found true do lunge attack.  
 if(spider.getPosition().y < boss.getY() && spider.getPosition().y > (boss.getY()-100) &&  
 spider.getPosition().x > (boss.getX()-40) && spider.getPosition().x < (boss.getX()+40)){  
 stateMachine.ChangeState(boss.lunge);

}

// Lunge attack is not ideal find another ideal attack  
 else{  
 Heuristic();  
 }  
  
 // Get state to run based on state weight  
 float keyValue = 0;  
 BossState stateToCall = null;

// Go through dictionary and find the highest weighted state  
 Enumeration<BossState> keys = boss.attackSelector.keys();  
 while(keys.hasMoreElements()){  
 BossState checkState= keys.nextElement();  
 // Get the state will the highest priority, save it and compare to the rest  
 if(keyValue < boss.attackSelector.get(checkState) && checkState!=this){  
 stateToCall = checkState;  
 keyValue = boss.attackSelector.get(checkState);  
 }  
  
 }  
 // Change state to the one with the highest priority  
 if(stateToCall != null ){  
 stateMachine.ChangeState(stateToCall);  
 }

// Else, keep running Patrol state

}

### Lunge:

*Called in Lunge.java from Centipede.java*

As described above, the Lunge attack will move the head down to strike at the spider player if it is in close proximity to the head. Once the head has reached the max distance it will be called back to it’s original position.

*/\*\*  
 \* Move Centipede head to attack spider  
 \*/*public void lungeAttack(){

// Set a new position for the head to travel to (go down the y-axis by

// 80 coordinates  
 Vector2 attack = new Vector2(0,body.getPosition().y-80);

// Get the direction  
 attack.nor();

// Scale for speed, note that negative is used because we’re going

// down the y-axis  
 attack.scl(-maxSpeed);

// Apply velocity  
 body.setLinearVelocity(attack);  
}  
  
*/\*\*  
 \* Lunge attack has reached max depth so return to original position  
 \*/*public void lungeAttackReturn(){

// Check if head is passed the y-axis limit (310) that we want the

// the attack to go down to  
 if(body.getPosition().y < 310){

// Set a new position for the head to travel to  
 Vector2 attack = new Vector2(0,body.getPosition().y+180);

// Get the direction  
 attack.nor();

// Scale for speed  
 attack.scl(maxSpeed);

// Apply velocity  
 body.setLinearVelocity(attack);  
 }

// Head has started to move back to original position, just snap it

// back because it is close enough

else{  
 body.setTransform(body.getPosition().x, 310, 0);  
 }  
  
}

### TailSwipe:

*Called in TailSwipe.java from Centipede.java*

The Tail Swipe attack has a few parts to it, theres the main call to send out the body to attack the player, then there are checks to make sure its returned to outside of the screen.

*/\*\*  
 \* Run the tail swipe attack by moving the 'butt' body along the x-axis of the player's current  
 \* position. Function gets called multiple times until the tail has reached the end of the game   
 \* field.  
 \** ***@param*** *spider  
 \*/*public void tailAttack(Body spider){  
 // Check if attack has been called the FIRST time, if so, set it’s

// staring position based on the player’s position  
 if(tailRunning==false){

// Set the ‘butt’ body start position based on player’s position  
 butt.setTransform(0, spider.getPosition().y, butt.getAngle());

// Save the starting y-coordinate to make sure the tail returns in a

// linear fashion (doesn’t go off diagonally)  
 tailReturnY = spider.getPosition().y;

// Set to true so first instance variables are only used.  
 tailRunning = true;  
 }  
 // Set end of map destination  
 Vector2 temp = new Vector2(1216,0);

// Give the vector a proper direction  
 temp.nor();  
 temp.scl(maxSpeed);

// Move at full speed  
 butt.setLinearVelocity(temp);  
  
}

*/\*\*  
 \* Put tail to regular position  
 \*/*public void tailBasePosition() {

// Set coordinate position to move tail out of screen  
 Vector2 temp = new Vector2(-900,0);  
 temp.nor();  
 temp.scl(maxSpeed);  
 butt.setLinearVelocity(temp);

// If tail location is outside of screen then set checks to false

// since attack is now fully complete  
 if(butt.getPosition().x <= -100){  
 retractTail();  
 tailRunning=false;  
 }  
}

*/\*\*  
 \* Centipede update for Assignment 3 Game 2.  
 \* This update is used to call whatever behavior is currently running and to make sure the tail  
 \* goes back to its position outside of the game screen  
 \*/*public void updateBehavior(){  
 bossState = stateMachine.getCurrentState();  
 bossState.Update();  
 x = body.getPosition().x; // Will be the center of the body  
 y = body.getPosition().y;  
 // Check butt  
 float varX = butt.getPosition().x;  
 // Tail hit end of screen so set tailHit to true and retract it  
 if(varX>=355f && tailHit==false){  
 retractTail();  
 tailBasePosition();  
 }

// TailSwipe attack is done so keep retracting it  
 else if(tailHit==true && tailRunning==true){  
 tailBasePosition();  
 }

// Tail is outside of screen, set to base position until called again  
 if(varX<=-100f){  
 butt.setTransform(-90, tailReturnY, butt.getAngle());  
 }  
}

### Beam:

*Called in Beam.java from BeamObject.java*

The Beam attack is the boss’s ultimate attack, it’s design is slightly different than the other attacks which is why it has its own class. Instead of a body object like everything else in the game, it’s a Rectangle shape object and checks the player’s location against it’s own area to determine if a collision happened. This was done since box2D’s collision library and body objects themselves, don’t like being altered after they’re created. Since I wanted the Beam to shoot down – effectively grow a rectangle downwards to imitate an actual beam – I wasn’t able to use box2D or a body object.

*/\*\*  
 \* Object class that initializes the Centipede's beam shot, moves it, renders it and checks  
 \* for collision. This simple collision was done instead of Box 2d because the beam shape will  
 \* change shape was it is being shot - box2D does not like bodies changing after they've been  
 \* initialized.  
 \*/*public class BeamObject {  
  
 private AssetManagerA3G2 assetMgr;  
 private TextureRegion end, beam;  
 private Rectangle rectangle;  
 public float startingX, startingY, height;  
 public boolean beamRunning = false;  
 private int speedOfBeam = 4;  
 private Spider spiderPlayer;  
  
 public BeamObject(AssetManagerA3G2 asstMgr, Spider spiderPlayer){  
 this.assetMgr = asstMgr;  
 this.spiderPlayer = spiderPlayer;  
 end = assetMgr.getBeamEnd();  
 beam = assetMgr.getBeam();  
 rectangle = new Rectangle(-90,100,80,60);  
 height = 20;  
  
 }  
  
 */\*\*  
 \* Check if the beam has hit the spider player or not  
 \** ***@param*** *spider  
 \*/* public void checkCollision(Body spider){  
 float spidX = spider.getPosition().x;  
 float spidY = spider.getPosition().y;  
 float rectX = rectangle.x;  
 float rectY = rectangle.y;

// Get the player’s position, +(-) 16 pixels to get the edge (position

// is the center of spider), then compare to rectangle’s current x and

// y positions.

// Check spider’s left edge position, rectX+80 == width of beam  
 if((spidX-16) <= (rectX+80) && (spidX-16) >= (rectX) &&  
 (spidY-16) <= (rectY+ height) && (spidY-16) >= (rectY)){  
 // Collision  
 spiderPlayer.removeHealth(20);  
 return;  
 }

// Check spider’s right edge position  
 if((spidX+16) <= (rectX+80) && (spidX+16) >= (rectX) &&  
 (spidY+16) <= (rectY+ height) && (spidY+16) >= (rectY)){  
 // Collision  
 spiderPlayer.removeHealth(20);  
 }  
  
 }  
  
 */\*\*  
 \* Beam has been fired so move it down starting from the position of the centipede's head  
 \*/* public void moveBeam(){  
 startingY -= speedOfBeam;  
 height += speedOfBeam;  
 rectangle.set(startingX, startingY, 80, height);  
  
  
 }  
  
 */\*\*  
 \* initialize the starting coordinates for beam attack  
 \** ***@param*** *centipede  
 \*/* public void readyBeam(Body centipede){

// Offset for Centipede’s mouth position  
 startingX = centipede.getPosition().x-30;  
 startingY = centipede.getPosition().y-50;  
 beamRunning = true;  
 }  
  
 public void renderBeam(SpriteBatch batch){  
 batch.draw(beam, startingX, startingY, 70, height);  
 batch.draw(end, startingX-9, startingY-8, 90, 30);  
  
 }  
  
 */\*\*  
 \* We're resetting the height so the beam will be small and start at the top when fired again  
 \*/* public void resetHeight(){  
 height = 20;  
 }  
  
 public Boolean beamRunning(){  
 return beamRunning;  
 }  
  
 public float getY(){  
 return rectangle.getY();  
 }  
  
  
}

## Bugs:

I don’t believe there are any bugs left to smoosh, there were a few during development, such as getting the tail to return properly to it’s base position. Although there is room for improvement, everything seems to run as intended anyway.

Checked:

* Tail return and not refiring when attack first starts
* Centipede head moves/stops as intended
* Beam fires as intended
* Rendering of assets seem to be in proper order
* FSM seems to run attacks in proper order based on priority weight
* Pause and both end scenes appear and work

1. https://en.wikipedia.org/wiki/Connect\_Four [↑](#footnote-ref-1)
2. David “Rez” Graham, *Game AI Pro - Chapter 9*, http://www.gameaipro.com/GameAIPro/GameAIPro\_Chapter09\_An\_Introduction\_to\_Utility\_Theory.pdf [↑](#footnote-ref-2)