COMP 452

GAME DESIGN DOCUMENT

ASSIGNMENT 1

Jody Pequin

Contents

[How to Play: 1](#_Toc117940059)

[Run 1](#_Toc117940060)

[Description: 1](#_Toc117940061)

[Controls: 2](#_Toc117940062)

[General Overview of Development: 3](#_Toc117940063)

[Description: 3](#_Toc117940064)

[Frameworks/Software Used: 6](#_Toc117940065)

[Assets Borrowed: 6](#_Toc117940066)

[AI Implementations: 8](#_Toc117940067)

[Arrive: 8](#_Toc117940068)

[Projectile (Spider’s Web Shooter): 10](#_Toc117940069)

[Flocking (Centipede’s Burst Shot): 11](#_Toc117940070)

[Seek 15](#_Toc117940071)

[Bugs 16](#_Toc117940072)

[Bug 1 16](#_Toc117940073)

[Bug 2 16](#_Toc117940074)

[Bug 3 17](#_Toc117940075)

[Bug 4 17](#_Toc117940076)

# How to Play:

Run:

To start the game, double-click on the jar file called “desktop-1.0.jar”. When the main menu screen opens mouse click to start the game intro, when the intro is done and spacebar is pressed, the game will begin.

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

## Description:

The user will play as “Spider Dude” with the goal to eliminate the “Evil Centipede” that threatens the spider lair.



Figure 1: (LEFT) Spider Dude (RIGHT) Evil Centipede

To defeat the enemy, the player must try to collect Web Sacs, an in-game pickup that will allow the player to shoot Webs at the enemy to decrease its health. There are 3 stages of the Web Sacs (Figure 2) and only when the Web Sac has reached level 3 is the player able to pick up a web to shoot at the enemy. The Web Sacs spawn in different places on the map each time one is picked up, only 3 will spawn at one time and will take time to regenerate again.

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

Description automatically generated

Figure 2: Different levels of the Web Sac Pickup

The player can hold ONLY 3 webs at a time and must time their shots to hit the centipede to avoid taking damage. Each web shot will do 10 points of damage to the enemy. The Evil Centipede will inflict damage to the player if it makes contact to the player (10 points of damage inflicted) or if one of the Centipede’s Burst Shot projectiles (Figure 5) hits the player (5 points of damage will be inflicted). The UI at the top of the screen will show the current health points both the Spider Dude and Evil Centipede have:

A screenshot of a video game

Description automatically generated with medium confidence

Figure 3: UI Health Bars of the player and enemy

To win, avoid the Evil Centipede and get its health to zero using your web shots and save your spider lair!

Player Tip:  
 If the Centipede is on your tail, run away to get some distance and shoot at it when it fires its burst shot!

## Controls:

Move Player: W (Up), A (Left), S (Down), D (Right)

Fire Projectile: SPACEBAR

Pause Game: ESC (Hit ESC again to resume game)  
 Left-Mouse Click to select Pause Menu button

# General Overview of Development:

## Description:

Welcome to *Bug Wars*! This will be the main game program for all 3 assignments. Each sprint (assignment) will add on to the game so at the end of the course this should have all/most functions of an actual game, for instance, the current Main Menu screen is very plain but for the next sprint it will be updated visually and allow the player to pick which game (assignment) they would like to play. I know these and the other features implemented are not technically necessary for the assignment but its just extra features I’d like to learn – time providing.

Much of the game design is derived from the game *Binding of Isaac*, it’s a straightforward shooter game in that the player will shoot at the enemy until it’s dead or they’re dead – then repeat. A couple of the mechanics such as the Web Projectiles that follow the player (Figure 4) and the Burst Shot the Evil Centipede (Figure 5) does, are a couple of the mechanics imitated from *Binding of Isaac*:



A picture containing grass

Description automatically generated

Figure 4: (LEFT) Projectiles that follow the player Isaac so he can shoot enemies  
 (RIGHT) Web Projectiles that follow Spider Dude

Figure 5: (LEFT) Mother's Heart boss that shoots a swarm of projectiles at Isaac  
 (RIGHT) Evil Centipede Burst Shot

Chart, box and whisker chart

Description automatically generatedThe code has been structured to be as modular as possible with the main java file being *Assignment1.java*. This file is considered “brain” of this first game, it will call most of the other classes for information and/or actions. Figure 6 shows a basic diagram of the general flow process, the arrows indicate that the particular class has requested information and/or action. There are small interconnected clusters but these mainly pertain to the Spider player object and Centipede NPC object, for instance, *Web* and *WebShooter* objects only exist if the *Spider* exists, same with the *Centipede* and its *SwarmShot* and *Projectile* objects. There is not point on creating objects if they’re not needed. Some of the generic classes, like *PauseMenu*, *FadeScreen* and *EndState* will be updated in the second assignment to be able to work with any of the games the player selects.

Figure 6: Class Flow Diagram. PNG version located in submitted documents called "ClassFlowDiagram\_1.png"

Diagram

Description automatically generated

Figure 7: Centipede construction

The Centipede’s body is made up of several bodies all connected by distance joints (see *Centipede.java* in *initDistanceJoint* function). The head of the Centipede has the seek algorithm applied to it and the butt of the Centipede has a dampener to slow that particular body down. This was done to give it a *snake* like look as it crawls throughout the level (Figure 7). Without this, all of it’s bodies would cluster together.

The Web Sacs random coordinate generation is just a simple use of Math.random:

// Set our body's starting position in the world  
bodyDef.position.set((int)(Math.*random*()\*592 + 8), (int)(Math.*random*()\*432 + 8));

We take the max value of X and Y in the area we want the generation, then minus 2\*Size of object and times that by Math.random (which generates a value from 0 to 1). Then we add the size of object so in case 0 is generated the Web Sac object will still get displayed on the screen. For example:

Our world is 608x448 pixels and our Web Sac is 8x8  
 X axis we want the Web Sacs to generate = 608 – (2\*8) = 592  
 Y axis we want the Web Sacs to generate = 448 – (2\*8) = 432  
  
 If Math.random == 0 { 0 \* 592 + 8 = 8} // Our new X value  
 If Math.random == 0 { 0 \* 432 + 8 = 8} // Our new Y value

OR

If Math.random == 1 { 1 \* 592 + 8 = 600} // Our new X value  
 If Math.random == 1 { 1 \* 432 + 8 = 440} // Our new Y value

This way the object will always stay visible on the playing field, we don’t want an object being spawned at the max X or Y values else its visibility will be cut.

All assets are found in

## Frameworks/Software Used:

* Android Studio – IDE  
  This IDE was chosen because it was recommended on LibGDX’s main website and it is also used in some classes at MacEwan University (my home university).
* LibGDX – Main game development framework  
  I’ve never used LibGDX before but from the positive reviews I found online, as well as the number of tutorials available, is why I chose this framework.
* Procreate and GIMP used for art  
  Procreate is just the art program I had available on my iPad and GIMP was used as a free alternative to Photoshop.
* Tiled software  
  Used to easily create a map with my terrain assets.
* OcenAudio used for audio manipulation  
  Audacity was the most popular software I found recommended online but I found it difficult to use for the purpose that I wanted. As an alternative I found OcenAudio and it was user friendly – which is what I was mainly looking for because this was just a small feature I wanted (explained below in Assets Borrowed).

## Assets Borrowed:

* Gaming text downloaded from <https://www.dafont.com/retro-gaming.font>
* Okami sound bites from <https://www.sounds-resource.com/playstation_2/okami/sound/7689/>
* Bug Wars logo design based on *Beast Wars* logo - <https://thefwoosh.com/2013/04/top-five-transformers-beast-wars-figures-that-need-another-look/>

OcenAudio Sound Manipulation:

The game *Okami* was used as the inspiration for the character’s voices in Bug Wars. The audio file “Okami\_09659.wav”, taken from the game *Okami*, was used as a starting point for creating the Spider’s voice in the main Assignment 1 intro (see *SpiderIntro.java*). This audio was taken and distorted to give the Spider his deep voice. Figure 8 shows the main audio file and Figure 9 shows the distorted audio version.

Chart

Description automatically generated

Figure 8: Original Okami sound file Okami\_09659.wav

Background pattern

Description automatically generated

Figure 9: Distorted Okami\_09659.wav sound file

# AI Implementations:

Arrive: Web.java

The Spider’s web shots were made to specifically to showcase the arrive AI. There are 2 different Box2D bodies used:

smallRadii – Body sensors stored in radiiBodies array in *WebShooter.java*  
 web – Body sensors made in *Web.java* & stored in webs array in *WebShooter.java*

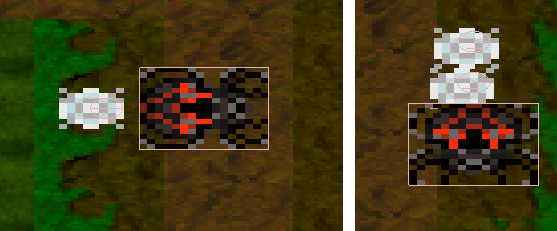
When the player walks across a Web Sac pickup (providing its at its level 3), a smallRadii will be following behind the character, this acts as the target for the Web object. These targets were made with a certain distance apart so it would look like a chain of webs, therefore giving the user an easy visual representation of the number of ‘bullets’ they would have to fire off.  
  


Figure 10: smallRadii and web body objects following behind the user after the Web Sac has been picked up

The main *arrive* AI code is found in *Web.java* in the function *followPlayer()*, which is continuously called in *WebShooter.java* in the *render* method. The green text in the code below explains the arrive AI implementation and is derived from Section 3.3 in the Textbook.

public void followPlayer(){

// When a web object is created it is assigned a radii target that is

// attached to the player  
 float x = radii.getPosition().x;  
 float y = radii.getPosition().y;

// Current web object  
 float x1 = web.getPosition().x;  
 float y1 = web.getPosition().y;  
 // The time we want to take to get to the target  
 float timeToTarget = 0.1f;

// Make coordinates proper Vector2s  
 Vector2 radiiPos = new Vector2(x, y); // Make coordinates proper Vector2s  
 Vector2 currentPos = new Vector2(x1, y1);  
  
 // Get displacement vector from enemy to target  
 Vector2 newDirection = radiiPos.sub(currentPos);

// Save vector in a temp variable  
 Vector2 temp = newDirection;

// Get velocity vector variable ready  
 Vector2 vel;

// Get the length (distance) of the displacement vector from enemy

// to target  
 float distance = newDirection.len();

// targetRadii is the “arrival radius” that the textbook talks about

// when we need to really slow our acceleration/velocity down.   
 float targetRadii = 4f;

// Large radii is the distance to start slowing down  
 float largeRadii = 64f;

// temp variable holder for ‘speed’ just so it doesn’t accidently

// get changed  
 float targetSpeed;  
  
 // At target, have no velocity  
 if(distance < targetRadii){  
 result = temp.nor().scl(0);   
 }  
 else {  
 // Not at the large radius so move at max speed to get there  
 if (distance > largeRadii) {  
 targetSpeed = speed; << speed is the max speed we want to go

// Inside large radius, start slowing down  
 } else {  
 targetSpeed = speed \* (distance / largeRadii) ;  
 }  
 // Based on how far away we are, set the target velocity and direction  
 vel = temp.nor().scl(targetSpeed);   
 Vector2 currentVelocity = radii.getLinearVelocity();

// Check acceleration and scale it depending on how fast object is

// going  
 result = vel.sub(currentVelocity); << Result used in Render below  
 result.scl(1 / timeToTarget);  
  
 // Throttle acceleration if too fast  
 if (result.len() > maxAcceleration) {  
 result.nor(); << Get the direction  
 result.scl(maxAcceleration); scale to our desired maxAcceleration  
  
 }  
  
 }  
  
}

*/\*\*  
 \* Render the visual of the webbing  
 \*/*public void render(SpriteBatch batch){  
 web.setLinearVelocity(result); << Set the linear velocity

Projectile (Spider’s Web Shooter): WebShooter.java

Keeping track of the number of projectiles and firing them off is in WebShooter.java under *update()* and *fireWebbing()*. *Update()* controls the target radii bodies that position the Web bullets behind the character and rotates the target radiuses (*followPlayer* functionexplained above controls the Web object positions themselves).

\*\*NOTE\*\* Remember there are Target radii bodies and Web object bodies, the Target radii is what the Web objects look for, both bodies get talked about below.

*/\*\*  
 \* Set the velocity and vector of where the web shot is going, remove Web from arraylist of  
 \* available webs to fire and put it in the 'websFired' array for later body disposal  
 \*/*public void fireWebbing(){

// Get the first web shot that is attached to the player  
 Web fireWeb = webs.get(0);

// Add 'firedWeb’ to new list, this list will be run later to destroy these bodies  
 websFired.add(fireWeb);

// Remove web shot from webs attached to player  
 webs.remove(0);

// Set the ‘fired’ web state to ‘FIRE’, this is used in collision detection to make sure that the web shot has been intentionally fired. Without this, if the Centipede or Game Boarder were to touch the web shot a collision event would happen that was not brought on by the player.  
 fireWeb.current = Web.WebState.*FIRE*;

// Destroy and remove the target radii body  
 world.destroyBody(radiiBodies.get(0));  
 radiiBodies.remove(0);

// Setup local Vector2 variables for calculations in switch method  
 Vector2 temp;  
 Vector2 curr = fireWeb.getBody().getPosition();  
 Vector2 newPos;

// Get the player’s rotation (orientation) to know which way to shoot the projection. We want the player to shoot the projectile the way the head is facing  
 switch(body.getRotation()){  
 case(-90): // Shoot RIGHT

// Vector that we want our web shot to eventually get to, because we’re shooting RIGHT we want the web shot to go to the farthest screen point in the X origin which is 608. The other cases in this switch statement do the same but in their respective locations.   
 temp = new Vector2(608,body.getY() );

// Get the direction  
 newPos = temp.sub(curr);  
 newPos.nor();

// Scale it so the velocity of the web shot goes fast   
 newPos.scl(1000);

// ‘setResult’ will set the web objects velocity for when ‘setLinearVelocity’ is called in Web.java under the ‘render’ method  
 fireWeb.setResult(newPos);  
 break;  
 case(90): // Shoot LEFT  
 temp = new Vector2(0,body.getY() );  
 newPos = temp.sub(curr);  
 newPos.nor();  
 newPos.scl(1000);  
 fireWeb.setResult(newPos);  
 break;  
 case(0): // Shoot UP  
 temp = new Vector2(body.getX(),448 );  
 newPos = temp.sub(curr);  
 newPos.nor();  
 newPos.scl(1000);  
 fireWeb.setResult(newPos);  
 break;  
 case(-180): // Shoot DOWN  
 temp = new Vector2(body.getX(),0 );  
 newPos = temp.sub(curr);  
 newPos.nor();  
 newPos.scl(1000);  
 fireWeb.setResult(newPos);  
 break;  
 }  
  
}

Flocking (Centipede’s Burst Shot): Projectile.java

The Evil Centipede fires off a burst shot every 20 seconds, 100 projectile body sensors are created in *SwarmShot.java* in the *fireSwarm()* function. Each projectile runs the *flock* function that is the flocking AI and has an ID number from 0-99. This ID number is used below to distinguish a projectile object from the same one that is in the variable array ‘projArray’ because we don’t want a projectile to try and ‘separate’ from its self. Because all of the projectiles are created in the extact same location (see *fireSwarm()*), there are odd behaviors that happen with them. For instance, it can be sometimes seen that 2 projectiles can’t quite get separated from each other until they move farther away from the rest of the projectile heard. This “feature” is easily removed if ‘align’ and ‘cohesion’ are set to 0 weight (this actually gives the burst shot behavior a better look but because of the assignment requirements I kept them in).

*/\*\*  
 \* Flocking AI that groups 'separate', 'align' and 'cohesion' together to create the flocking  
 \* behavior.  
 \* These 3 AIs were derived from https://natureofcode.com/book/chapter-6-autonomous-agents/  
 \* I found this tutorial on flocking alot easier to understand versus the books description.  
 \** ***@param*** *projArray  
 \*/*public void flock(ArrayList<Projectile> projArray){  
   
 // Return the vectors so it’s easier to multiply different weights to get different behaviors  
 Vector2 sep = separate(projArray);  
 Vector2 ali = align(projArray);  
 Vector2 coh = cohesion(projArray);  
  
 // Multiply by a weight to get a different behavior  
 sep.scl(2);  
 ali.scl(.7f);  
 coh.scl(1);  
  
 // Add all the effects to acceleration, acceleration is then added to velocity in the *update* method  
 acceleration.add(sep);  
 acceleration.add(ali);  
 acceleration.add(coh);  
  
}  
  
*/\*\*  
 \* Separate AI that will push the projectiles away from each other and stop them from colliding  
 \** ***@param*** *projArray*

*\* @****return*** *current (returns either (0,0) vector or the new acceleration we want applied to velocity  
 \*/*private Vector2 separate(ArrayList<Projectile> projArray){  
 // Current gets returned and is automatically set a (0, 0) if there are no projectiles too close to one another   
 Vector2 current = new Vector2(0,0);  
 int count = 0;

// Go through all projectiles created in projArray  
 for (Projectile p: projArray){  
 // Get the distance between current projectile (projBob) and the others (p)  
 Vector2 distanceDiff = projBod.getPosition().sub(p.projBod.getPosition());  
  
 // Check if the 2 projectiles are at the SAME spot and NOT the same one (id)  
 if(distanceDiff.len() == 0.0 && id != p.id){  
 // Push projectiles away at random positions and velocity, because all the projectiles are created at the same spot, these random assigned directions and velocities will give the separate behavior more of a ‘burst’ look – going in all directions. Without this, the projectiles will ALL flow in the same general direction (like a flock) but my goal was to make smaller flocks in all directions.  
 velocity.x = (float)Math.*random*() \* 21 - 10;  
 velocity.y = (float)Math.*random*() \* 21 - 10;  
  
 }  
 // Check if the 2 projectiles are close together and NOT the same one (id), the length of 16.0 was used because each projectile’s diameter is 8f, so this will keep them about 2 projectiles apart.  
 else if(distanceDiff.len() < 16.0 && id != p.id){

// Save distanceDiff in new variable for updating  
 Vector2 getNewPos = distanceDiff;

// New vector that is in the direction away from the ‘other’  
 getNewPos.nor();

// Push the projectile away based on the distance, the strength of this repulsion decays with distance  
 getNewPos.x = getNewPos.x / distanceDiff.len();  
 getNewPos.y = getNewPos.y / distanceDiff.len();  
 current.add(getNewPos);  
 count++;  
 }  
 }  
  
 // Only do this if there are projectiles close together  
 if (count > 0){

// Get the average of the summed values   
 current.x = current.x/(float)count;  
 current.y = current.y/(float)count;  
 // New Desired acceleration and direction   
 current.scl(maxSpeed);   
  
 }  
  
 return current;  
  
}  
  
*/\*\*  
 \* Try to align the projectiles with their neighbors*

*\** ***@param*** *projArray*

*\* @****return***sumVelocities*(returns either (0,0) vector or the new acceleration we want applied to velocity  
 \*/*private Vector2 align(ArrayList<Projectile> projArray){

// Return sumVelocities with new acceleration or (0,0) if there are no neighbors  
 Vector2 sumVelocities = new Vector2(0,0);  
 int count = 0;  
 // Go through projectile list and add up all the different velocities  
 for(Projectile p : projArray){  
 // Get the distance between current projectile and the others  
 Vector2 distanceDiff = projBod.getPosition().sub(p.projBod.getPosition());  
  
 // Check if the 2 projectiles are close together and NOT the same one (id)  
 if(distanceDiff.len() > 10.0 && id != p.id && distanceDiff.len() > 0.0){

// Add velocity to the total velocities  
 sumVelocities.add(p.velocity);  
 count++;  
 }  
  
 }  
 // Only do this if there we’re close enough neighbors  
 if (count > 0) {  
 sumVelocities.x = sumVelocities.x / (float)count; // Average velocity  
 sumVelocities.y = sumVelocities.y / (float)count;  
 sumVelocities.nor(); // Normalize to get the direction  
 sumVelocities.scl(maxSpeed); // Go in this direction at maxSpeed

// Subtract current velocity to get the desired velocity  
 sumVelocities.sub(velocity);  
 return sumVelocities;  
  
 }  
 else{  
 return sumVelocities;  
 }  
  
  
}  
  
*/\*\*  
 \* Calculate the average location of the projectiles and use that as a new target  
 \* Very similar to 'align' algorithm  
 \** ***@param*** *projArray*

*\* @****return***sumLocations*(returns either (0,0) vector or the new acceleration we want applied to velocity  
 \*/*private Vector2 cohesion(ArrayList<Projectile> projArray){

// Return sumLocations with new acceleration or (0,0) if there are no neighbors  
 Vector2 sumLocations = new Vector2(0,0);  
 int sizeArr = projArray.size();  
 int count = 0;  
 // Go through projectile list and add up all the different velocities  
 for(Projectile p : projArray){  
 // Get the distance between current projectile and the others  
 Vector2 distanceDiff = projBod.getPosition().sub(p.projBod.getPosition());  
  
 // Check if the 2 projectiles are close together and NOT the same one (id), we just want little flocks of projectiles so threshold is set to 5  
 if(distanceDiff.len() > 5.0 && id != p.id && distanceDiff.len() > 0.0){

// Add position to the total positions  
 sumLocations.add(p.getBody().getPosition());  
 count++;  
 }  
  
 }

// Only do this if there we’re close enough neighbors  
 if (count > 0) {  
 sumLocations.x = sumLocations.x / sizeArr; // Average location  
 sumLocations.y = sumLocations.y / sizeArr;

// Use the seek algorithm to get the desired velocity needed to get to our target. Target being the average center of the current neighbors  
 Vector2 newAcc = seek(sumLocations);  
  
 return newAcc;  
 }  
 else {  
 return new Vector2(0,0);  
 }  
}  
  
*/\*\*  
 \* Seek AI to get the new target position based on current target's coordinates  
 \** ***@param*** *target  
 \** ***@return*** *Vector of new target position  
 \*/*private Vector2 seek(Vector2 target){  
 Vector2 newPosition = projBod.getPosition().sub(target); // Displacement distance from enemy to target  
 newPosition.nor(); // Give the vector a proper direction  
 newPosition.scl(maxSpeed);

// Subtract current velocity to get the desired velocity  
 newPosition.sub(velocity);  
 return newPosition;  
}

Seek: *Centipede.java*

A simple seek algorithm is used for the Centipede to track down the down the player. It is given the player’s current position and travels to the direction at full speed. This algorithm works on the Centipede’s Head body, so as the head moves all other bodies follow behind it. This seek algorithm is slightly different from the one used in flock to give the Centipede’s movement a specific behavior. For instance, there is a Timer scheduled so when the game starts the Centipede won’t immediately go for the Spider, giving the player time to get ready. There is also a dampener applied to the last body (the butt) of the Centipede to extend the bodies out giving it an elongated look.

*/\*\*  
 \* Delay the initial AI start time when Assignment1 loads then seek out player.  
 \** ***@param*** *target players current position  
 \** ***@param*** *enemy enemy AIs current position  
 \*/*public void seekTarget(Vector2 target, Vector2 enemy){  
 // Set new variable to target Vector so we don’t change original value  
 Vector2 temp = new Vector2(target);   
 // Delay the initial AI start time when Assignment1 loads then seek out player  
 if(delayStarting == true) {.  
 Timer.*schedule*(new Timer.Task() {  
 @Override  
 public void run() {  
 delayStarting = false;  
 }  
 }, 3);  
  
 }else{

// Displacement distance from enemy to target  
 Vector2 newPosition = temp.sub(enemy);

// Give the vector a proper direction  
 newPosition.nor();  
 newPosition.scl(maxSpeed);  
 // Head toward the targets position at max speed  
 body.setLinearVelocity(newPosition);

// Damping placed to keep the end body somewhat in line with the head  
 butt.setLinearDamping(linearDamping);   
  
 }  
  
}

Bugs:

There are a couple of bugs in the game but none that break the game – just give it interesting features.

Bug 1:

Depending on where the Web Sacs autogenerate, it is possible for the Centipede to get caught (Figure 11). The Centipede can break out if the player moves around and/or the player collects one of the Web Sacs that is making the Centipede stuck.



Figure 11: Centipede caught between Web Sacs

Bug 2:

The Health Bar animation looks weird when either the player or enemy health gets low. The health bar is programmed to decrease in length based on how much health the player has lost but with the slanted health bar I designed it ends up not fitting the container very well. Due to time this was not fixed.

A screenshot of a video game

Description automatically generated

Figure 12: Weird health bar when health is low

Bug 3:

If the player stays still there is a chance that the player can get caught in the corner of the game screen if pushed there by the Centipede. This is because the Centipede is always seeking the target and both the Spider and Centipede bodies have collision detection on. To counter this, there is a health deduction timer of 2 seconds, for instance, if the player gets hit they can’t take damage again for another 2 seconds. If the player gets stuck in a corner, they just have to try to move away to break free and they should only take 1 hit of damage instead of multiple hits.

Bug 4:

When the Centipede does its AOE burst shot, there is a slight frame delay due to the 100 bodies that are being generated all at once.