COMP 452 – AI for Game Developers

March 27, 2024

Keith Mitchell

SID 3178513

Game Logic and Design

Game Logic

The core idea for the game came from Vampire Survivors, hence the name Werewolf Survivors. The game is a basic horde style survival game with automatic attacking. Since the enemies directly target the player, it uses basic steering behaviours and felt like a good place to start with AI. I decided to expand on the idea a bit and allow the player to control the direction of the attacks by using the mouse but keep the attack on a constant timer like in Vampire Survivors. To keep things simple, I chose to avoid a leveling system and other weapons/upgrades and instead went with an arcade scoring system with 3 lives. The point of the game is to try and get your score as high as possible before getting hit 3 times. Skeletons are worth 500 points while regular wolves are worth 100. Enemies spawn on a 4 second timer while the axe is thrown out on 3 second timer. If the player gets hit there is a 2 second window where they cannot be hit again. These numbers could be tweaked to adjust difficulty and game balance.

Game Engine

A huge chunk of this project was learning Swing and building out the “Game engine” that I would be able to use for the course. I tried to keep things modular and may have gone a touch overkill for the course. There are a few areas I want to refactor/change, but I will discuss those below in the bugs/future builds section. The Scene, MenuButton, GameObject and InputController classes can easily be reused along with the entirety of the game package (with minor tweaks based on the game).

I chose to build a system that utilizes delta time for the core game loop. Since the game loop can run freely using delta time, I added a frame cap to prevent incredibly low values of delta time which can cause issues when multiplied against floats for coordinates. With such minimal changes the values can lose precision and the assets never move. Even with a frame cap of 500 I was not witnessing this, but I left the frame cap at 120 as anything above that was not necessary for the game. One major hurdle I had in building the engine was this frame cap while also using delta time. I could have simply done a busy waiting while loop but that is incredibly CPU wasteful. The problem was that thread sleep times are not guaranteed by systems, they are minimum values. The CPU wont instantly schedule that thread again right when the sleep time is up. To fix this I added a calculation for extra time slept and use that as less time sleeping on the next frame. I also had to do a busy waiting if the time left in a frame was less then 1ms as thread sleeping is incredibly difficult with values that small. I am happy with how the core game engine loop worked out in the end as frame dips on weaker systems will not affect speed of movement (movement uses delta time), while also preventing incredibly fast systems from wasting CPU and power needlessly.

I also decided to separate the game window from the game itself. By passing the Graphics g class over to the game the game can hide the drawing of assets within the GameObject class. Custom draw(g) methods can be used for edge cases. One drawback with this approach is that anyone that uses this engine needs to understand the order in which things are drawn and the HUD/UI draw commands need to be programmed in manually. This could be fixed by added a uiElement abstract parent class, but this felt super overkill for this course.

Lastly, I tried separating out the InputControls and using “Scenes” to handle the input. This has some upsides in that each scene can decide how to handle inputs. However, it adds a layer of complexity if one “Scene” has multiple input types. For example, I originally wanted to do a game over splash screen within the GameCore scene class but this would mean every time input was handled, it would first have to check a state to see if the game was in “GameOver” state. If it was, then it would use the mouse clicking and movement for button controls on the UI and if not, it would be used for axe throwing. The fact that this mouse check is done constantly because the mouse is always moving would mean a ridiculous number of state checks every mouse movement. As such the GameOver screen is its on scene that first calls the GameCore to draw the background and then it can handle its own inputs. I really liked this solution as a Pause screen or settings could be done the same way. The only catch is that it doesn’t resolve situations where during the main game the controls need to change (getting in a vehicle for example).

Compiling/Game Setup

Before you can run the game you must first ensure that the Java Runtime Environment (JRE) is installed on your system. If you are wanting to compile the program yourself, you will also need the Java Development Kit (JDK) installed.

Compiling:
In order to replicate the development environment, I used the IDE IntelliJ. You can load the WerewolfSurvivors folder as a project. Depending on your compiling environment you may need to target the assets folder as the resource folder. You can usually do this by right clicking the folder in the IDE and marking it as the resources folder, though each IDE may differ. You then can hit use the IDE to compile/build/run the game. In IntelliJ, open the WerewoldSurvivors file (in the game package), and click play (Shift+F10).

You can compile without an IDE. Generally, this will involve navigating to the project folders and using command line prompts to compile all the .java files using the command line javac. This turns the files into .class files which can then be run using the java command line. (WerewolfSurvivors.java is the main entry point file).

If for some reason the assets cannot be found and you are getting null references, this might be due to miss targeting of the assets folder. The only location needed to adjust will be within the ResourceLoader class. Adjust the folder directory to match your systems directory structure to the location of the assets folder.

Running from .jar:
The .jar file is located in the WerewolfSurvivors folder. On most systems you can simply double click the .jar file and the game will launch. (Assuming you have installed the JRE).

To run the game using command line you can navigate to the folder containing the .jar file and the enter the command: java -jar WerewolfSurvivors.jar

AI Steering Behaviours

For steering behaviours I choose to use a basic seek() as is used in Vampire Survivors. The algorithm currently takes in an enemy object and player object and directs the enemy object towards the player object. The seek() method finds the x and y distance to the player and creates a normalized vector from that to get the direction. It then multiples that by the enemies' speed and moves the enemy.

I decided to do a physics system for the weapon which simply gets a launch vector using the mouse coordinates and the players location. The axe gets a vector towards the mouse location and then maintains that vector until it is shot again. Really basic weapon launching.

For the complex steering behaviour, I decided to do a custom behaviour to have skeletons attempt to dodge the players shots. The system uses a circle around the player to get angles for both the mouse location and enemy location compared to the players location. It then uses the difference in those angles to see if the enemy falls within the cone (angled cone from player). Vision cones can be done the same way. If the enemy is within the cone, it then is assigned a perpendicular vector away from the mouse angle. It then checks to see if the enemy falls within a cone degree around the mouse. From there it uses blending with seek() to ensure the enemy still attempts to move towards the player while dodging. The blend is 60% avoidance and 40% seek().

Finally, enemies use collision detection to mimic flocking and avoid overlapping. This collision detection runs last so that all units still move towards the player first using seek() and then reposition slightly to avoid overlaps. The collision detection algorithm compares each enemy against one another and then shifts if there is an overlap. It is a O(n^2) algorithm so is not an efficient solution but for the purposes of the game and creating a grouping effect it works well with its simplicity.

How To Play

Upon launching the game, the player is presented with a play button and splash screen. Once the user clicks on play the game launches. The player moves around using WASD controls and uses the mouse to control the direction of axe throws. By moving the mouse around the screen an axe will automatically be thrown at regular intervals in the direction of the mouse. When the axe hits an enemy, they are killed, and the axe does not stop going until a new one is thrown.

AI Behaviour Triggers:

In order to test the steering behaviours in the game move the player around the screen and the enemies will track towards the players location using seek(). Aim the mouse in the direction of a skeleton enemy and they will move away from the mouse using dodge(). Try moving the mouse around the skeletons and they will switch to the shortest direction out of the "aiming cone". As enemies begin to clump together, the collision detection and avoidance can be seen as enemies won't overlap one another. The shooting physics can be seen every few seconds when an axe gets thrown towards the mouse. When the axe connects with an enemy they will be removed, and the score will increase.

Bugs/Future Builds

Bugs

For the most part, minimal bugs have been noted. The main bug right now is that the mouse must be within the game window to update the mouse location for axe throwing.

There are also no checks in place for the max number of enemies on screen. In theory, this could burden lighter systems. I tested the game with 100 wolves on screen at once and I did not drop below the max 120 FPS. However, if I were to spend more time with the game a max enemy check would likely be worth adding as a protection from possible crashes.

There were also a few moments where input controls were finicky and required me to click on the game panel. My guess is that the panel was inconsistent in receiving input focus. As such, the player may need to click a second time on a button if the window did not properly have the focus for input listening. It is inconsistent when this happens and may be a Java Swing issue. It is worth noting that this controls issues was incredibly rare which made it that much harder to troubleshoot and may even be fixed as I can not force replicate it.

Sound/Animation Systems

These are the biggest misses with the current engine. Adding an animation timer and sprite sheet with draw offsets based on the current frame is not incredibly difficult, but it is time consuming. I wanted to get this current build out as I needed to manage my time and look at adding animations for the second assignment. Sound adds a ton of game feel, but aside from game feel it does not showcase the core requirements of the assignments. Due to timing, I chose to let this drop and will also investigate adding a sound system for assignment 2 and 3. Sounds also mean I should add a settings menu for muting and controlling the audio.

Hit Detection

The player hitbox is a bit unfair to the player and adding the flexibility for a custom hitbox to the GameObject class is something I would like to do in the future. This will be even more necessary with animations as different frames of an animation often require a unique hitbox (eg. Sword swing hitbox). For the core purposes of this assignment the current build felt sufficient, though from a game feel perspective the players hitbox should be smaller to be more forgiving to the player. In addition, all math uses the top left coordinates of sprites as its location, not the centre of the object, so the axe technically spawns in the top left of the player, not the centre. The same is true for the cone effect. All calculations are going to the draw location. A more accurate calculation would use the centre of the sprite. This is a quick fix in future iterations but also felt unnecessary for the purposes of the assignment.

Game Engine

I went through several refactors and changes during the development of the engine and some systems have not been updated to use the newer features properly. For example, I added Vector2D quite late in development and I need to do an overhaul of some of the algorithms/classes to utilize this for coordinates and velocity. When I would really like to do is create a normalized movement

vector and then set then GameObjects can have a directional vector stored. This is then updated by the AI management or the player inputs. Then, a call to move() can be within the gameObjects update() method. This removes the speed aspect from the AI manager as the move() method did not get fully refactored. Move takes a x and y velocity which means all that the movement method is doing is the deltaTime application to the velocity and adjusts the location. The AIManager doesn't care about delta time so passing it through just for the movement check is awkward. I would also prefer if each enemy called the AIManager to determine their movement during their update methods but this caused issues with having to pass the player into enemy updates and I didn't really like this solution. Overall, it still works fine, it just has some refactoring artifacts that need to be cleaned, specifically with regards to movement and velocity.

The only thing I would maybe add to the main engine loop would be the flexibility to multithread for game logic/drawing. AI checks also do not necessarily need to happen every single tick and so having a timer or separate update for certain types of logic could be more efficient. This complexity is not necessary for this game but is something I am considering. Multithreading would be interesting to tackle from a learning perspective as I have yet to build a custom multithreaded game. (I have built multithreaded for other software but not a game).

AI Management Methods

In future builds I could tweak the seek() method to instead take in a target location (Vector2D), instead of a player object. That way it could be used for more general movement, including chasing the player. Because I added the Vector2D class so late I partially cleaned up the algorithms but then still have to solve for a velocity within the algorithms. By splitting out the direction from the speed and instead only returning a Vector2D with the final direction, I could call seek() from within dodge() and then use the two normlaized vectors to blend them together and create a new directional vector that gets returned. The only reason I haven't done this is because it was further refactoring and I needed to limit my time spent overdoing it. Changing the move method and the way that gets called was one refactor too many and since it wouldn't change the end result, I decided to hold off on this change.

Overall Thoughts

Overall, I am really happy with how the core "game engine" stuff turned out. I have attempted to build more modular systems in prior courses to but have not gotten it this close before. There is still a bunch of things that prevent it from being completely modular, but as a starting architecture I feel good about the design. The only downside is that I spent far more time building the engine which meant I ran out of time to work on the game feel and balance.