CS4303 Video Games AI Practical

180013715

March 2022

1 Introduction

The goal of this practical is to allow us to implement concepts from AI and procedural generation in games. The requested game is implemented in Processing, and it is based on the classic video game **Robotron 2084**.

The game will be a multi-directional shooter. The setting is that of a future timeline, robots have banded together and revolted against the humans. The player operates a "superhuman" who has a projectile weapon, with which to defend themself against endless waves of robots. Additionally, there is one surviving human family, and the player should aim to rescue them before they are dead/transformed and before the player themself is killed.

2 To Run

This game was made in Processing 4.0 beta 6. To run the game, go to the *Game* directory, open *Game.pde* in the Processing IDE and press the *Run* button.

The size of the arena where the characters reside will scale according to the screen size of the device the game is running on. This was tested on a M1 MacBook Air (11.97 $\times$ 8.36 inches) and the lab machines (20.47 $\times$ 11.4 inches).

3 Design

3.1 Control Scheme

WASD is used to move the player operated superhuman, and arrow keys are used to fire. These controls are independent of each other. The player may move diagonally, but may only fire in the four basic directions.

3.2 The Play Area

I envision an arena for my game where the procedurally generated level layout and the characters are. Meta information such as number of lives remaining, wave number, and score are then displayed above the arena.

3.3 Enemy Designs

There are a total of four types of enemies in my game. They will be represented as rectangles of different colours.

- Basic Enemy - this enemy will simply roam and attack the player or a family member when they collide with them
- Attacker - this enemy will dart towards a family member when they are spotted and attempt to attack them
- Transformer - this enemy will also spot the family members and progress towards them. The family members will be transformed into an Omnipotent robot when touched by a Transformer
- Omnipotent Enemy - this enemy uses A* search to find a path towards the player and dart towards them

The enemies will have the same shape and speed, but each type of enemy possesses a different behavior towards the player and the family members. The basic enemy type does not have any intelligence behind their actions, they simply roam and attack any human they bump into; The attacker robot will use the line-of-sight mechanic so that when they spot a family member, they will proceed in that direction; The transformer robot will use line-of-sight in the same way as the attacker but the transformer will birth a new type of enemy when successful; The last type, omnipotent enemy, can be understood to see through wall to locate the player and can find the optimal path towards them.

3.4 Human Family Designs

Family members will be rescued when they are touched by the player. When they 'see' the player, they will progress towards the player so they are rescued. They will also flee from robots when they 'see' them. Line-of-sight is used here again.

There are a total of three types of family members in my game.

- Parent - this family member will be worth 2500 points.
- Guardian - this family member will be worth 2000 points.

- Child - this family member will be worth 3000 points.

The family members are worth different points, so the player should decided themself who to prioritise saving, as cruel as that sounds.

3.5 Obstacles and Power Ups

There are stationary objects in the game which will aid or be a hindrance to the player. Power ups help the player, their types are listed below:

- Freeze - freeze all enemies for a short period of time
- Invincible - player will become invincible for a short period of time, during which enemies touched by the player will score them points
- Destroy - kill all enemies within a certain radius

There are also obstacles, which will decrease the player's lives by one. They can be thought of as landmines. When an enemy steps on them, they will also be destroyed. However, when a family member steps on them, they will be unaffected. This design decision is not realistic, however it is better for the game play than obliterating family members that wander into obstacles as there as few of them as it is.

Stationary objects are both represented as circles. To distinguish the positive and negative effects they may have on the player, power ups are filled circles (each type has a unique colour) and obstacles are rings (like a trap).

3.6 Scoring

Destroying enemies will only score points if the player initiated the attack i.e. a bullet destroyed the enemy. Obstacles can destroy enemies too, but this will not gain points for the player. This is so that the player is incentivised to attack and not just hide in the corner.

3.7 Level Design

Each level should have a different layout of walls, corridors, and rooms. To generate a level, the given area is first split into a series of blocks. Each block will be either a wall, or a space where the characters can move in. I will be using a random walk algorithm to find a path (e.g. a subset of blocks) through the given space, then the blocks that have not been passed through are considered 'walls'.

Once the layout is generated, the characters and objects are spawned. The parameters used for character and object spawning is described in detail in the Implementation

section.

A wave is considered complete/over when there are no enemy robots remaining. In this case, the game will progress onto the next wave. The player starts with five lives, and they will earn a new life every three games. If after three games, the player still has full health, they will have to wait for another three games for a new life should they lose a life in the next game. I decided to have this earn new life mechanic be based on games rather than score because the number of enemies will increase by a lot as the waves process, so a fixed score for earning a new life would make the game play uninteresting in later stages.

3.8 Collision Detection

Consider the shapes in this game: player and the bullets they shoot are **circles**, obstacles and power ups are **circles**, enemy robots are **rectangles**, family members are a **circle** on top of a **rectangle**, and the blocks representing the level layout are **rectangles**.

The collisions we need to consider are between circles and circles, rectangles and circles, rectangles and rectangles.

To characters should be limited by the layout of the level, meaning they may be blocked by walls and tight corners.

4 Implementation

The game was developed incrementally. Each class/file is kept as concise as possible to ensure the code is readable and that the logic of the game is easy to follow.

Helper classes/files include *Utility*, *EnemyManager*, *FamilyManager*, *GameStateManager*, *ObjectManager*. As their names imply, the *Utility* file includes functions that are utilised quite often, like collision detection methods. The manager files are responsible for holding functions that handle character and object creations.

I thought implementing a *Timer* object class would be helpful, as many components (e.g. roaming behavior of characters, power up effects) of the game are based on the passage of time. The *Timer* constructor takes in a value in milliseconds and creates a timer based on this value. Its *stop* function will return whether enough time has passed. It also contains a function which can be called to restart the timer. An example usage is on the enemy's roaming behavior: when an enemy is created, it will initialise a timer for itself. When the enemy moves, it will check whether the stop time specified by the timer has been reached (i.e. if enough time has passed for the next di-

rection change). If it has, the enemy will choose a random direction and move that way.

The characters and objects are each stored in their own arraylists.

4.1 Drawing the Arena

The width and height of the arena where the characters reside are defined as 85% of the full width and height of the screen.

4.2 Moving Controls

I want the player to move in diagonal directions as well as horizontal and vertical directions. To achieve this, an array of currently pressed direction keys is utilised. The array contains boolean values for whether each of the four direction keys (i.e. WASD) is pressed. Note that the latest pressed horizontal/vertical key will override the older key, e.g. if A is pressed then D is pressed, the right effect will override the left effect.

If a vertical key is used while a horizontal key is pressed (or vice versa), then the boolean array will record this and set the direction of the player to a diagonal direction taking both keys into account. When a key is released, this action is also reflected in the boolean array.

4.3 Enemy Robots

The enemy robots all have the same speed and mode of attack. An abstract *Enemy* class is created so that the different enemy types will all hold basic properties like velocity, position, width, height, liveness etc. Each enemy robot will also have their own timer, which is initialised to a random value between 1.5 seconds and 5 seconds. This value defines how long the enemy robot must wait before its next direction change. The varying wait times make the enemy movements more unpredictable and interesting, in my opinion.

The enemy class has a *handleCollision* method which checks for collision with family members, collision with the player, collision with bullets, and collision with obstacles. The different enemy classes *BasicEnemy*, *Attacker*, *Transformer*, and *Omnipotent* will all call this in their *move* function which determines how they move.

4.4 Level Generation

As mentioned in the Design section, the random walk algorithm is used in generating the layout for each level. The first step is to split given space into grids based on the specified row number. For example, for the first level, the number of rows is set to be 20. The width of each block (i.e. cell) can be calculated as the full width of the arena

divided by the row number. The calculation of the height of each block is similar - first we get the number of columns by dividing the height of the arena by the width of each block calculated at the previous step, then we work out the height of the block by diving the full height of the arena by the number of columns. This two step process is required so the block fill up the arena and don't leave a gap between the edges of the arena and the block representation of the arena.

Two important parameters for layout generation are the maximum number of turns allowed, and the maximum length a corridor can be before the algorithm decides to make a turn. Through some trial and error, I decided that setting the maximum number of turns allowed to be four times the number of rows, and the maximum length of a corridor to be a quarter of the number of rows.

The algorithm's steps are outlined below:

- set up a 2D array that will represent the layout, wall has a value of 1 and walkable space has a value of 0. Initially, all values are 1s
- choose random starting point on the map
- while the number of corridors is not zero
  - choose a random length from maximum allowed length
  - draw a corridor that way while avoiding the boundaries of the arena (i.e. set value to 0)
  - decrement the number of corridors

The resulting map will have a series of 1s and 0s. Then, a 2D *Block* array can be made from this integer map. Each *Block* object will have a boolean associated with it which determines whether it is a wall or not.

The number of rows is incremented by one for each new level, this means the layout will become more convoluted. Look at the difference between the level 1's layout and level 40's layout.

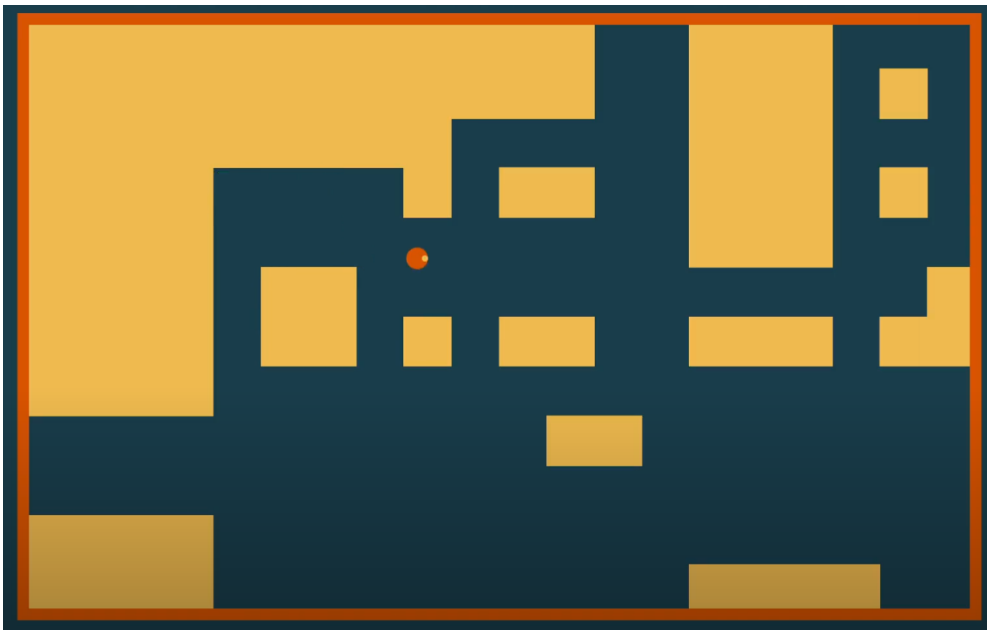

Figure 1: Level 1 Layout

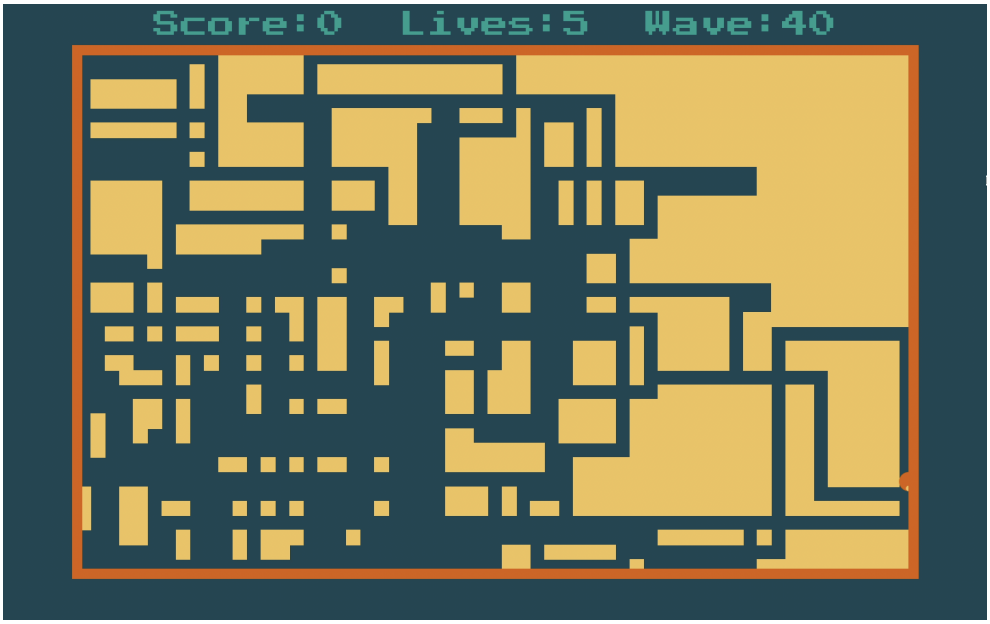

Figure 2: Level 40 Layout

At the end of each wave, the arraylists for the characters and the obstacles are cleared before generating the next wave.

4.5 Collision Detection with Walls

Knowing whether a character has collided with a wall is very important in this game. It would be inefficient to check the character against every block to determine if a wall has been met because the character can only be at one place at one time. We can use the current block the character is residing in to find adjacent blocks (e.g. by simply looking at their indices in the block array), and perform collision detection with these adjacent blocks.

When a wall is detected, the character (excl. player) will move in the opposite direction to get away from the wall.

When a player comes into contact with a wall, there needs to be a way to stop them from continuing through that wall. Four boolean values were introduced to the *Player* class for this reason, they represent whether going left, right, up, and down is allowed given the current circumstances.

In the player's *move* function, the player's position will only be updated according to the direction keys if the corresponding boolean values are satisfied. The boolean values are updated when a collision is occurring. For example, if the player is colliding with the block to their right, then the boolean defining whether right is allowed is set to false.

4.6 Line-of-Sight

Some enemy robots may spot their target using line-of-sight, in which case, they will move towards the target to attack them. The diagram below shows how an enemy 'sees' a family member. Two tests are performed on the two sides of the objects - whether we use the bottom left corner and top right corner pair, or the bottom right and top left corner pair depends on the objects' positions relative to each other.

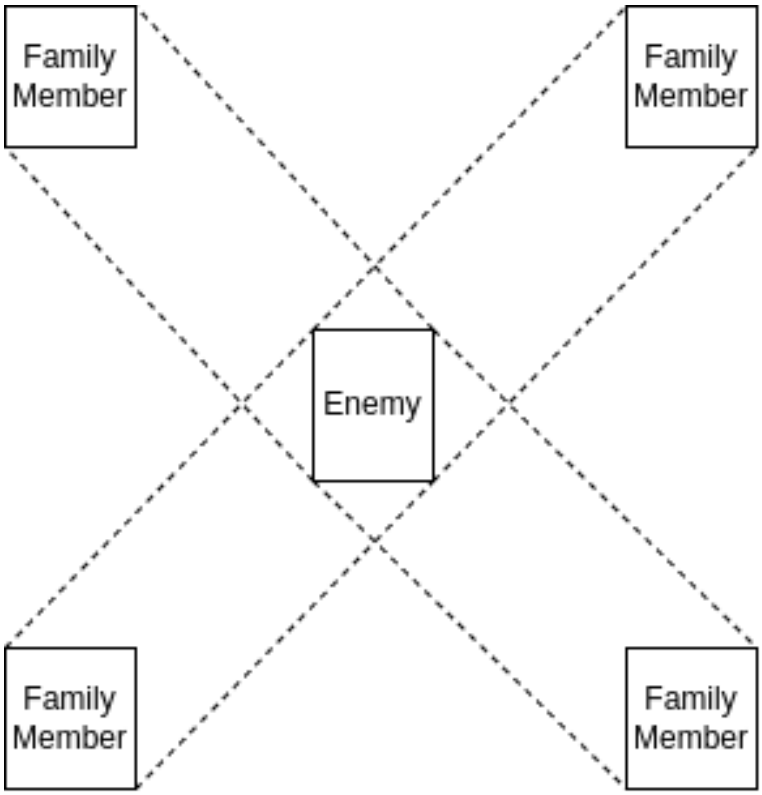

Figure 3: LoS With Rectangles

If either of the lines between the two objects are blocked by a wall (i.e. line-rectangle collision detection), then we can conclude that the objects cannot see each other. It would be inefficient to check all the blocks in the arena to see if they collide - we simply need to check the blocks close to (i.e. between) the starting block and the destination block.

For the example below, collision detection is only performed on cells (1,0), (2,0), (0,1), and (1,1).

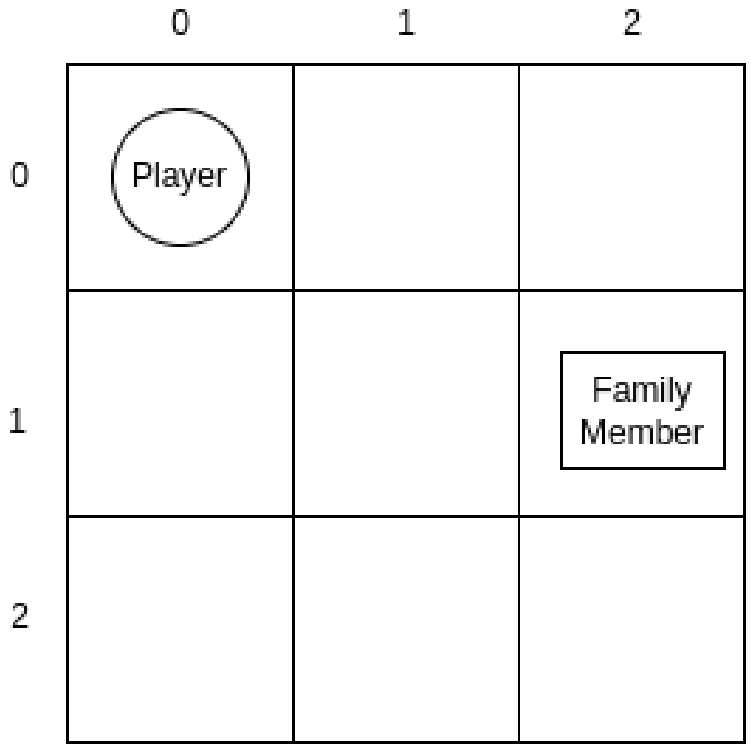

Figure 4: Block Check Diagram

Another point to mention is that the player is represented by a circular shape, so the edge method as described above will not work. An altered method is used so that the corners of the rectangular character seeks for a quarter of the player, as shown below.

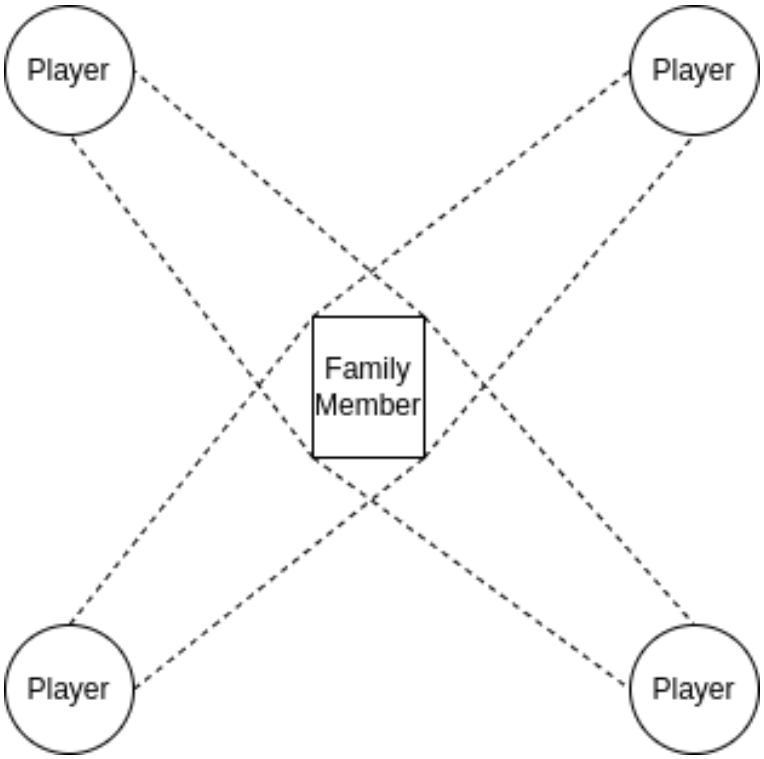

Figure 5: LoS With Circle and Rectangle

4.7 Power Ups

Three types of power ups were implemented. The freeze power up and the invincible power up are both limited by timers. Two timers are declared, one for managing the freeze effect and the other for managing the invincibility effect. When the player walks into a freeze power up, the freeze timer will start and the *canMove* state of each enemy is set to false meaning they will be frozen in place. The status of the timer will keep on being checked, and when the time limit is up, the *canMove* state of the enemies are set back to true, allowing them to roam once more. If the player walks into another freeze power up when the enemies are still frozen, the timer's limit will be set to a short period after the second power up i.e. the time limits do not accumulate.

The invincibility power up operates in a similar manner. When a player walks into an invincibility power up, they becomes so that no lives can be lost for a short while. They can also walk into enemies during this period and kill them.

The power ups can be easily extended to be applied to family members (e.g. family becomes invincible) and enemies (maybe a hard mode option).

4.8 A* Search

The *Omnipotent* enemy type can perform A* search targeting the player. They are named as omnipotent because they have complete information of the arena and will always steer towards the player, regardless of the walls in the way. An *Omnipotent* enemy is created when a transformation of a targeted family member is initiated by a *Transformer* enemy. The idea is that the hate and anger the transformed family member feels towards the player allows them to see through walls and proceed towards the player for revenge. Credit to 180014469 for this idea.

I used Week 5's A* solution with some added out of bound checks. To start A* search, the first step is to find the source and destination indices on the layout grid. The *AStarSearch*'s *search* method will then seek the optimal path to the destination from the source location. Note that a new AStarSearch object is created according to the new level layout.

4.9 Game Screens

There are four game screen states in my game: pre-game, during-game, transition, and post-game. The pre-game screen will prompt the player to press space to start the game; The during-game screen will draw the game while it is running; The transition screen is drawn when the current wave is over. It will start a timer allowing the player to get ready for the next wave and the program will go back to the during-game screen when the time limit is up; The post-game screen tells the player the game is over and the player can press Q to quit the game. I reused some methods from the first practical for this part, the code is referenced in the comments.

5 Testing

I recorded some video demos to highlight these features:

- Collision detection on walls and arena borders
- Basic enemy roaming
- Omnipotent enemy seeking/following player
- Attacker enemy using line-of-sight to seek their target, the target will attempt to flee
- Transformer enemy transforming a family member into an Omnipotent enemy
- Family member seeks player using line-of-sight

- Player destroying obstacles and bumping into obstacles (watch the number of lives)
- Freeze effect from power up
- Destroy enemies within a certain radius power up
- Invincibility from power up
- Level generation up to level 40
- Level generation with all characters and objects
- Gameplay with everything

6 Conclusion

For this practical, I created a game in Processing in which the player controls a 'superhuman' and rescues the last remaining human family while attacking and evading enemy robots.

The vast range of characters and objects in this game, especially the different types of enemies and power ups, make the game play quite interesting. It may take a new player a few runs to understand what the different types of characters and objects do, as their only visual variation is their colour. Given more time, I would make the visual representations more different. For example, draw eyes on the enemies that seek, and have an explosion type shape drawn when a *Destroy* power up is used.

The basic properties of the characters are kept simple so the different behaviours stand out more. Family members and enemies will travel at the same pace, but the rate at which a character changes direction is randomised.

There was a problem with the characters that sometimes they would bounce between two walls. This happens when the character is close to a tight corner, meaning the character will detect a wall, change its direction into another wall, then change its direction back into the first wall and so on. This issue is countered by the timer not being updated by a direction change due to arena border or wall collision, meaning the timer will be running when the character is bouncing between walls and eventually it will change its direction and hopefully that will break it out of the cycle.

Overall, I believe my implementation of the game satisfies the requirements as outlined in the specification.

7 Resources

http://www.jeffreythompson.org/collision-detection/line-rect.php http://www.jeffreythompson.org/collision-detection/rect-rect.php https://processing.org/reference/ https://www.freecodecamp.org/news/how-to-make-your-own-procedural-dunge
https://en.wikipedia.org/wiki/Robotron:_2084#Enemy_designs
A* solution code from Week 5.