1 Abstract

The specification outlined the creation of a variant of the classic video game Robotron 2084 through use of the Processing language. The final implementation includes the variations specified, including the use of Procedural Content Generation (PCG) to generate random levels and power ups available for the player to collect. In addition to this, the variant extends upon the specification through the inclusion of an upgrade shop, starting options menu and several player weapons. Furthermore, the code builds on top of an extensive class inheritance system started in the previous practical.

Figure 1: Robotron 4303 (Main Menu)

2 Player's Guide

2.1 Premise

Fight for humanity against killer robots in Robotron 4303! Run and gun with your pistol, expanding your arsenal and using power-ups scattered throughout each procedurally generated level. Rescue Final Family Members before the robots reach them! Upgrade your arsenal in the shop to keep up with ever increasing waves of robots. As you progress, you will face a larger variety of robots. Will you be able to hold on as humanity's last hope or will you perish at the robot's hands?

2.2 Controls

Mouse:	Aim
Left Mouse Button:	Shoot (Automatic)
Right Mouse Button:	Shoot (Single)
Arrow Keys:	Aim and Shoot
WASD Keys:	Move
Number Keys:	Select Weapon
m:	Skip to next Wave (Optional)

2.3 Installation Instructions

The game was created in Processing version 4.3. Besides the libraries standard with the installation, the official Sound library is also required. Finally, the code must be run in a folder matching the name of the main sketch file (CS4303Practical2).

3 Game Design

3.1 Player

The player is represented as a character on the screen and can be controlled with the keyboard and mouse. The player starts with 100 health points each wave and receives damage if hit by a robot or a laser. Furthermore, the player starts with two lives and can gain more every 25'000 points. The player and family member sprites are shown in Figure 2.

(a) Player (b) Final Family Member 1 (c) Final Family Member 2 (d) Final Family Member 3

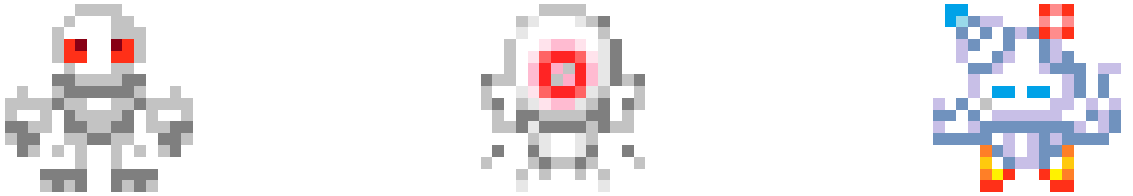
Figure 2: The Player and Final Family Members

3.2 Enemies

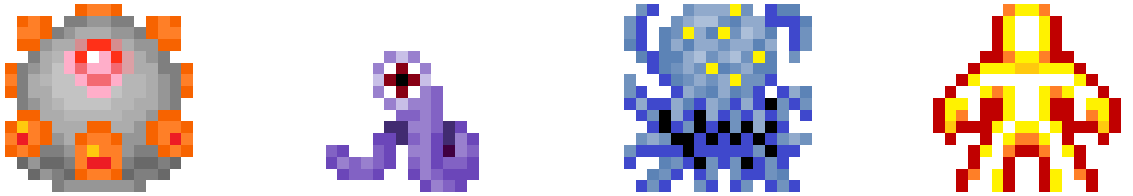
The game features a range of enemies inspired by those in the original Robotron game [1]. Each is summarised in Figure 3. Every enemy except the Turret can hit the player through melee attacks with a short cooldown. The Heavy, Rocket, Turret, Brains and Plasma enemies can additionally shoot at the player. The enemy robot sprites are shown in Figure 4.

- Grunts: The standard and most numerous enemy. They will chase the player directly when in view, hoping to hit them in melee combat.
- Heavy / Laser: Will chase the player directly like grunts, but move more slowly and fire powerful lasers at the player. Also have more HP.
- Rocket / Flying: Move about quickly through the map, strafing near the player and shooting at them. Lower HP.
- Turret: Stationary enemy found mostly in pits throughout the level that will occasionally shoot at the player. High damage and high HP.
- Worm: A fast moving robot that attaches to the player and family members, holding them stationary and damaging them over time. Very low HP.
- Brains: Will seek out family members in the map, transforming them into Plasmas if touched. Medium speed and minor increased HP.
- Plasma: A transformed family member. Moves and fires extremely quickly.

Figure 3: All Enemies



(a) Grunt (b) Heavy / Laser (c) Rocket / Flying



(d) Turret (e) Worm (f) Brains (g) Plasma (Transformed Family Member)

Figure 4: The Enemy Robots

3.3 Weapons

Weapons are scattered throughout the level and unlocked permanently once picked up. All weapons except the pistol have a finite number of shots, which recharge over time. Picking up a duplicate weapon or losing a life will instantly fully recharge shots. A summary of weapons is given in Figure 5.

- Pistol (Starter weapon): Low damage with fully-automatic fire.
- Rifle: Medium damage with fully-automatic fire.
- Pulse Cannon: Shoots medium damage splash shots with semi-automatic fire.
- Railgun: High damage piercing shots with low fire rate.
- EMP Cannon: High damage explosive shots with low fire rate.

Figure 5: All Weapons

The pistol, rifle and pulse cannon and EMP cannon all share a common laser projectile, with damage and size adjusted to suit each weapon. The EMP cannon uses a special laser variety that splits into multiple bursts upon hitting an enemy. Standard lasers perform hit detection through the position of a laser. The railgun instead instantly traces a beam as far as possible that damages any enemies inside it through use of a raycast. Weapons can be aimed and fired through either the mouse or the keyboard to the user's preference.

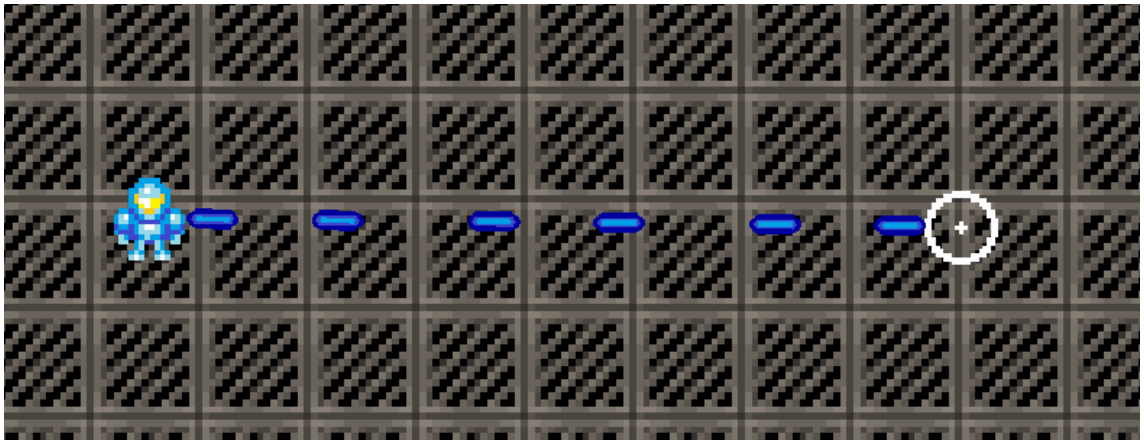


Figure 6: The Pistol Being Fired

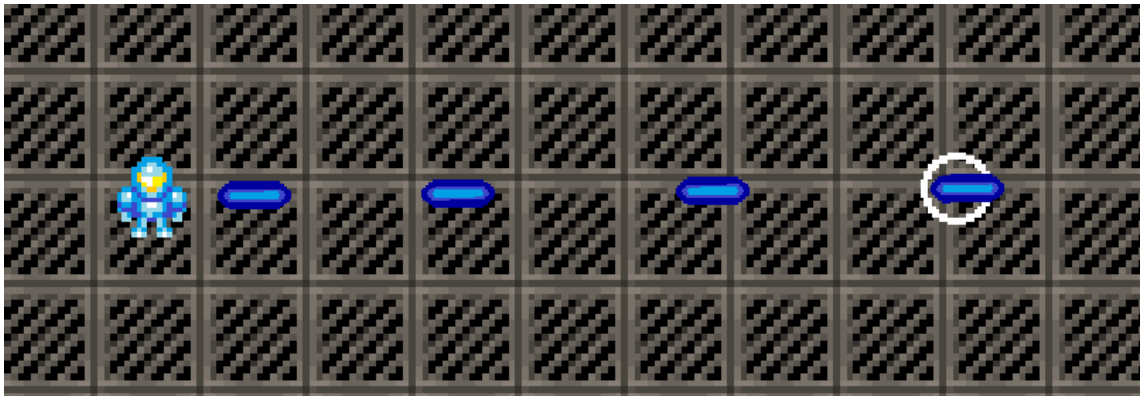


Figure 7: The Rifle Being Fired

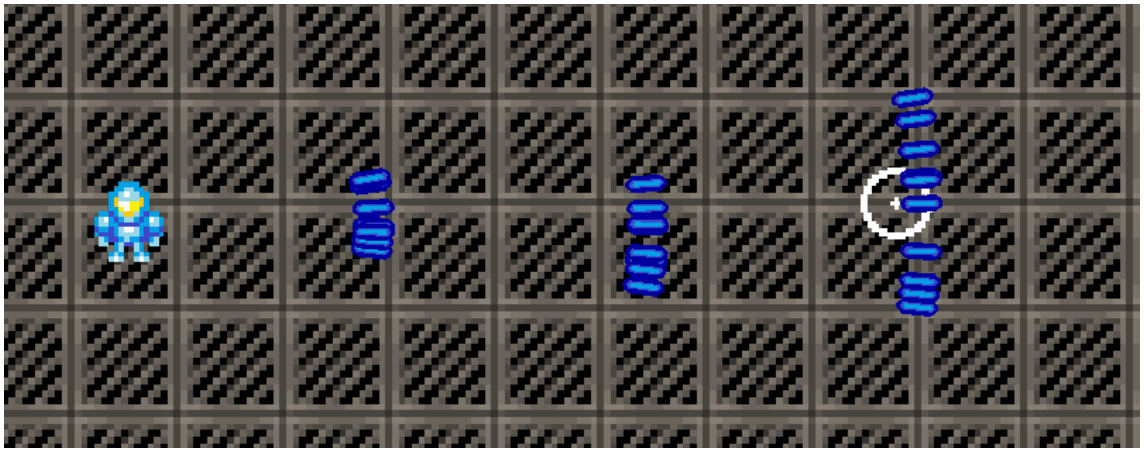


Figure 8: The Pulse Cannon Being Fired

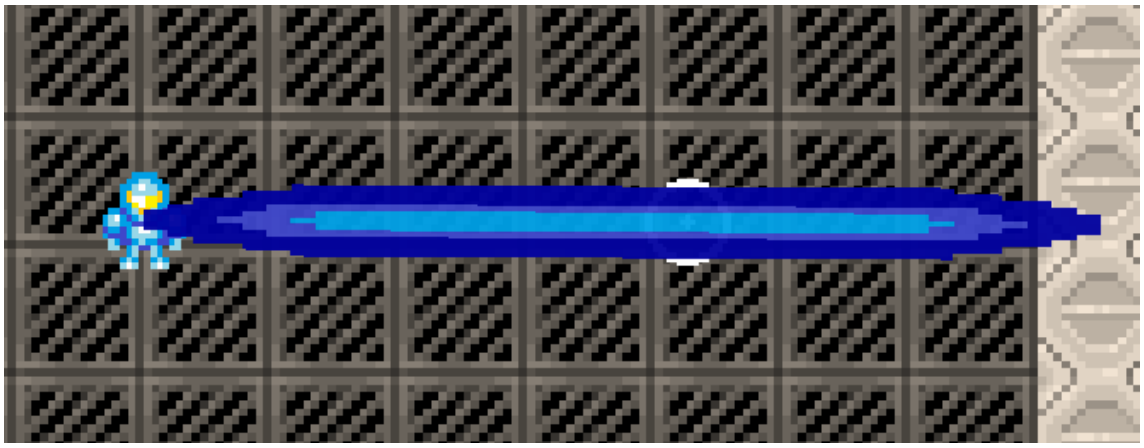


Figure 9: The Railgun Being Fired

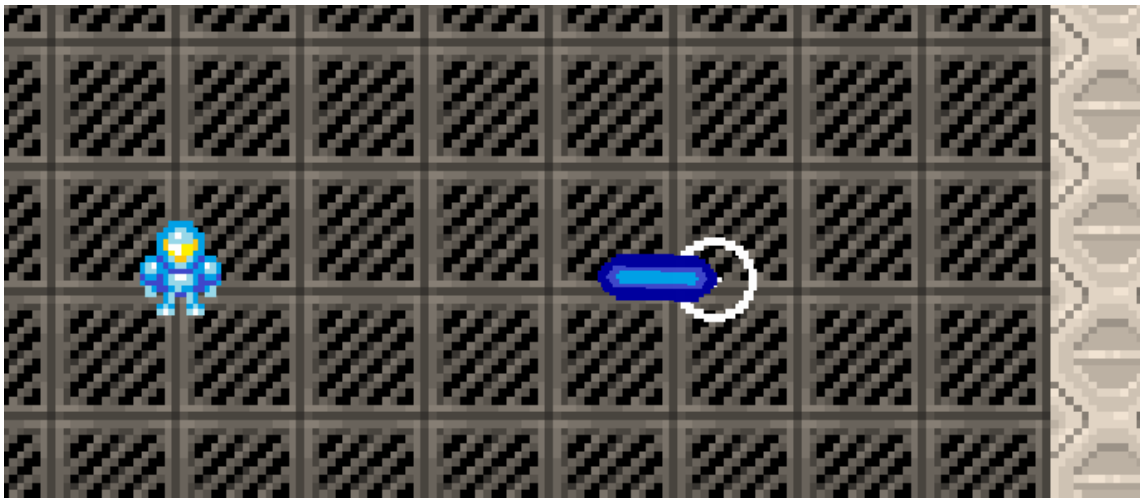


Figure 10: The EMP Cannon Being Fired (Before Split)

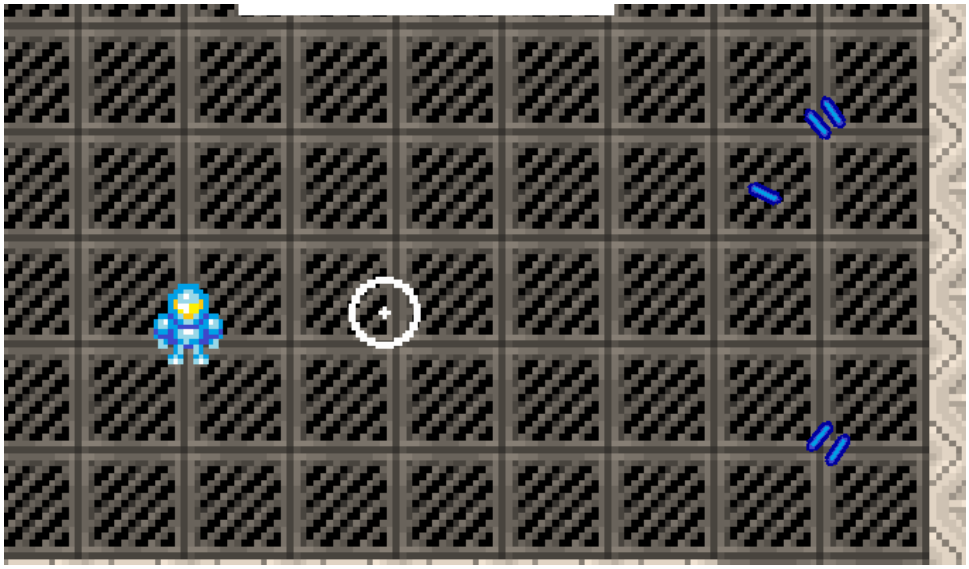


Figure 11: The EMP Cannon Being Fired (After Split)

To help the player keep track of which weapons are available and how much ammunition each has, a display is kept in the bottom left corner as in Figure 12. The background circle's opacity shows the ammunition remaining. Text next to the icon shows which number to press to switch to the weapon.

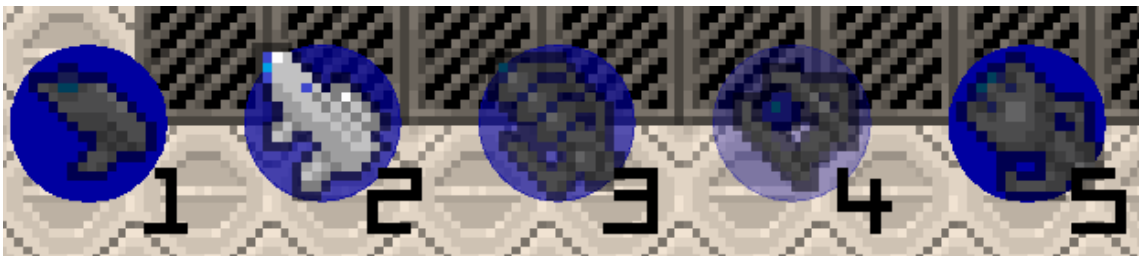


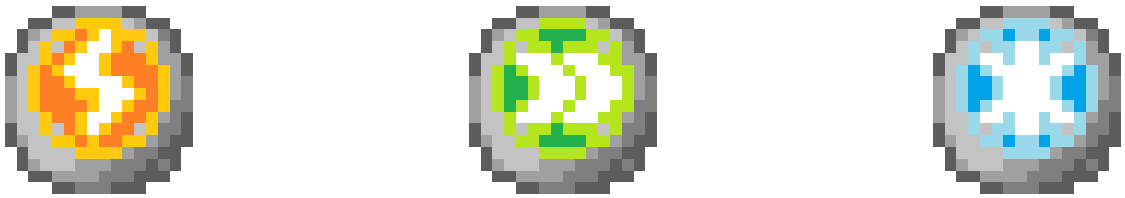
Figure 12: The Weapons Display

3.4 Power-Ups

Power-ups are also scattered throughout the level. Collecting a power-up activates it for a limited time. A summary of power-ups is given in Figure 13 and their icons are shown in Figure 14. A shorter Freeze ability is triggered automatically at the start of each wave. Enemies are highlighted in red and family members in green. The player is highlighted in blue, but not frozen. Together, these effects give the player some time to get their bearings and study the level at the start of each wave.

- Damage Boost: Doubles the damage of the player.
- Speed Boost: Doubles the movement speed of the player.
- Freeze: Freezes all enemies for several seconds.

Figure 13: All Power-Ups



(a) Damage Boost (b) Speed Boost (c) Freeze

Figure 14: The Power-Up Icons

3.5 Shop and Options

The upgrade shop is one feature adapted from the previous practical. As before, it is available at the end of each wave and allows the player to upgrade their weapons. Upgrades can be bought through credits mapped from the current score that the player has. They can be bought and sold freely as points permit, with the pause state between waves allowing the player to test any upgrades they buy. In addition to specialised upgrades, general upgrades for each weapon are also available. Each of the available upgrades is listed in Figure 15 and shown in-game in Figure 16.

- Damage Upgrade: Increases the damage of weapons.
- Recharge Speed Upgrade: Increases the rate at which weapons recharge.
- Rifle Fire Rate Increase: Makes the rifle shoot faster.
- Pulse Cannon Burst Count: Increases the number of bursts emitted per shot.
- Railgun Beam Width: Increases the size of the railgun's beam.
- EMP Cannon Explosion Bursts: Increases the number of bursts from EMP explosions.

Figure 15: All Shop Upgrades

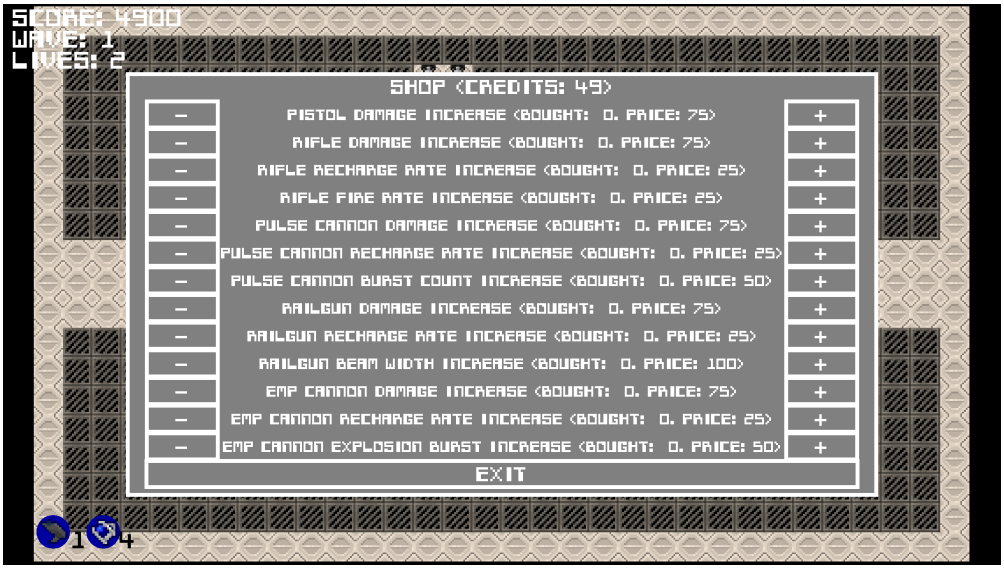


Figure 16: The Shop Menu In-Game

The options menu is also adapted from the previous practical. Before the game starts, the player can choose to access the options menu. This contains settings that allow the player to customise the difficulty of the game. The available settings are listed in Figure 17 and shown in-game in Figure 18.

- All Weapons At Start: Gives the player all available weapons at the start.
- Infinite Ammo: Gives the player infinite ammo for all weapons.
- Credits Multiplier: A multiplier on the number of credits awarded to the player.
- Player Weapon Damage Multiplier: A multiplier on how much damage the player does to enemies.
- Player Speed Multiplier: A multiplier on the player's movement speed.
- Power-up Time Multiplier: A multiplier on how long power-ups last.
- Enemy Spawn Count Multiplier: A multiplier on the number of enemies spawned per wave.
- Enemy Melee Damage Multiplier: A multiplier on the enemy's melee damage.
- Enemy Laser Damage Multiplier: A multiplier on the enemy's laser damage.
- Enemy Speed Multiplier: A multiplier on the enemy's movement speed.

Figure 17: All Options

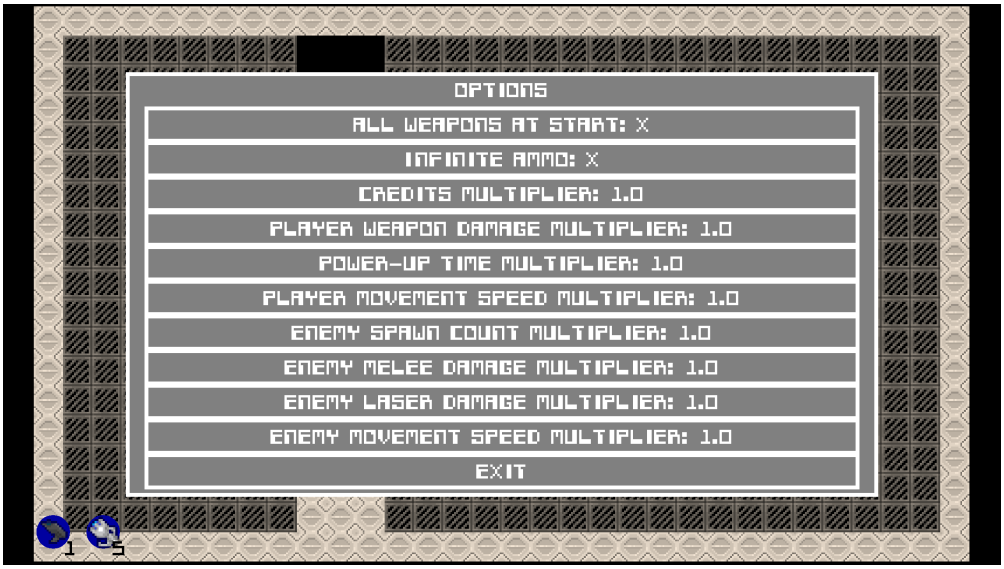


Figure 18: The Options Menu In-Game

3.6 Procedural Level Generation

3.6.1 Map Geometry

The design of each level is randomised with certain parameters to create consistent output. There are a range of tile types to provide variety as listed in Table 1.

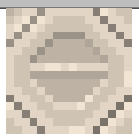
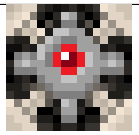
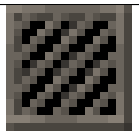
Tile Type	Passable	Blocks Shots and Vision	Image
Wall	✗	✓	
Pit	✗	✗	
Electrode	✗	✓	
Empty / Floor	✓	✓	

Table 1: Tile Types

The level is split up horizontally into one to three chunks and vertically into one to two chunks. Bridges and corridors connect rooms, while walls and pits serve as impassable space as in Figure 19. Walls block shots and enemy vision while pits don't, providing more variety. This level of subdivision allows for variation while keeping ample space for the player to move around. This is vital to match the style of the original game, as being forced into too small of a space would leave no room to manoeuvre, frustrating the player.

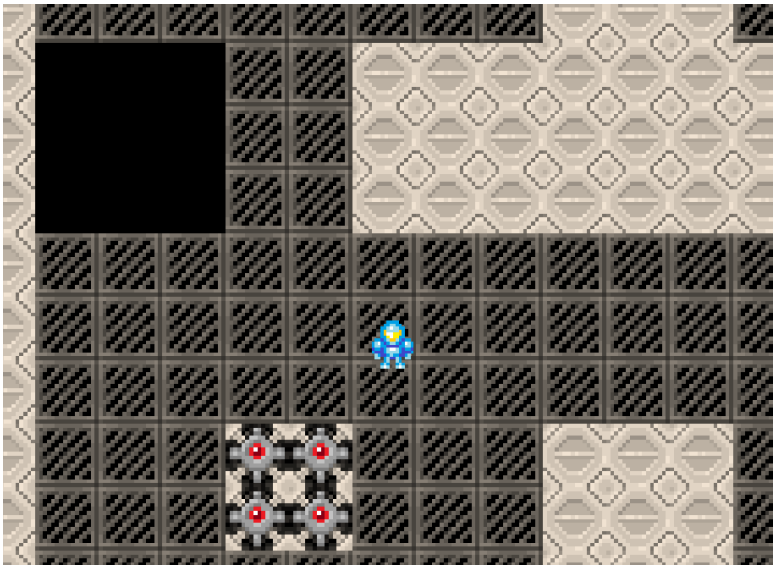


Figure 19: A Bridge (Top) and Corridor (Right)

After level geometry is established, additional electrode obstacles are placed into the level with dual purpose. They damage the player if touched, but also serve as soft cover for the player to hide behind from enemy vision and projectiles. Additionally, they can be shot to turn into an empty tile as in Figure 20.

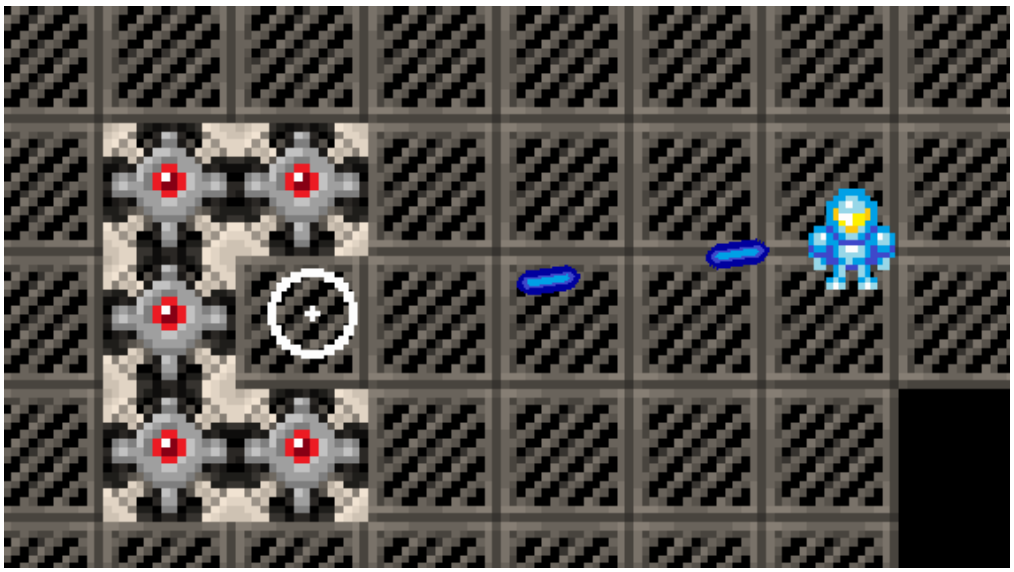


Figure 20: Electrodes Being Shot

3.6.2 Item and Character Placement

Finally, spawns for the player, the final family, robots and items are decided. The player is spawned with a safe starting distance from robots, while final family members and items are scattered throughout the level. To satisfy these design choices, the six chunks of level geometry are each given a specific type. An overview of each chunk type is given in Table 2. One family members is spawned in each mixed chunk. Family members are spawned in mixed chunks to give them a better chance to run away from enemies while not being too easy to collect. Electrodes are only spawned in mixed chunks to give more space to the player and enemies in their dedicated chunks. To give mixed chunks more variety, each

one has a fifty percent spawn chance for each of items, enemies and electrodes.

Chunk Type	Items	Family Members	Enemies	Electrodes
Player Chunk	✓	✗	✗	✗
Mixed Chunk	✓✓	✓✓	✓	✓
Enemy Chunk	✓	✓	✓✓	✗

Table 2: Chunk Types

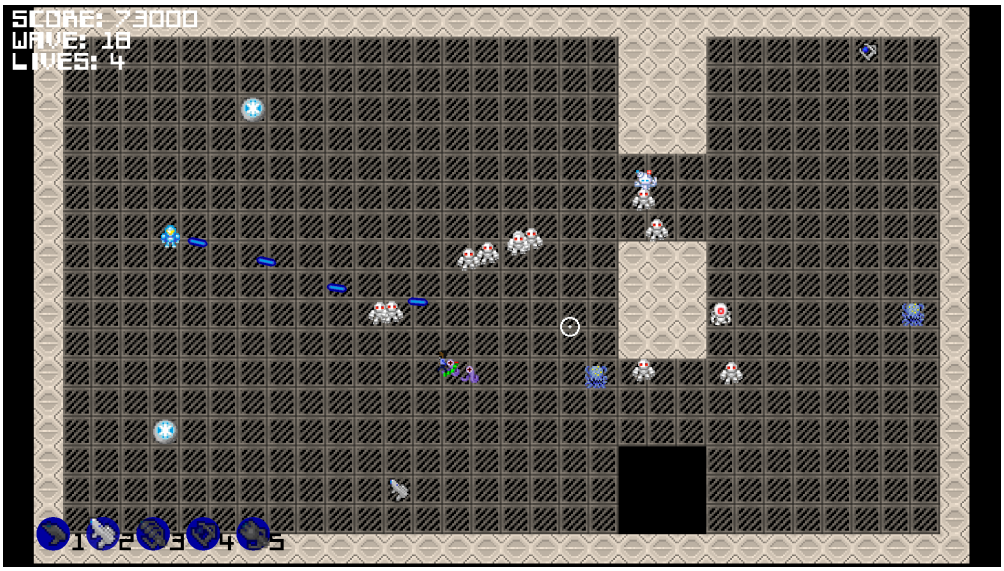


Figure 21: A Fully Generated Map

3.6.3 Wave Progression

The difficulty of each wave is determined by the wave number, which is used in different level parameter calculations. The number of robots is constantly increased across waves. The counts of easier enemies are increased faster than more challenging enemies. The range of enemy counts in the enemy chunk is listed below in Table 3. For mixed chunks, one enemy type is chosen randomly with half of the count. Turrets are additionally spawned in pits with rate random(wave/3).

Enemy Type	Number of Enemy
Grunt	5 + random(wave)
Heavy / Laser	random(wave/3)
Rocket / Flying	random(wave/3)
Turret	random(wave/10)
Worm	random(wave/4)
Brains	random(wave/5)
Plasma	0 (Spawned by Brains touching family)

Table 3: Wave Progression

3.7 Audio and Graphics

The design theme for the game was chosen as sci-fi to match the theme of the original game. A suitable sprite set was found online for most graphics [2]. This helped to make graphics

consistent. However, the original character sprites were static, so were modified to provide simple animations. The family members use a different sprite set that had animations in each direction [3]. One further addition was the inclusion of a custom Robotron font matching that of the original game [4]. The laser graphic was adapted from [5]. The laser sounds were taken from [6]. The crosshair graphic is from [7].

4 Implementation

4.1 Player

Player lives are directly mapped from points, with two additional lives given. When the player runs out of health, a lives used counter is incremented. Once the lives used goes higher than the lives gained, the game is over. Family members are collected by the player when they distance is low enough.

4.2 Weapons

Lasers can be fired if the number of shots available of a weapon is higher than zero. A timer is used to recharge the number available over time. Laser hit detection works as described in Design Section 3.3. The position of a standard laser can be checked to determine whether it has hit a wall, electrode or character. Raycasting is performed by taking the vector to the target and checking positions along the vector in small steps. This allows the railgun to determine how long to make the ray. Player weapons are kept in a list. A map from integers to weapons is also used to let the player switch weapons. The integer values of the number keys on the keyboard (1, 2, 3...) are used as the key to access the map. This can be converted back to a char to be displayed in the weapons display.

4.3 Items

Items modify the statistics of players and enemies directly. Freezing enemies sets a flag that prevents them from firing and moving. To prevent conditions where statistics are reset incorrectly, collecting the same power-up twice simply resets the timer. If this were not done, then collected two speed boost power-ups could reset the player's speed to what it was with one boost active.

4.4 Shop and Options

The shop and options menus hold values that can be checked by other parts of the code directly. As the object management system has limited control over render order, the level is cleared when a menu is accessed. This involves destroying any objects in the level and hiding the player's sprite for the options menu. To prevent the player from losing anything when accessing a menu or going to the next wave, any family members and weapon items are collected automatically.

4.5 Level Manager

The level manager holds a 2D grid of cells that store level tiles. It also includes code to convert between grid coordinates and screen coordinates. This requires calculations using the position in one coordinate system as well as the cell size used. To fit the level onto multiple aspect ratios, an offset in the screen is added (representing black bars). This works well for wider aspect ratios, but less for narrow aspect ratios. This offset must also be taken into consideration when converting between coordinate systems.

When characters move in the level, their coordinates are checked to support collisions. If a character were to move into a wall, their velocity in that direction (x or y) is ignored. To prevent characters from getting stuck on corners, they are given a speed boost along the wall when the velocity in only one direction is cancelled. This is done by changing the direction of the component into the wall.

Turret enemies can rarely spawn in chunks, but are mostly placed inside of pits. This involves tracking a list of pit tiles and then choosing random tiles from this list to place Turrets in.

States are used to manage the game loop. These include the welcome, level, post-level and game over states. Welcome and game over states are similar, allowing the player to change options before starting a new game. The level state represents the main part of the game loop where the player tries to defeat the enemies of a wave. The post-level state represents the state when the player has defeated all the enemies of a wave, providing them with the option to access the shop.

4.6 AI Behaviour

All AI make decisions based on a pathfinding and vision algorithm. Most enemies like Grunts will focus solely on hunting the player, while Brains and Worms will also seek out final family members. Final family members will roam around in search of the player and avoid any robots that present a threat to them. The A* search code completed in a tutorial was extended to read a Robotron 4303 map. AI can then use the path finder provided to navigate the map, following the closest cell on the path and removing it once close enough. As electrodes can be shot and converted to empty tiles, the constructors for these tiles adjust the path finder to take the change into account.

Vision uses raycasting as described previously, with small steps taken to the target to check whether any tile between the character and target blocks vision. Furthermore, it can be checked similarly whether the path crosses a pit. This allows the AI to decide how to go to a target as in Figure 22. If the current target character can be reached directly (no pits), then the velocity is set directly towards the target. If the target character is visible across a pit, then an A* path is used instead. A* paths are also used for going to the last place a target was seen, idle behaviour and family member evasion. Family members evade targets by checking the distance from adjacent cells to threats and following those that have a higher distance.

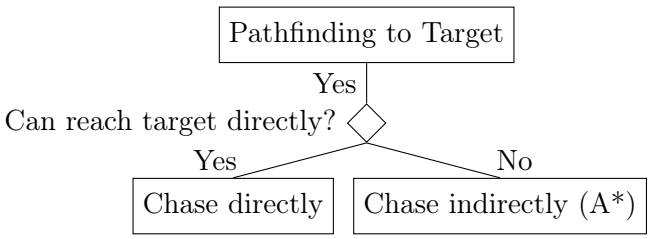


Figure 22: Pathfinding to the Current Target Character

4.7 Animation

An Animator class was created to handle loading of animations. Each character with animations is given a folder for its frames. Animation frames should be titled with the direction they are for (down, up, right or left) and a frame number (1, 2, 3...). Each animated character should also have a still frame, labelled with direction and 'Still' instead of frame number. Animated characters must at least have frames for the down direction

(facing the screen). References to down frames are copied to other directions if a character does not have full directional animations available. This does animate the character, but they will always be facing the screen.

4.8 File and Class Management

The inheritance system from the previous practical was adapted and extended in this practical. To make files tidier in the game directory, related classes were moved together as much as possible. Files were named with prefixes denoting their category. Later parts of file names extend these categories.

The base system has prefixes `CORE`, `PHYSICS` and `UI`. Robotron code is given its own prefix. A scene class was created, which `Robotron_Core_Robotron` inherits. This holds the top level of updates and passes on events such as key presses to relevant classes. When Robotron classes want to access the level manager, for instance, they cast the globally available current scene variable as `((Robotron)currentScene).levelManager`. The shop and options menu were edited directly in the base code, but could be extended and cast similarly in future.

5 Testing

The game was playtested to help build a better balance and discover bugs that a player may encounter. This involved playing several rounds with different options and shop upgrades to ensure each feature works as expected. During development, enemies were selectively spawned to study their behaviour. The options menu gives players more control in case they do not like the standard balance.

One issue from the previous practical that was resolved is the sound corruption from the Sound package. Previously, playing a large number of sound effects in short succession would result in audio delay and corruption. A function was created that uses a map of sounds, indicating whether each has been played during a frame. Each sound is only allowed to play once during a frame. Furthermore, a sound is stopped before playing it again. Not stopping the playback of a previous sound is likely what increased the chance of corruption. Further testing over longer sessions is required to determine whether this issue is fully resolved or takes longer to occur.

Another issue during development was with family member evasion. Sometime family members would run back and forth in the same spot when running from threats. To fix this, enemy evasion is halted when family members see the player. The bug did not reoccur in short testing. Longer testing would be beneficial.

6 Evaluation

Several key areas reflect positively on the game. Figure 23 details these.

- Procedural Level Generation: The levels generated by the game fit the style of the original Robotron game while offering unique maps, weapons and power-ups to add variety.
- Weapons: The arsenal that the player can collect each have a unique style that encourage the player to swap between them, keeping evading and shooting enemies engaging.
- Options and Shop: The options and shop menu let the player customise their experience and feel progression across waves.
- Enemy Variety: There is a large variety of enemies with their own behaviours and synergies, serving as engaging opponents for the player.
- Graphics: Having most assets from a single source means that the various sprites fit well together to form a common theme.

Figure 23: Positive Highlights

Some areas for future work are identified in Figure 24.

- Sounds: While a few sounds have been added, the game could benefit from more. Hit sounds could help provide feedback. Weapon sounds could be more distinct. A sound effect for rescuing family members could help raise the player's mood.
- Family Member Evasion: The current evasion algorithm could be tested further and refined to improve family member survival chance.
- Enemy Lasers: Each shooting enemy uses the same kind of laser. One idea during development was to give enemies player weapons with weaker statistics. This would require careful some additional consideration on how to keep this efficient and disconnected from settings and upgrades affecting the player's weapons.
- Code Quality: While sensible variable names were used to make code more readable, there is a lack of comments detailing the code. The lack of commenting utilities for Processing make this a harder job than for standard Java code.
- Balancing: While attempts were made to balance weapons, enemies and upgrades, more testing is required to improve this. Some weapons and upgrades may be stronger than others, such as the pulse cannon and railgun.
- Shop Ease of Use: The value of upgrades differs. For example, a damage upgrade from 30 to 35 would be much more useful than one from 35 to 40. Furthermore, some upgrades become redundant. For example, upgrading recharge rate after a weapon already recharges faster than it fires is useless. Including information on statistics after upgrades and limiting useless upgrades could help the player in their choices.
- Additional Content: Additional weapons, enemies, map layouts, options and upgrades would help give the game even more variety.

Figure 24: Future Work

7 Conclusion

The final implementation provides a well-developed variant of the game Robotron 2084, together with additional features not found in the original to allow for more varied gameplay. This includes multiple weapons, a shop and options menu. With additional time, more polishing and balancing could be done in areas like audio, enemy counts and AI behaviour.

References

[1] C. Sanyk, *Robotron: 2084 and zookeeper*. [Online]. Available: https://csanyk.com/2012/06/robotron-2084-and-zookeeper/ (visited on 24/02/2024).

[2] E. Flesher, *Sci-fi roguelike pixel art*. [Online]. Available: https://opengameart.org/content/sci-fi-roguelike-pixel-art (visited on 25/02/2024).

[3] T. Gamblin, *48 animated old school rpg characters (16x16)*. [Online]. Available: https://opengameart.org/content/48-animated-old-school-rpg-characters-16x16 (visited on 25/02/2024).

[4] P. H. Lauke, *Robotron 2084 font*. [Online]. Available: https://fontstruct.com/fontstructions/show/474939/robotron_2084 (visited on 24/02/2024).

[5] V. Vanhatupa, *Bullet collection 1 (m484)*. [Online]. Available: https://opengameart.org/content/bullet-collection-1-m484 (visited on 13/03/2024).

[6] Kenney, *Sci-fi sounds*. [Online]. Available: https://opengameart.org/content/sci-fi-sounds (visited on 14/03/2024).

[7] Calinou, *Simple crosshairs*. [Online]. Available: https://opengameart.org/content/simple-crosshairs (visited on 10/02/2024).