CS4303

Practical 3 - The Game

190004947

May 10th, 2022

University of
St Andrews

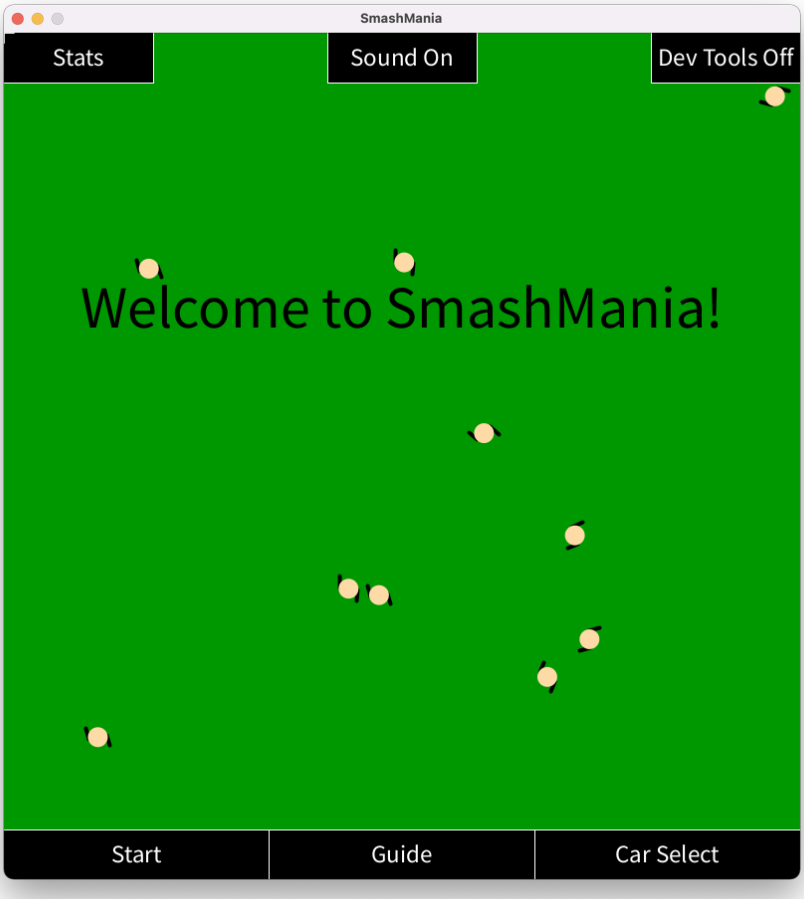
Word Count: 4,644

Contents

1 Introduction

My initial game plan which I pitched was called Traffic Mania and was a racing puzzle game in which the player would have to find their way to a destination using only road signs in the fastest time possible. After my one-to-one meeting with Joan where we discussed the game, I was cautious as to how fun such a game would be. Once I had implemented the driving mechanics for the game, the car felt fluid and smooth which pushed me to change creative direction. I decided that there needs to be police chasing the player car which I could implement using the kinematic and steering movement algorithms we saw in the AI lectures. Once I had implemented the first iteration of these police cars I decided to pivot the game objective from finding a destination to earning cash by running over pedestrians while trying to survive the police chase.

After completing the development of my game, I am delighted that I made this switch as the resultant Smash Mania has a much higher pace than the initial Traffic Mania leaving a more exciting play through.



2 Design Decisions

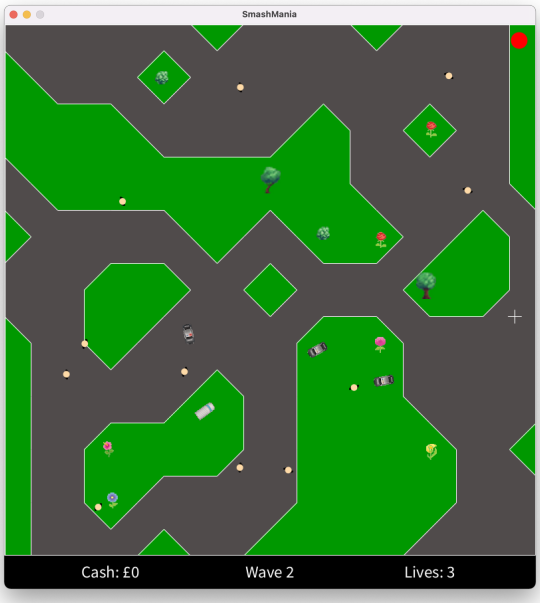
2.1 Title and Genre

My game is called Smash Mania and it is a single-player arcade game. It could also fall into the categories of driving and survival. Being primarily an arcade game, I chose sound effects for each event accordingly (look out for the Wilhelm scream when killing a human - it is a 1/100 chance). The music also follows the arcade theme and was produced by my younger brother whom I challenged to produce a better version of SmashMania using his game development engine.

2.2 The Play Area

During a wave, the play area will display the player car, the police cars, and the humans on top of a procedurally generated map decorated with trees, flowers, and grass. Present on all pages of the game is the databar; a black bar at the bottom of the screen which displays either relevant information or buttons to navigate to other parts of the application. During a wave, this bar displays the current cash balance, current wave, and number of remaining lives.

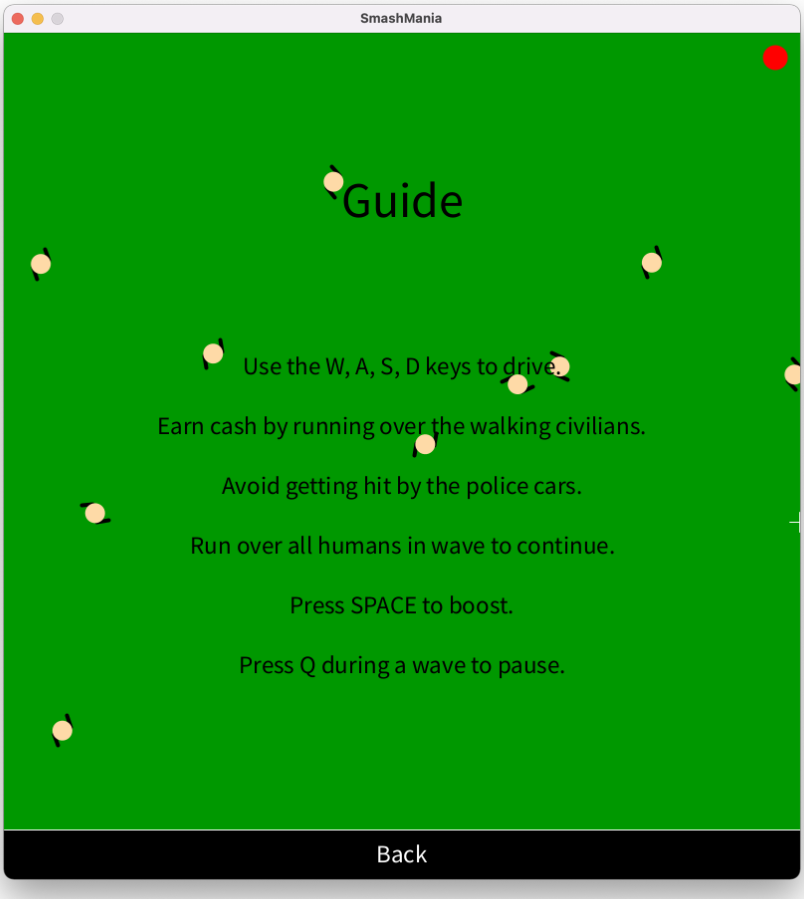
Additionally, if the player has unlocked any power ups, then ability cooldown bars will be displayed on the top left of the screen. Lastly, a small red circle will be displayed in the upper right corner if the developer tools are enabled.



2.3 Control Scheme

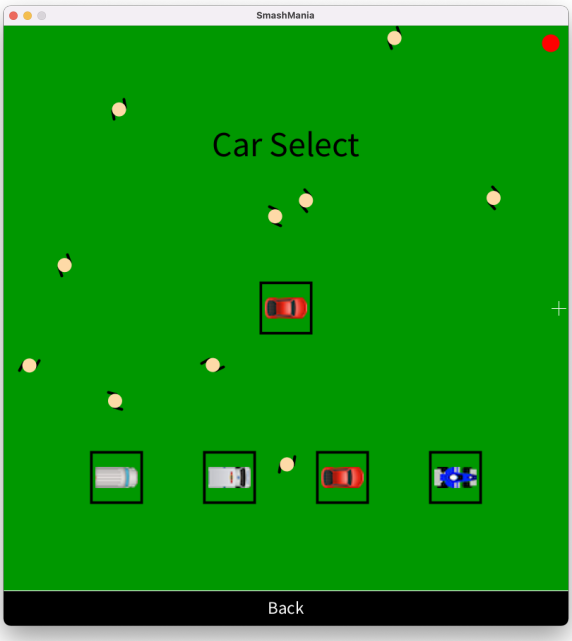
Smash Mania makes use of both keyboard input and mouse input. The player can navigate the various pages in the game by pointing and clicking on buttons displayed in the databar which are dynamically presented following the flow diagram shown in the GameState section.

The core mechanic of the game, driving, uses the WASD keys where 'W' is accelerate, 'A' and 'S' are turn left and turn right, and 'D' will either brake or reverse depending on context. If the boost ability is unlocked then it can be activated during a wave using the spacebar. Additionally, if the pulse ability is unlocked, it can be activated using the 'E' key placed conveniently by the driving keys. The final ability, air strike, is used by pointing and clicking on a position on top of which to drop the strike. In addition to these controls, there are also a number of developer tools which I used when building the game and can be used by the marker to test different game elements without having to play through over and over again. These extra controls will be explained in the 'Developer Tools' section below.



2.4 Player

The player in Smash Mania controls a car driving around a two-dimensional plane. New cars can be purchased in the item shop with cash earned from waves and the player can select which car they want to use on the car selection page. Each car has different properties, the standard minivan being the baseline. The cheapest unlockable car is the ambulance which is slower and has a wider turning circle, however, due to irony, the ambulance will earn 1.5x the cash compared to the other vehicles. The next car is the sports car which is faster and has a tighter turning circle than the minivan. Lastly, the F1 car is the fastest and has the tightest turning circle, however, it also has the worst grip on grass which can be alleviated by purchasing the off-road tyres.



2.5 Opponents

The opponents are police cars that will relentlessly chase the player around the play area. The player will lose one life should they come into contact with one of the police cars. Since the police cars use steering pursue, they will attempt to intercept the player's line of motion rather than chasing behind it.

The police cars can actually be eliminated in multiple ways. The first of which is simply through collision with the player car which will deduct a life and eliminate the police car. The other two ways are by using the pulse or air strike abilities.

2.6 Rules and Mechanics

A wave is considered complete when all of the humans wandering the play area have been killed one way or another. They could be killed by running them over with the player car, using an ability, or even by being run over by a police car. The player starts with three lives which cannot be replenished. I considered allowing the purchase of extra lives in the item shop but ended up deciding against it as it took away from the intensity of higher levels. The game is over when all three lives have been lost. To be able to reach higher levels, the player must purchase better cars and abilities. Cash can be earned for killing a human or eliminating a police car. The values are as follows:

- £100 for running over a human, unless the police ran them over which does not award cash.
- £150 for running over a human with the ambulance.
- £50 for killing a human with a pulse or air strike.
- £100 for eliminating a police car with a pulse or air strike.

The player car experiences friction as a sub-one multiplier to velocity each frame. If there were no friction then the player car would maintain its velocity and fly around instead of slowing to a halt. The amount of friction applied is calculated using an initial friction multiplier as well as a ratio which is multiplied by the difference between the player car's orientation and the direction of motion. The result of this is that friction is higher when the car is sliding perpendicular than when it is sliding forwards or backwards.

```
// Get the direction of momentum
float getVelocityOrientation() {
    return velocity.heading();
}

// Calculate friction proportional to how close the car orientation is to perpendicular with the direction of momentum
float getFriction() {
    float difference = Math.abs(orientation - getVelocityOrientation()) / PI;
    if (difference > 1) {
        difference -= 1;
    } else if (difference < -1) {
        difference += 1;
    }
    float ratio = 0.0;
    if (difference <= 0.5) {
        ratio = 2 * difference;
    } else {
        ratio = -2 * difference;
    }
    ratio = Math.abs(ratio);
    if (ratio > 1) {
        ratio = 1;
    }
    return (friction - (ratio * frictionMultiplier));
}
```

The generated maps consist of grass space and road space. When driving on the grass, cars will experience a slower max speed and the player car will experience worse friction. Different cars will react differently to the grass however. Note that these effects can be subverted by purchasing the off-road tyres.

```
// Check if player car is on the road or the grass and set max speed and friction accordingly
if ((map.closestPoint(playerCar.position) == 0) && !offroadTyresUnlocked) { // Grass
    switch (playerCar.type) {
        case 0:
            playerCar.maxSpeed = PLAYER_CAR_MAX_SPEED * GRASS_MAX_SPEED_MULTIPLIER;
            playerCar.friction = PLAYER_CAR_FRICTION * GRASS_FRICTION_MULTIPLIER;
            break;
        case 1:
            playerCar.maxSpeed = 0.75f * PLAYER_CAR_MAX_SPEED * GRASS_MAX_SPEED_MULTIPLIER;
            playerCar.friction = 0.01f + (PLAYER_CAR_FRICTION * GRASS_FRICTION_MULTIPLIER);
            break;
        case 2:
            playerCar.maxSpeed = 1.25f * PLAYER_CAR_MAX_SPEED * GRASS_MAX_SPEED_MULTIPLIER;
            playerCar.friction = (PLAYER_CAR_FRICTION * GRASS_FRICTION_MULTIPLIER) - 0.05f;
            break;
        case 3:
            playerCar.maxSpeed = 1.5f * PLAYER_CAR_MAX_SPEED * GRASS_MAX_SPEED_MULTIPLIER;
            playerCar.friction = 0.01f + (PLAYER_CAR_FRICTION * GRASS_FRICTION_MULTIPLIER);
            break;
    }
} else { // Road
    playerCar.updateCarStats();
}
```

The game includes three abilities at the players disposal which are all locked at the beginning of the game. The first of which is a boost which has the fastest cooldown time and gives the player car a short burst of speed. Next is the pulse ability which pulses a blue wave out from the car vaporising anything it touches. Last is the air strike, only available once per round, the player can click a position on the map to drop five bombs around that area also destroying everything it touches. The cooldown for each ability can be seen in the top left corner of the play area once they are unlocked.

When the player loses all of their lives and the game ends, the current cash balance is wiped but all unlocked cars and items remain. This gives the player the ability to reach higher and higher waves as they unlock better cars and power ups.

2.7 Goals

The goal of Smash Mania is two-fold. First and foremost, the player should attempt to reach the highest wave possible by avoiding collision with the police cars and running over humans. Secondly, the player should aim to earn enough cash to unlock all of the item shop items. By making the ambulance earn 1.5x cash, a player can decide to make the trade-off of car performance for cash.

2.8 Context

Smash Mania, while an independent game, can be compared to the original Grand Theft Auto title which was also a two-dimensional top-down game which would often have the player driving around running over pedestrians. Of course the GTA games goes much deeper than this but I think it is the closest relative of Smash Mania.

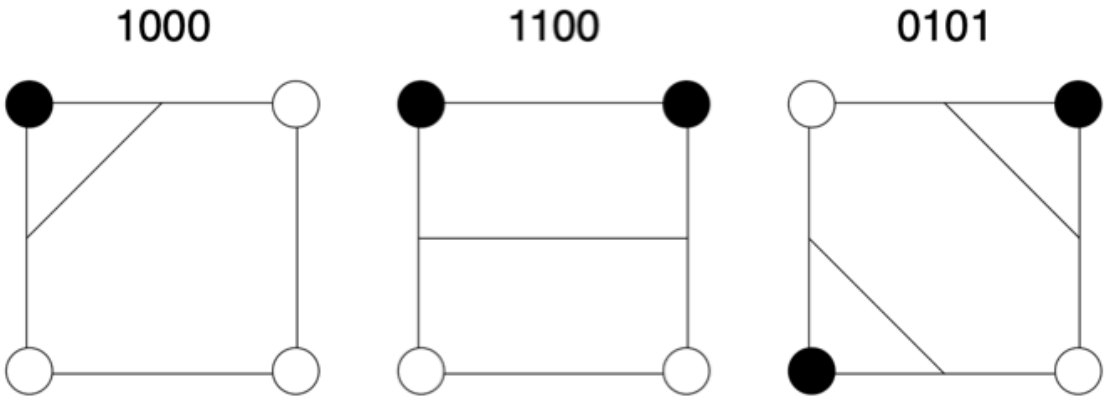
Another feature of the game, the item shop, is now commonplace in modern titles. This comes with the inclusion of micro-transactions through which players can buy in-game currency. This monetisation of in-game items can replace the revenue from game sales and allow game development studios to offer their games for free. This free-to-play phenomenon has blown up after Epic Games successfully released a free title, Fortnite, and generated over $5 billion in 2020 alone.

The arcade nature of the game of trying to reach a new highest wave was popularised before this in the industry of mobile gaming. With games such as Flappy Bird amassing over 50 million players, many companies and individuals raced to build the next big hyper-casual game. The addiction that can come from pursuing a new highest score proved to be a winning formula that saw thousands of similar games produced since the immensely popular Flappy Bird was released.

2.9 Map Generation

A new map is generated for each wave played using a marching squares algorithm. It begins by splitting the play area into a grid of points including rows and columns at 0 and at the maximum width and height. Each of these points are randomly assigned either a 0 or a 1, the former representing a grass zone and the latter representing a road zone. Now defining a square as the space between four assigned points, we iterate over each square converting the value of its adjoining points into a binary string. This string concatenates the values of the NW point, the NE point, the SE point, and the SW point in that order. The binary string is then converted into a decimal value which is passed into a switch statement. Depending on the value inputted, different space will be sectioned off as road.

Examples where black indicates 0 and white indicates 1:



2.10 Wave Parameters

The first wave begins with 10 humans to kill and 3 police cars to avoid each with a max speed of 1.0. Each successive wave will have one additional human and will increase the maximum speed of the police cars by 0.1 up until wave 20. Each of these variables that increase the difficulty of the game are set by global variables in the main SmashMania file which allowed me to optimise them to produce a fun but challenging experience. The final values are a result of an iterative optimisation process.

```
1  // Update number of humans
2  numHumans += HUMAN_COUNT_INCREMENT;
3
4  // Update number of police cars
5  if (wave % POLICE_CAR_WAVE_INCREMENT == 0) {
6      numPoliceCars++;
7  }
8
9  // Update police car max speed
10 if (wave < 20) { // To stop police cars from going too fast
11     POLICE_CAR_MAX_SPEED += POLICE_CAR_SPEED_INCREMENT;
12 }
```

2.11 Developer Tools

The developer tools allow the game tester to control the game outside of what would be legally available to a player. They can be toggled by clicking the 'Dev Tools' button on the homepage. If the developer tools are on then a small red circle will appear in the top right corner of the game to indicate that they are enabled.

When developer tools are enabled, the bottom row of keys ('z' to '/') can be used to jump to different game states. This can be used not only to test each page of the game but also to skip waves to test higher waves. This can be achieved by pressing 'M' during a wave to skip to the post-wave screen of that wave. Additionally, the arrow keys can be used to change the player car type at any time regardless of which are unlocked. The top row of keys ('1'-'-') are also reserved for developer actions. Number keys 1 through 9 will unlock each of the items that are for sale in the item shop while the 0 key will add £1,000 to the player's cash balance. Lastly, the '-' key will increment the number of lives that the player has.

2.12 Evaluation

When building my last game, Robotron4303, I spent hours implementing collision detection algorithms that would check each element against every other element to test for collisions. This proved extremely unfruitful, especially in the case of collision detection with the procedurally generated map. I ended up refactoring the codebase to use a quad tree which could pick specific points to be used in the collision detection checks massively reducing computational complexity. Since this was so powerful in maximising performance I used a quad tree from the start when implementing collision detection for Smash Mania. Also inspired by Robotron4303, I kept all my collision detection methods in a single utility class called CollisionDetectionUtils which will be explained below.

When developing each element of the game, I ensured that it's parameters were set using global variables initialised in the main SmashMania file. This allowed me to quickly and easily tune each part of the game all in one place during testing. I took advantage of this design as much as I could by playing my game over and over again tweaking parameters of the game to optimise the balance between fun and challenging.

Through testing I iteratively improved upon:

- Amount of cash awarded for each event.
- Performance stats for each car.
- Initial wave parameters.
- Wave parameter increments.
- Scale of each game element including the map resolution, car sizes, and human size.
- Power of each ability.
- Cooldown time for each ability.

Perhaps most extensively I tested the friction mechanics of the player car. Since the friction applied to the player car is dynamically calculated such that it will experience more friction while sliding sideways, it took a while to find a set of parameters that applied this logic without ruining the smooth feeling of the driving.

Once development was mostly complete, I created a class for map decorations (which is explained below). When I had my decorations working correctly, I decided to implement collision detection between the cars and the trees. After implementing the collision detection logic in its entirety such that cars could smoothly drive into a tree and slide away from it without abruptly crashing, I scrapped it all. Through testing the game, it became clear that tree collisions interrupted the flow of the game and was an additional variable to keep track of which was extremely difficult on higher waves where the police cars are faster and more plentiful.

3 Classes

3.1 Overview

My final submission includes 12 unique Processing classes each serving a specific purpose. Some are dependent on others such as QuadTree being inherently dependent on BoundingBox and Point. The main SmashMania file contains all of the global variables for the game so that parameters can be tuned from one place as described in the evaluation section above.

When designing an architecture for the application before development, I considered making the player car and police cars inherit from a parent class storing car mechanics. However, due to how little the two implementations overlap, I decided against it. The air strike is the only ability that has its own class. This is because boost and pulse could both be easily implemented within the player car class whereas the air strike needed its own variables, logic, and draw function which is best kept in its own file.

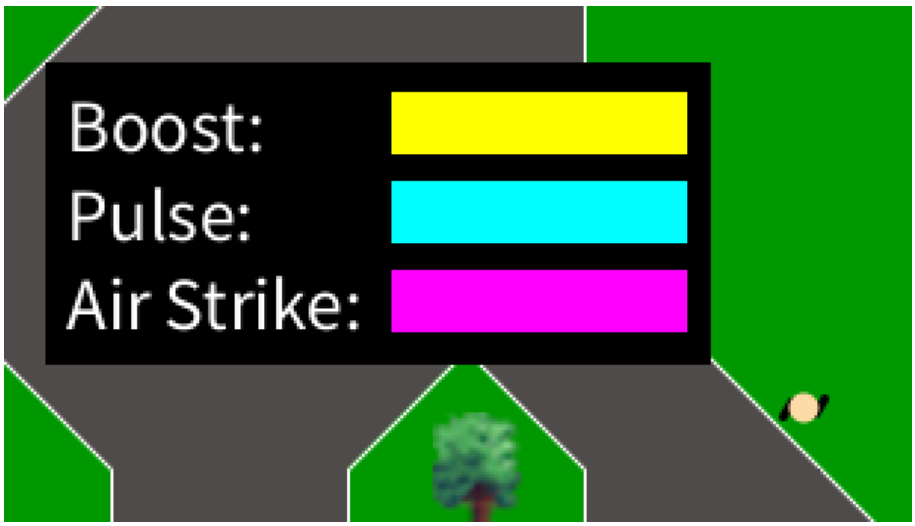
As in my Robotron4303 implementation, I have kept all of my collision detection methods in a separate class called CollisionDetectionUtils. Unfortunately I had to rewrite all of the methods to account for the rotation of the rectangles. Additionally, I used a different 'rectMode' in this implementation and so I had to think about the positions of rectangles from the centres rather than from the top left corners. The protocols I used to deal with the rectangle orientation is described in the CollisionDetectionUtils section below.

3.2 SmashMania

The main file is the SmashMania file. It contains the logic for each phase of the game and wave setup as well as all of the collision detection. As the glue that ties all the classes together, it contains the global variables that are used in the constructors of most other classes. The SmashMania file contains 26 methods: settings and setup for initialising variables, various draw functions, some helper methods, wave management methods, and functions to parse input.

The draw function here contains a switch which takes the current phase from the game state object as input and chooses which screen to draw by calling the appropriate draw function. For example, if the phase is homepage, then drawHomepage would be called. By using a switch like this it was very easy to keep each screen separated and to manage the flow of state through the game.

As well as the specific page draw functions, I also wrote some extra draw functions like drawLock and drawCooldownBar. The first of which of course draws a lock while the second takes a timer and a max timer as input and draws a cooldown bar with the correct proportion of the bar highlighted. Keeping this in its own function made it much easier to show the cooldown for each ability while being more intuitive than a text timer.



The two wave management methods, setupNextWave and resetWaves, set up the global game parameters for the next wave implementing the appropriate difficulty adjustments and reset all global parameters to initial values respectively.

Lastly, we have the three input handlers: keyPressed, keyReleased, and mousePressed. While the keyboard input handlers were simple if statements that check the key pressed and maybe apply a contextual condition based on which page the player is on, the mouse input handler was a little more complex to implement. After designing a flow diagram to show how a player should be able to move through the different pages of the game, I added buttons to the databars of the non-wave pages. Then in mousePressed, depending on which page the player is on, the X and Y coordinates of the clicked location are checked and compared to the locations of the buttons. If the click was within a boundary of a drawn button, then the relevant logic would execute. The primary purpose for mouse input was navigation through the application but later it ended up also being used to drop an air strike when in a wave.

3.3 PlayerCar

The PlayerCar class represents the player. It manages how the player is drawn, the driving mechanics, as well as the boost and pulse abilities. Drawing the player car consists of simply translating the axes by the current car position, rotating the axes by the current car orientation, and then drawing the correct sprite on the origin. Drawing the correct sprite only checks an integer variable that stores the car type and draws the relevant sprite, however, the ambulance has three states. To achieve animation of the ambulance, I used modular arithmetic with the global frameCount variable to manage which ambulance sprite was drawn.

```
// Draw the correct player car sprite, if ambulance, animate it
imageMode(CENTER);
if (type == 0) {
    image(minivan, 0, 0, width, height);
} else if (type == 1) {
    if (frameCount%90 < 30) {
        image(ambulanceOne, 0, 0, width, height);
    } else if (frameCount%90 < 60) {
        image(ambulanceTwo, 0, 0, width, height);
    } else {
        image(ambulanceThree, 0, 0, width, height);
    }
} else if (type == 2) {
    image(sports, 0, 0, width, height);
} else if (type == 3) {
    image(f1, 0, 0, width, height);
}
```

The boost and pulse abilities are controlled within this class in a similar manner. Both have a boolean flag to show whether the ability is active as well as a flag to show whether the ability has just been used. They also both have two timers: one to count how long the ability lasts for and one to count down the cooldown period. Once this was implemented, I decided to draw yellow boost lines coming from the back of the car when boosting and a blue pulse emanating from the car when pulsing.

```
if (pulsing) {
    fill(0, 255, 255);
    if (pulseTimer < maxPulseTimer/2) {
        currentPulseSize = pulseSize * pulseTimer/(maxPulseTimer/2);
        ellipse(0, 0, currentPulseSize, currentPulseSize);
    } else {
        currentPulseSize = pulseSize * (maxPulseTimer - pulseTimer)/(maxPulseTimer/2);
        ellipse(0, 0, currentPulseSize, currentPulseSize);
    }
}
```

3.4 PoliceCar

The police car class uses the same logic as the player car class to draw and animate the sprites but obviously uses very different logic for movement. The primary behaviour of the police cars is to use steering pursue as seen in the AI movement lectures. However, the class also includes kinematic flee and steering flee behaviours which I called from the SmashMania file to make the police cars avoid each other and avoid hitting humans. Initially, I implemented the police car AI behaviour with a finite state machine that determined what it should do as I did with the Robotron practical. However, changing from pursue to flee caused very strange looking police car movement. The best solution ended up being to leave the police cars using pursue and to call kinematic flee from the main file when another police car or a human entered its vision radius. This caused a cumulative movement away from the target and towards the player car.

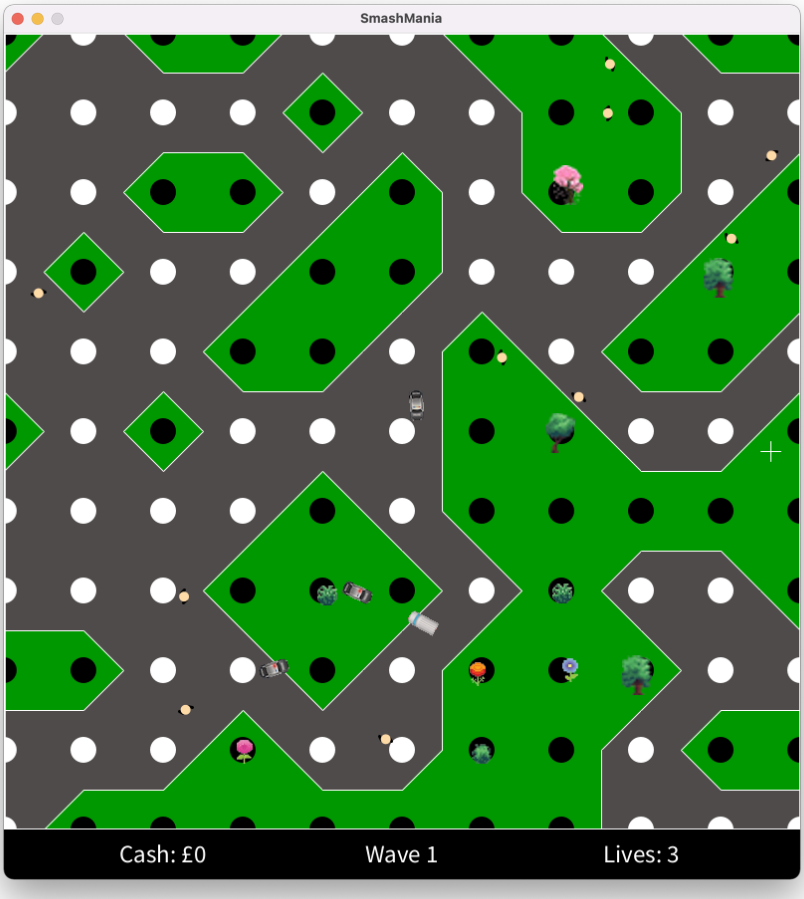
3.5 Human

The human class stores and manages the human's state, alive or dead, and draws them accordingly. If the human is marked as not alive, then it is drawn as four randomly sized blood splats placed randomly around the position that the human was killed. If the human is alive, then it is drawn as a beige circle with swinging arms. Before I arms, they didn't look very much like people which I wanted to be obvious and so I came up with a function which uses a sine wave alongside an age ticker to periodically make two arms swing backwards and forwards. The best solution after many iterations is shown below.

```
line(-(size/2)+1, 0, -(size/2), 2*(size/3)*sin(2*(PI/100)*age%100));
line((size/2)-1, 0, (size/2), -2*(size/3)*sin(2*(PI/100)*age%100));
```

3.6 Map

The Map constructor takes as parameters the game width and height, a resolution, and some parameters defining how decorations are placed. This resolution variable refers to how the game width and height are divided into rows and columns of points for the marching squares algorithm. After trying both higher and lower resolutions producing more and less detailed maps respectively, I ended up choosing a resolution of 80 which leaves 10 squares vertically and horizontally across the play area, each with four binary points on each corner.

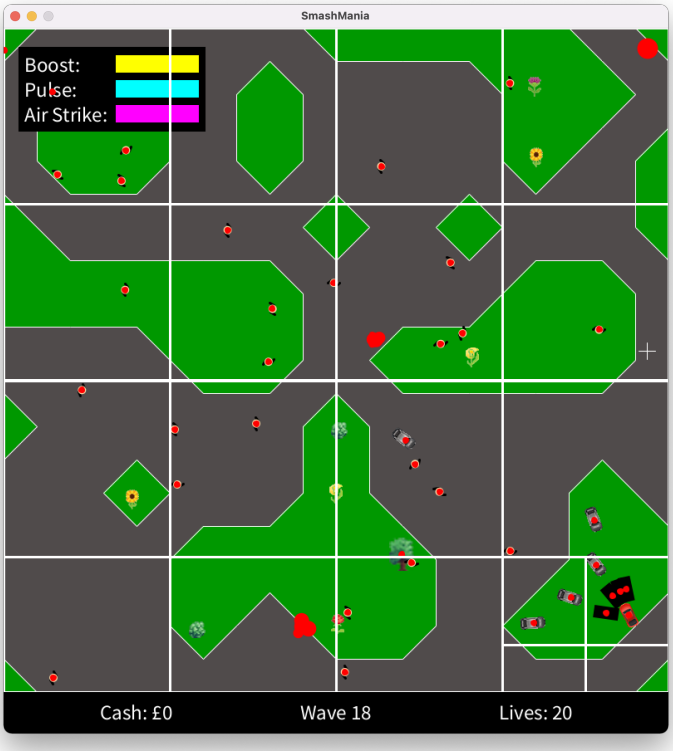


For decoration placement, the map's generate function randomly selects points labelled as 0 for grass to place decorations on. It then adds a random offset to this position as to avoid a uniform looking decoration placement. It concludes with a while loop asserting that no decorations will be overlapping each other.

3.7 QuadTree

The quadtree class stores a 4-ary tree of space partitions which is used to find points and objects within a given range in the game space in O(nlogn) time. It stores a boundary which is the overall space which the quadtree node claims, a capacity which is the maximum number of points that a quadtree node can hold before it subdivides, an ArrayList of points which are the points within its boundary, a boolean flag to mark if the node has been divided or not, and finally it contains references to four other quadtree objects. These four quadtree references are initially null when the quadtree object is instantiated but will store the children of that node if and when it is split.

The quadtree class has a recursive insert function and a subdivide function for when the capacity is reached. It also has a query method. Calling query with a space in the game as a parameter will return all points within the space passed in. These points can then be used for collision detection. Comparing this to a standard collision detection case where every point is checked against every other point, we now only have to check each point against all neighbouring points. I also implemented a draw function in the quadtree class, activated using the 'J' key, which draws the boundaries of every node in the tree as well as highlighting the position of every point stored by calling draw on the individual points.



3.8 BoundingBox

BoundingBox is also a very simple class and is another requirement of the quadtree. It simply represents a quadrilateral space by storing the position of the north west corner as well as a width and height. The bounding box is used to specify a range for the quadtree query function as well as for storing the boundary of a quadtree node.

3.9 Point

The point class is very basic. It is the object that is inserted into the quadtree. It stores a position and a reference to some object. This object could be anything from the player car to a human. When I query the quadtree for points in a given range, I can dereference the object (which is Object type) and run the relevant collision detection algorithms on the returned points.

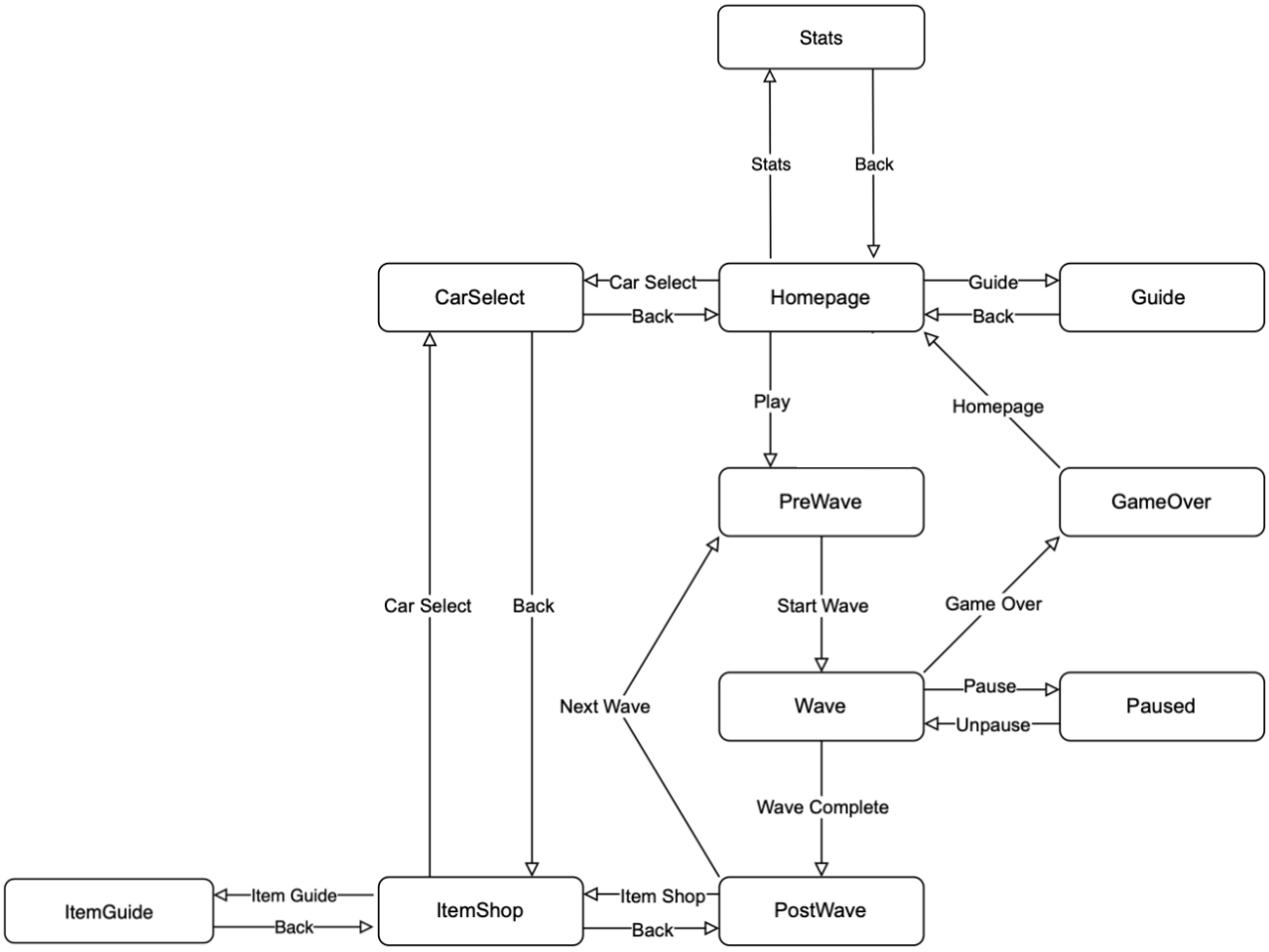
3.10 CollisionDetectionUtils

The CollisionDetectionUtils class stores no local variables but includes various methods for detecting collisions. Previously I have written methods to test for collisions but since now I have to deal with rotated rectangles, it was much more complicated. To check if two rectangles are overlapping, I have implemented a method which applies the hyperplane separation theorem. By creating an axis along each edge of both rotated rectangles and projected each point of both rectangles onto all axes, checking for collision becomes trivial. If there is no overlap between the points from the first rectangle with the points from the second rectangle when projected onto even just one of the axes, there is no collision. This is because we could then draw a line along this axis and separate the rectangles entirely which would not be possible if they were overlapping. This method spans just over 200 lines and so I will not be including a screenshot.

Since circles have infinitely many points around its circumference, perhaps this is not the best way to check for collision detection between a rotated rectangle and a circle. Instead, my method that handles collision between rotated rectangles (cars) and circles (humans) begins by rotating the rectangle back to 0 radians and also rotates the circle by the same angle about the centre of the rectangle. Now we can calculate the corner of the unrotated rectangle that is closest to the circle in its new position. If the distance between this corner and the centre of the circle is less than the circle radius, then the rotated rectangle and the circle are colliding.

3.11 GameState

The game state class is a finite state machine that stores two pieces of data: the current state of the game (phase) and the previous state of the game (lastPhase). These states correspond to the different screens that can be reached within the application. The states, 0 to 10, are the home page, the guide, the car select page, the item shop, the pre-wave page, the wave (the game), the post wave page, the paused screen, the game over page, the item guide, and the stats page.



3.12 AirStrike

The air strike class initialises a number of bombs or random size in a cluster around a clicked position. When a boolean flag is flipped to drop the bombs, the explosions are handled using bomb timers similar to how the pulse is drawn around the player car. There is no cooldown for air strikes as they are only available once per round.

3.13 Decoration

The decoration class stores images of each possible type of tree, flower, and grass that can be displayed on the map. The size of the decorations are controlled through global variables in the SmashMania file. The decoration object is instantiated with a 'type' which determines whether it is a tree, flower, or grass (each of which has a different probability which is defined in the map's generate method). The decoration class will randomly select an image from the appropriate array of sprites to be drawn onto the map with a randomly assigned offset.

4 Conclusion

To conclude, I am very proud of the game I have produced and have now, along with some family and friends, enjoyed a few hours of playtime. I have taken all the lessons that I have learnt from the first two practicals and applied them to Smash Mania, for example efficient collision detection. I also put more time into planning my design and architecture for this application which ended up being highly valuable in both cleaning up my codebase and in having a stronger overall understanding of where each piece of functionality should live.

I think that the marching squares algorithm produced nice, well-playable maps for each wave, however, it was not the result I had envisioned at the start. After a couple days spent researching how I would procedurally generate legal maps of roads, it was the best solution I came up with. If I were to do this practical again, or perhaps had more time, I would have experimented with additional custom rules as to how the binary map points were assigned. Maybe some rules about how many road points can be connected to one another would have produced more natural looking results.

Lastly, I think the items in the item shop are only limited by my creativity. There are many different abilities and items that could have been added to increase the number of features in the game. As an example, a small gun could be purchased and mounted to the front of the player car allowing the player to fire pellets at wandering humans.