Potion Simulator

Disclaimer: Some of the images used in this game have been obtained through the internet but have been modified by me using graphical software GIMP by: Cutting out a shape for a sprite out of a larger image, Removing a part of an image to use as a sprite, Removing the background of an image, Scaling the image, Combining images together to form a new image/sprite. All of the final images/sprites used in my game have been created as a result of my own actions from pre-made images. Not every image/sprite used in this game is 100% made from scratch by me. No image/sprite used in this game is 100% not made by me.

GAME LORE + MECHANICS (NO CODE)

Background:

The original plan for this game was to design an interactive potion brewing game where the player would 'catch' ingredients flying by to brew potions. The main menu was designed with this idea in mind, which is why there are recipes on the main menu screen that aren't in the game. The idea pivoted to the final product due to the original game idea not having the ability to fulfil most of the requirements and did not seem appropriate for this project.

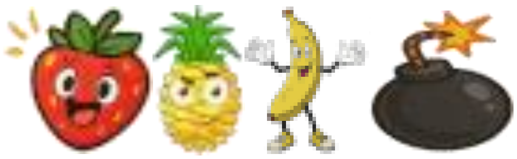
Final game overview:

The final game ended up being a 'Cauldron defender' as opposed to Potion Simulator but most of the original UI was designed for Potion Simulator.

Essentially, once you start playing the game you can control a 'Shield' which during development transformed into a wooden ladle to fit in with the game topic of potions. You can control this shield using 'A' and 'D' keys and move the ladle across the screen (no Y-axis movement). The shield is intentionally designed to be slow to move to increase the difficulty of the game as it isn't the most challenging game.

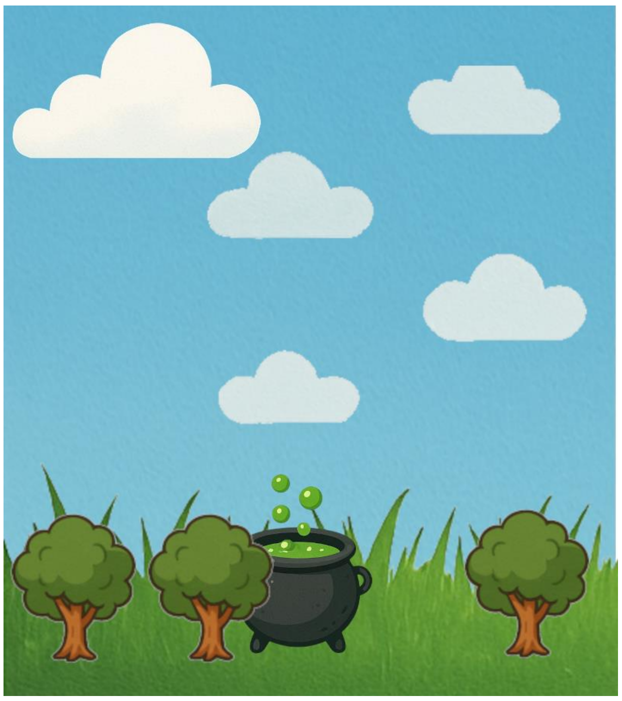
You use the shield to protect your cauldron which is brewing up your very delicate potions that cannot be disturbed by any additional ingredients getting into it (that's the game lore at least) so you must intercept any ingredients (or 'flyers' as per game files) headed towards the cauldron with your ladle. If a flyer reaches the cauldron it bubbles up and overflows, ruining your potion and you lose the game.

There are 4 types of flyers sent by your competition (other wizards brewing potions) to sabotage your operations. They are: Strawberries, Pineapples, Bananas and Bombs. Whilst they all look happy and cheerful, it is just a façade (maybe except the bomb).



The game is set in a peaceful valley where the weather is not a thing of worry.

(Mockup image using game sprites, not 100% representative of main game)



Main game mechanics:

To win the game you must last 2 minutes as that is the time the potion needs to fully brew.

If any flyer enters the cauldron the potion becomes unstable, and you lose.

Every 10 seconds the difficulty increases until difficulty level 10 after 1 minute 40 seconds. This increases flyer spawn rate and timer meaning more flyers will need defending off as you play the game. With each increase in difficulty there is a 20% chance an additional bomb will spawn.

Main game metrics are Time survived, Score and High Score.

Time survived = How long since you started playing.

Score = Receive score from defending flyers.

High Score = Highest noted score within the highscores.txt file.

Strawberries give 20 score points when intercepted.

Bananas give 10 score points when intercepted.

Pineapples give 30 score points when intercepted.

Bombs give -20 score points when intercepted.

You want to avoid intercepting bombs, when possible, but sometimes they will fall into the cauldron or will be near the ingredient you need to intercept. This adds difficulty as you need to extrapolate whether the bomb needs to be intercepted or not.

Within the game, the ingredients will go towards the cauldron (homing behavior), but it is not guaranteed that they will end up hitting it. This is intentional as if they all homed into the cauldron the player would just move 1cm each way and win each time. The added spread and uncertainty mean the players must move more and only target the ones that they think will end up making it to the cauldron. On the other hand, bombs only move downwards in a straight line but can still spawn on top of the cauldron and end up in it meaning an interception is needed. Generally, most flyers will end up making the cauldron, so it is intended for the player to intercept as many as possible but if they do not they reset back to the top.

The probability of a flyer being:

- Strawberry: 30%
- Banana: 30%
- Pineapple: 30%
- Bomb: 10%

(results may vary, as accurate as srand() function in cpp)

That would wrap up the non-technical overview of the game.

REQUIREMENTS + ACTUAL DOCUMENTATION

General marking criteria

• Your program crashes on exit or has a memory leak. (Lose 10% of your mark.)

The program does not crash.

The program has a memory leak, I tried to use smart pointers as much as possible during the game's development but some of the animation images aren't freeing the memory at the end of execution as they should. For the future this would definitely need addressing by storing the images within a smart pointer to the image OR by creating a proper destructor or a destructor function to properly free the memory from the images.

• Your program crashes at least once during its operation. (Lose 20% of your mark.)

The program does not crash at all during its operation.

• Your program crashes multiple times. (Lose 30% of your mark.)

The program does not crash at all during its operation.

• Your program crashes frequently. (Lose 40% of your mark.)

The program does not crash at all during its operation.

• Your program has some odd/unexpected behaviour/errors. (Lose at least 10% of your mark.)

No unexpected errors or behaviours known to me.

• Your program has a lot of unexpected behaviour/errors. (Lose at least 20% of your mark.)

No unexpected errors or behaviours known to me.

• You have changed framework files without prior agreement. (Lose at least 20% of your mark.)

Only added '#include "DrawingSurface.h"' to ImagePixelMapping.h because when I tried to override changePixelColour function the DrawingSurface parameter gave an error as it was unidentified so had to add the include to make it aware of the DrawingSurface being used within the function. Not sure if there was a way around it but it seemed to have fixed the issue and it's not technically changing any framework structure.

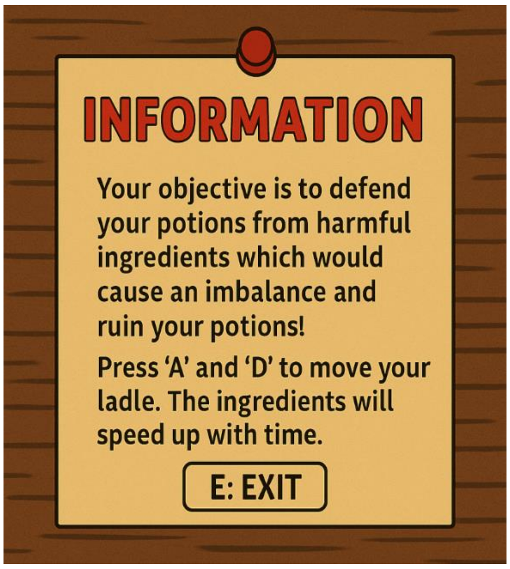
Functional Requirements

Handling of program states (total max 6 marks, 2 are advanced)

The game features a start-up state, pause state and running state called MainMenuState, PausedState and PlayingState respectively.

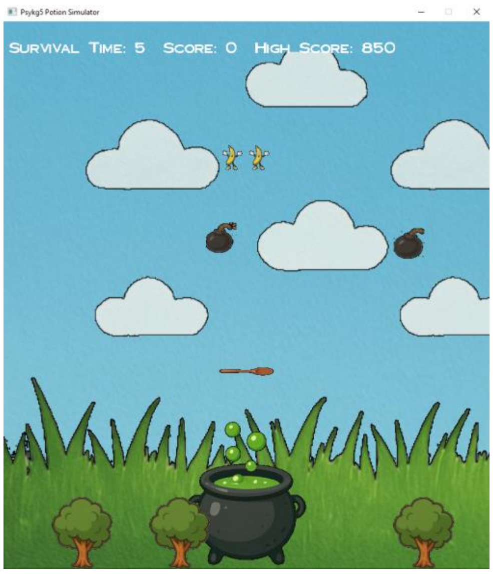
They all differ in appearance as shown below and have vastly different behaviours. In the main menu you can transition to InformationState or enter your player name. In the PlayingState you can play the game, move the foreground or scroll the foreground. In the PausedState you can unpause the game, which is different from the other states behaviour.

The program features 6 states: MainMenuState, PlayingState, PausedState, GameOverState, WinState and InformationState.

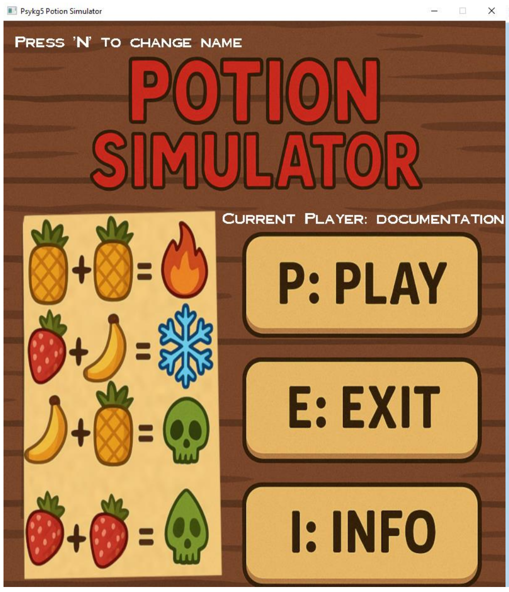

INFORMATION

Your objective is to defend your potions from harmful ingredients which would cause an imbalance and ruin your potions!

Press ‘A’ and ‘D’ to move your ladle. The ingredients will speed up with time.


Psykg5 Potion Simulator

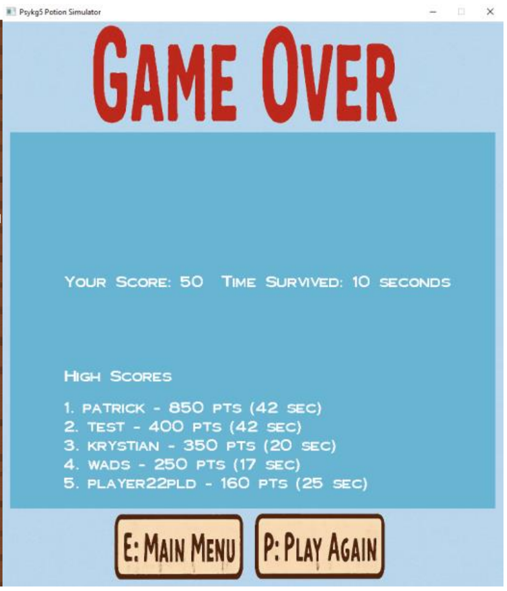
Survival Time: 5 Score: 0 High Score: 850


Psykg5 Potion Simulator

Press 'N' to change name

POTION SIMULATOR

Current Player: documentation

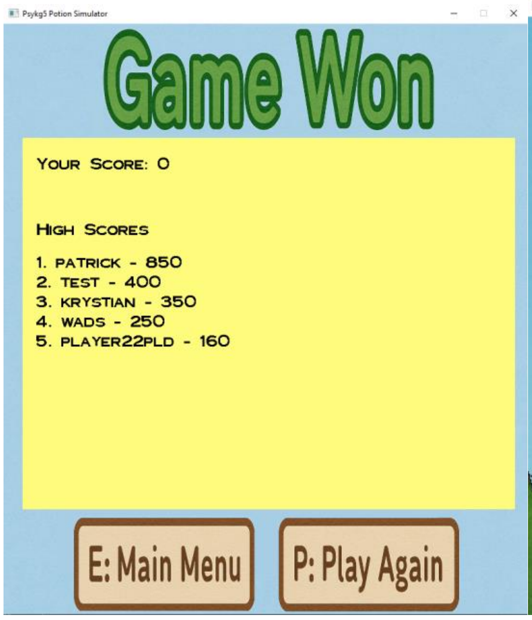

Psykg5 Potion Simulator

GAME OVER

Your Score: 50 Time Survived: 10 seconds

High Scores

1. patrick - 850 pts (42 sec)
2. test - 400 pts (42 sec)
3. krystian - 350 pts (20 sec)
4. wads - 250 pts (17 sec)
5. player22pld - 160 pts (25 sec)

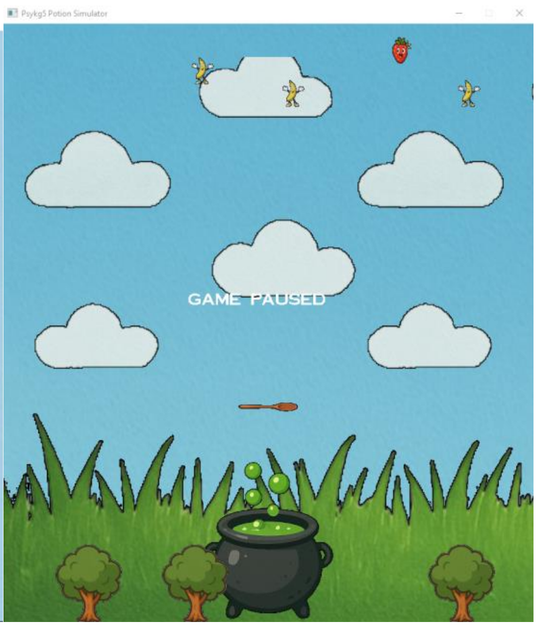

Psykg5 Potion Simulator

Game Won

Your Score: 0

High Scores

1. patrick - 850
2. test - 400
3. krystian - 350
4. wads - 250
5. player22pld - 160


Psykg5 Potion Simulator

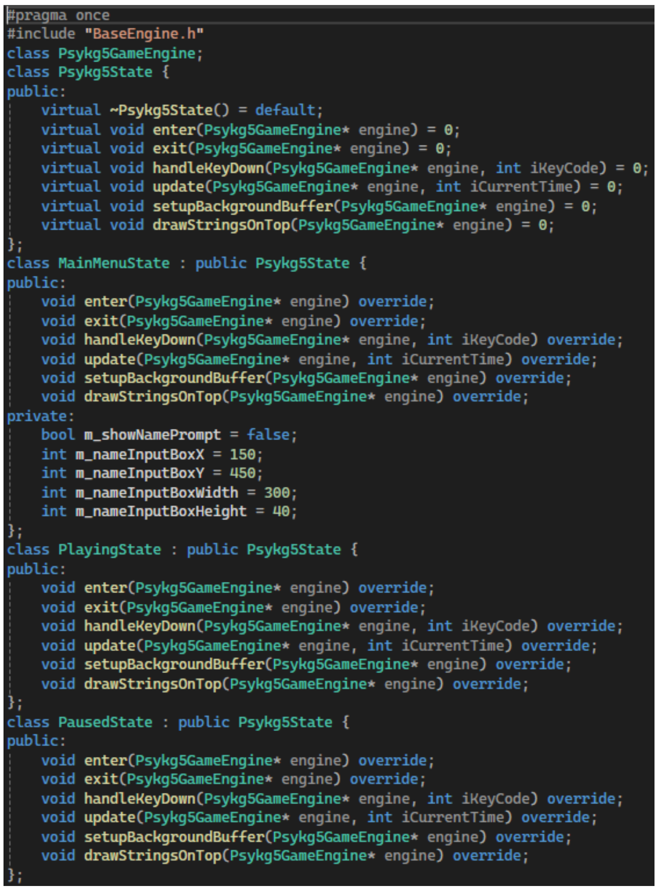
GAME PAUSED

There is a win state and lose state adequately named from the list above.

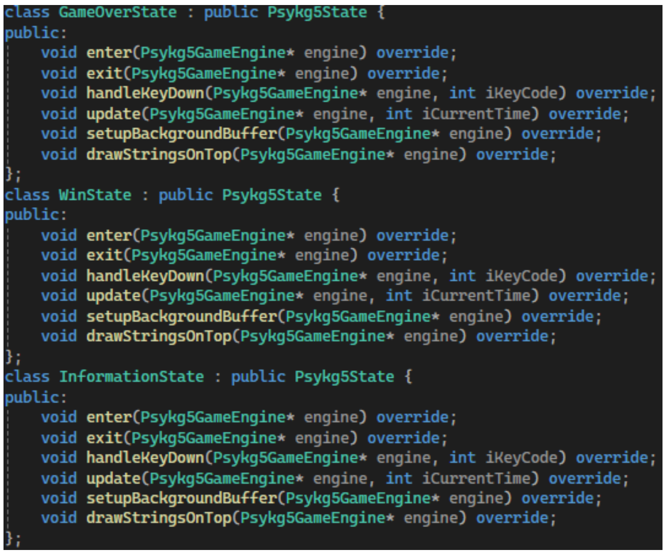
After a win/lose you are able to play the game again correctly.

The game implements the state model design pattern where there is an abstract state interface called Psykg5State which is then used to implement concrete states of MainMenuState, PlayingState, PausedState, GameOverState, WinState and InformationState which inherit and override all of the functions of Psykg5State.

Psykg5State acts as a context class which stores pointer to the current state, handles state transitions and delegates behaviour to the current state.



```
#pragma once
#include "BaseEngine.h"
class Psykg5GameEngine;
class Psykg5State {
public:
    virtual ~Psykg5State() = default;
    virtual void enter(Psykg5GameEngine* engine) = 0;
    virtual void exit(Psykg5GameEngine* engine) = 0;
    virtual void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) = 0;
    virtual void update(Psykg5GameEngine* engine, int iCurrentTime) = 0;
    virtual void setupBackgroundBuffer(Psykg5GameEngine* engine) = 0;
    virtual void drawStringsOnTop(Psykg5GameEngine* engine) = 0;
};
class MainMenuState : public Psykg5State {
public:
    void enter(Psykg5GameEngine* engine) override;
    void exit(Psykg5GameEngine* engine) override;
    void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) override;
    void update(Psykg5GameEngine* engine, int iCurrentTime) override;
    void setupBackgroundBuffer(Psykg5GameEngine* engine) override;
    void drawStringsOnTop(Psykg5GameEngine* engine) override;
private:
    bool m_showNamePrompt = false;
    int m_nameInputBoxX = 150;
    int m_nameInputBoxY = 450;
    int m_nameInputBoxWidth = 300;
    int m_nameInputBoxHeight = 40;
};
class PlayingState : public Psykg5State {
public:
    void enter(Psykg5GameEngine* engine) override;
    void exit(Psykg5GameEngine* engine) override;
    void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) override;
    void update(Psykg5GameEngine* engine, int iCurrentTime) override;
    void setupBackgroundBuffer(Psykg5GameEngine* engine) override;
    void drawStringsOnTop(Psykg5GameEngine* engine) override;
};
class PausedState : public Psykg5State {
public:
    void enter(Psykg5GameEngine* engine) override;
    void exit(Psykg5GameEngine* engine) override;
    void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) override;
    void update(Psykg5GameEngine* engine, int iCurrentTime) override;
    void setupBackgroundBuffer(Psykg5GameEngine* engine) override;
    void drawStringsOnTop(Psykg5GameEngine* engine) override;
};
```



```
class GameOverState : public Psykg5State {
public:
    void enter(Psykg5GameEngine* engine) override;
    void exit(Psykg5GameEngine* engine) override;
    void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) override;
    void update(Psykg5GameEngine* engine, int iCurrentTime) override;
    void setupBackgroundBuffer(Psykg5GameEngine* engine) override;
    void drawStringsOnTop(Psykg5GameEngine* engine) override;
};
class WinState : public Psykg5State {
public:
    void enter(Psykg5GameEngine* engine) override;
    void exit(Psykg5GameEngine* engine) override;
    void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) override;
    void update(Psykg5GameEngine* engine, int iCurrentTime) override;
    void setupBackgroundBuffer(Psykg5GameEngine* engine) override;
    void drawStringsOnTop(Psykg5GameEngine* engine) override;
};
class InformationState : public Psykg5State {
public:
    void enter(Psykg5GameEngine* engine) override;
    void exit(Psykg5GameEngine* engine) override;
    void handleKeyDown(Psykg5GameEngine* engine, int iKeyCode) override;
    void update(Psykg5GameEngine* engine, int iCurrentTime) override;
    void setupBackgroundBuffer(Psykg5GameEngine* engine) override;
    void drawStringsOnTop(Psykg5GameEngine* engine) override;
};
```

Input/output features (total max 6 marks, 2 are advanced)

The game saves and loads player names, time survived and their scores (if they are top 5 scores).

I made a custom Psykg5FileManager class which provides static methods for handling saving and loading game data. Every time the game needs to be saved or loaded this class is called and handles it and acts as a clean interface promoting the idea of encapsulation. This also ensures for more complex data handling it can implement more advanced features that can be interfaced easily by the game states for the future.

I created a custom struct to handle high score management called HighScore which is defined in the header file for Psykg5GameEngine. It stores player performance data such as the player's name (string), player score (int), survival time (int) and it also features custom operator overloading for sorting the scores for purposes such as displaying the high scores on game over screen or game won screen.

The scores are stored within a .txt file and are human readable in the format that is seen in the provided screenshot where the first value is the player's name, the second value being the score and the third value being the time they survived. The file can be edited for debugging or artificial score adding and will work fine within my game.

My function saveHighScores converts in-memory data to a string format using string streams and the string data can then be saved to the file. The function also makes sure to create a log in the console if it fails/succeeds for debugging but it was important for me to know that the data got saved.

Before the scores get saved it ensures the scores are properly sorted in descending order so the file is always sorted.

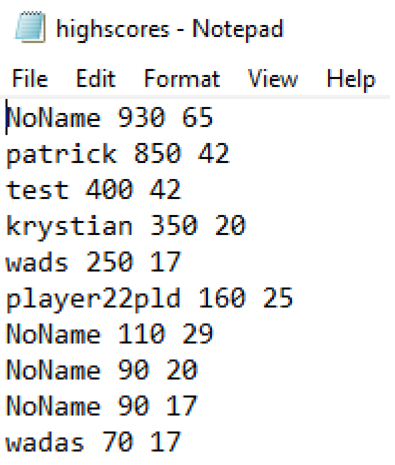
The function loadHighScores reads the file by parsing it back into memory and the data is then used to populate the HighScore objects and updating the High Score label within the playing state to display the score to beat.

The player is able to enter a custom player name within the main menu which will then be used when saving their data, if they choose not to the default NoName will be used.

The game automatically gets saved when the game enters game over screen or game won state. It also automatically loads the data when the game starts.

The game only saves 10 high scores to the file. This variable can be changed but see no significant change to the current game whether it is 10,50 or infinite. The structure is there.

The game does not save/load complex structured data as there wasn't much purpose of doing so for the game unfortunately. Psykg5FileManager can be extended to support more advanced actions for future development.



```
NoName 930 65
patrick 850 42
test 400 42
krystian 350 20
wads 250 17
player22pld 160 25
NoName 110 29
NoName 90 20
NoName 90 17
wadas 70 17
```

Displayable object features (total max 15 marks, 6 are advanced)

The game features multiple displayable objects from at least 3 displayable objects classes with different appearances and behaviours from each other. Such as cauldron, shield and flyer. There are 4 types of flyers, and they are automated. I have developed an intermediate class called Psykg5GameObject and is a bridge between my game's displayable objects and the framework's DisplayableObject. A behaviour added by my Psykg5GameObject would be animations as it has a system that can manage playing animation frames, timing or loops.

During game states such as PlayingState upon entry to that state displayable objects are being created and during the update of PlayingState new flyers are being created dynamically based on the current difficulty level of the game. Psykg5GameEngine also adds flyers using CreateFlyers function.

The objects cannot be collided with once destroyed as I have used a Boolean value of m_isDestroyed to keep track on whether the object can be interacted with or not.

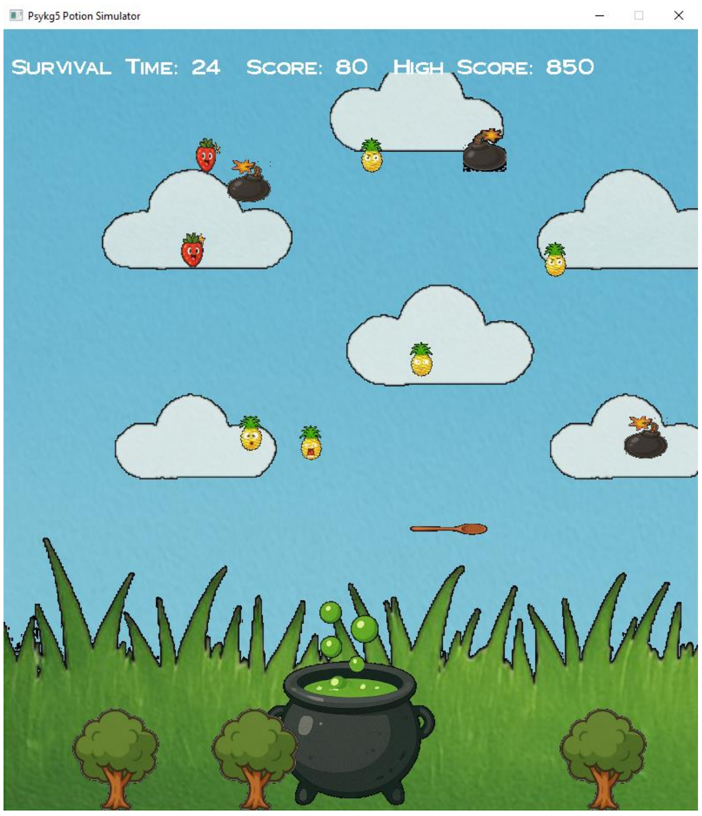
My game adds objects dynamically during the course of the game, not only at state changes, so it is not just recreating object arrays.

New objects such as Psykg5SparkleEffect are created and added to the container.

When objects are destroyed they are removed from the container such as when flyers get destroyed.

The game calls drawableObjectsChanged() all the times it is modifying object collection.

Functions such as removeDisplayableObject within DisplayableObjectConatiner handle the object deletion.



Complex intelligence on an automated moving object (max 9 marks)

The flyers do not move randomly. They move based on what type of flyer they are (ingredient, bomb). There are currently 4 FlightPatterns implemented for the ingredients to move in.

There is a helper function calculateArrivalPath which pre-calculates the path to ensure that parameters are being altered so the target is reached per flight pattern.

It calculates the vertical distance and time to reach target (cauldron):

verticalDistance = targetY - currentY

timeToTarget = verticalDistance / baseSpeed

It then adjusts parameters for the appropriate flight patterns. For zigzags it limits the width of the zigzag based on the horizontal distance to the cauldron and for circles it constraints the radius to ensure cauldron collision:

zigzagWidth = min(zigzagWidth, abs(targetX - currentX) * 2)

circleRadius = min(circleRadius, abs(targetX - currentX) / 2)

Patterns like moveZigzag or moveCircle use advanced algorithms to calculate deltas (distance to target) pattern adjustments (such as zigzag direction) and other factors.

Each ingredient has weighted probability of each flight pattern whereas bombs always use straight pattern for better game design.

Straight Pattern:

It is not linear movement, my game calculates the trajectory so that the flyer reaches the cauldron then adjusts movement on the X proportional to Y to reach the cauldron with ideal trajectory.

First it calculates the displacement vector from the flyer to the cauldron.

Then it calculates distance using Pythagoras theorem: distance = sqrt($deltaX^2$ + $deltaY^2$).

The direction vector is then normalized (unit length) by dividing by the total distance:

directionX = deltaX / distance

directionY = deltaY / distance

Then velocity is applied along this vector with a scaling factor:

moveX = directionX * baseSpeed * 0.5

moveY = baseSpeed

This ensures gradual movement towards the cauldron whilst maintaining a consistent speeds and adjusting the horizontal component to ensure the cauldron is reached.

Zigzag Pattern:

It uses some trigonometry to create oscillating horizontal movement whilst still going towards the cauldron.

The horizontal movement follows a modified square wave function implemented using:

currentX += zigzagDirection * 4.0 where zigzagDirection changes between +1 and -1

The vertical movement uses reduced speed to clearly highlight the oscillations before reaching the target too quickly:

currentY += baseSpeed * 0.6

Direction changes for the oscillations are calculated based on how far the flyer is from the cauldron:

timeToImpact = (targetY - currentY) / (baseSpeed * 0.6)

targetDeltaX = targetX – current

If the flyer is too far out the direction is adjusted:

```
if (abs(targetDeltaX) > timeToImpact * 4.0)
   zigzagDirection = (targetDeltaX > 0) ? 1 : -1
```

If the oscillation reaches maximum width of the zigzag it is also reversed:

```
if (zigzagDirection > 0 && currentX > targetX + zigzagWidth)
   zigzagDirection = -1
else if (zigzagDirection < 0 && currentX < targetX - zigzagWidth)
   zigzagDirection = 1
```

Circle Pattern:

Using trigonometry, it creates spiral flight paths that ensure the target is reached.

The angle is continuously updated using:

```
angle = (angle + 5) % 360
radians = angle * PI / 180.0 where PI = 3.14159…. (screenshot)
```

Ensuring the flyer is always moving downwards:

```
currentY += baseSpeed
```

And that the horizontal movement is following a sine function:

```
currentX += sin(radians) * 3.0 (creates circular motion)
```

It continuously adjusts the center point to ensure the cauldron is reached:

```
timeToImpact = (targetY - currentY) / baseSpeed
centerX = targetX + sin(radians) * circleRadius
deltaX = centerX - currentX
if (abs(deltaX) > 2)
   currentX += (deltaX > 0) ? 2 : -2
```

Homing Pattern:

Using mathematics and vectors it implements direct tracking of the target (stationary cauldron in this instance but it can track a moving object as well for future use).

Firstly it calculates the displacement vector to target:

```
deltaX = targetX - currentX
deltaY = targetY – currentY
```

Then it calculates the distance to the target using Pythagorean Theorem:

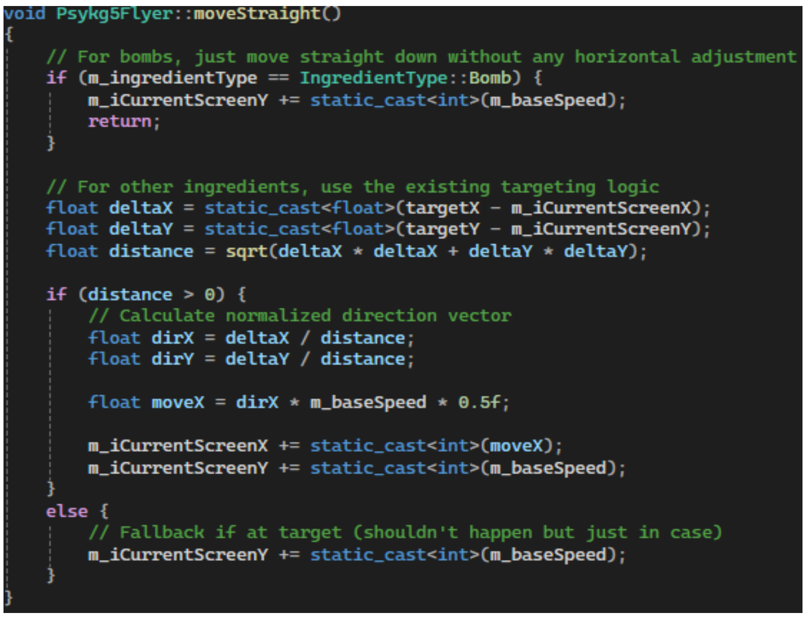
distance = sqrt(deltaX^2 + deltaY^2)

It then normalizes the direction and applies velocity towards the target:

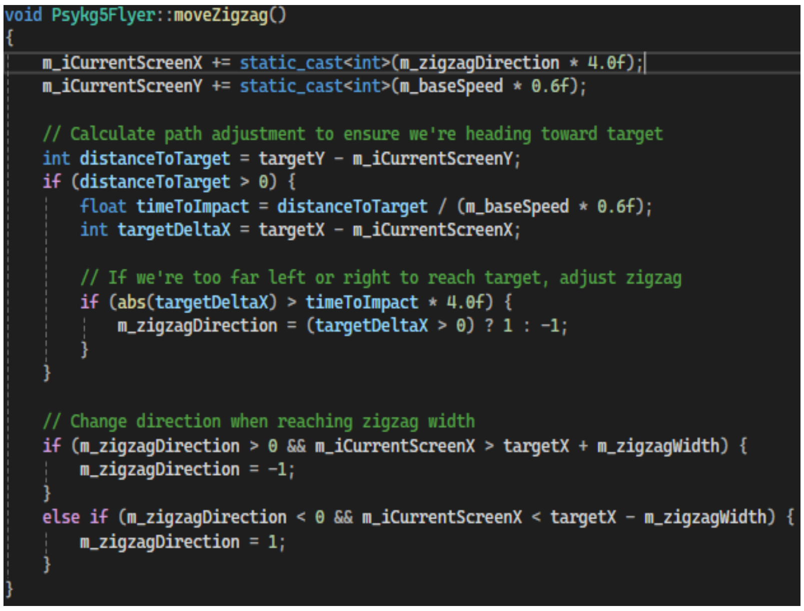
```
if (distance > 0) {
    moveX = (deltaX / distance) * 3.0
    moveY = (deltaY / distance) * 3.0

    // Ensure minimum downward velocity
    if (moveY < baseSpeed)
        moveY = baseSpeed
}
```

For complex calculations on automated moving objects, you could include the complex collision detection system, or the map applied to bombs upon impact.

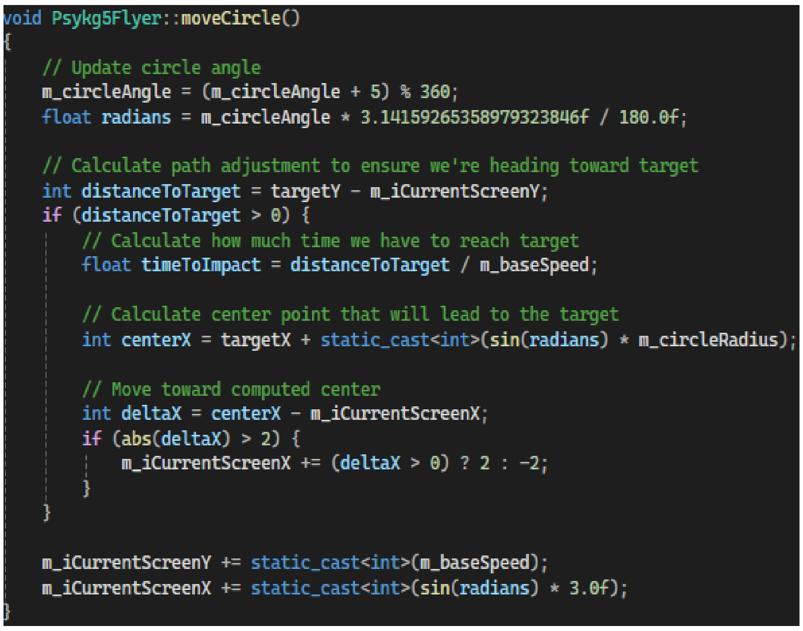
It is certainly not “Does this show you to be one of the best C++ programmers in the year?” worthy but I am quite proud of what I achieved.



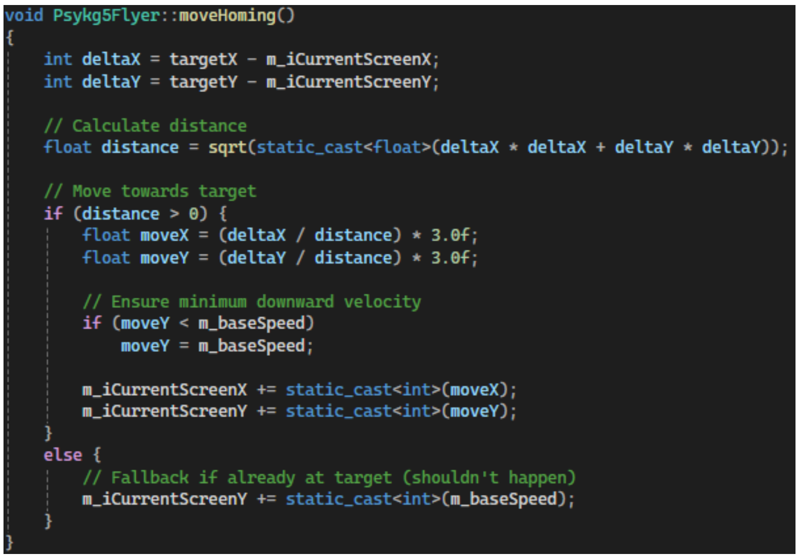
```cpp
void Psykg5Flyer::moveStraight()
{
    // For bombs, just move straight down without any horizontal adjustment
    if (m_ingredientType == IngredientType::Bomb) {
        m_iCurrentScreenY += static_cast<int>(m_baseSpeed);
        return;
    }

    // For other ingredients, use the existing targeting logic
    float deltaX = static_cast<float>(targetX - m_iCurrentScreenX);
    float deltaY = static_cast<float>(targetY - m_iCurrentScreenY);
    float distance = sqrt(deltaX * deltaX + deltaY * deltaY);

    if (distance > 0) {
        // Calculate normalized direction vector
        float dirX = deltaX / distance;
        float dirY = deltaY / distance;

        float moveX = dirX * m_baseSpeed * 0.5f;

        m_iCurrentScreenX += static_cast<int>(moveX);
        m_iCurrentScreenY += static_cast<int>(m_baseSpeed);
    }
    else {
        // Fallback if at target (shouldn't happen but just in case)
        m_iCurrentScreenY += static_cast<int>(m_baseSpeed);
    }
}
```



```cpp
void Psykg5Flyer::moveZigzag()
{
    m_iCurrentScreenX += static_cast<int>(m_zigzagDirection * 4.0f);
    m_iCurrentScreenY += static_cast<int>(m_baseSpeed * 0.6f);

    // Calculate path adjustment to ensure we're heading toward target
    int distanceToTarget = targetY - m_iCurrentScreenY;
    if (distanceToTarget > 0) {
        float timeToImpact = distanceToTarget / (m_baseSpeed * 0.6f);
        int targetDeltaX = targetX - m_iCurrentScreenX;

        // If we're too far left or right to reach target, adjust zigzag
        if (abs(targetDeltaX) > timeToImpact * 4.0f) {
            m_zigzagDirection = (targetDeltaX > 0) ? 1 : -1;
        }
    }

    // Change direction when reaching zigzag width
    if (m_zigzagDirection > 0 && m_iCurrentScreenX > targetX + m_zigzagWidth) {
        m_zigzagDirection = -1;
    }
    else if (m_zigzagDirection < 0 && m_iCurrentScreenX < targetX - m_zigzagWidth) {
        m_zigzagDirection = 1;
    }
}
```



```cpp
void Psykg5Flyer::moveCircle()
{
    // Update circle angle
    m_circleAngle = (m_circleAngle + 5) % 360;
    float radians = m_circleAngle * 3.14159265358979323846f / 180.0f;

    // Calculate path adjustment to ensure we're heading toward target
    int distanceToTarget = targetY - m_iCurrentScreenY;
    if (distanceToTarget > 0) {
        // Calculate how much time we have to reach target
        float timeToImpact = distanceToTarget / m_baseSpeed;

        // Calculate center point that will lead to the target
        int centerX = targetX + static_cast<int>(sin(radians) * m_circleRadius);

        // Move toward computed center
        int deltaX = centerX - m_iCurrentScreenX;
        if (abs(deltaX) > 2) {
            m_iCurrentScreenX += (deltaX > 0) ? 2 : -2;
        }
    }

    m_iCurrentScreenY += static_cast<int>(m_baseSpeed);
    m_iCurrentScreenX += static_cast<int>(sin(radians) * 3.0f);
}
```



```cpp
void Psykg5Flyer::moveHoming()
{
    int deltaX = targetX - m_iCurrentScreenX;
    int deltaY = targetY - m_iCurrentScreenY;

    // Calculate distance
    float distance = sqrt(static_cast<float>(deltaX * deltaX + deltaY * deltaY));

    // Move towards target
    if (distance > 0) {
        float moveX = (deltaX / distance) * 3.0f;
        float moveY = (deltaY / distance) * 3.0f;

        // Ensure minimum downward velocity
        if (moveY < m_baseSpeed)
            moveY = m_baseSpeed;

        m_iCurrentScreenX += static_cast<int>(moveX);
        m_iCurrentScreenY += static_cast<int>(moveY);
    }
    else {
        // Fallback if already at target (shouldn't happen)
        m_iCurrentScreenY += static_cast<int>(m_baseSpeed);
    }
}
```

Collision detection (4 marks, 2 is advanced)

I implemented Psykg5AdvancedCollision which is a header file that removes the need for UtilCollisionDetection as it is better. It performs the basic checks of UtilCollisionDetection for improved efficiency alongside more advanced methods. The cauldron collision uses a polygon for closer collision detection as opposed to a square or oval. Psykg5AdvancedCollision also features pixel-perfect collision detection between the shield and flyers.

When detecting a collision my implementation firstly does a quick rectangular broad check and returns if there is no apparent collision. This eliminates the need for extensive calculations each frame and optimises performance greatly.

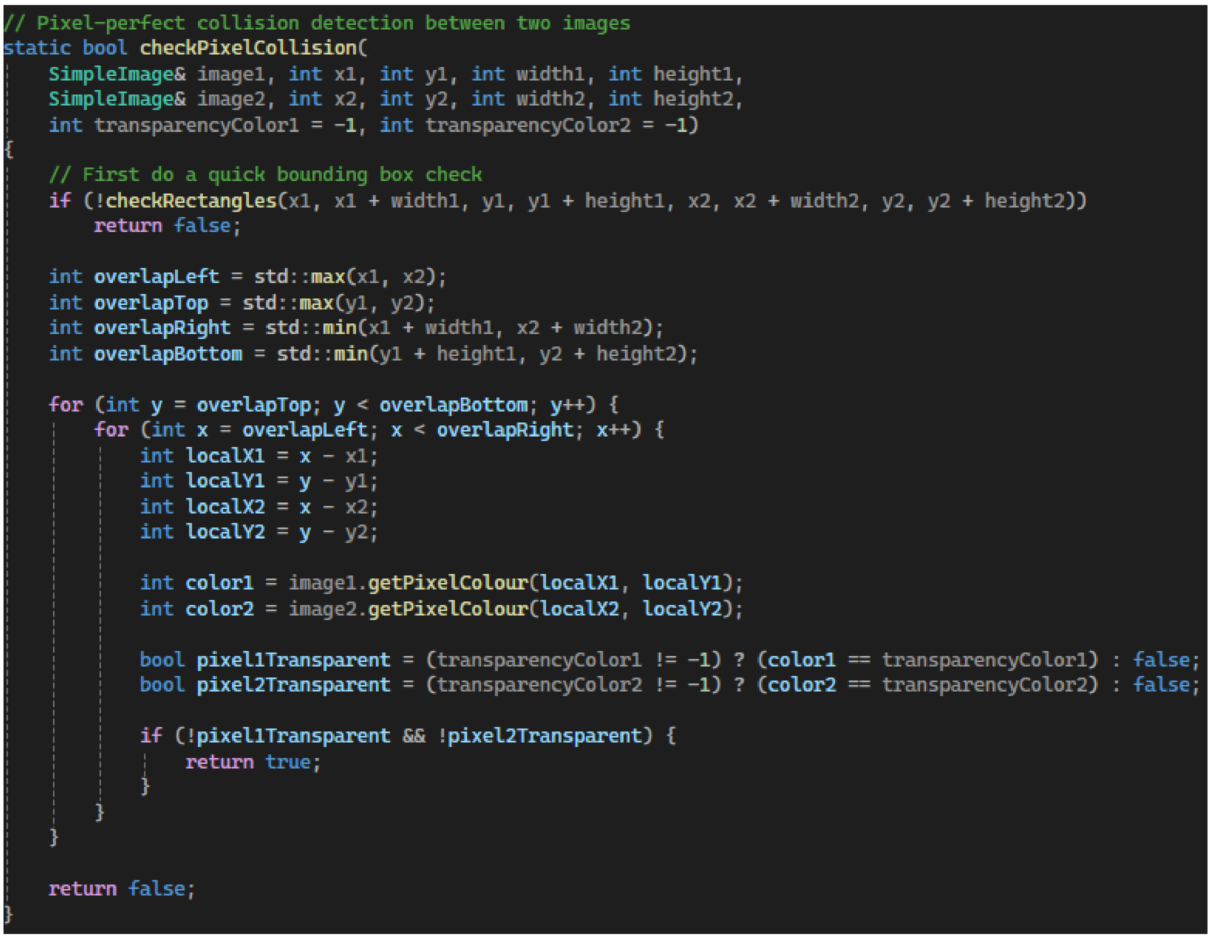
If there is a simple rectangular collision detected, it performs more tailored checks such as convex polygon intersection checks or triangle to triangle intersection checks. These are still not intense on the computation power so if there is no collision we can return again.

If both of these checks detect collisions we move onto pixel-perfect collision which looks at collisions at pixel level, but it goes over every pixel of the overlapping section. This affects the performance more than the other methods which is why the other methods are always checked before we reach this method. This is the final check and often indicates that a collision did in fact occur. The way it determines if a collision has occurred is if there are overlapping pixels which are non-transparent at the same pixel position.

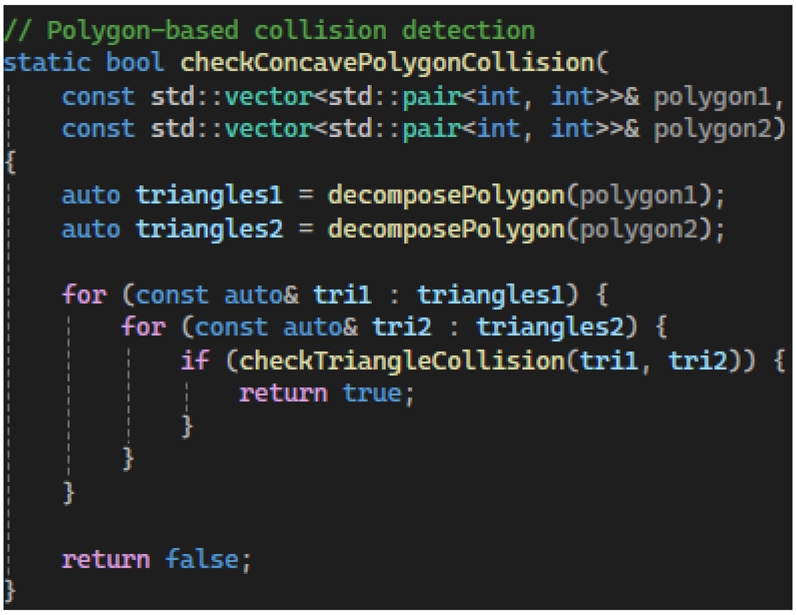
To perfect this system, I added an algorithm called ‘ear clipping triangulation’ which turns complex concave polygons to triangles as triangles are easier to work with. It identifies which vertices form valid triangles (ear vertices) and the polygon decreases each time an ear (triangle) is clipped from it. It aims to deconstruct the whole polygon into these ‘ears’. To achieve this, I created a function which will be used to determine whether a vertex forms a valid ‘ear’ called isEar which adds extra error checking to confirm that the triangle is safe to use for example making sure no other polygon vertices exist within the ‘ear’. Once we have decomposed the polygon(s) we can perform simple triangle collision to detect collisions.

To improve performance, I also cache the triangulated versions of polygons to prevent recalculating of all the ‘ears’ each time. All of the functions also have early returns/exits to ensure minimum calculation needed at each stage.

This system is very accurate and due to its performance optimizations can be further used when the game grows bigger with more complex objects, more collisions and more sophisticated polygons.



```
// Pixel-perfect collision detection between two images
static bool checkPixelCollision(
    SimpleImage& image1, int x1, int y1, int width1, int height1,
    SimpleImage& image2, int x2, int y2, int width2, int height2,
    int transparencyColor1 = -1, int transparencyColor2 = -1)
{
    // First do a quick bounding box check
    if (!checkRectangles(x1, x1 + width1, y1, y1 + height1, x2, x2 + width2, y2, y2 + height2))
        return false;

    int overlapLeft = std::max(x1, x2);
    int overlapTop = std::max(y1, y2);
    int overlapRight = std::min(x1 + width1, x2 + width2);
    int overlapBottom = std::min(y1 + height1, y2 + height2);

    for (int y = overlapTop; y < overlapBottom; y++) {
        for (int x = overlapLeft; x < overlapRight; x++) {
            int localX1 = x - x1;
            int localY1 = y - y1;
            int localX2 = x - x2;
            int localY2 = y - y2;

            int color1 = image1.getPixelColour(localX1, localY1);
            int color2 = image2.getPixelColour(localX2, localY2);

            bool pixel1Transparent = (transparencyColor1 != -1) ? (color1 == transparencyColor1) : false;
            bool pixel2Transparent = (transparencyColor2 != -1) ? (color2 == transparencyColor2) : false;

            if (!pixel1Transparent && !pixel2Transparent) {
                return true;
            }
        }
    }

    return false;
}
```



```
// Polygon-based collision detection
static bool checkConcavePolygonCollision(
    const std::vector<std::pair<int, int>>& polygon1,
    const std::vector<std::pair<int, int>>& polygon2)
{
    auto triangles1 = decomposePolygon(polygon1);
    auto triangles2 = decomposePolygon(polygon2);

    for (const auto& tri1 : triangles1) {
        for (const auto& tri2 : triangles2) {
            if (checkTriangleCollision(tri1, tri2)) {
                return true;
            }
        }
    }

    return false;
}
```

Animations, foreground scrolling and zooming (Max total 15 marks, 3 are advanced)

I developed Psykg5ScrollingBackground which builds up the main game background as layers so that there is a strip of grass at the bottom (stationary) with a blue sky (stationary) completing the stationary background and on top is a third layer which features small clouds that move automatically from one side of the screen to the other (can go off screen a little bit as well as it handles the layers being larger than the window) and will travel in the opposite direction to stay on screen at all times. Psykg5GameEngine overrides copyAllBackgroundBuffer to make sure that the background is correctly animated. Within virtMainLoopDoBeforeUpdate the background is constantly being redrawn, so it looks smoother. The background (only clouds as they are the only non-stationary layer) can also be scrolled using the arrow keys for additional player input. Each layer is also rendered with the appropriate function as the sky uses renderImage (no transparency) whereas the clouds use renderImageWithMask (transparency).

With the current implementation the background layers can move at differing speeds and can be larger than the window size, whilst maintaining smoothness. The individual layers are independent from one another and could be used to create a parallax scrolling system in the future with multiple layers moving at different speeds independently from one another.

The system also allows for certain layers to be manipulated for example in my current game using the arrow keys the clouds can be moved but not the sky or the grass as they are stationary elements. This also allows user input to change the appearance of the background.

I have added 3 sprites of trees to the foreground which can be scrolled using 'J' 'L' 'K' 'I' keys to move left, right, down and up respectively. These foreground objects can also be enlarged and shrunk using your scroll wheel or '-'(minus) and '+' (plus) buttons. This enlargement by buttons or scroll wheel ensures minimum zoom values and maximum zoom values to prevent any errors/crashes and to make the game more appealing and not ruined by it.

I developed Psykg5FilterPoints which is built on top of FilterPoints to implement both scrolling and zooming which is then applied to the foreground. All of the required methods are implemented within my class. This is applied to the tree elements in foreground allowing them to be moved and zoomed(enlarged and shrunk).

Within my game I have multiple animated moving objects such as flyers, cauldron (upon collision) or shield (upon collision). I have tried to design multiple frames per object to make the animations looks 'smoother' and 'visually impressive' and I think the end result is not too bad. The animations work time and them just being looped over and over using my custom Psykg5GameObject class. The system calculates if the current time is larger than last frame time plus the duration of the frame to move onto the next frame for added smoothness.

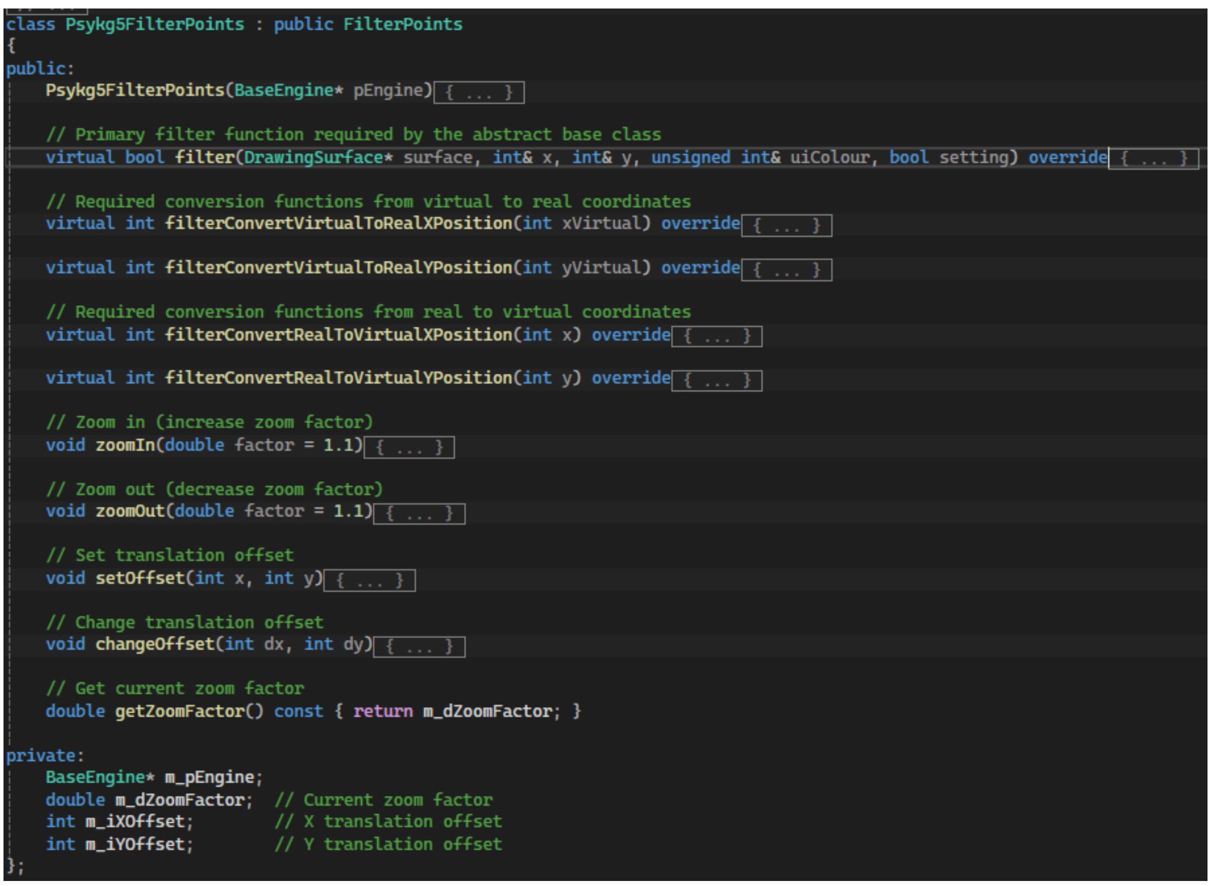
I also animated the bombs so that when they collide with the ladle they are bounced away with a custom wave effect map. I created a Psykg5WavePixelMapping class built on top of ImagePixelMapping class to handle this custom effect. This effect gets added to all the bombs but is only activated once they collide with the ladle.

The way the effect works is that it uses trigonometric functions (sine and cosine) to create wave distortions which then is applied to the image whilst it is animating and once it has done animating it gets removed with a sparkle effect similarly to the flyers.

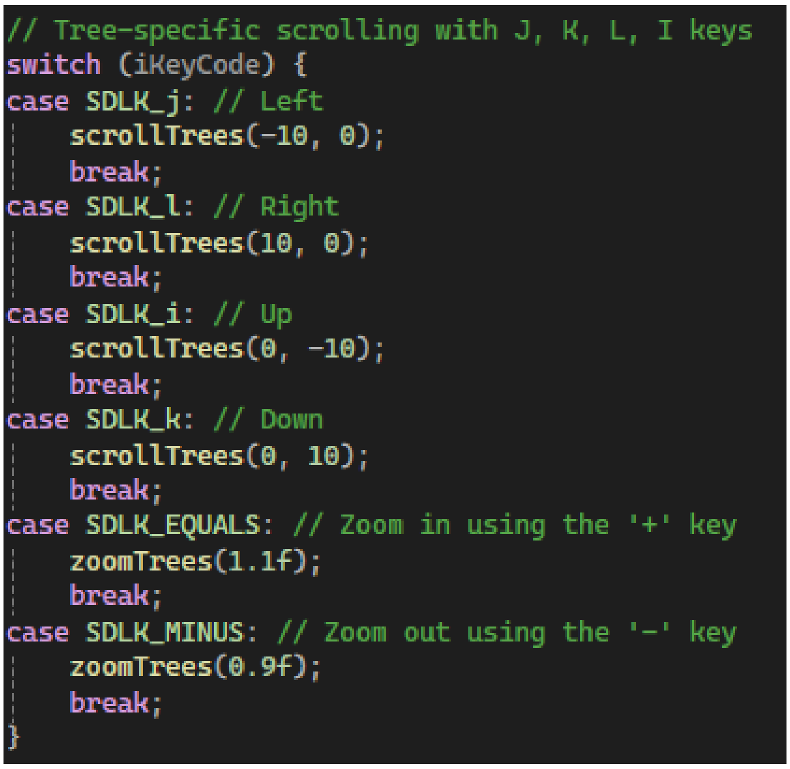
I made sure Psykg5WavePixelMapping handles boundary conditions for image edges accurately to avoid any crashes or unexpected behaviours.

Throughout my game collision triggers appropriate events such as the sparkle effect animation appearing at the point of collision with an ingredient flyer or the bomb wave mapping. The sparkle is then triggered at the final position of the bomb once it has been removed.

This decreases the performance of the game however I tried to optimize how these animations are handled to minimize this downside.



```cpp
class Psykg5FilterPoints : public FilterPoints
{
public:
    Psykg5FilterPoints(BaseEngine* pEngine){ ... }

    // Primary filter function required by the abstract base class
    virtual bool filter(DrawingSurface* surface, int& x, int& y, unsigned int& uiColour, bool setting) override{ ... }

    // Required conversion functions from virtual to real coordinates
    virtual int filterConvertVirtualToRealXPosition(int xVirtual) override{ ... }

    virtual int filterConvertVirtualToRealYPosition(int yVirtual) override{ ... }

    // Required conversion functions from real to virtual coordinates
    virtual int filterConvertRealToVirtualXPosition(int x) override{ ... }

    virtual int filterConvertRealToVirtualYPosition(int y) override{ ... }

    // Zoom in (increase zoom factor)
    void zoomIn(double factor = 1.1){ ... }

    // Zoom out (decrease zoom factor)
    void zoomOut(double factor = 1.1){ ... }

    // Set translation offset
    void setOffset(int x, int y){ ... }

    // Change translation offset
    void changeOffset(int dx, int dy){ ... }

    // Get current zoom factor
    double getZoomFactor() const { return m_dZoomFactor; }

private:
    BaseEngine* m_pEngine;
    double m_dZoomFactor;  // Current zoom factor
    int m_iXOffset;        // X translation offset
    int m_iYOffset;        // Y translation offset
};
```



```cpp
// Tree-specific scrolling with J, K, L, I keys
switch (iKeyCode) {
case SDLK_j: // Left
    scrollTrees(-10, 0);
    break;
case SDLK_l: // Right
    scrollTrees(10, 0);
    break;
case SDLK_i: // Up
    scrollTrees(0, -10);
    break;
case SDLK_k: // Down
    scrollTrees(0, 10);
    break;
case SDLK_EQUALS: // Zoom in using the '+' key
    zoomTrees(1.1f);
    break;
case SDLK_MINUS: // Zoom out using the '-' key
    zoomTrees(0.9f);
    break;
}
```

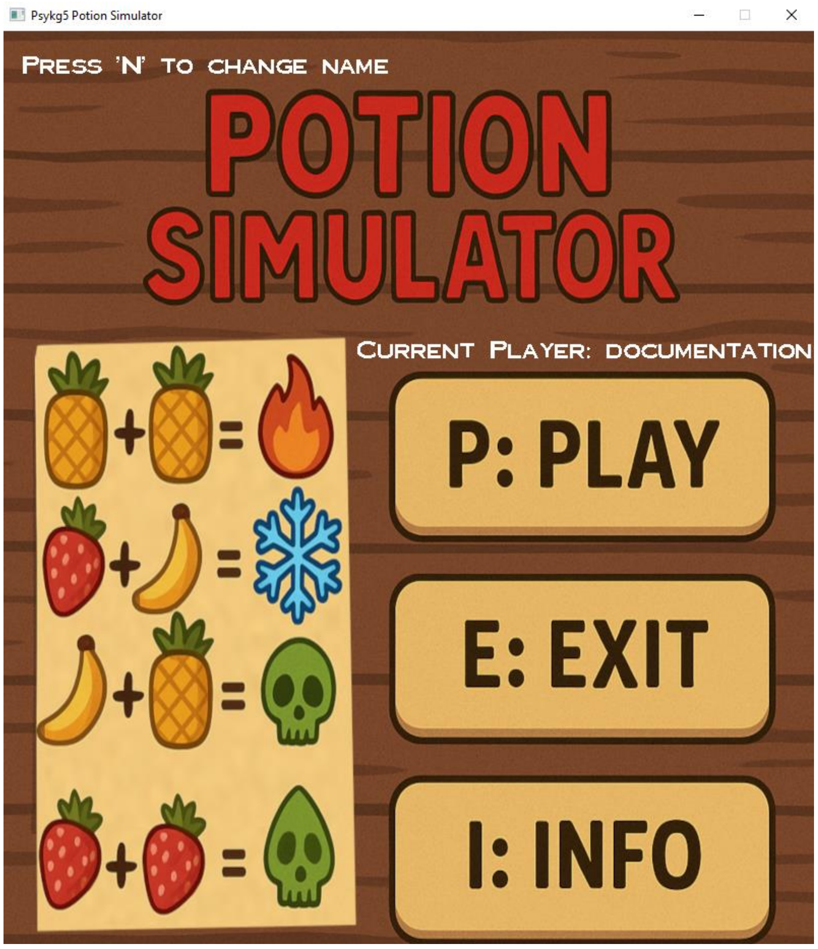
Tile manager usage (Max 4 marks, 2 are advanced)

I couldn’t find an appropriate use for a tile manager for this particular game and chose not to force it as the look of the game was satisfactory to me.

Other features (Max total 10 marks, 6 are advanced)

Allow user to enter text which appears on the graphical display (max 2 marks)

In the main menu stage of the game there is a foreground label which displays the current player (default is NoName) that can be altered by pressing ‘N’ to input a custom player name which will be displayed. You can use the backspace key to remove any characters.



Show your understanding of templates, operator overloading OR smart pointers (max 2 marks).

I developed a class Psykg5TemplateUtils to implement generic object management system.

ObjectManager<T> is a template class that provides type-safe container for managing game objects which can be reused. By the class having type T it enforces type constraints at compile-time in an attempt to prevent type errors which would occur at runtime.

The class also provides 2 different methods to add objects, one accepting raw pointers and one accepting smart pointers. This makes the class much more flexible to use and improves versatility.

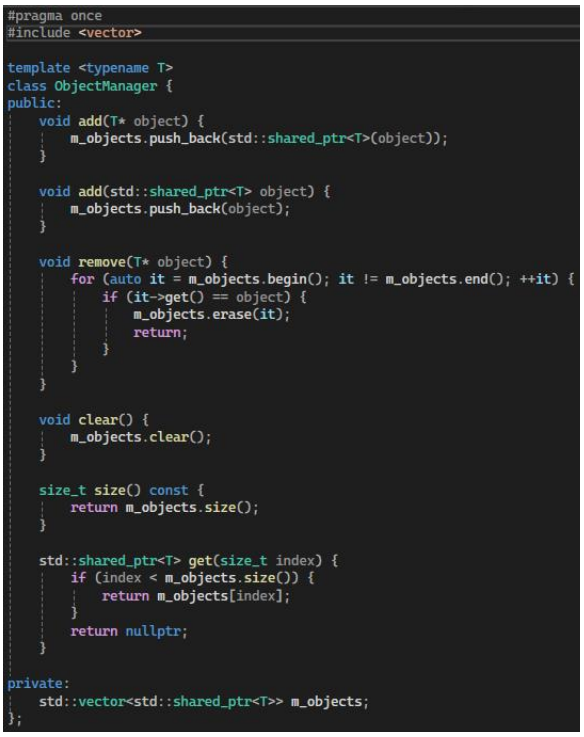
If a raw pointer is added it gets converted to a shared pointer in an attempt to ensure good resource management and prevent memory leaks from occurring.

This stand-alone class provides a good interface for the game to interact with when adding new game objects and managing them safely.

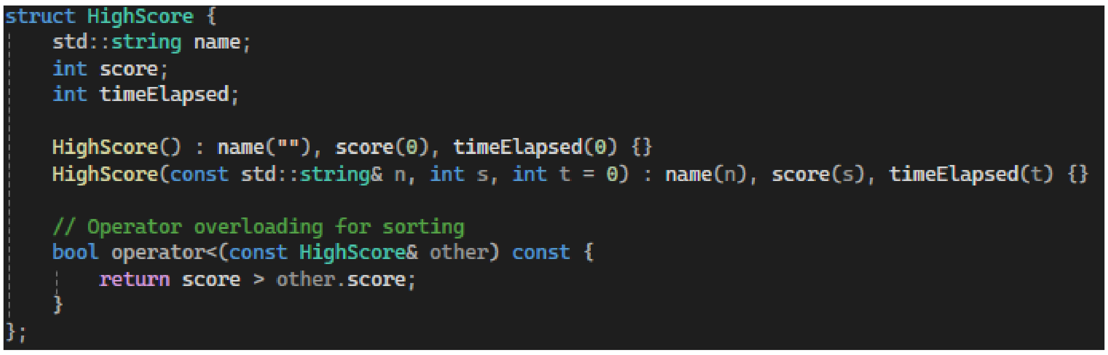
I tried to use as many smart pointers as possible to prevent memory leaks, they can also help with dealing with ownership. I used std::unique_ptr for objects which need exclusive ownership such as states and used std::shared_ptr within the ObjectManager for shared resources.

This heavy use of smart pointers helps to reduce manual memory management and removes the need for explicit destructor functions improving program stability as there are less memory leaks and resources are properly managed even during an exception.

Within my Psykg5GameEngine.h I have used operator overloading with '<' operator within my struct for HighScores that is used for handling player scores. I used '<' as the purpose this overloading serves goes well with the natural definition of the symbol making my code more readable. It also means that the struct can work with algorithms such as std::sort easier.



```cpp
#pragma once
#include <vector>

template <typename T>
class ObjectManager {
public:
    void add(T* object) {
        m_objects.push_back(std::shared_ptr<T>(object));
    }

    void add(std::shared_ptr<T> object) {
        m_objects.push_back(object);
    }

    void remove(T* object) {
        for (auto it = m_objects.begin(); it != m_objects.end(); ++it) {
            if (it->get() == object) {
                m_objects.erase(it);
                return;
            }
        }
    }

    void clear() {
        m_objects.clear();
    }

    size_t size() const {
        return m_objects.size();
    }

    std::shared_ptr<T> get(size_t index) {
        if (index < m_objects.size()) {
            return m_objects[index];
        }
        return nullptr;
    }

private:
    std::vector<std::shared_ptr<T>> m_objects;
};
```

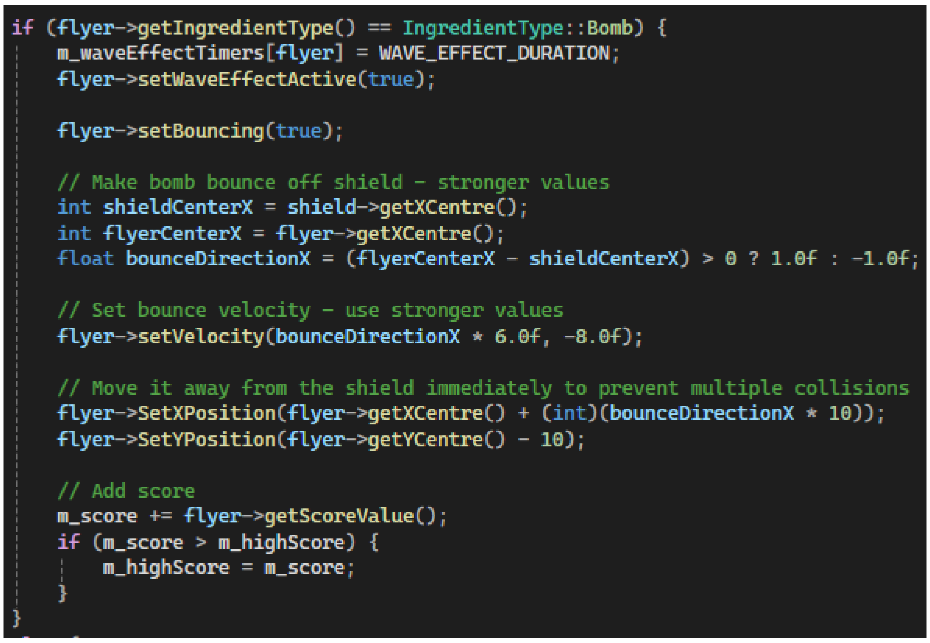


```cpp
struct HighScore {
    std::string name;
    int score;
    int timeElapsed;

    HighScore() : name(""), score(0), timeElapsed(0) {}
    HighScore(const std::string& n, int s, int t = 0) : name(n), score(s), timeElapsed(t) {}

    // Operator overloading for sorting
    bool operator<(const HighScore& other) const {
        return score > other.score;
    }
};
```

Additional complexity (max 6 marks, advanced)

I made use of my physics-based movement system that allows the bombs to ‘bounce’ away from the ladle upon impact whilst playing their custom wave function which also ensures performance stability as the shield is moved immediately after collision to prevent multiple ‘bounces’ due to the time delta between the ladle colliding and moving sufficiently away to not ‘collide’ again. I think this ‘bounce’ mechanic combined with the wavy transformation map applied to the image is quite complex.

I also think my collision system is really good; it uses an algorithm called ‘ear clipping’ for triangulation to ensure most accurate collision detection as that is something I am looking forward to using during continued development of the game after this module.



```
if (flyer->getIngredientType() == IngredientType::Bomb) {
    m_waveEffectTimers[flyer] = WAVE_EFFECT_DURATION;
    flyer->setWaveEffectActive(true);

    flyer->setBouncing(true);

    // Make bomb bounce off shield - stronger values
    int shieldCenterX = shield->getXCentre();
    int flyerCenterX = flyer->getXCentre();
    float bounceDirectionX = (flyerCenterX - shieldCenterX) > 0 ? 1.0f : -1.0f;

    // Set bounce velocity - use stronger values
    flyer->setVelocity(bounceDirectionX * 6.0f, -8.0f);

    // Move it away from the shield immediately to prevent multiple collisions
    flyer->SetXPosition(flyer->getXCentre() + (int)(bounceDirectionX * 10));
    flyer->SetYPosition(flyer->getYCentre() - 10);

    // Add score
    m_score += flyer->getScoreValue();
    if (m_score > m_highScore) {
        m_highScore = m_score;
    }
}
```

Overall impact/impression (Max 20 marks, 20 are advanced)

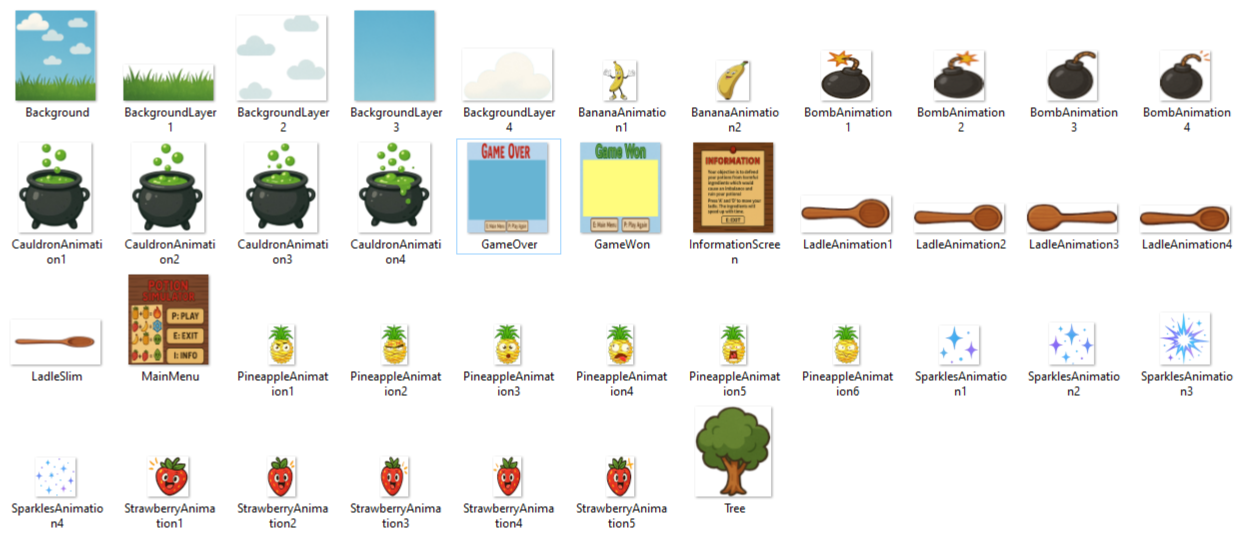
I am really proud of how the game’s appearance turned out. Background is custom made by me and includes a nice environment. Moving objects are nicely animated and fit into the cheerful environment of the game and there are more than 3 moving objects. I have more than one object that is user controlled (shield and the trees), and these accept both keyboard inputs (A D I J K L) and scroll wheel inputs. I have appropriate states with appropriate graphical design for each stage of my game and there is more than 2. My program works smoothly (not 120fps smoothly) however the infrastructure, framework and algorithm efficiency does hinder that performance slightly, but the game looks good and is totally playable with no lag spikes/crashes/inconsistencies.

Throughout the documentation there are images of the graphical aspect of my game, the demo video showcases the animations and animated aspects of it.

It’s definitely not the most knowledge-intensive game and it is pretty straightforward to play but I think the cheerfulness of it with the animations make it a game a child could enjoy.

Ideally the high scores system would make more sense if the game was endless but for the sake of having a win state to satisfy requirements it turned out this way.

Overall, I am pretty satisfied with how the game turned out even with the pivoting from original idea.


Background
BackgroundLayer1
BackgroundLayer2
BackgroundLayer3
BackgroundLayer4
BananaAnimation1
BananaAnimation2
BombAnimation1
BombAnimation2
BombAnimation3
BombAnimation4
CauldronAnimation1
CauldronAnimation2
CauldronAnimation3
CauldronAnimation4
GAME OVER
GameOver
Game Won
GameWon
INFORMATION
InformationScreen
LadleAnimation1
LadleAnimation2
LadleAnimation3
LadleAnimation4
LadleSlim
P: PLAY
E: EXIT
I: INFO
MainMenu
PineappleAnimation1
PineappleAnimation2
PineappleAnimation3
PineappleAnimation4
PineappleAnimation5
PineappleAnimation6
SparklesAnimation1
SparklesAnimation2
SparklesAnimation3
SparklesAnimation4
StrawberryAnimation1
StrawberryAnimation2
StrawberryAnimation3
StrawberryAnimation4
StrawberryAnimation5
Tree