Vampire Survivors

Jonathan Wells

November 7, 2025

1 Introduction

The primary objective of this project was to design and implement a simple, 2D, top-down, inspired Vampire survivors game using C++ and the provided GamesBaseEngineering framework. The player must survive against waves of enemies (ghosts, demons and shooter enemies) for as long as possible, with only a gun and a special ability power.

This project demonstrates fundamental programming concepts such as classes, functions, loops, conditionals, functions, randomisation, time management and inheritance as well as saving and loading a txt file.

The game can be found at the following git hub link along with this report and a short video showing the game in action.

https://github.com/jw9357/WarwickUniVampireSurvivors

2 Technologies Implemented

2.1 Virtual Camera

The world was made up of 80 tiles in width and 80 tiles in length, with each tile being 32 in width and 32 in length resulting in the world dimensions being 2560 x 2560. The player is centred in the middle of the camera at player X = 512 (half the canvas width) and player Y = 384 (half the canvas height), so the player never leaves the centre of the screen at all times. There is then a cameraX and cameraY which start at the centre of the screen, at the co-ordinates of 0 and 0 respectively and theses values are used to move the map by key presses from the player. At any time during the building of the game, if required for collisions, for example, the player's world co-ordinates was calculated by adding these two values together. For example, if a player were to move left and up, to the very top left of the world map, the player's world co-ordinates would be playerX + cameraX = 512 + (- 512) = 0 and playerY + cameraY = 384 + (- 384) = 0. Likewise if a player were to move down and right to the very bottom right of the world map, playerX + cameraX = 512 + (2048) = 2560 and playerY + cameraY = 384 + (2176) = 2560.

However, if the player were to move to the top left of the screen, at the co-ordinates of (0,0) on the world map, for example, then the black background surrounding the world map would be visible. Therefore, in order to solve this problem of not having the black background appear at any point in the game, the player's moveable area on the world map is confined to a certain area as shown in Figure 1 which shows the world co-ordinates and the player's movable area.

The end result being that the player has enough area to manoeuvre in, without the black background being shown at any time. Every movable enemy is then spawned on the edge of the world map and they start moving towards the player, and if outside the camera view, they are not rendered. When they come into the virtual camera view, an adjustment is made by using, not their world positions, but their screen positions, which is calculated by screenX = enemyX - cameraX and screenY = enemyY - cameraY , for example, in order for them to be rendered onto the screen when they appear within the virtual camera view.

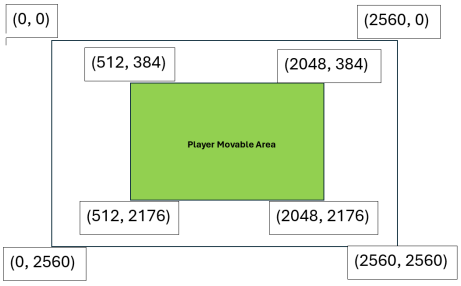

Figure 1: *World co-ordinates of map and movable player area.*

The below code shows an example how the enemy is rendered to the screen when the enemy enters the view of the camera.

```
1  void Enemy::drawEnemy(GamesEngineeringBase::Window* canvas, float cameraX, float cameraY)          //
       draws an enemyBlueGhost
2  {
3    // converts world enemy position to screen enemy position
4    screenX = enemyX - cameraX;
5    screenY = enemyY - cameraY;
6
7    if (enemyAlive)
8    {
9      GamesEngineeringBase::Image* currentDirectionImage;
10
11     if (lookingRight)
12     {
13       currentDirectionImage = &enemyRightImage;        // if looking right, use this image at this
       address
14     }
15     else
16     {
17       currentDirectionImage = &enemyLeftImage;         // if looking left, use this image at this
       address
18     }
19
20     for (int y = 0; y < currentDirectionImage->height; y++)
21     {
22       float drawY = screenY + y;                             // check screen y + pixel image y
23       if (drawY > 0 && drawY < canvas->getHeight())     // if can draw y, move on
24       {
25         for (int x = 0; x < currentDirectionImage->width; x++)
26         {
27           float drawX = screenX + x;                         // check screen x + pixel image x
28           if (drawX > 0 && drawX < canvas->getWidth())       // if can draw x, move on
29           {
30             if (currentDirectionImage->alphaAt(x, y) > 0)
31             {
32               canvas->draw(screenX + x, screenY + y, currentDirectionImage->at(x, y));  // if the code
       makes it this far, then image can be drawn
33             }
34           }
35         }
36       }
37     }
38     drawCollisionCircle(screenX + currentDirectionImage->width / 2, screenY + currentDirectionImage->
       height / 2, 30, *canvas);
39   }
40 }
```

2.2 NPCs attack character

A Player class was created which had all functionality to move the player, shoot a bullet and have score and health.

A parent Enemy class, which would control all aspects of any child enemy classes was then added to the game. This class had functionality such as taking in images, drawing the enemy, moving the enemy towards the player, bullet collision and player collision. From this class, children classes such as the coloured Ghosts, the Demon class and the Shooter class, which used the functions in the Enemy class, were then added. The game was structured in this way so any enemy added could either use the inherited functions of the enemy, or, in the case of the Shooter class, use slightly different functions. The Shooter class was the static enemy that could fire projectiles and is the only enemy targeted by the player's special ability, this enemy class being the enemy with the largest amount of health (See section 2.2.3 for enemy characteristics)

An EnemyManager class, which only takes in the Enemy parent class, was then created. This class could spawn random enemies using a switch statement and controlled the amount of different enemies in the game, moving the enemies, bullet and player collision and deleting enemies from memory. Once this structure was in place, it became easier to adapt the Shooter class, and if time allowed, add more different enemies to the game as they would inherit functionality from the parent Enemy class.

2.2.1 Spawning of enemies

The movable enemies such as the coloured ghosts and the demon are randomly spawned from a switch statement within the Enemy Manager class, slightly off the edges of the world map every 7 seconds at the start of the game and move towards the player with varying speeds dependent on which enemy is spawned. Their spawn functionality is within their parent Enemy class and they are spawned off the edge of the world map so that, if the player is on the edge of their movable area and therefore still within the world map, the moveable enemies will still spawn outside of the camera view.

The Shooter enemy, which is static and therefore can not move towards the player, overrides the Enemy parent class with its' own Spawn functionality and movement towards the player functionality and all Shooter enemies are spawned off the edge of the world map. This is discussed in more detail in 2.2.5.

2.2.2 Increasing frequency

This spawn time, which originally starts at 7 seconds is slowly decreased by 0.2 as the game continues, increasing the intensity of the game by increasing the amount of enemies being spawned into the game. The spawn delay never drops below 0.5 seconds so there will always be more and more enemies spawning into the game until the player is eventually dead. 7 seconds was chosen as a suitable time as the enemies have to move to the player's movable area from outside the edges of the world map.

2.2.3 Different enemies

There are 4 different movable enemies, RedGhost, WhiteGhost, BlueGhost and Demon who have different speed values and health values. All enemies have their own appearance/image and have the ability to change their image to either left or right, depending on where the player is on the screen, so the enemy is always facing the player. There is a 5th enemy called Shooter discussed in more detail in 2.2.5.

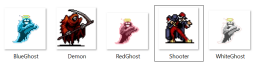

Figure 2: *Different enemies in game.*

The enemies shown in figure 2 have the following characteristics ...

- Blue Ghost - Speed 15, Health 50, points earned by player if killed 100
- Red Ghost - Speed 10, Health 100, points earned by player if killed 150
- White Ghost - Speed 7, Health 150, points earned by player if killed 200
- Demon - Speed 5, Health 200, points earned by player if killed 250
- Shooter - Speed 12, Health 300, points earned by player if killed 300

2.2.4 Directing enemies towards player

All movable enemies start at their world co-ordinates and move directly towards the player by using the difference in world co-ordinates between the enemy and the player which creates a direction vector. The magnitude of this direction vector is then found using Pythagoras' theorem and the enemy moves along this direction vector until the distance from the player reaches 10 or less. This value of 10 was chosen by play testing to prevent flickering if the two images overlap each other and also prevents division by zero when the normalised vectors are computed. A moveSpeedSlowRate variable was added for testing purposes which was adjusted to ensure that the enemies moved with a playable speed.

```
1  void Enemy::moveEnemyTowardsPlayer(Player& player)
2  {
3    float moveSpeedSlowRate = 0.1f;                                    // slows the enemy considerably,
       adjustable value
4    float dx = (player.playerX + player.cameraX) - enemyX;                        // use world co-ords of
       enemy and world co-ords of player
5    float dy = (player.playerY + player.cameraY) - enemyY;
6    float dxSquared = dx * dx;
7    float dySquared = dy * dy;
8    float distanceSquared = dxSquared + dySquared;
9    float distanceFromPlayer = sqrt(distanceSquared);
10
11   if (distanceFromPlayer > 10.0f)                                                // if distance is 0, the
       screen flickers as enemy on top of player so 10 is a good value
12   {
13     float NormalisedX = dx / distanceFromPlayer;
14     float NormalisedY = dy / distanceFromPlayer;
15
16     enemyX += NormalisedX * moveSpeed * moveSpeedSlowRate;
17     enemyY += NormalisedY * moveSpeed * moveSpeedSlowRate;
18
19   }
20
21   // while enemy moves, it changes looking direction based on player's position
22   if (enemyX < player.playerX + player.cameraX)
23   {
24     lookingRight = true;
25   }
26   else
27   {
28     lookingRight = false;
29   }
30 }
```

2.2.5 Static enemy with projectiles

The Shooter enemy spawns off the world map edge and is moved towards the player with a speed of 12. If it is within 300 units of the player, the Shooter enemy stops and becomes a static enemy. Therefore, it has its' own move functionality which overrides the parent Enemy class, because it has different movement behaviour from the other movable enemies in the game. After a short period, they fired blue bombs, spawned near the Shooter enemy. These blue bombs had the same functionality (overriding the parent Enemy class) as the coloured ghosts and demon, in that they move towards the player and follow them. The blue bombs can not be destroyed by the player, but if the player killed the creator of that blue bomb, both that shooter and its' respective blue bomb would also be destroyed.

2.3 Collision System

In terms of collisions, an idea of the bounding box collision was considered, but after a lecture regarding the River Raid game, the taught circle collision for every collision in the game was used.

2.3.1 Player and enemy

In regard to collisions between the player and the enemies, the player has a circle surrounding them, and the enemies each have their own collision circle.

Figure 3: *Collision circle between player and enemy.*

The difference in world co-ordinates between the enemy and the player is found, and this creates a 2D vector pointing from the enemy to the player. After some player testing, the radius of 50 was chosen, as this value was deemed to work sufficiently well for a reasonable collision. A comparison was then made between the 2D vector distance squared against this chosen radius squared and if the former was smaller than the latter, then a collision between enemy and player has occurred.

```
1  bool Enemy::enemyCollidePlayer(Player& player)
2  {
3    float dx = (player.playerX + player.cameraX) - enemyX;      // get world co-ords of enemy and player
4    float dy = (player.playerY + player.cameraY) - enemyY;
5
6    float radius = 50;
7    float radiusSquared = radius * radius;
8    float distanceSquared = dx * dx + dy * dy;
9
10   if (distanceSquared < radiusSquared)
11   {
12     player.health -= 10;
13     std::cout << "Player health: " << player.health << std::endl;
14     setEnemyDead();    // when enemies touch player, the player loses health and the enemy is set to
       dead and wil be deleted in Enemy Manager
15
16     if (player.health <= 0)
17     {
18       player.Dead();
19     }
20     return true;
21
22   }
23   return false;
24 }
```

This resulted in the player losing health of 10 per enemy collision and the enemy being destroyed on impact. The sqrt function was not utilised as this can be expensive when running in the main loop and the above method was sufficient enough to detect a collision. The enemy was also destroyed so there would be one less enemy in the game and the player has already been sufficiently punished by the collision. This also meant that, at some point, there will be too many enemies on the screen for the player to avoid and so the player will eventually die.

2.3.2 Player and water tile

In regard to collision between the player and detection of impassable water tiles, this was built within the main loop so water tiles could be detected on player movement.

The centre of each tile was first found, and if this tile was a water tile (water tiles are 1.png in the Resources file, see figure 9 in section 2.5.1 for all tiles used in the game), then the difference in the player's world co-ordinates and the water tiles centre world co-ordinates was calculated which formed a vector between the two centres. The radius of the water tile was then calculated along with the radius of the player image, and the squared distance between the vectors and then the sum of the radii were compared to each other. If this squared distance was less than the sum of the radii squared, then the player has collided with a water tile. The player's last position was stored and this is the position the player now had after impact with the detected water tile. This calculation had to be performed in the main loop for movement left, right, up and down as the player, on every movement key press, within the main loop, needed to be constantly detecting water tiles beneath their feet everytime they move.

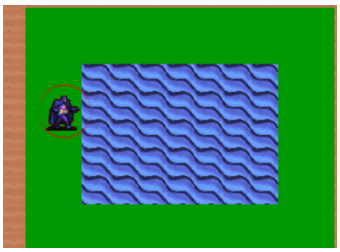

Figure 4: *Collision circle between player and water tile.*

2.3.3 Bullet and Enemy

On the key press of the player shooting, a bullet was produced which travelled horizontally from the player's position towards the edge of the screen, in the direction the player was facing. If the bullet reached either edge, it would be not drawn. The bullet has it's own collision circle and a technique of circle collisions, similar to player and enemy (2.3.1) was utilised.

Figure 5: *Collision circle between bullet and enemy.*

Again, after some play testing, the radius of 40 was chosen, as this value was deemed to work sufficiently well for a reasonable collision. On collision, the bullet would be destroyed (not drawn). It is understood that, although the bullet was not drawn after the collision, it still exists in memory and could be a potential memory leak but, since it will be reused when the player fires again, it is a feature, not a bug. Each enemy also has it's own enemyCollideBullet function that overrides the Enemy parent class, so that one bullet will deduct 50 from the health of each enemy until that enemy's health reaches zero and are dead, with all enemies having different health values as discussed in 2.2.3

2.3.4 Shooter Blue bomb and player

The static Shooter enemy can fire blue bombs at the player and the blue bombs have a similar collision to the coloured ghosts and the enemy, in that the blue bombs have their own collision circle. Again, when the blue bombs collide with the player, the blue bomb is destroyed (not drawn) and the player's health decreases by 10

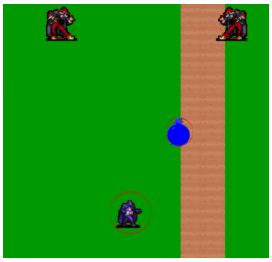

Figure 6: *Collision circle between blue bomb and player.*

2.4 Hero attacks NPC's

2.4.1 Linear Attack

The player can fire bullets, dependent on the direction they are facing and each bullet has it's own collision circle. Using a similar circle collision technique to player and enemy circle collisions, when the bullet collides with the enemy, the bullet is destroyed (not drawn) and the enemy is deleted from memory. The player can only fire again if the bullet is not on the screen, thus introducing a cool down effect if the bullet is still active on the screen.

2.4.2 Special Area of Effect

In regard to the special ability of the Player, timers were utilised timers which drew (draw cool-down) the ability on the screen for 3 seconds (four purple flames) and then a 15 second cool down timer (ability cool-down) would start after the ability disappears from the screen. The ability cool-down also resets back to 15 seconds if the player activated the ability too early i.e. before 15 seconds had passed. After this ability cool-down period had ended, the player would then be able to reactivate the special ability and the ability cool-down resets back to 15 seconds

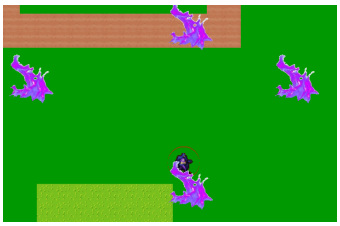

Figure 7: *Special Area of Effect.*

While building this, the player was punished too harshly if they tried to activate the ability before the cool-down ended but in the end, this feature was kept in the game. The Shooter enemy, and only this enemy, which can fire blue bombs and has the greatest health compared to all other enemies, is destroyed along with all bombs using this special ability.

2.4.3 PowerUp

A powerUp (purple clock) was also added which appeared after 60 seconds of game play. If collected by the player, the ability cool-down reduces from 15 seconds to 5 seconds. The powerUp is spawned at a random position within the player's movable area (Figure 1) only once in the game and is only rendered when within the camera view. It has its' own collision circle which can only be collided with by the player in a similar circle collision technique to the player and the enemies. Unfortunately, for the player, the powerUp may be spawned on a water tile within the movable area which is impassable by the player, which would result in the player not being able to collect it. This is a bug in the game and one where a solution was not found.

Figure 8: *PowerUp.*

2.5 Tile-based method for showing the background

2.5.1 Different tiles

There are 5 different tiles in the games stored in the resources folder and labelled as followed: 0 - Dark grass, 1 - water, 2 - path, 3 - rock, 4 - light grass. The water tile is the impassable tile by the player. However, the coloured ghosts and the demon can pass over this as well as the Shooter enemy.

The world map is made up of 6400 tiles ...

- 5024 Dark Grass tiles
- 426 Water tiles
- 568 Path tiles
- 303 Rock tiles
- 78 Light Grass tiles

Figure 9: *Different tiles in game.*

2.5.2 Loading the map

The map was loaded from a text file and consisted of a 80 x 80 grid of tiles which represents the entire world. This grid value for the world map allowed the player to have more of a movable area than the map originally given without the black background being shown at any time.

2.5.3 Infinite map

An infinite map was not able to be programmed but there was an idea of, when the player moves, when they reach a certain distance from the right edge of the screen, when moving right, for example, a vertical set of tiles would be created and stored. This set of vertical tiles would be added in the last column of the 2D map data array and every other vertical set of tiles in the array would be shifted left. The vertical tiles in the first array of the 2D map data array would then be deleted. This would keep the 2D map data array the same size, but every vertical column of tiles has changed i.e. shifted left. The same would be performed with the player moving up, down or right i.e. keeping the 2D map data array the same size but shifting columns or rows. However this wrap-around approach was not implemented in the game unfortunately.

2.5.4 Fixed map

A fixed map of 80 x 80 tiles was created to represent the world with a black border to represent the edge of the world. The player was programmed so that they were unable to leave the confines of their movable area and the camera only renders what is shown with the confines of the camera view dimensions.

2.6 Game Level running

A video has been included to show the game in action and can be found at the github link in Section 1. It is roughly 3 mins long and shows all features of the game as discussed in this report.

2.6.1 Fixed map

The finite map only is shown in the video along with gameplay showing all features of the game as described above. The infinite map was not able to be implemented

2.6.2 Score and FPS

The score is constantly updated in the console window as well as the frame count. When the player health reaches zero, a final score count is displayed and the words "Game Over" are printed to the console

2.7 Saving and loading the game

The player starts a game with starting values of a score of 0, health 500 and starting position in the world co-ordinates of (512, 384), with playerX as 512, playerY as 384, cameraX as 0 and cameraY as 0. The number of enemies (enemy count) in the game is 0, the enemy spawn time is set to 7 seconds and the powerUp delay time is set to 60 seconds from the start of the game.

Code was added so that when the player exits the game by pressing escape, at any point during gameplay, the player's score, their health and their location in the game was saved to a txt file. The number of enemies in the game and each enemies' world position along with the world position of any blue bombs in the game was also saved to a txt file. However, the type of enemy still existing in the game at this saved point was unknown and was attempted to be found in the Enemy Manager using a variable called ghostNumberType which recorded the type of enemy spawned in. A way to pass this information to the txt file was not found though due to lack of time.

When the player plays again, having saved from the previous time, the last position of the player and their score and health are returned from the txt file back into the game, so the player starts from the position at which the game was saved with the stats that the player had at this point. As the type of ghost was not found, the enemies existing at this point could not be passed back in unfortunately, due to not knowing what type of enemy to add to the game, even though their position was known.

If the user wished to start again and press the 'N' key at any point during gameplay, the players position, score and health would be reset back their original starting values. The enemies would be deleted from the game (set to dead status then deleted from memory) and the enemy spawn time would reset back to 5 seconds and the powerUp timer would reset back to 60 seconds.

3 Evaluation Section

3.1 Measuring FPS

The FPS starts at around 39 and, after playing the game with player health set to 500 for about 3 mins, and never killing an enemy, and only having enemies collide with the player, and avoiding all enemies for as long as possible, eventually the player died. During this game period, the FPS dropped to no lower than 31 by the end. The max size of the Enemy Manager is 500 which is enough enemies to kill the player as the enemies almost filled the screen at this point and were constantly colliding with the player and subtracting health until the player died. This was repeated several times and the FPS showed similar results.

4 Limitations

4.1 Limitations

If the game was built again, a closer watch would have been kept on the warnings that were appearing in the console, and solving these as the building of the game progressed. In the end, this number was reduced from 105 down to about 20 warnings, which mainly involved float to int conversions and unsigned mismatches. There was a slight problem with calculating the FPS which the lecturer assisted with. In terms of deleting the enemies from memory, the assistance of a private tutor was required but the concept was explained simply and the code commented sufficiently for the learner's understanding. Deleting enemies from memory did not form a part of the assignment marks, but this needed to be implemented as the enemies could be killed eventually by bullets but not disappear from the game and therefore still collide with the player. There was a desire for the player to be able to shoot the blue bombs thrown by the Shooter enemy and destroy them and keep the bomb destroy functionality separate from the Shooter destroy functionality when using the special ability, but a way to distinguish between the two was not found, so the special ability destroys both Shooters and bombs. However, if the Shooter throws a blue bomb and the player shoots the Shooter creator of this bomb, both the Shooter and Projectiles are destroyed. The bullet is also dependent on the

player's movement, so if fired, and the player moves up while the bullet is travelling, the bullet moves up. It was determined that this was an interesting feature as the bullet became a tracer bullet and so was left in the game.

5 Conclusion

5.1 Conclusion

In this project, a 2D survival game inspired by Vampire Survivors using C++ and the GamesBaseEngineering framework was implemented. The game demonstrates key concepts in game development, including player movement, enemy spawning, collision detection and managers along with cpp and h files which makes the code relatively easy to read and easy to implement and to check for bugs and errors. By implementing efficient algorithms such as circle-based collision checks and dynamic world updates, the game achieves relatively smooth performance even with multiple enemies and projectiles (blue bombs and bullets) on screen.

The project also highlights the importance of modular design and code reusability, allowing for easier expansion of features like new ghost enemy types, which could be created as their own class, and utilise the Enemy parent class and then be spawned in the Enemy Manager class.

Overall, this project reinforces practical C++ programming skills learnt by the programmer and provides a demonstration of fundamental game development principles such as real-time updates, memory optimization for deleting enemies and user interaction. Future improvements could include a infinite map, enhanced AI behaviour for enemies and additional gameplay mechanics to increase gamer enjoyment such as a variety of powerUps, different weapons for the player which can be collected and the addition of experience points, which, when collected through the scoring system (10XP for every 1000 points scored, for example), would unlock more special abilities.

6 Different Approach

6.1 Different Approach

There was an idea of building a character class as a parent of both enemy and player since they both have common variables such as health, speed, looking right or left, but this seemed overly complicated and there was a desire to keep their functionality separate, and not have too much inheritance occurring. The current structure though is as optimised and as simple as could be made, with comments and print strings throughout, given the programmer's current skill set and very little previous C++ knowledge before commencing the course. In hindsight, a closer eye should have been kept on the amount of warnings being shown on the console, but, at this time in the programmer's learning path, to have the game compile with no errors, only warnings was sufficient enough. The game was also originally built without the map moving, so the player could move around the screen freely. After introducing the fixed player at the centre and the map moving instead, almost everything had to be re-adjusted i.e. enemies being drawn only when they were visible. If the project were to be built again, a potentially optimal starting point would be starting with the fixed player at the centre of the screen and the moveable map, and build each enemy as the game progressed rather than re-adjusting everything at the end.

7 Additional

The sprites were taken from the following website

https://www.spriters-resource.com/pc_computer/vampiresurvivors/