Mustafa Demir
June 13, 2022

Contents

1 Introduction

This project focuses on designing and implementing extensions to an existing game written in C. The provided program is a two-dimensional platform game utilising SDL2 library for skeleton features such as handling graphics and taking input.

The goal is to extend the game by adding new features and modifying the existing ones, taking other platform games as reference and the skeleton code as the base. Many new features have been designed, adapted, modified and developed the code to implement these features. This report includes description of the design and implementation of the changes made. The development has been done under a Windows machine.

1.1 Project Goals

- Adding enemies that move and attack the player
- Adding kunai that can be thrown, as a weapon for the player and the enemy
- Adding health feature for the player as it can take damage
- Adding collectable hearts as items that heal the player
- Adding sprinting and power-jumping
- Adding stamina feature for the player as it uses stamina to sprint and power-jump
- Adding a health bar and a stamina bar over the player
- Adding idle, running, jumping and attacking animations for the player and the enemy
- Adding doors that connect to each other and keys as collectable items to unlock them
- Changing the texture for the map tiles, platforms and blocks and adding a background texture
- Adding background map tiles with no collision
- Adding a timer in the HUD to show the time passed
- Adding new levels to play and a level for tutorial to show how to play
- Adding main menu, level select menu and a pause menu
- Changing HUD to show the health, stamina, timer and level status
- Changing some sound effects and adding new ones for hearts, keys, doors, kunai (weapon) and taking damage

These features are key features for a typical platform game and it seems reasonable to make these changes as they enhance the user experience and bring more fun to the game. The timer makes the game more engaging for the user. Having enemies and combat mechanics adds challenge to the game, which is mostly desirable in a platform game. General improvements on the map, textures and the HUD makes it feel more like a modern platform game. Adding animations for the player, enemy and the doors improves the quality of the game and contributes to user experience more than a single static frame. Adding new collectable items and entities and adding sound effects for them makes the game more interactive and engaging.

Just like adding new levels, adding menus like the main menu, pause menu and the level select menu are essential for a game like this and it transforms the game to something more than just completing a single level. Adding the sprint, power-jump and stamina features to the game enables players to take different game-play approaches. As the main point of the game is to find items around the map to pass the level, adding doors that connect to each other contributes to the feeling of progressing through the game as it adds new ways of travelling through the map and as the keys are another type of items that need to be found in the map.

1.2 Game Features

After familiarising with the existing skeleton program, an approach was determined on how to implement the features. Listed each feature and the behaviour of the code for the feature as well as some of the requirements for the feature to be implemented properly/efficiently.

Enemies are entities similar to the player but they can only move in a specified range from the point they are spawned at. They move in the same direction until they reach their range than they change direction and keep looping until the player enters their sight. When this happens, enemies stop moving and start attacking (throwing kunai) at the direction of the player until either the player dies, leaves their sight or the player kills them. To implement this feature, initialising the enemies as an entities is required, as well as destroying them when they get hit by a kunai or player hops on their head. Moreover, a calculation to check if the enemy position is within a certain range of the enemy position is required to detect the player in sight. Coloring each animation frame for the player to a distinct color is also required to be able to easily tell the enemy apart from the player.

Kunai is the main weapon in the game and when it hits the player or an enemy, it decreases their health. It can be spawned by the player with a mouse click, or by enemies when they see the player. It is required to de-spawn them when they do not hit anything in a specified range to avoid them from going infinitely and to free the memory used by them. It is also required to detect and take action if a kunai kills the player.

Player health and collectable hearts work together as the health increases when a heart is collected. The health is decreased when the player is hit by a kunai. It is required to check if the player has missing health before increasing the health to not to go over the maximum health limit.

Player sprints when the shift key is pressed while moving and if the player has stamina. Also, holding down shift and jumping triggers a power-jump where the player can jump much higher. The stamina is decreased quickly when the shift key is pressed, and is increased slowly when the player is back to its normal speed. Checking if the next iteration of decreasing the stamina makes it go below 0 is required as it decreases quickly and as the stamina should not go below 0.

A health bar and a stamina bar is displayed over the player's head, and the numeric values for them are displayed on the status bar at the top of the screen to easily keep track of their levels while playing. It is required to calculate the positions where the health and stamina bars displayed on the screen as the player's position on the screen is not fixed when it is close to the edges of the map.

Animations for the player or the enemy plays by default and when an action like jumping, running and attacking takes place. Keeping track of the frame is required to play the correct frame, and it needs to be ensured that the frame number does not go out of the index of the frames. It is also required to have a mirrored texture for each image and to detect the player's or enemy's direction to be able to load the correct animation for the current direction. Moreover, all the frames in the animations should be the same or close to each other in width and height to keep the hit-box of the player consistent and accurate.

Doors work as pairs where player enters through one, it comes out of (teleports to) the other door. The doors are locked by default and require to find and collect the associated key to unlock. When approached by the player, a door starts to play door opening animation and sound if unlocked, and plays the door closing animation and sound when the player goes away or uses the door and teleports. It requires to unlock the doors once the associated key is collected. It also requires to not to teleport the player to the centre of the other door as it would immediately activate the door and the player keeps teleporting between the doors.

The tile textures are changed to better suiting textures and background tiles are implemented in addition to the default foreground tiles for design. Background tiles are placed just like the foreground ones but they do not collide. Also the single colour background is replaced with a background image to enhance the level ambiance all together with the map. It is required to omit the background tiles when the collisions are checked for each entity for them to act as if they are behind everything in the map.

Timer starts when a level is started to be played and calculates the seconds and minutes passed while playing and shows it on the HUD at the top of the screen. It is required to reset the timer when the user starts to play any of the levels and to pause the timer when the pause menu is opened.

Multiple levels with different maps and entities listed as playable levels with the first one being the tutorial level with tutorial messages to help the user to familiarise with the controls. The level info that shows the currently played level is added to the HUD at the top of the screen to keep track of the levels. When the user finishes a level, it automatically goes to the next level. It is required to check if the level finished is the last level and, if it is, to go to the main menu instead. Altering the default map and entity loading is required as well as creating levels from scratch to produce the files to load the new maps and entities from.

Main menu is the landing menu for the program and it has buttons that can be hovered and clicked. Buttons are for the main menu are "Play" to start playing the current level (first level by default), "Levels" button for the level select menu and an "Exit" button to exit the game. Level select menu is like the main menu and shows the available levels on the buttons and starts playing the selected level when a button is clicked. Finally, the pause menu is activated when the ESC key is pressed and deactivated when ESC is pressed again. In this menu, the game is paused and the buttons on the menu are "Continue" for continuing the game, "Levels" to go to the level select menu and "Main Menu" to go back to the main menu. It is required to keep track of the levels and the menus and to clear the levels as they are changed/finished to be able to replace the player, entities and the map with the new ones and to free the memory.

Changed and newly added sound effects reflects to the actions they are associated with.

Collecting a heart or a key makes a positive sound. An opening/closing door makes a door opening/closing sound. Throwing a kunai makes an attacking/throwing sound. Taking damage makes a hurting sound. It is required to manage the channels these sounds are played from to avoid overlapping.

1.3 Programming Style

Provided skeleton code utilises SDL2 library to handle graphics, sound, user mouse/keyboard input. At `parallelrealities.co.uk`, there are explanations for the skeleton code and how it was constructed. After examining the skeleton code and familiarising with the logic of it and seeing how the features already in place are implemented (using structs and defining types e.g.), utilising similar approach for the new features and sticking with C only seemed more reasonable.

Sticking with C only and using structs as main data structures is preferable for implementing the listed new features as it is much easier to add many of the features like the enemy, door, key, kunai and heart as it is possible to simply use the *Entity* struct to implement each of them. And the rest of the features does not require any object-oriented programming or benefits from using C++ to implement them. Moreover, the way the entities are stored as a linked list makes it easier to destroy and free the entities when the level is cleared. The struct *App* is also very useful to store the game state values meaningfully.

2 Design

In terms of how the game-play should work, when the user launches the program, the main menu comes up with the options to start the actual game, change the level user wants to play, and to exit the program. When playing the game, dying returns the user to the main menu, collecting all the pizzas loads the next level or the main menu if the level was the last level, changing the level through the pause menu clears the current level and loads the selected level, and finally, going back to main menu through the pause menu loads the main menu.

For the feature design, to realize the features to achieve a game-play as described, a menu feature should be implemented first. To implement the menu feature, a value should be stored globally to represent which menu to load and show. Best way of doing this would be storing the value inside the *App* struct and accessing it through the *App* type variable "app" as in "app.menu". This value is initially set to 1 initially to represent the main menu, 2 for pause menu, 3 for level select menu and it is set to NULL to represent that there is no menu to be displayed and the actual game should keep running. And in the main loop for the game, this "app.menu" value should be checked to load the correct menu and pause the game accordingly.

In Figure 1, the described logic flow is shown by demonstrating how the menus function and can be interacted by the user. Figure 2, Figure 3 and Figure 4 show how the described features are implemented in the game.

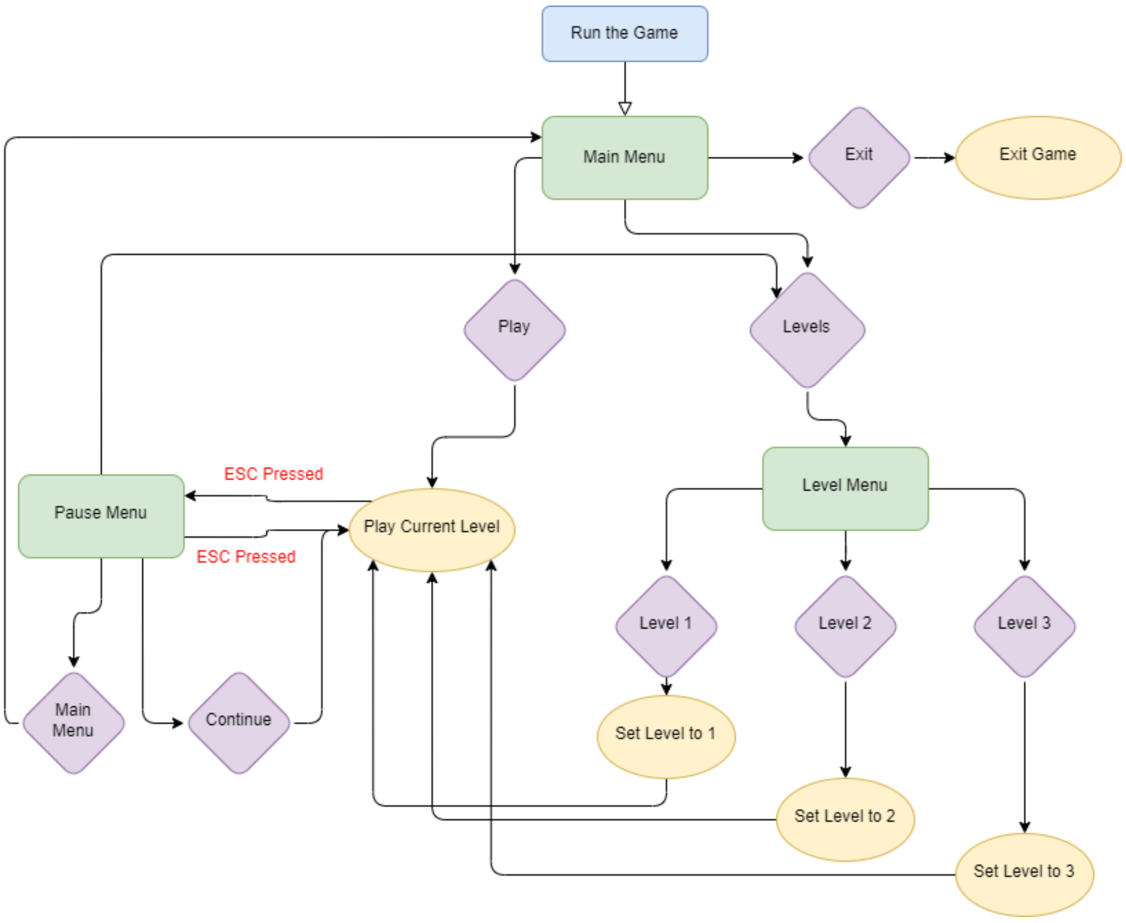


Figure 1: Game/Menu Logic

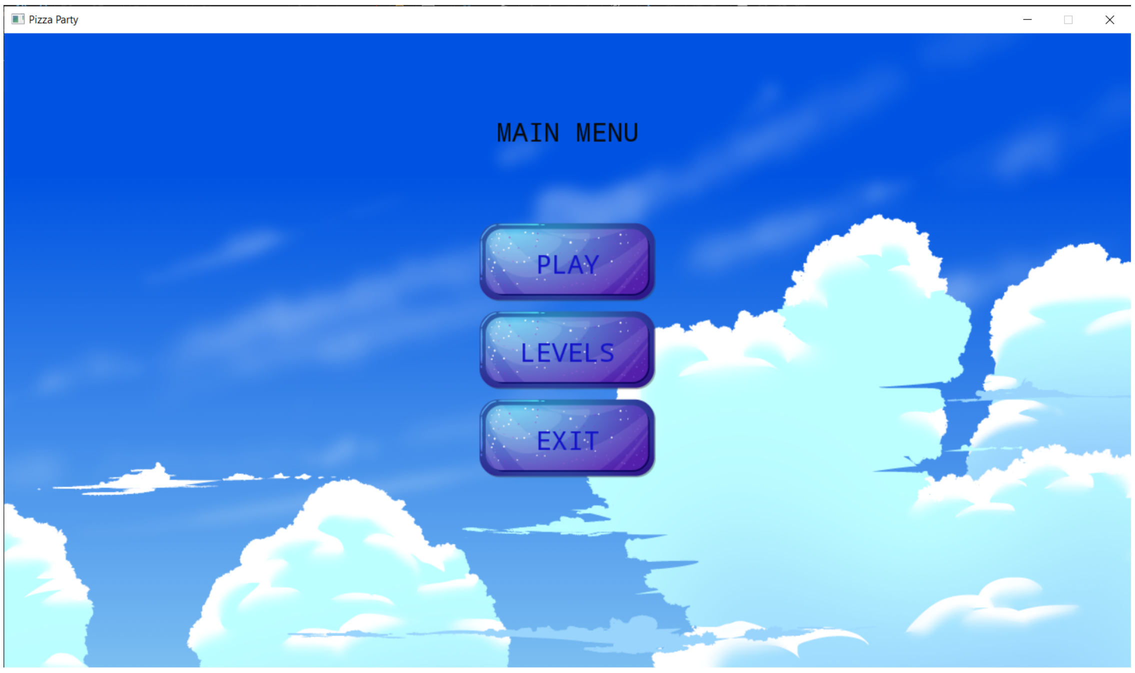


Figure 2: Main Menu

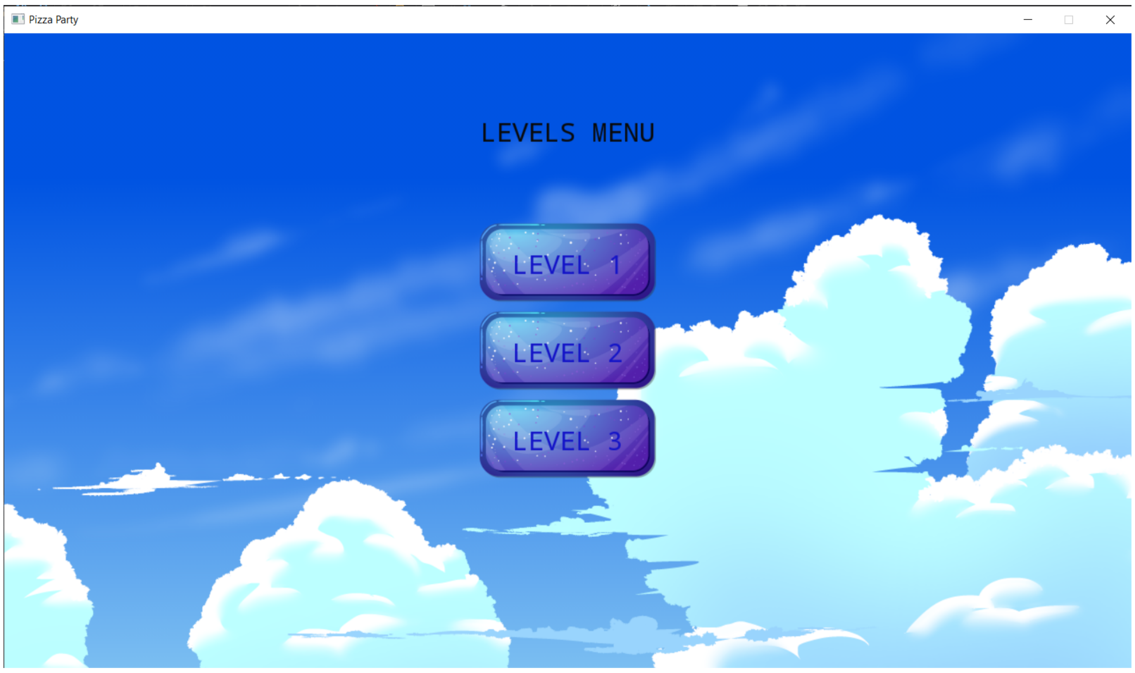


Figure 3: Levels Menu

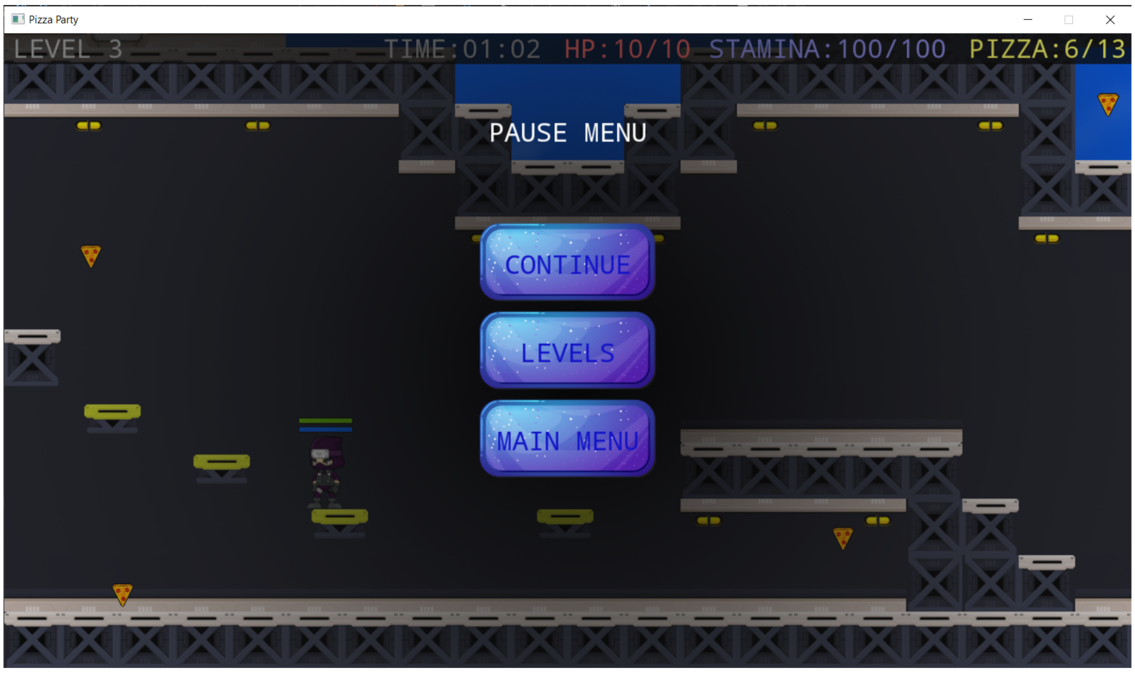


Figure 4: Pause Menu

To be able to implement the menu feature, buttons for the menus are also implemented. A struct *Button* is made to store buttons. Also, buttons are stored in an array inside the "app" struct to be able to access them with "app.buttons[*buttonIndex*]". Listing 1 shows the struct implemented for buttons. Buttons are initialised when the game is initialised and the *doButtons()* function is called in the main loop when a menu is present, where hovering and clicking on a button is detected and the buttons are drawn on the screen.

```c
struct Button { //To store a button
  float x;  //X position of the button
  float y;  //Y position of the button
  int w;    //Width of the button
  int h;    //Height of the button
  int hovered;//Flag to store if the button is hovered
  SDL_Texture *texture; //To store the background texture of the button
};
```

Listing 1: Button struct

For the actual game-play, some inputs have been added or changed. 'A' or the left arrow key moves the player left. 'D' or the right arrow key moves the player right. 'W', the up arrow key or the space key makes the player jump. The left or right shift key speeds up the movement and boosts the jump. Finally, the ESC key opens/closes the pause menu. The 'I' key was removed from the inputs as it was unconventional for a jump key in a platform game and the action for the space key was changed to jumping for the same reason. Moreover, mouse click is used for throwing a kunai in the game. Figure 5 shows all the possible keyboard inputs for the game.

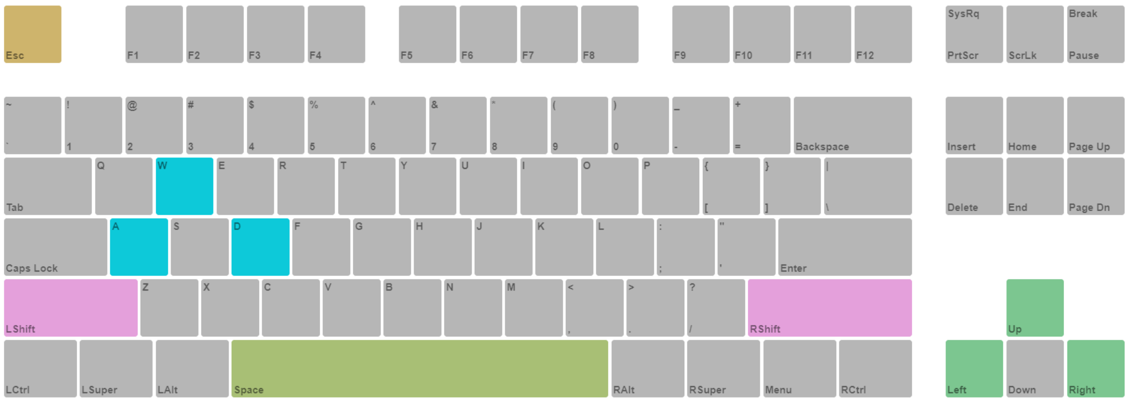
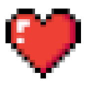
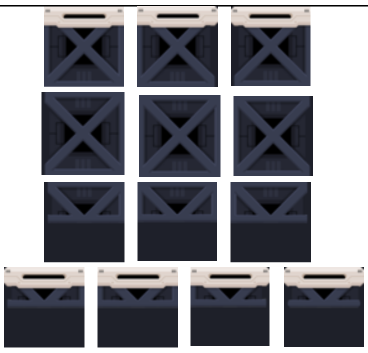
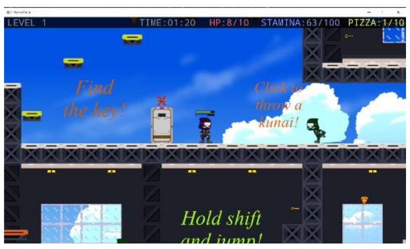

Esc
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
SysRq PrtScr ScrLk Break Pause
~ ` ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 _ - + = Backspace
Insert Home Page Up
Tab Q W E R T Y U I O P { [}] | \
Delete End Page Dn
Caps Lock A S D F G H J K L : ; " ' Enter
LShift Z X C V B N M < , > . ? / RShift
Up
LCtrl LSuper LAlt Space RAlt RSuper Menu RCtrl
Left Down Right

Figure 5: Keyboard inputs

The table below lists implemented features with explanations on how they are implemented and how they affect the game-play.

Feature	Frames	How it works in the gameplay?
Player (Changes)		Texture of the player is changed to fit with the game. Animations are added for the player where each type (idle, jump ext.) consist of 10 frames to make the movements smoother in the game to bring the feeling of a good platform game. Attacking feature is added to the player where the player throws weapons (kunai) in the facing direction to be able to defeat enemies. Player can also attack by jumping on enemies' head.
Enemy		The enemy mechanics are added where the enemy can attack the player with its weapon (kunai) when the player enters its sight. They move around within a range so player can have the option to avoid them. Running and attacking animations, each consisting of 10 frames, are added for the enemy as those are the only two actions they will have.
Kunai		Kunai is the main weapon in the game where it can be used (thrown) by the player or by the enemies. They give damage on hit or stick to a wall/object if they collide before de-spawning. They either belong to the player or an enemy where player's kunai does not damage the player and the enemy's kunai does not damage the other enemies. Sound effects were added to play when a kunai is thrown and when it damages either the player or the enemy.
Heart		Hearts are added to the game where they can be collected by the player and increase the player's health by 1. While the player's health is full, the heart cannot be collected. A sound effect was added to play when the heart is collected.
Doors		Doors were added where they come in pairs and with a key that unlocks both when collected. When the player enters through one of them, it comes out of the other pair. A key for a pair of doors is placed on a map and a sound effect is added to play when the key is collected by the player. When this key is not collected and the player approaches one of the doors it is associated to, a sign with a red cross

		and a key is shown on top of the door to convey that the key for the door is not collected. If the key is collected then the door is unlocked and when the player approaches the door, the added animations the added sound effect for door opening will start to play. Entering the door or getting away from it plays the door closing animations and sound. Finding keys to unlock doors to progress through the level is an iconic feature in the most platform games.
Menus		Menus are added where the user can choose multiple options by clicking the buttons on the screen. The main background image is used for the main menu and the level select menu. The pause menu is displayed on top of a transparent overlay to blacken the game screen. The title of the menu is displayed on the top side of the screen. The buttons are implemented in the game where a text is displayed on top of them. Hovering over the button changes the background image to a brighter version of it to give the feeling of hovering a button to the player. Clicking a button performs the action it is associated to.
Map Tiles (Foreground)		The tiles for the map are changed to ones with more suiting textures and the number of tiles were increased. Differences between some tiles might not be apparent, however, each of them is unique and the difference shows itself in the game.
Map Tiles (Background)		A new type of tiles is added to design the background of the map. These tiles have no collision as they represent the part of the map that is behind the player.
Levels	 LEVEL 1 TIME:01:20 HP:8/10 STAMINA:63/100 PIZZA:1/10 Find the key! Click to throw a kunai! Hold shift and jump! …	New levels are added to enhance the gameplay. Each level has different levels of difficulty and challenges. The first level is designed to teach player the basics of the added features that affect the game-play, for example how to open a locked door and how to throw a kunai.

3 Implementation and Development

3.1 Changes and Adaptations

The game differs from the original design in the ways of character design, game mechanics, map design and in program design. The main change in the game was the change in the program flow where in the original design, launching the program would start the only level available to play and beating the level by collecting all the pizzas in the level would result in exiting the program. In the new design, starting the program would bring up the main menu where the player can choose and start a level, and where the main exit point is. Beating a level either skips the game to the next level or brings up the main menu if it is the last level. This change is made by the changes in the main loop of the program. Listing 2 shows the new main function implemented with the changes in the main loop of the game. Comments describing the changes are included where the changes were made.

```c
int main(){
  long then;
  float remainder;

  memset(&app, 0, sizeof(App));
  app.textureTail = &app.textureHead;

  initSDL();

  atexit(cleanup);

  app.mapNo = 1;    //Set the map no to first map

  initGame();

  initStage();

  then = SDL_GetTicks();

  remainder = 0;

  app.menu = 1;   //Set the menu flag on to start from the main menu
  SDL_ShowCursor(1);  //Show the cursor

  while (1)
  {
    prepareScene();

    doInput();

    if (app.menu == NULL){    //If the menu flag is not set
      app.delegate.logic(); //Do game logic
    }

    app.delegate.draw();  //Do draw

    if (app.menu == 1) {  //If menu flag is set to main menu
      blit(loadTexture("assets/bg.png"), 0, 0, 0);  //Display the background image
      drawText(SCREEN_WIDTH / 2, 100, 10, 10, 10, TEXT_CENTER, "MAIN MENU");
    ↪ //Draw main menu text
      doButton(); //Do buttons
    }

    if (app.menu == 2)  { //If menu flag is set to pause menu
```

```
      blit(loadTexture("assets/overlay.png"), 0, 0, 0); //Display the overlay
      drawText(SCREEN_WIDTH / 2, 100, 255, 255, 255, TEXT_CENTER, "PAUSE MENU");
    ↪ //Draw main menu text
      doButton(); //Do buttons
      if (app.keyboard[SDL_SCANCODE_ESCAPE]) {  //If the escape key is pressed
        app.menu = NULL;  //Reset the menu flag to continue the game
        SDL_ShowCursor(0);  //Hide the cursor
        app.keyboard[SDL_SCANCODE_ESCAPE] = 0;  //Reset keyboard input for escape key
        Tstart += clock() - Ttemp;  //Add the time elapsed while on pause menu to
    ↪ start time
      }
    }

    if (app.menu == 3) {  //If menu flag is set to levels menu
      blit(loadTexture("assets/bg.png"), 0, 0, 0);  //Display the background image
      drawText(SCREEN_WIDTH / 2, 100, 10, 10, 10, TEXT_CENTER, "LEVELS MENU");
    ↪ //Draw levels menu text
      doButton(); //Do buttons
    }

    if (stage.pizzaFound == stage.pizzaTotal && app.menu == NULL) //If the level is
    ↪ finished and the game is running
    {
      if (app.mapNo < MAX_LEVEL) {  //If it is not the last level
        app.mapNo++;  //Increase map no
        clearLevel(); //Clear the level before start playing
        initEntities(); //Reinitialise entities
        initPlayer(); //Reinitialise player
        initMap();    //Reinitialise map
        Tstart = clock();//Start the timer
        seconds = 0;  //Reset counted seconds
      }
      else {  //If the last map is played
        app.menu = 1;   //Go back to the main menu
        app.mapNo = 1;    //Set the level to the first level
        SDL_ShowCursor(1);  //Show the cursor
      }
    }

    presentScene();

    capFrameRate(&then, &remainder);
  }

  return 0;
}
```

Listing 2: Main function and loop

As implementing the combat mechanics and the kunai features requires taking the mouse click input, some changes were made to original user input taking code to detect and store this input in the "app" struct as "app.mouseClick". Listing 3 shows the changes made, and explanations for them in the comments where the changes were made.

```
void doInput(void)
{
  SDL_Event event;

  while (SDL_PollEvent(&event))
  {
```

```c
    switch (event.type)
    {
      case SDL_QUIT:
        exit(0);
        break;

      case SDL_KEYDOWN:
        doKeyDown(&event.key);
        break;

      case SDL_KEYUP:
        doKeyUp(&event.key);
        break;

      case SDL_MOUSEBUTTONDOWN: //Get mouse button down input for mouse click
        app.mouseClick = 1; //Set mouse click flag
        break;

      case SDL_MOUSEBUTTONUP:   //Get mouse button up input for mouse click
        app.mouseClick = 0; //Reset mouse click flag
        break;

      default:
        break;
    }
  }
}
```

Listing 3: Taking input

The HUD is adapted to include the information on the new features such as the level info, timer, health info and the stamina info, as well as the pizza info that was originally there. New colors for the texts are set to distinguish the info. On top of this, health and stamina bars are implemented to easily keep track of the health and stamina values. The time calculations were made for the timer before displaying it in the HUD. Figure 6 includes a screenshot that shows the new HUD status bar and the health and stamina bars.

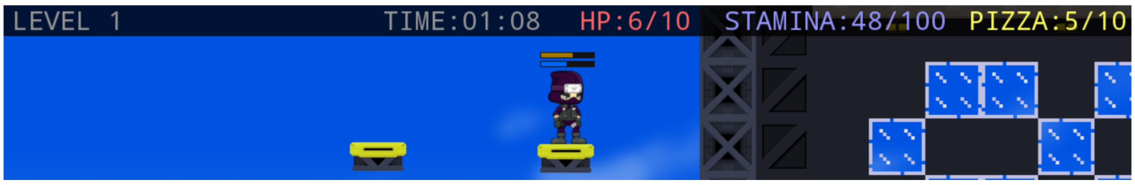

LEVEL 1 TIME:01:08 HP:6/10 STAMINA:48/100 PIZZA:5/10

Figure 6: HUD

As new entities were added to the game, some changes and adaptations were made to the *Entities* struct to implement the functionalities of the new entities, as well as changes to the function where the entities are loaded from the file to be able to load the new entities.

Moreover, some define values and enumerations from the original game were changed and some new ones are added as the old values did not provide the best experience for the user and the added values were necessary for the newly developed features. Listing 4 includes the changes made to the existing definitions, as well as the added definitions and enumerations.

```c
#define PLATFORM_SPEED 4
#define MAX_TILES    15
#define PLAYER_MOVE_SPEED 10

```

```
#define TOTAL_HP 10     //Total health of the player
#define TOTAL_STAMINA 1000  //Total stamina of the player (10x)

#define ENEMY_MOVE_SPEED 3  //Move speed of enemies
#define ENEMY_SIGHT 200   //Range of sight where enemies start to see the player

#define THROW_RANGE 220   //Range of kunai
#define KUNAI_SPEED 10    //Speed of kunai

#define MAX_LEVEL 3     //Maximum number of levels to play

enum
{
  SND_JUMP,
  SND_PIZZA,
  SND_PIZZA_DONE,
  SND_OPEN_DOOR,  //Door opening sound
  SND_CLOSE_DOOR, //Door closing sound
  SND_HEART,    //Heart collecting sound
  SND_KEY,    //Key collecting sound
  SND_HIT,    //Getting hit by a kunai sound
  SND_KUNAI,    //Sound of kunai being thrown
  SND_MAX     //Maximum number of sounds
};

//Enum for directions
enum
{
  RIGHT,
  LEFT
};

//Enum for entity types
enum
{
  PLAYER,
  PIZZA,
  HEART,
  PLATFORM,
  BLOCK,
  KUNAI,
  ENEMY,
  DOOR,
  KEY
};

//Enum for buttons
enum
{
  PLAY_BTN,   //First button on the menu to play the game
  LEVELS_BTN,   //Second button on the menu to go to levels menu
  EXIT_BTN,   //Third exit button on the menu to exit the game
  MAX_BUTTONS,  //Represents the maximum number of buttons
};
```

Listing 4: Definitions and enumerations

3.2 Development

The development process started with changing the assets and preparing the scene and the environment for the upcoming features. New assets for the map and the player animations were obtained. The textures for the map were replaced as well as adding the feature of background tiles. To be able to implement the functionality for the background tiles, these tiles should be omitted on collision checks for the entities to the world (tiles). In order to clearly and meaningfully separate the background tiles from the foreground tiles, negative indexing is used for them in the map files. When the map is loaded and initialised, these tiles with a negative index are treated differently and added to the background tile group. This way the two types were separated and the player or the other entities prevented from colliding with them. The textures for the entities "platform" and "block" were also replaced to suit with the new tile-set.

After this, the development for player animations were started. To implement animations for the player, each frame for the animations were needed to be loaded and stored for usage. Then, when the logic for the player is done, the next suitable animation frame should be loaded. To do this, an integer value to keep the index of the frame should be stored. This value should increase every *doPlayer()* call, but when this value used as the index of the frame to be loaded, it should not be out of index. Therefore, this value to keep track of the frame index should be capped at a specific value and be reset to 0 when this value is exceeded. When the indexing is sorted, texture of the player should be set to the texture at the calculated index of the appropriate animation group. For example, when the input is taken for going left, the next frame in the animation group *moving_left[]* should be accessed. Arrays are used as the programming feature to implement the animations. Listing 5 shows how this feature was implemented where line 6 is for keeping track of the frame, line 7 for the idle animation, 22 for moving left animation, 30 is for moving right animation, 58 is for jumping up (left and right) animation, 59 is for jumping down/falling (left and right) animation and 62 for attacking (left and right) animation.

```
 1 void doPlayer(void) //Player function to run every frame
 2 {
 3   player->dx = 0; //Reset x speed
 4   player->speed = 1;  //Reset player speed multiplier to default
 5 
 6   frame = (frame + 1) % 20; //Increase frame number (goes back to 0 when reaches 20)
 7   player->texture = (player->facing == RIGHT) ? idle_right[frame / 2] :
       ↪ idle_left[frame / 2]; //Set the idle frame according to the frame number
 8 
 9   if ((app.keyboard[SDL_SCANCODE_LSHIFT] || app.keyboard[SDL_SCANCODE_RSHIFT]) &&
       ↪ player->value > 0) {  //If either shift is pressed and the player has stamina
10     player->speed = 2;    //Double the player speed multiplier
11     player->value -= 5;   //Decrease the stamina
12   }
13 
14   if (player->speed == 1 && player->value < TOTAL_STAMINA) player->value++; //If
       ↪ player has the default speed and is missing stamina, reload stamina
15 
16   if (app.keyboard[SDL_SCANCODE_A] || app.keyboard[SDL_SCANCODE_LEFT])  //If A key
       ↪ or the left key is pressed
17   {
18 
19     player->dx = -PLAYER_MOVE_SPEED * player->speed;  //Set player x speed to
       ↪ negative player speed times the speed multiplier
20     player->facing = LEFT;  //Change facing flag to left
21 
```

```
    player->texture = moving_left[frame / 2]; //Set the moving left frame according to the frame number
  }

  if (app.keyboard[SDL_SCANCODE_D] || app.keyboard[SDL_SCANCODE_RIGHT]) //If D key or the right key is pressed
  {
    player->dx = PLAYER_MOVE_SPEED * player->speed;   //Set player x speed to positive player speed times the speed multiplier
    player->facing = RIGHT; //Change facing flag to right

    player->texture = moving_right[frame / 2];  //Set the moving right frame according to the frame number
  }

  if ((app.keyboard[SDL_SCANCODE_SPACE] || app.keyboard[SDL_SCANCODE_W] || app.keyboard[SDL_SCANCODE_UP])&& player->isOnGround) //If the space key, up key or W key is pressed and the player is on the ground
  {
    player->riding = NULL;  //Set riding to NULL as when jumped, player will not be riding any entity

    player->dy = -20 * player->speed; //Give player an amount of negative y speed (to go up) times speed multiplier

    playSound(SND_JUMP, CH_PLAYER);   //Play jumping sound on player channel
  }

  if (app.keyboard[SDL_SCANCODE_ESCAPE]) {  //If the escape key is pressed
    app.menu = 2;   //Set the menu flag to pause menu
    SDL_ShowCursor(1);  //Show the cursor
    app.keyboard[SDL_SCANCODE_ESCAPE] = 0;  //Reset keyboard input for escape key
    Ttemp = clock();  //Set the temporary time to stop the timer in pause menu
  }


  if (tempClick != app.mouseClick && app.mouseClick == 1) //If the state of the click changes (if mouse button clicked)
  {
    attacking = 1;  //Set attacking flag true
    frame = 0;    //Reset frame
  }

  tempClick = app.mouseClick; //Set the temorary click flag to current mouse click to be able to detect the change in mouse click state on the next iteration

  if (player->dy < 0) player->texture = (player->facing == RIGHT) ? jumping_right[abs((int)((-1 * player->dy) / 5 - 4))] : jumping_left[abs((int)((-1 * player->dy) / 5 - 4))]; //Iterate through jumping up animation y speed is negative (according to player's direction)
  if (player->dy > 0) player->texture = (player->facing == RIGHT) ? jumping_right[(int)(player->dy / (18/4.9) + 5)] : jumping_left[(int)(player->dy / (18 / 4.9) + 5)];     //Iterate through jumping down animation y speed is positive (according to player's direction)

  if (attacking) {  //If attacking flag is on
    player->texture = (player->facing == RIGHT) ? attack_right[frame / 2] : attack_left[frame / 2]; //Set the correct frame according to player's direction and the frame number
    if (frame == 7) spawnKunai(player->x + 15, player->y + 30, (player->facing == RIGHT) ? 1 : 0, 1); //Spawn a kunai at the centre of the player on the same
```

```
       ↪ direction as the player if on the 7th frame
64     if (frame == 19) {  //On frame 19
65       attacking = 0;  //Set attacking flag 0
66       frame = 0;    //Reset the frame
67     }
68   }
69
70   if (player->riding) { //If player is on top of an entity
71     if (player->riding->ex && !player->riding->ey) {  //If the entity is an enemy
       ↪ (has ex but no ey)
72       player->riding->health--; //Decrease enemy health (kills the enemy when jumped
       ↪ on its head)
73       player->riding = NULL;    //Clear riding
74       player->dy = -15;     //Give player some negative y speed to make it hop
75     }
76   }
77 }
```

Listing 5: Developed *doPlayer()* function

After the animations were implemented, sprint, power-jump and stamina features were to be implemented next. These features are implemented at the same time as they all relate to each other. The idea behind this feature is to enable the player to go faster or jump higher when it has the stamina to do so. Therefore, the stamina feature is implemented where the stamina of the player increases each *doPlayer()* call if it is not full already, and pressing either of the shift keys would make the player's movements faster and jumps higher, while reducing the stamina. If the stamina is emptied, these granted powers are removed. A speed multiplier value is used to implement this functionality and the default value of the speed multiplier is set to 1 and is multiplied my the move speed when moving or by the jump magnitude when jumping. Activation of these powers increases the speed multiplier value of the player which results in an increase in the speed when moving/jumping. Listing 5 shows the *doPlayer()* function this feature is implemented where the lines 4, 9-15, 19, 27 and 37 relate to this feature.

The next feature to implement was the weapon (kunai). As the enemy feature to be implemented next, implementing the weapon was the priority. The development of the kunai were to consist of spawning and moving it in a direction until it hits something or de-spawns. To initialise a kunai, the direction information should be given as well as the position info and the owner info. The texture of the kunai is loaded according to the given direction info on initialising. On the tick function of the kunai, the kunai moves according to its fixed speed if it the distance between its current position and its start position is not bigger than the fixed kunai range, and it is destroyed otherwise. The kunai also has a touch function just like a pizza, where if the entity it touches is the player or the enemy that will be implemented next, and the the one it touches is not the same type as the entity its owner is, it reduces the health attribute of the entity and destroys the kunai. However, if the collision happened with a wall, a block, or the edge of the map, then the kunai sticks there. This as allowed as it makes the game more realistic. If the player is being hit with enemy's kunai and has no health left, the player dies and the main menu is loaded. In Listing 5 line 63, it is shown how player calls the function to spawn a kunai. Figure 7 shows how the kunai is used in combat both by the player and the enemies.

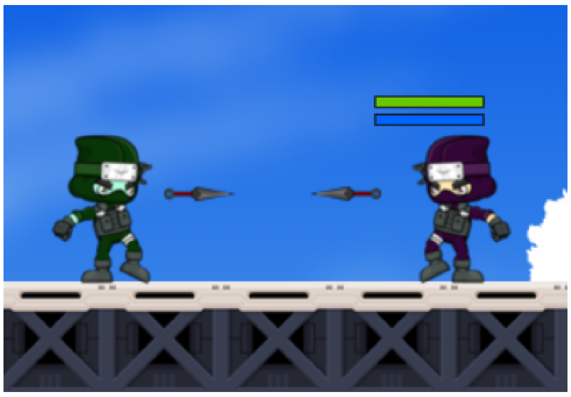


Figure 7: Combat and kunai

When the development of these features for the player and the kunai were finished, implementing the enemy feature came next. The idea of the enemy was to have an entity like the player that would attack the player when they were close to each other and move around the same path otherwise. To implement this, assets used for the player were altered to generate the enemy textures. After the assets were generated, the tick function for the enemy is implemented. In here, the movements of the enemy is calculated to keep it moving back and forth in a range and the distance between the enemy and the player is calculated to check whether to activate attacking to player or not. If the player enters the enemy's range, the move speed of the enemy is removed for the enemy to stop, then the attacking is activated, where the enemy spawns kunai in the direction of the player. Figure 7 shows the enemy in combat.

Once the combat mechanics were in place, the first feature with a new collectable item, health and hearts, came next. The idea was that the player has 10 health points which can be reduced by enemy attacks and the hearts collected will increase player's health. To implement this, a new type of entity "heart" were made. This hearts are initialised from the entities file and they act almost the same as pizzas where the tick function is the same to create a sin wave motion and the touch function is developed to check if the entity touching is the player and the player's health is less than player's maximum health. If so, player's health is increased, the heart is destroyed and the sound effect for collecting the heart is played, but if the player is on maximum health, then the heart is not collectable. Listing 6 shows how the initialising, tick and touch functions were implemented.

```c
void initHeart(char *line)
{
  Entity *e;  //Storing hearts as entites

  e = malloc(sizeof(Entity));   //Allocating memory for the heart
  memset(e, 0, sizeof(Entity)); //Reseting allocated memory
  stage.entityTail->next = e;   //Add entity to the linked list of entities
  stage.entityTail = e;      //Set the eitity as the last element on the linked list

  sscanf(line, "%*s %f %f", &e->x, &e->y);  //Set the x and y positions of the
    ↪ entity from the line

  e->health = 1;     //Set entity health
  e->type = HEART;  //Set entity type

  e->texture = loadTexture("gfx/heart.png");     //Load heart texture
  SDL_QueryTexture(e->texture, NULL, NULL, &e->w, &e->h); //Set width and height
    ↪ from the texture
  e->flags = EF_WEIGHTLESS; //Set the weightless flag as hearts should not be
```

```
    ↪ affected by gravity
  e->tick = tick;   //Set tick function for the heart
  e->touch = touch; //Set touch function for the entity
}

static void tick(void)  //Creates a sin wave motion
{
  self->value += 0.1;

  self->y += sin(self->value);
}

static void touch(Entity *other)  //Touch function for the heart
{
  if (self->health > 0 && other == player && player->health < TOTAL_HP) //If player
    ↪ has missing health
  {
    self->health = 0; //Destroy the heart
    player->health++; //Increase player health
    playSound(SND_HEART, 1);  //Play heart sound on channel 1
  }
}
```

Listing 6: Implementation of heart

Next, the doors and keys were to be implemented to enhance the level design and the game-play. The idea was that the player collect keys and unlocks doors to use them to teleport to other doors in the level to proceed. It was decided that the best way to implement this feature is to initialise 2 door entities and a key entity together and associate them to each other on initialising. This bundle of entities are initialised from one line on the entities file where the positions of the first and the second door, and the key is given. On initialising, the first door is initialised where its position is set to the first set of positions and the second set of positions is recorded as the position of the door on the other side, then the second door is initialised where the second set of positions is used for the position of the door and the first set of positions is recorded as the position of the door on the other side, finally a function is called to initialise a key, passing the position of the key to be initialised and the 2 doors it should unlock as entities. When initialising the key, the two doors are recorded. The tick function for the key is same as the pizza and the heart to create a sin wave motion. The touch function of the key is implemented to check if the entity touching is the player and if so, to access the doors it is associated to and change their locked/unlocked value to unlock them. Listing 7 shows how this function was implemented where the *target1* and *target2* are the doors associated with the key when it was initialised.

```
static void touch(Entity* other)  //Touch function for the key
{
  if (self->health > 0 && other == player)  //If the key is not collected and the
    ↪ touching entity is the player
  {
    self->health = 0; //Destroy the key
    self->target1->value = 1; //Unlock the door this key is for
    self->target2->value = 1; //Unlock the other door this key is for
    playSound(SND_KEY, 1);  //Play key sound on channel 1
  }
}
```

Listing 7: Key touch function

The tick function of the door is implemented to display an image at the top of the door to show that the player has not found the key yet if the door is locked, otherwise, to play the door opening animation and sound when the distance between the player's position and the door's position is less than the detection distance of the door. And it plays the door closing animation and sound once this distance is bigger than the range. The touch function for the door is implemented to check if the touching entity is the player, the player is very close to the centre of the door (as it is not realistic/smooth to teleport the player as soon as it touches the side of the door's texture), and if the door is unlocked. If this conditions are true, player is teleported to the location of the other door that was recorded plus an offset to correctly place the player. Moreover, the direction the player approaches the door is checked and when the player is teleported, another offset is added to place the player away from the door's centre in the opposite direction the player approached the other door. This provides a smooth transition, but more importantly, this prevents the player to be infinitely teleported between the two doors by placing the player at the centre of the door where the door will detect the player for teleporting. Listing 8 shows how the functionalities for the door were implemented.

```
static void tick(void)  //Tick function for the doors
{
  if ((abs(player->x - (self->x + self->w / 2)) < 150 && abs(player->y - (self->y +
    ↪ self->h / 2)) < 100)) { //If player is near
    if (self->value == 1) { //If the door is unlocked
      if (self->frame == 1) playSound(SND_OPEN_DOOR, 1);  //Play door opening sound
    ↪ on channel 1
        if (self->frame < 15) { //Until the last frame reached
          self->frame++;              //Go to the next frame
          self->texture = door[self->frame / 5];  //Set the frame as the texture
    ↪ (Creates the animation for opening the door)
        }
      }
      else {    //If the player is close but the door is locked
        app.noKeyPos[0] = self->x - stage.camera.x + self->w / 2; //Set the x
    ↪ position for the noKey image
        app.noKeyPos[1] = self->y - stage.camera.y - 20;      //Also set the y
    ↪ position for the noKey image
      }
  }
  else {  //If the player is away
    if (self->frame == 14) playSound(SND_CLOSE_DOOR, 1);  //Play door closing sound
    ↪ on channel 1 when frames goes down by one to 14
    if (self->frame > 0) {  //Until the first frame reached
      self->frame--;              //Go to the previous frame
      self->texture = door[self->frame / 5];  //Set the frame as the texture
    ↪ (Creates the animation for closing the door)
    }
  }
}

static void touch(Entity* other)  //Touch function for the doors
{
  if (other == player && (abs((player->x + player->w / 2) - (self->x + self->w / 2))
    ↪ < 10 && abs((player->y + player->h) - (self->y + self->h)) < 10) &&
    ↪ self->value == 1){ //If it is the player touching and the player is close to
    ↪ the centre of the door and close to the ground, and the door is unlocked
    other->x = self->ex + (self->w / 2) - (other->w / 2) + ((other->dx > 0) ? 15 :
    ↪ -15);  //Set player x to other door's location. If player came from the left
    ↪ teleport to the other door's left visa versa
    other->y = self->ey + self->h - other->h;     //Set player y to other door's
    ↪ location
  }
}
```

Listing 8: Door tick and touch functions

Lastly, multiple levels were implemented after finalising the game-play mechanics. To do this, the functions that load the game and the entities should be altered to fit the new multiple level design and the current level number should be stored inside a global variable. The selected level number is set to be stored in the "app" struct as "app.mapNo", and this should be checked when the map and the entities are loaded to load from the correct file. Listing 9 shows how these functions were implemented by showing the new map loading function as an example.

```
void initMap(void)
{
  memset(&stage.map, 0, sizeof(int) * MAP_WIDTH * MAP_HEIGHT);

  loadTiles();

  switch (app.mapNo)  //By current level
  {
  case 1: //Level 1
    loadMap("data/map01.dat");  //Load first map
    break;
  case 2: //Level 2
    loadMap("data/map02.dat");  //Load second map
    break;
  case 3: //Level 3
    loadMap("data/map03.dat");  //Load third map
    break;
  default:
    break;
  }
}
```

Listing 9: New function to load different maps

As levels can be initialised many times during run-time with the new functions and the new menu feature, another function were to be implemented to clear the already loaded level before loading the new one. In this function, loaded entities are removed from the linked list and freed, and the pizza counts were reset. Listing 10 shows the newly implemented function for this.

```
void clearLevel(void) {
  Entity* e, * prev;

  prev = &stage.entityHead; //Set previous to entity head

  for (e = stage.entityHead.next; e != NULL; e = e->next) { //Iterate through the
    ↪ entities
    prev->next = e->next; //Set the next of the previous to next of the current
    ↪ entity
    free(e);        //Free the current entity
    e = prev;       //Set the previous entity as the current entity
  }

  stage.entityTail = &stage.entityHead;   //Redefine the entity tail

```

```
    stage.pizzaFound = stage.pizzaTotal = 0;  //Reset the pizza counts
}
```

Listing 10: New function to clear the level

Moreover, new levels were needed to be designed. As the original level design was hard-coded in the *map01.dat* and *ents01.dat* files, the only solution was to manually enter the data. But, this seemed very inefficient, therefore, it was decided to implement a level editor where different entities and map tiles can be placed on the screen and a "dat" file is produced from this. It was decided that the easiest way of doing this would be developing a new external program, dedicated for this purpose. Therefore, a level editor program was designed before continuing. However, this is not included in the final project as it was not a part of this program and was external. Figure 8 shows the finished version of this external program with an example of how the levels were designed on this.

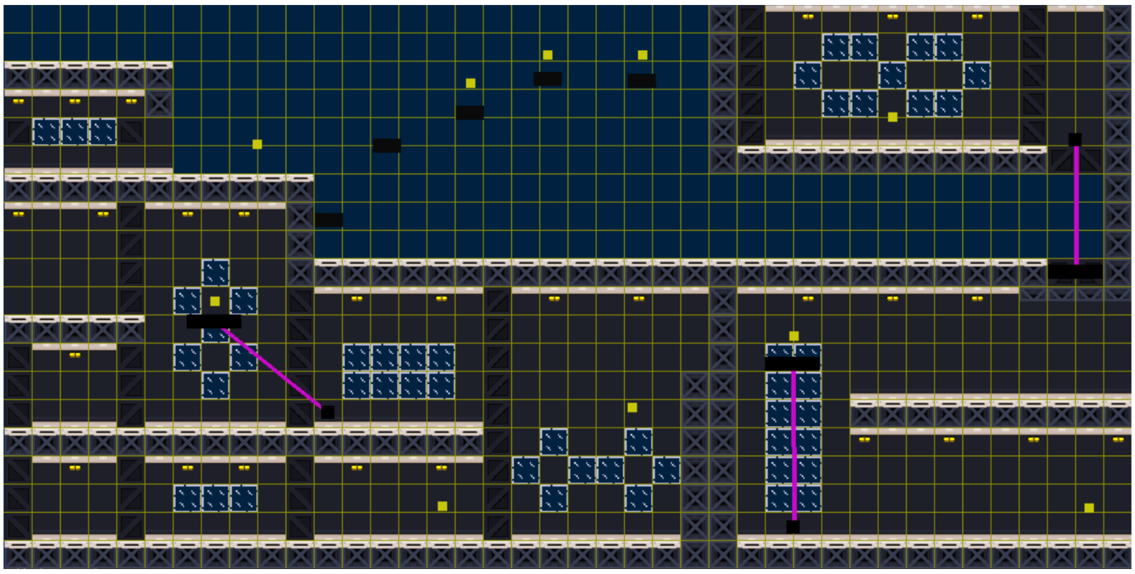


Figure 8: Designing Level 1 on the Level Editor

Once the preparations were over for implementing new levels, the original level was replaced with level 1, where the game-play is kept simple for the user that will be playing the game for the first time, and some tutorial messages were embedded in the level to help the user with the usages of some new features that might not be apparent to the user. Figure 9, Figure 10 and Figure 11 show the implemented levels.

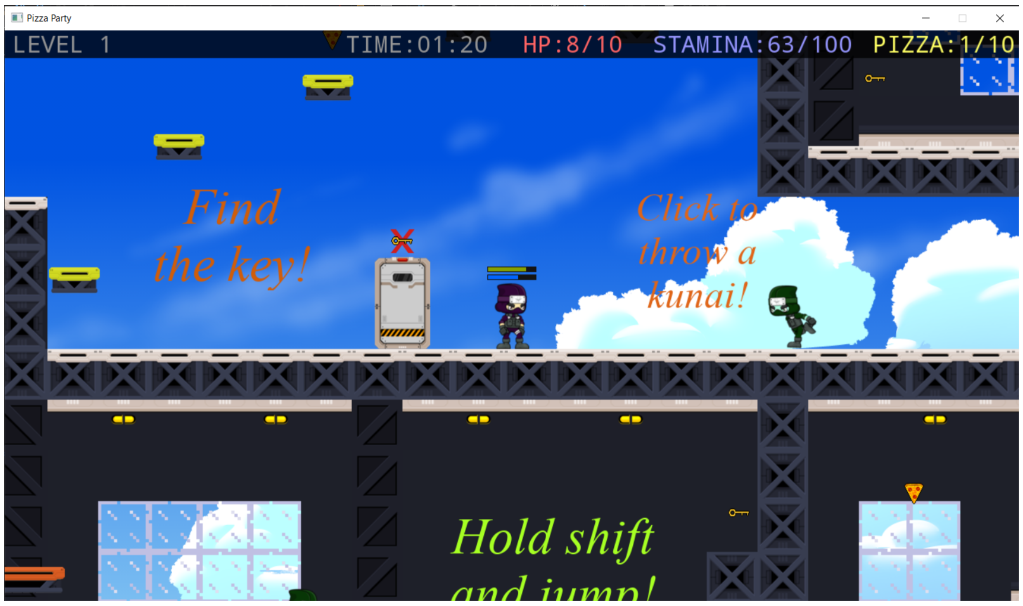


Figure 9: Level 1

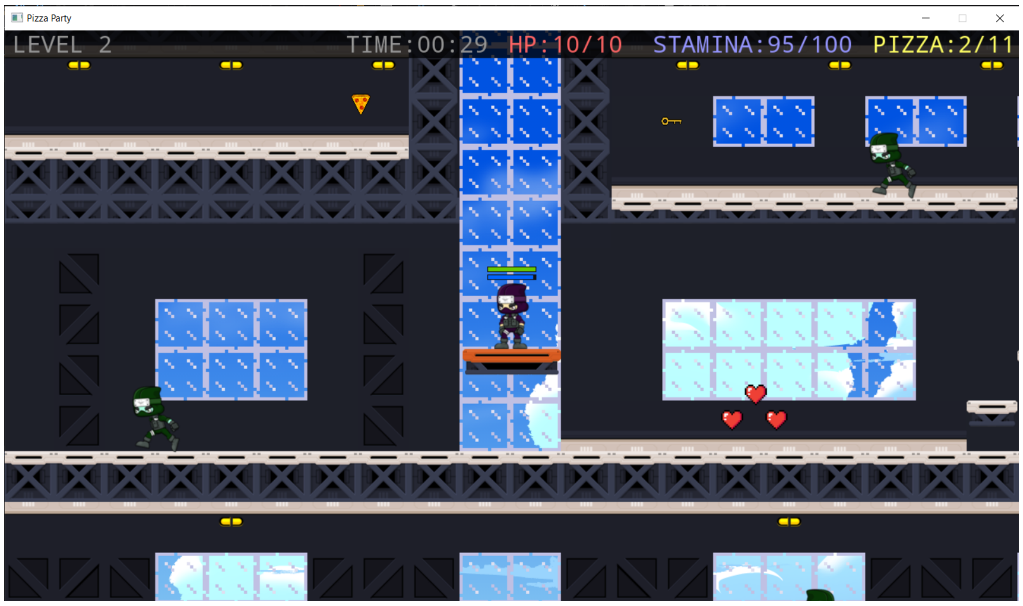


Figure 10: Level 2

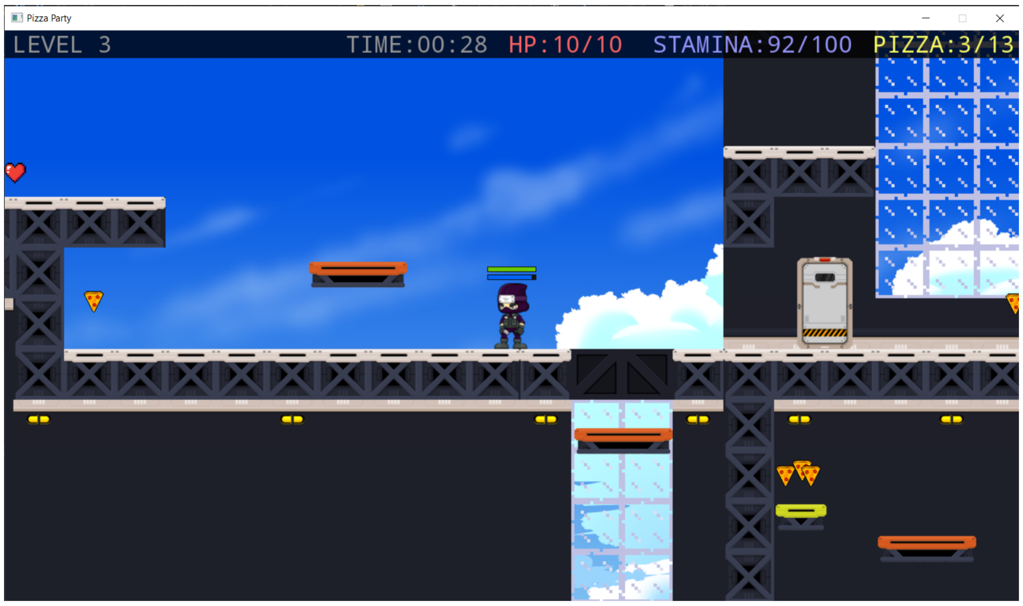
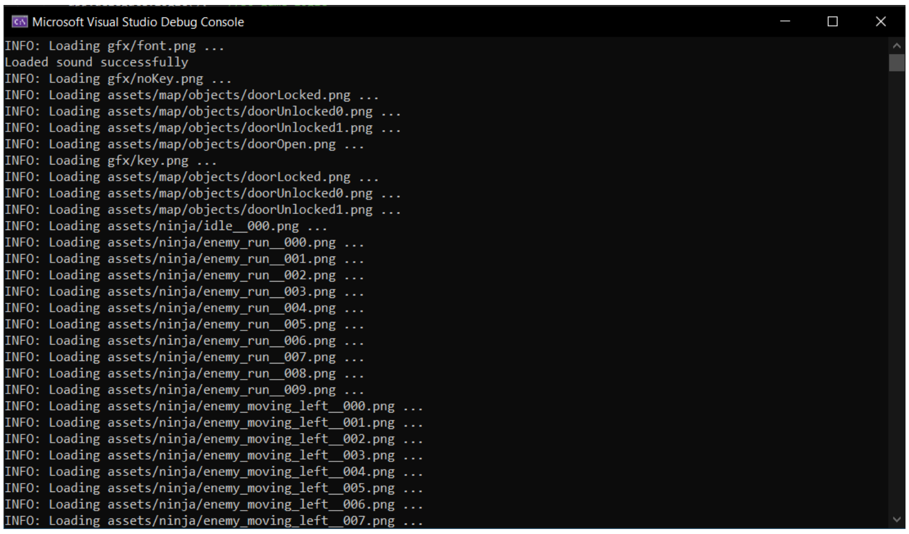


Figure 11: Level 3

3.3 Testing and Troubleshooting

Throughout the development, several issues were faced that needed to be troubleshooted and fixed. After the development of each feature or even each unit, testing was made to assure that the implementation achieved the desired functionality. When the development process was finished, another testing was made to ensure that the program and the game function as intended. Developed code was fully commented and clearly explained throughout the development process to make the testing and possible troubleshooting in the future easier.

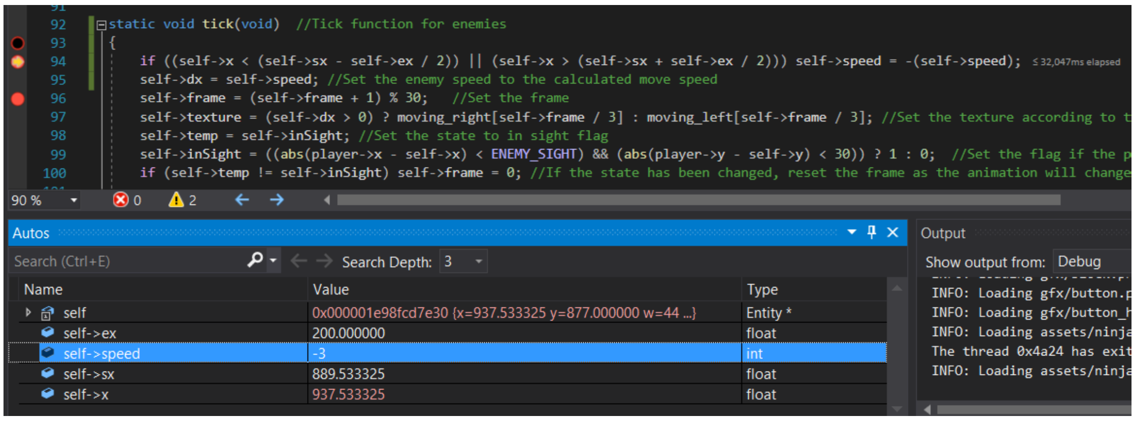
The console was enabled for testing and troubleshooting purposes where the console logs of the SDL library were shown. Moreover, the console provided the option to print out and see some values during run-time, which provided quick understanding of the causes of the issues while testing/troubleshooting. Figure 12 shows the enabled console.



```
INFO: Loading gfx/font.png ...
Loaded sound successfully
INFO: Loading gfx/noKey.png ...
INFO: Loading assets/map/objects/doorLocked.png ...
INFO: Loading assets/map/objects/doorUnlocked0.png ...
INFO: Loading assets/map/objects/doorUnlocked1.png ...
INFO: Loading assets/map/objects/doorOpen.png ...
INFO: Loading gfx/key.png ...
INFO: Loading assets/map/objects/doorLocked.png ...
INFO: Loading assets/map/objects/doorUnlocked0.png ...
INFO: Loading assets/map/objects/doorUnlocked1.png ...
INFO: Loading assets/ninja/idle__000.png ...
INFO: Loading assets/ninja/enemy_run__000.png ...
INFO: Loading assets/ninja/enemy_run__001.png ...
INFO: Loading assets/ninja/enemy_run__002.png ...
INFO: Loading assets/ninja/enemy_run__003.png ...
INFO: Loading assets/ninja/enemy_run__004.png ...
INFO: Loading assets/ninja/enemy_run__005.png ...
INFO: Loading assets/ninja/enemy_run__006.png ...
INFO: Loading assets/ninja/enemy_run__007.png ...
INFO: Loading assets/ninja/enemy_run__008.png ...
INFO: Loading assets/ninja/enemy_run__009.png ...
INFO: Loading assets/ninja/enemy_moving_left__000.png ...
INFO: Loading assets/ninja/enemy_moving_left__001.png ...
INFO: Loading assets/ninja/enemy_moving_left__002.png ...
INFO: Loading assets/ninja/enemy_moving_left__003.png ...
INFO: Loading assets/ninja/enemy_moving_left__004.png ...
INFO: Loading assets/ninja/enemy_moving_left__005.png ...
INFO: Loading assets/ninja/enemy_moving_left__006.png ...
INFO: Loading assets/ninja/enemy_moving_left__007.png ...
```

Figure 12: Console Enabled

During the testing of the enemy feature, where test runs made on the game, an issue was noted with the enemy movements, rarely resulting in an enemy being stuck going back and forth without changing its position. To resolve this issue, breakpoints - a debugging tool in Visual Studio - were used to pinpoint the piece of code where the problem is occurring and halt the execution of the program to check the run-time values at that moment. Figure 13 shows how this tool was utilised to find the source of the issue.



```
static void tick(void)  //Tick function for enemies
{
    if ((self->x < (self->sx - self->ex / 2)) || (self->x > (self->sx + self->ex / 2))) self->speed = -(self->speed);  ≤32,047ms elapsed
    self->dx = self->speed; //Set the enemy speed to the calculated move speed
    self->frame = (self->frame + 1) % 30;   //Set the frame
    self->texture = (self->dx > 0) ? moving_right[self->frame / 3] : moving_left[self->frame / 3]; //Set the texture according to t
    self->temp = self->inSight; //Set the state to in sight flag
    self->inSight = ((abs(player->x - self->x) < ENEMY_SIGHT) && (abs(player->y - self->y) < 30)) ? 1 : 0;  //Set the flag if the p
    if (self->temp != self->inSight) self->frame = 0; //If the state has been changed, reset the frame as the animation will change
```

Autos — Search Depth: 3

Name	Value	Type
self	0x000001e98fcd7e30 {x=937.533325 y=877.000000 w=44 ...}	Entity *
self->ex	200.000000	float
self->speed	-3	int
self->sx	889.533325	float
self->x	937.533325	float

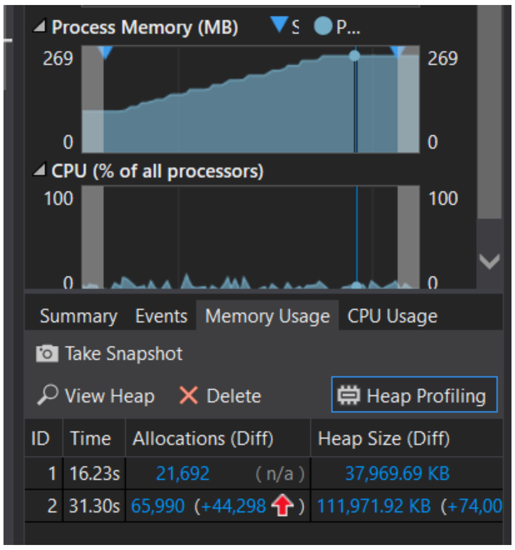
Output — Show output from: Debug

```
INFO: Loading gfx/button.p
INFO: Loading gfx/button_h
INFO: Loading assets/ninja
The thread 0x4a24 has exit
INFO: Loading assets/ninja
```

Figure 13: Breakpoints Utilised

This method made apparent that making the enemy speed negative of what it was when it reaches its range caused some problems. Therefore, the code was re-written to make the enemy speed negative of its move speed when it reaches its limit on the right and positive of its move speed when it reaches its limit on left.

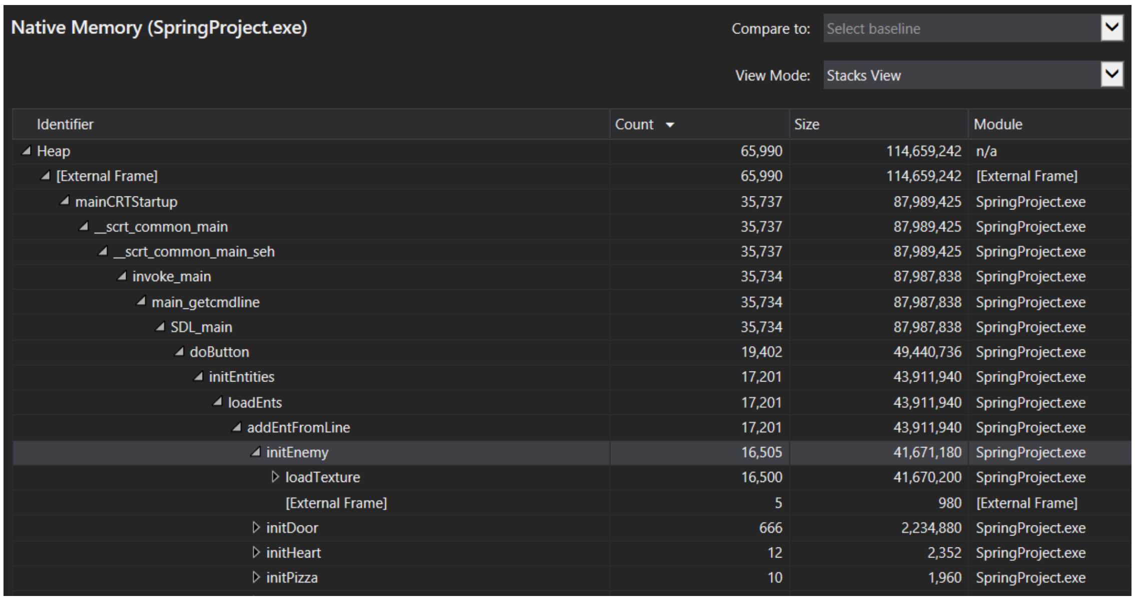
For testing purposes, memory and the CPU load were monitored each test run after each feature implementation. When the menu and level features were successfully implemented, changing the level and loading the menus on the test runs caused an increase in the memory. As Figure 14 shows, each time a level was loaded, the memory usage of the program went up around 13 MB, which might not seem that much but when it is stacked, it can reach to some undesired amounts quickly. Moreover, this is a memory leak which means that the unused memory is not being freed, and that is a bad practice.



ID	Time	Allocations (Diff)		Heap Size (Diff)
1	16.23s	21,692	(n/a)	37,969.69 KB
2	31.30s	65,990	(+44,298)	111,971.92 KB (+74,00

Figure 14: Memory and CPU usage

To tackle this problem, another Visual Studio tool - heap profiling - was enabled and snapshots of the memory was taken before and after the levels were loaded multiple times. As this can be seen in Figure 14, around 112 MB of difference were reached around 15 seconds. This heap profiling tool allows us to identify the function calls that used the memory between the snapshots. As Figure 15 shows, considerable amount of the memory was used by *initEnemy* function calls.


Native Memory (SpringProject.exe)

Compare to: Select baseline

View Mode: Stacks View

Identifier	Count	Size	Module
Heap	65,990	114,659,242	n/a
[External Frame]	65,990	114,659,242	[External Frame]
mainCRTStartup	35,737	87,989,425	SpringProject.exe
__scrt_common_main	35,737	87,989,425	SpringProject.exe
__scrt_common_main_seh	35,737	87,989,425	SpringProject.exe
invoke_main	35,734	87,987,838	SpringProject.exe
main_getcmdline	35,734	87,987,838	SpringProject.exe
SDL_main	35,734	87,987,838	SpringProject.exe
doButton	19,402	49,440,736	SpringProject.exe
initEntities	17,201	43,911,940	SpringProject.exe
loadEnts	17,201	43,911,940	SpringProject.exe
addEntFromLine	17,201	43,911,940	SpringProject.exe
initEnemy	16,505	41,671,180	SpringProject.exe
loadTexture	16,500	41,670,200	SpringProject.exe
[External Frame]	5	980	[External Frame]
initDoor	666	2,234,880	SpringProject.exe
initHeart	12	2,352	SpringProject.exe
initPizza	10	1,960	SpringProject.exe

Figure 15: Heap profiling

The other functions that used the memory included some of the texture loading functions, and the memory used by *initEnemy* was also caused by loading textures again every time a level was loaded, without freeing the memory that was already used. Therefore, to tackle this issue, it was decided to load the textures needed on initialising the program, and using these textures loaded on initialising entities like the enemy or on other functions that load textures when a level is initialised. In this way, the textures are loaded only once and not stacked on the memory. When implemented, this approached worked more than fine and brought the memory usage down by a lot while fixing the issue with loading levels.

When the development of the game was done, final tests were ran to ensure the final product works smooth without problems and as intended. Also, the if statements and the for/while loops used were checked, as well as the places that take user input, to see if the program was as robust as possible. When all the altering after this was done, the testing process was finalised.

4 Conclusion

To summarise the experience of building a game in C and using the SDL2 library, it was quite interesting to learn how the games like this were developed and it was fun to tackle the common problems by trying out different solutions. It was learned that how these type of games are developed, what kind of process is taken in the development, importance of the order of implementing the features, and, as developing such game for the first time, what the common mistakes made are and how to tackle them using the tools designed for them.

Initial ideas of thinking that building a platform game using a low-level language like C was replaced with ideas like thinking a language like this suits better for making a robust game and easily adding new features to it. The reason for this change in the idea was seeing how easily a feature like the enemy, that used to seem like a very complicated feature, can be implemented by using very basic methods and not so complicated programming.

If the game were to be developed again, some of the things to change would include spending more time on implementing variety of new features than spending time on replacing/adding more textures/animations. Moreover, if more time was provided, it would be spent on adding features such as adding multiplayer and new weapons. However, all the requirements/specifications that were listed before starting the project were met.

Although the features like adding hearts and sprinting were quite easier to implement than expected, features like doors and kunai were a bit harder to implement than expected as there are more details to take care of and as they tend to cause more issues.

Altogether, it was fun to build and develop this project to learn how to actually utilise newly learned aspects of programming. The amount of time given seemed reasonable. Support was available throughout the development. And experiencing programming issues and coming up with solutions increased my confidence in programming.

References

Ninja Adventure - Free Sprite (2022), `https://www.gameart2d.com/ninja-adventure---free-sprites.html`. [Online; accessed 10-March-2022].

Realities, P. (2022), 'SDL2 Game Tutorials', `https://www.parallelrealities.co.uk/tutorials/`. [Online; accessed 5-March-2022].