A tutorial on the use of SplashKit to develop a game

Introduction

When it comes to creating an immersive digital experience, game creation stands out as an exciting and dynamic subject within the field of software development. This report explores the development of a 2D space-themed game utilizing the Splashkit toolkit, a flexible framework that enables game creators to realize their ambitions for video games. The goal of this project was to create, design, and construct a game that included key game development principles, including User Interaction, Object Management, Collision Detection, and Dynamic Gameplay. This report will take you on a tour through the many parts of the game's source code, investigate the gameplay mechanics, and provide you with insights into how an interactive and compelling gaming experience is created. Let's start this investigation into game production by breaking down the components that go into building an enthralling virtual world.

For task 9.3HD, I have chosen to do the tutorial on the Splashkit to accomplish a task. My main tutorial will be based on the tasks related to a space game. This game is about a player collecting coins and dodging planets. The coins will be available in random places and the planets will keep coming in pairs as obstacles from the right side to the left side of the screen. The coins will also have an animation where the coins spin to improve the visual of the game. Every time the player hits the coin the score increases, and when the player hits the planet, the score reduces. when the planet hits the coin there is no change taking place except that the coin disappears without affecting the score. Additionally, there will be a crashing sound effect when the player collides with the planet to increase user interaction. Let's go through all the codes related to the function of this game to get a better understanding of how this works.

The player

First, we need to create a new project and include the necessary headers and libraries. Define constant, enumeration, and structure for the player. **#Include<cstdint>, #Include<string >, #Include" Splashkit.h"**, and **#include" planet_play.h".** the **planet_play** header has all the structures and enumerations needed for this game; **enum ship_kind, struct player_data, struct coin_data, enum planet_kind, struct planet_data, struct space_game_data**. The structures and Enums for the coin and planet will be discussed below. Now let's see how to implement and draw the player onto the screen and handle inputs for the player so that the player can move around the screen.

Before that, gather all the planet-related functions and procedures in a separate header file called **planet. h.**

Then, create a player.cpp file and inside it we need to create a function to draw the player, that is the **bitmap ship_bitmap** (). This function takes a parameter called kind from the ship_kind struct. This returns a specific bitmap based on the switch case value given in. In the switch case repetition, the values for the ships are given as constants from the **ship_kind enum.** Each ship is given a constant number so that the user can give input to change the player.

Then to draw the ship on the screen we need to create a function called **player_data new_player ().** This initializes the player_data structure with a result variable and sets the player's kind according to the given bitmap and initializes the bitmap's coordinates of x and y. It then returns the initialized player_data.

Then, the void **player_data ()** procedure will be created. This will take the parameter a constant player_data struct so that it won't be modified inside the procedure. Then we can name a variable called **player_to_draw**.

The main responsibility of this procedure is to draw the bitmap on the screen. This procedure assigns the appropriate bitmap from the ship_bitmap using the player_to_draw variable at the specified coordinate of player_to_draw.x and player_to_draw.y. This type of coding loads, draws the bitmap, and returns specific objects based on some criteria. This design keeps the code organized and makes it easier to update the game.

Finally, create a procedure **handle_input ().** This takes the parameter of the player from the player_data struct. This procedure contains the if statements for the player movements and player alteration. When the number keys between 1 and 3 are pressed, each ship will have its number so whenever a number between 1 and 3 are pressed, the player changes. When the arrow keys are pressed the user can move the player anywhere on the screen. For this, a constant is used to control the speed, **PLAYER_SPEED**. It is assigned as 7 in the planet_play header file as mentioned earlier. Each player's movement is controlled by a set of codes. A player will have its starting point on the screen by assigning an initial coordinate for x and y. To move the player, we only have to alter the x and y positions. So, every time the UP arrow is pressed, the y coordinate of the player will be reduced by the constant, PLAYER_SPEED. When the DOWN key is pressed the y coordinate will be increased by the constant, to move left the x coordinate will be reduced by the constant, and to move right, the x coordinate will be added to the PLAYER_SPEED.

Now, if we compile and run the code, we will be able to draw a ship that can be moved across the screen whenever the arrow keys are pressed.

The coins

To continue further we will now need to increase the complexity of the game by adding another element to the game, the coin(s). These coins will be spinning at random places on the screen. Every time the user hits a coin, the coin is collected and appears back in a random place and the score will increase by one. But for now, we will look at drawing the coin on the screen and later on in the report we will discuss collision and score.

First, we need to declare a struct in the **planet_play.h** file as **coin_data.** This will contain the coin_animation for the coin to spin and give a visual appeal and the x and y coordinates for the coin's position. Then create a separate file for the coin, **coin.cpp**.

Inside the coin.cpp file, create a function called **coin_data new_coin ().** This creates a new coin object and initializes its x and y coordinates by taking x and y as parameters for the function and the coin animation for the spinning. It creates a coin_data struct called result. Using this, it assigns the x and y coordinates for the coin on the screen. The coin_animation will create animation from the resources given named "coins" with the "spin" animation script. Finally, it returns the result variable to create the coin.

Now the coin is initialized, it is time to draw it on screen. For that, we use the overload function **coin_data new_coil ()** without any parameters. It loads the bitmap "gold_coin" using bitmap_named. Then calculates the random coordinate of the coin within the screen's width and height by calling the new_coin which was written earlier to calculate the x and y coordinate of the coin. This is then returned. It now draws the coin in random places on the screen.

Now to draw the coin with the animation, we create a procedure called **void draw_coin ().** It takes a constant coin_data struct by reference as a parameter. Inside the procedure, we call the function draw_bitmap () with the parameters of bitmap named "gold_coin", the x and y coordinates, and the coin animation to provide the animation effect whenever the coin is drawn on the screen.

Finally, to update the coin, that is, to provide a spinning effect when drawn to the screen, we create a procedure called **update_coin ()** with the parameters of coin_data struct passed by reference with a variable called coin. This is called typically in the game's update loop to advance the animation frames.

In summary, this part of the coin's code is solely responsible for creating, drawing, and updating the coin object. We can give either specific or random coordinates to draw them.

After implementing all these codes will now allow us to draw both the player and coins to the screen. Once we compile and run the code, we will see a ship and coin, where we can move the ship around the screen by pressing the arrow keys, changing the player according to the numbers pressed, and coins spinning around in random places. But we cannot still collect the coin and increase the score. For that, we will look at some more procedures and functions to achieve that.

To accomplish it, we need to create a new file called space_game.cpp. This file will have libraries such as #include "splash kit. h", "planet_play.h", <cstdint>, <string>, <cmath>, and all the vital functions and procedures to draw multiple coins and be able to collect them and increment the score.

First, we need to create a **bool player_hit_coin ()** function inside the player.cpp file. This takes parameters from constant player_data and coin_data structures passed by references with variables called player and coin respectively. It has two bitmaps of coin_bmp **(bitmap_named("gold_coin"))**, and ship_bmp(**ship_bitmap(player.kind**)). coin_cell = animation_current_cell(), this stores the current frame of the coin animation in use. Finally, we return a function called **bitmap_collision()** which has parameters of coin_bmp, coin_cell, and coin.x, coin.y, ship_bmp, 0, player.x, and player.y. the coin_bmp represents the coin image, the coin_cell represents the coin animation, coin.x, and coin.y represents their coordinates as to where they will appear on the screen, the ship_bmp represents the ship image, and the 0 argument represents the cell of animation for the player. Since there is no animation involved for the ship, the value is 0. Then at last there are the coordinates of the ship, the player. x and player. y which decides where will the bitmap appear on the screen. The return function checks whether the ship bitmap and coin bitmap collide with each other at their respective positions. It returns true when there is a collision detected and returns false when no collision is detected.

Before that, we will have to declare a few vectors in the **planet_play.h** file, we create a struct called **space_game_data**. This contains all the variables such as **player_data player**, **vector<coin_data> coins,** and **int score_counter = 0**. Now coming back to space_game.cpp, let's write the codes that make sure the coins are collected and the score is incremented.

First, we create a procedure called **remove_coin()** with vector coin_data which passes the variable coins by reference and int idx. This function frees up memory used by the animation so that it is not used up by all the animations. Idx is the specified place index where the memory should be cleaned up. Then the **coins.pop_back()** will remove the coin and replace it with the next bitmap that is about to come. This helps maintain the order of the coins in the vector after removal.

Afterward, we create a procedure called **void update_coins().** This takes parameters such as vector<coin_data>, constant player_data, and space_game_data which are all passed by references with variables called coins, player, and game respectively. This procedure is responsible for updating the state of the coins in the game. It iterates all over the coins, updates their animation, and checks whether any collision is detected.

Inside the procedure, the **player_hit_coin** is called to find the collision between the player and the coins. As soon as a collision is detected, the **game.score_counter** increments the score by 1 and marks the coin so that it can be removed once it is collected. Then there is a for loop which calls the **remove_coin** function which takes

two parameters, **coins,** and to_**remove[i].** This will be assigned the coin which should be removed with the current index value **i** that it represents.

Now let's create a function called **add_random_coin()** which takes **space_game_data** as a parameter and passes by reference the variable called game. This adds a random coin to the end of the vector list using the **push_back()** function which uses **new_coin** as a parameter to indicate that a random new coin is assigned to the end list of the coin vector. Since this is a procedure, it doesn't return anything.

Then, let's create a procedure called void **update_game()** which takes the space_game_data as a parameter and passes by reference the variable game. This calls the update_coins() to update the overall game state. Inside this function keyword "**rnd**" is used inside an if condition which checks whether the rate at which the coins are appearing on the screen is less than 0.03. then inside the condition, the **add_random_coin** is called to add coins randomly to the end of the list in the coin vector so that coins start to appear randomly all over the screen at a specific rate.

Finally, we create a **void draw_game()** which takes the parameter of space_game_data that passes by reference of a variable called game. This will first clear the screen and draw the player using the function **draw_player()** with a parameter of **game.player**. Then a function called **draw_text()** is used. This outputs the score visually on the screen for the user to view it. Then refresh the screen for the next bitmap to be drawn.

If we compile and run this code, we will be able to move the ship, collect coins, and increase the score every time a coin is collected. The collected coin will disappear and the new coins will be randomly generated.

This is not the end, we still need to add planets to the screen, make them move across the screen, and collide with the player and coin, each time they collide with a coin, the coin should disappear without affecting the score. Every time the player collides with the planet, the score should decrease by 1, and last but not least add a small sound effect whenever the planet collides with the player.

The planet

First, we need to draw and be able to change them whenever letters are assigned for them are pressed. In the planet_play.h file let's assign structs and Enums for the planet-related functions. Assign a struct called **planet_data** with a variable named kind and the x and y coordinates of the planet to where they appear on the screen. Then set the Enums called planet_kind. There we name all the available planets as constants. In the struct space_game_data which was mentioned earlier, will now have another vector array assigned for planets as well, **vector<planet_data> planets**.

We should create a planet.cpp file to write the main functions and procedures. Inside this file, we create a **planet_data new_planet()** function which will return the type of planet to be drawn, and their coordinates on the screen. The x coordinate is given a constant value because if it changes then the planet wouldn't move horizontally. Then we need to load the bitmaps using a **bitmap planet_bitmap()** which has a planet_kind kind parameter. This uses switch-case conditions to load each bitmap to the appropriate constants given in the planet_kind Enum. Then when the user is playing the game, they should be able to change the planet type similar to changing the ship type.

To draw planets randomly, we create a function called **planet_kind random_planet_kind()** which returns a static cast of all the planet kinds Enums available. The numbers are assigned from 0 to 7 by default in the Enums so the keyword rnd(8) is used to indicate them all.

So, we create a procedure called **void handle_planet_input()** and **void handle_planet2_input()** which takes planet_data as a parameter and passes the variable planet by reference. These procedures will use the if conditions to state which keys are assigned for which planets so that the user can press them to change the planet type. We could have used only one procedure to handle these inputs, but since we need to draw at least two planets to the screen, we use two procedures to differentiate between them and give inputs accordingly.

Then we create a procedure called **void update_planet()** and **void update_planet2()** which takes the planet_data as parameters and passes a variable called planet and planet2 respectively by reference. This function determines at what pace the planet should be moving and from where to where. It also calls the random_planet_kind() function so that random planets are drawn. The planet.x determines where the planet should start coming from, from the edge of the screen, a constant is initialized for this in the planet_play.h called **PLANET_SPACE**.

Finally, we create a procedure **void draw_planet()** which is responsible for loading and drawing the bitmap on the screen for us to see and play with. This takes a parameter from **planet_data** which is passed by reference with a variable called **planet_to_draw.** This initializes the bitmap again and calls the function planet_bitmap to draw the kind of bitmap that needs to be drawn along with the coordinates at which it needs to be and traveling to by calling the function **draw_bitmap()** with all those parameters.

Now to run them we should write the main codes in a separate file called program.cpp. in the main method, using functions like **open_window()** with the name and size of the window given as parameters, **load_resources()** will load the bitmaps from the supplied resources in the game file, **handle_input()** with a parameter of **game. player** which makes the player to have an input, **draw_game(),** and **update_game()** with a parameter called **game** which draws and updates the overall game when compiled and run.

With all these codes written, it is enough to draw the planet and move it from the left to right side of the screen. But this will not impact the score when colliding with the player or coins. Once the planet goes off the screen there won't be any other planets coming from the left side as an infinite loop of planets coming as an obstacle. To accomplish all that we must add some more blocks of codes to the files.

First, we shall create a procedure called void add_random_planet(). This takes space_game_data as a parameter and passes the variable game by reference. This function would add a random planet to the end of the list of planets by using the push_back() function so that when a planet disappears the next planet will be random and this function is responsible for that.

After that, let us write the code for the infinite loop of planets coming off the screen once the previous ones disappear. In the new_game() function in space_game.cpp, we need to now call the add_random_planet() with the result as a parameter to make sure random planets are called. Next, we have to add another for loop inside the **draw_game** function in the same file. This loop will draw the list of planets available in the planets index and increment it by 1 every time the condition is met. The **draw_planet()** function is called inside the loop. The parameter will be the indexes of planets to be drawn.

Now, go back to the update_game() function. There we write code to detect the planet has gone off the screen to the left, if it is true then a new planet is drawn on the right and starts moving across the screen to the left again. First, a Boolean function is created, **all_planets_off_screen = true**. A for loop is added, the loop uses a range-based for loop where the planet represents the planets in the vector one by one. Then the if condition inside the loop checks whether the planet has gone off the screen by examining the planet.x, the coordinate at which the planet is on the x-axis, and the width of the planet, planet_bitmap. If the sum of planet_width and

planet.x is greater than 0, that means that the part of the planet is still visible inside the screen and returns false and the condition will be the same until the condition is met and returns true.

Right after it returns true, there is another if condition that checks this, if it is true, the block of code inside the condition adds new planets to the game. The first line **int num_new_planets = rnd(3) +1;** will generate planets between 0 and 2 and then adds 1 to it. This results in a random number between 1 and 3. The for loop adds one planet for every iteration. The add_random_planet() function is called to generate planets. After adding a new planet, the game.planets.back().x function makes sure to set the starting position on the right side of the screen.

In summary, this code makes sure that once the planet moves off the screen, the next line of planets of 2 or 3 should start off the right side of the screen.

Finally, in the program.cpp, write a for loop with both handle inputs for the planets. handle_planet_input() and handle_planet2_input() are the functions that are called so that the user can press different keys to change the planet type. Now if we compile and run this code, we will be able to see a couple of planets moving from the right side to the left side of the screen. But this still doesn't affect the score by any means. For that, we will still have to write a few more codes.

Collisions

First, let us write the code to detect a collision between the planet and the coin. For that, we need to write a Boolean function called **bool planet_hit_coin()** which takes parameters of constant planet_data and coin_data which passes the variables planet and coin by reference respectively. It calculates whether the planet and coin bitmap's radii are less than the combined radii. If it is less than that, it means they are overlapping and indicates a collision. Inside this function, we give the planet_radius and coin_radius. Then write an equation to calculate the horizontal (x) and vertical distance (y) between the planet (planet.x) and the coin (coin.x). It calculates the Euclidean distance between the planet and the coin using the Pythagoras theorem. Finally, it checks and returns whether the distance is less than the sum of planet_radius and coin_radius.

Now to execute that, inside the procedure **update_planets()** write another for loop to iterate through all the coins in the game and check whether a collision took place with the current planet. The index j is set to 0, the loop continues as long as j is less than the number of coins in the game. Inside the loop, the **planet_hit_coin()**function is called to check the collision between the coin and planet and returns true if the collision is detected. Then to remove the coin when the planet collides, the to_remove.push_back(j) is used, this function will keep track of which coin should be removed from the game once they have collided with the planet.

Now if we compile and run this, the planet will move from right to left, and the coins that come in the path of the planets will disappear without affecting the score.

Up to now, the game is working fine but we still need to implement one other thing, which is the collision between the planet and the player. For that, let us create another Boolean function called **bool player_hit_planet().** This function takes constant player_data and constant planet_data and passes the variable player and planet respectively by reference. This function also behaves in a similar way as the planet_hit_coin(). This first assigns the player and planet radii, then calculates the horizontal (x) and vertical (y) distances and uses Pythagoras theorem to find the Euclidean distance between them according to their current x and y coordinates. Finally, it checks and returns whether the distance is less than the sum of player_radius and planet_radius.

This function will be again called inside the **update_planets()** procedure. Inside the function, a for loop is used to iterate through all the planets. the index "i" is set to 0 and loops as long as "I" is less than the number of

planets in the planets vector. Inside this loop, we call the update_planet() function to update the properties of the planet and its current position. This typically involves moving the planet on the screen, advancing its animation, and performing other updates related to planets. in the if condition used, it checks whether the planet is colliding with the player. For that the player_hit_planet() function is called with the parameter of player and planet[i] indexes. If a collision is detected it reduces the score by 1, **game.score_counter -= 1;.**

As mentioned earlier there should be a sound effect played whenever the planet collides with the player to increase the interaction with the game. for that, we must first download an appropriate audio file in .wav format and save it inside the folder where the game files are saved. Give it a short name. Then in the space_game.cpp file, create a Boolean function called bool is_sound_effect_playing() with the parameter of a constant string data type with reference to a variable called sound_effect_name. Inside this function, the **sound_effect_playing()** with the name of the audio file will be returned.

If we call this Boolean inside the **player_hit_planet()** if condition. Write another if condition in it. This must check if the sound effect is playing when the planet and player collide, if not, it makes sure to play the sound called "crash.wav". This ensures that this sound is played once when the ship collides with the planet. So, this whole update_game function is responsible for updating each planet's state and checking for the collisions between them. It plays a sound effect and decreases the score.

Finally, if we compile and run this code, we can see a ship moving everywhere I the screen whenever an arrow key is pressed, plus it also changes in different types. The many coins will appear randomly and the ship collects it to increase the score. Then planets from the right side keep moving across to the left side, while moving if it overlaps with a coin, the coin disappears, if it overlaps with the player, it reduces the score and plays a sound effect to denote collision. Finally, when the planets disappear off the screen on the left side, another couple of planets are generated from the same starting point. All these together make perfect game programming done in C++ coding.

Conclusion

In this game development task, we have successfully implemented a 2D space-themed game using the Splashkit library. This game involves the user navigating a spaceship through infinitely approaching planets and collecting coins to earn points. While developing this game we had to come across several key aspects of game development, such as Object Management, Collision Detection, User Interaction, Visual Feedback, Game Logic, and Dynamic Gameplay.

In summary, this game development project successfully involves the basic game development concepts with the use of the Splashkit library to create attractive and interactive gameplay. This project serves as a solid starting point for future game development efforts since it shows the possibility for growth and improvement.