Game Programming and System Architectures (CMP208) Report

By Mateusz Zaremba

Table of Contents

Introduction 2
Application Design 3
- Enumerated types 3
- Game state machine 3
- Game state machine in pseudocode 4
- Gameplay structure 6
- Camera structure 6
- In-game menu 6

Techniques Used 7
- Procedural level generation 7
- Collision 8
- Pickups 9
- Input 10
- Pause button 10
- Menu 10
- Menu-box-sprite positioning and menu-text positioning 11
- Menu-box-sprite positioning 11
- Menu-text positioning 12
- Sound 12

User Guide 12
Critical Analysis 13
Conclusions 13
References 14

Introduction

The purpose of the application is to create a 2.5D game for PlayStation Vita which utilizes features of the Box2D physics engine and the Games Education Framework.

To fulfil this specification, the application was inspired by a mobile game called ‘Chameleon Run’ made by Noodlecake Studios Inc. In this game, the player controls a dynamic, colour swapping ball during a single run with an option to restart the game after the run is over; the runs over condition can be triggered by restarting the game from the pause menu or by a loss or a win condition.

The colour swapping of the ball is between red and blue. The ball moves on its own in a positive ‘x’ direction. The player can control the ball’s jumping and colour swapping using PSVita’s button or touch controls. The game’s level consists of platforms in red, blue, brown and gold colour. Each blue and red platform has a green pickup in the middle of it.

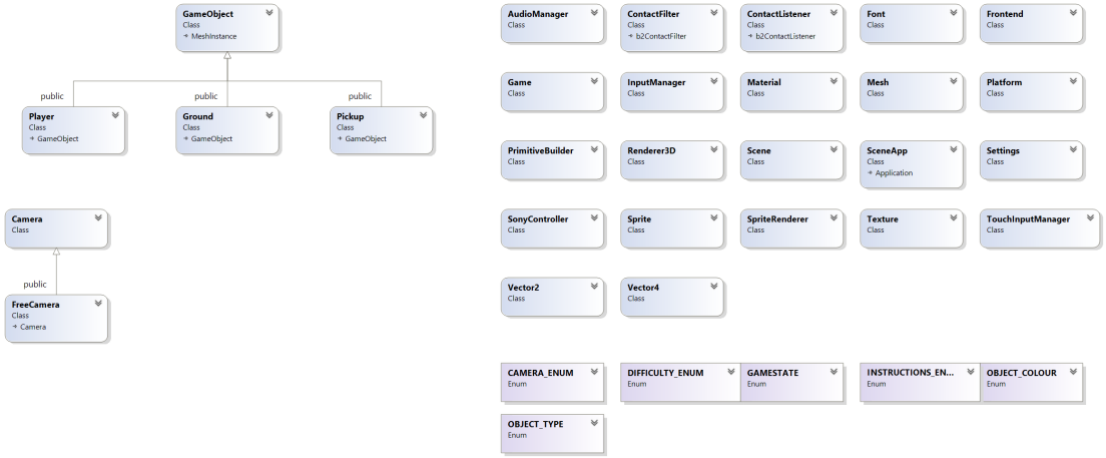
The premise of the game is to change the ball’s colour while airborne to match the platforms’s colour while collecting the pickups. Touching a brown platform, failing to match the ball’s colour with the platform’s colour or falling into the nether regions results in a lost round. To win the round the player must get to the end of the level and touch the gold platform. During the run, the player can pause the game. While the pause menu is active (or after a lost round when the pause menu activates itself automatically) the player can freely move the camera around the level. After the pause menu is turned off, the camera lerps back to its previous position. The player also collects pickups by colliding with them. The number of pickups collected is displayed in the left upper corner and reset before each new run.

Before starting the game, the player is presented with a frontend menu from which he can: start the game, go to the settings, read the instructions about the controls and the gameplay, or quit the game. In the settings, the player can choose the camera’s position during the run from 3 available, a difficulty level and a number of platforms in the level (10, 20 or 30), as well as go back to the frontend menu.

Application Design

The application consists of 3 scenes, being separate classes, i.e.: Frontend class, Settings class and Game class, which are governed by a game state machine set up in SceneApp class.

In the SceneApp class an input manager and an audio manager are being created and then passed to appropriate instances of the three above classes, as well as unsigned integers, which are passed in the same way to individual classes, to keep track of the settings chosen for the game, which are: the camera, the difficulty level and number of platforms in the level.



Visual Studio's Class View

Enumerated types

To keep track of the state of entities in the game, a group of enumerated types were created. Each one of them were put into separate files and then included, in appropriate files, when needed. For example, to keep track of the game's state and to use it with the game state machine, an enumerated type called GAMESTATE was created. It consists of: FRONTEND, SETTINGS and GAME. Other enumerated types are: OBJECT_TYPE, DIFFICULTY_ENUM, COLOUR TYPE, CAMERA_ENUM and INSTRUCTIONS_ENUM.

Game state machine

The SceneApp class holds pointers to the Frontend class, the Settings class and the Game class. The game state machine changes its state depending on the GAMESTATE enumerated type value, which is initialized in the SceneApp class and its address is passed to the 3 classes mentioned above, which accept a pointer to the GAMESTATE enumerated type. In the appropriate class, the passed gamestate pointer is assigned to a gamestate pointer in it. By triggering the game state machine's change, the gamestate pointer is dereferenced and assigned a new value. That way its original value in the SceneApp class is being changed.

If the state of the game state machine changes (depending on the current state) an appropriate class is created and all other classes are deleted. The render's function

game state machine holds a single call to the ‘render’ function in individual classes, similarly to the ‘clenup’ function, which holds a single call to the ‘release’ function in individual classes.

Game state machine in pseudocode

SCENEAPP UPDATE FUNCITON
1. IF GAMESTATE IS SET TO FRONTEND
    a. IF FRONTEND POINTER IS NULL
        i. initialize Frontend class
    b. IF GAME POINTER IS NOT NULL
        i. delete instance of the Game calss
    c. IF SETTIGNS POINTER IS NOT NULL
        i. delete instance of the Settings class
    d. UPDATE FRONTEND
    e. IF FRONTEND’S QUIT BOOLEON IS TRUE
        i. return false from SceneApp’s update function

2. IF GAMESTATE IS SET TO SETTINGS
    a. IF SETTIGNS POINTER IS NULL
        i. initialize Settings class
    b. IF FRONTEND POINTER IS NOT NULL
        i. delete instance of the Frontend class
    c. UPDATE SETTINGS
    d. IF SETTING’S QUIT BOOLEON IS TRUE
        i. return false from SceneApp’s update function

3. IF GAMESTATE IS SET TO GAME
    a. IF GAME POINTER IS NULL
        i. Initialize Game class
    b. IF FRONTEND POINTER IS NOT NULL
        i. delete Frontend
    c. UPDATE GAME
    d. IF GAME’S QUIT BOOLEON IS TRUE
        i. return false from SceneApp’s update function
4. RETURN TRUE FROM SCENEAPP’S UPDATE FUNCTION

```
// in the SceneApp class header file
// enumerated type to hold the game’s state
enum GAMESTATE { FRONTEND, SETTINGS, GAME }
// initialize GAMESTATE enum
GAMESTATE gamestate
// initialize pointer to the frontend class
Frontend* frontend
// initialize pointer to the settings class
Settings* settings
// initialize pointer to the game class
Game* game
```

```
// in the SceneApp class cpp file
// SceneApp update function game state machine
switch (gamestate)
{
	case (FRONTEND)
	{
		if (frontend is null pointer)
			initialize frontend class
		if (game is not a null pointer)
			delete game class
		if (settings is not a null pointer)
			delete settings class

		update frontend

		if (frontend's quit Boolean is true)
			return false
	}


	case (SETTINGS)
	{
		if (settings is null pointer)
			initialize settings class
		if (frontend is not a null pointer)
			delete frontend

		update settings

		if (setting's quit Boolean is true)
			return false
	}

	case (GAME)
	{
		if (game is null pointer)
				initialize game class
		if (frontend is not a null pointer)
			delete frontend

		update game

		if (game's quit Boolean is true)
			return false
	}
}
return true from the SceneApp update function
```

Gameplay structure

Gameplay mechanics has been designed using 3 different classes: Player class, Ground class and Pickup class. They all inherit from the GameObject class. GameObject class has all the logic needed for box2D physics simulation, collision detection and rendering. Then, each individual class that inherits from it, has farther functionality defined in it, like e.g.: initialization functions.
The game continues until the winning or the losing condition is triggered. If one of the condition is triggered, an appropriate text and menu pops up. If the player loses, a "YOU LOSE!" text is displayed with a menu allowing the player to restart the game or to go back to the frontend. The same menu pops up when the player wins, only the text displayed is "YOU WIN!". The same structure and mechanics are used for displaying a pause menu but in this case a "RESUME" button is present, allowing the player to resume the game.

Camera structure

The application also has its own virtual Camera class, which then a FreeCamera class inherits from; the virtual Camera class was intended to serve for easy future improvements. For example, if the game was to be changed to have two different cameras, one with first person perspective, and the other one with third person perspective, it could be easily done by creating a FirstPersonPerspectiveCamera class and ThirdPersonPerspective class, then both could inherit from the virtual Camera class. In the main game loop it would be possible to switch between them using polymorphism - a single pointer to the Camera could be reassigned to point to either FirstPersonPerspectiveCamera object or FirstPersonPerspectiveCamera object. Then a single call from the Camera's class pointer to a virtual function, e.g. CameraController, would be enough to limit or expand camera's controls. This solution is being used in this application. A pointer to the Camera class is being created in the Game class. Then it is initialized to point to the the FreeCamera class object, which has only one function defined in it – CameraController.

In-game menu

The game keeps track of its state. A flag is set to true when the pause button has been pressed and when the player lost or won the game. In the render function, "YOU LOST" text will not be displayed unless player's 'alive' flag is false. "YOU WIN" text will be displayed if player's 'win' flag is true. The menu box will not accept any input and menu-text will not be displayed unless 'pause' flag is true.

Techniques Used

Procedural level generation

The level initialization function handles settings up the platforms on different heights, placing pickups and setting up a starting colour.
The starting colour is chosen randomly each time the level is loaded (including restarting the game after a win or a loss). The function holds local variables for: an interval between platforms, a pickup radius, a length of the coloured platform, a length of a textured platform and a starting position vector in world coordinates.

The procedural generation uses a 'for loop'. It iterates as many times as the maximum number of platforms, chosen by the player in the settings. Firstly, a new Ground is pushed back to the vector of pointers of type 'Ground' and a new Pickup is push back to the vector of pointers of type 'Pickup' (To not confuse with GEF's 'Platform' class, a class representing platforms is called 'Ground').
A modulus operation on the current number of iteration is performed to decide what type of the platform should be initialized (Which also depends on the randomly chosen starting colour). If iteration modulus two equals zero (iteration % 2 == 0) a platform with the first colour and a pickup is initialized. For a textured platform without a pickup an iteration modulus three must be equal to zero (iteration % 3 == 0). If the iteration does not fit any of the previous criterion a second colour platform and a pickup is initialized.
Apart from calling the platform's and the pickup's initialization function, a few other actions are performed as well:

1. Change of the platform's height:
    a. In the first colour platform:
        i. If iteration modulus four is equal to zero - add two to the 'y' coordinate of the starting-position-vector.
        ii. Else – set the 'y' coordinate of the starting-position-vector to zero.
    b. In the second colour platform:
        i. If iteration modulus six is equal to zero - add two to the 'y' coordinate of the starting-position-vector.
        ii. Else – set the 'y' coordinate of the starting-position-vector to zero.
2. Create a local pickup-start-position vector to hold pickups starting position and add one to its 'y' coordinate.
3. Check if a pickup model was loaded in. An initialization pickup function is overloaded to be able to either use the mesh of the loaded model for pickup rendering or to use a primitive builder to render a pickup sphere:
    a. If a pickup model was loaded in – use an appropriate pickup initialization function.
    b. If a pickup model was not loaded in - use an appropriate pickup initialization function.
4. Check for the starting colour:
    a. If the starting colour variable is equal to zero – initialize a blue platform.
    b. Else – initialize a red platform.
5. Update the starting-position- vector:

a. Add a length of the coloured platform and an interval value to its ‘x’ coordinate.

Before each iteration, a quick check-up for the last iteration is performed. If the loop is in its last iteration state, a platform with a ‘FINISH’ colour is initialized. Checking it before each iteration improves performance and allows for an accurate initialization each time.
To improve the gameplay experience and to not put too much pressure on the player when the game starts, the player’s colour is being set to match the colour of the first platform in the level.

Collision

In the application, the collision detection uses a box2D ‘contact listener’ class. It has a ‘BeginContact’ and an ‘EndContact virtual functions which accept a ‘b2Contact’ pointer. A ‘ContactListener’ class was created to overwrite those virtual functions. A ‘b2Contact’ pointer allows for access to the fixtures of two colliding bodies in the ‘b2World’ and its user data.
In the ‘BeginContact’ function a local ‘Player’ pointer and a local ‘GameObject’ pointer are created and initialized to ‘nullptr’.
The user data of the first colliding body is cast to a void pointer and, knowing that every entity in the game inherits from the ‘GameObject’ class, its type is cast to a ‘GameObject’ pointer.
Then a check-up, to determine which body is the player is performed. If the type of the first body is ‘PLAYER’, its user data is cast to the previously initialized ‘Player’ pointer and the user data of the second body is cast to the previously initialized ‘GameObject’ pointer. The same action is performed if the second body is of type ‘PLAYER’, only in the reversed order.
The collision response checks:

1. if the ‘Player’ pointer is not null.
2. Then it checks if the ‘GameObject’ pointer is not null.
3. If both of those are not null, a type of the ‘GameObject’ is checked.
4. If the type of the ‘GameObject’ is ‘GROUND’:
    a. the player’s ‘start contact’ function is called.
    b. The ‘GameObject’ pointer is cast to ‘Ground’ pointer and saved in ‘current ground’, member variable pointer, of type ‘Ground’.
5. If the type of the ‘GameObject’ is ‘PICKUP’:
    a. A pickup is scheduled for removal by inserting it to the member variable – a set of ‘Pickup’ pointers (more on this here).
6. If player’s colour matches the ground’s colour and the ‘GameObject’ is of type ‘GROUND’
    a. Player’s jumping ability is reset.

In the ‘EndContact’ all the same actions are performed up until the type checking of the ‘GameObject’. If ‘GameObject’ is of type ‘GROUND’, the player’s ‘end contact’ function is called, and the ‘GameObject’ pointer is cast to the previously described ‘current ground’ pointer.

The ‘start contact’ and the ‘end contact’ functions accordingly increase or decrease the value of a ‘number of contacts’ integer. This could be also achieved with

a Boolean variable which could be set to true when the contact starts, and to false when the contact end but this approach would allow only for one contact at a time and the collision response would be triggered only once when the contact starts and ends. With an integer, a number of contacts can be easily tracked and an appropriate collision response can be called while the collision is happening. And this approach proves to be very useful with this application when the player accidently swaps the colour, while moving on the platform, allowing for an appropriate collision response to be called in the Game class.

Pickups

Collecting pickups requires a bit more sophisticated approach then simply deleting the pickup when the player collides with them. Because of the way the 'b2World' physics world is updated it is very dangerous to delete the 'b2Body' while the collision is occurring (A collision callback happens during the physics step, hence removing the physics body while is it being performed can lead to an undefined behaviour and result in the program crashing). A safe way to remove a 'b2Body' from the 'b2World' is to:

1. Schedule it for removal when the collision happens.
2. Remove the pickup's physics body within the b2World from the Game's update function.
    a. Iterate through the set of pickups-scheduled-for removal.
    b. Playing a pickup sound:
        i. Check if audio manager is defined.
        ii. Check if pickups sfx ID is defined.
        iii. Save its ID in a local integer variable.
        iv. Set the volume and pan for the sample.
        v. Set sample's pitch.
        vi. Play the sample.
    c. Create a local pointer to a 'Pickup' object and assign a pickup-scheduled-for-removal to it.
    d. Call 'delete' on the pickup-scheduled-for-removal. This will call a 'Pickup' class destructor which has a b2Body removal defined in it.
    e. The 'Render' function iterates through the vector of pointers to a 'Pickup' object to render the pickups in the scene. To not render the deleted pickup anymore, the pickup-scheduled-for-removal must be removed from it. It is achieved by finding it in the vector (using std::find function and previously defined local pointer to a 'Pickup' object), saving in a local variable (of type iterator) and erasing from the vector.
    f. The pickup counter is increased by one.
3. The set of pickups-scheduled-for-removal is cleared and prepared for the next removal.

The player's and a pickup's physics body should never collide but a BeginContact/EndContact callbacks should still be generated. To achieve this, pickup's fixture is made into a sensor, which allows for a collision response between two fixtures, without their actual physics bodies colliding.

Input

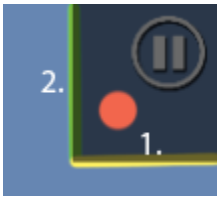
The game can be fully controlled with the Vitas button or touch input. The entities in the game have their own control functions. For button control it's a function that accepts a pointer to the 'SonyController' and for the touch control it's a function that accepts 'gef::Vector2' touch position vector.

Pause button

The pause button's sprite is placed in the right upper corner of the screen ('x' coord. – 935.0, 'y' coord. – 25.0). The application checks:

1. If the touch's 'x' coordinate is bigger than the sprite's 'x' coordinate minus fifty (`touch_position_.x > pause_button_x_ - 50.0f`).
2. If the touch's 'y' coordinate is smaller than the sprite's 'y' coordinate plus fifty (`touch_position_.y < pause_button_y_ + 50.0f`).

This way, in the right upper corner of the screen, a small touch input area is created.



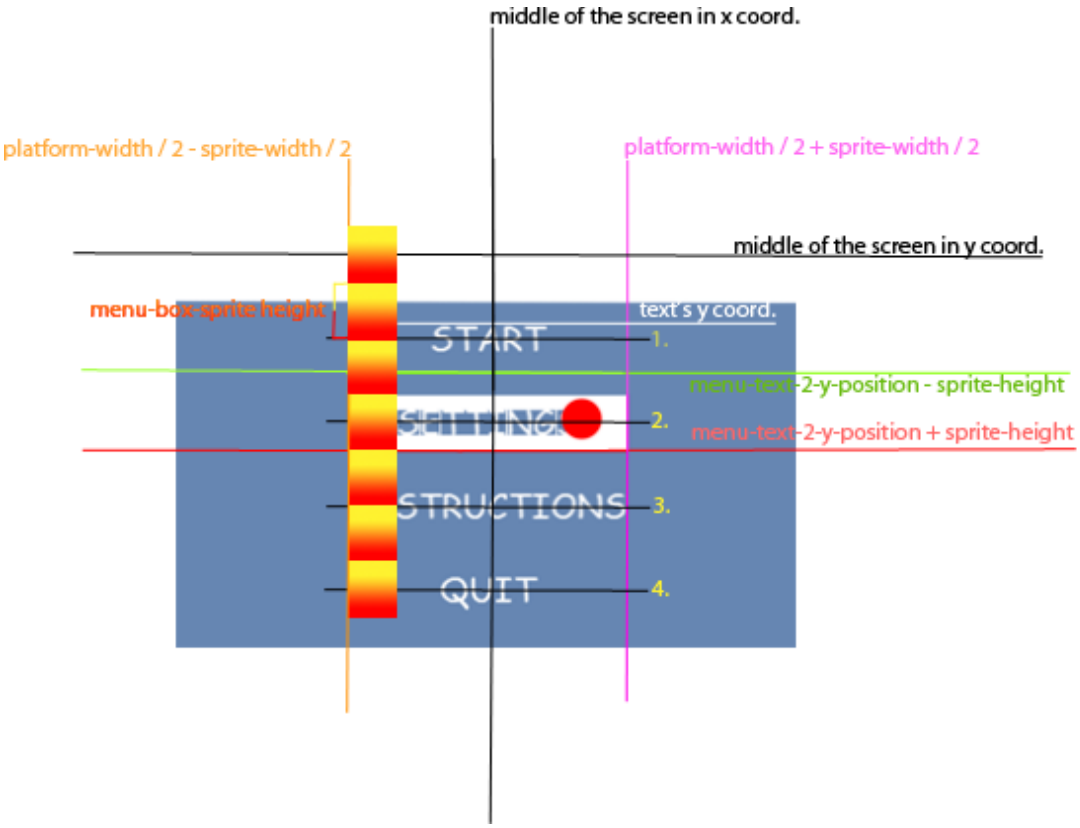
The shaded square is the screen's touch input area.

Menu

The in-game menu can be controlled by the Vitas button or touch input. The menu option choosing for button and touch input works very similarly. The application checks if the touch position is within the boundaries of the appropriate menu-text (if it's the button control the application will check for the position of the menu box). For example, going to the 'Settings' menu from the 'Frontend' menu through the touch control would be performed like this:

I. If the y coordinate of the touch position is bigger than y coordinate of the 'Settings' text minus the height of the menu box divided by two (`touch_position_.y > (menu_text_1_.y() - sprite_height * 0.5f)`).
II. If the y coordinate of the touch position is smaller than y coordinate of the 'Settings' text plus the height of the menu box divided by two (`touch_position_.y < (menu_text_1_.y() + sprite_height)`).
III. If the x coordinate of the touch position is bigger than the x coordinate in the middle of the screen minus the menu box width divided by two. (`touch_position_.x > (platform_.width() / 2 - sprite_width_ / 2)`).
IV. If the x coordinate of the touch position is smaller than the x coordinate in the middle of the screen plus the menu box width divided by two (`touch_position_.x < (platform_.width() / 2 + sprite_width_ / 2)`).

Menu-box-sprite positioning and menu-text positioning



The red-yellow gradient filled rectangle is a depiction of the menu-box-sprite height. The gradient divides it into two equal parts, so a half of the menu-box-sprite height can be easily noticed.

Menu-box-sprite positioning

Because sprite's position coordinates are in the middle of its 'x' and 'y' coordinates (in other words the centre of the rectangle sprite), the menu-box-sprite position is initialized to: middle of the screen in x coordinates (platform-width * 0.5); middle of the screen in y coordinates plus 1.5 times menu-box-sprite height (platform-height + sprite-height * 1.5). The menu-box-sprite's initial position is saved in a 'sprite-init-y-position' integer variable which is later used for boundary checking (so the menu-box-sprite does not go lower or higher than the menu-text).
When menu-box-sprite is moved up its 'y' coordinate is subtracted by 1.5 of its height and when it's moved down its 'y' coordinate is added by 1.5 of its height (Look at the distance between the black lines: it is 1.5 of the gradient rectangle's height).

The menu-box-sprite is clamped between the highest-menu-text (no. 1. in the graphic above) and the lowest-menu-text (no. 4. in the graphic above) by checking:

1. For going up:
    a. If the position of the highest-menu-text is smaller of equal to the current 'y' position of the menu-box-sprite minus its height (`menu_text_1_.y() <= menu_box_sprite_.position().y() - sprite_height`). If it is, that means that the menu-box-sprite would go above the highest menu-text and the condition will return 'false'.

2. For going down:
    a. If the position of the lowest-menu-text plus height of the menu-box-sprite is greater of equal to the current ‘y’ position of the menu-box-sprite plus its height (`menu_text_4_.y() + sprite_height >= menu_box_sprite_.position().y() + sprite_height`). If it is, that means that the menu-box-sprite would go below the lowest-menu-text and the condition will return ‘false’.

Menu-text positioning

Because text justification is to the centre, the x coordinate can be the same for all menu-texts, which is the same as the menu’s-box-sprite. The ‘y’ coordinate is menu’s-box-sprite initial position plus/minus menu’s-box-sprite height in different proportions. Keeping in mind that texts ‘y’ coordinate starts at the top of the text (look at the graphic), the menu-texts’ ‘y’ coordinates can be calculated accordingly:

1. Menu-text-1: menu-box-sprite’s initial position minus half of its height plus a height correction (3 units).
2. Menu-text-2: menu-box-sprite’s initial position plus its height and a height correction.
3. Menu-text-3: menu-box-sprite’s initial position plus its height multiplied by 2.5 plus a height correction.
4. Menu-text-4: menu-box-sprite’s initial position plus its height multiplied by 4.0 plus a height correction.

Sound

Having the ‘Audio Manager’ in the Game class, which elements are very often deleted and reinitialized was causing the sound to not work after a first reload and later was causing the whole application to crash. To solve this problem, the ‘Audio Manager’ is being initialized in the SceneApp class and then passed to the Game class by a pointer. To save the application a constant creation and deletion of the ‘Input Manager’, the same solution was later implemented for it (previously it was being created in each class separately).

User Guide

The application will run on Microsoft Windows operating system or Sony PlayStation Vita console and does not require a special version of the Games Education Framework. The audio is disabled on Windows because it was crashing the application.

When you run the application, you will be presented with a frontend menu. From there you can use up and down d-pad buttons to navigate the menu. You can toggle the appropriate menus or confirm the choice with the ‘CROSS’ button or simply tap the menu text to choose it. Left and right d-pad buttons will also let you choose the appropriate menu option. From there, you can read the instructions and game control guide.

Critical Analysis

Overall, my application fulfilled all the requirements described in the coursework specification. It makes use of the textured 3D geometry when creating the platforms for the level. The frontend has a form of a graphic-based splash screen (title and logo of the game, control guide and instructions displayed above the frontend menu). A camera, difficulty level and number of platforms can be set in the settings menu. All the entities in the game have a box2D physics body (the player has a dynamic body, while every other body is static [pickups are also sensors]). A collision response is being triggered many times during the game: pickup collection, collision response between the player and the platforms.

The Vitas touch and button input features are utilised when interacting with the in-game menus (although touch interaction can be sometimes inaccurate because the menus were not fully designed for touch control), controlling the player or the camera. There are examples of music and sound being played at appropriate moments.

The application makes good use of appropriate classes. However, I felt there are parts of the application that could be put into a class. For example, a 'menu' class could be created to hold all the menu logic for initialization and rendering the menu text and the menu-box-sprite, as well as all the menu touch and button input.

A good thing would be also having a way to indicate the process of loading the new scene. Right now, this process can be observed by turning on the 'FPS' counter and watching its state. If it is frozen, that means a new scene is being loaded.

Another thing that could be improved is the player's jumping mechanics. It would give the player a better feel of control if the ball were rising at the time of the button being pressed or screen being tapped, allowing for a long or a short jump. The gameplay experience could be also enhanced with an animated player, a skybox on the camera, environmental art and physics-driven particles generated when the player touches the platform (it would make for a smoke-like look).

Conclusions

I am confident that the module summarized all the knowledge and required all the skills I have learned since the beginning of my academic education. It required me to make a real use of an object-oriented programming. It was really apparent when I was trying to put all my game logic into one class, which resulted in huge frame-rate drop, because I was trying to update and render the entire scene every frame. Putting it into separate classes allowed for an initialization of the static elements in the memory, and for having a quite well working game-state-machine. The reason for making all the bank-account and pet programs is clear to me now.

All in all, I have really enjoyed the module and I am really glad that I spent time preparing for it.

References

Thank you to Grant Clarke for help with the collision response and to Chris Acornley for help with the game state machine.

Box2d.org. (2017). Links | Box2D. [online] Available at: http://box2d.org/links/ [Accessed 13 Jul. 2017].

Iforce2d.net. (2017). Introduction - Box2D tutorials - iforce2d. [online] Available at: http://www.iforce2d.net/b2dtut/ [Accessed 13 Jul. 2017].