C++ CW 4 Documentation File

Name: Diogo Jose Fidalgo Assuncao

Student ID: 14328314

1. Introduction:

My game is a platformer inspired by “The Matrix” movie, where the objective is to get the highest score possible by interacting with as many payphones as possible, while dodging the glitches with the help of different pills with different abilities , there are 4 different maps, and it is easy to add even more, with a couple different backgrounds.

2. Compilation and running:

Nothing special to consider.

3. File list:

Code .h/.cpp Filename	What this file is for, e.g. what class is in it and what it does
psydjfiEngine.h and .cpp	Subclass of BaseEngine – also handles sound
psydjfiTM.h and .cpp	Subclass of TileManager
MenuState.h and .cpp	Subclass of psydjfiTM – handles main menu state
GameState.h and .cpp	Subclass of psydjfiTM – handles game state
PauseState.h and .cpp	Subclass of psydjfiTM – handles pause menu state
TutState.h and .cpp	Subclass of psydjfiTM – handles tutorial stage state
ScoreState.h and .cpp	Subclass of psydjfiTM – handles Score saving and showing state Class score
psydjfiState.h	State Class Interface
playerObject.h and .cpp	Subclass of DisplayableObject – handles Player object
consumableObject.h and .cpp	Subclass of DisplayableObject – handles consumables (Pills) Class RedPill -- Subclass of consumableObject Class BluePill -- Subclass of consumableObject Class GreenPill -- Subclass of consumableObject
enemyObject.h and .cpp	Subclass of DisplayableObject – handles enemies Class mappingFlip
SaveNLoad.h and .cpp	Class Save – used for saving state Class Load – used for loading states

4. Resource file list:

I used the following resources:

- Images(./src/images/):
    - bluePill.png
    - brick2.png
    - digitalrain.jpg
    - dojo.jpg
    - enemy.png
    - glitch.png
    - greenPill.png
    - morpheus.png
    - neo.png
    - payphone.png
    - redPill.png
    - sprite.png
    - spriteAura.png
    - spriteHurt.png
    - subway.png
    - wood1.png
- Music(./src/music/):
    - dirt-rhodes-by-kevin-macleod-from-filmmusic-io.wav
    - hiding-your-reality-by-kevin-macleod-from-filmmusic-io.wav
    - metalmania-by-kevin-macleod-from-filmmusic-io.wav
    - rising-tide-faster-by-kevin-macleod-from-filmmusic-io.wav
    - tech-live-by-kevin-macleod-from-filmmusic-io.wav
        - All licences are included in the MusicLicenses.txt in the same directory
        - Credit is also given in the main menu of the game
- Text Documents(./src/saves/):
    - maps.txt
    - saveFile.txt
    - scores.txt

5. Requirements

Requirement 1: Add states to your program

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Menu State	0:00 2:23 4:41	0:27 2:55 6:48	MenuState.cpp – startup stage Changes to other states with handleEnter() Lines 125 - 143
Game State	0:59 2:56	1:56 4:07	GameState.cpp Changes to Pause State with ESC key – line 95
Tutorial State	0:33	0:58	TutState.cpp Changes to Game State – line 214 Changes to Menu State – line 217 Uses parameter returnToMenu to decide Goes to game engine, when it enters in Tutorial state by pressing play when there are no scores saved(assumes first time play)
Pause State	1:58	2:22	PauseState.cpp Returns to game state – line 74 Returns to Main menu – line 147
Score State	0:28 4:08	0:30 4:40	ScoreState.cpp Returns to Main menu – line 197

Other useful files are psydjfiEngine.cpp and psydjfiState.h for state model implementation.

I believe I should get 2 marks for this requirement.

State model is implemented by having psydjfiEngine calling the functions of the currentState pointer (psydjfiState*), the states are created by whoever is switching states and then the old state deletes itself, to switch it uses the setState() function in psydjfiEngine, this will update the currentState pointer and call initializeState() with it. All the states are subclasses of psydjfiStates and implement subtype polymorphism to overload the functions and have completely different outcomes.

Besides the state model I have also implanted 5 largely different states including a start-up, running and pause state.

Requirement 2: Save and load some non-trivial data

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Load Map	0:59		GameState.cpp ReadMap() reads from file – lines 420-434 LoadMap() uses the string read to load it into the tile manager – lines 123-168
Load State	2:55		SaveNLoad.cpp Load Class readFile() reads the save – lines 450 -462 LoadAll() saves the parameters into the class – lines 224-447 startLoadedGame() uses the saved parameters to start objects and states with everything they require to be the exact same as they were – lines 131 - 221
Save State	2:22		SaveNLoad.cpp saveAll() is a combination of saveState() that saves everything need from the state (lines 38 - 60), savePlayer() saves relevant information about the player and consumable object () (lines 69 -84) and saveEnemy() saves information relate to enemies, this may be called multiple times or none, relating to the number of enemies that exist at save time (lines 93-110) – lines 16-31 writeToFile() writes the string produced into the save file – lines 113 – 123
Load Score	0:28		ScoreState.cpp readScore() reads, saves and sorts all scores in the file – lines 145-172
Save Score	4:33		ScoreState.cpp saveScore() writes the name and score to the scores.txt file – lines (212 - 224)

I believe I should get 3 marks because the save state process expands over multiple classes of states and objects and returns right to the same position it was once it was loaded, this includes position and number of objects in enemyObjects case as well as player states as been hit or currently invincible, all these factors and being saved and loaded!

In addition, the maps are also loaded form a file, and the scores are also written and read from a file!

Note: Loading eliminates the previous save, and save quits to menu, this was done on purpose for gameplay reasons as this is game where the goal is to get the highest possible score being able to start with any score at any time would break the game, like this it makes sure if you die you will surely lose your progress, but you can rest assured you can save and keep going later!

Requirement 3. Interesting and impressive automated objects

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Player Object	0:59 2:56	1:56 4:07	playerObject.cpp
Enemy Object	1:46 2:55	2:22 3:10	enemyObject.cpp
Consumable Object	3:17	3:34	consumableObject.cpp

All of these objects are using images, but they also are all direct subclasses of DisplayableObject, I started out using ImageObject but found out it was quite limited and so implemented my own versions of it.

All of these objects also move, player Object is controlled by the player, the enemy object finds the shortest path and moves on the player, and consumable object moves up and down on the spot!

Consumable object is more of an interface but can be created and is used at the beginning as a placeholder, it has 3 different subclasses that use sub class polymorphism to implement different behaviours to the consumables.

Enemy object is created using coordinate mapping and it moves towards the player while finding the shortest possible route!

Player Object is by far the most impressive to me, it implements 3 different sprites to signal damage and invincibility these sprites consist of an animation cycle of 16 frames, I want to mention I made the sprite myself and I am not much of an artist but I think it didn't turn out bad for my first time, another thing of note is the way movement is handled is not just a key press that makes him go one way or the other, I tried to implement a system that mimics physics so an object has horizontal and vertical acceleration, the acceleration is raised by a force you apply when pressing keys, when you create acceleration one way then friction will be applied in an opposite way, this is why you stop once you stop pressing the key, the force stops, but friction continues to diminish acceleration until it reaches 0, for vertical this friction is applied as a gravitational force that always pushes down unless you are in contact with a surface, as with any other surface the acceleration is negated when in direct contact against it. Finally, the object collision that checks all the border pixels between the mask and player to make up a pixel perfect collision!

Requirement 4. Impact/Impression (and requirement L: Sellable quality)

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Game State			gameState.cpp
Player Object			playerObject.cpp
Save and Load			SaveNLoad.cpp
Enemy Object			enemyObject.cpp

After many hours working in this project, I am very proud of how it came out, it was not an easy project to make but I think it was worth the work!

The First thing I'd like to point is the was the game looks all images but the background and faces where made by me, I am not much of an artist and it was quite a learning experience from drawing sprites to tiles, I think as a finished product it quite good looking, but I might biased.

In a programming aspect to me what stands out is the Game State, especially its map loader, and game spawner, allowing for a different game every time but one that you can master for a better score. Besides this the Player object with its movement mimicking real physics and its pixel perfect collisions. Also, the enemy object and it's smart and efficient path finding abilities. To finish up the save and load classes are also very smart managing to deduce a lot from the least amount of data to make a time and space efficient load and save cycles.

I also think the other classes complement each other quite well, for example I am quite proud of the UI because it is both easy to understand and quite appealing visually.

The tutorial is also a great way to start the game, and learn the basics in a nice un-usual faction.

It is also good to mention I've made it such that expanding the game will be as easy as possible, things like the random map loading allow for easy introduction of new maps, and the tile manager allows for more and different kinds of tiles, everything so you have a product that is maintainable and expandable if desired.

I think all this points out to the quality of the final product!

Requirement A: Correctly implement scrolling and zooming using the framework's FilterPoints class

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Zooming	0:33	0:58	TutState.cpp The tutorial state's foreground is zoomed in at all times – lines 100 Set on constructor, to be applied after translation
Scrolling / translation	0:41 0:50 0:54 0:57		TutState.cpp Tutorial state uses scrolling to show you the objects you will encounter during the game this can all be found in the handleState() – lines 119, 135, 165, 187 Set on constructor with zoom as the next filter

Both are initialized on constructor, with translation being followed by Zoom, the compression happens when initializing the state and remains until the state is changed, translation changes several times to show the several objects and images the player will find in the game

Requirement B: Have advanced animation for background and moving objects

Item	*Video start time*	*Video end time*	*Source files and line numbers*
playerObject	0:59 2:56	1:56 4:07	playerObject.cpp, Player object is animated with a sprite that is updated by changing the offset of where the image is drawn, this happens in the updateSprite() function – lines 395 -430
Main Menu / Score Menu	0:00 2:23 4:41 0:28 4:08	0:27 2:55 6:48 0:30 4:40	The background animation is achieved with a scrolling background in the same way for both states Using mainLoopDoBeforeUpdating() and CopyBackgroundBuffer() MenuState.cpp – lines 187 -200 ScoreState.cpp – lines 38 - 52

I believe I should get 2 marks.

Background animation although it is a simple background scroll, the image was carefully edited to make the seems almost imperceptible, and even though it is a simple method I believe the result is very visually impressive.

Whatever background lacks on being complicated the player object certainly makes up for it, it is composed of a group of 3 sprites with 16 frames for each cycle, that are updated according to the acceleration of the player, these change between each other to give player effects like flashing red when hit or having an aura that flashes when it is about to end, to make the animation smooth when walking I also had to make sure it didn't activate for every iteration of virtDoUpdate() or else the animation would be fast and ugly, you can see this applied in lines 356 – 359 of the playerObject.cpp file, the sprite is updated with the updateSprite() function which will assign the right offset for the current speed.

Requirement C: Interesting and impressive tile manager usage

Item	*Video start time*	*Video end time*	*Source files and line numbers*
psydjfiTM	------	---------	psydjfiTM.cpp
spawner functions	1:04 1:06 1:12 1:21 1:24 1:30 1:38 1:46 3:08 3:17 3:41 3:45 3:51 3:56 4:00		gameState.cpp – lines 213 - 283
Save and Load	2:22 2:57		SaveNLoad.cpp saveState() – lines 38 -60 startLoadedGame() – lines 131 - 221
Collisions	4:06		playerObject.cpp handleCollisions() – lines 116 - 321
Path finding	1:46 2:55	2:22 3:10	enemyObject.cpp handleMovement() – lines 94 - 172

I believe I should get 2 marks.

Although the tiles themselves are simple images for the most part I believe the truly impressive use is within game state where the tile manager is key to spawn the objects, the spawner functions uses the tile manager to know where they can or can't spawn both new tile types and objects without it organization like not having traps next to each other or an objective not spawning in the same place as a trap, this is where the tile manager shines. (see Spawner Functions in gameState.cpp)

But this is not all it does, it is also a key element in the saving and loading state process, the save and load use the a sort of "bitmap" (not actually because it uses more than 2 values) to know where all traps objectives and consumables are, this makes the saving and loading much faster and efficient , not needing to load and save even more variables. (see saveState() and startLoadedGame() in SaveNLoad.cpp)

Also, there is also the collision with the player, if the player is in an objective tile he will be able to interact with it, if he steps on a trap tile he loses a life, and he can stand on the wood and brick tiles. (see handleCollisions() in playerObject.cpp)

Finally, it is also used in the enemy object path finding algorithm where he saves the tiles where he can change Y levels so he can calculate which is the fastest route to the player. (see handleMovement() in enemyObject.cpp)

Requirement D. Creating new displayable objects during the game

Item	*Video start time*	*Video end time*	*Source files and line numbers*
ConsumableObjects	3:17		gameState.cpp spawnConsumable() – lines 276 - 312
enemyObjects	1:46		gameState.cpp spawnEnemy() – lines 316 - 347

Both objects are created when gameState's gameSpawner() calls their respective functions, the consumable objects if picked up will turn invisible and be deleted when/if another consumable is spawned as there is only one at a time, the enemyObject will be delete when it makes contact with the player or its lifespan expires, these states are properly loaded and saved, consumable objects visibility is saved its position can be found with the tile manager, the position and lifespan of enemy objects are both saved and loaded as well to make these objects work as if nothing happened

Requirement E. Allow user to enter text which appears on the graphical display

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Score name	4:07	4:32	scoreState.cpp drawStringsOnTop() – lines 55 – 142 keyDown() – lines 175-201

Used when saving a score player can input a 3-letter name, no special characters accepted, delete eliminates previous typed letter, saves these so it can then write into the score file.

Requirement F. Complex intelligence on an automated moving object

Item	*Video start time*	*Video end time*	*Source files and line numbers*
enemyObject	1:46 2:55	2:22 3:10	enemyObject.cpp handleMovement() – lines 94 -172

I believe I should get 2 marks.

Enemy Object will use an algorithm to find the shortest path to the player.

For this algorithm it simplifies the tile manager map to contain only 7 vertical layers interlocked between walkable layers and platform players, it also saves the tiles that allow it to change layers. (this is actually done in loadMap() and then sent to the enemyObject)

If the player is in the same layer then the object will move towards him.

If the player is in another layer then the object will calculate which of the tiles that allow it to change layers is closer, it calculates this distance by adding the distance between this space and the enemy with the distance between the space and the player, it will then move toward the one where this distance is shortest, if the player changes layers into the same on the enemy is it will forget its previous target and go straight to him.

This makes the higher speed version of the enemy extremely effective, which is why its lifespan suffers so much when opposed to the default speed version.

Requirement G. Non-trivial pixel-perfect collision detection

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Player Object	0:59 2:56	1:56 4:07	playerObject.cpp handleCollisions() – lines 116 -321
Updated virtIsPositionWithinObject	3:09		enemyObject.cpp – lines 216 - 222 consumableObject.cpp – lines 86 -92

Player Object uses a pixel perfect collision, the way it does this is it goes through all pixels in the image.

It first finds the border pixels, if a pixel is the same colour as the background colour then ignore else check if any of the connecting pixels is a background colour if so, it is a border pixel and we can use it for collision else you can ignore it.

Now that we have these borders pixels we can use them to check for collisions, but we only check the connecting pixel that is background coloured if this pixel is in contact with a window border, tiles with certain values then there is a collision, both consumable objects use an updated overloaded version of virtIsPositionWithinObject() that also checks if it is inside the object and that pixel is not the background colour either.

Requirement H. Image rotation/manipulation using the CoordinateMapping object

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Class mappingFlip	1:46		enemyObject.cpp – lines 248 - 271
enemyObject	1:46 2:55	2:22 3:10	enemyObject.cpp virtDraw() – lines 68 - 71

Class mappingflip applies a flip it has got 2 settable options, horizontal and vertical flip, these can be set separately, together or none, giving 4 different results. It does this when horizontally having the x position removed from the image width, and in vertical removing the y position from the image height this will flip an image through a horizontal and vertical axis that go through the centre of the image.

As I was having problems with applying mapping and masking at the same time with the original framework, I decided to add a masking option to the constructor of mappingflip, this means if you set a colour in mappingflip that colour will not be drawn, mimicking the renderWithMask() function!

Requirement I. Integrate sound using SDL

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Sound Functions	0:00	5:35	psydjfiEngine.cpp – lines 167 - 250

The sound present in the game is all background music, all controlled by the functions in psydjfiEngine, the callback function was a bit confusing at first until I understood you just needed to copy into the stream buffer, also had a hard time not reopening the audio multiple times, which was leading to an error message I had setup. To play a song just can playMusic() with the song path as the parameter and if you want to turn off the music use toggleMusic().

Requirement J. Show your understanding of templates and/or operator overloading

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Class score	----------	----------	scoreState.cpp – lines 260 - 277

The comparison operators for the score class were all overloaded to their reverse, except ==.

I did this so the sort function for the vector could be used with the default comparison function but still order the scores from largest to smaller, I am aware I could have easily set up a different function but felt like it was a good application of overloading operators.

Requirement K. Use your own smart pointers appropriately

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Unique_ptr	----------	----------	MenuState.cpp – line 222
Shared_ptr	----------	----------	psydjfiEngine.cpp – line 14

To test my comprehension of smart pointers I used a unique pointer in MenuState.cpp for a pointer I was using shortly and then had to delete, it was a great success!

After I understood it a bit better I made the Tile Manager pointer that is shared between most states and classes a shared pointer, it is initialized in psydjfiEngine, and then spread out to every other function with the getTileManager() function. This also removed the need to purposely delete the tile manager on exit of program! And we can also make sure the pointer is not deleted until psydjfiEngine finishes.

Requirement M. An advanced feature I didn't think of but you had pre-approved

Item	*Video start time*	*Video end time*	*Source files and line numbers*
----------	----------	----------	----------