C++ CW 4 Documentation File

Name: Luqman Ariffin

Student ID: 14326114

1. Introduction:

My program is a game, the objective is to survive and collect as many points by killing enemies. Creates random maps, if existing maps do not exist (thus loads and creates maps). Different types of enemies with different actions that dynamically appear. Makes use of both keyboard and mouse controls. Top 5 scores and player's name saved when game is closed.

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
PsylhaEngine.h/cpp	Main BaseEngine subclass
PsylhaTileManager.h/cpp	Main TileManager subclass
PsylhaObject.h	Main DisplayableObject subclass
Button.h/cpp	PsylhaObject to create interactive buttons. Button subclasses defined within Button.h
Shot.h/cpp	PsylhaObject to display shots. Shot subclasses defined within Shot.h
DisplayableText.h/cpp	PsylhaObject to display text to foreground by object.
PsylhaActor.h	PsylhaObject subclass for specific 'moveable' objects
Player.h/cpp	PsylhaActor for the player
Enemy.h	PsylhaActor subclass for specifically enemies
Skeleton.h/cpp	Enemy subclass for skeleton
Demon.h/cpp	Enemy subclass for demon
State.h	State base class for all states
MenuState.h/cpp	State subclass for menu
NameState.h/cpp	State subclass for entering name
ScoreState.h/cpp	State subclass for viewing highscores
RunState.h/cpp	State subclass for playing game
PauseState.h/cpp	State subclass for pausing game
DeathState.h/cpp	State subclass for player death
ExitState.h/cpp	State subclass for exiting program
Score.h	Base class for score information
TextContainer.h	Base class for text information
TileImage.h	Base class for animated tiles

4. Resource file list:

All images in **src/resource** directory. Further separation depending on image:

/enemy/demon	
/left/	***/right/***
idle0.png	
idle1.png	
idle2.png	
idle3.png	

/enemy/skeleton	
/left/	***/right/***
idle0.png	
idle1.png	
idle2.png	
idle3.png	
run0.png	
run1.png	
run2.png	
run3.png	

/player/				
/left/	***/right/***	***/large/***	emptyheart.png	fullheart.png
idle0.png				
idle1.png				
idle2.png				
idle3.png				
run0.png				
run1.png				
run2.png				
run3.png				
hit.png				

/tile/	
/bluefountain /	***/redfountain /***
bottom0.png	
bottom1.png	
bottom2.png	
bottom3.png	
mid0.png	
mid1.png	
mid2.png	
mid3.png	

/tile/floor/
floor_1.png
floor_2.png
floor_3.png
floor_4.png
floor_5.png
floor_6.png
floor_7.png
floor_8.png

/tile/wall/
wall_left.png
wall_mid.png
wall_right.png
wall_side_front_left.png
wall_side_front_right.png
wall_side_mid _left.png
wall_side_mid _right.png
wall_top _mid.png

/tile/button/
false.png
true.png

/back.png
/title.png

5. Requirements

Note on video and times: many items happen at once when the game runs and they happen multiple times. Video start time will be when I believe it is most evident. It may be necessary to keep in mind the item in order to find it in the timestamp. Sorry for the inconvenience.

Requirement 1: Add states to your program

Item	*Video start time*	*Video end time*	*Source files and line numbers*
State	-	-	PsylhaEngine.h line 24 – setState method. Resets state pointer. PsylhaEngine.h line 28 – setStateNoFlag method. Resets state pointer.
Menu State	0:00 0:33	0:04 0:35	PsylhaEngine.cpp line 34 – state applied to pointer directly in engine Button.h line 90 – MenuButton class, virtMouseUp method. calls setState method in PsylhaEngine with new instance of MenuState. DeathState.cpp line 67 – MenuButton instance
Name State	0:05	0:10	Button.h line 76 – NameButton class, virtMouseUp method. calls setState method in PsylhaEngine with new instance of NameState. MenuState.cpp line 57 – NameButton instance
Score State	0:18	0:20	Button.h line 47 – ScoreButton class, virtMouseUp method. calls setState method in PsylhaEngine with new instance of ScoreState. MenuState.cpp line 55 – ScoreButton instance
Run State	0:21 3:47	0:31	Button.h line 105 – PlayButton class, virtMouseUp method. calls setState method in PsylhaEngine with new instance of RunState. MenuState.cpp line 54 – PlayButton instance PauseState.cpp line 50 – keyUp method. If any key is pressed, calls setStateNoFlag method in PsylhaEngine with new instance of RunState.
Pause State	3:43	3:46	RunState.cpp line 285 – keyUp method. If ESC key is pressed, calls setState method in PsylhaEngine with new instance of PauseState.
Death State	0:32	0:34	Player.cpp line 72 – virtDoUpdate method. If statement checks health. If true, call setState method in PsylhaEngine with new instance of DeathState.
Exit State	0:39	0:40	Button.h line 118 – ExitButton class, virtMouseUp method. calls setState method in PsylhaEngine with new instance of ExitState. MenuState.cpp line 56 – ExitButton instance

I believe I deserve two marks for this.

I applied the state model. There is a State base class (State.h) to which all other state classes inherit. This allows for subtype polymorphism, different behaviours occurring while the same methods are called in PsylhaEngine.cpp. This is because the subclasses override the virtual methods in the base class for its own actions. An if-statement to check for the ExitState is added to safely quit the program.

PsylhaEngine.cpp contains a State smart pointer, through unique_ptr, which allows applying new states and destroying old state instances. As it uses State, it allows for any subclass of it to be assigned.

PsylhaEngine.cpp has two methods to call to apply a new state; setState and setStateNoFlag. Both methods reset the smart pointer. setState applies a flag for the engine to know that a new state was applied. This is used in virtMainLoopPostUpdate, to safely destroy all objects without invalid read errors occurring due to being called during the loop and handling events (which could be trying to notify objects that no longer exist). Applying the new state's foreground and background after updating the objects ensures safe handling. setStateNoFlag allows resetting the smart pointer while bypassing the flag, so nothing existing is affected.

Those two methods are usually called by subclasses of Button, which are specific to switching to certain states. Certain states (PauseState and DeathState) are not accessed by a button press but rather an event (such as a key press or no lives left). If these Button subclasses are created and added to the object array, this allows for a way to switch states.

Requirement 2: Save and load some non-trivial data

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Loading Scores	0:35	0:47	PsylhaEngine.cpp line 89 – loadScores(). Scores read from file, inserted into vector.
Loading Name	0:39	0:45	PsylhaEngine.cpp line 113 – loadName(). Name read from file, stored in string variable.
Loading Map	0:21		RunState.cpp line 36 – setupBackground(). Map read from file, loads tile values into PsylhaTileManager to create map.
Saving Map	2:23	2:28	RunState.cpp line 49 – setupBackground(). If map file does not exist, program generates new random map and saves it onto file for future use.
Saving Scores	2:38	2:40	ExitState.cpp line 19 – saveHighscore(). Writes all scores from vector into existing file (guaranteed to exist due to loadScores()).
Saving Name	2:32	2:35	ExitState.cpp line 31 – saveName(). Writes name into existing file (guaranteed to exist due to loadName()).

I believe I deserve two marks for this.

Scores and name values are loaded and saved upon start and termination of program respectively. Map data is also read and written. Tile values are based upon data read from file.

When loading data, it is read from its respective file. If the file does not exist, the program handles this by creating new file to specified destination and filling it with default data. Data is saved by truncating existing file (guaranteed to exist by attempting loading in beginning of program) and writing data into file with expected format for loading. Saving map data is slightly different as it is only performed when the file does not exist, thus it does not need to truncate any file.

File operations are performed only when vital, otherwise it is dealt with only within the program. For example, if there are multiple changes to name or the high score list, it is only updated within the program until the program is exited appropriately (meaning exited through the program's custom exit button).

Requirement 3. Interesting and impressive automated objects

Item	*Video start time*	*Video end time*	*Source files and line numbers*
PsylhaActor Tile Collisions	 1:12(topleft corner) 3:04	 1:17 3:15	PsylhaActor.h line 36 – doCollisionDetection(…) method. Custom collision based on tile values. Skeleton.cpp line 109 Player.cpp line 156
Enemy homing on player	0:22 0:24		Skeleton.cpp line 86 – virtDoUpdate() method. Walks towards player locations. Demon.cpp line 92 – virtDoUpdate() method. Shoots towards player.
Enemy collisions	1:06	1:20	Skeleton.cpp line 54 – virtDoUpdate() method. Demon.cpp line 61 – virtDoUpdate() method. Both checks for PlayerShot object and Player object
Health hearts updated dynamically	3:50	3:55	RunState.cpp line 234 – PsylhaObject updated based on current health
DisplayableText object for score	1:58	2:05	RunState.cpp line 243 – DisplayableText updated to current points

I believe I deserve the one mark.

My intermediate class between the framework and my end class is PsylhaObject. PsylhaActor is a subclass of PsylhaObject and has atleast three subclasses which have different appearances and behaviours. Demon and Skeleton objects load images from their respective folders (see resource file list).

Custom tile collisions allow PsylhaActor objects to move only on 'valid' tiles (floor and button). Collision boundaries were placed at the 'feet' of the objects, which visibly made sense to the user. This collision system not only pushes the actor object back (to cancel out its movement), but allows it to 'slide' along the collisions in a slower pace. This collision system has been tested to not have unexpected behaviour (objects do not go over invalid tiles even in awkward shapes). Base detection is done by reading the value of the tile which would be a few pixels in the direction the object is going (a few pixels left if going left, etc). To allow for 'sliding' effect, slower movement is applied to that direction while in collision. This requires another check for invalid tiles in that direction. Thus, ensuring expected behaviour in even awkward corners. The collision detection also had to be able to recognise different behaviours with the same keystrokes. For example, if the user presses up and left, it could mean colliding upwards and sliding left or colliding left and sliding upwards. If this was not accounted for, unexpected behaviour occurs (such as suddenly walking over holes).

Enemy homes in to the player using Observer pattern, where the enemy objects are the observer and the Player is the subject. The Player object notifies the Enemy objects of its position, which the enemy uses to move or shoot towards.

Enemy collisions with other objects (PlayerShot and Player) are done using the framework's CollisionDetection, checking rectangles and if its positions overlap. If it does, it would remove and delete the Enemy object.

Enemies are dynamically created and placed into the game. This is dependent on the number of existing enemies and type of enemies. For example, currently if two Demon objects exist, then no more Demon objects will be created. Enemy objects are created and placed in set time intervals. Enemy objects are assigned to a random valid location (only valid location would be floor tiles, where tile values are 1).

During RunState, information on the game (player health and score) were updated dynamically according to its running data.

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

Item	*Video start time*	*Video end time*	*Source files and line numbers*
At least three moving objects			See requirement B
User controlled object	0:21		Player.h/cpp – virtDoUpdate() Checks key presses to control location and action of Player object.
Mouse input	0:04 0:21		Button.h – shows multiple uses of method virtMouseUp() to check input about whether the user clicked the button. Player.cpp line 136 – uses location of mouse as destination for Shot object.
Keyboard input			See requirement E
Visually looks good			See requirements B and C
Appropriate states			See requirement 1

I believe I deserve 1 mark for this requirement.

Items listed are based off the requirements sheet. Reference to other requirements for further information on how the game applies these requirements (to not repeat these requirements).

I aimed for high visual impact, minimally using primitive drawing operations to maximise this. All animations appear smooth to the user. Multiple states to appropriate navigate and act within the program.

Player.cpp is the user-controlled object. It is controlled by key presses, which are checked in virtDoUpdate(). The player can navigate around the map and shoot.

Mouse input was used primarily to register button presses.

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*

Requirement B: Have advanced animation for background and moving objects

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Background animation	0:55	1:00	PsylhaTileManager.cpp line 191, 207 – initial drawing to background. RunState.cpp line 253 – updateTiles(). Method is run on every loop of the engine, creating a smooth animation in the background surface.
PsylhaActor animations	0:21	0:35	Player.cpp line 48 – virtDraw() Demon.cpp line 32 – virtDraw() Skeleton.cpp line 31 – virtDraw() Image animation sprite loaded depending on current frame.

I believe I deserve at least one mark for this. It is unclear what needs to be done to be considered 'impressive'. All animations that are run (both foreground and background) can be seen as smooth (it does not look unnatural/stuttering). Images change according to engine's time function (getModifiedTime()), showing it does change over time.

Objects are animated by the engine calling each object's virtDoUpdate() method, which is where file paths are updated depending on direction and action. All objects are updated through Player object calling redrawDisplay() at the end of its virtDoUpdate() method. This shows only one call is being used to have to redraw the whole display. It is done in the Player object as it is the only object that will always be present during RunState, which is where redrawing is critical.

Background animations, which are objects that are not within the object array and therefore not called to update, are updated manually in the RunState method updateTiles(). This method updates the file path of the tile object and then renders the image to the background surface. This method is called in the start of the main loop, thus creating a smooth animation.

Requirement C: Interesting and impressive tile manager usage

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Six distinct tile types	0:21		PsylhaTileManager.cpp Line 37 – floor tiles Line 81 – hole tiles Line 112 – front facing wall tiles Line 160 – side wall tiles Line 191 – wall fountain tiles Line 207 – floor button tiles
Animated tile object	0:55	1:00	PsylhaTileManager.cpp Line 15 – wall fountain tile images stored in TileImage object Line 17 – floor button tile image stored in ButtonTile object

I believe I deserve two marks for this requirement.

While there are only six tile types, there are more tile images that the tile manager works with. This allows simplification of map-making (fewer varying values) while still having a satisfying and natural result (it does not look like there are missing or misplaced images). This also allows random generation of the map, as it follows set conditions for what time of image should be placed in the tile.

To elaborate with an example, front facing wall tiles are one of the more complex tiles to draw an image for. Depending on the tile values around the current tile, different images will be drawn. Within front facing wall tiles, there are five images that can be drawn. Two back corner images, two front corner images and the middle front wall images. To better visualise, pause the video at 0:21. A similar process is within most tile types.

As mentioned in Requirement B, there is a smooth animated tile. The wall fountain tiles are smoothly animated to the background surface. All images are stored in the object (TileImage), therefore only loaded once. Tile is redrawn with every loop. Reference requirement B for more information.

For the second mark, the ButtonTile is an interactive tile. If the player steps on top of it, the image is updated to show it is stepped on. Image also updates when the player steps off. When the player steps on the button, the colour of the fountain is changed, and changed only once until the player steps off and then back on. This is done following the custom tile collision mentioned in requirement 3, except it is interaction now. It checks if any part of the Player's 'feet' is touching the button tile. Repeated action (of when the ButtonTile is stepped on) is avoided by using a flag to limit calling the action method to only once.

Requirement D. Creating new displayable objects during the game

Item	*Video start time*	*Video end time*	*Source files and line numbers*
Enemy objects created dynamically	0:21		RunState.cpp line 208, 221 – Object is created and placed in a random valid location.
Shot objects created dynamically	0:22 0:25		Player.cpp line 131 – created when on event Demon.cpp line 99 – created in set intervals.
Enemy objects destroyed	1:23 1:25		Skeleton.cpp line 79 – Skeleton object destroys itself Demon.cpp line 86 – Demon object destroys itself
Shot objects destroyed	1:24 1:35 1:26		Shot.cpp line 45 – Shot object destroys itself as expected Skeleton.cpp line 74 – Shot collides with Skeleton and is destroyed early. Demon.cpp line 81 – Shot collides with Demon and is destroyed early.

I believe I deserve the one mark.

Objects are created when needed and properly destroyed to ensure no memory leaks. Enemy objects are created in a set interval and have a limit of how many of their type (skeleton or demon) can be active at once. Their location is also randomly set to a valid tile location. Shots are also created dynamically, where PlayerShot objects (subclass of Shot) are created when the player presses ‘spacebar’ and EnemyShot objects (subclass of Shot) are created in a set interval.

Shot objects are destroyed when they reach their given end destination (PlayerShot targets the cursor and EnemyShot targets the Player) or if they reach an obstacle before (if PlayerShot hit an enemy before reaching the cursor). Enemy objects are destroyed upon collision with either Player or PlayerShot. The process of deleting the object include removing it from the object array and also deleting it, as removing it alone does not delete it.

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*
Capture key press and shows respective letter input	2:58	3:00	DisplayableText.cpp line 58 – virtKeyDown(…) Displays character which was inputted on screen dynamically.
Capture BACKSPACE key	2:50	2:52	DisplayableText.cpp line 47 – virtKeyDown(…) Handles if BACKSPACE was pressed.
Overflow handled	2:52	2:54	DisplayableText.cpp line 55 – virtKeyDown(…)

I believe I deserve the mark for this requirement.

This is shown in NameState, where user enters their name. This is possible due to DisplayableText class, which has the option to be editable or not. This allows for the object to be used to display text to foreground using an object, in order to keep track of the string drawn, with the option to be able to read key presses or not. It is also possible to set character limits. Names are limited to three characters and only letters are available.

If any valid key is pressed, its corresponding character is pushed to the back of the string. However, if the three-character limit is reached, it does nothing. If the backspace key is pressed, it pops the last character of the string. If the string has no characters, it does nothing. It is updated to the screen as it is an object placed in the object array.

For implementation of DisplayableText (and Button), a TextContainer is used. Constructor of TextContainer includes size of text, the text string and text colour. From creating this, the width and height of the text can be found, and therefore used to properly align the text on the screen. It also groups text data, making it simpler to reuse.

Requirement F. Complex intelligence on an automated moving object

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

Requirement G. Non-trivial pixel-perfect collision detection

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

Requirement H. Image rotation/manipulation using the CoordinateMapping object

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

Requirement I. Integrate sound using SDL

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

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

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

Requirement K. Use your own smart pointers appropriately

Item	*Video start time*	*Video end time*	*Source files and line numbers*
State unique pointer	-	-	PsylhaEngine.h line 42
Score shared pointers	-	-	PsylhaEngine.h line 45 PsylhaEngine.cpp line 107 – creating individual smart pointers

I believe I deserve the mark for this requirement. Under the std namespace, unique_ptr and shared_ptr were used to store these smart pointers. make_unique and make_shared were used respectively to create these smart pointers.

Using a shared pointer would work over raw object pointers in most cases, as the pointer is only destroyed when all instances of it are no longer in use/ out of scope. However, these two cases had very appropriate uses for smart pointers.

The State smart pointer is a unique pointer, meaning there can only be one instance of it. As it is initialised in the header file of the engine, it essentially exists the lifetime of the program as it is always in scope. However, its 'unique' property ensures that no instance of it will ever exist outside the PsylhaEngine class. This is important as the State pointer is essential to running the program and allowing different behaviours to happen safely.

The Score vector comprises of smart shared Score pointers. Of most objects throughout the program, this vector is used quite a lot. In NameState, it is read from to show all the scores. In DeathState, it is used to insert the Player's score into the vector appropriately. In ExitState, it is used to save the high scores into the file. Using a smart pointer ensures it is handled appropriately in all these instances where it could be difficult to manage. Most other objects stay within the State subclass that it was created in, not needing to be used as much as the Score objects.

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*