Module Code
COM00008I

BSc, BEng and MEng Degree Examinations 2019–20

DEPARTMENT OF COMPUTER SCIENCE

Software Engineering Project (SEPR)

Open Group Assessment

Module	Software Engineering Project (SEPR)
Year	2019/20
Assessment	3
Team	The Dicy Cat
Members	Michele Imbriani Daniel Yates Luke Taylor Isaac Albiston Martha Cartwright Riju De Sean Corrigan
Deliverable	Implementation Report

Additions

The main addition to our game is a minigame to the project which gets increasingly harder as the game progresses as to meet UR_minigame [1]. This minigame consists of a 2D platformer in which the user has to plant water bombs around the place to take back control of the monuments which the aliens have invaded. The platformer consists of a firefighter, some aliens and spikes in the ground which the user has to avoid. The architecture of the minigame is similar to the architecture of the main game but with a few additions to make it more maintainable and simpler.

Generally, we decided to keep much of the architecture the same as it was in previous sprints as we found it adequate and efficient for what we needed it for.

One main feature of the architecture we continued was the use of a “state” system which uniformly set all main screen displays to be interactable in similar ways and controlled by one “GameStateManager” [2]. Using this design, we could use the preexisting infrastructure to create a “MiniGameState” which provided a base to build the minigame from. This also keeps within the lines set out by the FR_game_states requirement which specifies that the displays on-screen shall be controlled by a state stack.

From here we used the same inheritance structure of entities and units for the objects we needed to move around the screen however instead of using the “Character” class we created our own “MiniGameUnit” class which allowed us to implement the different style of movement needed in a platformer.

To store and control all of the new classes inherited from “MiniGameUnit” we created a “MiniGameUnitManager” class which provided one simple place to control all entities which we used in tandem with a “TextureManager” class, which stores all textures statically in one easily accessible place. The use of a TextureManager is one step towards achieving NFR_user_interactions as it will speed up the running of the game, reducing any processing delay between the user interactions and the game. This benefit of using these two classes is amplified by the fact that we can more easily clean everything up as the game is going on and at the end of an instance due to them all being stored together.

A UML diagram for the architecture of the minigame is found below[3]. Yellow classes denote ones found in NPStudios UML[2].

The other addition to the game was the implementation of the patrols and other levels the game needed to meet the requirements [1]. This addition allows us to meet the requirement UR_four_trucks which was updated to require 4 engines to be able to move around and attack the 6 fortresses. Implementing the 3 new levels brings the total number of levels up to 6 each of which represent 1 of the 6 fortresses we have implemented in the game. This, therefore, meets the requirement UR_six_ETs.

The other requirement this addition meets is FR_patrol_areas. The previous group had already created an alien class with a parameter for waypoints and a function to follow the path. We implemented an extra function that would generate a random patrol route within a given area. This area could be separately defined for each level.

Changes

The primary and largest change that was made to the codebase (excluding the addition of the minigame and the patrols) was the overhaul of the way that the map and the map collisions were handled.
For each level in the game, the previous team used an application called Tiled to create a tiled map. They then exported this tiled map as an image and rendered the image into the game. This was a simple and concise way to render the maps in the game. However, this approach to the map rendering led the team to have no easy way to program in the collisions for the map- that is, there was no way for the program to "know" whether a tile was collidable or not. To implement collisions with the map, the other team ended up creating a new object in the game for every collidable tile. This solution would have been time-consuming and led to large portions of code being used to create objects that matched the sometimes complicated collidable tiles on the maps.

Seeing as we needed to make three more maps for each new level, as well as a platformer minigame map for each existing level in the game, we would have to make a total of nine new maps for the game. We realised that individually programming in an object for each collidable tile for each map would be far more work than we could reasonably complete time, and so we decided to overhaul the way that the map collisions were handled.

In our new system, we exported the tiled maps from tiled as '.tmx' filetypes. LibGdx has inbuilt features to render tiled maps from these filetypes- the 'TiledMap' class and 'TiledMapRenderer' interface. Using these features, we could no only render the tiled map but could program the game to examine what tile was where, and whether or not this tile was collidable. We created an extra layer in each tiled map that consisted of black tiles (where we wanted the fire trucks to collide with the map) and white tiles (where we wanted the fire trucks to be able to drive freely). Then, we created an enum that stored the tile id of the white and black tile types, as well as whether or not those tiles were collidable. This allowed us to update the collision method so that it checked what tile the fire truck was going to move on, and prevent it from being able to move to a collidable tile.

This new system does not fundamentally change the way that the game meets requirements. However, because the new system allows for new levels to be added, with collisions, efficiently and concisely. Therefore we have implemented the requirement of having the fire truck correctly collide with parts of the map in a more favourable way than it was implemented previously. A diagram to show how MiniGameState and PlayState use the “TiledMap” class can be found in the bibliography.[4]

Other changes made were smaller things that allowed us more flexibility and better resource management such as the transformation of the Entity class. When we picked up the project it was based on 4 hardcoded attributes which were accessible through getters to which we decided to turn into a Sprite class which is a premade class by Libgdx which gives access to information about a particular texture. This allowed us to use a more powerful render function on our objects so in the minigame simple transformations such as reflecting the graphic of the fireman could be done without having to create a new graphic manually.

A full list of changes and additions can be found below

File Changed	Added/Changed	What?	Justification
Entity	Changed	Based on Sprite classes instead of 4 attributes	Sprite classes allow for more powerful libgdx render functions to be run
PlayState	Changed	Patrol system	Requirement UR_patrols
PlayState	Changed	Overhaul of the collision system. (removal of previous obstacle based system and implementation of tiled maps)	Collisions now based on map files (.tmx) which store the location of passable and impassable terrain
TiledGameMap	Added	New system	Reads and renders .tmx files as graphical maps in the correct position on the screen
TileType	Added	New system	Stores collisions for each map and calculates whether tiles are collidable or not Meets requirement FR_deny_collisions
PlayState	Changed	Runs to minigame when the level is complete	Requirement UR_minigame NFR_main_focus
MiniGameState	Added	Contains the workings of displaying the minigame	Calculates rendering the map and handling inputs from the user such as on-screen button pushing. Keeping architecture the same to keep FR_game_states
MiniGameUnit	Added	Inherited class from Unit. Designed to store information needed about minigame objects such as speed and gravity strength	Allows multiple methods such as move() to be combined into one class rather than the same code being copied in multiple
MiniGameUnitManager	Added	Contains the logical workings and manipulating the units on screen.	Splits up logical manipulation and graphical rendering over two different classes. Provides the

			advantage of being easily understandable and being a central location for multiple entities to be controlled
Bomb, Boss, Enemy, FireFighter	Added	Objects in minigame	Classes added to provide the functionality to the minigame. Each provides an element needed for it to operate
Heart	Added	Class added can have two states of full or off, represents how much health user has left	Indicator to the user how much health they have left in the minigame
TextureManager	Added	Class storing all textures the minigame needs to run	Stores all files statically in one class. Means only one texture per file will ever be loaded, increasing run speed and provides easy access to all files
LevelSelectState	Changed	Levels 4 - 6 added	Requirements UR_six_ETs FR_6_levels
PlayState	Changed	2 firetrucks added	Requirement UR_four_trucks

While most requirements were met, one which was not was UR_pause which required the ability to pause the game which was not implemented.

Bibliography

[1] Updated requirements
[2] Previous Architecture
[3] MiniGameUML
[4] Tiled Map UML