Top Right Corner Implementation Report

Game Executable: https://sepr-topright.github.io/SEPR/documentation/assessment4/game.zip
Requirements: https://sepr-topright.github.io/SEPR/documentation/assessment4/reqs.pdf

There were two main requirement changes for this assessment, increasing the maximum number of players to four, and adding an extra game phase where the player must capture the chancellor. This is a change to **requirements 1.1** and the introduction of **requirement 1.28**. Nearly all the other requirements were already satisfied by the code from previous assessments, however some changes and fixes were required. A version of the requirements in which changes (since assessment 3) have been highlighted in yellow and additions in green can be found here: https://sepr-topright.github.io/SEPR/documentation/assessment4/reqscoloured.pdf

Implementing the capture the chancellor mode (**req 1.28**) involved creating a new actor in GameScreenActors, and adding some methods in order to control its behaviour. The first step was to add an extra phase to the game, after the generate resource phase. To do this, an extra case was added to the switch part of the nextPhase() method. This phase lasts 15 seconds and the timeout animation is shown at the bottom of the screen. The chancellor object is an ImageButton which is moved across the screen. There are two methods, setChancellor() and removeChancellor() which will set the position of the chancellor ready to move onto the screen or off the screen and out of view. Within the GameScreen class there are two other methods, activateChancellor() and stopChancellor() that set the chancellor and change the music. Whilst the chancellor is set, he is moved consistently by the updateChancellor() method. Additionally, the next button is grayed out and disabled so that this phase cannot be accidentally skipped. It has also been disabled for the previous stage, resource generation. This is done by the enableNextBtn() and disableNextBtn() commands in the GameScreen class.

In order to increase the number of players a new variable, numberOfPlayers, was created along with a method to change it to any value (in the RoboticonQuest class). A SelectBox, numberOfPlayersDropDown, was created as an object on the start menu in order to initially set the number of players to 2, 3, or 4. The phase cycle works for any amount of players defined in the playerList variable. The GameOver screen had to be modified to show the scores of all players at the end of the game for **requirement 1.22**. To do this, extra if and else statements were added to the widgetUpdate() method in GameOverActors to draw the scores of players 3 and 4 only if the game is being played by 3 or 4 players. In order for this to work, the scoreCalc() method from the main game class (RoboticonQuest) had to be updated to properly calculate scores for all four players, rather than the usual two.

The code that we acquired from the previous exchange was missing some features that needed to be added or fixed. One thing that was missing was background textures for all screens. This meant implementing the code that renders the background by adding the textures to the Actor classes for each screen and then adding them to the stage. Each screen class has a resizeScreen() method which is used to scale the textures to whatever size screen is used, even though this version of the game is locked to a single resolution. This is slightly different for the main game screen, as the tilemap is rendered separately from the stage, and the background can obscure either the tilemap or the pop-up boxes, so for this screen the background is rendered separately as well.

The amounts of resources and money awarded by the random events were altered so as to balance the game. The textures for each event was also changed to make the game look nicer on the whole. The random events relate to **requirement 1.25.** In this code, the random events are implemented as their own screen, as opposed to the pop-up box style from the previous assessment. This is handled by the RoboticonRandomScreen class. To change the events, the textures themselves had to be edited as the graphics and text are part of an image. To change the actual resources awarded we only had to change some numbers in the execEv method, which uses a case/switch system to call player.event() with the correct parameters.

Resource generation (**req 1.16**) worked but wasn't easily testable using automated tests. The inherited Player.generateResources() method created an animation and added the resources to the player's inventory but didn't return the amount of resources given to the player (so we couldn't easily check that the correct quantities were generated as we can't seem to get tests to work with LibGDX - the library used to create the animation). So we exchanged it for the generateResources() method from our old code. The RoboticonQuest.generateResources() method also had to be exchanged so that it worked alongside this change as now Player.generateResources creates the resources and returns a data structure that lists the

resources that were generated with the animation now being created in the RobotioconQuest.generateResources method. This change allowed our tests to work correctly.

The code we inherited displayed uncustomized roboticons in the drop-down menu during the roboticon placing phase (**req 1.15**). This was undesirable, so we took the Player.getCustomisedRoboticonAmountList method from our old code and replaced all occurences of Player.getRoboticonAmountList with it (the only difference between the two methods is that getRoboticonAmounntList also returns the number of uncustomised roboticons and therefore they would appear as an entry in the drop down menu). This method uses an array of strings to represent the number of customised roboticons in a player's inventory. This change helps to prevent confusion when using the menu, as uncustomized roboticons cannot be placed.

Some fixes were needed for the roboticon market (**req 1.12, 1.14**). This was done by editing RoboticonMarketActors so that the objects are added to the actors table at the correct positions. This includes fixing the phase text to the top of the screen and moving it to the right, as it previously only showed the phase number and not the text that lets the player know what to do (and it was too far to the left and didn't line up with the text on other screens). The next button was moved to match the positions of the next buttons for the other screens. The colour of the text throughout the game was also reverted back to the default colour for the libGDX library (white) as the previous colour (pink) clashed with other elements of the GUI, this was done by changing the skin variable in the RoboticonQuest class back to the default skin that comes bundled with LibGDX. Additionally, the icon for the game was updated to match the chancellor image (this was done by adding statements to the main method in the DesktopLauncher class to add the icon to the windows title bar).

The code we inherited had its own resource market implemented but we decided against it and completely replaced it with our own design (from assessment 3), as we thought a different system was needed to properly meet **requirements 1.17, 1.18, 1.19, 1.20, 1.21 and also for 1.24.** We felt that the game flowed more smoothly if we merged the gambling phase into the market phase (and better met the requirements as these two features are meant to accessed together see **requirement 1.24**). The market is a single screen for both players, who must communicate and agree to use the market correctly. The market we inherited involved a separate screen for each player (even though the brief and requirements clearly state that players should access it simultaneously). With the game being designed to run on a single machine, there is no easy and efficient way to prevent one player from being able to access the other's' market screen as a form of sabotage, so there really is no benefit to having a separate screen for each player. The screen is implemented so that the player can buy or sell any resource type - using a drop-down select box and +/- buttons - to the market for the default rates (**1.19, 1.20**) or they can sell any resource - also using drop-down and +/- buttons - to the other player for any value, player to player sales were not supported in the market that we inherited (**1.18**). The market screen shows the inventories of all players and the prices for each resource. We did keep the the rock-paper-scissors gambling feature inherited rather than use the one we made in assessment 3 as we felt that the rock paper scissors game was more fun (**1.24**). The player can select how much money they want to gamble and then choose either rock, paper or scissors. We made small modifications to the MarketScreenActors class to use the rock paper scissors game. Essentially all code that implemented the simple gambling mini game that we implemented was removed and we simply placed an instance of the RoboticonMinigameActors class we inherited on the screen. Although we did have to make a few small changes to the RoboticonMiniGameActors class, we had to modify it so that the player was to gamble could be selected from a drop down menu (this is because previously each player accessed the gambling screen in turn so the current player was used instead). This involved creating a new private method createPlayerDropDown() and insisting that all the other code used the selected player rather than the "current player" from the RoboticonQuest class (which would always be the last player as the market phase is accessed after all players have had their turn in a given round).