Architecture Overview

Overall, we as a team have made two types of changes in regards to the implementation of Kroy. The first group of changes relate to having the requirement of implementing the full project brief (e.g adding patrols, making the station destroyable). The other group of changes relate to changing features, adding additional ones, or fixing bugs to better fit our updated requirements (e.g stopping camera glitches, fixing memory leaks, and having the ability to control many different types of firetrucks instead of being stuck on a single type for the entire game). Chronologically, we had two crucial tasks to complete when we first inherited the project.

The first task was to read over their Assessment 2 requirements, and determine whether requirements needed adding or updating to ensure our Assessment 3 code meets what was outlined with the project brief. While these are already previously discussed in other sections, it is important to note this in the architecture overview because it shows our thought process deciding what we needed to code to complete the game. Direct examples are that SFR_PATROL_FIRESTATION is a requirement that states ET Patrols should be able to destroy the firestation: a requirement that was omitted from the previous group's Assessment 2 requirements but is something we knew we needed to implement for Assessment 3. Another requirement I will discuss later is UR_FUN, the game needed to be fun to play and this shaped some of our design decisions when it came to our implementation of Kroy.

The second task was to fix a critical bug that stopped us working on the game. The game had an issue which caused a memory leak, and the game would crash often.We didn't want to work on an unstable version of the game because this would have made testing our code a harder task, so we fixed that before we began implementing our new code. The issue itself was caused by creating "Debug" objects every frame regardless of whether the debug option was enabled, but not disposing of them when debug was set to off. The simple fix for this was to check if debug is enabled before any relevant debug method, which the previous group did not do. i.e `if (Kroy.debug)`

We also changed some methods and attributes in our Kroy class to be static, such as mainGameScreen. This was necessary for us to implement both jUnit and Mockito. Thanks to this, we were able to continuously test our game as we implemented it to ensure it worked as expected.

The additions and changes we have made are in the table below. Which requirements they relate to are also shown.

Requirements	Relevant Code	Discussion
UR_ET_IMPROVEMENT	`((Entity) nearestEnemy).applyDamage((float) (flowRate * Math.max(0.5, GameScreen.gameTimer * (1/600))));  //Applies damage to the nearest enemy`	To ensure the game becomes harder overtime, we've implemented a damage multiplier. As time reaches 5 minutes, a fire truck's damage decreases down to a maximum of 0.5x.

	Line 229 in method playerFire in class FireTruck	Since GameScreen already has a timer that we use for checking when we should destroy the fire station, we've made this variable static so is usable in FireTruck, instead of implementing an additional 5 minute timer in the FireTruck class which would duplicate code.

UR_PATROL
SFR_PATROL_DAMAGE
SFR_PATROL_HEALTH
SFR_PATROL_DIFFICULTY
SFR_PATROL_FIRESTATION
	```if (Kroy.mainGameScreen.gameTimer <= 0) { //Once timer is over applyDamage(100); //Destroy fire station }```
	
	Lines 52-54 in method update in class FireStation
	
	```//we should spawn a patrol near every fortress if it given it's been 10 secs. if (lastPatrol >= patrolUpdateRate) { lastPatrol = 0; for (Vector2 position: patrolPositions) { //Randomize the positions a little bit float oldX = position.x; float oldY = position.y; float randX = (float) (oldX - 400 + Math.random() * 400); float randY = (float) (oldY - 400 + Math.random() * 400); gameObjects.add(new UFO(new Vector2(randX, randY))); } }```
	
	Lines 280-297 in method updateLoop in class GameScreen
	
	The entire UFO class
		Patrols have been implemented as described in the brief. When the game does its standard render loop, now also checks whether it's been 30 seconds since we last spawned patrols. If it has been 30 seconds, then for every living fortress, it will spawn a patrol with a random location (but very close to that fortress). The benefits of putting this in the updateLoop is that since we have to iterate through every entity in the game each frame, we may as well do the patrol checks in this loop instead of perhaps creating another method that checks if it's been 30 seconds, loops through every entity *again*, and then spawns the patrols (unnecessary additional array accesses). The patrols will fly around in a predetermined pattern, and will shoot at the firetruck the player is controlling. The player can also destroy any nearby patrols. Additionally, we also destroy the station after 5 minutes, even if this isn't explicit in the code that this is handled by the UFO class, it is implied by the game so it fits the requirement.

UR_FORTRESSES	```private void fortressInit(int num) {```	We decided to create 6 different types of firetrucks,

SFR_FIRETRUCKS_SELECTION	gameObjects.add(new Fortress(fortressPositions.get(num), textures.getFortress(num), textures.getDeadFortress(num), fortressSizes.get(num)));
	}
	private void firetruckInit(float x, float y, int num) {
	firetrucks.add(new FireTruck(new Vector2(x, y), truckStats[num], num));
	}
	// Initialises the FireTrucks
	for (int i = 0; i < 6; i++) {
	firetruckInit(spawnPosition.x + -50 + (((i + 1) % 3) * 50), spawnPosition.y - ((i % 2) * 50), i);
	fortressInit(i);
	}
	
	Lines 159-162, 169-171, 144-147 in method show() in method GameScreen
		as well as wrote code to meet the requirement of needing 6 fortresses. Instead of writing out the same line each time (e.g new FireTruck(1), new FireTruck(2), ect.) we have all the firetruck and fortress stats in their own arrays, which means we can simply iterate 6 times and instantiate each new firetruck and fortress since they'll have unique stats in each index of their respective array.

UR_DRIVE
UR_FUN
UR_FIRETRUCKS_UNIQUE_SPEC
UR_FIRETRUCK_MIN_START
SFR_FIRETRUCKS_STATS
SFR_FIRETRUCKS_SELECTION
	private Float[][] truckStats = {
	//Each list is a configuration of a specific truck. {speed, flowRate, capacity, range}
	{400f, 1f, 400f, 300f},
	//Speed
	{350f, 1.25f, 400f, 300f},
	//Speed + Flow rate
	{300f, 1.5f, 400f, 300f},
	//Flow rate
	{300f, 1f, 450f, 400f},
	//Capacity + Range
	{300f, 1f, 500f, 300f},
	//Capacity
	{300f, 1f, 400f, 450f},
	//Range
	};
	
	public void updateLives() {
	if (lives>1) {
	lives -= 1;
	if(firetrucks.get(0).isAlive()) {
	switchTrucks(0);
	}else if(firetrucks.get(1).isAlive()) {
	switchTrucks(1);
	}else if(firetrucks.get(2).isAlive()) {
	switchTrucks(2);
	}else if(firetrucks.get(3).isAlive()) {
	switchTrucks(3);
		We have changed a core functionality of how the game works. Initially, you could only pick between 4 different classes of firetrucks. Once an option was selected, you were stuck with this type of firetruck (with very specific stats) for the entirety of the game. We decided that it may be more fun that not only should you have 6 unique firetrucks instead of 4, but you also have the ability to switch between these trucks at any time during the game (as long as the truck you'd like to play hasn't been destroyed). This wasn't too hard to implement because we had chosen DicyCat given they were a real-time based shooter like our original game, and we were able to reuse similar code that we had worked in in Assessment 2 since we had

	}else if(firetrucks.get(4).isAlive()) {
	switchTrucks(4);
	}else if(firetrucks.get(5).isAlive()) {
	switchTrucks(5);
	}
	} else {
	gameOver(false);
	}
	}
	
	Lines 66-73, 76-78, 559-610 in class GameScreen (some code omitted), 207-259 in class MenuScreen
		a similar solution. We now meet all the requirements needed to meet our brief in regards to firetrucks.

UR_MINIGAME	// Creates a task to generate pipes
	Timer.*schedule*(new Task() {
	@Override
	public void run() {
	createPipe();
	}
	}, 0, 2);// 0 seconds delay, 2 seconds between pipes
	}
	pipes.forEach(o -> {
	o.update();
	o.render(batch);
	if (o.gameEnd(*player*)) {
	// If the player hits a pipe, then the game ends
	gameOver();
	}
	;
	});
	pipes.removeIf(o -> o.isRemove());
	
	The entirety of Goose, Pipe, and MinigameScreen classes. (example code shown for MinigameScreen)
		The requirement we were given was very vague. We were told to make a game similar to “super mario or flappy bird”, so we went with the second option. Since game screens are all seperate classes, we could create MinigameScreen without having to change code anywhere else (except adding the option in MenuScreen and Kroy). The game thematically fits an alien + York theme, since we’ve replaced the bird with a goose and pipes with asteroids. The game isn’t relevant to the main game however, and your score in the minigame has no effect on your score in the main game.