Recommendations for extensions to the game engine

by Tute04Team160

While we were working throughout the entire assignment, there were problems we encountered that we thought could be somewhat improved on.

Problem 1: The map is too large for starting implementation.

Although the assignment is supposed to model an actual “rogue” game design process via object-oriented design, we can’t help but feel like the given game map might be too big for the coding and testing process.

The size of the given map is 25 x 80 characters - this alone is quite a huge map. And from the given requirements in the assignment, the way the dinosaurs move and how the fruits bear their fruits are totally random. Mixing both of these factors turns it into a game where testing certain functionalities may be quite difficult. On top of that, as the map is too big, there might be a lot of errors that are in the game, still undetected.

A recommendation to fix this is to introduce a smaller map at the start. Say, probably 12 x 20 characters. That way, it’s easier to test out certain functionalities – a quick example would be, as the Brachiosaurs and Stegosaurs can walk into each other more frequently or walk into a tree or bush to feed on the fruits. Or the Allosaurs are able to meet the Brachiosaurs more often and hence we can test out if the Allosaur is attacking correctly, whether said Brachiosaur turns into a dinosaur corpse correctly, and will the Allosaur feed on said corpse correctly. Then the game map can be resized to the intended figures.

Problem 2: Climate class?

At the start of the 3rd part of the assignment, we have been assigned to implement lakes, thirst levels of the dinosaurs, and also rain. While this adds another way to kill of the dinosaurs (since, as of at the end of the 2nd assignment, the only few ways for the dinosaurs to die are: getting attacked by an Allosaur, the dinosaurs could not find food before they turn unconscious, or getting hit by the player’s laser gun), we feel like there could have been a better way to implement the climate.

A recommendation for this implementation would be leaving a class to manage the climate of the game, instead of leaving up to the Sky class to randomly rain. Another recommendation for the implementation of climates is to introduce more kinds of climates, such as hail or dust, where in these climates, the dinosaurs may take a certain amount of damage each turn, and the plants (trees and bushes) may not be able to grow and bear fruits at the usual rate. But given that we only have a short timeframe to implement everything, on top of difficult schedules, we'll say this should only be something extra to add.

Problem 3: Map IDs

We're also tasked to create a second map in addition to the current map we were given. While adding a map isn't a difficult task, we quickly realized that there is no reliable way of switching the maps. The problem comes from the GameMap class file (which is given in the engine package, meaning that we were not allowed to edit any of it), where there is no ID attribute. This isn't an issue as we only have two maps where the second map is at north of the first given map, and we could still implement in a way that said map will switch to the other when the player approaches the top-most border.

However, what if we want to, for example, implement multiple maps in the future, or a way for the player to teleport to a certain map if they wish, it would be easier to switch to said map with their given ID. Hence, a recommendation here would be having an ID attribute for each GameMap instance for easier reference.

Problem 4: Not exactly adhering to some SOLID principles

We have also found out that the engine package doesn't exactly adhere to some SOLID principles. Although the game engine does what it's supposed to do perfectly for a simple rogue game, we realized that some parts of the codes could have been done better.

Take the Actor base class for example, which is inherited by Player and Dinosaur. Notice that the Actor class has a list of items called Inventory as one of its attributes – this means that the classes that inherits this file has said list of items as well. It's not really a problem until we realize that the Dinosaurs might not need this list even if they have it. This is a violation of the Liskov Substitution Principle ("L" in SOLID). This could have been solved if we remove all methods and attributes regarding Inventory and separately create a class for Inventory and only let the Player (or any Actor that does need Inventory… for whatever reason) inherit

said class. This way, the Dinosaurs wouldn't have an Inventory and the Liskov Substitution Principle wouldn't be violated.

A few interface files given like GroundInterface in the interface package, is also not adhering to some SOLID principles as well. The GroundInterface has a few methods that would make sense for a few classes that inherit said interface but wouldn't for the other classes. For example, getFruits() and removeFruits() would make sense for the Tree or Bush classes since they have the ability to bear a fruit, but these methods wouldn't be used by Lake and Sky classes even if inherited, vice versa for the remaining methods. This is a violation of the Interface Segregation Principle ("I" in SOLID) as they could have separated getFruits and removeFruits() to a new interface, and the remaining methods for another new interface like OtherGroundInterface. Similarly, this also violates the Single Responsibility Principle ("S" in SOLID) as this interface is handling more than one responsibility.

But what have we learnt from these assignments?

As we are reaching the end of the unit and its assignments, we can agree that the most important takeaway from this unit is object-oriented design (OOD) is a very useful process to create software projects.

From the start of the assignment, the use of UML class diagrams and sequence diagrams allows us to grab a hold of how the project should function. Although it can be quite a confusing process as we were forced to brainstorm what kind of methods do we need to complete a certain functionality with some assistance from the given base code, it has proven to be very helpful to us during the process of coding out the classes and methods as now we have a draft of which classes or methods do we need to use.

Next is the use of some OOD principles such as SOLID. With the use of abstract and interface classes, not only we are able to cut down a significant amount of code, but also easier to keep track of bugs. One common instance of this during our implementation process was the use of extending the Dinosaur base class across four kinds of Dinosaurs (Allosaur, Brachiosaur, Pterodactyl, and Stegosaur), and also the use of extending the Ground base class across a few other classes like Tree, Bush, Lake, and Sky.

Besides those, we were also able to implement so that the dinosaurs (other than Pterodactyls) cannot pass through certain Ground objects just by modifying their capabilities in the base Dinosaur class. This is extremely useful in real game design environments, as there could be a lot of objects in the game that share a fair amount of code, so the game developers might as well let the objects inherit from a base class. That way, they can debug the objects quicker if something goes wrong.