Hyunsirk Choi

CSC 481

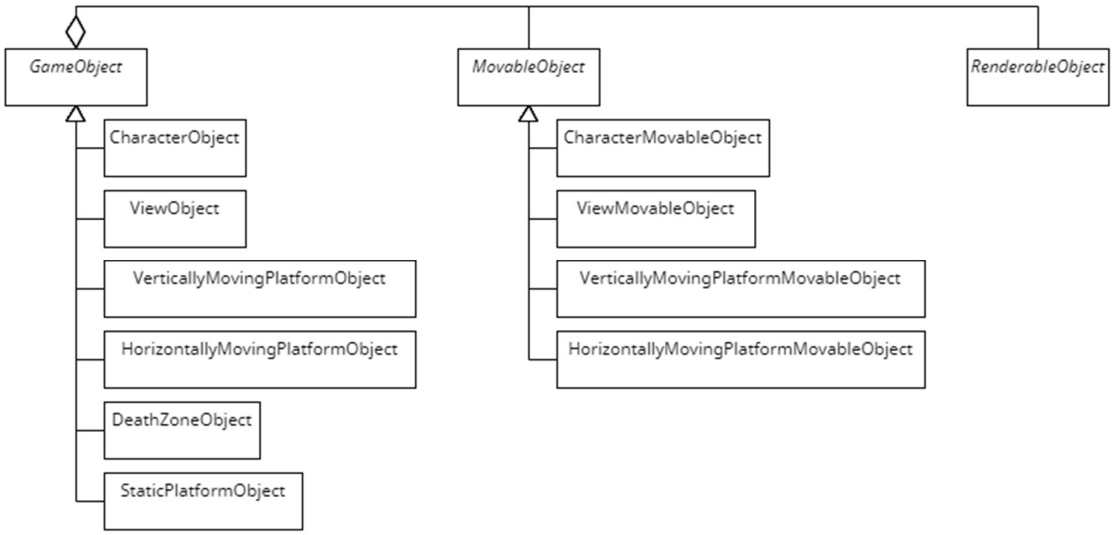
Dr. David Roberts

26 Oct 2022

Homework 3 Writeup

Section 1: Game Object Model

I have chosen to use the Rigid Component Model for my game object model. The simple relationships between classes are shown below:



As there are different variants of `MovableObject`, each child of `GameObject` will have an appropriate `MovableObject` or not at all. The `MovableObject` has a pure virtual function named `run()` that the inheriting variant class will define its behavior depending on the type of the `GameObject`. For example, the `CharacterObject` will have `CharacterMovableObject` as one of its fields and `CharacterMovableObject::run()` is defined to update the character position based on the keyboard inputs. Similarly, `VerticallyMovingPlatform` and `VerticallyMovingPlatform` objects will have their own `MovableObject` that has their own defined `run()` function for its intended behavior. While there are several variants of `MovableObject`, `RenderableObject` is a stand-alone class that only has a `render(sf::RenderWindow *window)` function that behaves exactly the same amongst all `GameObject`s; however, while `ViewObject` and `DeathZoneObject` also have defined `RenderableObject` field, both objects can set the `RenderableObject` as `NULL` since both objects are never rendered. I have defined the `RenderableObject` field just for visual representation reasons for placements.

There are statically defined constant type values in the `GameObject` for

each existing child class as shown below:

```
const std::string GameObject::CHARACTER_OBJECT_TYPE = "character";
const std::string GameObject::STATIC_PLATFORM_OBJECT_TYPE = "staticPlatform";
const std::string GameObject::VERTICALLY_MOVING_PLATFORM_OBJECT_TYPE = "verticallyMovingPlatform";
const std::string GameObject::HORIZONTALLY_MOVING_PLATFORM_OBJECT_TYPE = "horizontallyMovingPlatform";
const std::string GameObject::DEATH_ZONE_OBJECT_TYPE = "deathZone";
const std::string GameObject::VIEW_OBJECT_TYPE = "view";
```

The `GameObject` has a `std::string type` field which is used to indicate the specific type of a generic `GameObject` instance while using polymorphism to create a generic object list. This `type` field is useful as the `main` thread has a cluster of `GameObject` lists that needs to identify like `CharacterObject` to feed keyboard input or detect collisions with any platform objects.

The purpose of the `ViewObject` is to attach to a `CharacterObject` and update the `sf::RenderWindow window` view as the `CharacterObject` tries to move around the environment.

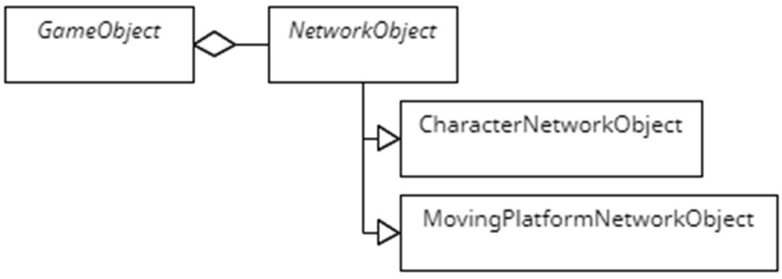
The screenshots show a `CharacterObject` on a `StaticPlatformObject` with an attached `ViewObject`. As shown on the right screenshot, a `CharacterObject` is able to freely move around the `ViewObject` boundary without updating the view; however, as the `CharacterObject` tries to escape the `ViewObject` boundary, the `ViewObject` will follow around the attached object while updating the view so the `ViewObject` is always centered to the screen. So as `CharacterObject` moves a certain distance, the `sf::RenderWindow window` will side scroll so the `CharacterObject` is somewhat centered. As the view updates to center the `ViewObject`, the attached `CharacterObject` is not always completely centered.

As there is only `DeathZoneObject` and no such thing as like `SpawnPointObject`, it is because the `DeathZoneObject` has a separate spawn point where a CharacterObject will respawn as soon as there is a collision with the actual death zone.

The screenshot shows a `CharacterObject` on a `StaticPlatformObject` with other 2 colored blocks. The green block is the spawn point and the red block is the death zone. As soon as the `CharacterObject` object collides with the red block, the `CharacterObject` will be respawned on the green block dropping to the ground. While this is just an example, both spawn point and death zone are not rendered. The death zone covers below all `StaticPlatformObject` which imitates death on fall into the void.

Section 2: Multithreaded, Network Scene

As the game server deals with all of the interactions and events, the player client simply receives the game scene from the server and updates each `GameObject` position. To simplify the process of generating each `GameObject` state as `std::string` and updating the `GameObject` based on the network message, I have added an additional field to the `GameObject` and the relationship is shown below:



The new `NetworkObject` is similar to the `MovableObject` in terms of having different variants based on the `GameObject::type`. There are two pure virtual functions `receiveNetworkMessage(std::string message)` and `createNetworkMessage()`. Each function is used in the player client and the

game server to receive and generate network messages for the corresponding `GameObject`. There are only two variants of the `NetworkObject` as there are only two kinds of moving objects. While the CharacterObject has its own variant, the `VerticallyMovingPlatform` and `HorizontallyMovingPlatform` are both moving platforms in which the network message format will be identical.

While there is also `ViewObject` which is movable by attaching to an object, the `ViewObject` will be created and updated locally by the player client to track the client's own `CharacterObject` rather than following other players.

The game server generates network messages based on the 3 existing moving platforms and all existing players. As the 3 moving platform is updated by the game server and will never be deleted, these network messages come first, and then the players' state. As the players join and disconnect from the game server, the game server's network message will only include active players. On the other hand, the player client simply sends its player index/number and its local keyboard input to the game server. The player client receives the latest game scene network message from the game server and updates all moving `GameObject`s.