CSC591 (Game Engine Foundations) Homework 3

Homework Spec

The homework spec can be found in `[root_dir]\docs\HW3.pdf`. Below is a brief excerpt from the same:

Your task for this assignment is to add more functionality to your Processing engine. Specifically, you will focus on the addition of an event system and replays, which will require you to include a notion of game time in your engine system.

Introduction

I have been implementing my assignments on top of each other. For this one, I had to make substantial change on how my network architecture works. That being said, I saw some pros and cons to a strictly game object approach or using a event based approach which I will talk about in the following sections. As before, I will give a brief description on how to run my program and then describe the individual features asked for in the homework specification.

Running my program

There are two ways to run my multiplayer game / engine demo:

1. **JAR**:
    1. Find the JAR file for this project in `[root_dir]\out\artifacts\multiplayer_game_engine_jar\multiplayer-game-engine.jar`.
    2. Open a command line and type `java -jar multiplayer-game-engine.jar s` (for server).
    3. For running clients, open other command lines, type and execute `java -jar multiplayer-game-engine.jar c` as many times you want for any number of clients.
    4. Remember that you need the run the server first and then the clients, otherwise this might throw some exception. This should be normal, as for most multiplayer games, the headless server generally is always running.
    5. Also currently, the client searches for a running server in `localhost`, so running the server and client in different computers will not work. If you still want to run it in different computers, follow my second way of running the program and before building it, open `Constants.java` and assign the server's IP to the `SERVER_ADDRESS` String variable.
2. **IntelliJ**:
    1. Install IntelliJ Community Edition.
    2. Import and build my project.
    3. There should be two run configurations - one for the server and one for the client. Run the "Server" first and then the "Client". The shortcut for running programs in IntelliJ is `Alt + Shift + F10`.
    4. If you don't find the run configurations, make two yourself. For the sever, give a command line argument of `s` and for the client, give a command line argument of `c` (without the quotes).

The game/playground that I built using my engine has the following features:

1. The server is completely headless, and you won't be able to do anything with it. But as I am using Processing, it is not possible to compeletely make it headless, i.e. to make the display screen invisible. So I have (albeit a little hacky-ly) just reduced the window

size for the server process to a mere 10x10, and nothing gets displayed there, thus almost giving it the headless feel.
2. When you start the client(s), you will see the same `FallingStairs` (color and position) coming on their screen as well across all clients. These are dynamic game objects - stairs which come from the top and disappear below the screen.
3. There will be some `StandingStairs` as well which are basically static game objects. These are all sent from the server.
4. On the client screen(s), you can move the client square around using `A`, `D` and `SPACEBAR` and play the game.
5. Open multiple client windows and close any you want - the engine will properly show squares for only clients that are alive.
6. The controls for recording are shown on the screen. Please use them in order, because otherwise they might produce undefined behavior.
7. First press `R` to start recording.
8. Then, press `H` to stop recording.
9. You then have three options to choose from for the playback. You can choose `N` for normal playback, `F` for faster playback (double speed) and `L` for slower playback (half speed).
10. Once you have played the recording, you can record and play again, but there is no way to access the last played recording, i.e. the replays are not permanently saved. They all have a lifespan of one playback.

Screencast

Before describing the individual features, I would like to let you know that I have uploaded a screencast to YouTube so that it's easier for you to check what I've done. The demo shows three clients and shows all the three replay modes - slow (L), fast (F) and normal (N) one by one.

https://www.youtube.com/watch?v=PwNJ7zd9ZhE&feature=youtu.be

Timeline

My timeline class holds the concepts of anchoring, tic size and different iteration types. In total, I have five different timeline instances:

1. `realTimelineInMillis`: This holds the actual real time in milliseconds. The anchor in this case is a null timeline, i.e. the time held in this timeline should not be offset at all. The tic size is `1,000,000`.
2. `gameTimelineInMillis`: This holds the game time from the start of the game. All events use this timeline. The tic size is `1,000,000` and the anchor is `realTimelineInMillis`.
3. `totalTimelineInFrames`: This holds the total framecount. Tic size is `1`. Anchor is null timeline.
4. `gameTimelineInframes`: This holds the framecount from the start of the game. Tic size is 1. Anchor is `totalTimelineInFrames`.
5. `replayTimelineInFrames`: This is maintained by the `EventManager` class. The anchor is `gameTimelineInFrames` from when the replay playback has been started. Tic size is variable and can be one of `0.5`, `1` or `2` based on the playback speed.

The times for each of the timelines are noted from `System.nanotime` for the real time based timelines and `PApplet.frameCount` for the frame loop based timelines. These timelines have been enough for me to implement events and replays.

Event Management System

The event management system demanded a sea change in my architecture. Previously, I was sending game objects from/to the server/client. These game objects were passed on to the game controller (`SimpleRaceManager`) in my case and then it was its duty to properly see what needs to done with each of them. For things coming from the server itself, it knew it was probably `StandingStair` or `FallingStair` and added them to proper lists so that other threads can use it properly and update the game objects represented by them. For the things that were broadcasted from the server and were actually coming from other people, it knew that it must be the

`Player` locations and accordingly put them in a hashmap keyed by their connection IDs. This was also passed to the `MainEngine` and it would update and draw these game objects as well.

Now, some of the architecture remains the same for the event system as well, but the core idea of sending game objects has been replaced by sending and receiving events instead. This has been pretty complicated based on the nature of my game as well. First, I will give a brief overview on the individual stages in my event management system and then move on to my own take on the pros and cons of this system.

1. **Registration**: For registration, I did not use inheritance, but a simpler and more effective approach of having an interface of `EventHandler` in the engine. The game has to implement this interface and it has one method of `onEvent(Event event)`. It implements this method and then as required, builds other methods for handling specific methods. The controller class has a `getEventHandler` method which responds back with the proper implementation of the `EventHandler` to the `EventManager` class so that events can be handled. Having no inheritance mechanisms, event handler registration is very easy. For event type registration, it has been done once manually at the start of the event management system. The engine provides some default events which are registered automatically. The rest have to be registered by the game controller.
2. **Raising**: The events are raised by just calling the `raiseEvent` method in the `EventManager` class. Each event has the following characteristics which make them unique and also serializable (to be able to send across teh network):

    1. `String eventType`: Everything in my game now is driven by events and nothing is achieved by transporting game objects across. That has resulted in quite some number of event types. The event types provided by the engine are:

        1. `USER_INPUT`: Any kind of keypresses are notified through this. Some keypresses which are related to replays are handled by the engine before passing them on to the game. They ultimately get passed on to the `Player` class from the controller.
        2. `RECORD_START`: To start the recording. A snapshot of the game objects cluster is taken and the `EventManager` starts recording all events hence raised in a parallel queue.
        3. `RECORD_STOP`: To stop the recording. `EventManager` simply stops recording events to the event queue from now on.
        4. `RECORD_PLAY`: To play the recording. Based on the speed requested, the playback starts. I'll come to this later in the **Replays** section.
        5. `PLAYER_DISCONNECT`: Some player has disconnected, so his traces should be removed frm game objects and event queues.

        The event types from the game's perspective are:

        1. `PLAYER_DEATH`: Player has collided with a death stair. It is regenerated from a `SpawnPoint`.
        2. `PLAYER_COLLISION`: Player has collided with a stair. The collision response code now comes into the event handler.
        3. `STAIR_SPAWN`: Either a `StandingStair` or a `FallingStair` has been spawned by the server. A new game object is created as a result and added is registered with the engine, this basically means adding to the game object cluster.
        4. `PLAYER_SPAWN`: A new player has joined. Every other player is regenerated to simplify confusions.
        5. `NULL`: Needed for CMB. Will come to this later.

    This is also required to maintain **Event Priorities**. When the event types are registered, any one of the `LOW`, `HIGH` or `MED` priorities are chosen. Then when the events are raised this is how each event can surpass another in the event queue even though it was raised earlier.

    1. `List<Object> eventParameters`: All the event parameters that are needed are bundled into this list and kept in the event class. This is how most of the information regarding the events are passed across the network.
    2. `String timelineName`: This keeps the timeline name to which the event adheres to.
    3. `int connectionID`: Contains where the event comes from. The server has a connection ID with everybody as `-1`.
    4. `float time`: Denotes when it was raised and will be dependent on the timeline.
    5. `boolean backup`: Sometimes, when new players join in, certain events which were already handled by others, should be sent to them, but maybe not all. For example, the spawning of the `StandingStairs` and other `Players` should be sent to all newly joined `Players`. This boolean holds whether this event will need backup or not, to send to when other new `Players` join.
    6. `float frame`: This is needed for replays.

Most of this information is passed on to the constructor and then it creates an event accordingly. In the `raiseEvent` method two things generally happen based on whether the current running entity is a server or a client. If it is coming from server, then it is broadcasted to all other clients and that's it. **NO** events are **handled** on the server. This actually makes it a little difficult to synchronize events. If the events were handled on the server and then just responses sent back to the client, then most of your synchronization is already done, but not in this case. If the event is coming from a client, then it's time and priority are both taken into account to form a `score` and then it is put in a proper queue (`eventQueue` for each client connection) inside a timeline map (if multiple timelines are needed, currently all events other than replays are measured with the `gameTimelineInMillis` , so there is one entry in the map), based on where it originally came from. This queue is a priority queue with an appropriate `EventComparator` . So the events get sorted properly and the most prioritized event is at the top, ready to be handled. Also, all these events that need to sent to others are put in another separate queue, which is kind of like a bag of tasks. The `GameServer` or the `GameClient` thread consumes from this queue and the `EventManager` thread keeps filling this up. One `waits` on the queue monitor when it's empty and another `notifies` after filling the queue with an event.

3. **Handling**: The events are handled based on their priorities. The `MainEngine` keeps calling `handleEvents` method in the `EventManager` class in every loop. This one polls events from the `eventQueue` for each client connection and then with the help of CMB algorithm, handles it. To handle, it simply has to call the `onEvent` for the event handler that has been registered with the engine class. The rest is taken care by the game where it does a switch case on the event type and delegates handling of different events to different methods. The replays are handled a little differently, which I will come to later.

   I have faced quite some problems while implementing this architecture. From a network perspective, previously all game objects were passed on to the controller and it did whatever it deemed proper. But now, the network server or client after receiving events, simply raises them. Rest of the job is taken care by the well defined event management system. There is a `toBroadcast` boolean function argument in the `raiseEvent` method so that the system does not get caught into an event deadlock. Thus, events received at the client end from other people are never broadcasted, and things received at the server end are always broadcasted. On the other hand, events generated at the client end **ARE** broadcasted. Building this system considering everything works seamlessly was a little difficult, but once it got done, I think this created a much more flexible and easy-to-plug-in system than sending game objects. Because, I can now simply create a new event type and raise it from the game's perspective and then put a function to handle that event in the event handler. And that's it. The rest will be taken care by the engine. Previously, after receiving game objects, I had to do a lot of conditional checks here and there to deal with them and it was not scalable.

Replays

Replays fit in almost perfectly to my event management system. But for some cases, I faced some problems. I will first give a sequence of steps which my replay system does to achieve this:

1. When the `RECORD_START` event is handled, a snapshot of the entire game objects cluster is taken. This snapshot is basically serializing the cluster into a `ByteArrayOutputStream` . Thus the snapshot resides in the memory. This is done once, but at the same time a flag is toggled in the `EventManager` which allows it to record every event in a parallel event queue just before handling those events. At this time, I also store the start of the recording as the time received from the `gameTimelineInFrames` so that I can offset that from all other events during replays.
2. When the `RECORD_STOP` event is handled, the flag in the `EventManager` is toggled again, so as to stop storing the events into the parallel queue. The replay events are **NOT** stored, as they might form deadlocks.
3. When the `RECORD_PLAY` event is handled, at the engine level, all the current game objects are stored in a separate map as a backup. This backup will be restored when the replay has finished. Note that no serialization is done in this case. This is because, when I was taking a snapshot, I needed a deep copy of the game objects, not just a reference copy. So that when I deserialize the snapshot, it should give me the exact values of the individual game objects at that stage. If I hadn't serialized and had just stored the reference of the cluster, then all the values would have changed throughout the recording process anyway and that wouldn't have served my purpose. Next, the current `eventQueue` is backed up and replaced with the recorded event queue. It was not required to serialize the event queue, because the individual events are never changed, whereas in the case of game objects they are changed in every frame by their `update` method. With the old game object snapshot and the recorded event queue in place, the replaying ideally needs nothing more, as the game update loop would update the game object positions and the events would keep getting handled. But one of the most important thing here is the time at which the events are raised and it is much more difficult than it

sounds. For a game where nothing is grid-based or move-based and very continuous in nature, replaying very accurately is a pain. Moreover, my events are more practical - user input, player collision, etc. I don't have an event for every position pixel change, because that does not sound realistic to me. So basically, I have events which determine how a state would be reached for a particular game object. The game object itself will, based on the information from the event, reach that state by updating itself accordingly. I thought that's how events should be, more like being the reasons for change instead of the change itself. Thus event handling also has a fair segregation in terms of functionality. That being said, it was very crucial for me that once I handle an event, I handle the next event exactly after say `x` time and this `x` should also have been the time between these two events when they were originally raised when the recording was going on. This seems easy and I thought so too. Hence, I just made by `handleEvent` thread sleep for `E2 - E1` milliseconds, assuming that it will wake up and do the handling in the proper time (my event handling was being done in a separate thread earlier). But this infused too much uncertainity that I hadn't predicted before. Sometimes the game update thread would update the game objects a little more or less than what's necessary and things won't look the same in the replay. For this, I chucked sleeping `E2.time - E1.time` milliseconds and starting storing the `frameCount` of when that event was going to be handled in each event. This is when I introduced the `replayTimelineInFrames` timeline. Then I used to sleep the `handleEvent` thread for `FPS` number of milliseconds which was equal to a hardcoded value of say `50`, thus allowing the game update loop to update the game objects in that time. This **ALMOST** fixed the problem. Next, I had started to understand that having the `handleEvent` thread as a separate thread is pointless and is creating more havoc than efficiency. Thus, I started calling the `handleEvent` method from the update loop instead of allotting a new thread for it. And that almost guaranteed that no frames will be skipped in the replay. Because they are all in the same thread, I check in my handle event method, whether the `E2.frame - E1.frame` is equal to or less the time shown by the `replayTimelineInFrames`. If so, I handle the events and otherwise I just skip handling any event.

For speed in replays, my `replayTimelineInFrames` is initialized with a tic size of `1`, `2` or `0.5` for `NORMAL`, `SLOW` and `FAST` replay methods. Accordingly, all game objects `update` methods take in a `frameTicSize` parameter. In cases of `NORMAL`, `SLOW` and `FAST` replays, the values of `1`, `0.5` and `2` are passed to these methods. Thus, all game objects update slowly or fast and the anchor time to which all the events are checked in the replays also moves in the same speed.

No events are allowed to be raised during the time the event replay is happening. As all events required for the events to properly show themselves, are present in the recorded event queue. If a movement user input event ultimately resulted in a player collision method, the latter will be present in the recorded event queue and no need to raise it while playing the replays.

CMB

Though event synchronization was working almost properly before applying CMB, it now provably works perfectly. To implement CMB, I have kept different event queues for every client connection and if any one of the queue is empty, then I don't handle any events. The event which has the lowest timestamp/score among all the queues will be handled first and when any event queue gets empty, I will exit the event handling. I did not need to do any change specifically for the replay system to make this work. It is obvious that if there are no events coming from a client, then the handling of events for this client also will completely stop. This is impractical. So to make it work, as we learned in class, I send `NULL` events from every client at every frame. It **NEEDS** to be done in every frame, because otherwise there will be plenty of cases when the user will press something and the result won't be shown on the screen for some time. Even if that time is very less, it is not a desirable thing at all. Anyway, with sending `NULL` events at every frame fixes this, it undoubtedly causes a lot of network overhead. This is impractical as well. So I doubt that any popular game engine or game uses CMB for event synchronization. The CMB algorithm is implemented in the `whichEvent` method in my `EventManager` class, if you want to have a look.

Appendix

Not needed as the screencast shows everything.