Project Documentation For M14 - Vehicles & DB & Net

Author: Mason Fraser
Date: 08-Nov-19

Table of Contents

1. Project Overview 3
2. Project Requirements 3
3. Database Implementation 4
3.1. Persistent Database 4
3.1.1. Saving the Data 4
3.1.2. Loading Data 5
4. 3-Tier Model 6
4.1. Host Application 6
4.2. Server 7
4.3. Client 8
5. Diagrams 9
5.1. Class Diagram 9
5.2. Sequence Diagram 10
6. Testing 11
6.1. 3d Application running 11
6.2. Server Data Movement Console Output 11
7. SOLID 12

1. Project Overview

Implement Networking into existing 3D Vehicles, save their state in a database so it can be reloaded from previous state, run a server and pull from clientside.

2. Project Requirements

- Complete outstanding work from previous versions of the assignment.
- The client applet/application can ~~stop, start, and~~ re-size without error**.**
- Implement a 3-tier model of the Vehicles

    1. Vehicles client applet/application, much as currently is, but gets Vehicles data from Server.
    2. Server supplies client with Vehicles data.
    3. Server accesses DB on server for Vehicles data.

- Use a sequence diagram to show relevant movement of data in the system.
- You may minimize the complexity (for now), by only accessing Vehicles data at the start, and storing it at the end.
- Note that Vehicles data starts in the server DB, is read by the Server App and sent to the client App. Use supplied sample code as a guide, if you wish, but make sure the code in the end ***is your own*****.**
- Copy the table below into your package-level documentation, and PDF. For every concept, describe how you implemented, or could implement it.

- You may defer full JavaDocs until the next assignment submission..

3. Database Implementation

ObjectDB has been implemented to allow for a save state, the system will save on a button press (Numpad_7), or every frame, which places all the items on screen into an array and send it of in a insert query to the database, on next boot, everything saved on screen will return in its last-saved position.

3.1. Persistent Database

Used a Object Oriented persistent database application provided by ObjectDB JPA.

3.1.1. Saving the Data

```
    public synchronized void saveGame(List<GameItem> gameItems) {
        em.getTransaction().begin();

        Query q2 = em.createQuery("delete FROM ObjDbConverter obj",
ObjDbConverter.class);
        q2.executeUpdate();

        for (GameItem items: gameItems) {
            if (items.getThisIs() != null) {
                ObjDbConverter obj = new ObjDbConverter(items.getPosition().x,
items.getPosition().y, items.getPosition().z, items.getThisIs(),
items.getIdentity());
                em.persist(obj);
            }
        }
        em.getTransaction().commit();

        query = em.createQuery("SELECT obj FROM ObjDbConverter obj",
ObjDbConverter.class);
//        List<ObjDbConverter> results = query.getResultList();
//        for (ObjDbConverter p : results) {
//            System.out.println(p);
//        }
    }
out.println(p);
    }
```

The way I handle this, is by creating a new object, a data container for all the important data that is contained within Vehicle objects, and with this new "Converter object", which gets stored into the ObjectDB as a separate thing, which later we pull from and

3.1.2. Loading Data

```
4.          TypedQuery<ObjDbConverter> q2 = em.createQuery("SELECT obj FROM
    ObjDbConverter obj", ObjDbConverter.class);
            List<ObjDbConverter> results = q2.getResultList();

            for (ObjDbConverter obj: results) {
                if(obj.getVeh_type() != null){
                    if(obj.getVeh_type().equals("LandVehicle")) {
                        Vector3f vector = new Vector3f(obj.getX(), obj.getY(),
    obj.getZ());
                        Vehicle house = new LandVehicle(vector, 0, 0, 100);
                        house.setVelocity(0.002f, 0.000f, 0.003f);
                        house.setRotationVel(new Quaternionf(0.06f, 0.01f,
    0.03f, 0.0f));
                        house.setScale(0.150f);
                        scene.setGameItems(new GameItem[] {house});
                    }
                    if(obj.getVeh_type().equals("AirVehicle")) {
                        Vector3f vector = new Vector3f(obj.getX(), obj.getY(),
    obj.getZ());
                        Vehicle house = new AirVehicle(vector, 0, 100);
                        house.setVelocity(0.002f, 0.000f, 0.003f);
                        house.setRotationVel(new Quaternionf(0.06f, 0.01f,
    0.03f, 0.0f));
    //                      house.setScale(0.150f);
                        scene.setGameItems(new GameItem[] {house});
                    }
```

Here we handle checking what is in the database, and creating new objects based on what they are and adding them to the game scene, you can see I check what type of item I'm adding, if the object is not null (Null will always be terrain), then compare against what items are stored in the results of the query.

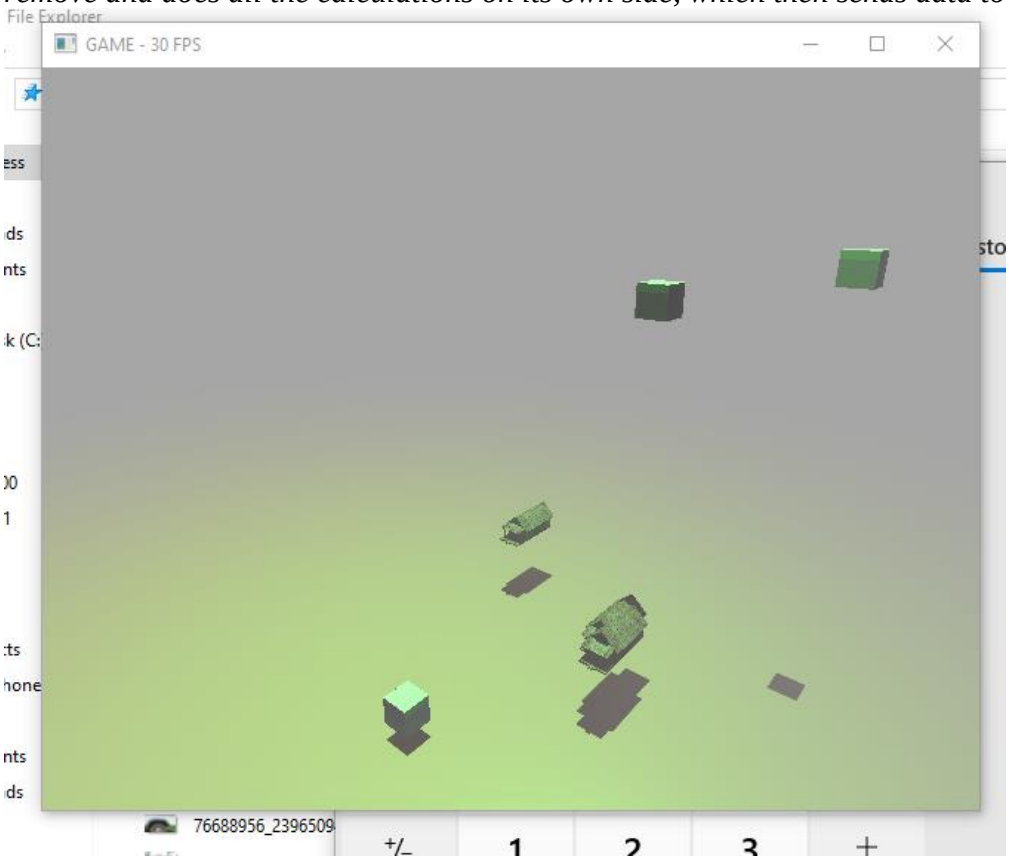
This then pulls everything from the database, and compares against the data. If the data type is a vehicle, which is stored as a string, it then creates a new vehicle object and stores it on screen.

4. 3-Tier Model

Application uses a three-tier model, Host → Server → Client.

4.1. Host Application

The System has a host application, which runs a 'master' program that can add or remove and does all the calculations on its own side, which then sends data to the server.



4.2. Server

The Server runs in conjunction with the host client, when the host client gets launched it calls the Server class to allow for a socket to connect to the server for data transfer to occur between clients. With the server running data can be pulled on the client side of things.

```
public class Server extends Thread{

    public static final Integer serverport = 53475;
    private ServerSocket serverSocket;
    ArrayList<NetWriter> clients = new ArrayList<NetWriter>();

//    ObjectOutputStream outputStream;
//    Socket socket = serverSocket.accept();

    public Server(int port) {
        try {
            serverSocket = new ServerSocket(port);
            System.out.println("We are SERVER SIR.");
            System.out.println("IP=172.16.176");
            System.out.println("Port=" + serverport);
//            outputStream = new ObjectOutputStream(serverSocket)
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    @Override
    public void run() {
        while(true) {
            mySleep(10);
            try {
                Socket socket = serverSocket.accept();
                NetWriter writer = new NetWriter(socket);
                clients.add(writer);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }


    public synchronized void sendToClient(List<ObjDbConverter> obj) {
        for (NetWriter e: clients) {
            e.sendObject(obj);
        }
    }
```

```
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
Directory = C:\GIT\w0422681\PROG2200\Game_02
Java = Oracle Corporation
Java = 1.8.0_211
We are SERVER SIR.
IP=172.16.176
Port=53475
```

4.3. Client

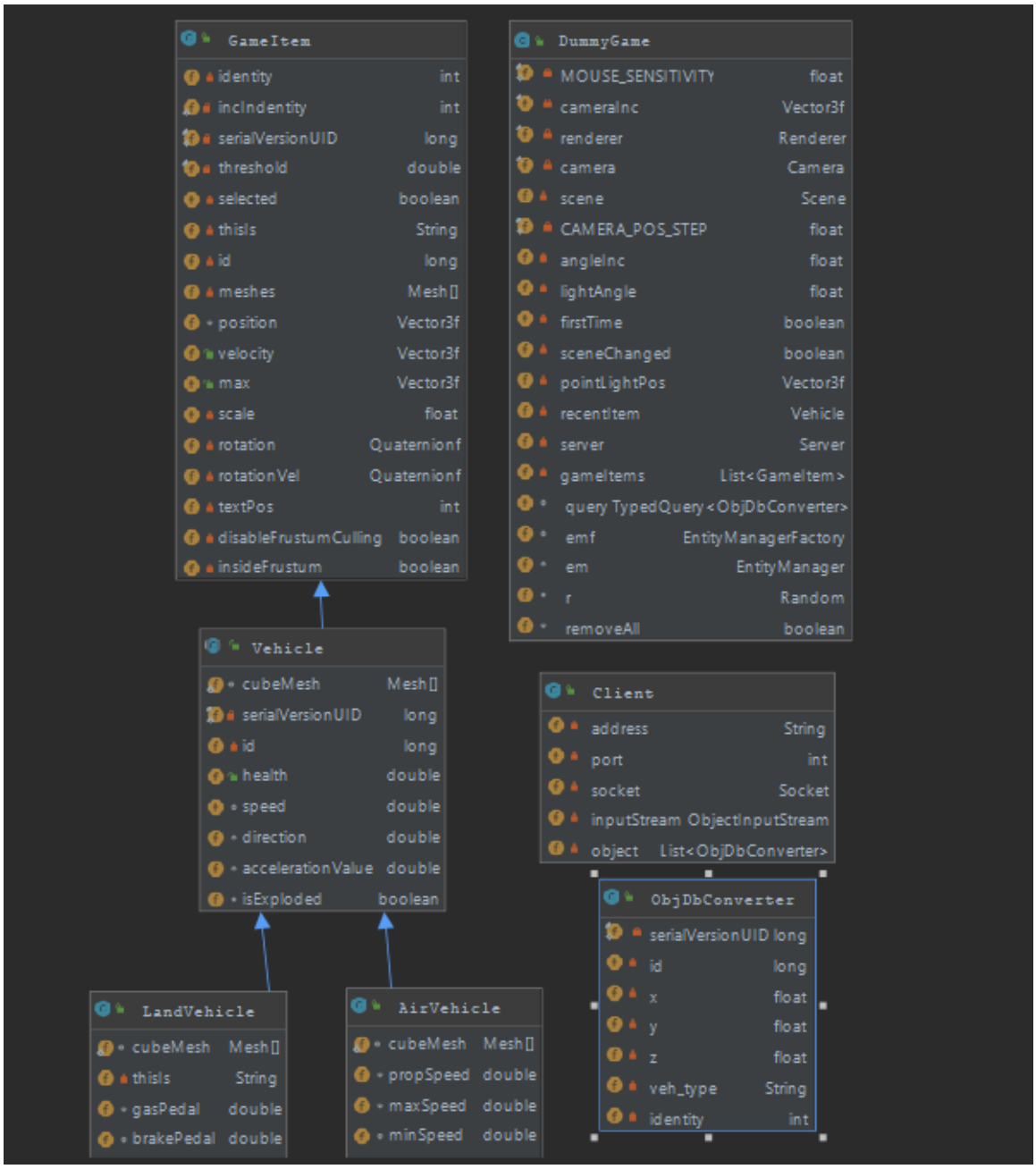
The client is a dumbed down version of the server, with all the physics calculations and inputs removed. It basically just reflects what happens on the host server. It connects to the server, and gets the data the server writes out to the connections. The client then stores this data in a list of objects which then get converted back into objects the client can use, then they will be drawn in by the library and updated via the data sent by the server which is provided by the Host Client!

```
public class Client extends Thread {
    private String address = "172.16.176.140";
    private int port = Server.serverport;
    private Socket socket;
    private ObjectInputStream inputStream;

    public void setObject(List<ObjDbConverter> object) {
        this.object = object;
    }

    public List<ObjDbConverter> getObject() {
        return object;
    }

    private List<ObjDbConverter> object;

    public Client() {

        try {
            socket = new Socket(address, port);
            System.out.println("We are the Client SIR.");
            inputStream = new ObjectInputStream(socket.getInputStream());
            object = (List<ObjDbConverter>)inputStream.readObject();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }


    @Override
    public void run() {
        while(true){
            try {
                object = (List<ObjDbConverter>)inputStream.readObject();
                System.out.println(object);
            } catch (IOException | ClassNotFoundException e) {
                e.printStackTrace();

            }
        }
    }
```

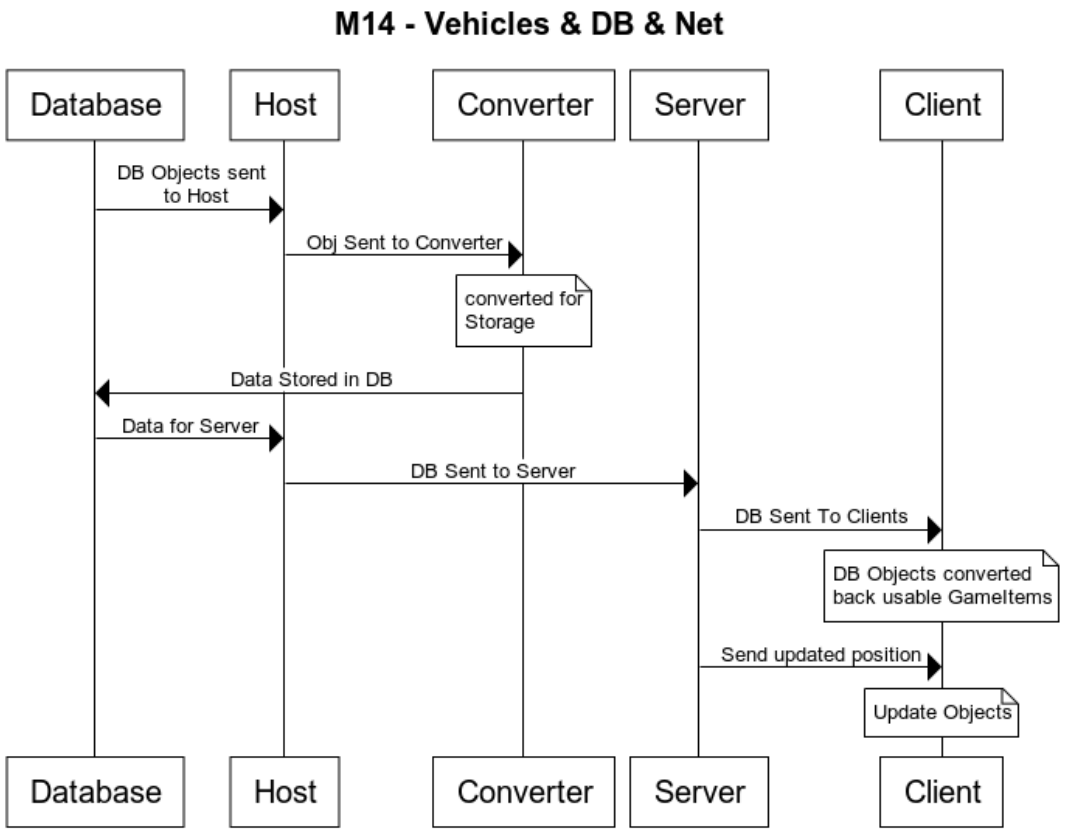
5. Diagrams

The sequence and class diagrams showing how classes are built and interact with each other, as well as the movement of data through the system.

5.1. Class Diagram



5.2. Sequence Diagram

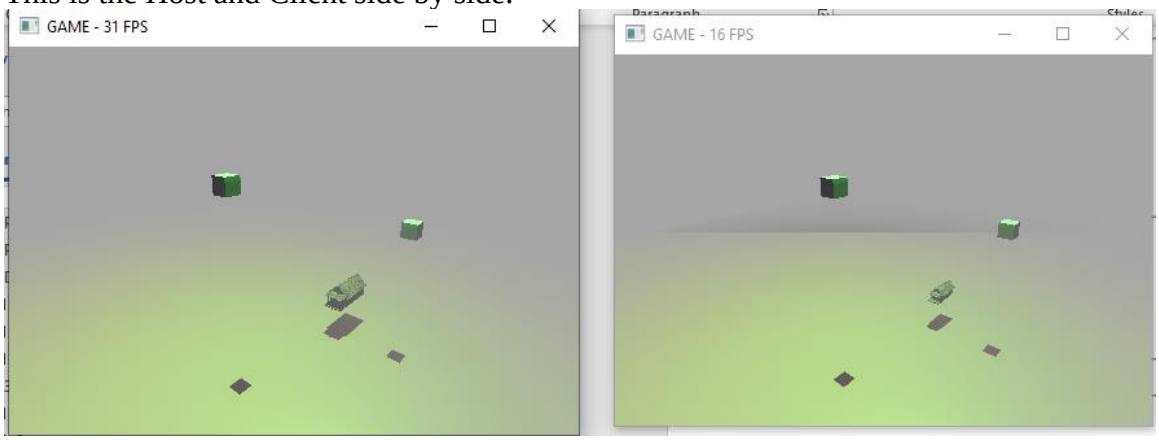


6. Testing

Showing Images of the full program running

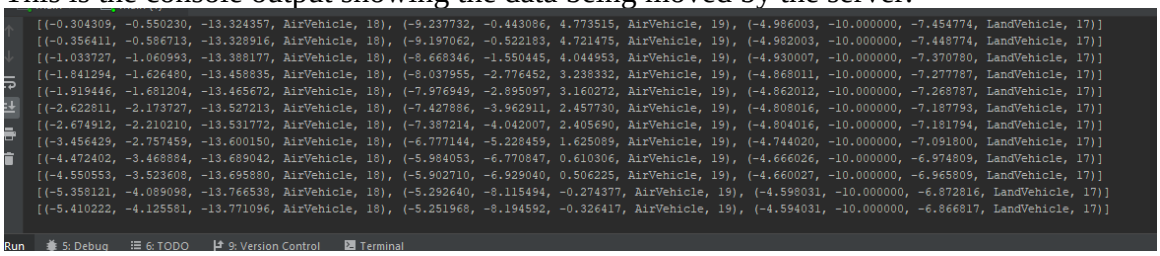
6.1. 3d Application running

This is the Host and Client side by side.



6.2. Server Data Movement Console Output

This is the console output showing the data being moved by the server.



7. SOLID

	acronym	Concept	My Application of this concept
S	SRP	Single responsibility principle	Methods and Classes are designed in a way for one purpose and one purpose only
O	OCP	Open/closed principle	Proper Scoping exists, classes and methods are open for extension, but closed for modification
L	LSP	Liskov substitution principle	Proper Inheritance is used in classes, all high-level classes have access to their base class methods.
I	ISP	Interface segregation principle	Interfaces are small and purpose built
D	DIP	Dependency inversion principle	high level modules should not depend on low level modules