System design document - HowDoIFlyThisThing

System design document for the HowDoIFlyThisThing project

Version: Forth iteration

Date: 2014-05-05

Author: Joakim Thorén

1 Introduction

1.1 Design goals

The design must be modular and it should be easy to implement new functionality and gameobjects. Rendering functionality must be completely separated from everything but itself, in order to implement an easy client-server architecture.

1.2 Definitions, acronyms

- GUI, graphical user interface
- Java, platform independent programming language
- JRE, the Java Run time Envorionment. Additional software needed to run an Java application.
- Host, a computer where the game will run.
- ECS, entity-component-system architecture.
- Component, a class of data and methods whose functionality is used to realize corresponding interfaces
- KryoNet, framework for easy networking

2 System design

2.1 Overview

The application will use the MVC-pattern. The model will be completely standalone, as will the view. The controller knows and owns both the model and the view.

2.2.1 Model overview

2.1.1.1 Model concept of design

The models core functionality resides in the gameworld class whose role is to execute the functionality of gameobjects within the game, such as Asteroids and Spaceships. This is inspired from the Entity-Component-System way of architecture, where every gameobject is a

collection of components of data which is then used for calculation in a system. This is not true in this application as the components are responsible for calculation aswell. On top of the gameworld there are classes for implementing rounds and network-sessions. Gameworld is the caller of the components calulation-methods.

2.1.1.2 Components in GameWorld

The components are classes which are instanciated inside a gameobject. This gameobject implements an interface, ie IMovable. IMovable provides a method move(), which should do moving calulations. The MoveComponent also provides a move() method which is forwarded and realizes move() within spaceship. This is a modular and robust design that's easy to maintain. HashMap of Moveables exists within Gameworld, which is looped through and within the loop move() is called on each IMoveable (value is the interface and key is the object). This makes sure order of GameObjects is irrelevant.

2.1.1.3 Deletion of GameObjects

Whenever an GameObject is deleted from the GameWorld, for example a SpaceShip is destroyed, the HashMaps of interfaces within GameWorld that are looped through in gameWorld.update() should be notified of this particular GameObject that is destroyed and erase the interfaces from each HashMap. This is done by maintaining a HashMap of Lists of Hashmaps. First tier hashmap is indexed by object, it returns the value - a list of the hashmaps which the objects resides in. In this way each object can via observer pattern fire a onDeath event, which the GameWorld listens to, and then the GameWorld can remove the source of the onDeath event from each list which the source was belonging to.

2.1.1.4 Creating of GameObjects

All GameObjects such as spaceships and asteroids are created from a corresponding global, singleton factory such as SpaceshipFactory or AsteroidFactory. This doesn't solve any problem for now other than having an uniform source of object creation.

2.1.1.5 Event handling

Events within the model are dealt with through the GameWorld as a listener, and whenever an object is created from a factory and then added to the gameworld via gameworld.add(GameObject) method, Gameworld starts listening for events from objects. Events are used whenever something that doesn't happen reguraly occurs. Whatever happens reguraly, on each game-loop tick, are done via components and systems in GameWorld.

2.1.1.6 Model network state-pattern

The core model gameworld is owned by a round which in turn is created by a HostState. The other state is ClientState. These two represents whether the application is running as host or client. The core difference is that the host does all computation and sends results to client

(position of objects and their type, point of client spaceship and sounds etc). The client sends, on it's update a list of inputs to the server and ties this input with an user. The client is a "dumb" client in that sense. The common interface of these two classes is "ModelNetworkState" which is provides methods for passing data from model to controller. If model-state is host it forwards data directly from gameworld to view via controller and if model-state is client it sends the latest recieved data from server to the view.

The data which is sent between model, view and network is always copied and in the case of drawables, it is encapsulated into a "DrawableData" class which packages all neccesary information for view to render an object into a single class. A list of DrawableData is retrieved from the model, and this list is easy to broadcast to each client. All this is achieved using KryoNet.

2.1.1.7 KryoNet

KryoNet requires classes that should be sent via networking to be registered. This is done in an utility class "NetworkUtils" in a static metod inside controller constructor. This guarantees uniform and in-order registration of classes which is required by KryoNet. For sake of consistency all classes that should be sent over network is encapsulated within a class with the "NetworkPacket"-suffix (ie DrawableDataNetworkPacket).

Whenever an user connects to the server he is added to the model as an user. This user is directly tied with connectionID. This connectionID is tied to a user via a HashMap. Whatever input is sent from this connectionID is given to the user, who executes it - causing the spaceship to do whatever it should do on the server.

2.1.2 Controller overview

The controller knows the view and the model. It forwards data neccesary for the view in order to draw things a list of DrawableData. This class provides position, rotation and enum which tells what type it is - a spaceship or asteroid for example. Note that with this approach the model is completely independent of the view, Slick2D - no Slick2D classes is ever used within the model.

A class User is used for maintaining keybindings to specific spaceships. User takes advantage of the state-pattern, it can be either in spectator-mode or play-mode. User enters play-mode whenever a new round starts. A newly created spaceship is binded to the user. User listen to its spaceship, whenever spaceship explodes and sends it's death-event User will switch state and enter spectator-mode in which user keybindings no longer affects the spaceship.

2.1.3 View overview

The view is placed on a thread of it's own, due to Slick2D. The only data that passes through the Main thread (Model and Controller) to the view thread is the list of DrawableData, which controller gives to the view (not vice versa!). This is properly synchronized between the two threads. Upon

recieving the list of DrawableData the view translates the enum type of the drawables to an actual Slick2D image.

2.2 Software decomposition

The application is decomposed into the following modules:

- Model
    - Gameworld (Closely related to the commonly known System manager from ECS)
    - Factories
- Controller
    - Input handling via Users
- View

2.3 Concurrency issues

There were issues with getting list of IDrawables from model to view, due to model writing to this list while view was iterating through it. Solved by sending a copy of the list to the view thread.

Also input is recieved from view thread and passed over to the main thread. This is was during a period out of shape and broken, with crashes happening fairly frequently (average lifetime before concurrencyModification exception is about 5min). Solved by proper synchronization.

2.4 Persistent data

N/A. No write to harddrive occurs at any point.

2.5 Access control and security

N/A

2.6 Boundary conditions

Application launched as normal application but existed via in-game menu. This is possibly subject to change in future SDDs due to lack of time implementing GUI.

3 References

Appendix