System design document - HowDoIFlyThisThing

System design document for the HowDoIFlyThisThing project

Version: Sixth iteration

Date: 2014-05-21

Author: Joakim Thorén

This version overrides all previous versions.

Table of contents

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 and 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.

- Client, a computer that displays a game form a Host and interacts with it.
- ECS, entity-component-system architecture.
- Component, a class of data and methods whose functionality is used to realize corresponding interfaces
- Slick2D
- KryoNet, framework for easy networking
- Lazy add, technique for thread safe way of using Collections on different thread by adding elements to a *list of elements to be added later on*, and then with polling adding them to real list on main thread.
- Tick, one full run-though of a loop

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.

2.1.1 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 calculation-methods.

Calculations regarding vectors in various areas of model uses a VecmathUtils class, which provides useful methods ontop of the vecmath library. VecmathUtils is essentially an extension of the original vecmath library.

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 that object if necessary. Events are used whenever something that doesn't happen reguraly occurs. Whatever happens regularly, on each game-loop tick, are done via components and systems in GameWorld. Examples on non-recurring code would be sound, which are sent as events. This is more efficient than polling.

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 provides methods for passing data from model to controller. If model acts as host it forwards data directly from gameworld to view via controller and if model-state is client it sends the latest received data from server to the view via the common interface ModelNetworkState.

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 necessary 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 Rounds

A round is owned by HostState. Whenever a user is connected to the host, the user is added to the round. This list of users is maintained by Round. Depending on the state of the round, which can be either "ActiveRound" or "InactiveRound", different things happen. If the round is active, set added players as spectators; if round is inactive, give them a spaceship. Whenever a round is over (when only 1 user remains) a countdown-timer starts. When the timer countdown is finished the round begins, setting Roundstate to ActiveRound and all users currently connected to host are given a spaceship.

The class Round have to consider many possible scenarios regarding user management. What happens if a user disconnects during round countdown or if a players client crashes during a fight? All possible scenarios are handled effectively by adding users in a lazy way. Users can request a spaceship, which will give them a spaceship on next tick - not directly. Whenever requesting a spaceship their request is stored in a set of requests, which is processed on each tick and thereafter cleared. This is due to ensuring safe threading as the addUser is usually called from server thread, causing modification of Users list. This modification could happen on iterating over the set in main thread. Therefore it is vital that the users are added in a defined order to this set, which is only doable by adding a user to a *list of users that should be added later on*.

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 player-mode. User enters player-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 and instead moves the camera around the map.

2.1.1.8 KryoNet

KryoNet requires classes that should be sent via networking to be registered. This is done in an utility class "NetworkUtils" in a static method 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). These are stored in a separate packet to make the code easier to read.

Whenever an user connects to the Host 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, a list of DrawableData, to the view and renders this.This class DrawableData 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. All other view related features are done the same way - the controller forwards data from model to view each tick. Exception to this are animations which are handled as events from model. On controller recieving e.g. an explosion event the controller calls an appropiate method in view, i.e "createExplosion()".

The Controller is owned by the LauncherController that also handles the calling cleanup methods when the game ends. The launcher controller also handles what information is displayed in the Launcher.

2.1.3 View overview

2.1.3.1 From model to view

Rendering is done with the game-library Slick2D. 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.

The view also maintains an ArrayList of animations that are drawn every render uppdate. When an animation has been drawn its full animationcycle it is removed from the list of animations. The animations are added in the list from a methodcall from the Main thread.

2.1.3.2 Launcher

The launcher is started and controlled by its own controller, LauncherController. LauncherController is the only class instanced by the main method.

The launcher is made up of two different panels, StartPanel and OptionsPanel. The StartPanel provides the choices to host or join a game, it also alows the user to swich to the OptionsPanel. The OptionsPanel allows the user to toggle fullscreen and change keybindings, it also alows the user to switch to the StartPanel. The actions preformed by the user are forwarded to the LauncherController through use of the ObserverPattern. Upon receiving these events the LauncherController calls the appropriate method. It is form here that the actual game is started through instanciating the Controller.

2.1.3.3 Sound
The controller creates the class SoundEffects, which is responsible to create and play sounds. Whenever Gameworld receive an event, the name of the event is added to a HashSet. In the update method of the controller the set is obtained from Gameworld via getters, and used as parameter in the call to SoundEffects method playSound(). If the set is null the method will do nothing, else there is a check of which sound to play.

The host first sends the set of sounds to the client, then clear it so that no sound will be played more than once. If the method getListOfSounds() are called from the client, the values of the set is copied into an new HashSet, the set is cleared and then the new set is returned.

2.2 Software decomposition

2.2.1 General
The application is decomposed into the following modules:

- Model
  - Gameworld (Closely related to the commonly known System manager from ECS)
  - Networking (rounds and ModelNetworkStates)
  - Factories
  - Components
- Controller
  - Input handling via Users
  - Network packets
- View
  - Launcher Frame and Panels
  - Slick thread

2.2.2 Package dependencies
Package dependencies are showing in fig 1.0 that the MVC-pattern is followed as intended with a independent model and view.

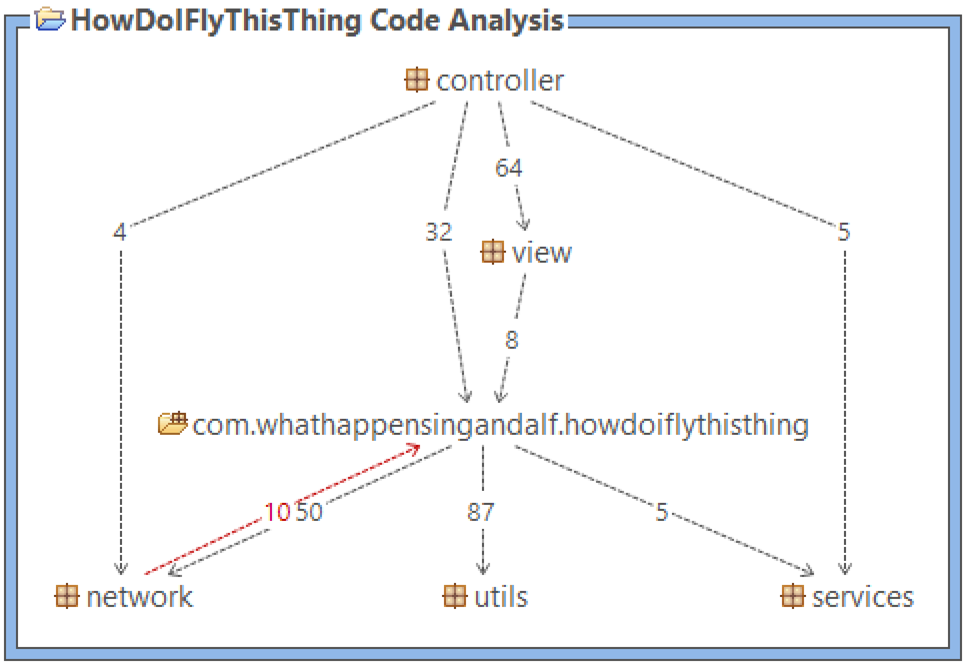
HowDoIFlyThisThing Code Analysis
controller
64
4
32
5
view
8
com.whathappensingandalf.howdoiflythisthing
10
50
87
5
network
utils
services

Fig 1.0

Network-package is thightly coupled with model-package. This is intended. Network classes was put in a package of it's own due to organizing reasons.

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 received from view thread and passed over to the main thread. This 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

The *.jar-file starts a launcher with possibility to Host, Join or Exit. Exit will close the program. Hosting or joining starts a Slick2D window. Exiting from this window is done by X:ing out or ESC. Both of these returns the user to the Launcher. The launcher is closed by X:ing out or pressing Exit.

3 References

1. MVC, http://en.wikipedia.org/wiki/Model-view-controller
2. ECS, http://en.wikipedia.org/wiki/Entity_component_system
3. Slick2D, http://slick.ninjacave.com
4. KryoNet, https://github.com/EsotericSoftware/kryonet
5. State-pattern, http://en.wikipedia.org/wiki/State_pattern
6. Visitor-pattern, http://en.wikipedia.org/wiki/Visitor_pattern
7. Observer-pattern, http://en.wikipedia.org/wiki/Observer_pattern
8. Vecmath library, http://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/desktop/java3d/forDevelopers/j3dapi/javax/vecmath/package-summary.html

Appendix

Design model is not embedded here. See project/documents/DesignModel.pdf.

- HowDoIFlyThisThing - Outmost class of model. This is what controller knows about
- ModelNetworkState - State interface if model is acting as host or client
- NoneState - default state, all methods does nothing
- ClientState - Implements ModelNetworkState and realizes it by using data recieved from networkpackets
- HostState - Implements ModelNetworkState and realizes it by using data directly from simulation. Maintains connected users.
- Rounds - Decides when a new round should begin
- RoundState - State interface if round is active or inactive
- ActiveRound - Treat users in rounds as intended when round is active
- InactiveRound - Treat users in rounds as intended when round is inactive
- GameWorld - Core model class. Maintain object lifetime and execution. Contains lists of component-classes.
- WorldBorder - Identical to javas Dimension class. Represents border of world.
- GameObject - All substantives in the game implements this. Specifies type of object such as “SPACESHIP” or “ASTEROID”
- IDrawable - Interface which provide data used for rendering an object
- IRechargable - Used for gameobjects that can recharge their shields
- ICollidable - Used for gameobjects that can collide with other gameobjects
- IMoveable - Used for gameobjects that can move

- IArmable - Used for gameobjects that can shoot
- IPickup used for gameobjects that can be picked up
- IProjectile used for gameobjects that can be projectiles
- DrawableData - realizes the IDrawable interface by having references to owners member variables. Sent via network or directly to controller if client or host
- Shield - realizes the Rechargeable interface by having references to owners member variables. Recharges.
- CollidableComponent - realizes the ICollidable interface by having references to owners member variables. Utilizes visitor-pattern.
- MoveableComponent - realizes the IMoveable interface by having references to owners member variables. Perform move calculations.
- ArmsComponent - realizes the IArmable interface by having references to owners member variables. Publishes bullet-events.
- HealthPickup - Is a pickup and gives health to spaceships on collision
- WeaponPickup - Gives it's projetile to the spaceship so that the spaceship will use that projetile when firing
- Missile - Implements IProjectile and got different values causing it to behave different from ordinary bullet
- CookieCracker similar to a projectile but does damage to asteroids upon collision (by using visitor-pattern)
- Bullet - default projetile in spaceships
- Spaceship - Owns many components and thrusters. Capable of firing bullets and is steered by activating thrusters.
- ThrusterComponent - Contains three thrusters and provides methods for controlling these
- Thruster - Physics calculation when it's activated
- Hull - Represent hull (health) on a gameobject
- User - Takes input and executes it on associated spaceship or spectator camera depending on it's state.
- IUserState - If user is a player (has a spaceship) or is a spectator (has no spaceship)
- Playerstate - Executes input on spaceship
- SpectatorState - Executes input on camera
- Networkpackets - Encapsulates data into a packet which is sent with kryonet.