System design document for PiFive

Version: 0.9
Date 2012-05-21

Author

Anton Lindgren, Johannes Wikner, Victor Lindhé, John Hult

This version overrides all previous versions.

1 Introduction

1.1 Design goals

We want to have an object oriented designed system implementing a model-view-controller pattern driven by the model development. The model should be independent from view and controller, so that we can switch to another graphics engine for example.

1.2 Definitions, acronyms and abbreviations

- GUI, graphical user interface.
- Java, an object oriented programming language.
- JMonkeyEngine3 (JME/JME3), a graphics and physics engine/library.
- Blender, a 3D model design program.
- MVC, model-view-controller. Design pattern to increase flexibility and independence between the parts of the software.
- Nifty-GUI, a framework used for drawing GUI-components to screen.

2 System design

2.1 Overview

The application is initialised through the main.Main class which acts as a glue creating our MVC-structure as well as setting up the JME3 environment. Controller and View are then supplied with the components from this JME3 environment to be able to delegate tasks to the framework.

The controller's most important task will be sending key events based on different gamestates to the model so that the game can respond to user interaction. This is done through different listeners implementing interfaces from JME3 so that it can use the framework’s inputmanager.

The view will then be responsible for drawing the corresponding 3D representation as well as displaying GUI components depending on the current gamestate. The view depends heavily on the Java game framework JME3 and the Nifty-GUI plugin that JME3 supports. This dependency is realized through delegation to these frameworks.

The model will use a framework to control it’s physical world via a mapping-package called physics. This will give a loose binding to the physics-engine since only a few classes will

depend on it. Game mechanics will to a high extent be able to be implemented without any knowledge of the physical implementation.

2.1.1 Settings

The Settings class is used for enabling the user to modify the game-mechanics to provide a more flexible game where the user is in control of how he or she wants to play.

The class Settings implements the design-pattern Singleton and maps different strings to an Integer value specified from a settings file found in project/assets/settings.

All model classes should then use the Settings class to fetch parameters that help build game-mechanics.

2.1.2 The model functionality

The model is represented globally by the interface IGame which is used by view and controller as an entry-point to the rest of the model classes. The implementation of IGame mostly connects the method calls from the controller's different listeners to the lower model classes that represent game objects. These lower classes will then tell it's graphical representation via events when information has changed.

2.1.3 The model's physical representation

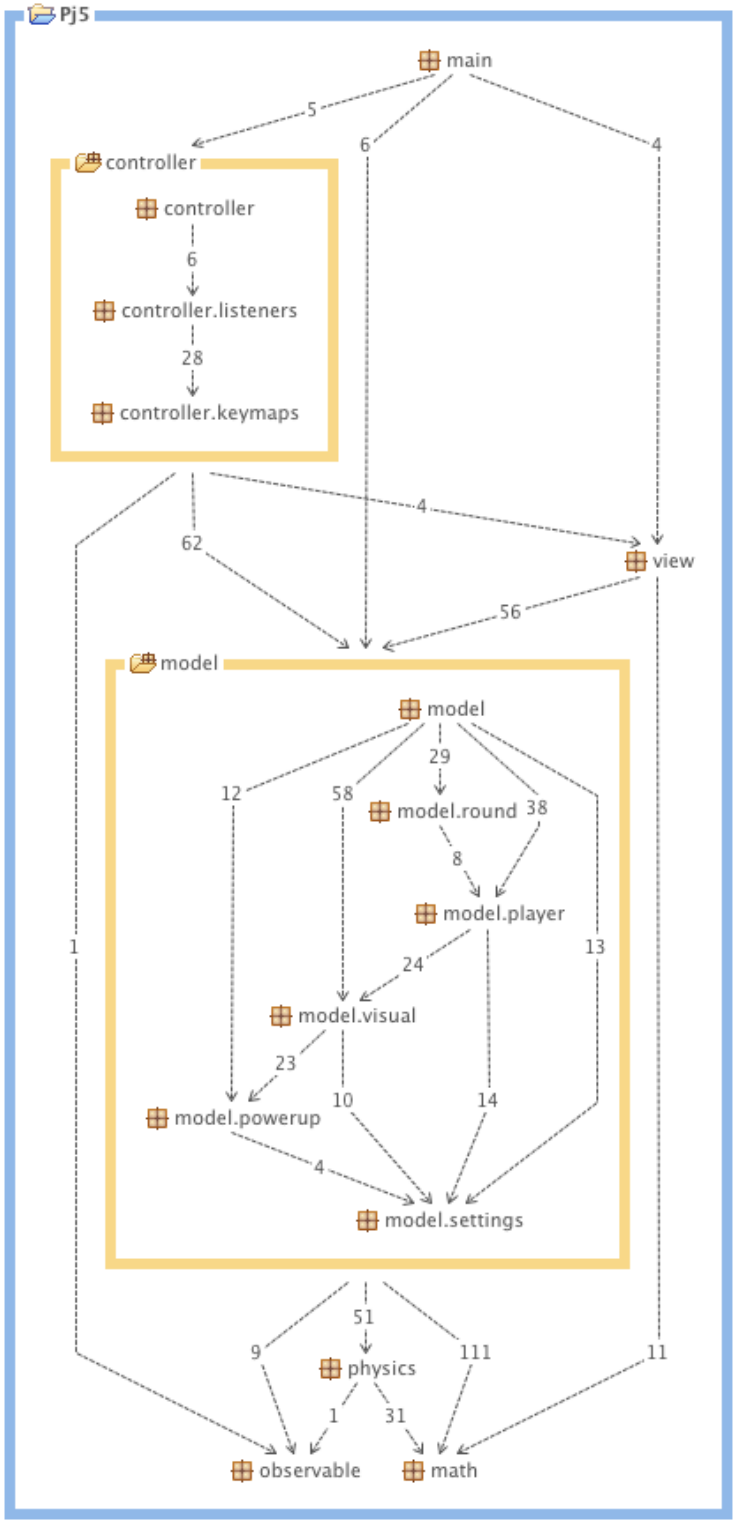
The lower model classes containing information that represent the game objects hold no variables to represent position, direction or other parameters that could be identified with a physical world. Instead these objects (ex. model.visual.Unit) contain information that is relevant to our domain model like hitpoints and powerups.

To simulate a physical world we instead use a package called physics that maps method calls from our model to an implementation of IPhysicsHandler and IPhysicalBody which holds thoroughly simulated physical information. This has made the application able to use complex collision detection and force appliance to control units. Which has enhanced gameplay with a great extent.

2.1.4 In-game messages

The StatusBox class in model.visual handles all strings that the model wishes to display to the user. It is a singleton which can be accessed to publish messages through. The game is only meant to function in english so therefore the model supplies it's own strings to the statusbox.

2.2 Software decomposition


Pj5
main
5
6
4
controller
controller
6
controller.listeners
28
controller.keymaps
4
62
view
56
model
model
29
12
58
model.round
38
8
model.player
1
13
24
model.visual
23
10
14
model.powerup
4
model.settings
51
9
111
11
physics
1
31
observable
math

2.2.1 General

- main, contains only one class where the programs main method lies. It starts up the model, the view and the controller and imports settings. It updates the controller.
- controller, updates the model and the view. Handles input.
    - controller.listeners, handles input from the keyboard
    - controller.keymaps, has the different layouts for each player's key
- view, renders all the graphics on the screen and GUI. Uses JME.
- model, handles all game data. The interface IGame is the model's core and connects different submodels such as a model for players and a model for rounds.
    - model.round, handles game rounds
    - model.player, handles players
    - model.visual, handles the things in the model that should be rendered graphically. The visual objects has physical objects in the physical package.
    - model.powerup, handles different power ups a player can get while playing.
    - model.settings, handles the settings import. Singleton pattern.
- physics, the physical world. Uses JME.
- observable, an interface for adding listeners
- math, contains mathematical classes

2.2.2 Decomposition into subsystems

The only subsystem we have is the physics package, handling physics with classes from the JME library. Although it depends on JME library, the model speaks its own language and doesn't know the implementation of the physics package, so it is replacable.

2.2.3 Layering

In the bottom we have math and physics. Then model. Then view. Then controller. Although we do not follow an exact layer design, it resembles this kind of design to a great part.

2.2.4 Dependency analysis

There are no circular dependencies at all. We have solved this by using interfaces, abstract classes and listeners.

2.3 Concurrency issues

No concurrency issues, the JME Framework will take care of resource-heavy methods such as rendering and collision-detecting and that is not something we can improve more than using the framework properly.

2.4 Persistent data management

The persistent data will be stored as three different types.

- Blender models stored as .blend in assets/blends/
- GUI layouts will be stored as .xml in assets/xml/
- Settings will be stored in a plain text file in assets/settings

2.5 Access control and security

N/A

2.6 Boundary conditions

N/A

3. References

JMonkeyEngine: http://www.jmonkeyengine.com
MVC: http://en.wikipedia.org/wiki/Model-view-controller
NIfty-GUI: http://nifty-gui.lessvoid.com/