System design document for project Gentlemen’s Dodgeball

Table of Contents

1 Introduction
  1.1 Design goals
  1.2 Definitions, acronyms and abbreviations
  1.3 References
2 Proposed system architecture
  2.1 Overview
    2.1.1 Model/Controller pairs
  2.2 Software decomposition
    2.2.1 General
    2.2.2 Tiers
    2.2.3 Communication
    2.2.4 Decomposition into subsystems
    2.2.5 Layering
    2.2.6 Dependency analysis
  2.3 Concurrency issues
  2.4 Persistent data management
  2.5 Access control and security
  2.6 Boundary conditions
  2.7 References

Version: 1.1

Date: 14/4 2011 (Revised 26/5 2011)

Author: Erik Sikander, Karl Bristav, Gustav Olsson, Viktor Åkerskog

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The software architecture should:

- Use the Model-View-Controller design pattern
- Be flexible to enable easy implementation of new features
- Make it easy to manage the game world
- Support the chosen external libraries (LWJGL, JBox2D)

1.2 Definitions, acronyms and abbreviations

GUI - Graphical User Interface
MVC - Model-View-Controller; A design pattern that separates the model, controller and view functionality of an application.
Game instance - A copy of Gentlemen's Dodgeball running on a computer.
Game world - A virtual world that is visualized by the game instance.
Entity - Represents a game object within the game world. Consists of a model/ controller pair.
Player - A person interacting with the game instance. A player has a representation inside the game world that other players can see and interact with.
Team - A set of players working together. The game will initially feature a total of two teams that are identical in every aspect except graphical representation. Each team will initially consist of one player.
Level - The arena on which the game is played.
Flag - An artifact that belongs to a team.
Ball - A team-neutral entity used by a player to knock out other players by means of throwing.
Knocked out - A player who is temporarily incapacitated, unable to pick up flags.
Scoring - An act which awards a team with score points. Examples include the act of knocking out another player, capturing the opposing team's flag and returning the own team's flag.

1.3 References

MVC - http://en.wikipedia.org/wiki/Model-view-controller

2 Proposed system architecture

In this section we propose a high level architecture.

2.1 Overview

The game will use the MVC design pattern where parts of the view is integrated with the model in order to support the graphics pipeline of OpenGL 1.1. The model will be semi-fat.

All game objects consist of a model/controller pair where all functionality that is generic to a specific game object belongs to the model (such as the details of how a player moves). The corresponding controller is responsible for functionality that is not generic to a specific game object (such as the control mechanism that is used to move a player model or talks to other controllers). Such a model/controller pair is known as an Entity in the software architecture.

The model will be automatically tested but some aspects of the game, such as certain gameplay elements as well as some controllers, may be too difficult to write effective tests for and are tested manually.

2.1.1 MVC structure

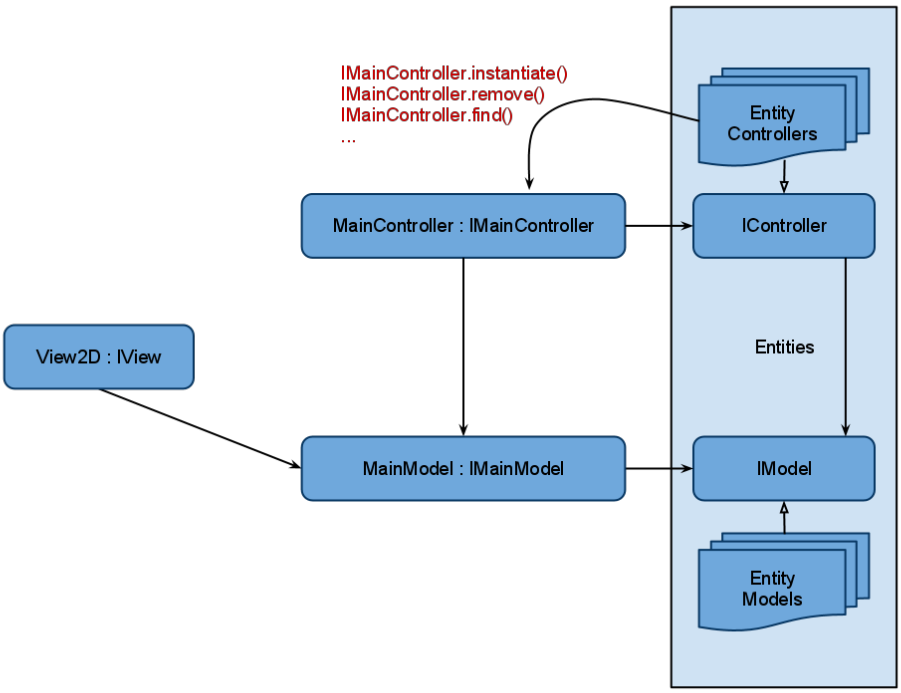

Figure 1: System Architecture Overview

The software architecture consists of a main model, a view and a main controller as seen in Figure 1.

The main model and main controller are responsible for storing and managing all entity models and entity controllers respectively, hence their names.

All updates are polled and there are no observer patterns in the application. Unlike a traditional application, not all actions and updates in the game are initiated by a user. There may be a lot of things happening at any given moment within the game world and observers would in practice be called every frame, hence the game uses a standard game loop.

2.1.2 Entities (Model/Controller pairs)

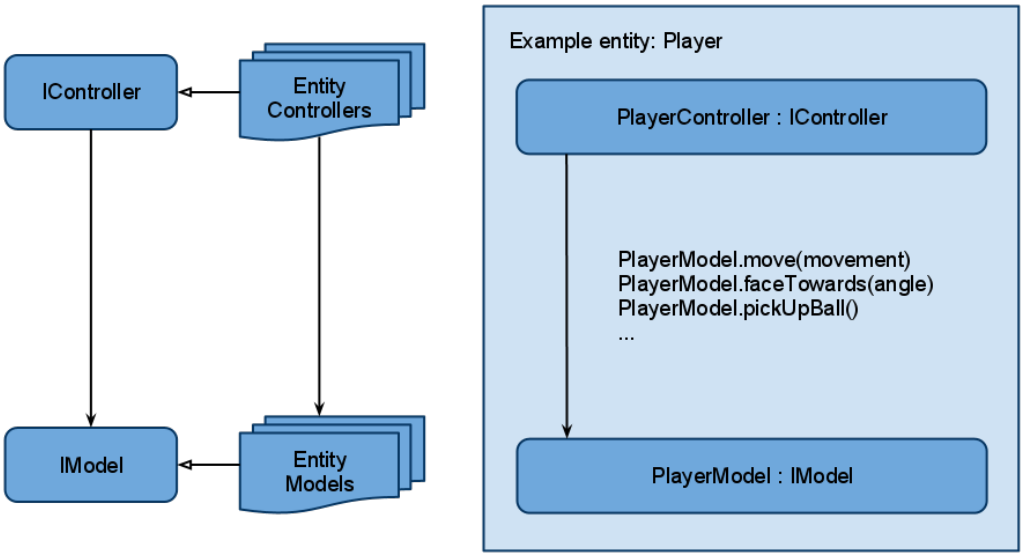

Figure 2: Entities

An entity may be instantiated at any time from any part of the controller logic code using the IEntityFactory interface, which connects a model and a controller to form a pair as seen in Figure 2. This structure is essential because it separates generic from non-generic functionality of an entity, reduces code redundancy (by giving the model a very generic interface, enabling different controllers to use the same model) and improves testability of core functionality. For example, it is easier to test player movement code when it is not directly connected to keyboard input.

Being able to instantiate an entity from controllers is important for the gameplay logic. Controller instances (and connected models) will be easy to find in order to reduce static references and get-method calls.

2.1.3 TypeMap

All entity models and controllers are stored in a TypeMap. A TypeMap internally stores instances of subtypes S to a supertype T as values with the key Class<S extends T>. This enables easy and fast access of all instances of a specific type and this is where the game world exists in the main model.

2.2 Software decomposition

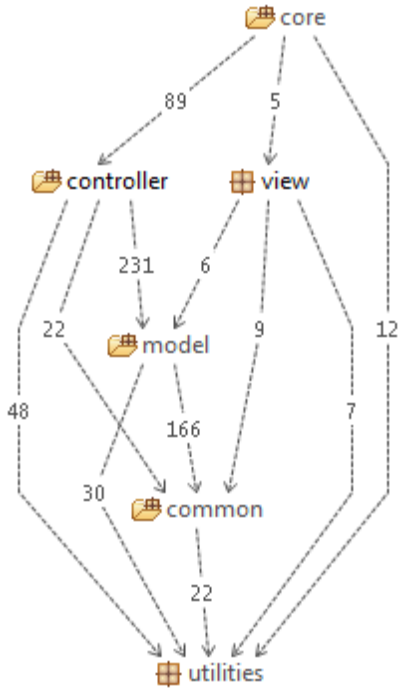

Figure 3: High level design

2.2.1 General

- core package - includes initialization class and window handler
  - core.levels package - includes preset map levels.
- controller package - includes the main controller objects and all entity controllers
  - controller.common package - includes shared objects of the controller
  - controller.components package - includes all components that work with entity controllers or provide information to them
  - controller.entities package - includes all entity controller objects
    - controller.entities.gameplay package
    - controller.entities.effects package
    - controller.entities.gamemode package
    - controller.entities.props package
- view package - includes view objects that render the game world to the screen
- model package - includes main model objects and all entities model
  - model.common package - includes shared objects of the model
  - model.entities package - includes all entity model objects
    - model.entities.gameplay package
    - model.entities.effects package
    - model.entities.gamemode package

        - model.entities.props package
    - common package - includes objects that are shared across the application
        - common.body package - includes all objects for the physical representation in the game world
        - common.geometry package - includes all objects for the graphical representation in the game world
    - utilities package - includes utilities objects that may be used across the whole application

2.2.2 Tiers

No tiers. There will only be one application.

2.2.3 Communication

There will be no communication because the application does not provide network functionality.

2.2.4 Decomposition into subsystems

The IComponent implementations in the controller.components package can be considered subsystems of the application.

Some future entities may be considered subsystems of the application depending on their functionality, but there are none initially.

2.2.5 Layering

The layering of the project is as shown in Figure 3. Higher layers are at the top of the figure.

2.2.6 Dependency analysis

Dependencies are as shown in Figure 3. There are no circle references in the application.

2.3 Concurrency issues

The application will be single threaded and no concurrency issues has to be considered.

2.4 Persistent data management

The application has no persistent data.

Future features may include persistent game levels. All persistent data, if implemented, will be stored in text files; either as order-dependent values or as XML-tags.

2.5 Access control and security

Since the game is a local multiplayer game running on one computer, there are no security issues.

2.6 Boundary conditions

N/A. The application is launched and exits as a normal desktop application

2.7 References

- For more information on the project, see the RAD located at: **gentlemen/ documentation/RAD.pdf**