System Design Document for Not Enough Space, 20.

Version: 0.5

Date: 2017-05-08

Author: Philip Nord, Julia Ortheden, Jonas von Essen, Fredrik Viberg

This version overrides all previous versions.

1 Introduction

General info. What is this? What does it describe?

1.1 Design goals

This document contains a technical description of the construction of the application Not-Enough-Space, a computer based arcade game located in Space.

Some design goals:

- The design must be testable and it should be possible to test just specific parts of the application.
- The application must have an MVC structure with a model completely without dependencies of the framework.
- The application will be made for desktop use, standalone, and single player mode.
- A graphical user interface for Mac/Windows/Linux platforms will be a requirement.
- The application must have a 3D-GUI and 3D navigation.
- It must be possible to add new game modes with relative ease.

1.2 Definitions, acronyms and abbreviation

All definitions and terms regarding the Not-Enough-Space game can be found under References.

Round - one round inside the game.
Program - the application itself.
Phase/Part - used interchangeably and refer to the two different phases of the game.
Beam - “Tractor beam”, the classic UFO abduction beam.
jME/jME3 - JMonkey Engine, the framework used to develop the application.
NiftyGUI - A GUI framework integrated in jME.
MVC - Model-View-Controller pattern.

See RAD document for further non-technical definitions etc.

2 System architecture

Overview

The application is built around a clear MVC-structure. The model package is completely independent from the framework, and contains all data, behaviours and calculations. The model doesn't care how it's presented, and communicates with the view and control through an event bus and an interface called PlanetaryInhabitant (PI). The PI contains method declarations that cover all functionality related to in-game 3D entities that the model requires from the framework, without relying on specific implementations. To make the application work, there is a thin control layer that handles communication between the PIs and the view (the 3D scene, could also be regarded as the jME model). The control package also contains a jME-specific implementation of PI.

To clarify, the model doesn't actually keep track of entity positions. The only reference from the model to the 3D space is the PI, from which positional data can be retrieved and modified.

Model - Control relation

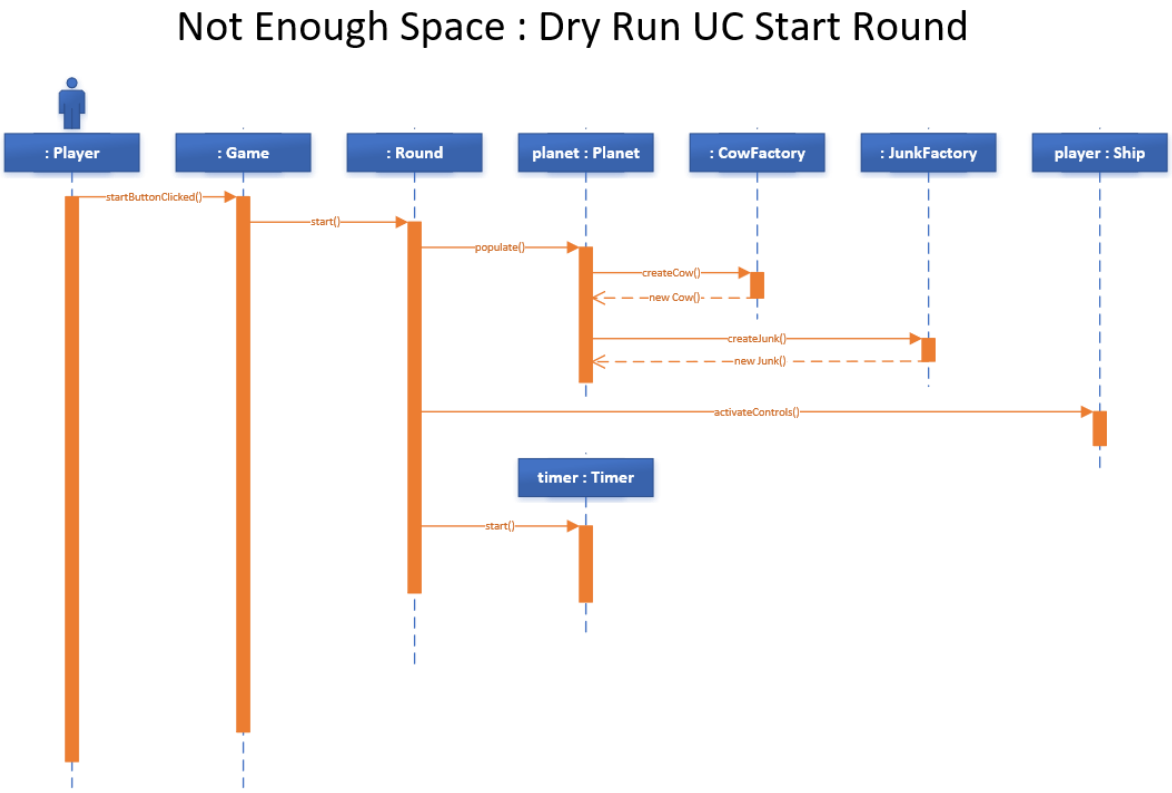
While the model is independent from the framework, it doesn't contain a clear entry point for control. This is due to the haunting fact that the model can not actually run without *some* framework, since the model doesn't handle 3D calculations or the update loop on its own. The result of this is that the model needs to be controlled in very specific ways, by a thin but wide control layer. The controller needs to call the update method in each model entity individually, and it needs to assign PIs to each new entity created (through EntityCreatedEvents). The framework also needs to supply collision detection events, between *some* of the entities but not all. The reason for this is that the model shouldn't mimic the functionality of jME and just hide it behind an abstraction layer - it should ask only for the help it actually needs. Both the PI interface and the event package are created through the eyes of the model, leading to true framework independency.

Administration

Controlling which and whether a game mode is active, exiting the application, and pausing the game are all administrative tasks. These are not in any way handled by the model, and as such are left to the framework implementation. This is currently handled by a StateManager, which controls and synchronizes NiftyGUI screens, input and jME AppStates together with model GameModes. The different states of the game are menu, in-game and paused. There is currently only one game mode, but the code supports adding more modes.

Dependency analysis

--insert dependency diagram--



3. Subsystem decomposition

File handling system responsible for icons, images, models etc? --Inte vårt system, jME sköter det. Står även om det nedan.

3.1 Core

3.2 Service

4. Persistent data management

The application makes use of the jME framework to store and load files. The files used are 3D-models for the jME view and their associated textures and materials, as well as GUI-related images. JME uses a global asset manager class for loading the files, while NiftyGUI uses xml-files to control the display.

No user data is stored between sessions. *Highscores might be added.*

5. Access control and security

No different roles exist.

6. References