System design document for Tankz

Table of Contents

Version: 1.0

Date: 2013-05-15

Author: Albin Garpetun, Daniel Bäckström,
Johan Backman, Per Thoresson

This version overrides all previous versions.

1 Introduction

1.1 Design goals

Our goal was to design the game with as much use of the MVC pattern as possible. Due to limitations in the game engine, the use of true MVC was nearly impossible. The model classes should be relatively stand alone, and can be tested with very little graphical representation. The connection between model, view and controller should be constant, and should be easy to overview.

1.2 Definitions, acronyms and abbreviations

- JME - JMonkey Engine. The game-engine used in the development.
- HUD - Heads Up Display. On-screen graphical representations for powerups, health and time.
- Spatial, graphical representation of an object.
- MVC - Model View Controller.

2 System design

2.1 Overview

In this section we explain the overall design choices.

The application is implemented towards a form of MVC pattern, where the models contain most of the logic of the actual game state. These models are listened to by the corresponding view for changes in various properties. The models also sends events when the view has to know that something has happened other than a property change, for example when the view should show an effect.

When working with JMonkey the controllers are usually the ones to listen for collisions, so the models just get alerted when a collision happens and can then take appropriate actions. However the controllers has to know when the collision shapes in the world should be enabled or disabled to allow collisions to happen when they should. Hence, the models has to alert the controllers when they are not in a visible state in the gameworld. These events are then passed via the view to reduce the creations of event objects, since the view also has to know about visibility so that they don't get rendered anymore.

By just disabling the rendering and collisions of the objects in the world, we did not have to remove or add things to world during gameplay. We just render them invisible, disable the collision shapes, move, rotate and apply forces which is much less costly in performance.

2.1.1 Appstates

The implementation uses the existing functionality and support that exists for different app states in JMonkeyEngine. An app state is exactly what is sounds like, a state of the application, Menu, Game, LoadingScreen, all these are app states. What would intuitively feel like a “state” when playing the game, is probably an app state. An app state can be “attached” and “detached”. You can have several app states active at once, for example one for the Game, and one for the PauseMenu. This is why you can see the game still being active in the background, when pausing. Our game consists of four appstates; MainMenu, LoadingScreen, Game and PauseMenu. When a state has been “attached”, the update-method is called from the game engine (JME) and there the state of the application is updated.

2.1.2 Managers

When developing for JME, you often use manager classes that, for example, handles the loading of files, loading of graphics, physics interactions, input, etc. During the development a whole lot of managers was created, which simply acts as utility classes, that can be used to handle different materials, loading, sound or game worlds. The managers often communicate with each other, when the GameMapManager uses its method cleanup(), it tells every other manager to use their cleanup() method.

2.2 Software decomposition

The system is made up of five major packages and smaller domain-specific packages in those major packages.

2.2.1 Decomposition into subsystems

The list of packages:

- application
- controller
    - entityControls
    - managers
- model
- utilities
- view
    - effects

- entity
- gui
- maps
- sounds
- viewport

2.2.2 Model

The classes in the model represents both concrete and abstract objects of the world. Such as Player, AExplodingObject and Cannonball. These classes are used for handling logic and storing and updating data about the objects in the world. Most models has an update method, that gets called from the controllers update methods, and does just what it sounds like, it updates the state of the model. This could include turning a vehicle with it steering value, or lowering the hitpoints of a vehicle.

Some of our powerups references the player, the reason to this is that the powerup has to know which player and vehicle it should activate the powerup on. We use the method usePowerup(player) for this, but sometimes the powerup has to store the player as an instance variable since the powerup might be active for some time and should reset some attributes when its life time has ended.

Our exploding objects that can do damage on classes that implement IDamageableObject also references the player that launched/dropped the exploding object. The reason here is that when the projectile deals damage and potentially kills another players vehicle, we have to increment the kills for the player that sent the projectile.

2.2.3 View

Several objects are considered view objects, such as the game world, hud, menus etc. Those objects update when the model signals that some information has changed. The menus and HUD, 2D objects are fully separated from the model and controller but since we build this game around a game engine the game world and world objects needs to be a hybrid solution.

When you use jMonkeyEngine some information that usually would be found in the model are found in the view. The collision shapes (the physics shape) of objects is created by the view. This is because they most of the time are dependant of the view representation. A missile collision shape should be exactly the bounding box of the 3D-model shape.

The view also contains different maps, or GameWorlds as they are called. A map has the responsibility to load the map terrain and sky box when called. Because it intuitively feels like the map is a view, these map classes were created. This decision is also backed up by fact that you load different data, such as textures on different maps. Thus a class for every map is motivated

and with the use of polymorphism the map can easily be replaced by others.

The model are saved as “user data” on the view entity with a keyword, this is used to notify the right model when it collides. However this requires the models to implement the JMonkey interface Savable which specifies two methods to implement; read and write. However since we do not currently support saving, those methods are left unimplemented and throws an UnsupportedOperationException if called. This will later be changed but we had no time to implement saving properly.

2.2.4 Controller

The controllers in the application are responsible for handling input and collision related events, and redirect this to its assigned model. The controller also calls the update of its model when needed. In JmonkeyEngine there are some useful classes that we extend to simplify the controlling, such as AbstractController, which we use for many of our world objects, and VehicleControl for the vehicles. When collision shapes collide the controllers are responsible to alert its model that it has collided, and then the model decides what needs to be done. This is achieved by extracting the model “user data” from the colliding spatials with the same keyword as specified when the model was saved. When the model is extracted it can be notified that it has collided.

2.2.5 Layering

N/A

2.2.6 Dependency analysis

The system relies on JMonkey Engine which in turn depend on a number of libraries:

- Lightweight Java Game Library (OpenGL)
- OpenGL
- JBullet
- Nifty GUI

2.3 Concurrency issues

The JMonkey Game Engine manages the concurrency when it comes to loading assets, instantiate objects etc. So this is well in order. But to further optimize this our time consuming calculations could be moved to a separate thread away from the game engine then moved or nestled into the update loop. This could potentially cause some concurrency issues.

2.4 Persistent data management

For both the model and the view the application needs to contain static variables inside of them

in order for the application to work properly. (The features of the tank, for example, speed, mass, etc) These values are hardcoded in the according class. So if you want to change the tanks speed, you simply enter TankModel and edit the static variable. This solution is not optimal for allowing user configuration, but was reasoned to be the easiest solution to understand if a new programmer looks at the source code.

2.5 Access control and security

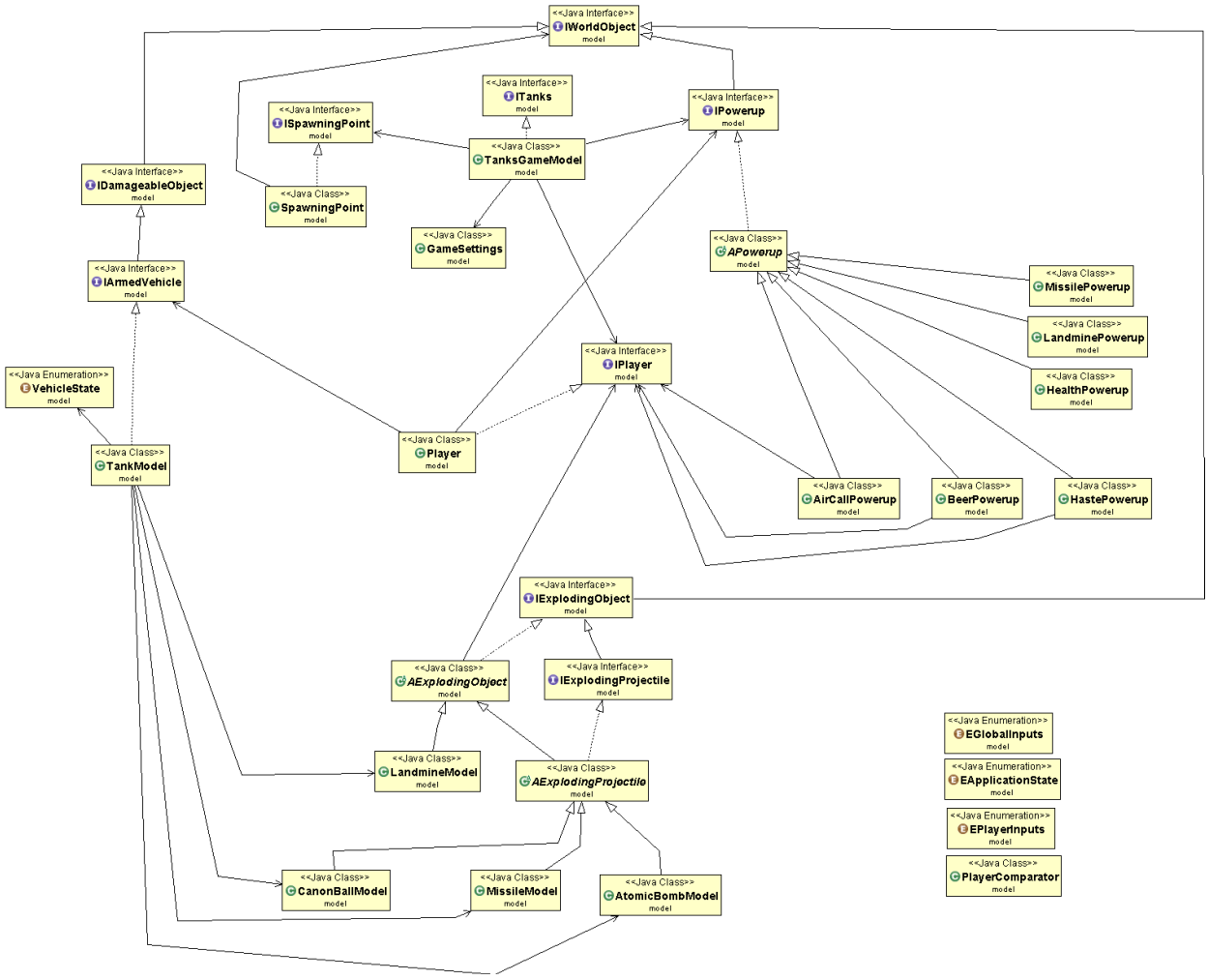
N/A

2.6 Boundary conditions

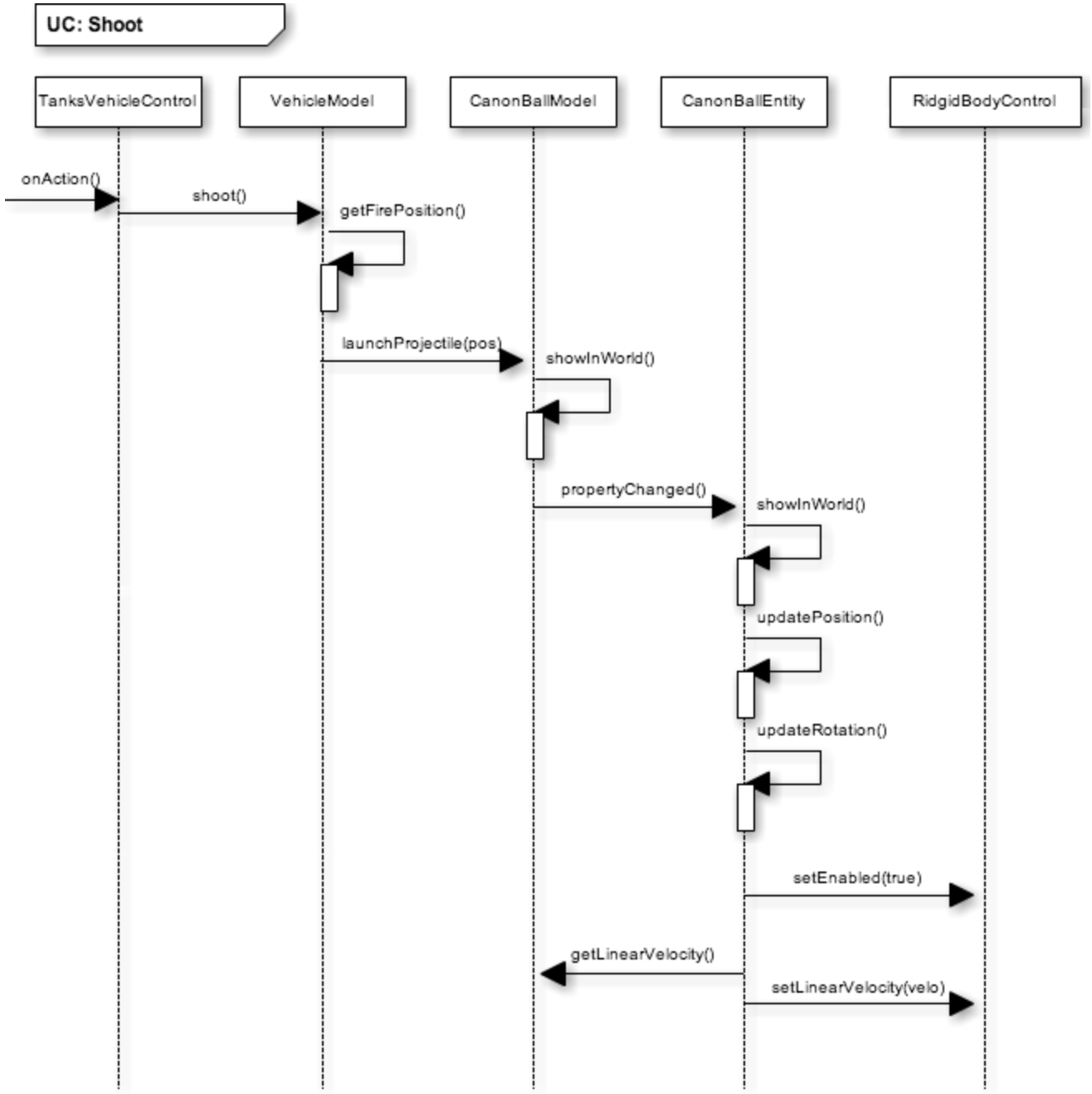
The application runs as a desktop application on any operating system that can run Java Runtime Environment.

3 References

UML Diagram of the model package

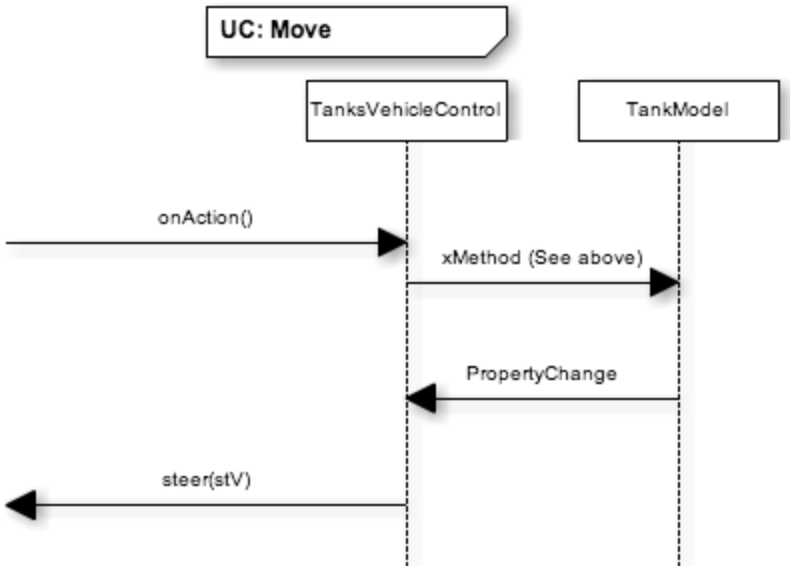


Application Flow



The rest is handled by the game engine

xMethod is either steerLeft() or steerRight()



xMethod is either accelerateForward() or accelerateBack()