System design document for Twirl

Version: 1.1

Date 26/5 - 2013

Author Eric Arnebäck, Linnéa Andersson, Andreas Wahlström, Johan Gerdin

This version overrides all previous versions.

Table of contens

1 **Introduction**

1.1 Design goals

The design should use MVC in a way that makes it easy to reuse the model when porting the game to other platforms. The physics engine should also be easily replacable. The model should be testable.

1.2 Definitions, acronyms and abbreviations

Box2D The physics engine we're using.
AndEngine the game engine we're using
Physics Body The body class is a class in Box2D used to represent the physical body of a shape in the game.

2 System design

The game almost uses a MVC model. Only the actual game logic has been cleanly separated into MVC. The menu system and high score list is instead divided up between the the view and the controller. This was due to time constraints.

2.1 Entities

Entities are the objects shown in the game, like planets, enemies, and the player. The ancestor of all the entities is the abstract Entity class. But no entity used in the game directly extends Entity, they rather extends one the classes CircleEntity, PolygonEntity, or RectangleEntity. Which one they chose to extend depends on their shapes. I.e., Obstacles are polygon-formed so they extend PolygonEntity.

CollisionResolver is responsible for responding to collisions between entities.

2.1.1 Power-up

One type of entity is the Power-up. Power-up:s are box-shaped entities, and when the player picks one up a special effect is activated. Only the Player-entity can pick up power-ups.

To create a new power-up, first extend Power-up, then pass down a power-up duration(in seconds) to the base constructor, and then implement activateEffect and deactiveEffect. Then also specify a color the player will be colored with after picking up the power-up. An optional text can also be specified for the power-up. A fading text will be shown in the view if a text is specfied.

The class PowerUpFactory is used when placing out new Power-ups for new levels by the random level generator(will be discussed in a later section). When adding a new power-up, you must also add it in the PowerUpFactory to make sure it is placed out in the random level generation. And you must also change the constructor of RandomPowerUp when creating a new Power-up.

2.1.2 Physics

Every Entity has an IPhysicsEntity. This interface has all the physics functionality that the entities in the model needs. By using this interface we can ensure that the model is independent of the physics engine used. It can be used to retrieve the velocity of the physics body that represents the entity, for example. The view implements this interface. So all the physics functionality is in the view. This is because this was the most convenient way of doing it. Box2D and AndEngine were too tightly coupled to separate the physics functionality out of the view into a package of its own.

IPhysicsWorld exposes the physics functionality that GameWorld needs, which we will now discuss.

2.1.3 GameWorld

GameWorld implements IGameWorld, which is the top-level interface exposing functionality of the model. IGameWorld is used to get information about the current state of the game(like whether the game is over), and to update the game(with touch input).

GameWorld is like the gateway to all the other model functionality. All the entities of the game are managed by GameWorld by using the package private class EntityManager. It also has a PowerUpList; this class is responble for properlu activating power-ups when the player picks them up, and also deactivates them when the power-up duration is over.

2.1.4 RandomLevelGeneration

The application implements a class RandomLevelGenerator which is responsible for randomly creating entities and placing them out at empty positions for new levels.
In randomly generated levels, we must ensure that entities are not placed on top of each other, and so we need some kind of collision handling. Box2D was to inflexible for this purpose, so we implemented collision handling from scratch using the Separating Axis

Theorem. This functionality is implemented by the package model.util.sat

2.2 Software decomposition

2.1.1 General

- controller, the controller classes.
- model.entity, the entities of the app and the gameworld(manages all the entities in the game, and also handles level change and the like)
- model.powerup, the power-ups of the game.
- model.resource, basically classes for mapping resources to file names. These resources are then loaded and displayed by the view.
- model.physics, interfaces that describe the physics functionality that the model needs. There is also a vector class here that we nicked from box2d
- model.util Utility classes used in the model.
- model.util.sat, Implementents the separating Axis Theorem to handle collisions. Used in ramdom level generation.
- View, the view related classes.
- View.scene, Scenes are separate screenes of viewing in AndEngine; for example, the high score list is one scene and the main menu is another scene. All the scenes of the game are stored here.
- View.entity, the views of all the entities are stored here.
- View.loading, utility classes for the loading scenes are stored here.
- View.andengine.entity, AndEngine also has a class called Entity. We needed to extend this Entity class a couple of times for some classes in the view, and these classes are stored here.
- View.resource, the resources that the model wishes to show to the user(sounds effects and images) are stored here, and of course shown by the view. As familiar, the paths to these resources can be found in the model.resource package.
- View.loader, classes that simplify the loading of resources in AndEngine
- view.opengl, AndEngine has no support for primitives like polygons and circles, so we had to implement that from scratch, using opengl. The classes that handle this are put in this package.
- View.physics, these classes expose the physics functionality from Box2D that we need.

The interfaces in model.physics are also implemented here.

2.2.2 Decomposition into subsystems

We don't have any subsystems. Everything is handled in one, big system.

2.2.3 Layering

The layering is as indicated in the Figure “package structure” below . Higher layers are at the top of the figure.

2.2.4 Dependency analysis

Dependencies are as shown in Figure “package structure”. There are no circular dependencies.

2.3 Concurrency issues

Practically all multithreading is handled by AndEngine. The only times where we use our own threads are during the loading scenes and when removing all bodies from the physicsworld. The loadings scenes utilizes a worker thread to load all game resources, meantime in the main thread a loading animation is shown. However, those are shown for a such a short period of time, plus they don't listen to any user input, so they should not pose any problems.

Box2D bodies must be destroyed when they are removed from the game(ie, when the entity has been eaten). Otherwise we'll have a memory leak. But this deletion must be done on a separate thread, called the update thread by AndEngine, otherwise it will crash. And that is why are temporarily starting a new thread when deleting bodies.

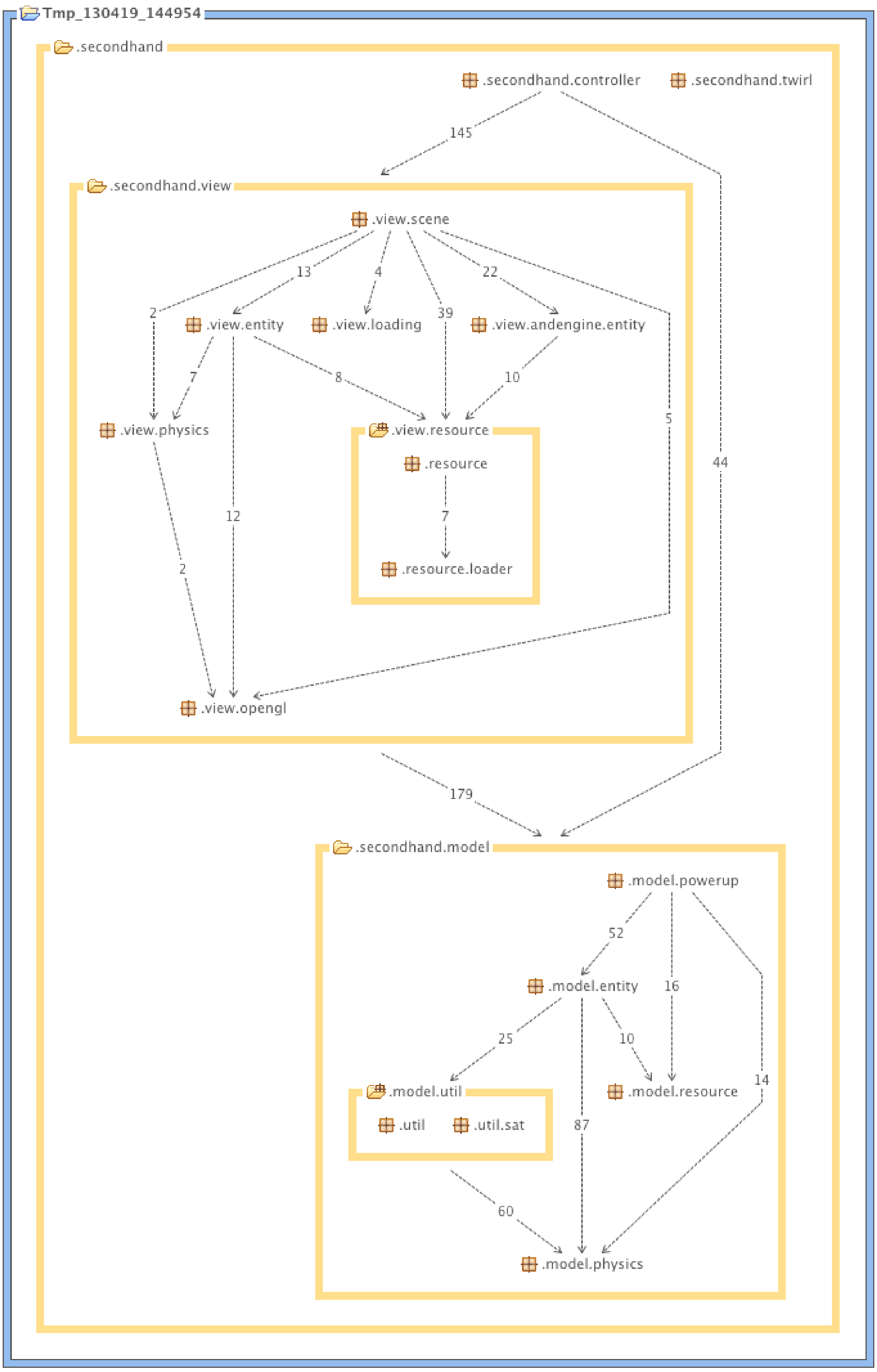


Figure: "Package structure"

2.4 Persistent data management

- Localization files for the localizible text of the app will be stored using XML, using the system for localization offered by the android OS, using strings.xml files. See http://developer.android.com/guide/topics/resources/localization.html
- The high score list is stored using a flat ASCII-based format. See the appendix

2.5 Access control and security

NA.

2.6 Boundary conditions

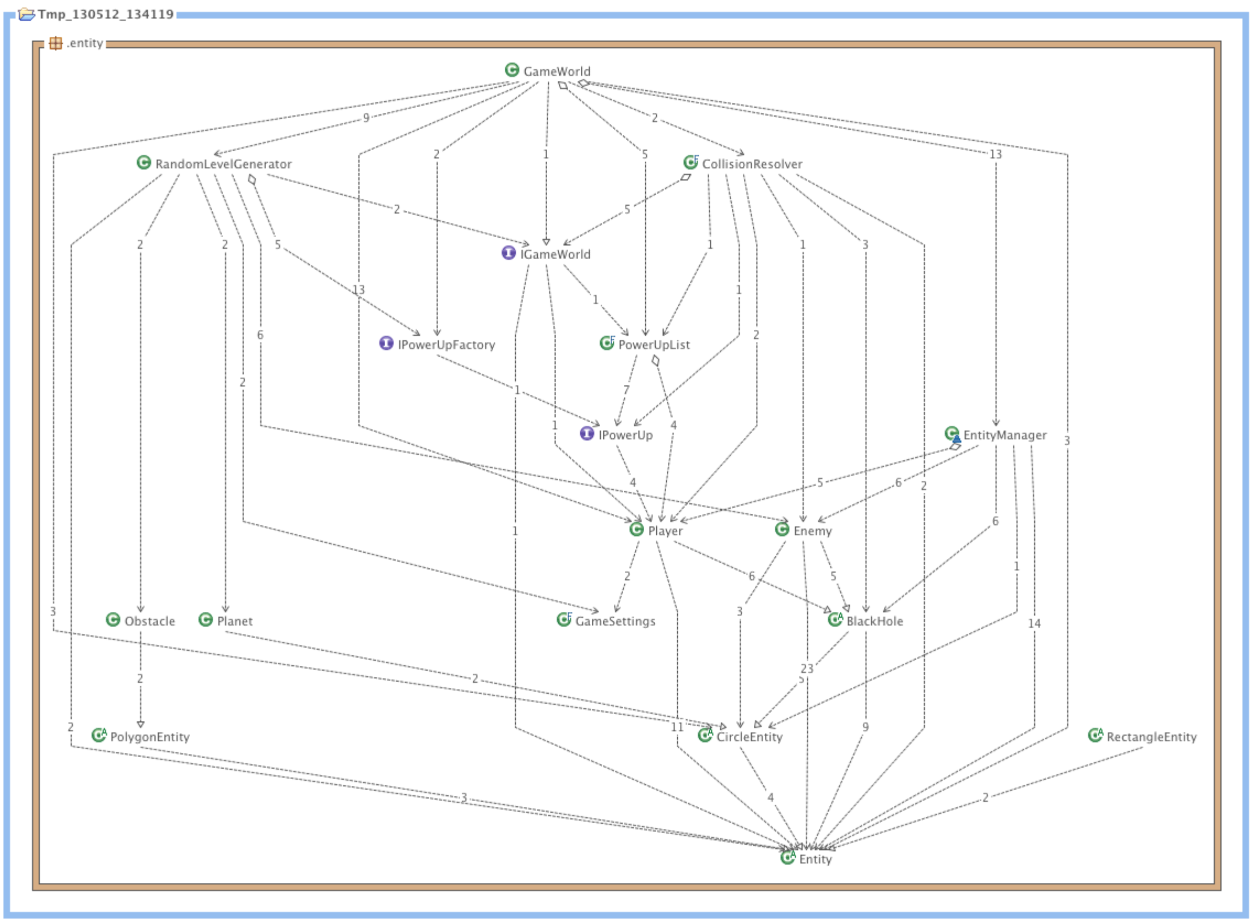
The app is launched like any other android app, by clicking on the icon. It is exited by pressing the back button. By pressing on the home button the application will freeze and wait for the user to procede playing when he/she reenters the app.

3 References

APPENDIX

The sequence diagrams can be found in sequence_diagrams/. They are the two pdf files.

Class diagrams for packages

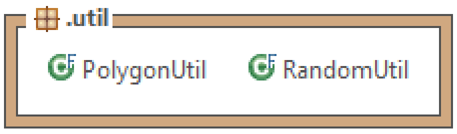


Model.entity package class diagram

- Entity, the base class that all entities extend from
- CircleEntity, circle-formed entities derive from this class
- PolygonEntity,polygon-formed entities derive from this class
- RectangleEntity,rectangle-formed entities derive from this class
- BlackHole, all types of black holes(enemy and player) derive from this class. Has functionality for eating other entities.
- Enemy, an enemy, AI-controlled, black hole
- Player, a player-controlled(by touch input) black hole.
- Obstacle, an uneatable, polygon-formed, obstacle in the game,
- Planet, an eatable planet in the game.
- IPowerUp, interface that all power-ups classes implements.
- CollisionResolver, helper-class resonsible for resolving collisions between entities.
- EntityManager, manages all the entities in a list, and is responble for updating them

every frame, and makes sure that removed entities are removed from the world.

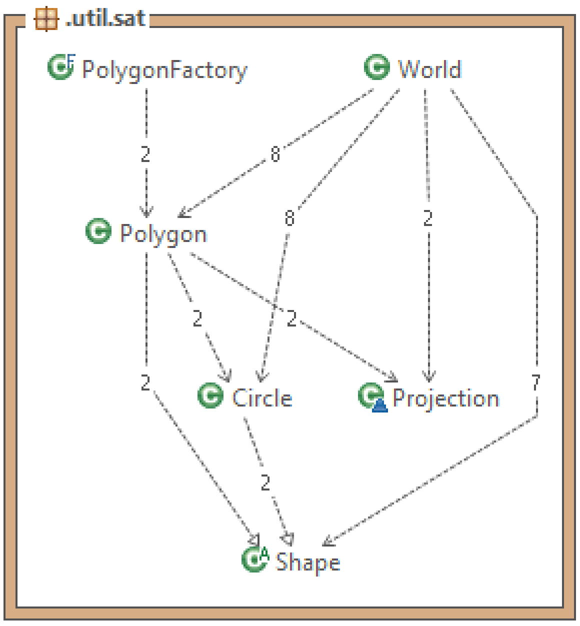
- GameWorld, the world in which the game takes place. Responble for transitioning to the next level. Uses RandomLevelGenerator to create the next level, and EntityManager for managing all the entities of the world.
- IGameWorld, interface that GameWorld implements.
- RandomLevelGenerator, used to generate new random levels.
- GameSettings, singelton class for storing the game settings of all games. Is manipulated in the menu system, in the options scene.
- PowerUpList, used for managing the power-ups of the player. It activates and deactiviates(when duration time has run out) power-ups for the player.



Model.util package class diagram

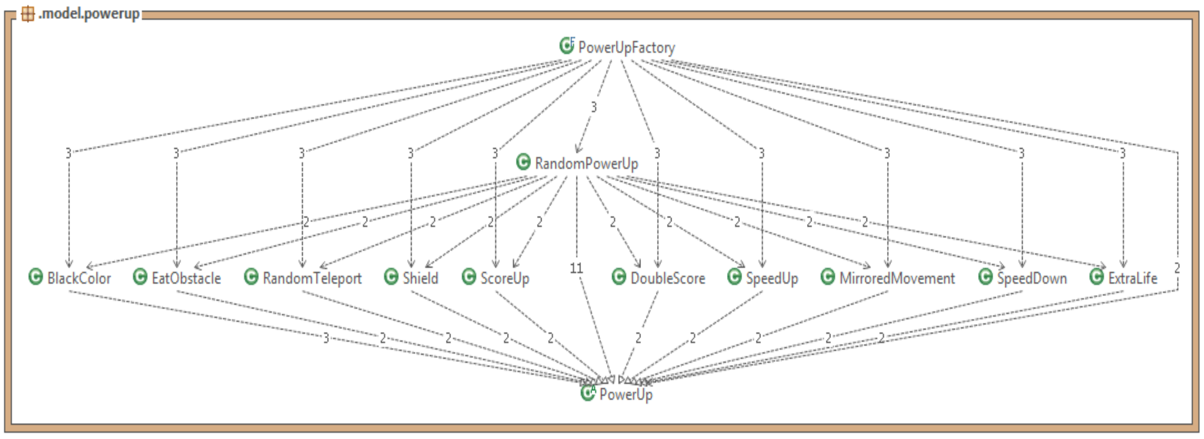
- PolygonUtil, utility functions related to polygons(obstacles are polygons)

- RandomUtil, utility functions for random number generation.



Model.util.sat package class diagram

- Shape, A geometric shape
- Circle, A circle shape
- Polygon, A polygon shape
- PolygonFactory, utility functions for creating polygons.
- World, A world of shapes. The size of the world can be fixed. You can ask this class whether a shape is colliding with some other shape in the world. You can also add shapes to the world(used for random level generation)



Model.powerup package class diagram

All the power-ups have already been described in the RAD, and PowerUpFactory has also already been outlined, so they are not described here.

File Formats

The format for storing the high score list is a simple ASCII-format simple. A list of 5 persons is stored as follows:

```
Name
score
Name
score
Name
score
...
```

So name and score is separated by a newline character. And the separate entries are too separated by a newline character The file is assumed to be sorted, so the first name found in the file is the one with the highest score.