15/05/2015

Emergent Architecture Design

Games Context Group 1 (a.k.a. Funky Donkey Studio)

Danilo Dumeljić	ddumeljic	4282442
Stephan Dumasy	sdumasy	4286723
Dennis van Peer	dvanpeer	4321138
Olivier Dikken	odikken	4223209
Jonathan Raes	jmraes	4300343

Abstract

This document describes our games architecture design. This includes its stucture, subsystem decomposition, design patterns and more..

(Note the word "currently" has many occurrences. This can be read as a synonym of "at the time of writing")

Delft University of Technology

1. Introduction

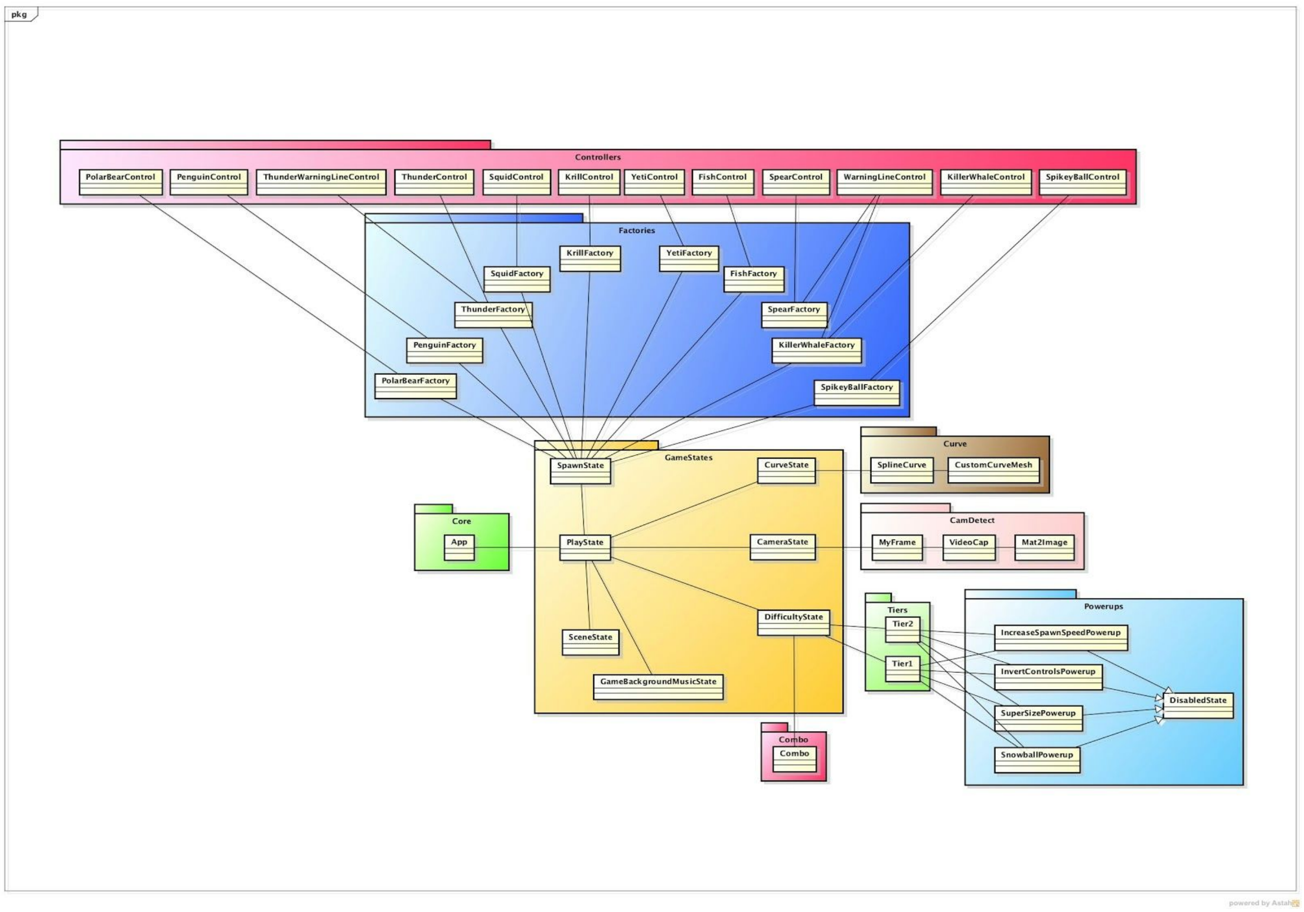
1.1. Design goals

We will try to keep the system structure as simple as possible (without having a negative effect the end product), because we have a very specific setup which imposes many constraints on the possible locations where the game can be realised. Our hardware setup needs to be able to perform correctly for a real queue, without being an obstacle to the environment. Furthermore the player input via camera detection needs to be able to run smoothly in real time for the product to be playable. Our game does not require persistent data, nor does it require concurrency.

2. Software Architecture Views

2.1. Subsystem Decomposition (modules and dependencies between them)

First a general overview of our code structure:



We have created several modules to logically divide our code, and define dependencies between them. Like this, the gameplay objects, the curve formation, power-ups and camera detection modules can be worked on independently.

The game starts with the `App` in the core package. Our `App class,` extending from the by jMonkeyEngine provided `SimpleApplication` class, is useful because it provides a way to run setup code prior to starting the game and deconstruction code on exit. `App` initializes the `PlayState` class, which in turn initializes the other state classes that define the game, like `SceneState` that initializes the environment models and `SpawnState`.

Our `SpawnState` class takes care of spawning the game's objects, the `SpawnState` has a hashmap containing concrete instances of our `FactoryInterface`. The factories implementing this interface are responsible for creating target-, penguin- and obstacle-objects. An example of this would be the `PenguinFactory`. `PenguinFactory` defines a game Object (called `Spatial` in jMonkey). When SpawnState calls `makeObject()` on a the `PenguinFactory`, it returns a `Spatial,` which can be added to the scene.

Our `CurveState` class updates the curve using the data it receives from the camera detection module. It interprets the input from the camera and transforms the dataset into vertices - a datastructure that the representation of the curve (`CustomCurveMesh`) understands. The `SplineCurve` class get its mesh from the `CustomCurveMesh` class, in this class we programmatically make a custom mesh for the `SplineCurve`.

The `CameraState` is the state that keeps track of the camera state (whether it has been opened for example), and communicates dependent information with `CurveState` like for example whether the camera has been initialized, or what the control points from the camera are.

The `DifficultyState` manages the difficulty of the game by enabling and disabling tiers. Currently there are two tiers. Every tier enables / disables specific powerups (which are disabled by default). Moving from one tier to the other increases the game's difficulty, like for example allowing tougher obstacles to be spawned, or less invincibility power-ups.

Design Patterns

We implemented a few design patterns to help us solve specific problems, think creational patterns for instantiating game objects, and behavioral patterns to define communication between objects well, in general all in an effort to make our code modular, and easily extensible.

Abstract Factory pattern - We use this pattern for the creation of all our in-game objects, including the Penguins, different kinds of fish and all the different obstacles that appear in our game. Every object has its own concrete implementation of the `FactoryInterface`, each one creating its own object, all using the same `createObject()` method as defined in the interface. `SpawnState` consequently uses these factories to spawn the different objects through an `ArrayList<FactoryInterface>` at runtime. In spawnstate `Reflection` is used to find and instantiate all the classes implementing the `FactoryInterface`, so that minimal effort is required to add a new object to the game.

Observer / Observable - We use this pattern to keep the HUD elements updated with the current combo count. `DifficultyState` keeps the combos and implements our Observable Interface. Whenever the combo changes its observers are notified. `ComboDisplay` implements our Observer interface and whenever it is notified of an updated combo, it calls `updateText()` to update the HUD.

Bridge pattern - The control points are obtained from the camera detection by the `CurveState` through an interface making use of the bridge pattern. This is done so that the camera detection system and `CurveState` can be modified without interrupting the proper functioning of the application.

Command pattern - The command pattern is used to play sound effects. When a sound effect needs to be played a new command is instantiated that can play the sound, this class implements the interface Sound. Which specifies the play method that plays the sound. This command is then added to the command queue in `SoundState`, where it will be played on the next update loop.

Jmonkey Engine 3

We tried to take advantage of the power of the engine as much as possible. That's why we used a lot of states for important classes. States are really powerful since they contain an update method that is called every cycle and can be enabled/ disabled any time. Also we separated the control from the spatial by using controllers. Spatials can have multiple controls that control their behaviour. We extended the controls provided by the engine with our own abstract controls containing some common methods to avoid duplication in the specific concrete controls controlling the spatials.

2.2. Software & Hardware Requirements

Player Input system

- ❖ Hardware:
  - ➢ Camera
  - ➢ Computer
- ❖ Software:
  - ➢ Camera image describes silhouettes (background - foreground)
  - ➢ Algorithm detects highest points on silhouettes and generates a dataset which will be interpreted to generate a wave

Game controller

- ❖ Hardware:
  - ➢ Computer
- ❖ Software:
  - ➢ Process player input received from the "player input system"

    - ➢ Position the object, slope, targets and obstacles
    - ➢ Check when the object reaches the target
    - ➢ Handle event object-obstacle collision
    - ➢ Keep track of combo count
    - ➢ Decide when to display powerups and when to trigger bonus levels
- ❖ Makes use of data from:
    - ➢ Player input system
    - ➢ Game physics system

Game physics system

- ❖ Hardware:
    - ➢ Computer
- ❖ Software:
    - ➢ Set gravity, friction and other physics rules
    - ➢ Compute the line / slope using the game controller data
    - ➢ Compute how the object moves along the line
- ❖ Makes use of data from:
    - ➢ Game controller

Display system

- ❖ Hardware:
    - ➢ Screen/Projector
    - ➢ Computer
- ❖ Software:
    - ➢ Output the positions of the gameplay elements to the display
    - ➢ Output the GUI to the screen
    - ➢ Draw the graphics (textures, particle systems…)
- ❖ Makes use of data from:
    - ➢ Game controller

2.3. Hardware/software mapping (mapping of sub-systems to processes and computers, communication between computers)

Camera mapping

We need to process the feeds of several cameras on one system. To this end we will be using openCV to create silhouette descriptions of the players. We will use the Bridge pattern to decouple the abstraction from the implementation, separating the camera detection system from the game, defining a clear interface through which the two will communicate.