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 contains our game's architecture design.

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

1. Introduction

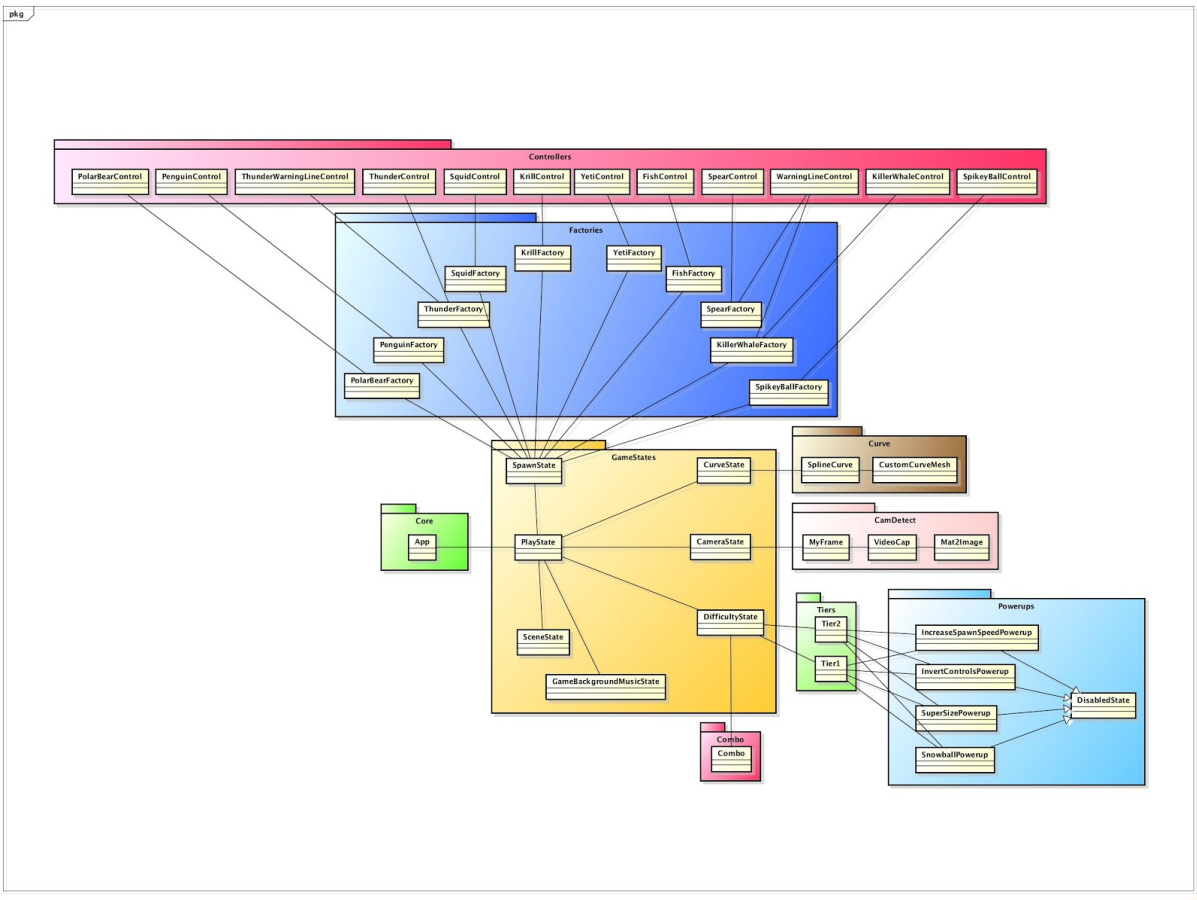
1.1. Design goals

We will try to keep system structure as simple as possible (without having a negative effect the end product), since 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:



.

Our architecture decouples the different aspects of our application. In this way the physics, gameplay objects, GUI and camera detection modules can be worked on independently.

The game starts with the App in the core package. Our App class communicates with the PlayState class, this class initializes the other gamestate classes and attaches them to the StateManager. We have different GameStates that all have their own possibilities.

Our SpawnState class takes care off all the spawning, the SpawnState has a hashmap containing instances of every factory. These factories are responsible for creating objects for targets, penguins and obstacles. We implemented our factories with reflections, this way it is very extendable and new spawnable objects can easily be added.

Our CurveState class takes care of most of the attributes for the curve. It handles the input from the camera and adjusts the curve every update cycle. The SplineCurve class get it's mesh from the CustomCurveMesh class, in this class we programmatically make a custom mesh for the SplineCurve.

The CamDetectState is the state that handles the camera and it's input.

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), the higher the tier how harder the powerups should be.

Design Patterns
We used quite a few design patterns to make our code even better.

Observer / Observable - We use this pattern with the combo and the DifficultyState. The combo is the observable and the DifficultyState is the observer. So every time the combo changes the DifficultyState is notified and can change the difficulty accordingly.

Bridge pattern - The control points are accessed by the CurveState 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

Reflections - We made use of reflections in our factories. This way we can easily add new spawnable objects.

Jmonkey Engine 3

We really tried to take advantage off the power off the engine. That's why we used a lot of states for important classes. States are really powerful since they contain an update cycle and can be enabled/ disabled any time. Also we put all of the behaviors from spatial classes in their own control classes. This way spatials with can have multiple controls which adjusts their behaviour. And controls can be reused for multiple spatials which avoids the duplication of code.

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. see also https://sourcemaking.com/design_patterns/bridge.

2.4. Persistent data management (file/database, database design)

-

2.5. Concurrency (processes, shared resources, communication between processes, deadlocks prevention)

-

3. Glossary

-