University of Utrecht

Pong

Documentation supporting practical assignment 1 of the course Gameprogramming, minor Game- and Mediatechnology

Derk-Jan Karrenbeld and Max Maton
F121502 / F121713
9/30/2012

Introduction

This documentation is part of the first practical assignment of the course Gameprogramming (INFOB1GP) and should be regarded as such. In this document we discuss and provide additional information and insights on the way we designed our version of Pong. The code is both commented and self-documented by namespaces, naming conventions and general program flow, but this should document should ease any necessary clarifications. The document tries to cover all the requirements set by the practical assignment.

Contents

Game state management

We divided the game flow in separate game states. Each state consists of one or more different displays we call screens. For example there is a TitleScreen, a MenuScreen, a GameOverScreen and a PlayingScreen. To direct the game flow there is a service called the ScreenManager which dictates what screens are updated and drawn and processes any transitions between two or more screens.

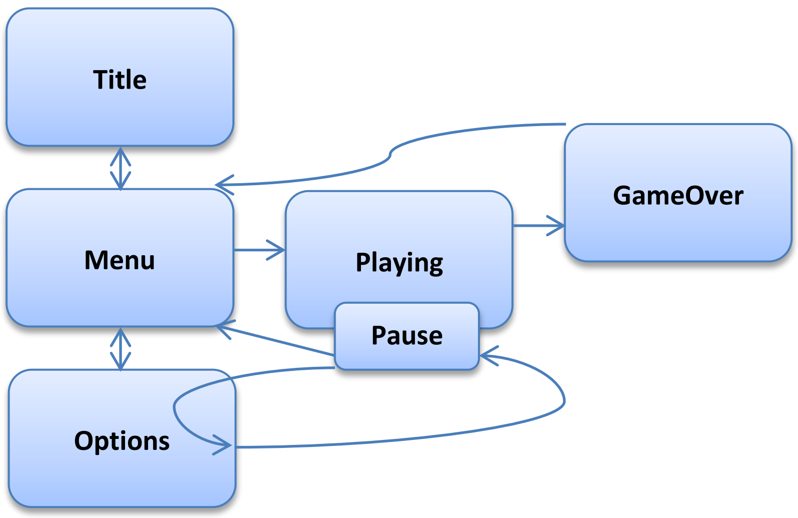

Figure 1. Gameflow in Pong

A screen is made visible by adding it to the manager and the transition will kick in. We generally used a fade effect where we fill the complete backbuffer with black pixels. A screen will hide if a screen is placed on top that is not a popup or if a screen's exit command is called. Using a ScreenManager new screens are easily added or changed. All the code to display a certain state in the game flow is all bundled in one neat class.

Input management

The input is captured by the InputManager. It is a service that can be accessed from any object with a reference to the service or the service container (available in the Game class). A function was created to mimic the key presses in any other windows program where pressing a key yields a key press, pauses and then keeps yielded presses at a steady rate.

The input for the screens is read out in the HandleInput function which is only called if the game is active (so focused and not blurred). This way you can't accidently exit your game if you switch out of the game into another screen. The input for the paddle is read out from the PaddleController subclasses. This can be a GamePadController, KeyboardController or AIController. A key press or AI action results in setting a property of the controller. This property tells the game which way the paddle should move.

Mouse support

We explicitly chose not to support the mouse because either you have a huge advantage positioning the paddle, or you need to limit the speed of updating the paddle. In any way, the gameplay is not positively affected if one of the players is using a mouse.

Four player support

Since we had no gamepad to test with, we were not able to test four players. The game is perfectly capable handling more players with some additions to the controller direction output. However, a typical computer can only handle 3 key presses at the same time. Supporting four players on the keyboard is – not even regarding the lack of space – not feasible.

Input selection

When selecting input you have the option to pick two AI's or both players using WASD. We chose to support this so you can play single player games with yourself, for testing purposes and for watching a battle between two artificially controlled paddles.

Artificial Intelligence

The easy AI simply tries to center the paddle on the ball's center. The hard AI interpolates where the ball will be once it hits the side of the screen and tries to go there. The end position of the ball is calculated by multiplying the y/x speed with the distance to the wall. The resulting y position is then bounced mathematically off the walls of the level to calculate where the ball will end.

Game Structure

The Ball and Paddle both interface the same ICollidable. All ICollidables can be registered in the CollisionManager. This service box-collide all collidables with each other and calls a handling function when a collision occurs. We did not add bullet collision checks (interpolate the ball to see if it just hit or is going to hit an object), so in theory the ball could skip through the paddle into the DeadZone, also an ICollidable. To counteract these problems we enlarged the paddles. Since we draw them on the edge of the screen, this didn't yield any problems. If not, we could have drawn part of the paddle with the color TransparantBlack.

The PlayingScreen receives a Level object. This object contains all the paddles, balls and players joining this game. It is the main data unit of our game and is shared with almost all objects it also holds. The level object can't draw itself, so that is handles in the screen. The reason for this is that not all actors are necessarily drawable.

The Player objects hold the number of lives a player still has. These are diminished by one each time a player's DeadZone collides with a Ball. These lives are drawn in the Heads-up Display mentioned later in this document. The Ball object manages its own collisions and appropriate sounds. The Paddles manage their own movement.

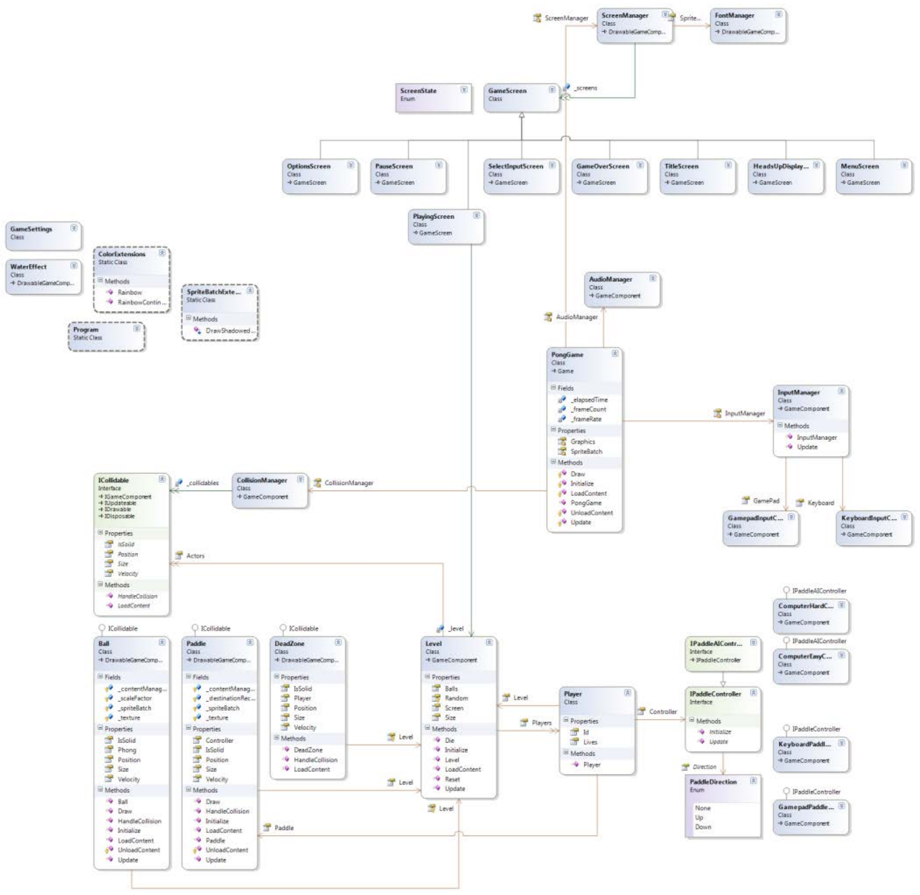

Figure 2. The image is available in the folder this document resides in.

Graphics

Each screen has its own ContentManager. The reason for this is that you can safely unload all assets when you are done with a screen. There is no point in hogging these resources. Fonts and Sounds are loaded in their respective managers and are preserved between screens. The assets are only loaded the first time they need to be accessed.

Background Water effect

A big part of the visuals is the water effect influenced by the ball and paddles. This water effect is created using a simple algorithm popular in the demoscene.

The algorithm represents the water as a raster of water heights, internally represented as a texture. Each step the velocity of the water is defined by subtracting the current height from the average height of the surrounding pixels. This creates a typical ripple effect. The ball sets the velocity of the water to -1 each turn which causes the water to lower at that spot.

The background texture is then drawn by offsetting the sampled position of the texture with the delta-x and delta-y of the water heights to simulate diffraction.

Ball Phong Shading

In order to create a 3d like ball we use phong reflection model[1] and draw the block like it is a perfect circle by discarding all pixels outside the radius. A light is simulated at the center of the level in order to create a sense of movement.

Heads-up Display

The HUD is a simple popup screen that uses the same blank texture to draw the paddle, ball and lives to create a semi-transparent overlay. The values are read out from the Level object.

[1] http://en.wikipedia.org/wiki/Phong_reflection_model