Project Technical Document

Team 06: Air Hockey for Pixelsense

This document contains technical and lower level information for the PixelSense project.

Note: This is a live document which details the lower level aspects of the project. The information contained in this document is highly subject to change and will expand over time as the project continues to develop.

Contents

TD Section 1 - Development Details

1.1 - PixelSense Product Details

- Version 2 of the Microsoft Surface
- Uses a standard Windows 7 Operating System

1.1a - Sensors

- Each pixel uses an infrared sensor
- There are no other sensing capabilities (such as capacitive or resistive touch)

1.1b - Screen

- 1920 x 1080 - 16billion colours
- 40-inch screen. (35” x 20”), (54 dpi)

1.1c - Audio

- Stereo Speakers

1.1d - Connectivity

- 2 USB ports on the base of the front
- Wireless LAN

1.2 - Development Tools

1.2a - Microsoft Visual Studio 2010

- Supported by XNA/Surface

1.2b - Microsoft Surface SDK 2.0

- Microsoft’s Surface/PixelSense development kit. This framework will be wrapped to maximise framework independance.

1.2c - Microsoft XNA Framework - C#

- Microsoft’s XNA game framework/engine which will be used for this project. This framework will be wrapped to maximise framework independance.

1.3 - Project Tools

1.3a - Google Drive

- Shared documents will be available to the team for coordination and documentation
    - Game Design Document
    - Technical Document
    - Worklog
    - Research, Links and other information
    - Minutes and Administration

1.3b - GitHub

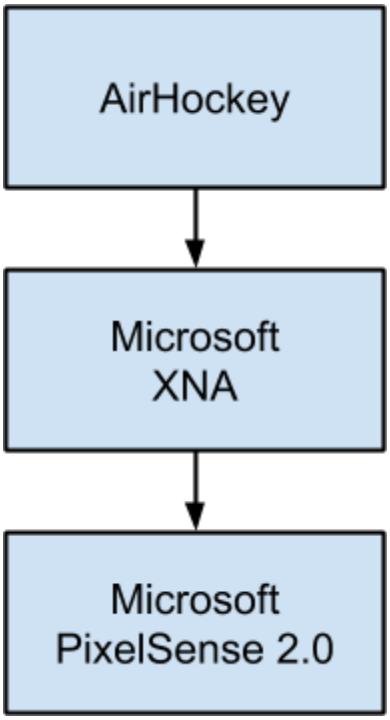
- For version control. Using a swinburne account supplied from Clinton Woodword

---

TD Section 2 - Domains and Modules

This project consists of three major library/functional domains, and development of AirHockey will be in a domain that interacts with the XNA Framework, however will use minimal direct calls to the framework.

A reason for this architecture is to enable the core AirHockey project to remain relatively consistent whilst being able to change frameworks and/or platforms.



AirHockey Domain
This domain is where all method calls and classes will reside in relation to the Air Hockey game. This also includes all the data structures that the game will use.

To keep dependencies to a minimum with an engine/framework, predefined data structures, classes, and methods provided by the framework will be “wrapped”.

Microsoft XNA Domain
The XNA Framework contains functionality aimed at game development for various platforms. It also makes use of the PixelSense framework which enables the AirHockey domain to access data about the PixelSense.

Microsoft PixelSense 2.0 Domain
Direct interaction with the PixelSense is handled by this domain, to which XNA already interfaces with. The AirHockey domain does not interact with this domain directly at all.

Domain Interaction
When developing AirHockey for PixelSense, all method calls will only be within the AirHockey domain.

For example a method call to draw a bitmap by classes in the AirHockey domain will call a wrapper method first (which is in the AirHockey domain). This wrapper will then pass bitmap parameters to the XNA domain to be rendered.

By having a wrapper, switching frameworks will be much simpler and easier. This also takes into consideration that the XNA Framework has been discontinued.

TD Section 3 - Program Structure

There are five projects in which this solution builds to.

Solution: AirHockey

- AirHockey.Constants
- AirHockey.GameLayer
- AirHockey.InteractionLayer
- AirHockey.LogicLayer
- AirHockey.Utility

Layers

AirHockey for PixelSense is divided into five layers, each focusing on different responsibilities of realizing the game outlined in the game design. Further details about classes contained in the layers is covered in Section 4 - Classes and Roles

3.1 - Constants Layer

Purpose

Store global settings and constants. Generally all immutable and readonly fields are held here.

Collaborators

Does not reference any layers

Development Notes

No functions.

Simply put only constants are allowed hence the name and all new constants should be put in the appropriate class. As of 15/05/2014 there are currently three classes: AirHockey Values, Global Settings and Rendering Values.

3.2 - Game Layer

Purpose/Role

Load resources, create views and objects, and also Input.

Collaborators

References the Constants, Interaction, Logic and Utility layers

Development Notes

When creating a new view, create the appropriately named class in the view folder and then also make a folder for that views objects/content.

3.3 - Interaction Layer

Purpose/Role

Entry point of the game and game loop

Collaborators

References the Constants layer.

Development Notes

This layer is finished and should not be touched without consulting the team.

3.4 - Logic Layer

Purpose/Role

Home to all AI and Physics

Collaborators

References the Constants layer.

Development Notes

All AI and Physics functions are contained here. At the moment the physics (collisions) has a PhysicsObject interface that allows us to perform physics calculations on objects when necessary.

3.5 - Utility Layer

Purpose/Role

All utility classes that the Game Layer requires.

Collaborators

Does not reference any layers

Development Notes

Any new utility classes that could be used by numerous objects or views should be placed in here.

TD Section 4 - Design Patterns

4.1 - Singleton

Description: Ensure a class has only one instance, and provide a global point of access to it.
Example: The Physman class that manages physics logic mainly focused on the area of collisions.

4.2 - Component Pattern

Description: A component that is the abstraction for all components. In C# these are typically interfaces.
Example: Example is the IPhysicsObject interface that can be used to define a physics object.

4.3 - Template Pattern

Description: In the template design pattern, one or more algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed.
Example: There are a number of abstract classes that are overridden to allow for behaviour changes. One example would be the Update method of the UserInterfaceComponent.

4.4 - Message Pattern

Description: Allows the exchange of messages between components.
Example: Example is the IMessageHandler which can be used by components to handle messages. In our project it is used to Get and Set data.

TD Section 5 - Template Classes / Components

5.1. Component Classes

All component classes use a common base class called ComponentBase

Audio Component
A common base class for all audio components. Such as a component for playing level music.

GraphicsComponent
A common base class for all graphics components. Such as a component for drawing a View's background.

UserInterFaceComponent
A common base class for all GUI components. Such as a component for drawing the GUI for a menu.

InputComponent
A common base class for all Input components. Such as Player Input and AI Input.

PhysicsComponent
A common base class for all physics components. This can also be used directly since basic physics interactions are handled by the stored data.

5.2. Utility Classes

Vector class
A class which contains data about a vector such as X and Y positions, and contains methods to return calculated values based on that data.

TD Section 6 - Entities

Entities are general purpose objects like the towers or pucks in our game. So far we have created test entities to test input and a particle system. In our current stage of the development we have yet to solidify any game entities yet. We do however know what entities we will require to implement as listed in section 1 of the game design document.

TD Section 7 - Input

Microsoft PixelSense Input

Byte/Identity Tags (TagPoint)
The tags are used to control the movement/placement of the tower objects on to the screen, each unique tag ID will be assigned to different towers the players can have.

The use of tags also enables us to separate the towers functionality giving the location/movement to where the tag is placed which then leaves the other core functionality such as firing projectiles to different input methods.

The tags will also be used to check whether they are still placed on the screen, if they aren't currently placed on the screen the towers will go into re-charging state to re accumulate any lost energy the tower might have lost.

Finger Touch (TouchPoint)
The finger input is used to determine at what location the projectile should be place e.g. worm hole tower, or how far the finger has swiped or moved from the origin point giving us the distance from two points which also gives us the direction of where the project should shoot from and how far it should travel e.g. slingshot tower

The finger touch will also change the states of the towers(active/de-active), when the finger is near the tower say 60 pixels away from the origin point of the tower (the center) it will activate the tower enabling it to in turn fire a projectile by dragging the finger a certain distance and letting go to shoot the projectile, or when activated will enable the player to place a projectile onto the screen, once the projectile has been placed or shot from the tower and the finger is off the screen it will deactivate the tower.

The finger touch will also control menu and in-game options/buttons on the screen and enable the transitions to different views or menu options when a button is pressed.

TD Section 8 - UI/UX

The following section explains how our views, dialogs and objects inherit from one another to form the interface and experience our users will see.

Our project contains a GameViewBase class that all game views/screens inherit from. This class sets all the properties of the view like the skin and background. It can also set the dialog for that specific view.

Dialogs are smaller screens inside a view that we will use to show menus or information. Dialogs inherit from the GameDialogViewBase class that sets its parent view, background, skin, height, width and position.

Views also contain components or objects that will have to inherit from a GameObjectBase class. The base class sets up an input, audio, graphics, physics and user interface component for that object including message handlers.

To know which object belongs to which view there is a GameObjectContainerBase class that the GameViewBase class inherits which contains a list which all objects for a view are added to. The objects in the list are updated and rendered.

TD Section 9 - Audio/Visual Assets

Visual

Assets:

- Puck (multiple themes)
- Towers (multiple themes)
- Goals (multiple themes)
- Tower Board (multiple themes)
- Playing Board/level(multiple theme)
- Pause menu(static design)
- Main Game page
- Level/theme selection page
- About page

Special Effects:

- Projectiles(multiple themed)
- Buffs / traps
- Particles (on board,puck)
- Collision
- Scoring

Audio

- Collision SFX (multiple)
- Scoring/Hitting goal SFX
- Navigation SFX (selecting, pausing, starting game)
- Win Game SFX
- Buffs / traps SFX
- Projectiles/shooting SFX
- Countdown(before game start, after resume) SFX
- GamePlay background music (multiple)