2008:017 HIP

BACHELOR'S THESIS

Multiplayer Games and Physics on Multi-touch Screen Devices

Hans Hartman

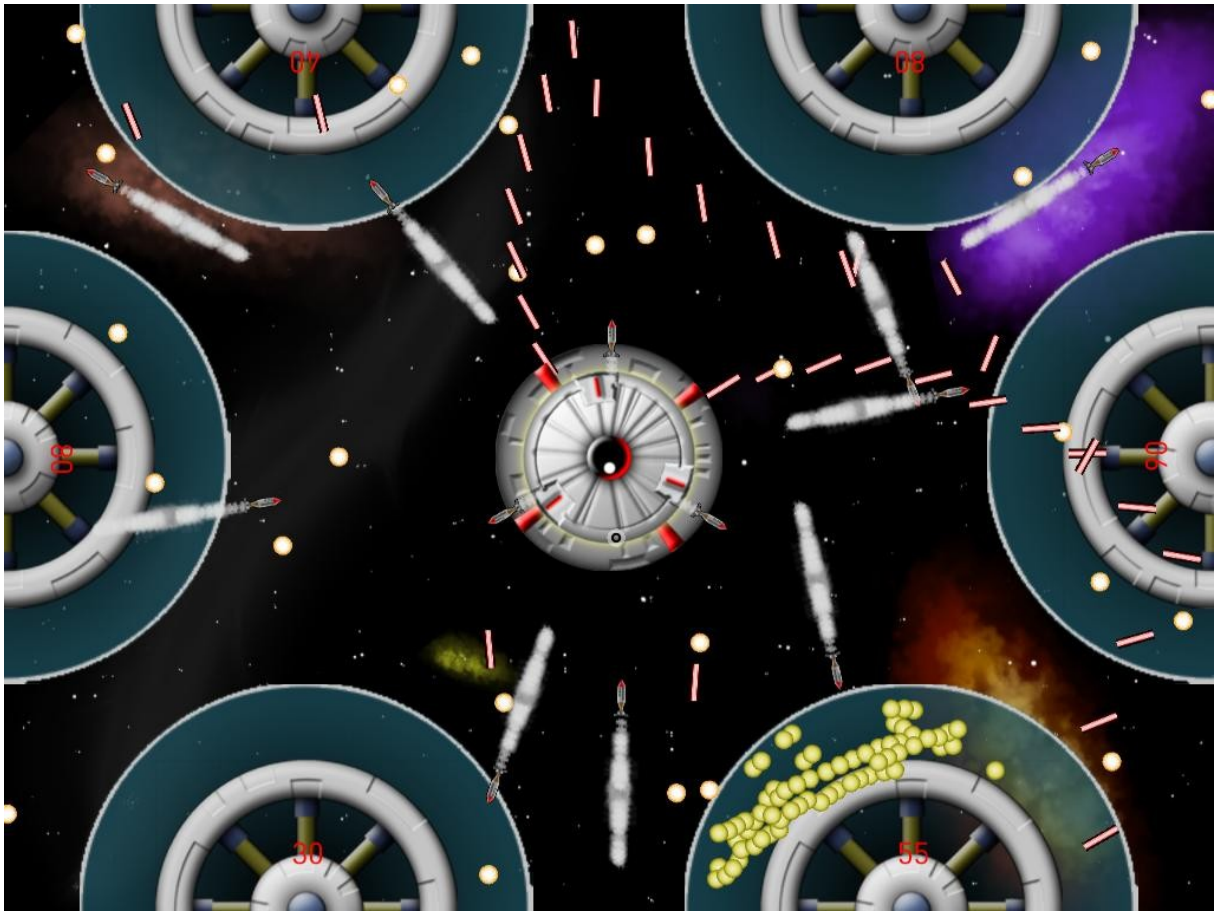
Luleå University of Technology

BSc Programmes in Engineering
BSc programme in Computer Engineering
Department of Skellefteå Campus
Division of Leisure and Entertainment

2008:017 HIP - ISSN: 1404-5494 - ISRN: LTU-HIP-EX--08/017--SE

Luleå University of Technology, Campus Skellefteå
Bachelor of Science degree in Games Programming (GP), 180 ECTS

Multiplayer games and physics on multi-touch screen devices



Hans Hartman

Preface

This thesis project was done at NUI (Natural User Interface), during a period of 10 weeks in the spring of 2008.

Two design students from LTU Skellefteå hade made a game concept for the NUI Framework, I and another student was given the task to make there game a reality.

I would like to thank Harry van der Veen, Johannes Hirche, Mikael Bauer and the other staff at NUI.

I would also like to thank my fellow students doing bachelors thesis projects at NUI, Erik Sundén, Johan Larsson, Fredrik Medeström, Lasse Partanen and of course the game designers, Robert Carlsson and Johan Lövdahl. All of them have been grate assets during this project.

Abstract

The introduction of multi-touch screens in the computer world has opened up new and creative ways for game designers and developers to make multi player games. This paper explains how to design and execute the code for a game on such hardware, and explains some potential pitfalls and ideas for how to get around them. The game was designed by two first year students that studies game design at LTU. So all of our code was based on what there design document stated.

Sammanfattning

Införandet av multi-touch skärmar innom dator världen har öppnat nya och kreativa sätt för spel designers och utvecklare att göra multi player spel. Denna rapport förklarar hur man designar och skapar kod för spel på en sådan hårdvara och förklarar några potentioella fallgropar och ideer på hur man undviker dem. Spelet är designat av två första års studenter som går spel design på LTU. All vår kod är baserad på deras design document.

Index

1. Introduction

1.1 Abbreviations and Terms

NUI – Natural User Interface.
TTL – Time to live
XML – Extensible Markup Language
LTU – Luleå Tekniska Universitet
CPU – Central Processing Unit.

1.2 Goal and Purpose

The goal of this project was to create a working game based on the designers' concept **[1]** and make a physics test application. Our main focus was on implementing the multiplayer Survival game "Project Plasma", later renamed to "Station Defender", for the Natural UI touch screen device **[2]**, this was done in cooperation with the game designers.

1.3 Background

1.3.1 The Game

Station Defender is a game developed for a multi touch screen developed by Natural UI and have been formed to be shown at different shows/expos where people can come up and play without having to wait for their turn to play.

The game's goal is to survive for as long as possible, there is a enemy station in the middle that shoots lasers, plasmas and missiles in all directions and the player have to survive by building different kind of walls that will protect him/her from the shots.

Walls are built by drawing different signs on the screen with the fingers and this is what makes the game stand out from a lot of other games. There will be three different wall types in the game; anti plasma, force field and mirror walls which all differ from each other.

The game play is basically a rock, paper, scissors setup.

Walls / Shots	**Plasmas**	**Laser beams**	**Missiles**
Anti plasma Walls	Bounce	Through	Destroy
Force Fields	Through	Destroy	Bounce
Mirrors	Destroy	Bounce	Through

1.3.2 NUI Framework

As the game is designed with the NUI device in mind, choosing the NUI framework **[4] [6]** was a natural thing to do, and as it was written in C++ using OpenGL among with other software, we decided to use C++ as well.

1.3.3 Box2D

We decided to take a look at Box2D **[3]** and try to make an interface so that you could use it with the NUI framework. As it hade every thing we needed for our physics application. As my part in the actual implementation was limited, the implementation will be left out of this paper.

2. Methods

2.1 Design and Implementation

This selection contains the code design and implementation for Station Defender. We decided to split the game in to the following classes:

2.1.1 Project plasma

This is the main game class, it handles updates and draw calls, it also creates the players and the enemy when initialized. A state handler is used to keep track of which state the game is currently in.

All user inputs are handled by this class and the correct actions are then taken. The games settings are loaded from an XML file, more on the XML files in *selection 2.1.6*.

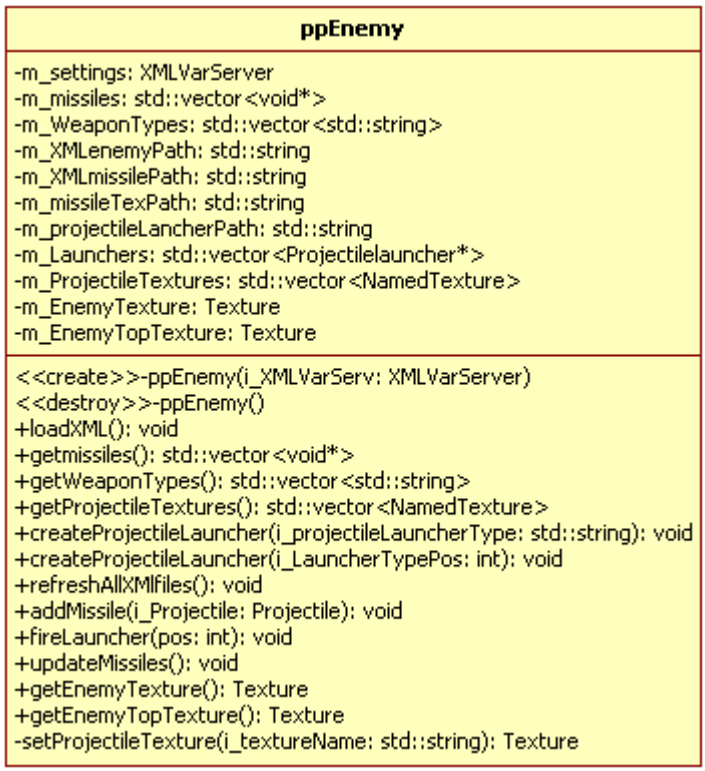
2.1.2 Enemy

The *ppEnemy* class is responsible for creating launchers and tracking the projectiles.

Creating the launchers is done from an XML file and this approach means that there can be any amount of launchers added to the enemy.

It also updates the launchers and projectiles during the update phase. A added feature in the update was the canFire Boolean that indicates if there are any more projectiles allowed on screen at this particular time, this was added as an limiter when the edge bouncing feature is activated.

Edge bouncing is a feature that when activated bounces projectiles that contacts any outer edge of the playing area.



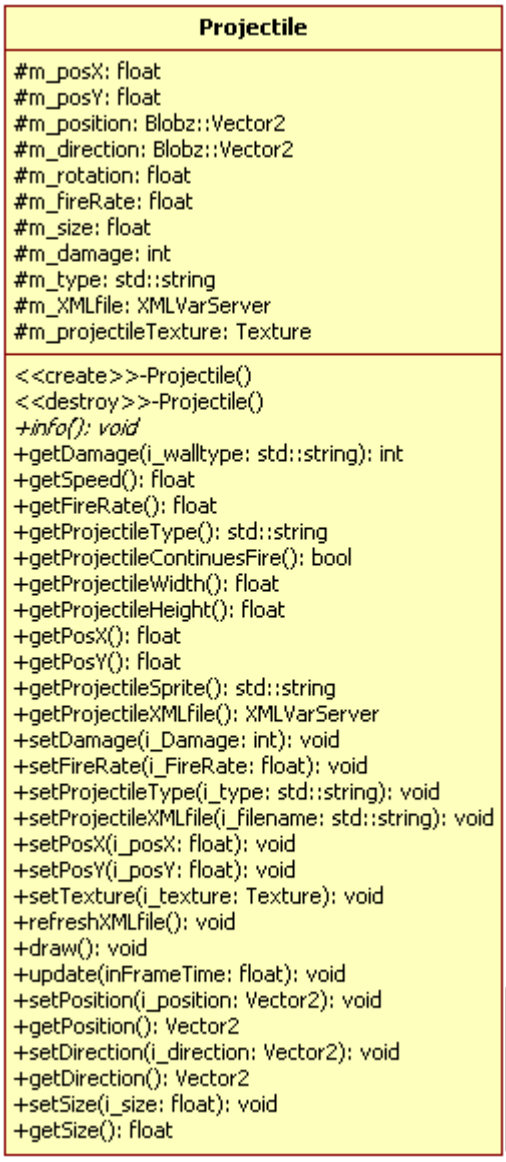
2.1.2.1 Projectile Launcher

The *Projectilelauncher* class main task is to create projectiles and fire them at the players.

The update rotates the launchers and fires the projectiles if the launcher has any projectiles left before it has to cool down and isn't cooling down at the time.

All of the projectile launchers data is scripted in XML files and loaded at creation, at this time the projectile type is identified and the data is loaded from the correct XML file and stored in a ProjectileHolder struct, this data is supplied to the projectiles constructor when a projectile is fired.

2.1.2.2 Projectiles

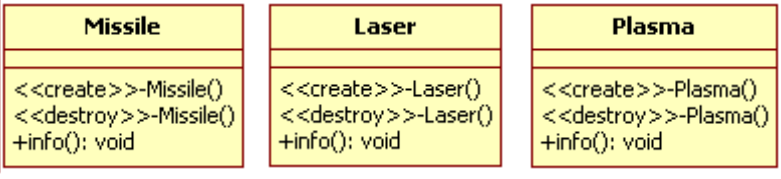


The *Projectile* class is the base class for the three projectile types Laser, Missile and Plasma.

Each projectile keeps track of all information about projectile, like the ID of the last player and or edge the projectile has bounced on. This is to reward players who manage to bounce a projectile and hit another player's base.

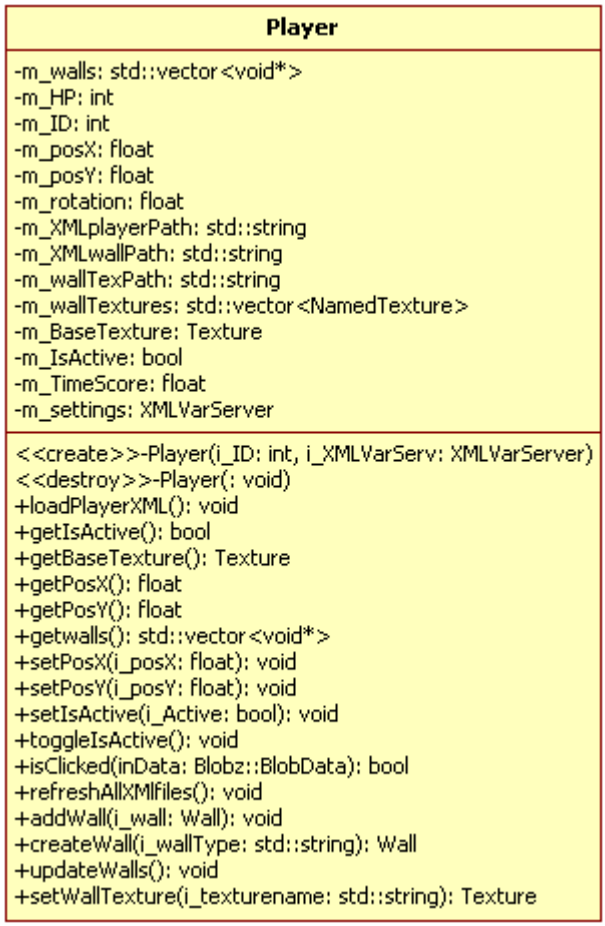
Besides the ID, it also has information about the sound and animation that should be played when the projectile is in flight. The update function updates animation and the position of the projectile. Neither sound nor animation is compulsory and can be left out of the XML without any modifications to the code.

The Laser, Missile and Plasma classes were created so that they could have unique features, this was a decision made early in the design phase and when we later changed some features of the projectiles lost most of its original qualities.



2.1.3 Player

The *Player* class main task is to keep track off which walls the player has created as well as how much life he has and how long he has managed to stay alive.



The update updates the time score of the player and calls the walls update function and then checks if the TTL is less or equal to zero and if it is it then removes that wall as its time is up.

It also contains a couple of different functions to determine wither a users input or projectile is in contact or close to the player area.

These functions include isClicked that checks if a user clicks on a player area that is currently not active; and isWallableArea that determines weather or not it is possible to create a wall at that position.

The function addToFingerVector adds current blob data to the appropriate finger vector. The blob data is the data received from the NUI framework and contains data about a finger "blob" currently interacting with the screen, the data precision is dependent on hardware and software configuration.

Once a user lifts a finger from the screen and there's a finger vector containing data the *createWallChains* is called and it creates a "solid" wall of the type specified by the user.

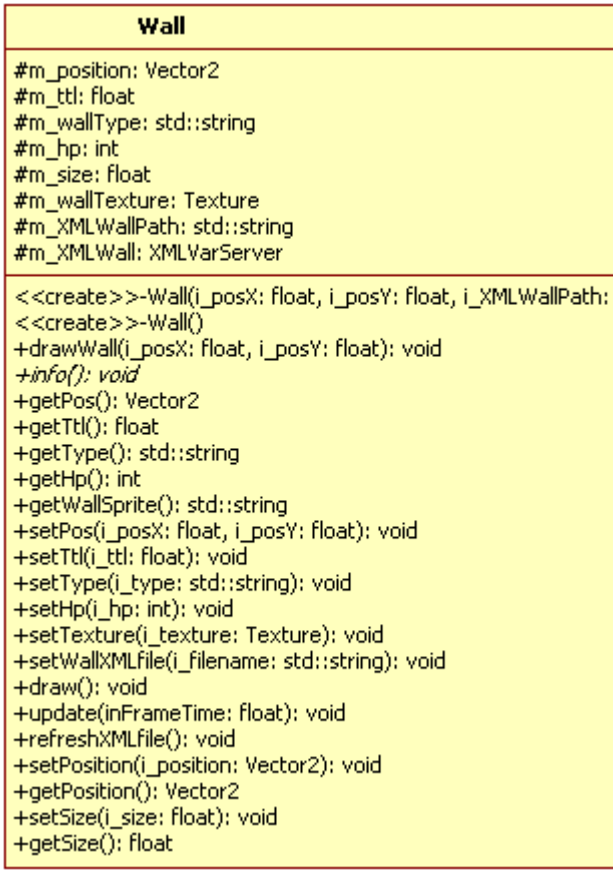
To balance CPU load the *chkFingers* is called to see if the finger vectors are longer than a certain length and if so it calls the *createWallChains* function, this is done in update.

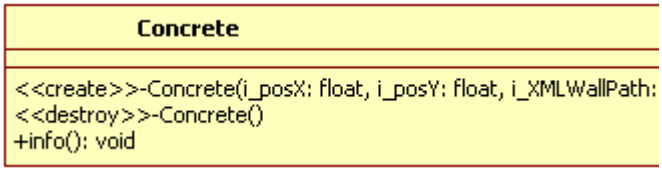
2.1.3.1 Walls

The *Wall* class is the base class for the three types of walls Concrete, this was renamed in the final stages of the creation process and is now called Anti-plasma wall, Mirror and ForceField.

As with the projectiles the initial thought was that the wall classes should be able to have unique features, this also was changed later in the development.

Besides the update function, there are some functions to detect whether they collide or not with projectiles or other walls.
All wall data is also retrieved from an XML file and stored in a WallHolder struct.





2.1.4 Animation

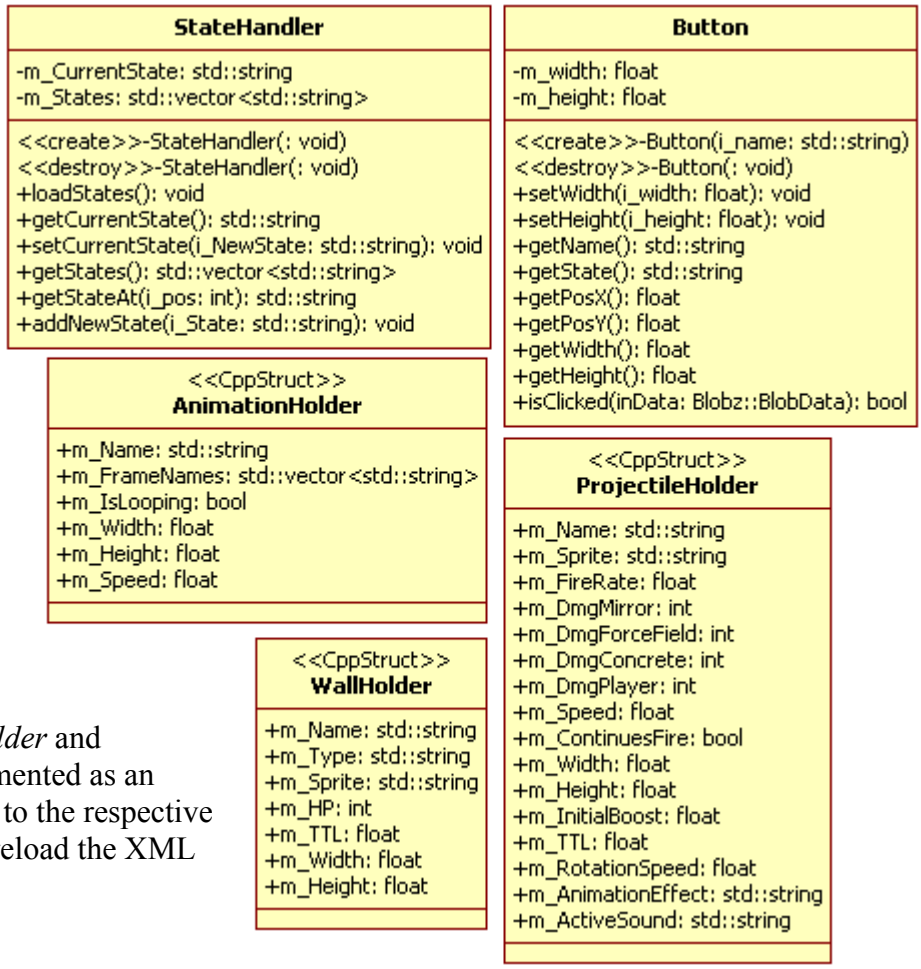
The *Animation* class handles animations. Update updates the current frame if the delay timer has reached zero and sets the isDone flag so that the main class can remove the animation when done.

The animation it self consists of an XML file and arbitrary amount of images need to create the wanted animation effect.

This is loaded once into an AnimationHolder struct and is sent to the constructor from the main class.

2.1.5 Other Classes and Structs

There are a couple of other classes present in the game, like the state class, that handles the games states and the *XMLVarServer* class that Johan Larsson wrote and kindly let us use in this project. This is based on previous work **[5]**.



The *Button* class is a simple class to handle buttons as one of the criteria for the designers' project was to have a menu. This and the fact they also needed to have a splash screen meant that a basic state class was needed.

AnimationHolder, *ProjectileHolder* and *WallHolder* Structs were implemented as an easy way to send the XML data to the respective constructors without having to reload the XML file.

2.1.6 XML

Using XML files to store data that was likely to be altered during the development and test phase of this project was a natural decision as I have used them in earlier projects **[5]** and found them a really use full tool, when testing and balancing games, it also shifts some of the workload to the designers and or testers as no rebuild of the source code is required for the changes to take affect.

```
<animation>
        <frame name="smoke01.png"/>
        <frame name="smoke02.png"/>
        <frame name="smoke03.png"/>
        <frame name="smoke04.png"/>
        <frame name="smoke03.png"/>
        <frame name="smoke02.png"/>
        <looping value="false"/>
        <width value="32.0"/>
        <height value="32.0"/>
        <speed value="0.1"/>
</animation>
```

Structuring a logical and easy to understand XML file takes some careful designing.

However sometimes a last minute addition or other unforeseen changes, can make the file somewhat confusing at first glance.

```
<enemy>
        <weapon name="PlasmaLauncher.xml"/>
        <weapon name="LaserLauncher.xml"/>
        <weapon name="MissileLauncher.xml"/>
        <!-- type = weapon as listed above starting with 0 -->
        <!-- start_rotation right = 0.0, up = 90.0 , left = 180.0 -->
        <!-- rotation_direction left = counter clock wise, right = clock wise, only lower case! -->
        <!-- start_delay_mod = adjustment of first fire, 0.0 = no delay, 0.5 = half normal delay, 1.0 = normal delay -->
        <launcher type="0" start_rotation="0.0"    rotation_direction="left" start_delay_mod="0.3"/>
        <launcher type="0" start_rotation="180.0"  rotation_direction="left" start_delay_mod="0.3"/>
        <launcher type="1" start_rotation="0.0"    rotation_direction="left" start_delay_mod="1.0"/>
        <launcher type="1" start_rotation="90.0"   rotation_direction="left" start_delay_mod="1.5"/>
        <launcher type="2" start_rotation="0.0"    rotation_direction="left" start_delay_mod="2.0"/>
        <launcher type="2" start_rotation="120.0"  rotation_direction="left" start_delay_mod="2.0"/>
        <launcher type="2" start_rotation="-120.0" rotation_direction="left" start_delay_mod="2.0"/>
</enemy>
```

The loading of the XML files was done using two different methods depending on what type of data was going to be obtained in the XML file.

Looking at the settings file we can see that the word “value” occurs a lot in the file this is the data syntax for the first method of data retrieving the word in blue is the “name” of the value and the text in black is the actual value.

A call to load the value from “path_root” would look some thing like:
m_settings->getStringVariable("path_root");

The other method uses the data syntax “name” to specify its value; this is to distinguish between the two methods. If we look at settings again and at “states” we see that there are a couple of states in the file and they all have the same blue “name” this is the states the game can be in, and they are loaded using this code:

```
<settings>
        <!-- PATHS -->
        <path_root value="Data/ProjectPlasma/"/>
        <path_missiles value="Xmls/"/>
        <path_missile_textures value="Weapons/"/>
        <path_enemy value="Xmls/"/>
        <path_player value="Xmls/"/>
        <path_walls value="Xmls/"/>
        <path_wall_textures value="Walls/"/>
        <path_projectilelaunchers value="Xmls/"/>
        <path_sound value="Sound/"/>
        <path_Animation value="Xmls/"/>
        <path_Animation_textures value="Animations/"/>
        <!-- Sounds -->
        <sound_background value="GameMusic3.wav"/>
        <sound_volume value="1.0"/>
        <sound_muted value="false"/>
        <!-- Textures -->
        <logo value="logo.png"/>
        <quit_screen value="quit_screen.png"/>
        <splashscreen value="bg_splash.png"/>
        <startingarea value="fFieldDome.png"/>
        <gameover value ="game_over.png"/>
        <background value="bg_game.png"/>
        <playerbase value="base_player2.png"/>
        <playerboundary value="boundary.png"/>
        <enemybase value="npc_bottom.png"/>
        <enemytop value="npc_top.png"/>
        <texture_sound_volume value="sound_vol.png"/>
        <texture_sound_bar value="sound_bar.png"/>
        <texture_sound_point value="sound_point.png"/>
        <!-- States -->
        <state name="Splash"/>
        <state name="Menu"/>
        <state name="Options"/>
        <state name="Mute"/>
        <state name="Game"/>
        <!-- Player(s) starting positions -->
        <player1 x="0.25" y="1.0" radius="128.0" center="
```

```
void ProjectPlasma::LoadStates()
{
    TiXmlDocument doc("Data/ProjectPlasma/Xmls/Settings.xml");
    doc.LoadFile();
    TiXmlElement* root = doc.FirstChildElement( );
    if ( root != NULL)
    {
        TiXmlElement* tempPtr = root->FirstChildElement();
        while(tempPtr != NULL)
        {
            if(tempPtr->Attribute("name") != NULL)
            {
                m_state->addNewState(tempPtr->Attribute("name"));
            }
            tempPtr = tempPtr->NextSiblingElement();
        }
    }
    else
    {
        std::cout << "Unable to load file: Data/ProjectPlasma/Xmls/Settings.xml" << std::endl;
    }
    m_state->setCurrentState(m_state->getStateAt(0));
    doc.Clear();
}
```

3. Results and future work

We managed to create the game within the given time frame, and kept to the designers design although some changes were done along the way. The biggest change for actual game play is the ability to place walls freely inside the placement zone, originally the walls could only be placed at the edge of the base this meant that bouncing projectiles was almost impossible.

The game is fully playable and up to six players can play at the same time, protecting there base with the three different wall types. The six players limit is stored in Settings.xml and can be changed if needed, how ever there must be a neutral area between every player so that the finger input can be correctly assigned to the right player.

As the bulk of the variables are stored in XML files the designers can continue fine tuning the game to reach optimal balance and enhance the players' experience.

The future work on this game could include adding some of the grate features that dint make it on to this version of the game. One of the features, being the ability for the player to catch and then "throw" the missiles using gestures.

4. Discussion

I'm satisfied with the overall result of this project. Even though we started late and hade some pitfalls to navigate we managed to create a working game that met the requirements of the design document.

Given that we only hade access to the NUI hardware the last week of the project and that all "actual" game testing was done during that timeframe. I'm rather impressed with the improvements we managed to implement during the last days.

Unfortunately that meant that we had to reallocate time from the physics app to address the issues that arose during the testing sessions.

With the benefit of hindsight there could have been a couple of alterations to the design of the classes, as mentioned earlier both walls and projectiles could have been single classes instead of three different types.

Giving the enemy the ability to move around and shoot some what randomly might have been a nice touch.

Implementing the *Holder Structs was a good idea that should have been in the original design. This is a lesson I'm going to keep in mind on any future projects I work on.

Although I doubt that Station Defender ever will hit the shelf's, it was a really good learning experience and the game turn out rather decent considering the time frame we hade to do it.

I would like to see this game at shows/expos like the designers initially hade envisioned.

The games rather straight forward controls and simple rock, paper, scissors game play should be a perfect fit for an expo environment, given that any one can join at any time also fits.

5. References:

[1] Station Defender design document , 2008 - http://svn08.gscept.com/svn/mp8/Documents/ProjectPlasma_DesignDocument.doc

[2] Natural User Interface - http://www.naturalui.com

[3] Box2D - http://www.box2d.org

[4] Mikael Bauer. Application Development for Touch-Screen Interfaces. 2007, LTU Skellefteå - http://epubl.ltu.se/1404-5494/2007/17/LTU-HIP-EX-0717-SE.pdf

[5] Banana Republic Productions, Region Wars 2008 - http://blogs.gscept.com/sp4/

[6] NUI Group - http://www.nuigroup.com