Strategy2D: Turn-based Strategy Video Game Engine for Mobile Devices

Author: **Javier Calvo Villazón**

Supervisor: **Lluís Solano Albajés**

November 2014

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

Over the last decades, video games have grown up to the point of becoming one of the biggest entertainment industries. They have come a long way from being a niche product, targeted mostly at children and presenting themselves as a new kind of toy, to what they represent now; a product all kinds of people can enjoy and be appealed to, regardless of their age and background.

The irruption of smartphones has played a major role in this trend, making them available in platforms with much larger audiences and allowing people to play their games while they travel replacing game focused consoles.

However, the majority of successful titles launched in mobile devices have been targeted at casual players, who do not have much experience with video games, and, in consequence, tend to have simplified mechanics. While this is a nice approach for the development of games for these platforms, it is also true that now that these players have lost their initial fear to video games and joined the gaming community, they might also want to find new types of games that bring new concepts and represent different challenges. This is a great opportunity for providing these newcomers with more traditional video games that can offer a different level of complexity and diversity. And also to appeal those old players that no longer have enough time for playing as they used to.

All of this, of course, taking into account the limitations of mobile devices and their specific features.

In this scenario, we can see that turn-based strategy games, which are perfectly fit to touch controls and do not require immediate response by the player, represent one of the most interesting genres to adapt to mobile devices.

2. OBJECTIVES

The main objective of the project is to design and implement a development framework for the creation of two-dimensional turn-based strategy games for the current generation of mobile devices.
With this objective in mind and looking into the current situation of game engines, it has also been possible to perform an analysis of what are the main requirements the engine must meet.

- Make sure that it provides the required features for the development of games that follow the typical structure in the turn-based strategy genre: A two-dimensional grid map in which units and buildings pertaining to different factions interact with one another in order to accomplish an objective.

- Given the previous structure, make it highly configurable; providing developers with the needed tools to innovate by adding new features, gameplay mechanics, visuals and new content: simplifying the process and ensuring that the development workload is reduced, especially in graphics, control and management handling.

- Be multi-platform so that the games created with it can reach as many users. It has been established that the engine must be at least fully compatible with the two dominant mobile platforms: Android and iOS smartphones and tablets.

- Acquiring knowledge of game engine design and development techniques to ensure the accomplishment of the previous objectives.

3. CONTEXT

It is important to define the roles of the actors that will take part in the project's life cycle, as to provide an explanation that avoids confusions and misunderstandings before entering the following chapters.

There will be two main types of actors involved:

3.1 Users

They are the game developers that will use the engine to create their strategy games and the main users of the project.
As it will be explained later, the development framework chosen is Cocos2d-x, based on the C++ programming language. For this reason, these users must have certain knowledge of this language. Having knowledge on Cocos2d-x would also be helpful, especially for handling sprites, animations and reproducing sound effects, although the framework is very accessible and will not be an obstacle for newcomers.

3.2 Players

They are the people who will play the games developed with the engine and the final target of the game developers. They will also be final users of the engine, but since they will not directly interact with most of its layers and to differentiate them from the game developers I will refer to them as players.

3.3 Game Engines

Game Engines are software development frameworks specifically designed for the development of video games. They were born to free developers from having to repeat parts of the creation process several times and to offer a stable, adaptable and robust tool that can give an abstraction from some of the most platform dependent procedures and core mechanics.

These engines are mostly built by delivering an interface that provides the developers with a data-driven method of development, which does not require the typical algorithmic approach and which, if well structured, can provide a much easier usage without giving up a huge amount of possibilities for the development.

Some engines are designed only to offer a very specialized tool that handles one specific part of the games that will be developed. That is the case of some of the most popular engines, which normally handle Graphics by controlling the rendering phase or Physics, managing the way different objects interact with each other.

Others, however, are designed with a much more specific purpose, offering a solid yet adaptable tool for the development of games that will share most of their core mechanics.

This last will be the case of Strategy2d, which will offer a development tool for the creation of games pertaining to the same genre.

3.3.1 The Game Engine Scene

With this purpose in mind, it was very important to take a look at the scene to get ideas on how game engines are designed and how they offer their users the tools needed for the creation process.

In the past, only companies with big budgets were able to produce their games and release them with a publisher. For this reason, engines were usually bought to other companies that were specialized in its development; such was the case of some of the most important engines of that time:

Figure 1: Quake Logo

The Quake Engine, developed by *Id Software* was one of the first game engines to be used by companies other than their creator. It was originally conceived for the development of the video game *Quake*, but was later used by other developers such as Valve for the creation of *Half-Life*.

Figure 2: Unreal Engine Logo

The Unreal Engine, developed by *Epic Games,* was originally created for the development of the video game *Unreal*. Through its first third iterations (the fourth one has been recently released and is still in beta phase) it has become the most used video game engine in history, featuring in more than a hundred games.

Another option was the development of internal engines designed specifically for the development of games within the same company. Some of these were developed for a single game to help through its development or to offer a tool for the creation of similar instances or games pertaining to that same genre.

A very early example for this last option would be *SCUMM* (Script Creation Utility for Maniac Mansion), which offered a development framework for Graphic Adventures and was originally conceived for the sole development of *Maniac Mansion*, although it was later used for all of the Graphic Adventures created by *LucasArts* (such as *The Secret of Monkey Island, Day Of The Tentacle, Sam and Max or Indiana Jones and the Fate of Atlantis)*.

Figure 3: Maniac Mansion, the first video game to use SCUMM

A different example can be found in the *Infinity Engine*, created by *Bioware* for the development of the Role-Playing Game *Baldur's Gate*, which was later used by other Isometric RPGs such as *Planescape: Torment, Icewind Dale* or *Baldur's Gate II.*

Figure 4: Planescape Torment, one of the Role Playing Games developed using the Infinity Engine

Finally, a very interesting example of genre-targeted video game engine can be found in *RPG Maker*, a series of programs designed for the development of Role Playing Games that were very popular between non-professional developers, empowering them to design and publish their own games. Some examples are Eternal Eden or To the Moon, which won several awards and became commercial successes.

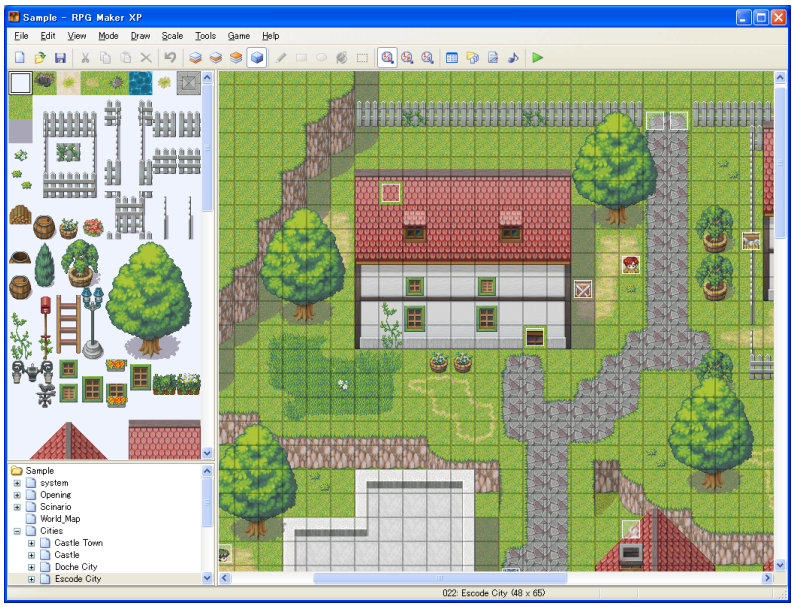

Figure 5: RPG Maker

3.3.2 Game Engines Today

With the boom of indie game development, a new model for open game engines has become widely popular. Some of them are plainly open-source and are sustained thanks to the donations of their users while others have updated their business models to adapt to this new market of indie developers.

Also, due to the arrival of the wide variety of platforms for mobile devices, it has become very important for these engines to be cross-platform and fully compatible with their characteristics.

The main example of this is *Unity 3D*, which offers a built-in interface for the development as well as a scalable and portable framework based on the purchase of extensions and subscriptions.

Figure 6: Unity Logo

Another example is the recently released *Unreal Engine 4*, which has adapted its business model from previous iterations to a subscription plan similar to the one offered by *Unity 3D*, although offering direct access to its subscribers to the source code and allowing them to make their own plugins and extensions for selling them to other subscribers.

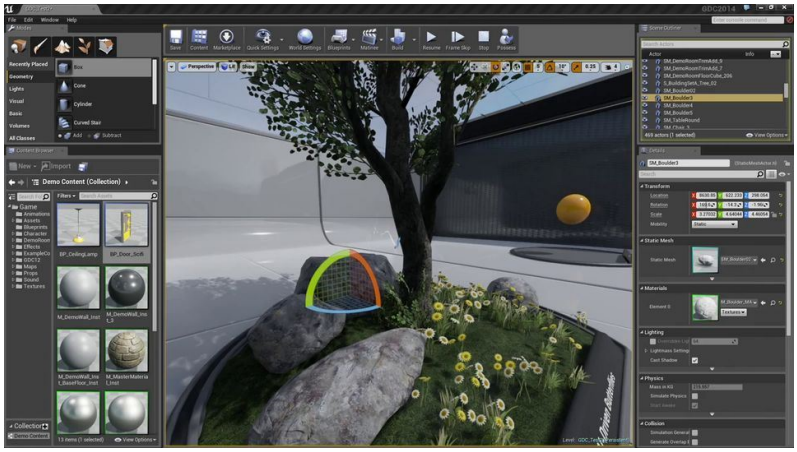

Figure 7: The Unreal Engine 4 editor

The CryEngine, developed by Crytek, has also proved to be one of the most technologically advanced game engines in the market, counting with some of the most graphically powerful games of all time, such as the Crysis series, Ryse: Son of Rome, Star Citizen or Evolve.

Figure 8: Crysis 3, using the CryEngine 3

Corona SDK, by Corona Labs, also offers a subscription model for a very portable engine that supports development for Android, Windows Phone and iOS devices using the Lua scripting language.

Figure 9: Corona SDK Logo

A final example is *Cocos2d-x*, the one chosen for the development of the project, which will be explained in detail in the Framework section of this document.

3.4 Turn-based Strategy Video Games

Turn-based strategy games (often called TBS Games) are strategy games where players take actions in turns when playing, so that not all of them can play simultaneously and they can take more time to plan their next move.
Their main inspiration lies in classic board games such as chess, although with the flow of time they have also taken elements from more modern games such as Risk, Diplomacy, Gettysburg or Warhammer, and also from Role Playing Games such as Dungeons and Dragons.

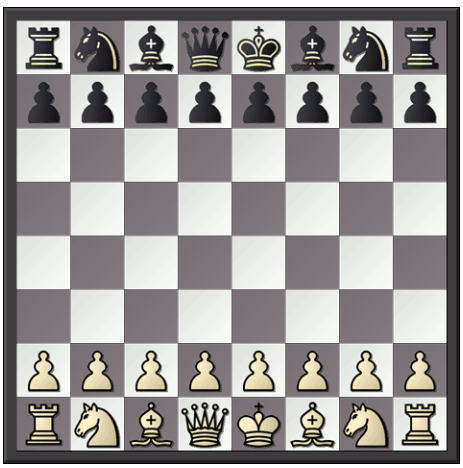

Figure 10: Chess Board

There are many variations among turn-based strategy games; some of them are more centered in battle and do not focus on other aspects of gameplay, while others offer a much wider approach.

All of them have served as reference in order to design an engine that offers enough tools to develop games with all these mechanics.

3.4.1 Turn-based Tactics

When starting the project, the main focus was to be able to reproduce battle focused games, and for that reason I looked into the gameplay mechanics of this subgenre. In it, the player is able to select units that pertain to her team and deploy them through the map in order to attack and destroy the enemy's units. One of the most important games in this genre and the one that served as main inspiration was Advance Wars, by Intelligent Systems (Nintendo).

Figure 11: Advance Wars, one of the main exponents of TBS Games

In it, every unit counts with a limited range of movement each turn and can only perform one skill. Buildings are used to produce new troops, consuming resources for it, making them essential to recruit new units.

3.4.2 Tactical Role Playing Games

Among the battle centered games, there are some that present Role Playing Mechanics in which units are able to grow their power and learn new skills and abilities. For this reason the engine was opened, making it more configurable so that these characteristics could also be developed.

This subgenre, with the name of Tactical Role-playing Games, was born as a mixture of both RPGs and Turn-based Strategy Games in the early 80s, with the release of Ultima III.

As it happens for Role Playing Games, there are two main flows in terms of design, Japanese RPGs and Western RPGs.

Japanese games usually have a great separation between battles and main game flow, which runs in parallel and has completely different mechanics.
One of the main series in the subgenre is also developed by Intelligent Systems: Fire Emblem.

Figure 12: FIre Emblem: Path of Radiance

In it, units have unique personality and stand by the player through the whole adventure, unless they die in battle.

Other very successful examples of the subgenre are Final Fantasy Tactics or Tactics Ogre.

Figure 13: Tactics Ogre

In the case of Western RPGs, however, there is not so much separation between the battle phase and the game itself. It is the case of games like the original Fallout or Arcanum, in which their turn-based battles are performed in the same scenarios where the rest of the gameplay takes place.

The most recent example is Wasteland 2, sequel to the 1988 classic.

Figure 14: Wasteland 2

3.4.3 World Strategy

This last subgenre is the one composed by those games in which battle is not necessarily the main focus, and strategy can lie in different aspects of the gameplay, such as the economy or the relationship with other factions.
One recent example would be the last iteration on the X-COM series, in which the player takes control of anti-alien agency that has to protect the different countries of the world from an alien invasion while handling the budget and taking decisions in order to save as many people as possible.

Figure 15: X-COM: Enemy Within

Perhaps the best example for the subgenre would be the Civilization series, in which players take the role of the leader of a nation and must guide it from its foundation through history managing the gathering of resources, the construction of cities and buildings, the investigation of new technologies, the relationships with the rest of the factions in the game and military conflicts.

Figure 16: Civilization V

The conclusion of the analysis of this last subgenre was that it would not suppose too much an effort to include additional tools for controlling the economy of the factions in the game, so tools for handling these aspects were added.

4. THE DEVELOPMENT FRAMEWORK

4.1 Framework Choice

Since one of the project's main objectives was to offer a highly portable and scalable engine, we needed a multiplatform framework that allowed developers to design their games for different mobile devices.

With this in mind, I searched for graphic engines that could take care of the graphic elements and that allowed the development for multiple platforms, mainly iOS and Android.

At the time this research was conducted Unreal Engine 4 had not been released yet, so after gathering information I ended up with three main engines that fulfilled the objectives: Unity3D, Corona SDK and Cocos2d-x, all of them exportable to iOS and Android.

While all of them represented good options, I decided that in order to provide users with as many configurable possibilities, it was a very important requirement to have full control over how the framework handled the different events and the possibility of adding layers on top of it. In this regard, both Unity3D and Corona SDK have closed environments, with many of their features being totally opaque to the user and providing only an interface based on scripting languages (C# and Javascript for Unity3D and Lua for Corona).

Cocos2d-x, in contraposition, is an open-source project and, as such, includes all of its source code so that users can see how it works internally and gives them the possibility of modifying it. Also, since it is based on C++, it offers a much wider control over the flow of the program, which translates into more possibilities to personalize it and more efficiency, having no scripting but pure native code.

Another criterion was the payment: While Cocos2d-x is totally free and open to development, all the others have subscription models in which users must pay in order to have access to all of their features. So in order to offer a more accessible engine that required no subscriptions of any kind for any of its components, Cocos2d-x was the most appropriate framework.

Finally, a major factor for the decision was the graphics focus. Unity3D and Unreal Engine are both centered on 3D graphics and, for this reason, users must use workarounds in order to configure pure 2D games; Cocos2d-x, instead, is totally centered on two-dimensional graphics and all of its interface is prepared for treating this type of games.
For these reasons I ended up with the decision of using Cocos2d-x for the development of the project.

4.2 Cocos2d-x

Cocos2d-X is an open source game development framework written in C++ that can be used to build from games to apps and any type of cross platform programs that need a GUI (Graphical User Interface).

Figure 17: Cocos2d-x Logo

It allows multi-platform development and represents the most important development framework for 2D video games, with more than 400,000 developers using it.

It has a lot of components that allow the creation of all types of video games, such as physics or storage libraries, but our focus will be on describing the most

important features required for turn based strategy games. But first should give an overview of its general structure:

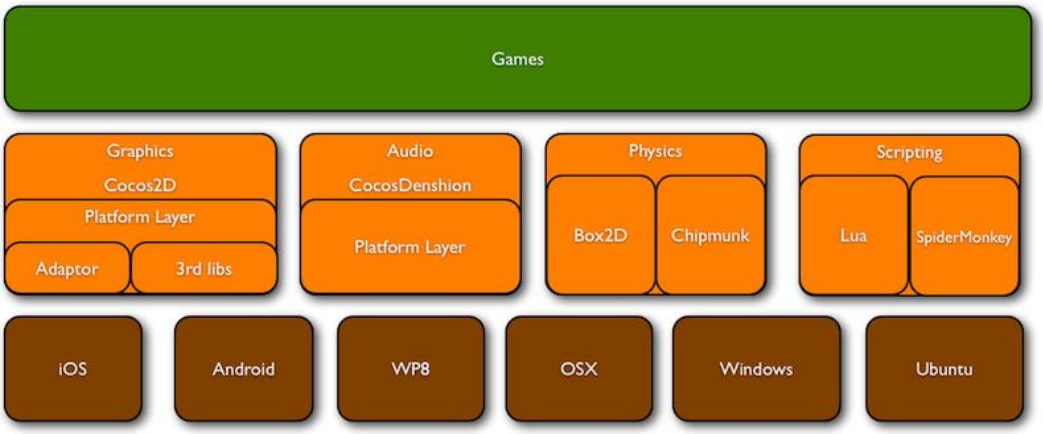

Figure 18: Cocos2d-x Layout Overview

4.2.1 Audio

Audio and Sound Effects represent a very important part of any kind of video game, and so is the case for turn based strategy games.

The approach followed by Cocos2d-x is simple and intuitive: it contains a direct wrapper for the *OpenAL* audio library, which can reproduce all kinds of sound effects with multiple channels and three-dimensional positional audio.

Figure 19: OpenAL Logo

This wrapper receives the name of *CocosDenshion*, and it provides with a singleton class that can be commanded to reproduce a sound effect at any time during the game flow by just passing the name of the audio file. Of course, these sound effects can be played just once or in a loop.

A code example for playing a sound effect would be the following:

```
CocosDenshion::SimpleAudioEngine::getInstance()->playEffect("sound.wav");
```

In which "sound.wav" is the file containing the audio file. The fact that different platforms cannot reproduce the same audio formats must be considered, so users must use a fully compatible format (such as *wav*) or set a different file for each target device.

4.2.2 User Input

The way the user input is handled is the most basic part of any video game, since by definition a video game requires the interaction of the player. In the particular case of this project, targeted at mobile devices, this interaction must be performed through the touch controls.
For this purpose, Cocos2d-x offers the possibility of defining callbacks for the following actions of the user:

- Touches Started
- Touches Moved
- Touches Ended

To do so, it allows you to wrap between the hardware detection of the user interaction and your defined callbacks by providing a *Director singleton* class which contains an *Event Dispatcher* that can add listeners for them and to which you can link your functions. These listeners have a priority parameter, which tells Cocos2d-X what is the order in which the callback functions should be called.
Each one of these wrappers pass to your callback an array of the touches made by the user on the device screen that correspond to the action of the function's name, which provide information about the current location of the touches and, for touches that had already been started, also about its previous locations.

These callbacks can be defined in the context of a Node class, which will be explained later, and implemented to affect the state of the graphic and domain elements of the game.

A code example for the definition of these callbacks would be the following:

```
MyNode* node = MyNode::create();

cocos2d::EventListenerTouchAllAtOnce* listener = cocos2d::EventListenerTouchAllAtOnce::
                                                        create();

listener->onTouchesBegan = CC_CALLBACK_2(MyNode::onTouchesBegan, node);
listener->onTouchesBegan = CC_CALLBACK_2(MyNode::onTouchesEnded, node);
listener->onTouchesMoved = CC_CALLBACK_2(MyNode::onTouchesMoved, node);

cocos2d::EventDispatcher* ed = cocos2d::Director::getInstance()->getEventDispatcher();
ed->addEventListenerWithSceneGraphPriority(listener, node);
```

In which the MyNode class would have the following defined functions:

```
void onTouchesBegan(std::vector<cocos2d::Touch*> touches, cocos2d::Event* event);
void onTouchesEnded(std::vector<cocos2d::Touch*> touches, cocos2d::Event* event);
void onTouchesMoved(std::vector<cocos2d::Touch*> touches, cocos2d::Event* event);
```

4.2.3 Graphics

Cocos2d-x offers an abstraction layer over OpenGL ES (Embedded Systems) while, at the same time, it gives access to some of their features directly if it is needed. It has also support for compiling and linking GLSL shaders with the different renderable objects in your scene.

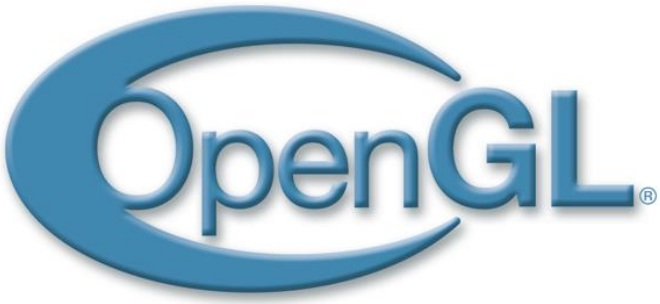

Figure 20: OpenGL Logo

Through this abstraction, it provides a very simple and adaptable way for configuring the scene that appears on the screen, which is mainly based on the coordinates system, the creation of scenes and the defined Node class and its variants.

4.2.3.1 Coordinate System

Cocos2d-x provides a representation of the screen coordinates taking into account the device's orientation (landscape or portrait). It internally handles the device's both horizontal and vertical sizes, which can be consulted at any time via the Director class, as in the example:

```
cocos2d::Size size = cocos2d::Director::getInstance()->getVisibleSize();
```

The Size class contains a width and a height value that allows you to abstract from the device's orientation and adapt your graphics to the screen dimensions in order to support multiple resolutions.

It must be taken into account that in this coordinate system, the origin (0, 0) represents the bottom left corner of the device and so the X-axis grows to the right and Y grows up.

Figure 21: Cocos2d-x Coordinate System

It also supports negative values for the positions of the elements on screen, which allows to easily implement scroll. These positions are represented with the Point class, which simply has an X and Y coordinates.

4.2.3.2 Scenes

Scenes are the representation of independent executions within the framework. Only one Scene can be running at a time, and it is there where all Nodes are stored. They are managed through the Director class, which has an internal stack of Scenes. When the push function is called, the running Scene is stored in memory and paused, while the new one takes control. Later, when the pop function is called, the Scene that is now on top of the stack resumes its execution exactly from the point where it left.

This structure allows the creation of multiple screens or stages and even the addition of multiple games within the same application.

4.2.3.3 The Cocos2d-x Node Class

A scene in Cocos2d-x is composed by Nodes, which are the base graphic representation for any element. Each of them can have other nodes as children identified with a unique identifier, which allows the creation of complex scenes by implementing structured hierarchies.
This way, the transformations applied to the parent Node affect all of its children proportionally in a recursive way.
It is especially useful in the particular case of this project for implementing the scroll and the zoom features by setting a parent Node that contains all the elements on the board.
Also, the positions of all the children are relative to their parent Node, so a (0,0) position inside another Node refers to the center of such Node, which makes it possible to maintain coherence between them.

Attributes

The most important attributes of the class, in addition to the unique identifier, are:

- Position: The X and Y coordinates in which it is located (for Nodes with a parent these are relative to its parent, and for others directly to the device).

- Size: The original size of the Node in terms of width and height.

- Scale: two values that refer to the proportional size of the Node in both width and height respect its original size.

- Bounding Box: The Bounding Box that contains the Sprite with its current dimensions (taking into account the original size and the scale factor applied).

- Rotation: The angle in degrees to which the Node is rotated (clockwise).

- Visible: a Boolean value that tells whether the Node will be drawn into the scene or not.

- Z Order: The depth of the Node, when two or more Nodes share the same space in the screen, one must be drawn on top of the other. This attribute gives a three-dimensional component to Nodes and determines the order in which they are rendered. It is also used for computing the final color when transparencies (Alpha Values) are used, determining the order for mixing the colors in the right way and order in the *Alpha Blending* process.

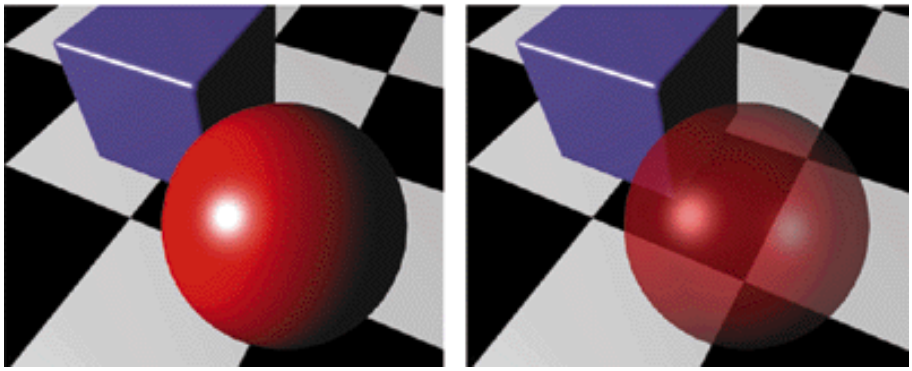

Figure 22: Alpha Blending enables the presence of semi-transparent objects on-screen

Actions

All nodes can run actions, which are then handled by an ActionManager class that executes them according to the parameters set.
These actions refer to transformations and other special features that need to be performed with precise timing and in a certain order. Some of them are performed instantly while others require a certain interval of time. They can also be set to be finite and finish after one execution or to be performed indefinably until they are removed or the game exits. They have a unique identifier that helps checking if they have finished or change some of their paremeters.
These are some interesting examples of the many possible actions:

- MoveTo: which moves the Node into a certain position in a given time, interpolating the movement between frames at the required speed.

- ScaleTo: which scales the Node up to a certain scale proportionally according to the set time.

- RotateTo: which rotates the Node until a certain degree interpolating the rotation to the given time.

- Flip: which flips the Node in the set specified axis.

- RemoveSelf: which removes the Node at the given time.

And this is the diagram for all the Actions within the framework:

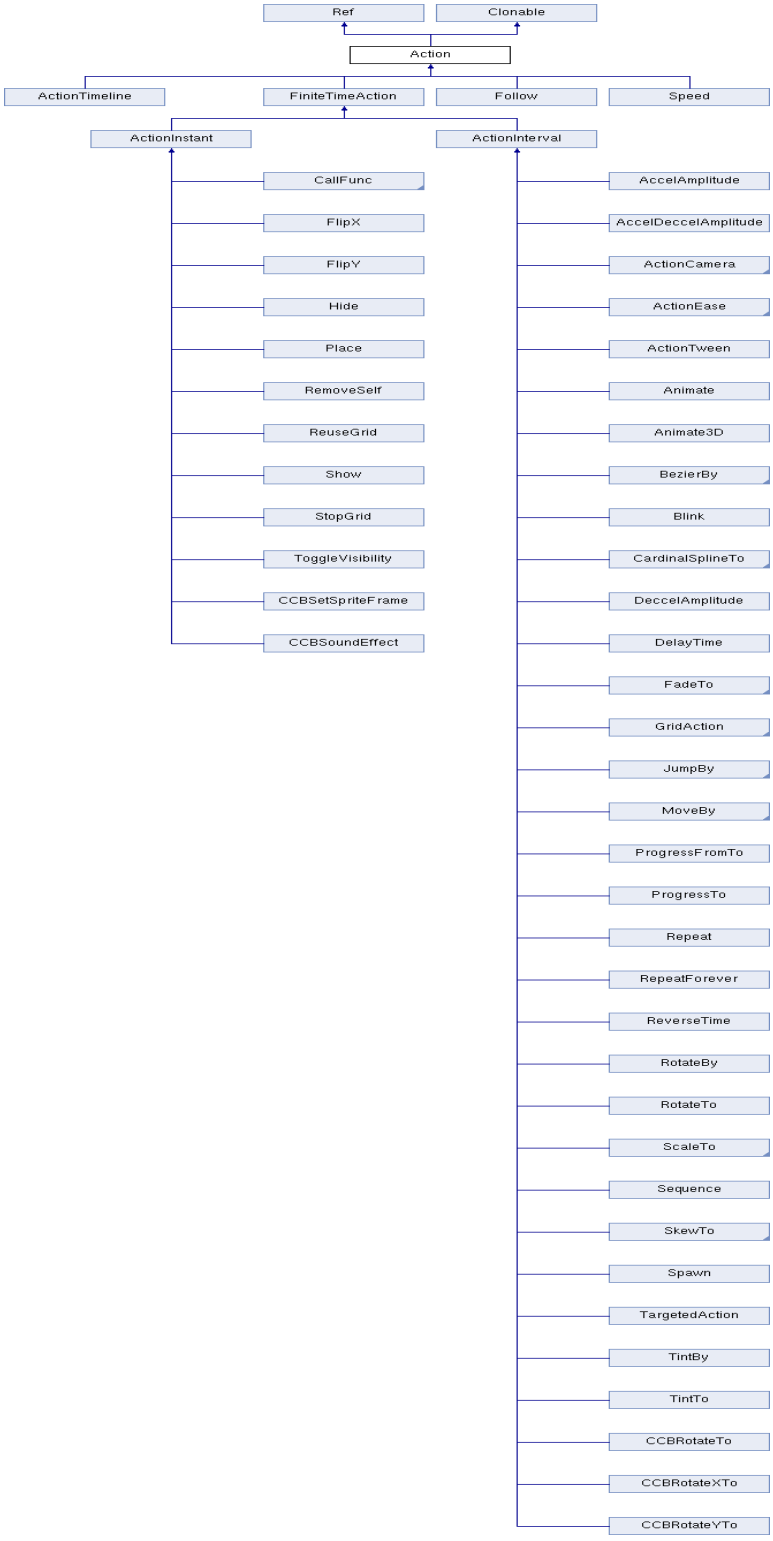

Figure 23: Cocos2d-x Action Class Diagram

All of them can be added to Sequences that will perform them in order and won't let one be executed until the previous one has finished. Here is a code example for this:

```
cocos2d::Sequence* seq = cocos2d::Sequence::create( MoveTo::create(duration, position),
                                                    ScaleTo::create(duration_scale, scale),
                                                    RemoveSelf::create(),
                                                    NULL );
seq->setId(1);

node->runAction(seq);
```

In which we create a Sequence that will first move the Node to the position in a certain time set by the duration parameter, then after reaching that position scale it with another duration and finally remove it.

4.2.3.4 Node Subclasses

The Node class provides a base for other drawable objects that have more specific ways to be rendered. There are several of them, from GUI elements to direct game elements, but these are the most important and useful ones in the context of the project:

Layer

It allows to directly set the color of the Node with RGBA (Red, Green, Blue, Alpha) values. Also, they have another extension with the name of **LayerGradient** that allows the creation of Layers in which the color of its pixels vary according to their position by interpolating the given colors for the edges.
In the project, layers are used to contain the map elements and also to display with a special color the reachable positions for a movement.

Sprite

It is used to assign a two-dimensional image or an animation composed by several images to the Node. This way it is possible to personalize the elements that appear on the scene as one wants by providing assets.
The loader supports all standard formats for the images and automatically detects them, without requiring a special action for each one of them.
In the project, sprites are used to represent all the game elements that form the map.
An example for implementing the read operation of a Sprite would be the following:

```
cocos2d::Sprite* sprite = cocos2d::Sprite::create(“sprite.png”);
sprite->setId(1);
sprite->setPosition(10, 10);
myLayer->addChild(sprite);
```

In which we add a sprite with the *“sprite.png”* image to the layer *myLayer* into the position [10,10], having the number 1 as identifier, which will help us later to easily get it.

Button

They are the Cocos2d-x implementation of UI buttons. They can be assigned a callback function for any time they are touched by the player. In this callback functions they provide information about the type of touch, which can be a touch that begins, a touch that ends or a touch that has moved.

4.2.4 Setting up the environment

For setting the development framework, the first step is to download Cocos2d-x from its website: http://www.cocos2d-x.org and select to download the C++ version (there is a different one set up for JavaScript and html-5).
Once downloaded, it just requires installing the recently added cocos console, a terminal application that will allow to create, compile and run the Cocos2d-x projects.
For this, the next step is to move through the computer file system with the terminal up to the *cocos2d-x* folder and then run *setup.py*, which will install it in the system. Once installed, to create a new project the following command line must be entered:

```
cocos new nameProject –p com.nameCompany.nameProject –l cpp –d pathProject
```

In which *nameProject* is the name of the project about to be created, *nameCompany* the name of the organization, *pathProject* the location of the folder in which the project will be generated and *com.nameCompany.nameProject* the package name for Android, in case the project is built for this platform.
If everything goes well you will the following message will be displayed:

```
Running command: new
> Copy template into pathProject
> Copying cocos2d-x files...
> Rename project from 'HelloCpp' to nameProject
> Replace the project name from 'HelloCpp' to 'MyGame'
> Replace the project package name from 'org.cocos2dx.hellocpp' to
'com.nameCompany.nameProject'
```

Afterwards, in the path set a project with the following structure will be generated:

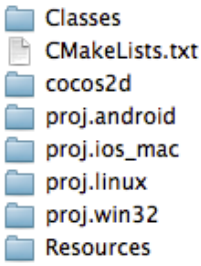

Figure 24: Cocos2d-x Project Folder Layout

Where the Classes folder will contain the developer's source files, Resources the materials needed by the project, cocos2d all the source files needed by cocos2d itself and the proj folders the necessary files to generate the project for each platform.

Note that being open source, the cocos2d source files can be modified if needed and allow the developer to see exactly how the framework works, but they constitute the source code and altering them has some risk.

- proj.win32: a *vcxproj* and the rest of configuration files for a visual studio project that already contains the cocos2d source files and the classes inside the Classes folder.

- proj.linux: a main.cpp file for starting an application in Linux Operating Systems.

- proj.ios_mac: An Xcode project already configured. To deploy your projects into an ios device you need to do so through the xcode program.

- proj.android: The necessary files to export the project to an Android device. For doing the following environment variables setup is needed:

    - NDK_TOOLCHAIN_VERSION: Version of NDK
    - NDK_ROOT: Path to developer's android-ndk folder
    - ANDROID_SDK_ROOT: Path to Android SDK

- ANT_ROOT: The path to the Apache Ant application, for generating the *apk* file.

To compile the code the developer needs to configure the *AndroidManifest.xml*, *Android.mk* and *Application.mk* just as it would be done with a normal Android project and then run *build_native.py*. Once it has been successfully compiled, must run *ant* with the *debug* or *release* option to generate the *apk* file that will be installed in the device.

5. THE ENGINE

For the design of the engine, I have followed the Model View Controller Architecture; creating two differentiated layers that control the different elements and interact with each other.
However, since some of the elements of the design are also accessible by the game developers and require their configuration, I have also divided the structure of the project into logic layers that have a better representation of their role and that make using the engine easier to understand.

5.1 Engine Structure Overview

- Strategy2D: An instance of the engine, it defines the configurations for a game and contains all the other elements of the structure.

- View: It manages the graphic elements and the user interactions with the device. The main class for this layer is *LayerMap*, a Cocos2d-x *Layer* subclass.

- Domain: It manages the state of the game and its different elements. The main classes of this layer are *ActionManagerStrategy2D*, which handles the user's actions and provides an interface between the *View* and the *Domain*; and *ControllerGame*, which handles all the configurable elements of the system. However, the core of its implementation lies in the callbacks: User-defined functions that are called when special events take place during the game. These callbacks allow users to take any actions and affect the game elements in any way they want in these key moments of the gameplay.

5.2 The Strategy2D Module

It is the main class of the system and handles the connection between the different layers of the engine while providing users with the necessary interfaces to set up and configure the execution.

It contains an instance of *LayerMap*, *ActionManagerStrategy2D* and *ControllerGame* and is constructed with the number of rows and columns of the map.

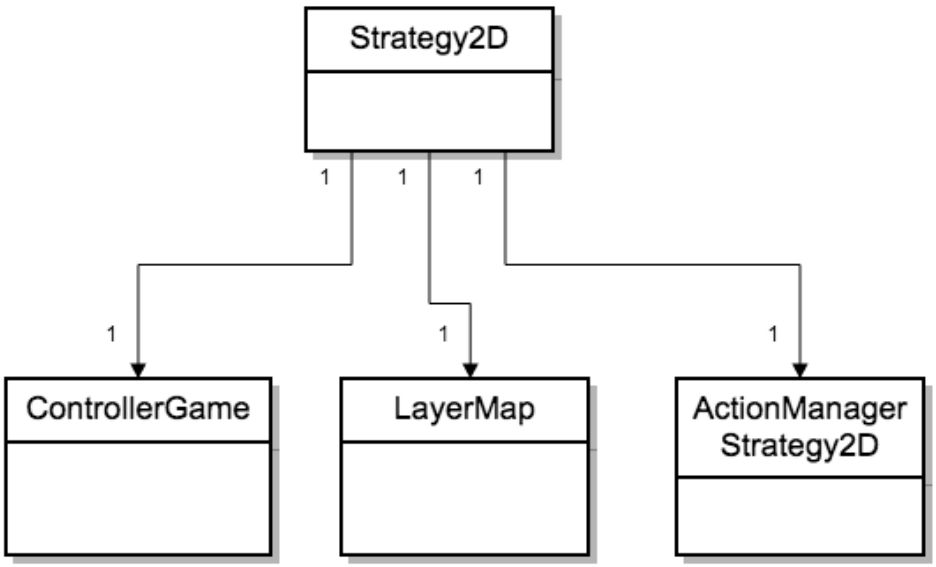

Figure 25: Strategy2D class diagram

Each *Strategy2D* instance refers to a different configuration of the engine which runs independently, although can be easily combined with other executions (either other *Strategy2D* executions or any other kind of game running under the Cocos2d-x framework).

These different executions are handled using the Cocos2d-x *Director* class, which, as explained, provides with a Scene handler that can move through different contexts and establish where the current execution lies.

5.2.1 Usage

There are two types of usage for the *Strategy2D* class, depending on the execution state. On the one hand, users need to create *Strategy2D* instances to specify their general parameters and configure them as they please and then start their execution by running such instances. On the other hand, it is also used in the callbacks that they define and that run during the execution time. This way they are provided with an interface that allows them to get information and apply new configurations.

5.2.2 Implementation

In order to provide these possibilities, however, it was needed to implement the class with some specific features.

In order to allow accessing the class through any point of the execution and especially in the user-defined callbacks, users needed a static method that provided them with the running instance of the module. However, since I wanted to allow the execution and configuration of multiple *Strategy2D* instances simultaneously, I could not just define the class as singleton in order to make it accessible from any point.

So in order to implement it, I created a static stack of *Strategy2D* references that stored the executing instances and provided a static function called *getInstance()* to get the running one, which is at the top of the stack.

For doing this, whenever an instance is started by calling the *run()* method, the stack automatically adds the reference to this instance on the top its stack.

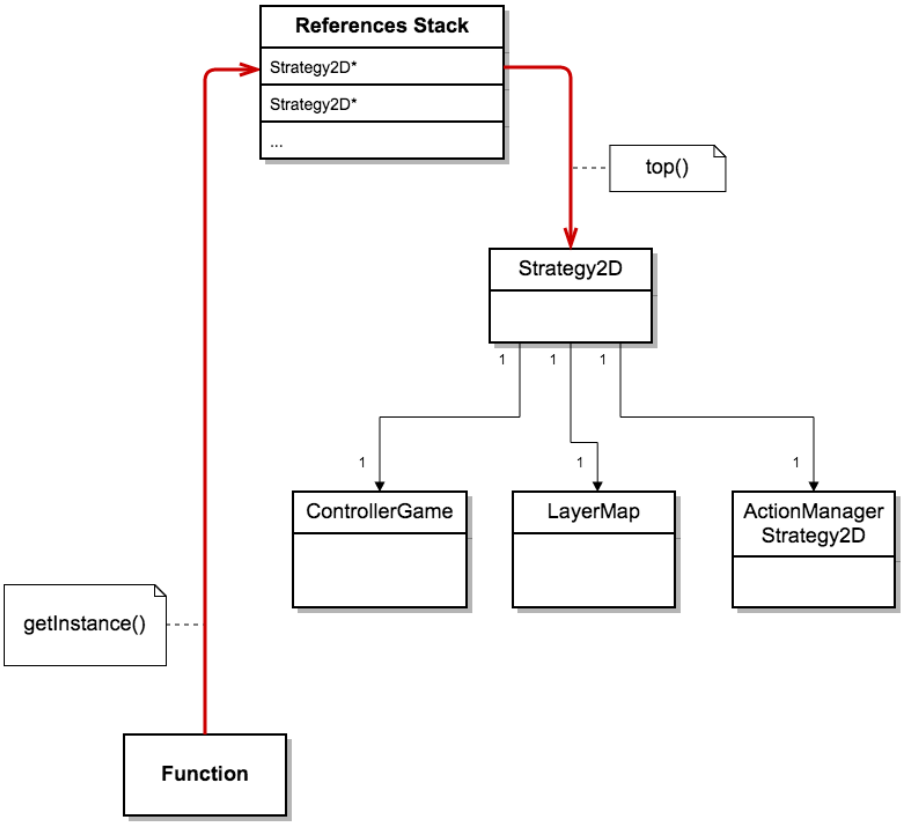

Figure 26: The system can get the currently running Strategy2D instance from any part of the code

Each instance of Strategy2D runs a Cocos2d-x *Scene* as well, which later controls the execution of the game.

Therefore, for running an instance of *Strategy2D*, the *run* function first pushes the current instance into the stack and then calls the *Director's pushScene* method with the instance's *Scene*, this way Cocos2d-x takes control over it and makes the context switch. Also, since a single instance cannot be run more than once at the same time, so I also do a control with the Boolean variable *running*, which tells whether a Strategy2D is running or not.

This is the code for the *run* function, which of course is not static and needs an instance to be called:

```
bool Strategy2D::run()
{
    bool success = false;
    if (!running)
    {
        strategy2D_instances.push(this);
        controller_game->initGame();
        scene = cocos2d::Scene::create();
        scene->addChild(layer_map);
        cocos2d::Director::getInstance()->pushScene(scene);
        success = true;
    }
    return success;
}
```

It is not necessary to delete the Scene instance, since the Cocos2d-x Director takes its ownership removes it from memory when the *popScene* method is called. This is the code for the *stop* method, which stops the execution of the currently running instance (if it is a Strategy2D Scene):

```
void Strategy2D::stop()
{
    if (!strategy2D_instances.empty() && Director::getInstance()->getRunningScene() ==
           strategy2D_instances.top()->getScene())
    {
        strategy2D_instances.pop();
        Director::getInstance()->popScene();
    }
}
```

Finally, when the Director changes the running *Scene* and switches context, the local variables of the previous context are destroyed and lost, so they cannot be referenced from this new context; this causes the stack references to be Dangling Pointers:

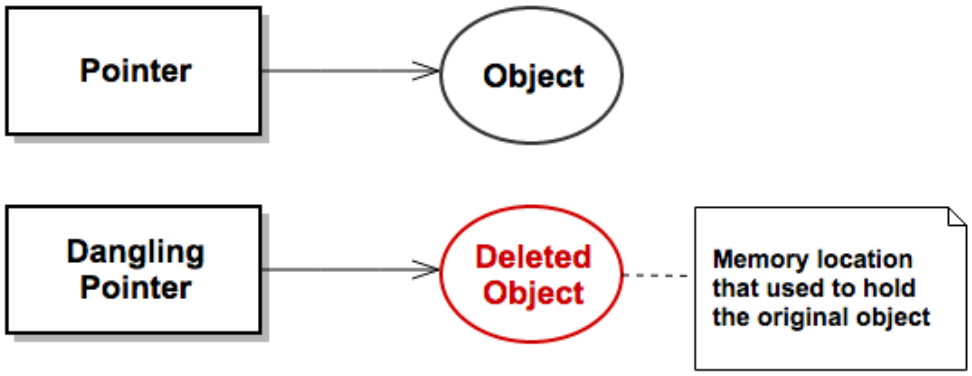

Figure 27: Dangling Pointer concept

For this reason, the *Strategy2D* instances that are stored in the stack need to be accessible from any context.

To prevent this, the system must not allow the normal declaration of *Strategy2D* variables and only allow their allocation in a context free environment. This is done by making the constructor of the class a private method that cannot be accessed from the outside and, instead, providing a static instance generator that returns a pointer to the new instance.

```
private:
    Strategy2D(const long &nrows, const long &ncolumns);
    ...

Public:
    static Strategy2D* create(const long &nrows, const long &ncolumns);
    ...
```

And the implementation of the *create* method:

```
Strategy2D* Strategy2D::create(const long &nrows, const long &ncolumns)
{
    return new Strategy2D(nrows, ncolumns);
}
```

5.3 View Layer

This is the layer that controls the graphics and handles the user-input. It has been designed according to the Cocos2d-x framework, using the tools provided by the graphics engine and its user-input libraries. In addition, I have tried to provide an adaptable environment that relies on Cocos2d-x features and that can be totally configured and expanded by the user, as it has been explained in the Development Framework section.

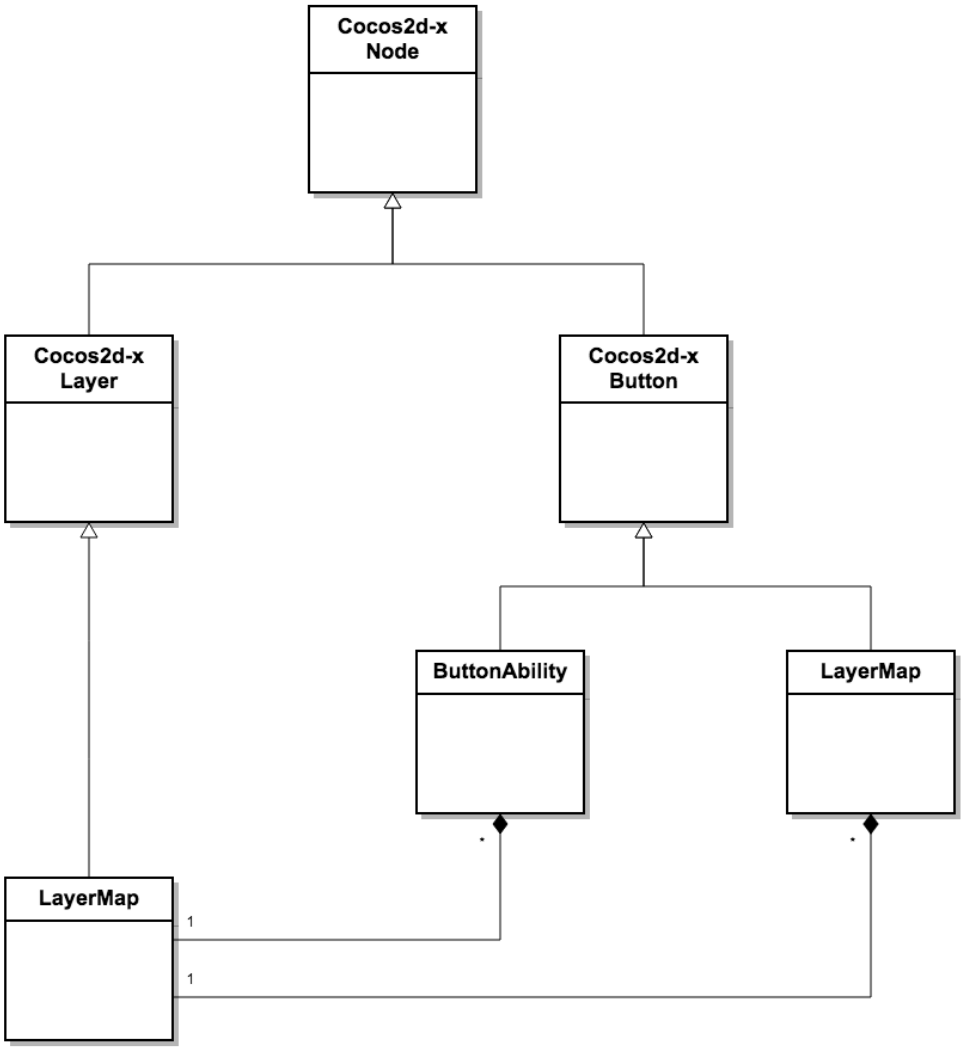

Figure 28: View Layer class diagram

5.3.1 LayerMap

This class is the representation of the game map and all of its specific features and functions. It is a Cocos2d-x *Layer* subclass and internally contains the Cocos2d-x representations for all the game elements as children.
It handles the creation, movement and destruction of all the graphic representations of the game and the user's interactions, making the required transformations for the scroll and zoom of the map and passing its actions to the *ActionManagerStrategy2D*, which processes them and determines how these affect the domain layer.

It is composed by:

- Background Layer:

It contains the background *Sprite* that is rendered when there is nothing to be displayed on a part of the screen, either because there are no graphical elements or because they have a transparent component. The *Sprite* is set by the user and is scaled to exactly fit on the screen of the device.

- Map Layer:

It contains the Game Elements that are part of the game board, including *Units*, *Buildings*, *Terrains*, UI elements such as the reachable positions indicators and any elements that the user wants to add to the game map. The scrolling and zooming transformations are applied to this layer.

- Other Elements:

The buttons used to display the selectable abilities or units to be recruited (which I will further explain in the following section) must be shown on top of the map and cannot be affected by the scroll or zoom, since their position and size must remain invariable. For this reason, they are added to *LayerMap* as independent *Node* children.
Also, users have access to the *addChild* function, which will allow them to add any element, not only as children of *LayerMap* but also for any of the Sub Layers, in case they want the same transformations to be applied on them.

Attributes

In addition to the Sub Layers, it also contains some configurable parameters that are used to handle the graphics or to provide the user with important information.

- Num Columns: The number of columns of the map.

- Num Rows: The number of rows of the map.

- Zoom Rows: The current zoom of the map in terms of the number of visible rows on the screen. This abstraction of the zoom from the number of visible pixels allows the user to set the zoom that he wants to be displayed regardless of the screen size of the device on which the game will be played.

- Max Zoom Rows: The maximum number of rows that can be visible on the screen when zooming out. Therefore, the minimum zoom of the map.

- Min Zoom Rows: The minimum number of rows that can be visible on the screen when zooming in. Therefore, the maximum zoom of the map.

- Limit View To Map: A Boolean value that tells if the player can scroll outside of the map's bounding box.

- Screen Size: It tells what is the size of the screen on which the game is being played. Knowing it the user can establish specific features depending on the resolution of the device and adapt his or her animations and visuals.

User Input Handling

The user has the following possibilities regarding the controls inside the map:

- Scroll: Moving through the map to change the visible contents in the screen. It is performed when the player pans with one single finger (or touch):

Figure 29: The pan movement

- Zoom: Changing the scale of the map in order to adapt the size of the visible parts of the screen to his desires. The user can either zoom in (make content bigger in order to see it with a higher detail) or zoom out (make content smaller in order to fit more elements inside the screen and have a more general view of the map). It is performed when the player pinches or zooms with two or more fingers:

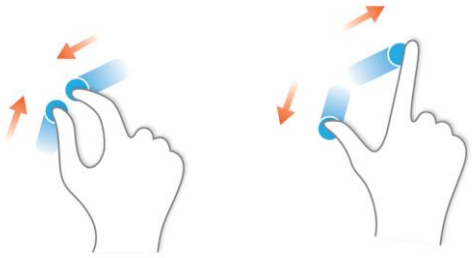

Figure 30: Pinch and Pan movements

- Selection: Selecting a *Game Element* (*Unit* or *Building*) in order to perform actions with them, or selecting a position to move them or to set it as target of an *Ability*. It is performed when the player taps a valid position inside the map:

Figure 31: Tap selection

In order to allow the player to zoom, multi-touch must be enabled on the device; this is automatically done when building the project for android devices (and is controlled thanks to the *AndroidManifest* file, but for iOS development multi-touch must be enabled in *AppController's didFinishLaunchingWithOptions* function, adding one line into the source code:

```
...
_viewController.wantsFullScreenLayout = YES;
_viewController.view = eaglView;
[eaglView setMultipleTouchEnabled:YES]; //this line of code
...
```

Notice that this code pertains to the Objective C language, but Cocos2d-x automatically provides you with access to the iOs wrapper (with the following files: *main.m*, *AppController.mm* and *RootViewController.mm*) inside the iOs folder.

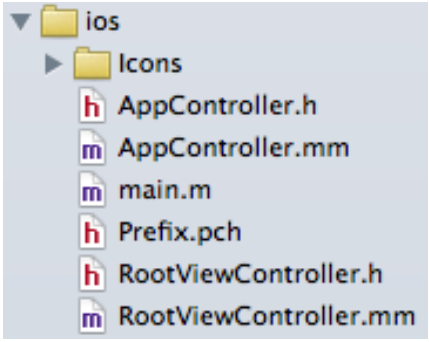

Figure 32: Structure of the iOS specific code inside Cocos2d-x

To implement the handling I programmed callbacks for all the multi-touch functions of Cocos2d-x and specified the following variables:

- Number of Touches: The number of started touches on the device that haven't still ended. When there is only one touch the player is scrolling, when there are more he or she is zooming in or out.

- Id Selection Touch: The id of the first touch in the device (only valid if no more touches are started, since it is used for knowing when the player is trying to tap for selecting a *Game Element* or a position in the map).

This is the code in the callbacks that handle it:

```
void LayerMap::onTouchesBegan(std::vector<cocos2d::Touch*> touches)
{
    number_of_touches += touches.size();
    if (number_of_touches == 1)
    {
        id_selection_touch = touches[0]->getID();
    }
    else
    {
        id_selection_touch = INVALID_ID;
    }
```

```
}
```

At first we store the total number of started touches. When there is more than one touch being performed at the same time, the invalid id of the touch tells us that the player is not trying to perform a tap.

```
void LayerMap::onTouchesMoved(std::vector<cocos2d::Touch*> touches)
{
    if (number_of_touches == 1 && !touch_lock)
    {
        //the player is scrolling
        manageScroll(touches[0]);
    }
    else if (number_of_touches >= 2 && touches.size >= 2)
    {
        //the user is zooming
        manageZoom(touches[0], touches[1]);
        touch_lock = true;
    }
}
```

When the player moves his or her fingers across the screen, we detect how many touches have been started with the *number_of_touches* variable.
If there are two or more fingers moving at the same time, we know that the player wants to zoom, so we handle it with the *manageZoom* function and set the *touch_lock* so that a selection cannot be performed while this zooming keeps going on.
Otherwise, if there is single touch (*touch_lock* = false), then the action is scrolling, so we handle it with the *manageScroll* method.

```
void LayerMap::onTouchesEnded(std::vector<cocos2d::Touch*> touches)
{
    number_of_touches -= touches.size();
    if (number_of_touches <= 0)
    {
        number_of_touches = 0;
        touch_lock = false;
    }
    if (touches.size() == 1 && !touch_lock)
    {
        Touch* touch = touches[0];
        if (touch->getID() == id_selection_touch)
```

```
        {
            manageAction(touch);
        }
    }
}
```

When all the touches that the player had started end, we reestablish the value of the *touch_lock* variable and ensure that the *number_of_touches* is zero.
If only one single touch has ended and there was no zooming being performed, then we know the player was performing a tap and we handle it with the *manageAction* function.

These are the management functions that handle the specific actions:

- Manage Scroll:

```
void LayerMap::manageScroll(Touch* touch)
{
   cocos2d::Node* layer_map = getChildByTag(ID_LAYER_MAP);

   double total_distance = touch->getStartLocationInView().getDistance(
                                touch->getLocationInView());
   if (total_distance >= distance_no_selection)
   {
      //if the player has moved too much his finger he does not want to select
      id_selection_touch = INVALID_ID;
   }

   double distance_x = touch->getPreviousLocationInView().x -
                       touch->getLocationInView();
   double distance_y = touch->getLocationInView().y –
                       touch->getPreviousLocationInView().y;

   double prev_pox_x = layer_map->getPositionX();
   double prev_pos_y = layer_map->getPositionY();

   double new_pox_x = prev_pos_x – distance_x;
   double new_pos_y = prev_pos_y – distance_y;

   cocos2d::Rect screen_rect (0, 0, screen_size.width, screen_size.height);

   layer_map->setPositionX(new_pos_x);
   if (limit_view_to_map && !rectContainsRect(layer_map->getBoundingBox(),screen_rect))
   {
```

```
      //we revert the movement in the x axis
      layer_map->setPositionX(prev_pos_x);
   }

   layer_map->setPositionY(new_pos_y);
   if (limit_view_to_map && !rectContainsRect(layer_map->getBoundingBox(),screen_rect))
   {
      //we revert the movement in the y axis
      layer_map->setPositionY(prev_pos_y);
   }
}
```

At first, we check the total distance the finger has covered while touching the screen and if it is longer that *distance_no_selection* (which has been set to 15 pixels by observation, although can be set any other value by the user), we decide that then the player just wants to scroll and does not want to select a single position to perform and action, so we invalidate the *id_selection_touch* to disable selection.

Afterwards, we compute the distance scrolled in the last callback and compute the new position for the layer.

If the user has decided to limit the view to the map, we must check if the screen doesn't contain any space that doesn't pertain to this layer and, if this is the case, revert the movement in any of the axis that causes this problem.

We do this using an inverse approximation, checking if the *Bounding Box* of *LayerMap* contains the *Bounding Box* of the screen by using the *rectContainsRect* function, which internally checks if both the bottom left and top right points of the screen are contained in the *LayerMap's* Bounding Box:

```
bool rectContainsRect(const cocos2d::Rect& container, const cocos2d::Rect& contained)
{
   cocos2d::Point bottom_left (contained.getMinX(), contained.getMinY());
   cocos2d::Point top_right (contained.getMaxX(), contained.getMaxY());

   bool contains = (container.containsPoint(bottom_left) &&
                    container.containsPoint(top_right));
   return contains;
}
```

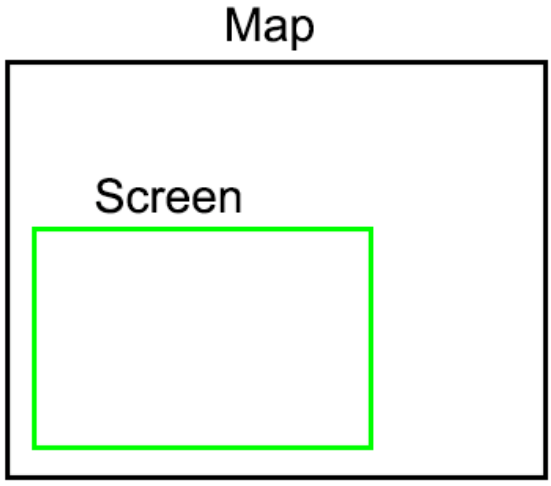

Figure 33: Example in which the screen's bounding box fits inside the map

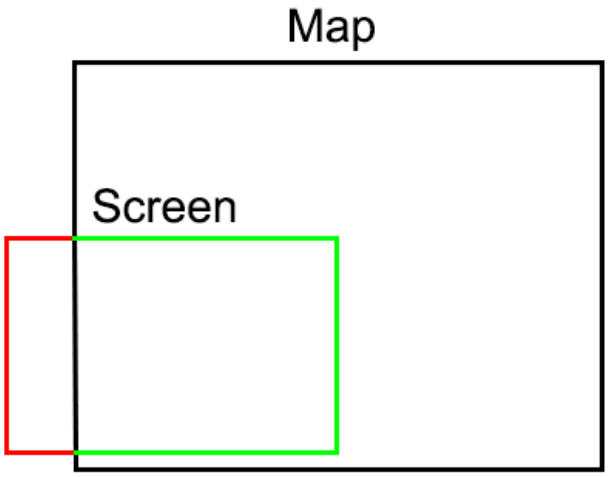

Figure 34: Example in which the screen's bounding box doesn't fit inside the map

- Manage Zoom:

```
void LayerMap::manageZoom(cocos2d::Touch* touch1, cocos2d::Touch* touch2)
{
   cocos2d::Node* layer_map = getChildByTag(ID_LAYER_MAP);

   // get current and previous positions of the touches
   cocos2d::Point curr_pos_touch1 = Director::getInstance()->convertToGL
                                                (touch1->getLocationInView());
   cocos2d::Point curr_pos_touch2 = Director::getInstance()->convertToGL
                                                (touch2->getLocationInView());

   cocos2d::Point prev_pos_touch1 = Director::getInstance()->convertToGL
                                                (touch1->getPreviousLocationInView());
   cocos2d::Point prev_pos_touch2 = Director::getInstance()->converToGL
                                                (touch2->getPreviousLocationInView());

   cocos2d::Point curr_pos_layer = curr_pos_touch1.getMidPoint(curr_pos_touch2);

   // calculate new scale
   double prev_scale = layer_map->getScale();

   double curr_scale = prev_scale * (curr_pos_touch1.getDistance(curr_pos_touch2) /
                                     prev_pos_touch1.getDistance(prev_pos_touch2));

   if (min_scale != INVALID_SCALE && (curr_scale < min_scale))
   {
      curr_scale = min_scale;
   }
   else if (max_scale != INVALID_SCALE && (curr_scale > max_scale))
   {
      curr_scale = max_scale;
   }

   layer_map->setScale(curr_scale);

   // if the scale has been changed -> fix position accordingly
   if (curr_scale != prev_scale)
   {
      Point real_curr_pos_layer = layer_map->convertToNodeSpace(curr_pos_layer);
      double delta_x = real_curr_pos_layer.x * layer_map->getContentSize.width() *
                       curr_scale – prev_scale;
      double delta_y = real_curr_pos_layer.y * layer_map->getContentSize.height() *
                       curr_scale – prev_scale;
      layer_map->setPosition(layer_map->getPositionX() – delta_x,
                             layer_map->getPositionY() – delta_y);
   }
}
```

We compute the new scale for the layer by getting the amount of pixels moved for the two touches (getting the distance between the current position and the previous one) and making this scale proportional to the current scale.
Once we have this scale value, we check that it fits between the limits established by the user for the maximum and minimum number of visible columns. If it is bigger than this maximum, then we set it to have the maximum value. This same operation is performed in case it is lower than the minimum.
Once we have applied this new scale, we also have to move the layer in order to adapt the center of the screen to the point between the two fingers, which is done by computing the delta values for the offset in the positions.

- Selection:

```
void LayerMap::manageAction(cocos2d::Touch* touch)
{
   cocos2d::Node* layer_map = getChildByTag(ID_LAYER_MAP);

   //the position is relative to the original size of the layer (without scale)
   Point position_in_layer = layer_map->convertToNodeSpace(
            cocos2d::Director::getInstance()->convertToGL(touch->getLocationInView()));

   //this is the original size without any scale
   cocos2d::Size layer_size (layer_map->getContentSize());

   bool inside_layer_x = (position_in_layer.x >= 0 &&
                          position_in_layer.x <= layer_size.width);
   bool inside_layer_y = (position_in_layer.y >= 0 &&
                          position_in_layer.y <= layer_size.height);

   bool inside_layer = inside_layer_x && inside_layer_y;
   if (inside_layer)
   {
      int pos_x = position_in_layer.x / (layer_size.width / n_columns);
      int pos_y = position_in_layer.y / (layer_size.height / n_rows);

      pos_y = flipPosition(pos_y, n_rows);

      Strategy2D::getInstance()->getActionManager()->handleTouch(Position(pos_x,
                                                                  pos_y));
   }
}
```

At first we get the position of the touch inside the Layer that contains the map in terms of the original size. Also, we get this original size from the Layer with the *getContentSize* method.
Then, we do a check to see if the touch fits inside the Layer's dimensions (because if the view is not limited to the map the player can also touch in empty space). If this is the case, then we need to discretize this touch in terms of the number of columns and rows in which the Layer is divided.
For the case of the Y coordinate, we also need to flip it since Cocos2d-x's coordinate system considers that the Y-axis starts at the bottom while we consider it to begin at the top:

```
double LayerMap::flipPosition(const double& pos, const double& number_of_positions)
{
   return (number_of_positions – 1 – pos);
}
```

Finally, we call the *handleTouch* function of the *ActionManager*, which knows what is the current state of the game and will be able to decide what are the intentions of the player (if he or she is selecting a *Unit* or *Building*, a position to move them, etc) and how these will affect the game state.

5.3.2 ButtonAbility

It is the implementation of a button that contains the required information of Unit *Ability*.
They are used when the player has selected a Unit and wants to perform an ability with it. The system then checks for the Abilities available for the *Unit* and displays them as multiple *UI Buttons*, each one assigned to one of these *Abilities*, so that the player can select the one he wants.
They are implemented as a subclass of a Cocos2d-x *Button* and have an additional parameter that contains the name of the *Ability* to which it is assigned.

Figure 35: Sample of an Ability Selection Menu, composed by ButtonAbility

Each *Ability* has an associated *Sprite* for the *Button*, set by the user, who can choose to display any kind of image or text. They are positioned in the location of the screen specified by the user and having the desired size in proportion to the screen.

The callback for the touch event is implemented within the context of the *LayerMap*, which detects what is the ability name stored in the button and passes this information to the *ActionManager*, who interacts with *ControllerGame* to get the range of the selected ability, which is later displayed.

5.3.3 ButtonRecruit

They are used for recruiting *Units* when a recruitment *Building* is selected. They display the array of possible *Units* to be recruited by the selected *Building*. They are also implemented as a subclass of a Cocos2d-x *Button* and have an associated *Sprite*.

Figure 36: Sample of Recruit Selection Menu, composed by ButtonRecruit

They have two additional parameters: One, the name of the *BaseUnit* they represent and the other, a Boolean value that tells if the *Unit* can be recruited by the team, which is associated to a function that allows the user to configure an economic system for his or her game. This way, the user can set different Sprites for the case when the *Unit* can be recruited or the opposite. He can also use the *Sprite* to show the cost of each type of *Unit*.

Figure 37: Another sample of a Recruit Selection Menu, this time showing unaffordable items

5.3.4 Platform mobility

Since one of the main objectives of the project is to offer a high mobility of the engine between different platforms and devices, a special effort of the development has gone into making all of the elements compatible and completely scalable.

Cocos2d-x provides an abstraction layer from the hardware in terms of rendering and user-input handling, but the scale of the graphics needs to be handled accordingly to the size of the screen of the device, without making wrong scales that would crush the original proportions of the *Sprites*.

Figure 38: An example of bad scaling, in which the original proportions of the sprite are broken

For this reason, as I have explained in the *LayerMap* section, I decided to use the number of visible columns for the zoom as an abstraction to compute the size of each *Sprite* in the map. Whenever the user changes the zoom value, the engine automatically scales the sizes of the whole map to fit the screen size of the device using the following function to compute the new scale:

```
Double scaleForZoom(const double& visible_columns)
{
   //width in pixels for each tile so that the screen fits exactly visible_columns
   double tile_width_to_fit = screen_size.width / visible_columns);
   //width in pixels for the whole map so that the screen fits exactly visible columnns
   double map_width_to_fit = num_columns * tile_width_to_fit;

   double scale = map_width_to_fit / map_layer->getContentSize().width;
   return scale;
}
```

Also, since the number of *visible_columns* is a floating-point value, it accepts any kind of configuration without ugly scaling jumps.

An example of a configuration would be the following, in which we set the number of visible columns to 7 and test the game executing it both with landscape and portrait orientation:

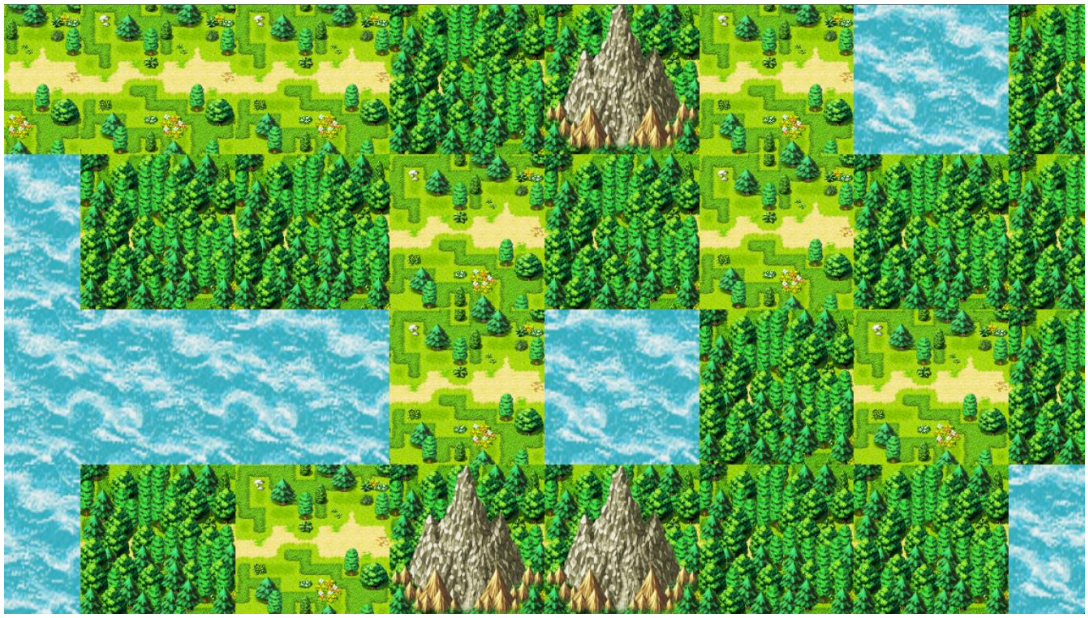

Figure 39: A sample of a map being displayed on an iPhone 4s using landscape configuration

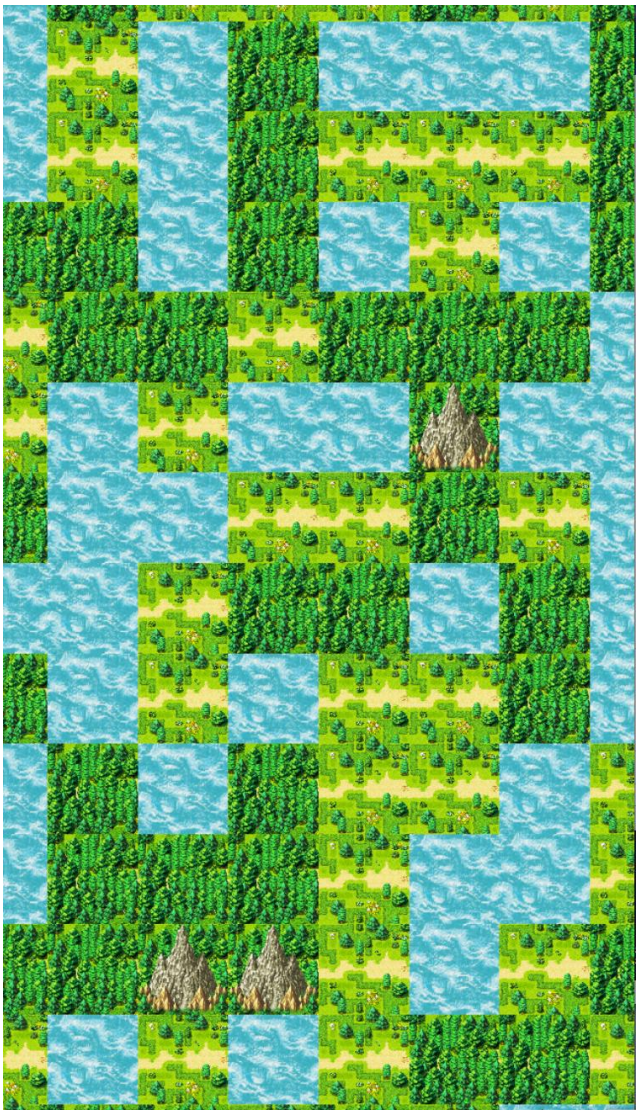

Figure 40: A sample of the same map being displayed on an iPhone 4s using portrait configuration

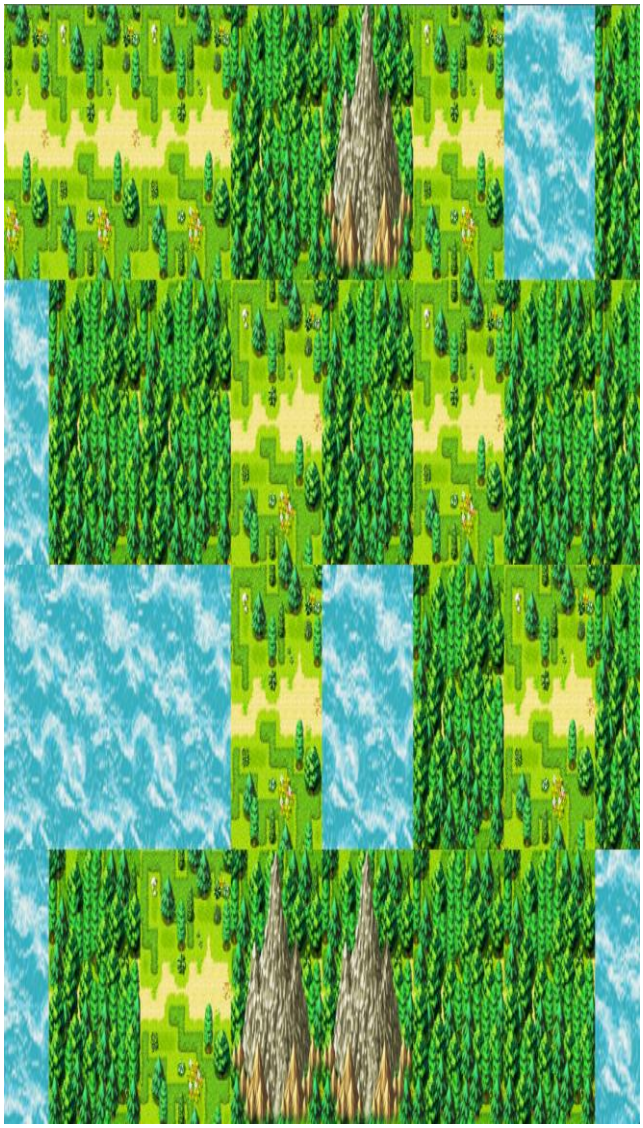

Figure 41: A sample of a bad scaling of the map, which would happen in case we tried to show the same content regardless of the proportions

This configuration allows the user to set up the visibility of the map so that there are no deformations and is equivalent in all devices. Also, he or she can even configure different zooms taking into account the size of the device's screen.

Thanks to this structure we also keep total compatibility between the touch controls in all platforms, since all the touch handling is performed in terms of Layer proportions (as seen in the user-input section).

5.4 Domain Layer

I will start by explaining what is the normal flow of the game, taking into account the available control scheme in mobile devices, without any buttons and limited to their touch screen.

Later, I will define what are the elements that compose the games and explain how the *ControllerGame* class contains and handles them.

One previous consideration is that the users will be able to modify the state of these game elements and reestablish their configurations through the defined callbacks, which will grant them access to these elements, as I will explain in the following sections.

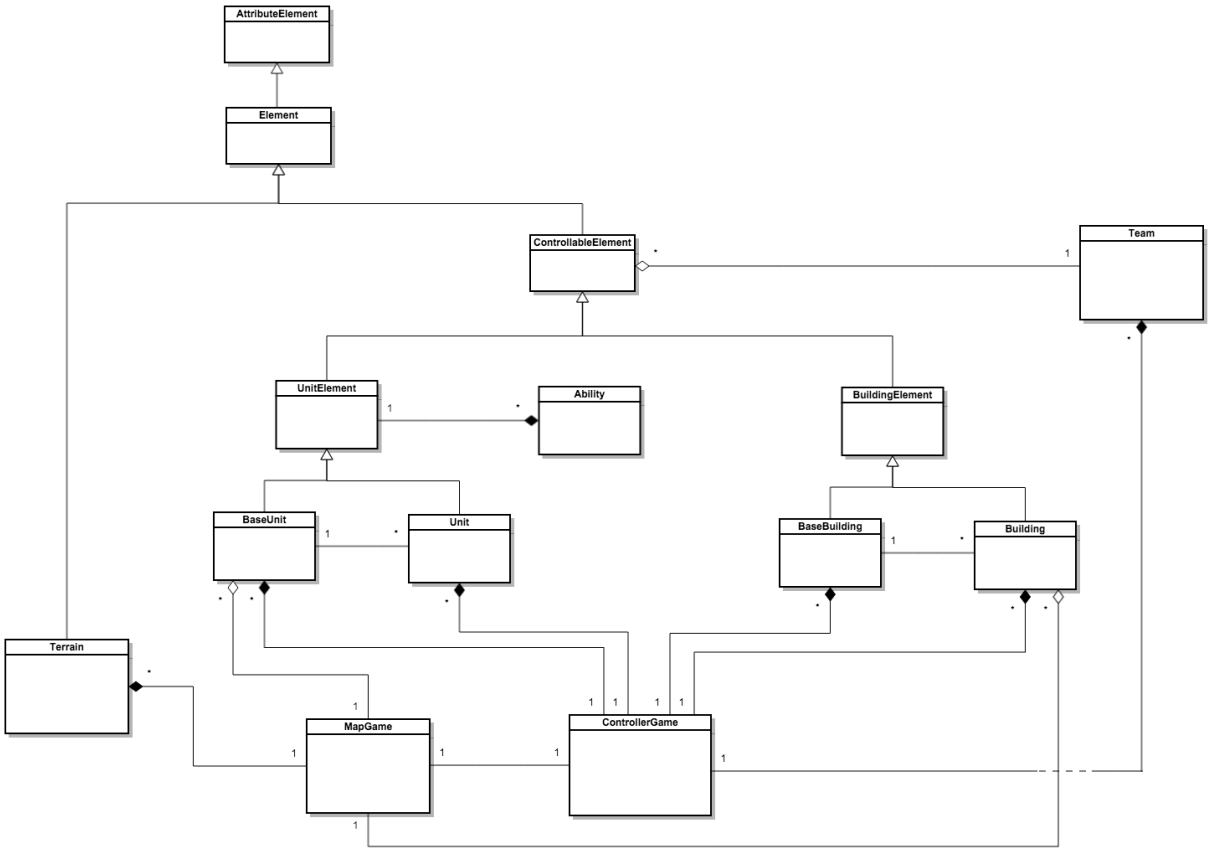

Figure 42: Domain Layer diagram

5.4.1 ActionManager and the Game Flow

Whenever the user wants to perform an action with its *Units* or *Buildings*, these actions must be handled in order to affect the system in the intended way so that it responds accordingly.

To manage the effects of these actions, we have defined a class named *ActionManager* that handles them and ensures that the player acts in the intended way and following a certain order.

This defines the normal Game Flow by which the player is able to control his or her *Units* and *Buildings*:

5.4.1.1 Unit Action Flow

- *Units* and *Buildings* are positioned across the map:

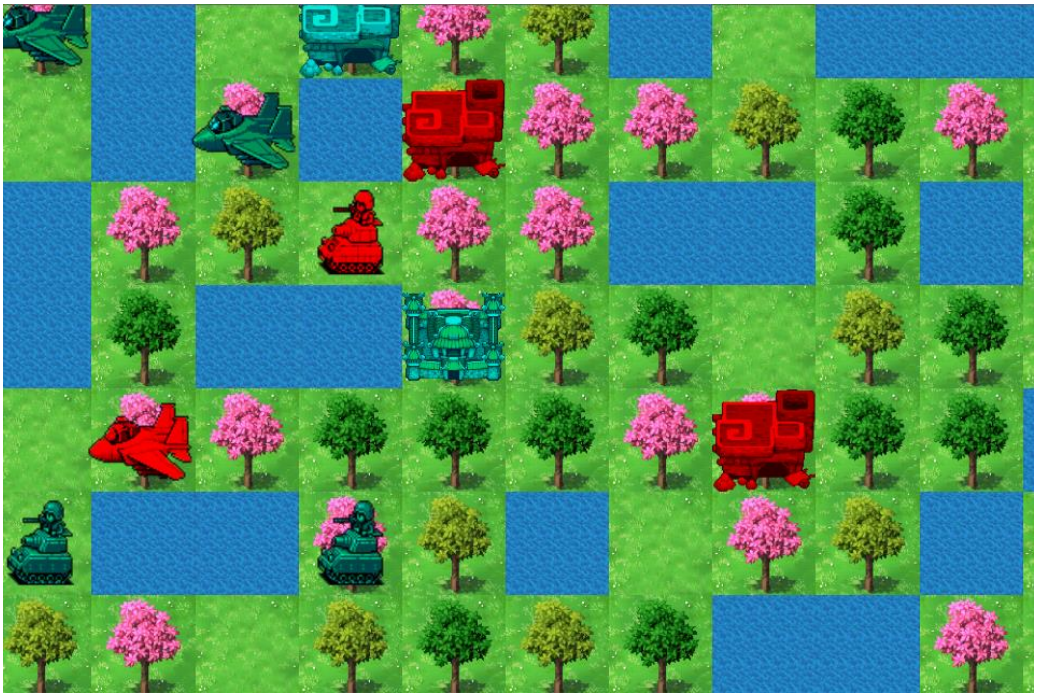

Figure 43: Normal map layout

- The player taps the screen at a certain position, occupied by the *Unit* he or she wants to select:

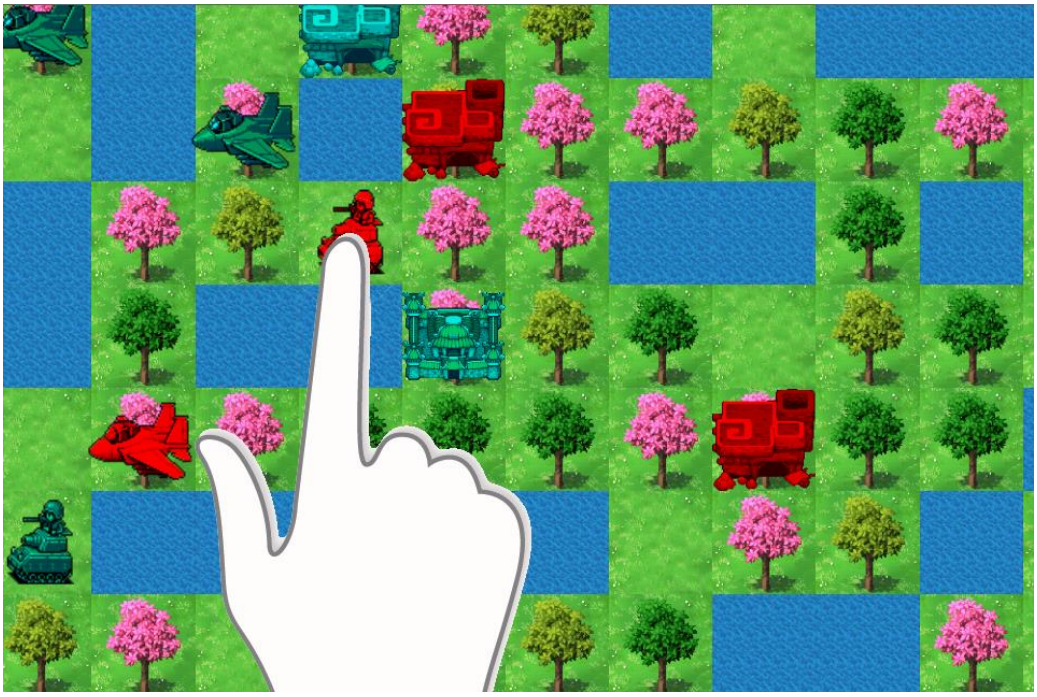

Figure 44: The player taps on a Unit to select it

- The system computes the possible movements for the selected *Unit* and displays which are the reachable positions by showing a special layer on top of them (in this case with a transparent white color):

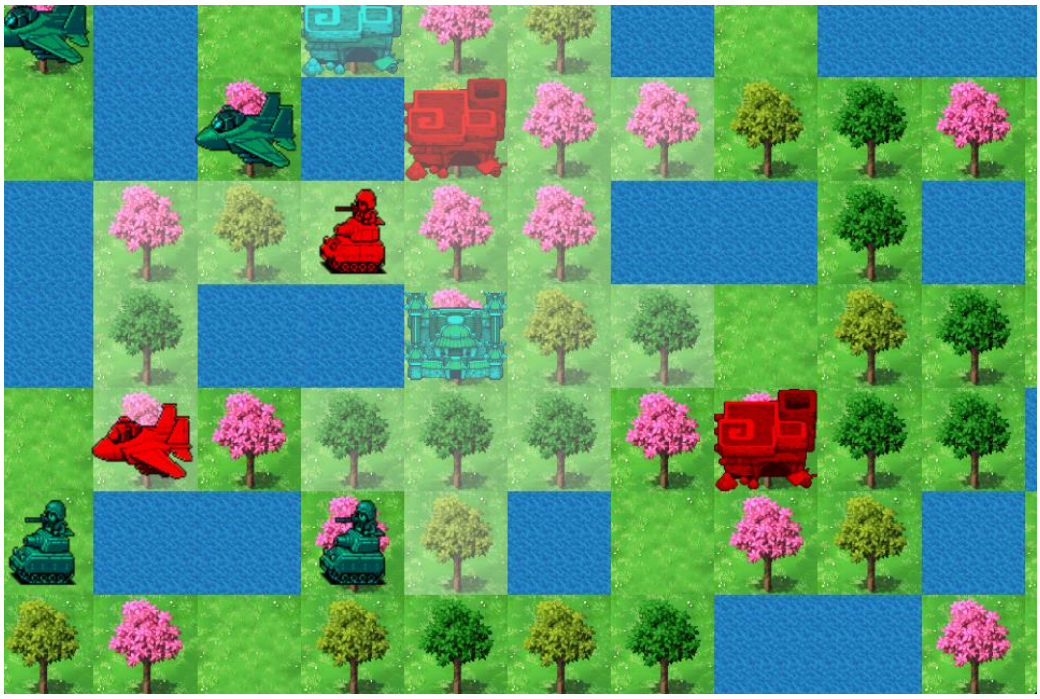

Figure 45: The system displays the positions the selected Unit can move to

- The player selects one of the reachable positions for moving the *Unit*:

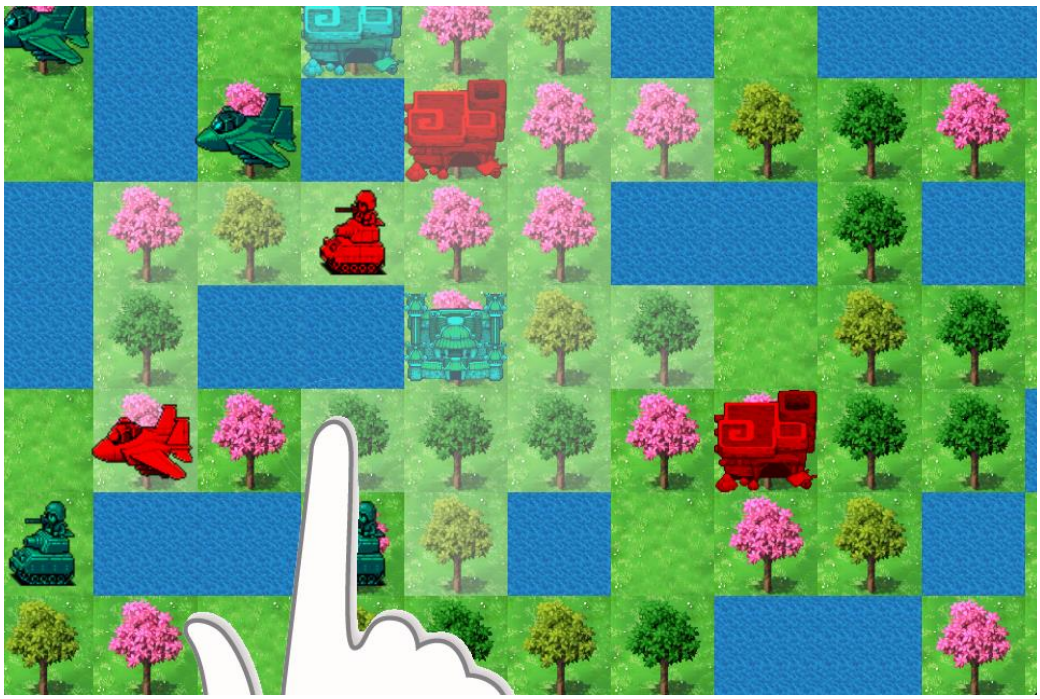

Figure 46: The player taps at the position he wants to move the Unit

- The Unit moves to the position through the shortest path and the *Ability* Selection menu is displayed:

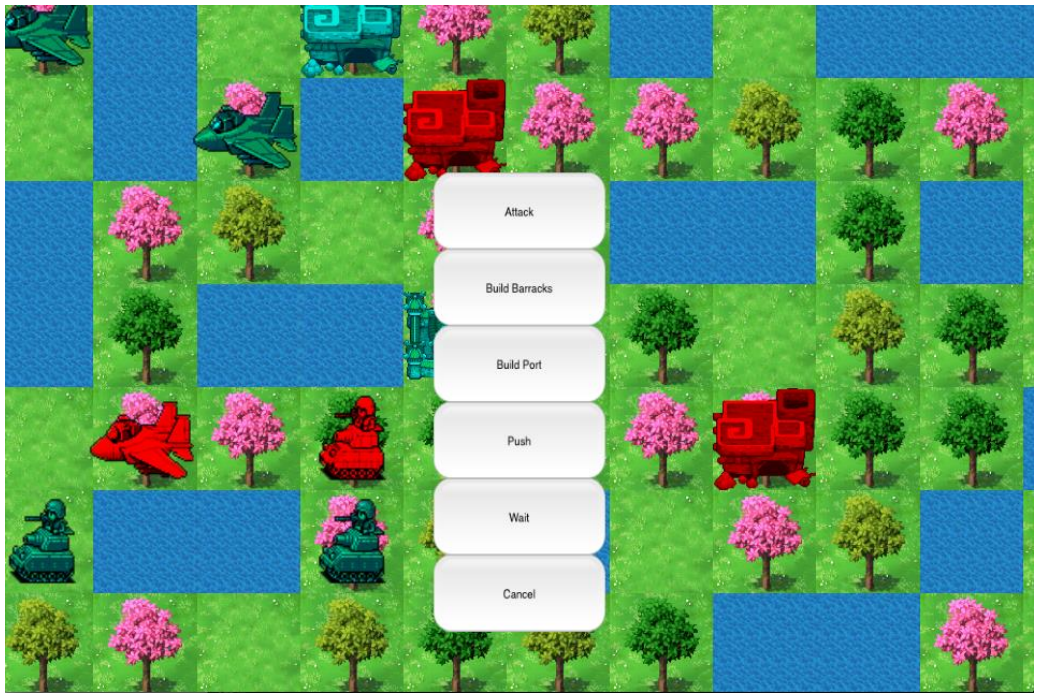

Figure 47: The system displays the Ability Selection Menu

- The player selects one of the available *Abilities* by touching the associated button:

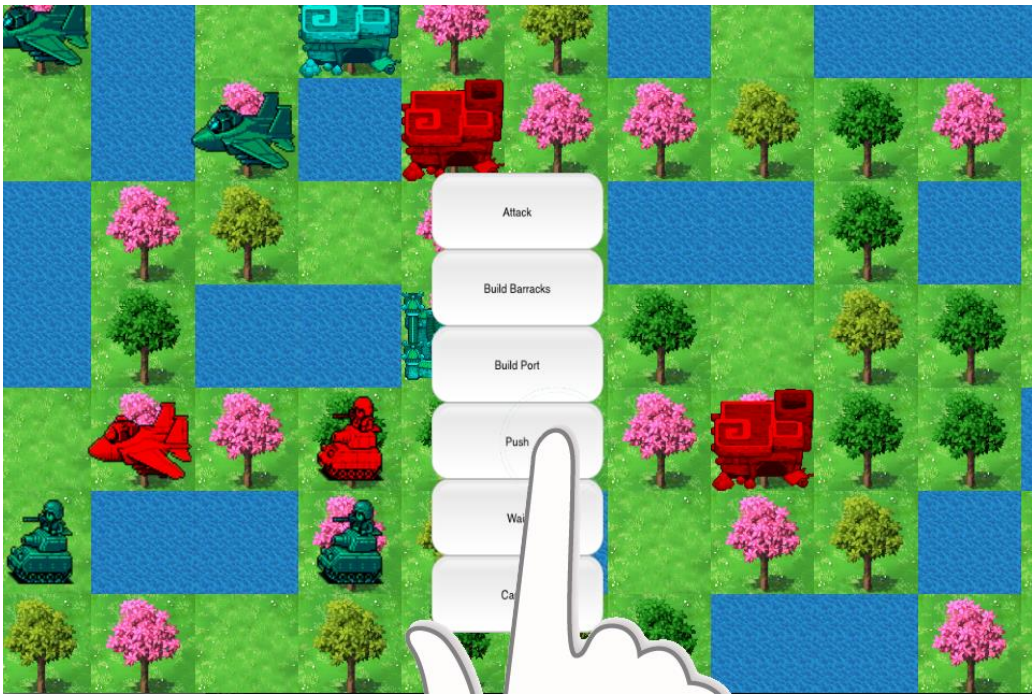

Figure 48: The user selects the Ability he wants the Unit to perform

- The system computes which are the positions that can be targeted by the selected *Unit* and *Ability* and displays them again with a special layer:

Figure 49: The system shows the positions at which the Ability can be targeted

- The player selects one of the reachable positions to perform the *Ability*:

Figure 50: The player selects the target position for the Ability

- The *Ability* is then performed and the *Unit* is unselected (in this case the "push" *Ability* pushes the target *Unit* into the opposite direction. Also, the user has decided in this case that *Units* can only perform one action per turn blacks, so the selected *Unit* is blacked out to indicate that it has already performed its action).

Figure 51: The Ability is performed

5.4.1.2 Building Action Flow

- *Units* and *Buildings* are positioned across the map:

Figure 52: A normal map layout

- The player tap at the screen at a certain position, occupied by a *Building*:

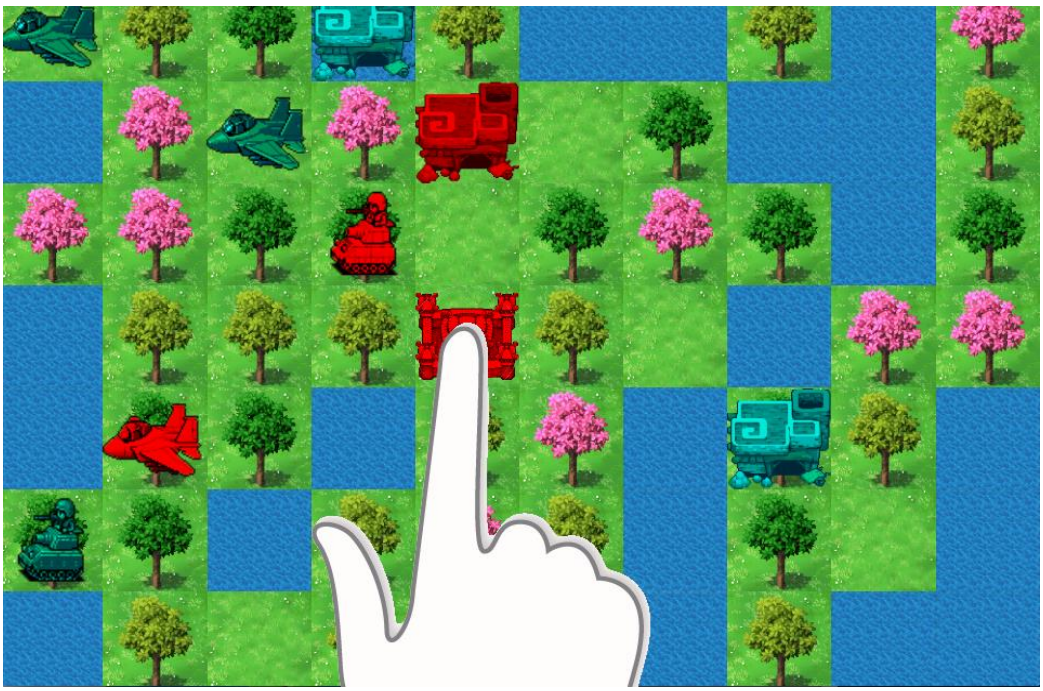

Figure 53: The player taps at a Building to select it

- The system loads which are the possible recruitments for the selected *Building*, and if it can recruit any type of *Unit*, then the recruit selection menu is displayed

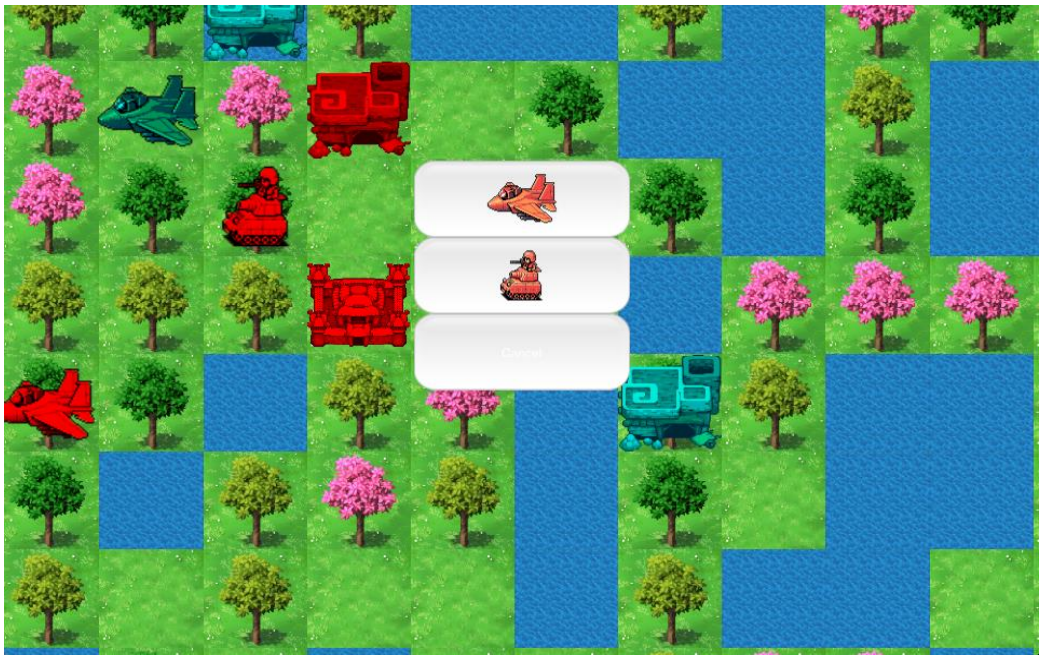

Figure 54: The system shows the Recruit Selection Menu, composed by ButtonRecruit

- The player selects one of the possible recruitments:

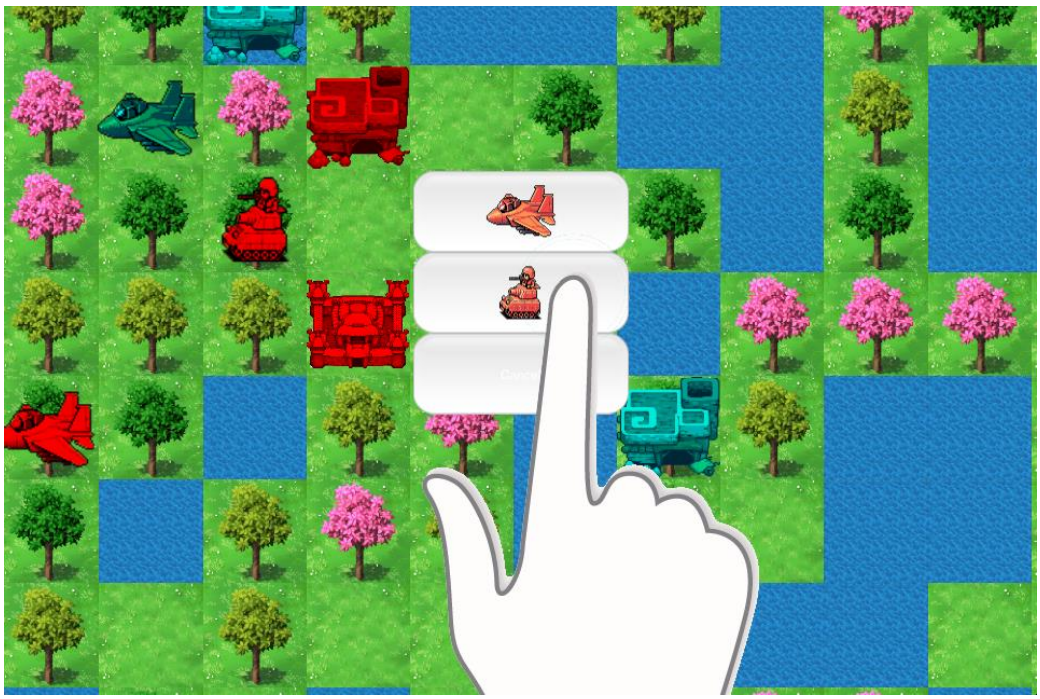

Figure 55: The player selects a BaseUnit to recruit

- If the selected recruitment is affordable by the *Team*, then the *Unit* is added to the map in the *Building's* position (in this case, as for the *Unit* Action Flow, the user has considered that a newly recruited *Unit* cannot move in the same turn it is recruited, so it also blacks it out).

Figure 56: The Unit is recruited at the Building's position

The is just the normal flow, since the player can also cancel some of his or her actions and move back to the prior state, as you will see in the flow charts of the following section.

5.4.1.3 The ActionManager class

To control the previously defined flow, it is necessary to have a special class that provides with an interface between the View Layer and the Domain Layer and that is able to handle the user-input directly obtained by the View Layer.
By keeping an internal state of the player's actions, this class decides what is the action the player wants to perform and how these actions affect the Domain Layer.
The View Layer provides it with a simple action performed by the player, which it handles. This action can be a single touch on the map, the pressing of an *AbilityButton* or the pressing of a *RecruitButton*.

All of these are connected, since *AbilityButtons* are only displayed when the player has previously moved a *Unit* and *RecruitButtons* when he or she has selected a *Building*.

These are the flow diagrams for these different cases:

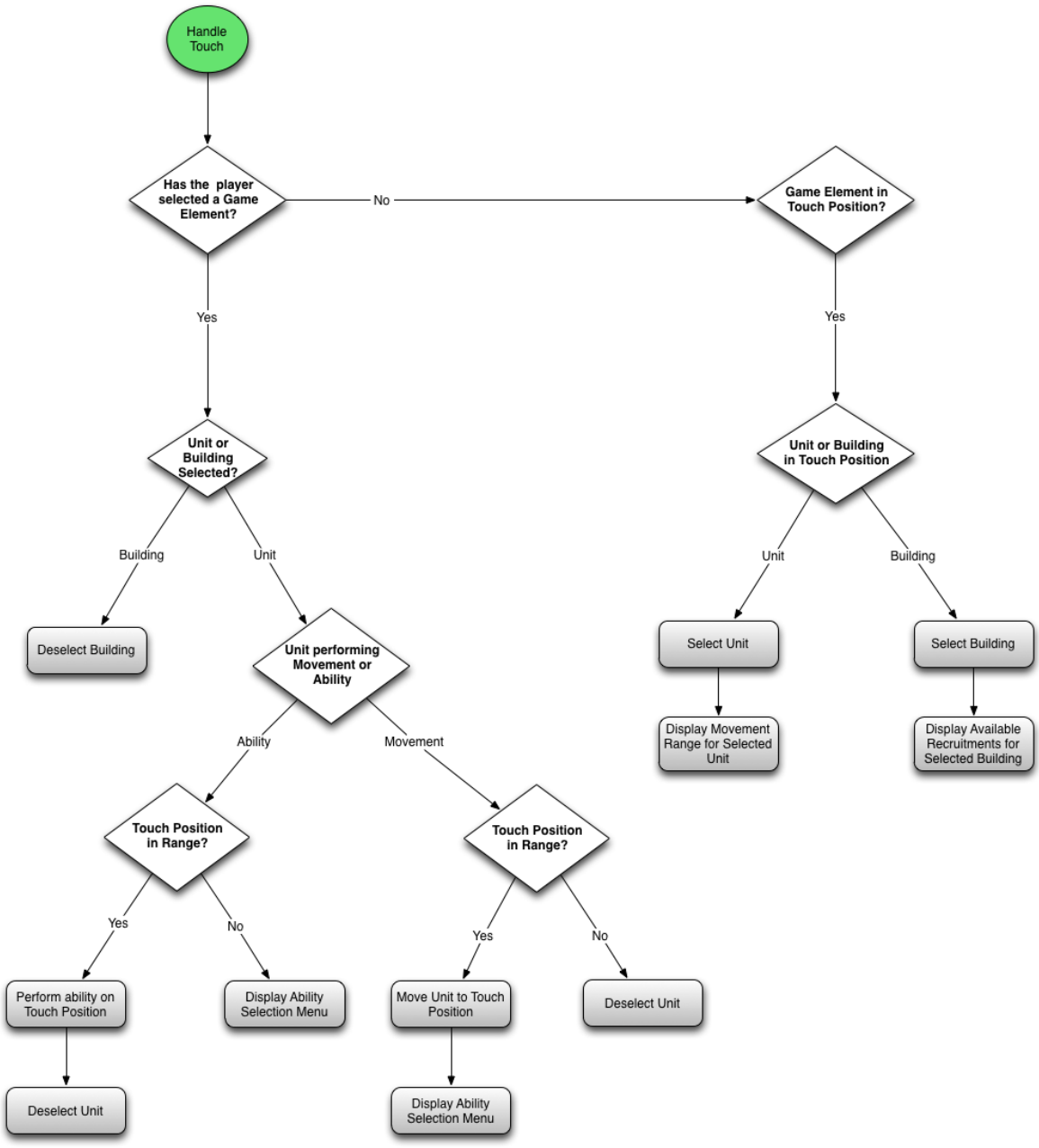

Figure 57: Handle Touch flow chart

This way, the system detects what is the action the player is trying to perform when selecting a position in the map and reacts accordingly, moving a *Unit* or performing an *Ability* when needed and displaying the right menus and visual indicators.

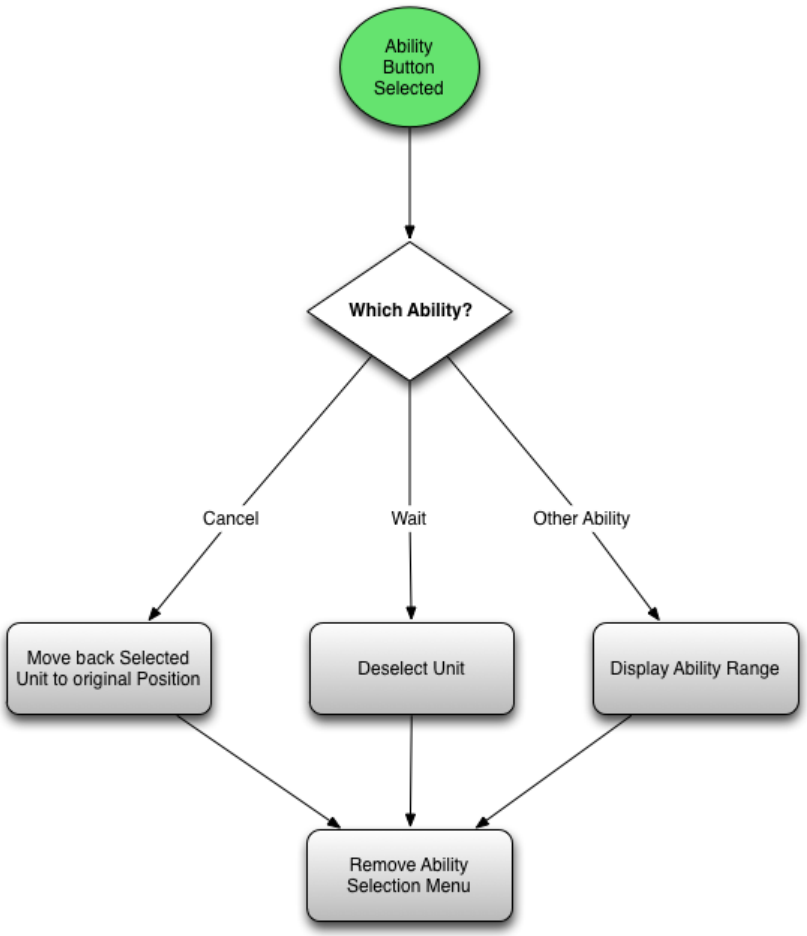

Figure 58: Ability Button Selected flow chart

When the player selects an *Ability* from the menu, *ActionManager* stores its name for performing it then when the player selects a position (as seen in the previous diagram). Additionally, it also computes which are the positions in range for the selected *Unit* and *Ability* and displays it for the player.

If the player selects the Wait *Ability*, then the *Unit* is simply unselected (remaining at the position to which the player has moved it in the previous action).

However, if the pressed button is not an *Ability* but instead the cancel button, the previous movement is undone so that the player can start from the beginning.

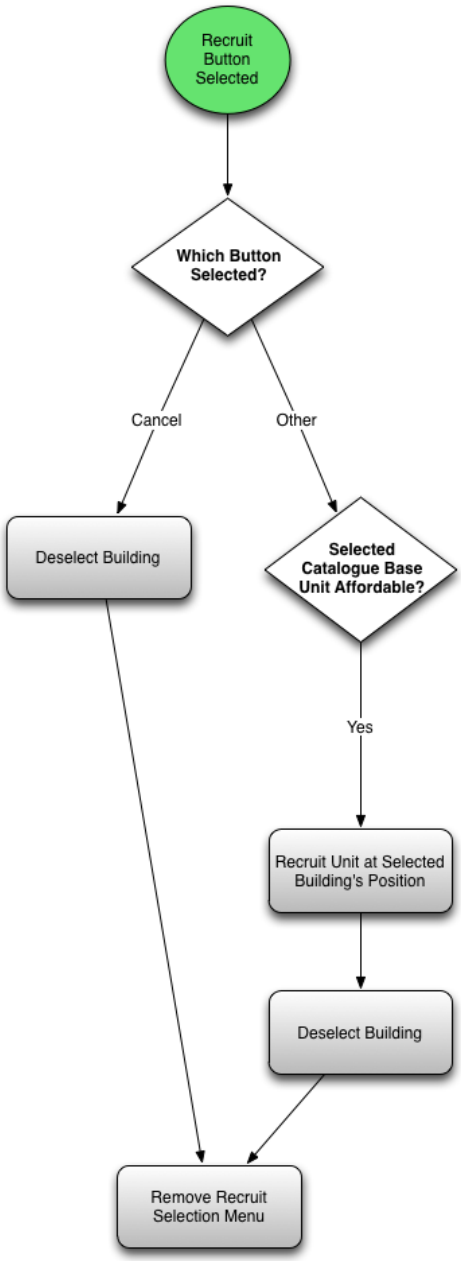

Figure 59: Recruit Button Selected flow chart

When the player selects a recruitment from the menu, *ActionManager* checks if it is affordable by the selected Unit's team by calling the user-defined function that decides it. If this is the case, then it handles how the purchase affects the Team and adds a *Unit* at the selected *Building's* position with the selected *BaseUnit* as template. Either if this is the case or the player has selected the cancel option, it deselects the *Building* and removes the menu from the screen.

If the selected recruitment is not affordable then nothing happens, so the system keeps waiting for the player to select a valid option.

5.4.2 Configurable Elements

They represent the elements to which the user has direct access and that he can define and modify during the course of the game.

5.4.2.1 Game Map

It is the representation of the two-dimensional board in which the game takes place; it is formed by a certain number of positions (defined by their vertical and horizontal alignment) in which the different kinds of Game Elements can be placed.

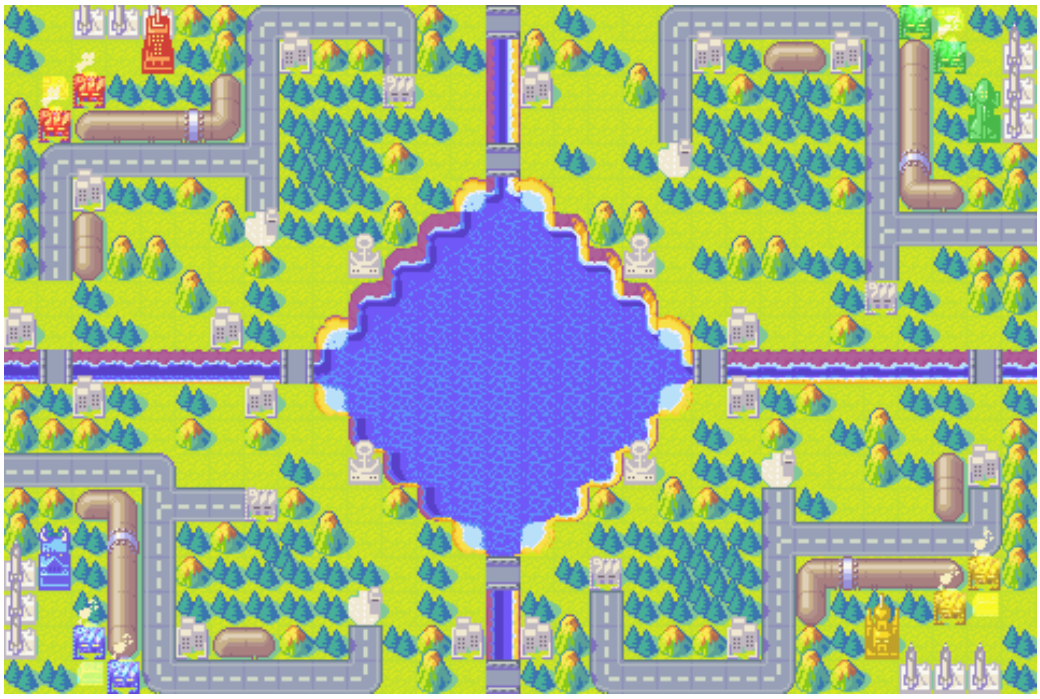

Figure 60: Game Map concept

In each position the following Game Elements can be found:

- Terrain
- Building
- Unit

There must be a *Terrain* associated to each one of the positions of the map, while the presence of *Units* or *Buildings* is not mandatory, since *Units* can move through them and *Buildings* can be created and removed from the map, as I will explain in the following sections.
In terms of visibility, the *Terrain* must always remain behind, while *Buildings* must be on top of *Terrains* and *Units* must be on top of both *Terrains* and *Buildings* (if they are present).

For example, in a case with this *Terrain*, *Building* and *Unit* occupying the same position:

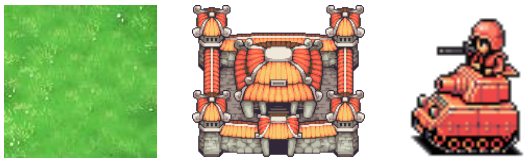

Figure 61: Terrain, Building and Unit sprites examples

The result should be:

Figure 62: The order of the rendering must be: Terrain, Building and Unit

To achieve this, for each position the order for the rendering of the sprites must be: *Terrain*, *Building* and *Unit*.
This is controlled by setting the value for the depth attribute (cocos2d-x *Z-Order*) of the sprite of each one of them.

5.4.2.2 Teams

Teams represent associations of *Controllable Elements* (*Units* and *Buildings*) that compete one against each other. They can be either controlled by the player or by the computer, for which the user must set an Artificial Intelligence algorithm that controls it .

Each *Team* has configurable attributes, which the player can access, define or modify at any time in any of the callbacks. This includes the definition of resources and anything related to its members.

Although the player manages them, they are not considered *Game Elements* since they are abstract representations and cannot be directly controlled; despite the fact the player indirectly does it by controlling their members.

Attributes

- name: The name of the *Team*, used as identifier within the system.
- units: The list of *Units* that form the *Team*.
- buildings: The list of *Buildings* that form the *Team*.
- color_enabled: It tells if the user wants the system to apply a color mask to the graphic representations of the members of the *Team*.
- color_elements: The color mask that is applied to the sprites of the *Team's* members (if it is enabled).
- color_movement_layer: The color of the layer that indicates the positions that are reachable for a member of the *Team* in a turn movement.
- turn_end: A Boolean value that tells if the *Team* has finished its movement in the current turn.

- AI_controlled: A Boolean value that tells if the *Team* is controlled by the Artificial Intelligence or, in contraposition, controlled by a player. If it is controlled by the Artificial Intelligence, then ControllerGame will control the actions of its members.

5.4.2.3 Game Elements

They are the representation of the individual entities that form the game. They have a unique identifier and a position inside the game map, as well as a graphic representation set by an instance of a Cocos2d-x Sprite class.
All of them have configurable attributes that can be set by the user thanks to their inheritance from the *AttributeElement* class, which defines a mapping of attribute name to value.
The final player can directly control some of them during the game, while some others cannot be controlled, which is the case of the terrains.

5.4.2.3.1 Terrains

They define each of the tiles that occupy a position in the game map, with their own traits, attributes and ways to affect the gameplay.
The user can access their public functions to configure them as they want and to make them react to the actions of the rest of game elements during the game course.

Figure 63: Examples of terrains

5.4.2.3.2 Controllable Elements

These are the game elements the player and the Artificial Intelligence will be able to control directly; they always pertain to a *Team* and have their own attributes that allow them to be extensively configured by the user (inherited from *AttributeElement*).
They have the following specific attributes:

- Team: A direct reference to the *Team* where they pertain.

- Blocking Actions: Since they can be controlled by an AI, the system needs to know which are the animations that need to be performed by Cocos2d-x. This allows the rest of *Controllable Elements* to start performing their actions only when the previous *Element* has finished theirs.
  This is also used when using the *kill* function for the *ControllableElement*, which will not remove it from the system instantly but wait for any of the added actions to be finished (this allows, for example, to perform death animations and wait for the attacking *Element* to finish its animations).
  This Blocking Actions attribute is simply a list of Cocos2d-x *Actions*, which the user can fill with those animations that need to be finished before the turn of the Element ends. When trying to decide if a certain *Element* controlled by the AI has finished its actions, *ControllerGame* will check if all the Cocos2d-x *Actions* referred in this list have finished their execution.

They are configured using a Base that defines their starting configuration when they are created and added to the game. This way the user can define templates for Element Types and later populate the game with actual instances.
There are two types of *Controllable Elements*: *Units* and *Buildings*.

5.4.2.3.2.1 Units

They move through the map and can perform abilities over the rest of *Game Elements* in the game. In most games they are the representation of the *Game Elements* that fight against each other. The structure of their implementation is the following:

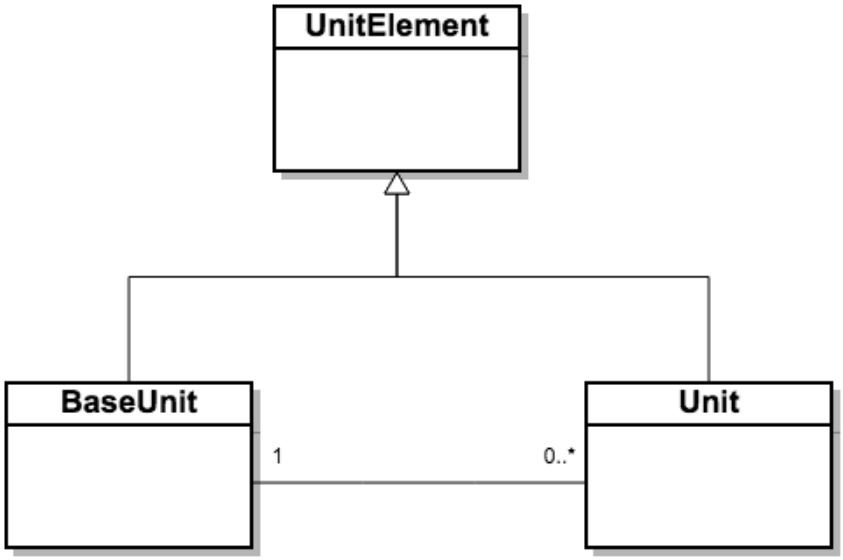

Figure 64: Unit class diagram

They are defined by a *BaseUnit*, which serves as a template for their creation and gives them their original attributes and graphic representation.
Since a *Unit* and a *BaseUnit* basically represent the same concept, with the exception that *Units* can be controlled and must contain information about their position and identifier, they both get their shared implementation from a parent class named *UnitElement*.

UnitElement

It contains the implementation of the shared attributes of *Units* and *BaseUnits*, which are the following:

- All user defined attributes that come from their inheritance from the *AttributeElement* class.

- Movement Points: The number of points a *Unit* has to move through map positions. This parameter is used for defining the range of movements for a *Unit* in a single turn, computed by the Path Finding algorithm.

- Movement Speed: The speed for the animations of their movement through the map in terms of number of positions per second.

- Can capture: A Boolean value that indicates if a *Unit* can capture *Buildings* or not.

- Abilities: The array of *Abilities* of the *Unit*, which will be explained later.

- Sprite: The graphic representation of the *Unit* in the map.

- Movement Blocks: A Boolean value that tells if the movement of a *Unit* should block the rest of Units from starting their actions. This is done with the same purpose as *ControllableElement's* Blocking Actions, but in this case the movement is directly controlled by the engine so it is the one who knows whether the movement has ended or not.

Unit

In addition to the attributes that come from their inheritance from *UnitElement*, they are also *Controllable Elements* and are included in the map, for this reason they also inherit from *ControllableElements* and, because of this, they have an id that identifies them in the game, an assigned *Team* and a position in the map.
Finally, they also contain a direct reference to their *BaseUnit* template, which tells what is their type and grants access to their original attributes.
Each turn, the player can move them through the map and perform abilities with them. For this reason, they can also contain ActionUnits that define what their actions will be for the current turn (they are formed by a position for movement, an *Ability* reference and a target position for the ability). These can be used when these *Units* are controlled by an Artificial Intelligence (to perform the actions when the right moment comes) and, for example, for having direct access to the players decisions and send them through the Internet to have an online multiplayer game.

BaseUnit

They define the original template of the *Unit* instances and, in some sense, their type of *Unit*. To be identified within the system, they have a unique name that is defined by the user when he or she creates them.
Also, since these are not meant to be unique for each *Unit* but be a definition for all *Units* that come from the same *BaseUnit*, they have two additional parameters:

- Sprites for Teams: A map from *Team* names to *Sprites* that defines what is the *Sprite* to be displayed for a *Unit* taking into account its *Team*. This allows the easy configuration of different *Sprites* for *Units* that pertain to the same type so that they can be distinguished.
  An example would be the following, in which we have two *Teams* that fight each other and where we want each one of them to have a different color code (one yellow and the other green):

Figure 65: Example of two different sprites associated to different Teams for the same BaseUnit

By configuring it, the Engine will automatically detect what is the *Team* of the *Unit* and display the *Sprite* associated to that *Team* for the *BaseUnit*:

```
Team team_green("Green");
Team team_yellow("Yellow");

BaseUnit base_tank("Tank");

cocos2d::Sprite* sprite_tank_gr = cocos2d::Sprite::create("tank_green.png");
cocos2d::Sprite* sprite_tank_yell = cocos2d::Sprite::create("tank_yellow.png");
base_tank.setSpriteForTeam("Green", sprite_tank_gr);
base_tank.setSpriteForTeam("Yellow", sprite_tank_yell);
```

- Possible Movements: A vector that defines what are the possible movements of a *Unit* in terms of moving up and down from their starting position. An example for this could be the definition of vectors that allow only moving to positions that are up, down, left or right ([1,0], [-1,0], [0,1], [0, -1]), which is the default setting, or enabling also diagonal movement ([-1,0], [-1,1], [-1,-1], [0,1], [0,-1], [1,0], [1,1], [1,-1] }:

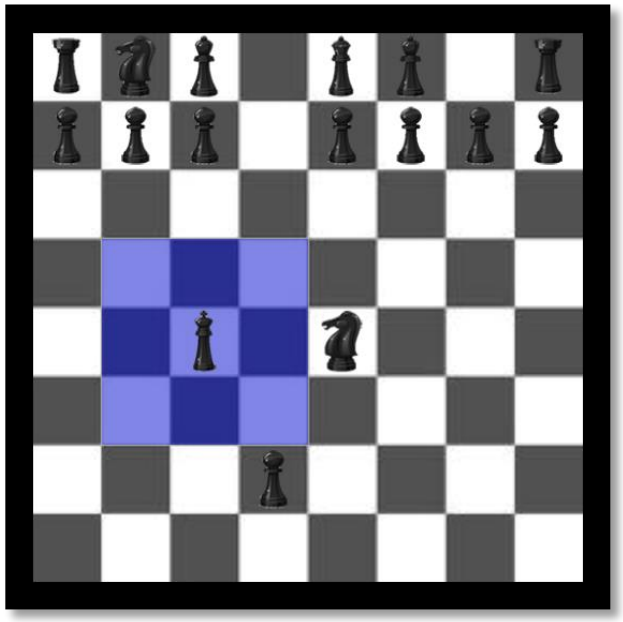

Figure 66: Chess movement for a King, implemented with the engine

But allowing to configure it allows far more possibilities such as the following (implementing a chess game with the engine):

- Horse movements using the following vector:
  { [2,1], [2,-1], [-2,1], [-2,-1], [1,2], [1,-2], [-1,2], [-1,-2] }

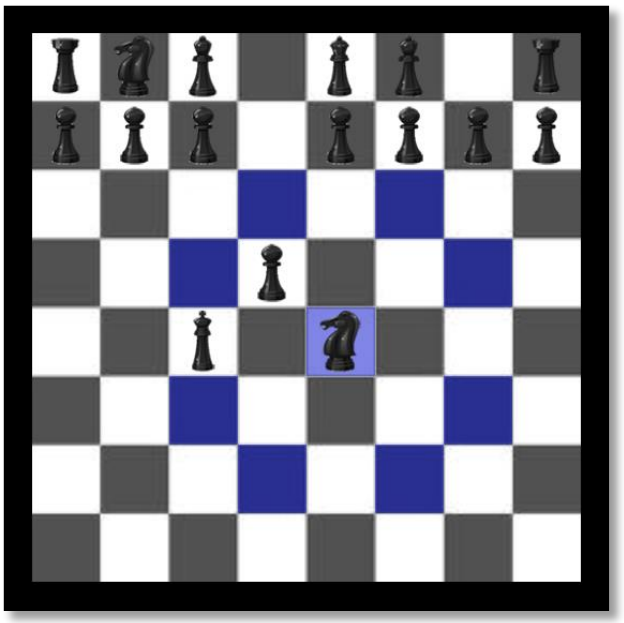

Figure 67: Chess movement for a Horse, implemented with the engine

- Queen movements using the following vector:
  { [-1,0], [-1,1], [-1,-1], [0,1], [0,-1], [1,0], [1,1], [1,-1]}

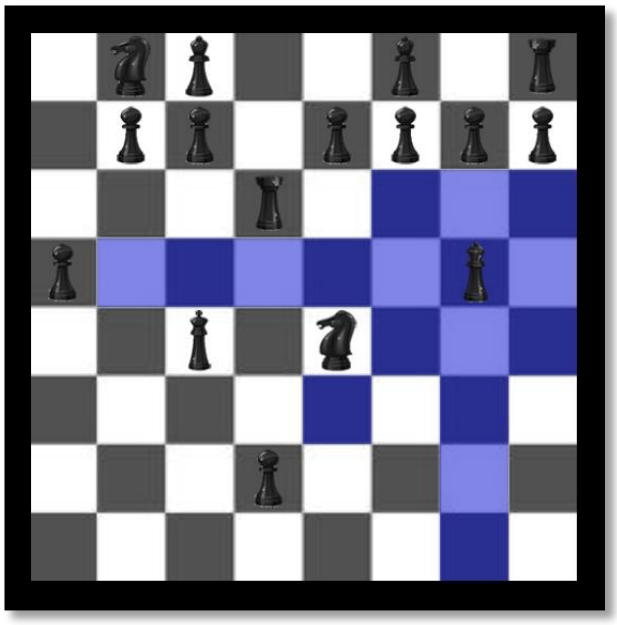

Figure 68: Chess movements for a Queen, implemented with the engine

- Bishop movements using the following vector:
  { [1,1], [1,-1], [-1,1], [-1,-1] }

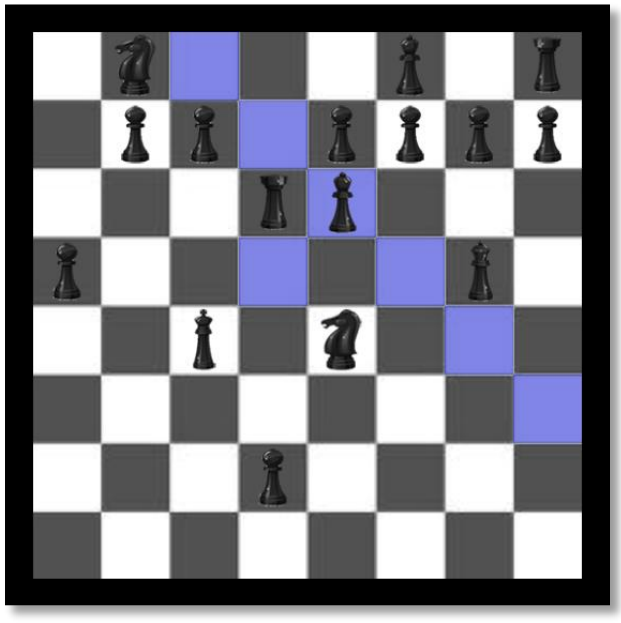

Figure 69: Chess movements for a Bishop, implemented with the engine

Implementation

To create a *Unit*, the user needs to provide the engine with the name of that *BaseUnit* that will define it and serve as template, the name of the *Team* this Unit will pertain to and that position in the map it will occupy. These names for the *BaseUnit* and the team must be already added into the system. An example for it would be the following:

```
BaseUnit base("Tank");
strategy->addBaseUnit(base);
Team team("Team1");
strategy->addTeam(team);

strategy->addUnit(row, column, "Tank", "Team1");
```

When this happens, the engine gets the instance of the *BaseUnit* and *Team* associated to this names and checks their presence in the system. If all goes well, then the *Unit* constructor will handle the creation of this new Unit taking into account the configuration of the *BaseUnit*, as follows:

```
Unit::Unit(long id, BaseUnit* base, Team* team)
: ControllableElement(id),
  UnitElement(*base->clone(team->getName()))
{
    ControllableElement::setTeam(team);
    this->base = base;
}
```

As it can be seen, the *UnitElement* part of the *Unit* instance is created by generating a clone of the *BaseUnit* passed by parameter. However, in this case the function is virtual and ends up calling the clone function defined in *BaseUnit* before, which will ensure that the right *Sprite* is set for the *Unit* taking into account the *Team* for the *Unit*:

```
UnitElement* BaseUnit::clone(const std::string& name_team)
{
    cocos2d::Sprite* sprite_team = getSpriteForTeam(name_team);
    UnitElement* unit_element = new UnitElement(*this);//this copies all the attributes

    if (sprite_team != NULL) //a Sprite has been defined for the team
    {
        cocos2d::Sprite* sprite = Utils::copySprite(sprite_team);
        unit_element->setSprite(sprite);
    }

    return unit_element;
}
```

Where the copy constructor for *UnitElement* is called and does the following:

```
UnitElement::UnitElement(const UnitElement &obj)
{
    setMovementPoints(obj.getMovementPoints());
    setCanCapture(obj.getCanCapture());
    setAttributes(obj.getAttributes());
    setAbilities(obj.getAbilities());

    if (obj.getSprite() != NULL)
    {
       sprite = Utils::copySprite(obj.getSprite());
    }
    else
    {
       sprite = NULL;
    }
}
```

So the *Unit* gets exactly the same attributes as the ones defined for the *BaseUnit* and a specific *Sprite* if it has been defined for its *Team* (otherwise it will get the normal Sprite defined for the *BaseUnit*).
Also, the *copySprite* function gets the texture associated to the source Sprite and creates a new copy for it (doing it recursively to get all of its children too).

5.4.2.3.2.2 Buildings

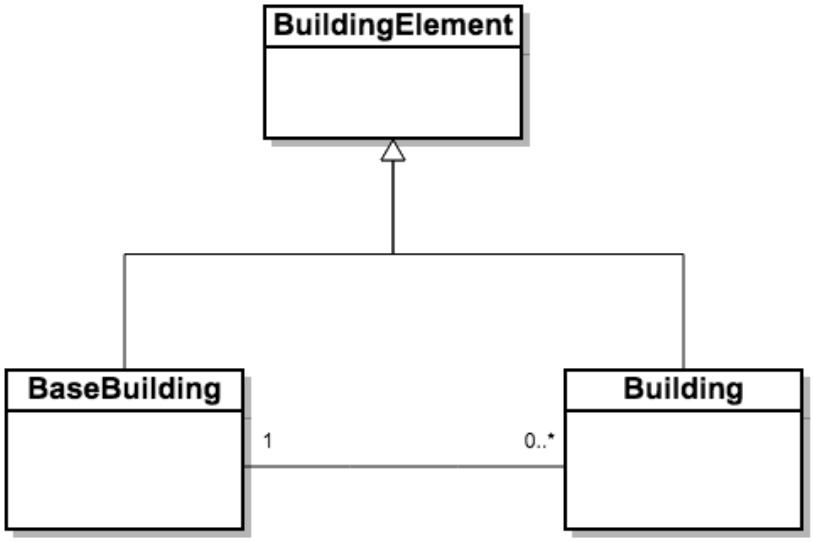

Figure 70: Building class diagram

As happens with *Units*, *Buildings* come defined by a *BuildingElement* class. This class ensures that most of the attributes are shared between the *Building* and *BaseBuilding* classes.

BuildingElement

As for *Units*, it contains the definition and implementation of the shared parts between *BaseBuilding* and Building:

- All user defined attributes that come from their inheritance from the *AttributeElement* class.

- Capture enabled: A Boolean value that indicates if *Units* can capture this *Building* or not.

- Catalogue: The list of *BaseUnits* that can be recruited from this *Building*, always that the function of cost provided by the user tells that it is affordable.

Each *BaseUnit* is associated to a button that will be displayed when the *Building* is selected, having a configurable texture for both when they are affordable and not affordable.
If the *Building* doesn't have any element inside its catalogue, then it won't display any menu when selected.

Figure 71: Example of affordable and unaffordable buttons for recruiting a Unit

- Sprite: The graphic representation of the *Building* in the map.

Building

As for *Units*, in addition to the attributes that come from their inheritance from *BuildingElement*, they are also *Controllable Elements* and have an id, an assigned *Team* and a position in the map. They also contain a direct reference to their *BaseBuilding* template, which tells their type and grants access to their original attributes.
At the beginning of each turn the player can set their actions (like providing resources to the *Team*). Although *Buildings* cannot move nor deal *Abilities*, when they have a *Catalogue* they can recruit those *Units* that are affordable by the *Team*. As *Units*, they contain *ActionBuildings* that tell if they perform any recruitment at the current turn. These are used by the Artificial Intelligence and can also be used by the user to know exactly what are the actions the player has selected (this can be especially interesting, as an example, for sending data to other devices in a multiplayer game).

BaseBuilding

They define the original template of the *Building* instances and, in some sense, their type. To be identified within the system, they have a unique name that is defined by the user when he or she creates them.
They only have one additional parameter:

- Sprites Teams: The same map from team name to *Sprite* that defines what is the Sprite to be displayed for it taking into account its *Team*.
  One example would be the following, in which we have two *Teams* that fight each other and where we want each one of them to have a different color code (one red and the other green):

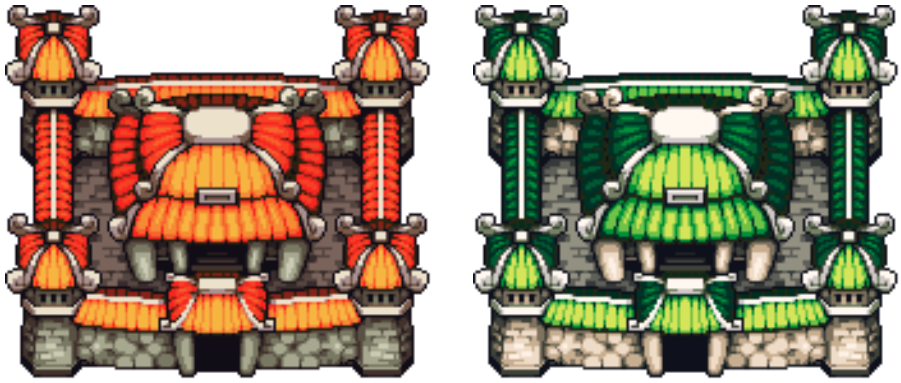

Figure 72: Example of two different sprites associated to different Teams for the same BaseBuilding

Implementation

Their implementation for handling the *BaseBuilding* templates is almost the same as for *Units*; and the only difference lies in the copy constructor, which in this case copies the *Building* specific parameters:

```
BuildingElement::BuildingElement(const BuildingElement &obj)
{
   setCaptureEnabled(obj.getCaptureEnabled());
   setCatalogue(obj.getCatalogue());
   if (obj.getSprite() != NULL)
   {
      sprite = Utils::copySprite(obj.getSprite());
```

```
    }
    else
    {
        sprite = NULL;
    }
}
```

Also, a new class has been defined for each *BaseUnit* that they can recruit. This class is called *CatalogueElement*:

```
Class CatalogueElement
{
    public:

        std::string button_sprite_affordable;
        std::string button_sprite_affordable_pressed;
        std::string button_sprite_unaffordable;
        std::string button_sprite_unaffordable_pressed;
}
```

Finally, each *BuildingElement* then contains a map of *CatalogueElements*:

```
std::map<std::string, CatalogueElement> catalogue; //string = BaseUnit name
```

5.4.2.4 Abilities

They are the possible actions that *Units* can take along with their movement. The player, when selecting an *Ability* with a *Unit*, must set which is the target position from the available range. They can affect both the dealer *Unit* as well as other *Units* and *Terrains*.
Each *BaseUnit* contains a map of *Abilities*, which are identified by their name.
An *Ability* contains the following attributes:

- Min Range: The minimum distance at which the *Ability* can be performed.
- Max Range: The maximum distance at which the *Ability* can be performed.

- On Allies: A Boolean value that tells if the *Ability* can be performed with a *Terrain* occupied by allied *Units* as target (an example would be a healing *Ability* to help allies in battle).
- On Enemies: A Boolean value that tells if the *Ability* can be performed with a *Terrain* occupied by enemy *Units* as target.
- On Empty Terrains: A Boolean value that tells if the *Ability* can be performed on a *Terrain* with no *Units* on it.
- Button Sprite and Button Pressed Sprite: The name of the *Sprites* to be applied over the *ButtonAbility* associated to the *Ability*.

An *Ability* is essentially defined by its *dealAbility* function, which defines what are the effects of the *Ability's* execution. This function is totally configured by the user, who can decide all the details of its implementation. The system ensures that it receives the following parameters:

- Dealer Unit: The *Unit* that is dealing the *Ability*.
- Receiver Unit: The *Unit* that is the main target of the *Ability* (that who occupies the target position, if there is any).
- Dealer Terrain: The *Terrain* at which the Dealer Unit is located.
- Receiver Terrain: The *Terrain* located at the selected target position.

Users can then define what are the effects for any of the *Game Elements* and, in case they want to affect any other *Element*, they can also use the *Strategy2D* module to gain access to any of them.

Also, since *Abilities* are configurable per *Unit*, this allows them to learn new ones. This way the user can implement RPG mechanics such as leveling up in which *Units* get new *Abilities*.

5.4.3 ControllerGame

The main class of the Domain Layer, it directly contains the instances of all the elements defined in the previous section.

In order to be able to access any of them at any time, they are structured in map containers that use the unique identifiers (or names) as primary key. These containers are implemented as templates that can store any type of elements (one for those Elements that use a primary key of number type and one for those that use one of type *string*).
This is the implementation for the *string* primary key container (the implementation is almost the same for both):

```
Template<class T>

class ContainerLongId
{
private:

   std::map<std::string,T> elements;

public:

   bool addElement(T element)
   {
      bool success = false;
      if (!elementExists(element->getName())
      {
         success = true;
         elements[element->getName()] = element;
      }
   }

   T getElement(const std::string& name_element)
   {
      T element = NULL;
      if (elementExists(name_element))
      {
         element = elements[name_element];
      }
      return element;
   }
```

```
   bool elementExists(const std::string& name_element)
   {
      typename map<string,T>::iterator it_exists = elements.find(name_element);
      return (it_exists != elements.end());
   }

   bool removeElement(const std::string& name_element)
   {
      elements.erase(name_element);
      return true;
   }

   std::map<std::string,T>& getElements()
   {
      return elements;
   }

   long size()
   {
      return elements.size();
   }

};
```

So *ControllerGame* contains the following definitions:

```
ContainerStringId <BaseUnit*> base_units;
ContainerLongId <Unit*> units;

ContainerStringId <BaseBuilding*> base_buildings;
ContainerLongId <Building*> buildings;

ContainerStringId <Team*> teams;
```

It also provides all the necessary interfaces for modifying the state of the system. Normally, these functions require the passing of specific parameters but also of the identifiers of the involved *Game Elements* to be able to obtain their instances. All the interactions between the *ActionManager* and the domain are performed through this class' interfaces, but it is also available for its usage by the user in any of the callback definitions.

For removing *Controllable Elements* from the system, the user can either remove them directly by removing their instance or, if she prefers it, remove them

whenever their blocking actions are finished. For this, when the user calls the *kill* method in a *Unit* or *Building*, it adds this element to a list of future removals and, in every update, it checks whether their blocking cocos2d-x *Actions* have finished. If this is the case, then that is the time for removing them from the system.
This is especially useful for synchronizing the removal of *Units* with the end of the attacking animation.

Finally, it also handles the turn passing, updating the *current_team* variable, which can be consulted and modified by the user at any time (so that he or she can control how turns are handled).
Also, for AI-controlled *Teams* it ensures that their actions are performed in the right order and synchronizes their movements (one action cannot start until all the blocking cocos2d-x *Actions* of the previous one have been finished).

5.4.4 Path Finding

The term Path Finding refers to the search of the shortest route between two points in a map. It represents one of the most important computing problems due to its usage in artificial intelligence.
It is also one of the most important mechanics in any strategy video game and especially on those that are turn-based.

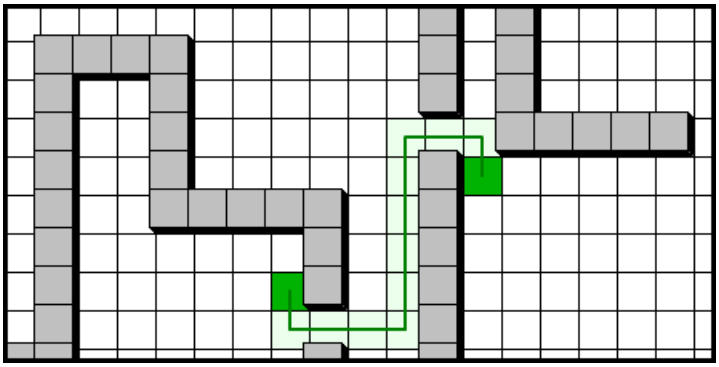

Figure 73: Path Finding concept

It has an entire field of research in computing due to the importance it can have in lots of contexts and to the effects it can have in the efficiency of a program. Most of this research is based on graph theory and especially on the *Breadth-First Search* (BFS) and *Depth-First Search* (DFS) search algorithms.

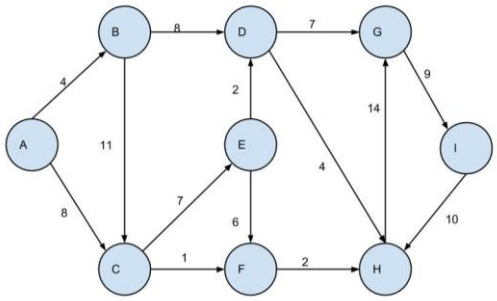

Figure 74: A weighted graph, used to represent Path Finding problems

5.4.4.1 Dijkstra's Algorithm

One of the most well known algorithms to be applied in path finding is *Dijkstra's Algorithm*, which can solve the shortest past problem for a graph without any negative edge path costs.

It starts at the initial node, which in our context represents the starting position, and computes the cost of moving from that node to any of its neighbors (those connected to it by an edge). The costs for every accessible node are stored and afterwards we visit that with the minimum cost. The procedure is then repeated for this node and for all the following using the same scheme, only now setting the cost of the neighbor nodes as the sum of the cost of the current one and their actual cost. All this taking into account that only previously unseen nodes can be added and storing all of them within the same pool, in order to get the global minimum cost.

A typical approach for computing the algorithm is the use of a priority queue that automatically sorts the nodes in terms of their cost of visiting.

5.4.4.2 Path Finding within the engine

In the context of a turn-based strategy game, however, the approach is a little bit different from usual, since what must be searched is not only the path from one position to another, but the path for all the positions that can be reached within a single turn, taking into account the Unit's movement range.

Figure 75: A picture of X-Com: Enemy Within, a turn-based strategy video game in which path finding plays a major role

For the implementation, we define the *PositionPath* class, which contains the x and y coordinates along with the associated cost of movement.
We also define its < comparison operator in order to be able to sort them inside of a priority queue:

```
Class PositionPath
{
public:
   long x;
   long y;
   double cost;

   PositionPath()
   : x(INVALID_POS)
   , y(INVALID_POS)
   , cost(-1)
   {
   }
```

```
   PositionPath(const long& x_in, const long& y_in, const double& cost_in)
   : x(x_in)
   , y(y_in)
   , cost(cost_in)
   {
   }

   bool operator <(const PositionPath& pos2) const
   {
      //flipped = lower cost better
      return (cost > pos2.cost);
   }
};
```

Once we have defined this class, we have the actual path finding function, which returns the reachable positions both as a matrix of *PositionPath* that will tell what is the path from any reachable position to the selected *Unit* and will have invalid values for unreachable positions and as a plain list that will directly tell which they are.

```
bool MapGame::pathFinding(Unit* unit, Team* team_unit,
          vector<vector<PositionPath> > &reachable, list<PositionPath> &list_reachable)
{
   //at start all positions are unreachable (see PositionPath default constructor)
   reachable = vector<vector<PositionPath> > (rows, vector<PositionPath> (columns));

   PositionPath pos (unit->getPosX(), unit->getPosY(), 0);
   reachable[pos.y][pos.x] = PositionPath(pos);
   vector<pair<long, long> > possible_movements = unit->getBaseUnit()->
                                                        getPossibleMovements();
   list_reachable.push_back(pos);

   priority_queue<PositionPath> queue;
   queue.push(pos);

   while (!queue.empty())
   {
      PositionPath current_pos = queue.top();
      queue.pop();

      for(int I = 0; i < possible_movements.size(); ++i)
      {
         //we evaluate all the candidate positions available for the Unit
         Position candidate_pos (current_pos.x + possible_movements[i].first,
                                 current_pos.y + possible_movements[i].second);
```

```
        if(validPosition(candidate_pos) && reachable[candidate_pos.y][candidate_pos.x]
           == -1 && !occupied(candidate_pos))
        {
          Terrain* candidate_terrain = getTerrain(candidate_pos);
          double cost;
          Terrain* prev_terrain= getTerrain(Position(current_pos.x,current_pos.y);

          CCASSRT((ref != NULL && moveTo != NULL), "You must define a movement
          Function,use Srategy2D::setMovementFunction(cocos2d::Ref*,
          FUNCTION_MOVEMENT(bool moveTo(Unit*, Terrain*, Terrain*, double&))");

          bool can_move = (ref->*moveTo)(unit, candidate_terrain, prev_terrain, cost);
          double total_cost = cost + current_pos.cost;

          if (can_move && unit->getMovementPoints() >= total_cost)
          {
            PositionPath pos_path(candidate_pos.x, candidate_pox.y, total_cost);
            PositionPath pos_path_from(current_pos.x, current_pos.y, total_cost);
            reachable[candidate_pos.y][candidate_pos.x] = pos_path_from;

            queue.push(pos_path);
            if (display_over_units || getUnit(candidate_pos) == -1)
            {
            //add pos only if its not occupied (even if the unit can move through it)
              list_reachable.push_back(pos_path);
            }
          }
        }
      }
   }
   return true;
}
```

In this implementation of the algorithm, we start with an empty priority queue that stores the accessible positions sorted in terms of their cost (with a minor cost representing a higher priority). At the beginning, we insert the initial position into the queue and start a loop that will only end when the queue is empty.

Each iteration, we set the position at the top of the queue as the current position and remove it from the queue.

Then we check which of the positions that are in range from this current position are reachable by the unit using the cost function (which computes the cost of moving from one position to another taking into account both the terrain and the *Unit*), the possible movements of the *Unit* (defined in its *BaseUnit*, as explained in the *Unit* section) and the movement parameters (which tell if a *Unit* can move

through positions already occupied either by *Units* of the same team or enemy units).

Those positions whose cost added to the cost to reach the current position is lower than the *Unit's* action points will be added to the queue and sorted with that total cost. Also, we will store that to reach that new position, the shortest path comes from the current position, this will ensure that later we will already have the shortest path for every reachable position, since each one will contain a reference to the position from which the shortest path comes except for the starting position, which will have a token value to indicate that no other movement is required.

Finally, only positions that are not occupied will be displayed unless the user has set the *display_over_units* options, in which case they will be shown as reachable but the player won't be able to move the Unit on to them.

The algorithm will only stop when the unit can't reach any new positions with the cost function provided by the user.

An example would be the following, in which we have this map situation:

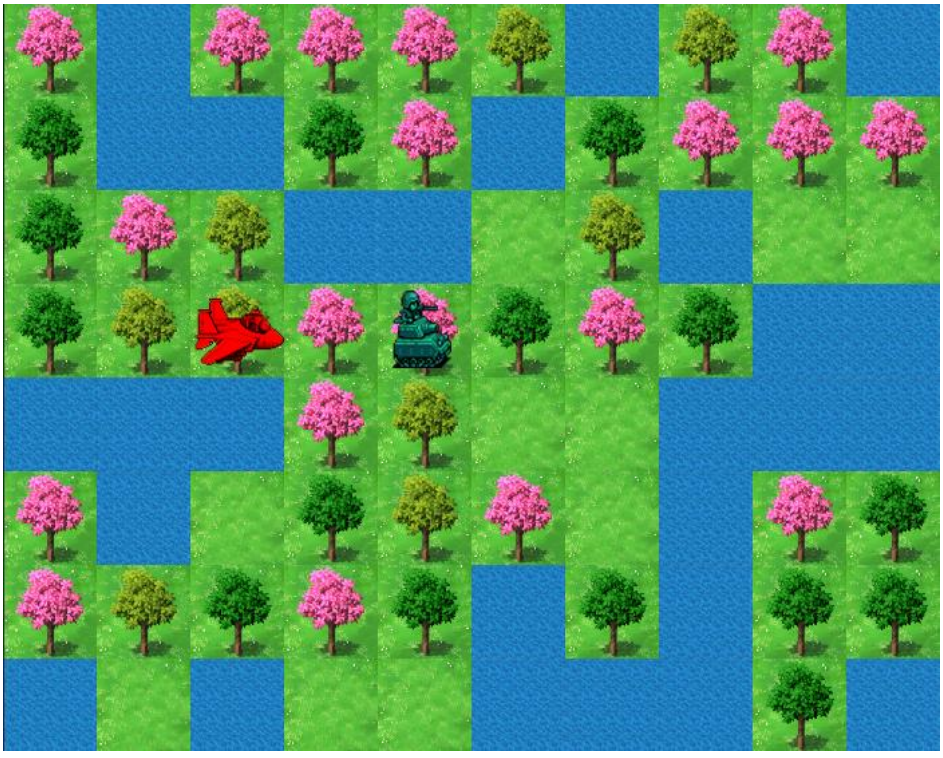

Figure 76: Path Finding Map layout

And where the tank *Unit* has been selected. It cannot move through water *Terrains* or positions already taken by enemy units (as is the plane). Also, the *Unit* has been set so that it can only make four possible moves from a position: up, down, left or right.

The tank has been given a total of 40 movement points per turn, and the cost function, for the case of a *Unit* of this type that wants to enter a grass *Terrain* sets the cost at 10 points. For the case of water *Terrains* it simply doesn't allow the movement.

For the following example we will arrange the map's positions by indicating its column and its row, following the format [column, row].

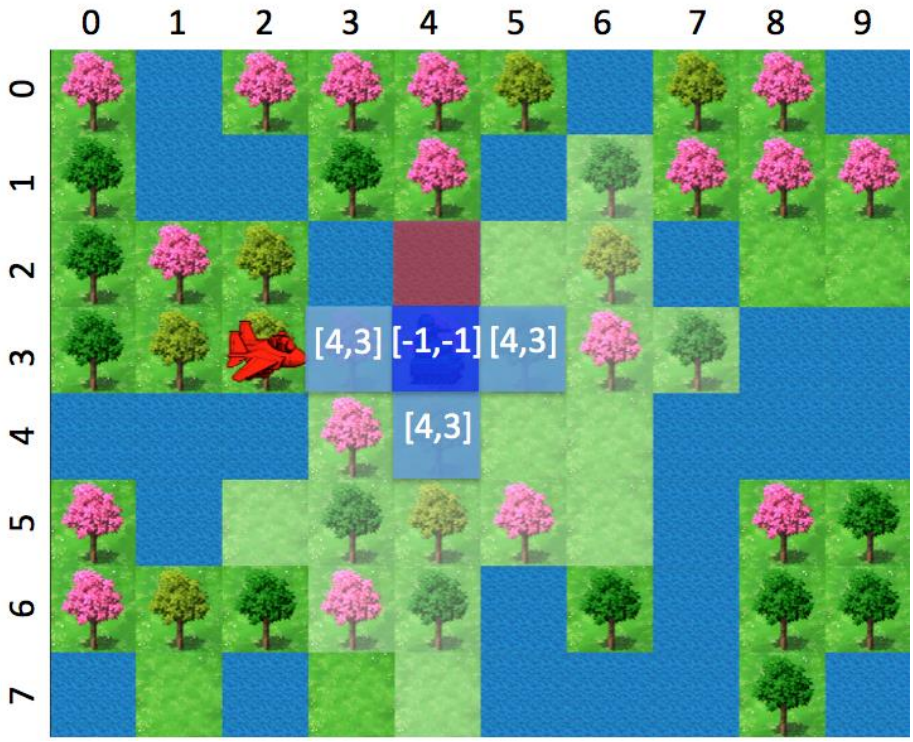

Figure 77: Path Finding 1

We start at the selected *Unit's* position, which is [4,3] and set for it the token value [-1,-1] to indicate that this is the starting position itself. The current position is also marked in the image with dark blue.

The *Unit*, for now, has all its 40 movement points. Following the scheme of the neighborhood vector (the possible movements of the *Unit*), we try to expand to

the positions that are left, right, above or below of the current one. For each one we check the cost function, which in case of the up position (marked in red) doesn't allow the movement and for the rest of them sets a cost of 10 (and so they are added to the queue, as the light blue color over them indicates).

For the three possible movements we store that the position from which the shortest path comes from is the current one, [4,3].

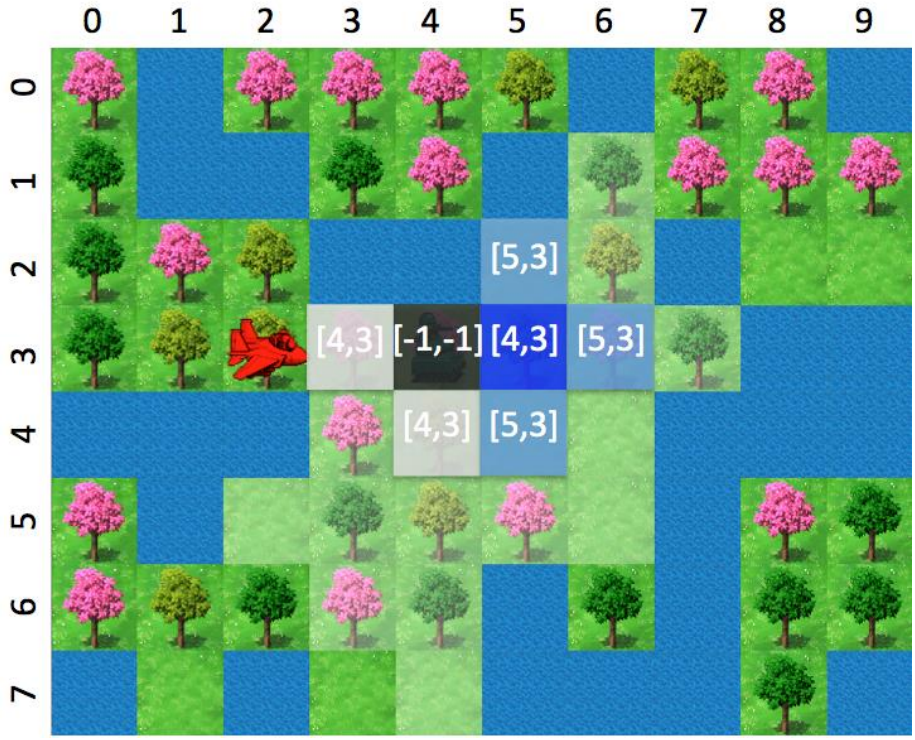

Figure 78: Path Finding 2

We then remove the previous position from the priority queue (marked in black) and move to the next one, which is [5,3] (though the three of them have equal cost). The positions that now are in the priority queue are marked in white.

We expand to its neighbors, with the exception of [4,3], which has already been added to the queue or visited and so is avoided. The cost returned by the function is also of 10, which added to the previous 10 from the current position gives us a total cost of 20 for the movement, a cost still affordable.

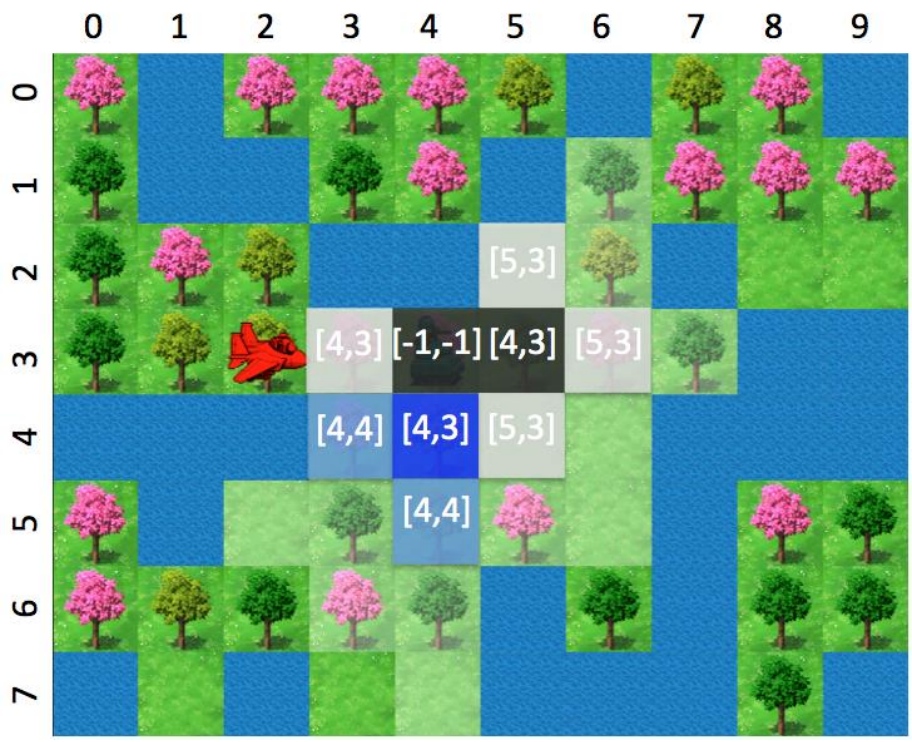

Figure 79: Path Finding 3

We also remove the [5,3] position from the queue (marked in black) and repeat the process for the current position [4,4], which doesn't add the [5,4] position because it is already in the queue nor [4,3] because it has been processed and adds the other two with a cost of 20.

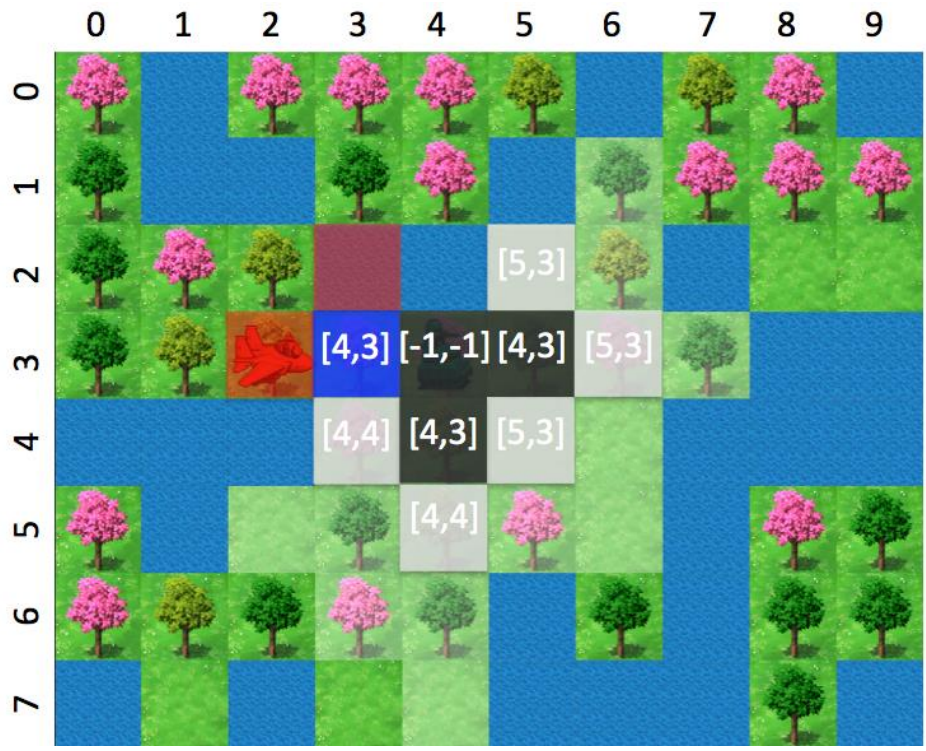

Figure 80: Path Finding 4

In this case, we cannot propagate any position, since the two possibilities that haven't been already covered cannot be reached. For the case of the one above, it is a water terrain and the function disallows the movement. For the case of the left one, the movement parameters are now configured to disable *Units* to move through positions occupied by enemies.

This same process is repeated until the priority queue is empty, which happens when no other positions can be added either because there are no neighbors or because the function returns a cost higher than the movement points of the unit. At the end, the available positions are marked with a special color so that the player knows who they are and can easily decide to which he will send the selected unit.

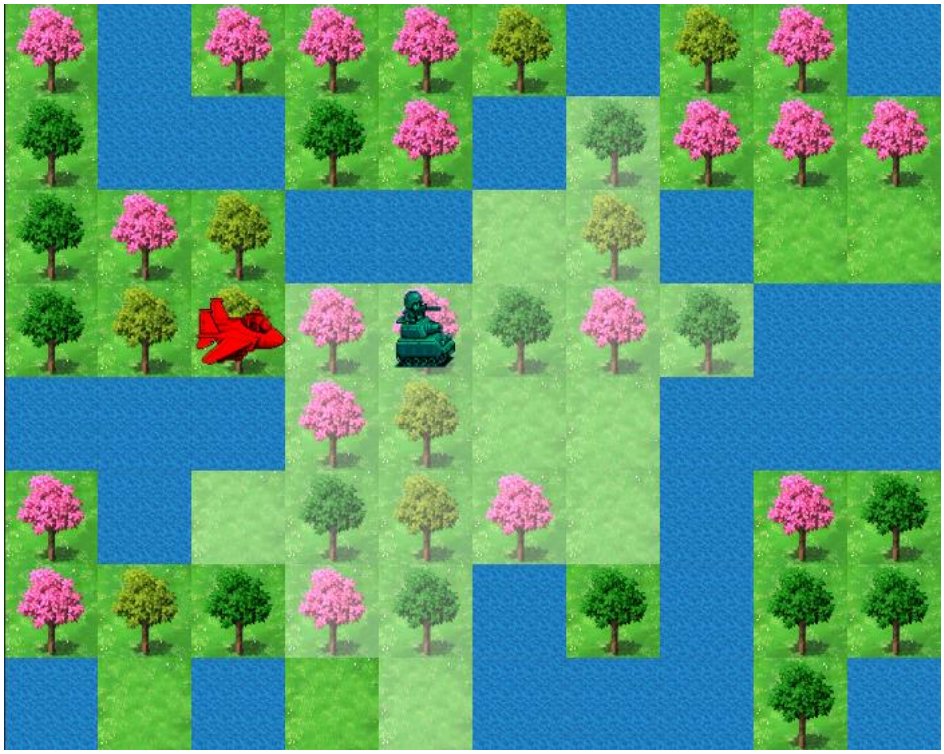

Figure 81: Path Finding available positions

In this case, are marked with a semi-transparent white layer, although the user can select the color that better fits him.

Internally, as explained, the system stores for each position the coordinates of the position from which the shortest path comes, except for the initial one, which stores the invalid value that serves as token.

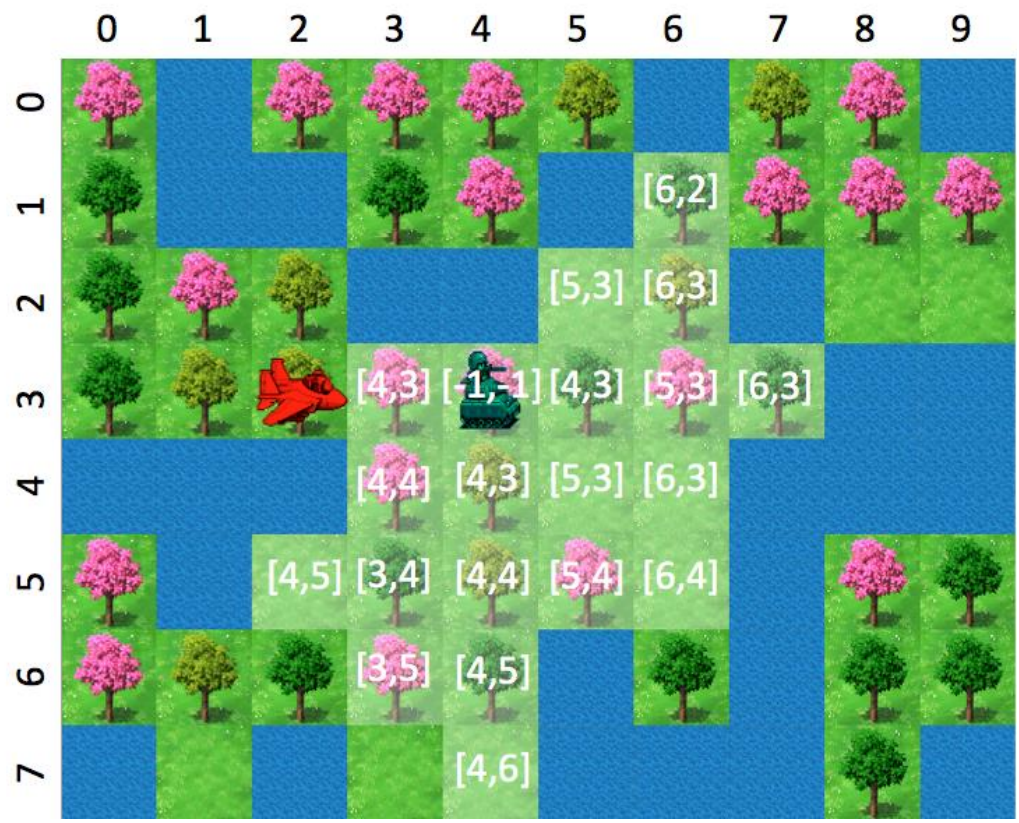

Figure 82: Shortest path from any of the positions in range to the Unit

Then, when the player selects a position for the *Unit* to move to, the system just has to loop looking into the positions for the previous one until it reaches the token. Inverting the order of the visited positions provides the shortest path for the unit.

For example, if the player selected the position at the [6,5] coordinates:

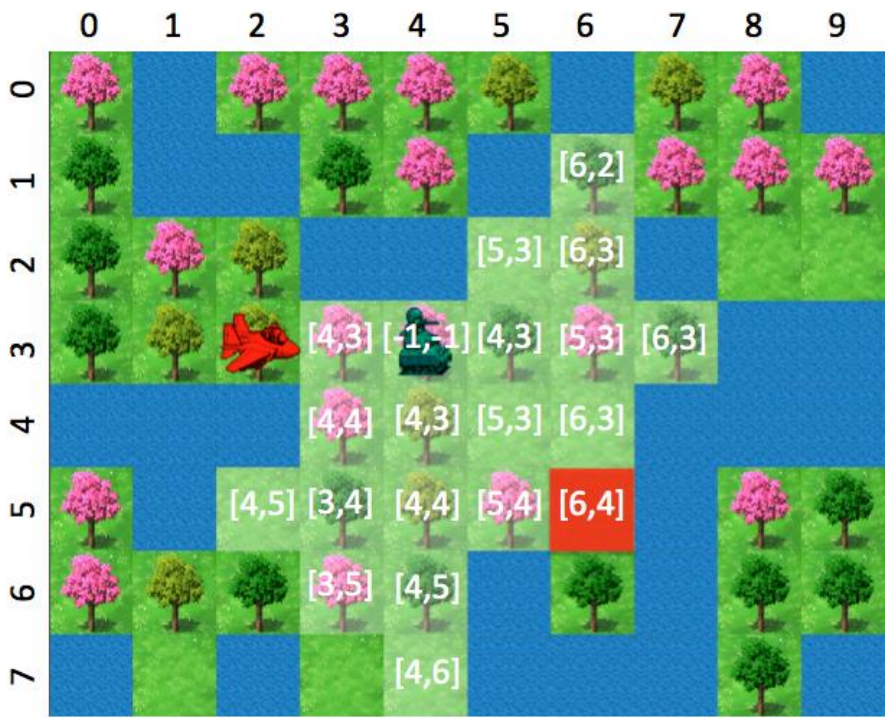

Figure 83: the User selects a position in range

The value stored at the [6,5] position (marked in red) tells us that the shortest path comes from the [6,4] position.

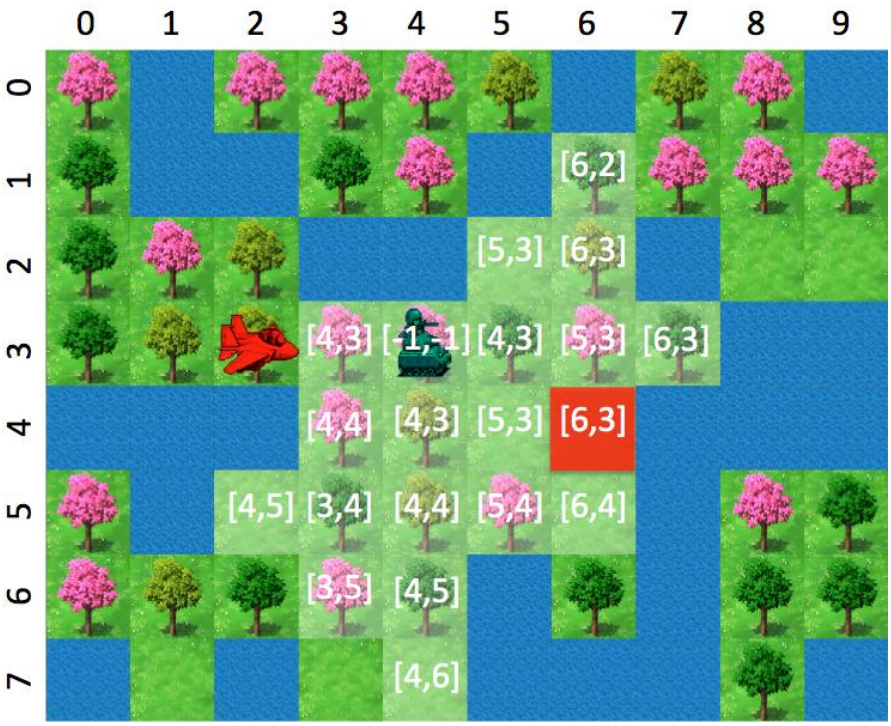

Figure 84: The algorithm moves through the positions following the shortest path storing the values

For the [6,4] position it tells us that it comes from the [6,3] position. If we repeat the process we keep getting the positions that form the shortest path, until we get the [-1,-1] token position.

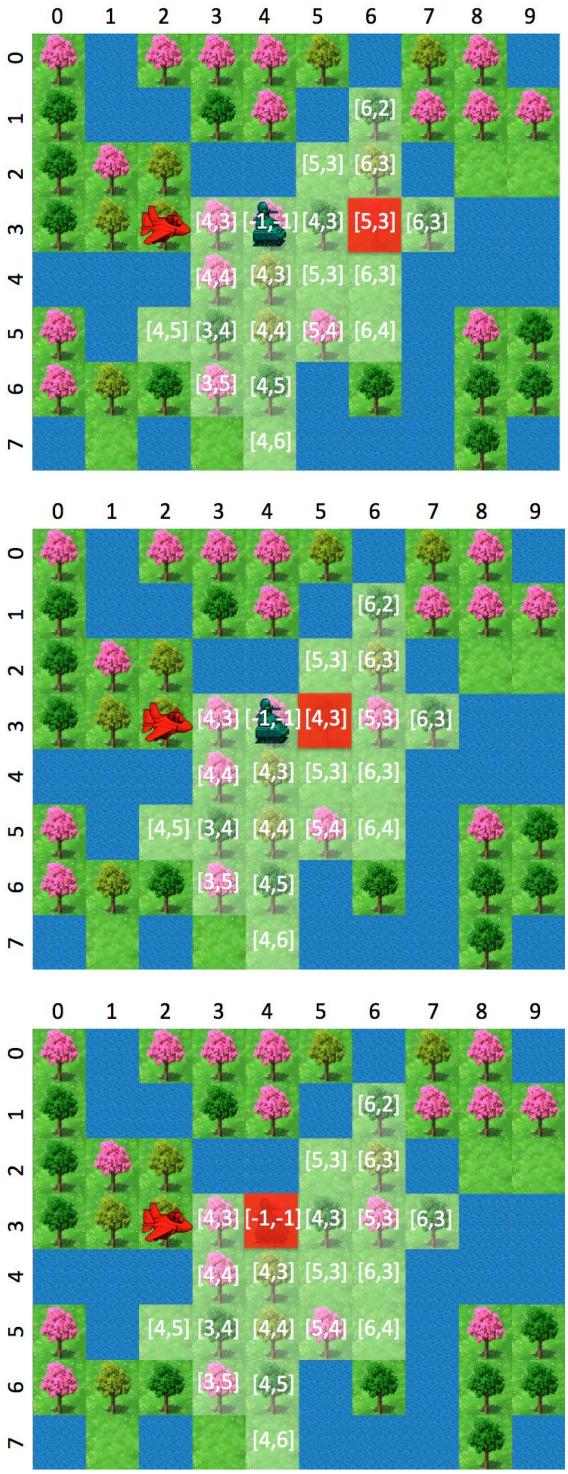

Figure 85: The algorithm completes the path back to the selected Unit

When we reach the invalid position we stop the procedure. At the end we have visited the following positions: [6,5], [6,4], [6,3], [5,3], [4,3].

Since we know that the last found is the original position of the unit, we remove it from the array and invert the vector. Then we have that the path the unit must follow to reach its objective is: [5,3], [6,3], [6,4], [6,5].

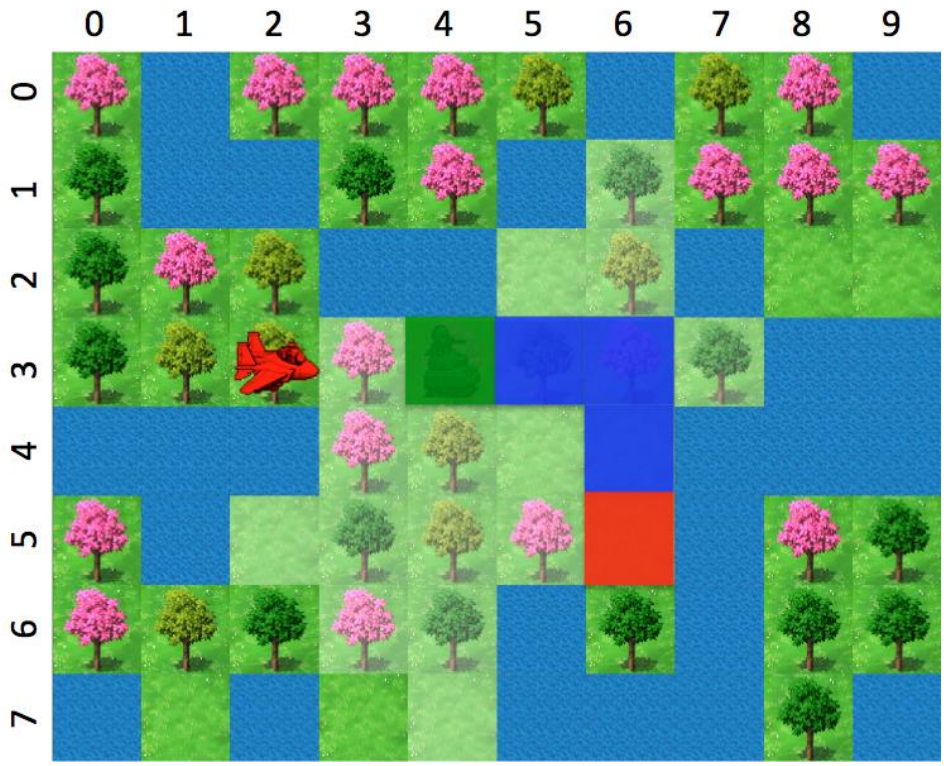

Figure 86: Shortest path from the Unit to the selected position

Finally, the *Unit* must move through the defined path until it reaches its destination. For this process, we take the movement time per position defined by the user, and moves through them at the right speed.

Also, when a change in the Unit's horizontal orientation is detected, the *Unit* is flipped over the horizontal axis to make it face the right direction of the movement.

5.4.5 Callbacks

As explained in the previous sections, many of the configurations of the engine rely on the user's implementation and setting of some callback functions that will tell the system what are the acts that must take place in any situation.
There are a lot of callbacks to be defined; some of them are mandatory in order to have some features in the game while others are just defined to help the user in key moments of the gameplay.
These are the different callbacks that the user can define for the game:

- **Movement function:**

It determines whether a *Unit* can move into a position or not. In case it can, then it also provides with the cost of such movement. The user must define it order to allow Units to move around the map.
As explained, it is used in the Path Finding function to determine which positions the selected *Unit* can reach and to determine the shortest path to any of them.
It returns a Boolean value that tells if the movement is possible and takes the following parameters:

- Unit: The *Unit* for which the movement is being checked.
- Terrain: The *Terrain* of the target position.
- Previous Terrain: The *Terrain* from which the shortest path comes.
- Cost: An output value for the cost of the movement.

And the following definition:

```
bool movementFunction(Unit* unit, Terrain* terrain, Terrain* previous_terrain,
                      double& cost);
```

- **Deal Ability function:**

It determines the effects that an *Ability* has when performed. In order to allow the player deal these *Abilities*, the user must have defined its associated callback.
It returns a Boolean value that tells if the *Ability* has been successfully performed, if the result is "false" then the system brings back the *Ability* selection menu.

It takes the following parameters:

- Dealer: The *Unit* who is dealing the *Ability*.
- Receiver: The *Unit* at the position targeted by the *Ability* (if there is no Unit, then this a null reference).
- Terrain Dealer: The *Terrain* at which the dealer *Unit* stands, this can be useful in order to have terrain bonus for the *Units* (a *Unit* who is in an elevated terrain has advantage over one that is not, for example).
- Terrain Receiver: The *Terrain* at the targeted position.

In case the user wants to have more information about the positions then he or she has the possibility of getting the *Strategy2D* instance and asking for more information through *ControllerGame*.

The function has the following definition:

```
bool dealAbility(Unit* dealer, Unit* receiver, Terrain* terrain_dealer,
                 Terrain* terrain_receiver);
```

- **Turn Start Function:**

It determines the actions to be performed when a *Team* starts a turn, and the engine calls it every time a new *Team* starts its turn. By default it does nothing, so there's no need for the user to define it unless he or she wants to do anything special at that moment.

It is the most convenient function for defining the Artificial Intelligence's decisions by setting the desired actions of the turn for the *Units* and *Buildings* of the *Team*.

It only takes the *Team* that is starting the turn as parameter and has the following definition:

```
void turnStartFunction(Team* team);
```

- **Turn End Function:**

It determines the actions to be performed whenever a *Team* ends its turn. It also does nothing by default.

In case the user wants to develop an online multiplayer game, it is a good place to send any other players what are the actions that the *Team* has selected for the current turn, since all Units and Buildings have their associated ActionTurns (whether they are controlled by an AI or by a player).

It has the same definition as Turn Start:

```
void TurnEndFunction(Team* team);
```

- **Unit Selected Function**:

It is called whenever a player selects a *Unit*. It determines what happens in this case and return a Boolean value that tells if the player can select the Unit.
By default nothing is done and it returns true, but the player can set any code that he or she wants.

It should be used for telling the system if the *Unit* can be selected taking into account any kind of parameter (Some examples might be checking if the *Team* of the *Unit* is not controlled by the AI, the *Team* of the *Unit* is the one that is

currently moving or a more complex one in which we check at the attributes of the Unit to see if it is not paralyzed for this turn).

A typical usage for this (seen in other games of the genre) would be to change the brightness of the *Sprite* of the *Unit* to make it more visible and to start an animation.
It only takes the *Unit* itself as parameter and has the following definition:

```
bool unitSelectedFunction(Unit* unit);
```

- **Unit Deselected Function:**

It is called whenever the player deselects a *Unit*, whether because he or she has cancelled the selection or because the action has been performed. Its original conception is to provide a function in which undo the actions taken in the selection function (for example, putting back the normal color of the *Sprite* or setting its normal animation).
It has the following definition:

```
void unitDeselectedFunction(Unit* unit);
```

- **Unit Action Ended Function:**

It is called when a *Unit* has successfully performed an action in the turn (a movement and an Ability). In case of online multiplayer, it is probably the best place to send what the performed action was, if the user wants to synchronize every action of the *Controllable Elements* individually and not all the *Team* members' at the same time (otherwise Turn End would be a better place).

It has the following definition:

```
void unitActionEndedFunction(Unit* unit);
```

- **Building Selected Function:**

It is the equivalent to the Unit Selected Function but for *Units*, with the same purposes and a very similar definition:

```
bool buildingSelectedFunction(Building* building);
```

- **Building Deselected Function:**

It is the same as for *Units*, with this definition:

```
void buildingDeselectedFunction(Building* building);
```

- **Building Action Ended Function:**

The same as for *Units*, only this time for a Building:

```
void buildingActionEnded(Building* building);
```

- **Unit Added Function:**

This function is called whenever a new *Unit* is added to the map. By default the function does nothing, so it is up to the user to configure it as he or she pleases.

Originally, I added it because Cocos2d-x had a limitation regarding animations:
Although *Sprites* can be cloned by recursively searching for the textures of the *Sprite* (which is done by the *copySprite* function in *Utils* when setting the *Sprite* for a *Unit* based on the *Sprite* of its *BaseUnit*), there is no way to obtain all the Cocos2d-x *Actions* of the given sprite without knowing their tags. This made it impossible to set idle animations for the *Unit's* sprites.
After thinking of ways in which the player would tell the system the tags for the animations of the *Unit*, I finally decided to create this function in which he or she could do anything with the *Sprite* when the *Unit* was added.

Also, this way the user is given more control over how *Units* are added to the map, allowing him or her to do things like setting special entry animations or tweaking some of their attributes depending on their position or *Team*.

Finally, I made the decision of also providing the user with the *Building* that was recruiting the *Unit* (only in case the *Unit* was added by recruiting it from the Building). This allows to set specific features for the *Unit* regarding if it has been recruited from a specific *Building*. One example would be to set it as "already moved" when recruited to avoid the player from moving them in the same turn they are added.

It has the following definition:

```
void unitAddedFunction(Unit* unit, Building* building);
```

- **Building Added Function:**

This is the equivalent for *Buildings* of the Unit Added function. In this case, since a *Building* cannot be recruited, it only takes the *Building* itself as parameter, in contraposition to the one for *Units*:

```
void buildingAddedFunction(Building* building);
```

- **Team Can Buy Function:**

This function determines, for *Team* and a *BaseUnit*, if the given *Team* can recruit a *Unit* based on the provided base (by returning a Boolean value). Here the user can check for any attribute both of the *Team* and the *BaseUnit*.

An example would be to check for a "price" attribute inside the *BaseUnit* and compare it to a "funds" attribute inside the *Team*. If the price is lower than the funds, then the *Team* can buy it.

It has the following definition:

```
bool buildingCanBuyFunction(Team* team, BaseUnit* base_unit);
```

- **Team Buy Function:**

Here the user should set what are the effects for a *Team* who buys a *Unit*. For the previous example, the user should subtract from the *Team's* funds the *BaseUnit's* price. It also returns a value just in case anything went wrong and has the following definition:

```
bool buildingBuyFunction(Team* team, BaseUnit* base_unit);
```

- **Update Function:**

This function is called every time a new frame is rendered. It allows the user to make any checks and perform any actions without having to wait for certain actions to take place.

An example for this would an asynchronous Internet connection in which he or she needs to check whether the game has received a new message or not.

It has the following definition:

```
void updateFunction();
```

5.4.5.2 Implementation

To implement these callback settings, I have developed an API that allows the user to pass functions as parameters for the engine, so that it can store them and call them at the adequate time.

Also, since it is very useful to store a state over the course of the game and even be able to consult any user-defined variable, it also allows these functions to be static and non-static members of a class. For the case of non-static functions, the user should pass an object instance to call the method.
A condition for this object passing is that it inherits from the cocos2d *Ref* class, according to the system of callbacks of cocos2d-x.

For doing this, the engine contains function variables as well as object instances to call them; an example would be the following (for the Unit Added function, which, as most functions, can be found in *ControllerGame*):

```
cocos2d::Ref* ref_functionUnitAdded;
FUNCTION_UNIT_ADDED functionUnitAdded;
```

To call these functions, it provides wrappers that execute them. These wrappers must take as arguments the actual parameters to be passed to the function:

```
void executeFunctionUnitAdded(Unit* new_unit, Building* building)
{
   (ref_functionUnitAdded->*functionUnitAdded) (new_unit, building);
}
```

For this particular case, for example, *ControllerGame* ensures that it is called every time a new *Unit* is added to the game, either passing the *Building* when the *Unit* is added because of a recruitment, or passing a NULL value when it's not.

In order to set them, there are setter functions that take both the caller object and the function as parameter:

```
void setFunctionUnitAdded(cocos2d::Ref* ref, FUNCTION_UNI_ADDED newFunction)
{
   ref_functionUnitAdded = ref;
   functionUnitAdded = newFunction;
}
```

5.4.5.3 Usage

In order to set a function as callback, the user must use the defined setter functions available at the *Strategy2D* class. As explained before, the object that contains the function as member must be a cocos2d *Ref* subclass.

To help making the casts (which would take not only the return type and parameter types, but also the type of the class that has it as member), there are type definitions in *Utils* for each possible callback function:

```
typedef (cocos2d::Ref::*FUNCTION_MOVEMENT) (Unit* unit, Terrain* terrain,
                                            Terrain* prev_terrain, double &cost);

typedef (cocos2d::Ref::*FUNCTION_CAN_BUY) (Team* team, BaseUnit* base_unit);

typedef (cocos2d::Ref::*FUNCTION_BUY) (Team* team, BaseUnit* base_unit);

...
```

For setting the function, the user should pass both the object and the function as parameter, as follows:

```
Strategy->setMovementFunction(this, FUNCTION_MOVEMENT(&MyClass::movement));
```

Which should be defined as a member function in the class:

```
bool MyClass::movement(Unit* unit,Terrain* terrain,Terrain* prev_terrain,double& cost);
```

As long as the function does not use any internal attribute of the object class, the user can also pass a NULL value for the object. If they are used, however, then there are risks of having problems of memory corruption.

5.4.6 Artificial Intelligence

Strategy2D allows the user to configure the Artificial Intelligence for any AI Controlled *Team* of the game in any of the defined callbacks, adapting the game to any kind of circumstance. In order to do this, he or she must take advantage of the many consulting and setting functions defined in *ControllerGame*, which can be accessed from the *Strategy2D* instance at any point of the execution.
Then, by using the *addActionTurn* function (both for *Units* and *Buildings*), the system will be able to know what are the desired actions for this *Controllable Elements* to be performed at the current turn (These actions are cleared up when a new *Team* takes control of the turn). *ControllerGame* will finally take care of the set actions and make the *Units* and *Buildings* perform theirs in the set order, synchronizing them by using their Blocking Actions (defined in the Unit and Building section) to know when their movements have come to an end.

5.4.7 Internet Connection and Multiplayer

It is also up to the player to configure how the game connects to the Internet. For this, he or she can use any of the callback functions (especially Turn Ended and Action Ended, as explained in the callbacks section), where there is freedom to use any kind of Internet connection library (Cocos2d-x itself provides with some network interfaces such as *HttpClient* or *SocketIO*).
To synchronize the movements between the different devices that would be playing the same online game, he or she can use the *ActionTurn* attributes that are stored for *ControllableElements*.
The user also has the possibility of configuring multiplayer games locally, using the same tablet (as if the tablet itself were the board) and is able to set any number of *Teams* either controlled by players or by the AI.

6. USAGE EXAMPLE

In order to see how the Engine is used to create a game and to show the array of possibilities that it offers to the user, there is no better way than developing a demo that puts its elements to test and explaining how every feature is configured.

In this example I will develop a game that will confront a two *Teams*, one controlled by a player and the other by the Artificial Intelligence, who will fight for controlling the map and wiping out the *Units* of their rivals.
It will have an economy system related to the ownership of Buildings and the recruitment of new *Units*.
In addition, there will be several *Unit* types, each with their own attributes and *Abilities*.

6.1 Starting

For starting, we will need to include the *Strategy2D* header file, which will grant us access to the *Strategy2D* module, with all its functions and object types.

```
#include “Strategy2D.h”
```

Once we include this file we can declare a *Strategy2D* instance and call the create method, in this case we will declare also two variables for the number of rows and columns of the map:

```
long num_rows = 20;
long num_columns = 20;
Strategy2D* strategy = Strategy2D::create(num_rows, num_columns);
```

In addition, we set the starting, maximum and minimum zooms in the game in terms of number of visible columns and we limit the view to the map:

```
strategy->setZoom(7, 2, 10); //starting, minimum and maximum number of visible columns
```

```
strategy->setLimitViewToMap(true);
```

And provide the engine with the path of the default *Wait* and *Cancel Sprites*:

```
strategy->setCancelSpritesNames("ui/button_cancel.png","ui/button_cancel_pressed.png");
Strategy->setWaitSpritesNames("ui/button_wait.png","ui/button_wait_pressed.png");
```

6.2 Terrain Configuration

We will need to configure the map and provide the engine with the required sprites. For this we first must decide the types of *Terrains* there will be in the game:

- Grassland: A *Terrain* covered by grass and a few trees, all kinds of land and air *Units* will be able to move through it.

Figure 87: Sprite for grassland terrains

- Wood: A *Terrain* covered by extensions of trees, all kinds of land and air *Units* will be able to move through it but with a bigger cost than for Grassland.

Figure 88: Sprite for wood terrains

- Mountain: A *Terrain* with high elevations, only air *Units* will be able to move through it.

Figure 89: Sprite for mountain terrains

- Water: Both for seas and lakes. Air and water *Units* will be able to move trough them. In this particular case we will animate the sprite with a *Cocos2d-x SpriteFrame* using a composition of several images that will be displayed in an endless loop.

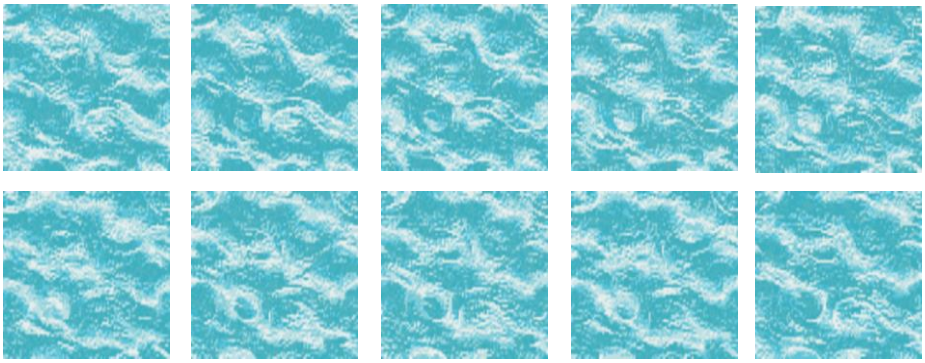

Figure 90: Sprites for water animation

Once we have the materials for setting up the graphics, we need to create the logic map that will represent its structure. For this we need to create two matrixes, one containing the *Sprite* for each tile and the other containing the definition of the *Terrains* located at every position.

In this case, we will generate a random map inside a function and call the *Strategy2D* setter:

```
vector<vector<Sprite*> > map_sprites(num_rows, vector<Sprite*> (num_columns));
vector<vector<Terrain> > terrains(num_rows, vector<Terrain> (num_columns));

generateMap(map_sprites, terrains);

tile_size = 900; //the width in pixels of each sprite

strategy->setTerrainsWithSpriteAssociation(terrains, map_sprites, tile_size);
```

In which the *generateMap* function creates a random map with the distribution of *Terrains*. For every position the probability of it being grassland *Terrain* is of 32.5%, wood *Terrain* also of 32.5%, mountain *Terrain* of a 10% and water *Terrain* of a 25%.

To configure we do the following for every case:

- Grassland:

```
Sprite* sprite = Sprite::create("terrains/grass.png");

Terrain terrain;
terrain.setAttribute("cost_soldier", 1);
terrain.setAttribute("cost_tank", 1);
```

```
terrain.setAttribute(“cost_plane”, 1);
terrain.setAttribute(“cost_ship”, -1);
```

- Wood:

```
Sprite* sprite = Sprite::create(“terrains/wood.png”);

Terrain terrain;
terrain.setAttribute(“cost_soldier”, 2);
terrain.setAttribute(“cost_tank”, 2);
terrain.setAttribute(“cost_plane”, 1);
terrain.setAttribute(“cost_ship”, -1);
```

- Mountain:

```
Sprite* sprite = Sprite::create(“terrains/mountain.png”);

Terrain terrain;
terrain.setAttribute(“cost_soldier”, -1);
terrain.setAttribute(“cost_tank”, -1);
terrain.setAttribute(“cost_plane”, 1); //only planes can move through them
terrain.setAttribute(“cost_ship”, -1);
```

- Water:

```
Sprite* sprite = Sprite::create(“terrains/water_anim/water_anim10.png”);

//we create the animation:

string name = “terrains/water_anim/water_anim”;

Vector<SpriteFrame*> animFrames(11);

for (int i = 0; I < animFrames.size(); ++i)
{
   std::stringstream ss;
   ss << i;

   Rect rect (0, 0, 900, 900); //the size of each sprite that composes the animation

   SpriteFrame* sf = SpriteFrame::create(name + ss.str(), rect);

   animFrames.pushBack(sf);
}
```

```
double time_lapse = 0.05f; //the lapse between each frame

Animate* anim = Animate::create(Animation::createWithSpriteFrames(sf, time_lapse);

//we make the sprite run the animation in an infinite loop:

sprite->runAction(RepeatForever::create(anim);

//terrain configuration:

terrain.setAttribute(“cost_soldier”, -1);
terrain.setAttribute(“cost_tank”, -1);
terrain.setAttribute(“cost_plane”, 1);
terrain.setAttribute(“cost_ship”, 1); //only planes and ships can move through them
```

After building the map we have a result like the following (in real time the water is animated):

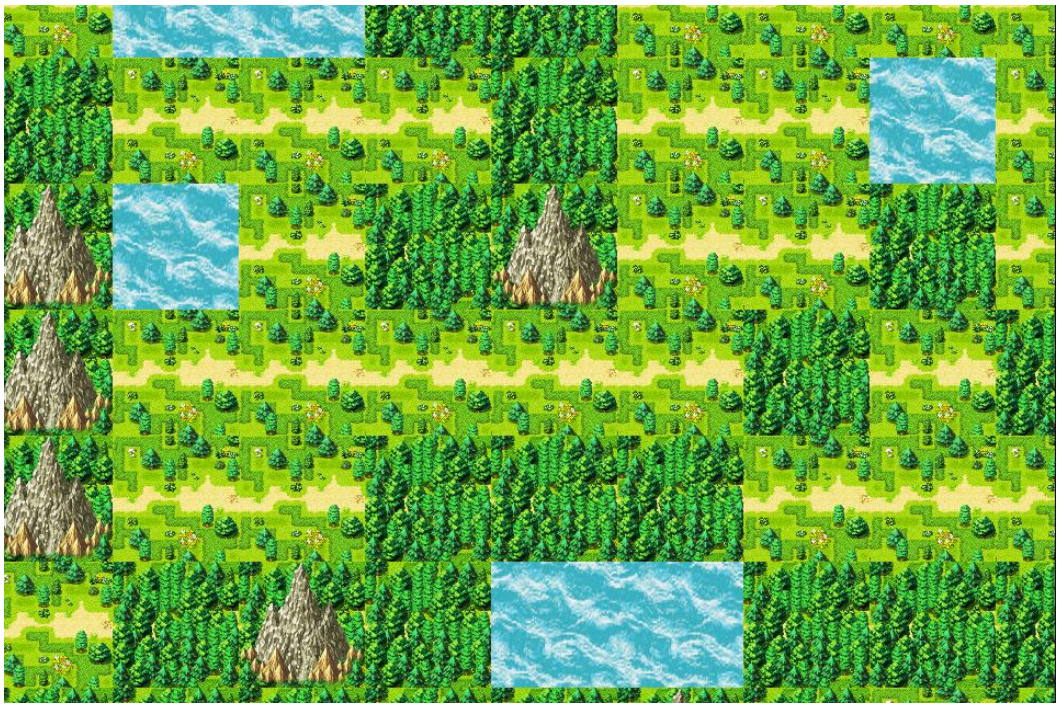

Figure 91: A map generated with the defined terrains

6.3 Team Configuration

Once we have configured the *Terrains*, we must add one or more teams to the game in order to have them battle and compete for victory.
In this case, we will configure two *Teams* represented by a different color. Each one will have two attributes named "funds" and "fuel" that will serve as resources for building new *Units* and making one of them controlled by the player and the other by the Artificial Intelligence.
We start with the user controlled *Team*, that will be represented by the red color. It will have 150 of funds and 50 of fuel and its layer for displaying the reachable positions both for movement and *Abilities* will have a semi-transparent red color:

```
Team team_red("red");
team_red.setAttribute("funds", 150);
team_red.setAttribute("fuel", 50);

Color4B color (255, 50, 50, 80); //RGBA
team_red.setColorMovementLayer(color);
team_red.setColorAbilityRangeLayer(color);

team_red.setAIControlled(false);
```

Later, we also configure the AI controlled *Team*, since it will not be controlled by a player, there is no need to set any color for the reachable positions. For parity reasons, we will give it the same resources.

```
Team team_blue("blue");
team_blue.setAttribute("funds", 150);
team_blue.setAttribute("fuel", 50);
team_blue.setAIControlled(true);
```

Finally, we add them to the system with the following calls (we must take into consideration that the order when we add them will determine the order of the turns):

```
strategy->addTeam(team_red);
strategy->addTeam(team_blue);
```

6.4 BaseUnit Configuration

In this chapter we will first explain which will be the types of *Units* available in the game and how we configure their attributes. Later, we will also explain what will be their *Abilities* and how to allow configure them so that Units can use them.

For every *Unit* we will configure these attributes:

- hp: Health Points, when they reach 0 the *Unit* disappears from the map.
- strength: The maximum amount of damage a *Unit* can perform with an attack.
- resistance: The maximum amount of damage that can be absorbed by a *Unit* without lowering their hp.

We will have the following types of *Units* in the game:

- Soldiers:

They are the most basic units in the game, being able to only move through land *Terrains* and having a low range of attack. Although they have much strength or resistance, they are the online *Units* that can capture or build *Buildings*.
They are configured using the following code:

```
BaseUnit base_soldier(“soldier”);
base_soldier.setAttribute(“hp”, 40);
base_soldier.setAttribute(“resistance”, 10);
base_soldier.setAttribute(“strength”, 20);
base_soldier.setAttribute(“cost_fuel”, 0);
base_soldier.setAttribute(“cost_funds”, 10);
base_soldier.setCanCapture(true);
base_soldier.setMovementPoints(4);

//we configure two sprites for the two teams
Sprite* sprite_soldier_red = Sprite::create(“units/soldier_red.png”);
Sprite* sprite_soldier_blue = Sprite::create(“units/soldier_blue.png”);

base_soldier.setSpriteForTeam(“red”, sprite_soldier_red);
base_soldier.setSpriteForTeam(“red”,sprite_soldier_blue);
```

Figure 92: Sprite representation for soldier Units (for red and blue team)

- Tanks:

They are much more powerful and resistant than soldiers, but also have a limited range of attack. They can only move through land *Terrains* as well.

```
BaseUnit base_tank (“tank”);
base_tank.setAttribute(“hp”, 100);
base_tank.setAttribute(“resistance”, 50);
base_tank.setAttribute(“strength”, 30);
base_tank.setAttribute(“cost_fuel”, 10);
base_tank.setAttribute(“cost_funds”, 20);


base_tank.setMovementPoints(5);

Sprite* sprite_tank_red = Sprite::create(“units/tank_red.png”);
Sprite* sprite_tank_blue = Sprite::create(“units/tank_blue.png”);
base_tank.setSpriteForTeam(“red”, sprite_tank_red);
base_tank.setSpriteForTeam(“blue”, sprite_tank_blue);
```

Figure 93: Sprite representation for tank Units

- Planes:

They have a much wider range of movement and can move through all kinds of *Terrains*. They also have a lot of strength but, in exchange, they are not very resistant to attacks.

```
BasePlane base_plane (“plane”);
base_plane.setAttribute(“hp”, 60);
base_plane.setAttribute(“resistance”, 20);
base_plane.setAttribute(“strength”, 35);
base_plane.setAttribute(“cost_fuel”, 30);
base_plane.setAttribute(“cost_funds”, 20);

base_plane.setMovementPoints(5);

//we also set their speed to 8 tiles per second in order to make them look faster to
the eye as well:
base_plane.setMovementSpeed(8);

Sprite* sprite_plane_red = Sprite::create(“units/plane_red.png”);
Sprite* sprite_plane_blue = Sprite::create(“units/plane_blue.png”);
base_plane.setSpriteForTeam(“red”, sprite_plane_red);
base_plane.setSpriteForTeam(“blue”, sprite_plane_blue);
```

Figure 94: Sprite representation for plane Units

- Ships:

Although they can only move through water and are the slower *Units*, they also have the highest range and firepower. They can attack their enemies from the sea and even have an *Ability* to destroy land *Terrains* in order to open their path.

```
BaseShip base_plane (“ship”);
base_ship.setAttribute(“hp”, 140);
base_ship.setAttribute(“resistance”, 60);
base_ship.setAttribute(“strength”, 40);
base_ship.setAttribute(“cost_fuel”, 40);
base_ship.setAttribute(“cost_funds”, 40);
base_ship.setMovementPoints(3);

//we also set their speed to 3 tiles per second in order to make them look slower to
the eye as well:
base_ship.setMovementSpeed(3);

Sprite* sprite_ship_red = Sprite::create(“units/ship_red.png”);
Sprite* sprite_ship_blue = Sprite::create(“units/ship_blue.png”);
base_ship.setSpriteForTeam(“red”, sprite_ship_red);
base_ship.setSpriteForTeam(“blue”, sprite_ship_blue);
```

Figure 95: Sprite representation for ship Units

6.4.1 Abilities Configuration

There are the following *Abilities* within the game:

- Attack:

All the types of *Units* can perform it and can only be dealt to *Units* that are in contiguous positions:

```
Ability attack (“attack”);
attack.setButtonSprites(ui/button_attack.png”, “ui/button_attack_pressed.png”);
```

```
attack.setRange(1, 1); //minimum range = maximum range = 1 (only contiguous positions)
attack.setOnEnemies(true);
attack.setOnAllies(false); //disable attacks on allies
attack.setOnEmptyTerrains(false); //disable attacks on Terrains with no Units
attack.setDealAbilityFunction(this, FUNCTION_DEAL_ABILITY(&AppDelegate::attack));

base_soldier.addAbility(attack);
base_tank.addAbility(attack);
base_plane.addAbility(attack);
base_ship.addAbility(attack);
```

Figure 96: Button Sprites for the Attack Ability (unpressed and pressed)

In which the attack function does the following:

```
bool AppDelegate::attack(Unit* dealer, Unit* receiver, terrain* terrain_dealer,
                         Terrain* terrain_receiver)
{
   double strength_dealer = dealer->getAttribute("strength");
   long resistance_receiver = receiver->getAttribute("resistance");

   long damage_absorved = rand() % resitance_receiver;

   double damage = strength_dealer – damage_absorved;
   if (damage > 0)
   {
      dealDamage(receiver, damage);
   }
   return true;
}
```

And *dealDamage* is a function used by all *Abilities* that deal any damage:

```
void AppDelegate::dealDamage(Unit* unit, const double& damage)
{
   double remaining_hp = unit->getAttribute("hp") – damage;
   unit->setAttribute("hp", remaining_hp);

   Sprite* sprite = unit->getSprite();

   if (remaining_hp <= 0)
```

```
   {
      //no more hp
      FadeOut* fade_out = FadeOut::create(1); // the unit will fade out for 1 second
      sprite->runAction(fade_out);

      //the unit won't be removed until the animation ends
      unit->addBlockingAction(fade_out);

      unit->kill(); //the system will remove it when the blocking action ends
   }
   else
   {
      //we display a label with the % of health (as in advance wars)
      Label* label = dynamic_cast<Label*> (sprite->getChildByTag(TAG_LABEL_HP_UNIT));
      if (label != NULL)
      {
         //we get the original health from the BaseUnit
         double prop_hp = remaining_hp / unit->getBaseUnit()->getAttribute("hp");

         long hp_label = prop_hp * 10;
         if (hp_label == 0)
         {
            hp_label = 1;
         }

         stringstream ss;
         ss << hp_label;
         label->setString(ss.str();
      }
   }
}
```

- Build Barracks:

An *Ability* to buy a Barracks *Building* in the occupied position. Only soldiers are able to perform it.

```
Ability build_barracks ("build_barracks");
build_barracks.setButtonSprites("ui/button_barracks.png",
                                "ui/button_barracks_pressed.png");
build_barracks.setRange(0, 0); //only the same position
build_barracks.setOnEnemies(false);
build_barracks.setOnAllies(true); //the Unit itself occupies the position
build_barracks.setDealAbilityFunction(this, FUNCTION_DEAL_ABILITY(
                                    &AppDelegate::buildBarracks));

base_soldier.addAbility(build_barracks);
```

With the following function definition:

```
bool AppDelegate::buildBarracks(Unit* dealer, Unit* receiver, Terrain* terrain_dealer,
                                Terrain* terrain_receiver)
{
   return buildBuildingAtPos(“barracks”, dealer->getPos(), dealer->getTeam());
}


bool AppDelegate::buildBuildingAtPos(const std::string &name, const Position& pos,
                                     Team* team)
{
   ControllerGame* controller_game = Strategy2D::getInstance()->getControllerGame();

   BaseUnit* base = controller_game->getBaseBuilding(name);

   double cost_fuel = base->getAttribute(“cost_fuel”);
   double cost_funds = base->getAttribute(“cost_funds”);

   double remaining_fuel = team->getAttribute(“fuel”) – cost_fuel;
   double remaining_funds = team->getAttribute(“funds”) – cost_funds;

   if (remaining_fuel >= 0 && remaining_funds >= 0)
   {
      team->setAttribute(“fuel”, remaining_fuel);
      team->setAttribute(“funds”, remaining_funds);
      controller_game->addBuilding(pos, “barracks”, team->getName());

      return true;
   }
   return false;
}
```

- Build Harbour:

The same *Ability* as Build *Barracks*, but this time for Harbours. It is configured almost exactly like Build Barracks but instead of passing “barracks” as parameter for the *buildBuilding* function we pass “harbour”.

- Bombard:

An *Ability* for ships with which they can shoot at long ranges and destroy the *Terrains*, leaving water at their place and drowning any land *Unit* that was occupying that position. With it ships can open their way through the land.

```
Ability bombard (“bombard”);
bombard.setButtonSprites(ui/button_bombard.png”, “ui/button_bombard_pressed.png”);
bombard.setRange(2, 4);
bombard.setOnEnemies(true);
bombard.setOnAllies(false);
bombard.setOnEmptyTerrains(true); // by default is false
bombard.setDealAbilityFunction(this, FUNCTION_DEAL_ABILITY(&AppDelegate::bombard));

base_ship.addAbility(bombard);


bool AppDelegate::bombard(Unit* dealer, Unit* receiver, Terrain* terrain_receiver,
                          Terrain* terrain_dealer)
{
   setWaterAttributes(terrain_receiver); //change costs attributes and sprite animation

   if (receiver != NULL && receiver->getBaseUnit()->getName() != “plane”)
   {
      receiver->kill();
   }
   return true;
}
```

- Board and Deploy:

A set of two *Abilities* that allow ships to take one *Unit* from land and transport it across the sea. With it, *Units* can cross water to reach other *Terrains*.
Since a ship can only take one *Unit* at a time, it loses the *Ability* while transporting it, but gains the Deploy, which allows the ship to deploy the transported *Unit* on a piece of land.

```
Ability board (“board”);
board.setButtonSprites(ui/button_board.png”, “ui/button_board_pressed.png”);
board.setRange(1, 1); //only for units in contiguous positions
board.setOnEnemies(false);
board.setOnAllies(true);
board.setOnEmptyTerrains(false);
```

```
board.setDealAbilityFunction(this, FUNCTION_DEAL_ABILITY(&AppDelegate::board));

base_ship.addAbility(board);

bool AppDelegate::board(Unit* dealer, Unit* receiver, Terrain* terrain_dealer,
                        Terrain* terrain_receiver)
{
   if (receiver != NULL)
   {
      string base = receiver->getBaseUnit()->getName();
      if (base == “soldier” || base == “tank”)
      {
         Vector<FiniteTimeAction*> movements;
         //we animate the Unit to move to the ship
         movements.pushBack(MoveTo::create(0.25, dealer->getSprite->getPosition()));
         //when it reaches the position we make it invisible (it is inside the ship)
         movements.pushBack(ToggleVisibility::create());

         Sequence* seq = Sequence::create(movements);
         receiver->getSprite()->runAction(seq);
         dealer->addBlockingAction(seq);

         //we set an invalid position for the unit
         receiver->setPosition(INVALID_POSITION);

         //we store the id of the unit for later putting deploying it
         dealer->setAttribute(“id_unit”, receiver->getId());

         //the ship can only have one Unit on it, so no ore boards
         dealer->removeAbility(“board”);

         //we add the deploy Ability
         Ability deploy(“deploy”);
         deploy.setRange(1,1);
         deploy.OnAllies(false);
         deploy.setOnEnemies(false);
         deploy.setOnEmptyTerrains(true);
         deploy.setDealAbilityFunction(this,
                                       FUNCTION_DEAL_ABILITY(&AppDelegate::deploy));
         dealer->addAbility(deploy);
      }
      return true;
   }
}

bool AppDelegate::deploy(Unit* dealer, Unit* receiver, Terrain* terrain_dealer,
                        Terrain* terrain_receiver)
{
   ControllerGame* controller_game = Strategy2D::getInstance()->getControllerGame();
```

```
   Unit* boarded_unit = controller_game->getUnit(dealer->getAttribute("id_unit"));
   strinstream ss;
   ss << "cost" << boarded_unit->getBaseUnit()->getName();

   //if the Unit can move to the Terrain
   if (terrain_receiver->getAttribute(ss.str()) != -1)
   {
      boarded_unit->setPos(terrain_receiver->getPos());

      Sprite* sprite = boarded_unit->getSprite();
      //we must set the sprite originally at the ship's position to show the movement
        animation towards the land
      sprite->setPosition(dealer->getSprite()->getPosition());

      Vector<FiniteTimeAction*> movements;
      movements.pushBack(ToggleVisibility::create());
      movements.pushBack(MoveTo::create(0.25,
                        terrain_receiver->getSprite89->getPosition());

      Sequence* seq = Sequence::create(movments);
      sprite->runAction(seq);

      dealer->addBlockingAction(seq);

      dealer->removeAbility("deploy");

      //we add the board Ability
      Ability board ("board");
      board.setButtonSprites(ui/button_board.png", "ui/button_board_pressed.png");
      board.setRange(1, 1); //only for units in contiguous positions
      board.setOnEnemies(false);
      board.setOnAllies(true);
      board.setOnEmptyTerrains(false);
      board.setDealAbilityFunction(this, FUNCTION_DEAL_ABILITY(&AppDelegate::board));

      dealer->addAbility(board);

      return true;
   }
   return false;
}
```

6.4.2 Adding BaseUnits to the System

Finally, in order to be able to access this *BaseUnits* in the engine and create *Units* using them as template, we need to add them to *Strategy2D*:

```
strategy->addBaseUnit(base_soldier);
strategy->addBaseUnit(base_tank);
strategy->addBaseUnit(base_plane);
strategy->addBaseUnit(base_plane);
```

6.5 BaseBuilding Configuration

The configuration for *Buildings* is very similar to the one for Units. The only difference lies in the fact that *Buildings* do not have *Abilities* but are able to recruit *Units* instead.
We will have the following types of *Buildings*:

- Barracks:

The Recruitment *Building* for land and air Units. It is configured with the following code:

```
BaseBuilding base_barracks(“barracks”);
Sprite* sprite_barracks_red = Sprite::create(“buildings/barracks_red.png”);
Sprite* sprite_barracks_blue = Sprite::create(“buildings/barracks_blue.png”);
base_barracks.setSpriteForTeam(“red”, sprite_barracks_red);
base_barracks.setSpriteForTeam(“blue”, sprite_barracks_blue);

//we add the unit types so that they can be recruited with these type of buildings

base_barracks.addBaseUnitToCatalogue(“soldier”, “ui/recruit_soldier_button.png”,
“ui/recruit_soldier_button_pressed.png”, “ui/recruit_soldier_button_unaffordable.png”,
“ui/recruit_soldier_button_pressed_unaffordable.png”);


base_barracks.addBaseUnitToCatalogue(“tank”, “ui/recruit_tank_button.png”,
“ui/recruit_tank_button_pressed.png”, “ui/recruit_tank_button_unaffordable.png”,
“ui/recruit_tank_button_pressed_unaffordable.png”);
```

```
base_barracks.addBaseUnitToCatalogue(“plane”, “ui/recruit_plane_button.png”,
“ui/recruit_plane_button_pressed.png”, “ui/recruit_plane_button_unaffordable.png”,
“ui/recruit_plane_button_pressed_unaffordable.png”);
```

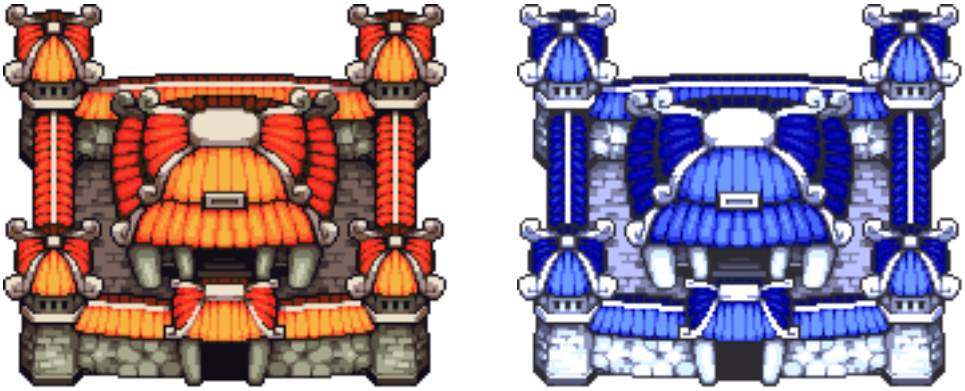

Figure 97: Sprite representation for barracks Building

- Harbour:

The Recruitment *Building* for bringing water *Units* to the game, in this case just ships:

```
BaseBuilding base_harbour(“harbour”);
Sprite* sprite_harbour_red = Sprite::create(“buildings/harbour_red.png”);
Sprite* sprite_harbour_blue = Sprite::create(“buildings/harbour_blue.png”);
base_harbour.setSpriteForTeam(“red”, sprite_harbour_red);
base_harbour.setSpriteForTeam(“blue”, sprite_harbour_blue);

//we add the ship type so that they can be recruited with these type of buildings

base_harbour.addBaseUnitToCatalogue(“ship”, “ui/recruit_ship_button.png”,
“ui/recruit_ship_button_pressed.png”, “ui/recruit_ship_button_unaffordable.png”,
“ui/recruit_ship_button_pressed_unaffordable.png”);
```

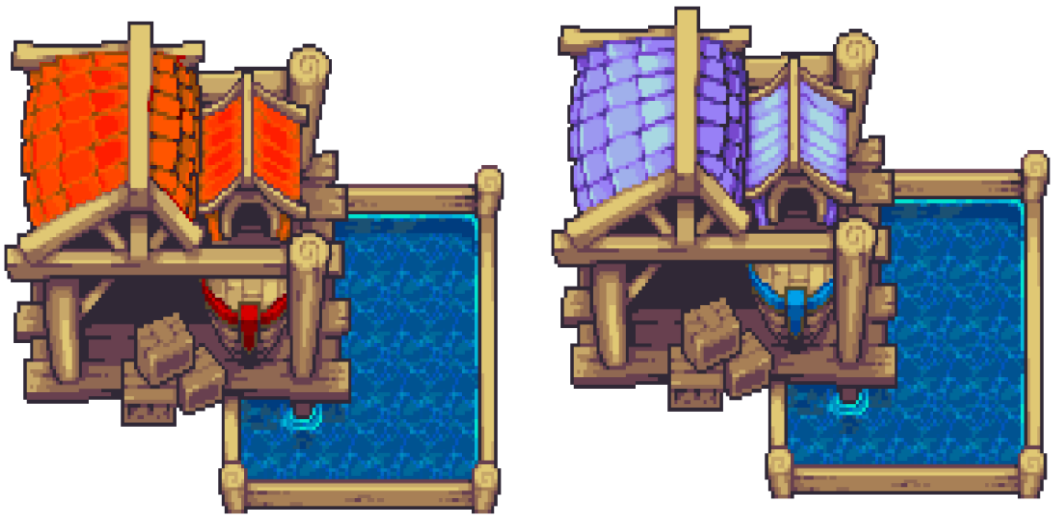

Figure 98: Sprite representation for Harbour Building

Then, we add them to the system with these calls to *Strategy2D*:

```
strategy->addBaseBuilding(base_barracks);
strategy->addBaseBuilding(base_harbour);
```

Following this configuration, we also must define the buy function callbacks, in order to set when a *Team* can recruit a *Unit* and how this affects the team. In this case we will directly use the cost attributes, in a very similar way to the function for building *Buildings*:

```
strategy->setFunctionCanBuy(this, FUNCTION_CAN_BUY(&AppDelegate::canBuy));
strategy->setFunctionBuy(this, FUNCTION_BUY(&AppDelegate::buy));

bool AppDelegate::canBuy(Team* team, BaseUnit* base_unit)
{
   double cost_fuel = base_unit->getAttribute("cost_fuel");
   double cost_funds = base_unit->getAttribute("cost_funds");

   double remaining_fuel = team->getAttribute("fuel") - cost_fuel;
   double remaining_funds = team->getAttribute("funds") - cost_funds;

   return (remaining_fuel >= 0 && remaining_funds >= 0);
}
```

```
bool AppDelegate::buy(Team* team, BaseUnit* base_unit)
{
   double cost_fuel = base_unit->getAttribute("cost_fuel");
   double cost_funds = base_unit->getAttribute("cost_funds");

   team->setAttribute("fuel", team->getAttribute("fuel") – cost_fuel);
   team->setAttribute("funds", team->getAttribute("funds") – cost_funds);

   return true;
}
```

6.6 Path Finding Configuration

Taking into account the configuration that we have set for *Buildings* and *Units*, when trying to determine whether a Unit can move through a *Terrain* and at what cost, we just need to check the cost attribute for the name of its *BaseUnit*. This is the code of the movement function:

```
bool AppDelegate::moveTo(Unit* unit, Terrain* terrain, Terrain* previous_terrain,
                         double & cost)
{
   bool can_move = false;
   cost = terrain->getAttribute("cost_" + unit->getBaseUnit()->getName());

   if (cost > 0)
   {
      can_move = true;
   }

   return can_move;
}
```

To set this function so that the engine uses it when computing the path finding we use the following line of code:

```
strategy->setMovementFunction(this, FUNCTION_MOVEMENT(&AppDelegate::moveTo));
```

6.7 Turn Handling

In order to set the gameplay, we must ensure that only the *Units* that pertain to the *Team* in command of the turn can be selected and moved. Also, we must limit their movements so that they can only perform one action each turn. For this, we will use a set of callback functions:

```
bool AppDelegate::unitSelected(Unit* unit)
{
   //if it hasn't moved and pertains to the current team, which is not the AI
   if (unit->getAttribute("moved") == false && unit->getTeam() ==
      Strategy2D::getInstance()->getControllerGame->getTeamCurrentTurn() &&
      !unit->getTeam()->getAIControlled())
   {
      return true;
   }
   return false;
}

void AppDelegate::unitMovementEnded(Unit* unit)
{
   //we mark it as moved so that it can't be selected again
   unit->setAttribute("moved", true);
}
```

And the same code for the *Building* functions. Then, at the turn start we start them all as not moved:

```
void AppDelegate::turnStart(Team* team)
{
   map<long, Building*>::iterator it = team->getBuildings().begin();
   for (; it != team->getBuildings().end(); ++it)
   {
      it->second->setAttribute("moved", false);
   }
   map<long, Unit*>::iterator it = team->getUnits().begin();
   for (; it != team->getUnits().end(); ++it)
   {
      it->second->setAttribute("moved", false);
   }

   if (team->getAIControlled())
   {
      handleAI(team);
   }
```

```
   else
   {
      startMenus(team);
   }
}
```

If the *Team* is controlled by the AI we must set what will be the actions for its members, which is done in the *handleAI* function.

In order to display some information about the resources and provide a way for the player to end its turn we configure some menus (in here we suppose that they are already created and added to the *Layer*, though when configuring it the user must set which will be their positions):

```
void AppDelegate::startMenus(Team* team)
{
   LayerMap* layer_map = Strategy2D::getInstance()->getLayerMap();

   Label* label_fuel = layer_map->getChildByTag(LABEL_FUEL);
   std::stringstream ss;
   ss << "Fuel: " << team->getAttribute("fuel");
   label_fuel->setString(ss.str());

   Label* label_funds = layer_map->getChildByTag(LABEL_FUNDS);
   ss.clear();
   ss << "Funds: " << team->getAttribute("funds");
   label_funds->setString(ss.str());

   //we set as visible and enabled the button for passing turn (the AI can't):

   Button* button_end_turn = layer_map->getChildByTag(BUTTON_END_TURN);
   button_end_turn->setVisible(true);
   button_end_turn->setEnabled(true);

}
```

Where the button for passing turn has the following callback function (configured using the normal Cocos2d-x *Button* API):

```
void AppDelegate::endTurnTouched(Button* sender, TouchEventType type)
{
   if (type == TOUCH_EVENT_ENDED)
   {
      Strategy2D::getInstance()->getControllerGame()->getTeamCurrentTurn()->endTurn();

      //we disable the button, if the team is not controlled by the AI
      //it will be set again
      sender->setEnabled(false);
      sender->setVisible(false);
   }
}
```

6.8 Running the Game

Finally, we must set where the actual instances of the *Units* and *Buildings* will be placed. In this case we provide the *Teams* with two soldiers, one plane and a barracks *Building* at each corner of the map:

```
strategy->addUnit(1, 3, “soldier”, “red”); //row, column, BaseUnit and Team
strategy->addUnit(2, 2, “soldier”, “red”);
strategy->addUnit(3, 3, “plane”, “red”);
strategy->addBuilding(2, 2, “barracks”, “red”);

strategy->addUnit(19, 17, “soldier”, “blue”);
strategy->addUnit(18, 18, “soldier”, “blue”);
strategy->addUnit(17, 17, “plane”, “blue”);
strategy->addBuilding(18, 18, “barracks”, “blue”);
```

Finally, we must call the run function in order to start *Strategy2D*:

```
strategy->run();
```

With this, we will be able to run the game on any Android or iOS device. Which will look like the following screenshot:

Figure 99: In-game capture of the demo

7. PLANNING AND COSTS

7.1 Planning

In order to achieve the objectives of the project, I needed to establish a development process that allowed me to combine my formal job with the creation of the engine. For this reason, I organized myself for being able to work on the project on those available moments.

Video games, with their focus on user experience, require an approach in which all new features developed need to be tested and evaluated immediately. This method generates an iterative process in which every new feature is tested against all the different parts of the project and adapted to fit in the most natural way.

Developing an engine shares some similarities and, since it is a tool for the creation of video games, the best way for testing it is thinking in possible ways of using it and applying them to see the effects.

Through the development of the different stages of the project, while adapting the different modules to fulfill the objectives, I also kept testing all the newly included features in order to see how they adapted to the whole project. Applying real-life cases in which I implemented new mechanics in order to see if the engine allowed their implementation in the intended way and checking for any bugs or discordances in the interface.

To be able to organize the process, I divided the development into the following sections:

- **Vision and Setup:**

Before starting the development, I first needed to look into the project's objectives to organize how the project would be developed. I looked for games belonging to the turn-based strategy genre, tested them checking their features and thought of ways for enabling their implementation in the engine. Finally, I

also thought on what would be the tools that I would use and decided for Cocos2d-x.

- **Cocos2d-x Learning:**

In order to be able to develop the project, I first needed to familiarize myself with the Cocos2d-x framework and test whether it would allow me to develop the engine I had thought of. This led me to develop a range of simple games in order to test all the capabilities of the framework, which gave a much wider view of what Cocos2d-x was and the functionalities it offered.

- **View Development:**

Since it was vital to be able to visualize the effects the modifications of the Domain on the screen in order to test any feature, I first centered on having a working View Layer that would allow me to have a tangible product early on the development and also make the testing process easier.

- **Domain Development:**

The Domain is the core of the engine, so while developing the View Layer I also started developing the different modules that composed it according to the vision and the design previously decided for the project. When I had a working View Layer I was able to quickly test any of the new features added to the Domain.
Development was a very iterative process and sometimes depended on the deployment of demo builds that implemented new mechanics, which triggered the need for new features or presented problems that needed to adapt some of the modules.

- **Demo Development**

While designing and implementing the Domain Layer, I also created several demos that tested the features added. This way, whenever a new feature was implemented I was able to see how it was being used from a user perspective and, when playing with it on the device, to see it also as a player.
This section also implied finding, adapting and creating art for the game, which is also one of the main contents of video game development.

- **Documentation:**

While developing the other sections, I also kept writing the documentation of the project

Some of the sections were developed in parallel following an agile approach, but this was the specific time dedicated to each one of them:

Sections	Hours
Vision and Setup	25
Cocos2d-x Learning	60
View Development	120
Domain Development	340
Demo Development	155
Documentation	90
Total	**790**

The project development took place from January 2014 up to the ends of October 2014. Below we have the Gantt chart for the development:

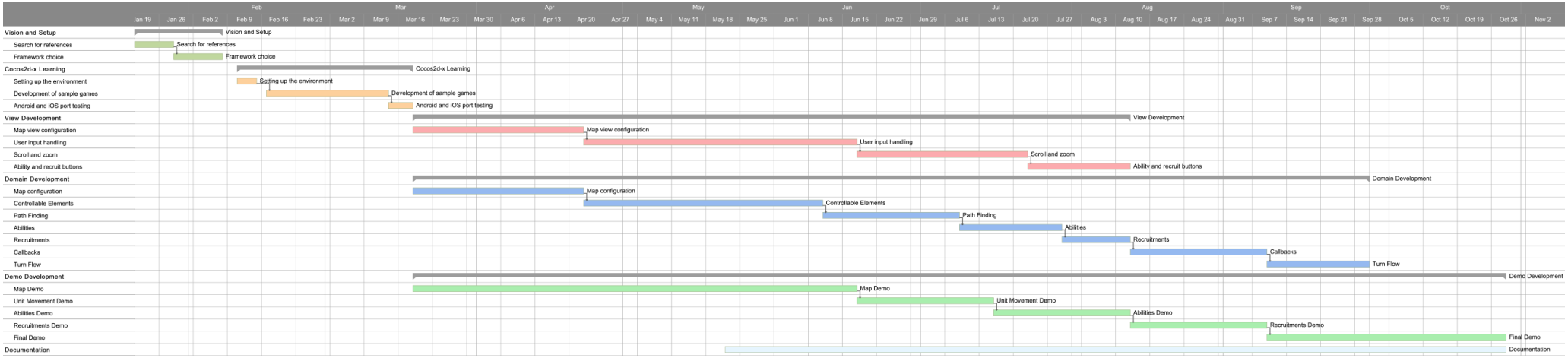

Figure 100: Gantt chart of the project development

7.2 Costs

Once we know what was the job performed during the development of the project, we can make an evaluation of its costs. For this, we must count both the human resources and the tools employed:

7.2.1 Human Resources

With the exception of the Demo Development, all other sections were performed as an engineer, but for this last, there was also a big workload in making and adapting the art to the demos and designing the features. For this reason, I will consider that a third of the time employed on this section was spent in the role of game designer (51 hours).

I have considered that the salary per hour of an engineer is of 30 €, and for a junior game designer of 20 €.

Taking this into account we have the following costs:

Sections	Hours	Cost/Hour	Cost
Vision and Setup	25	30	750 €
Cocos2d-x Learning	60	30	1,800 €
View Development	120	30	3,600 €
Domain Development	340	30	10,200 €
Demo Development (Engineer)	104	30	3,120 €
Demo Development (Designer)	51	20	1,020 €
Documentation	90	30	2,700 €
Total			**23,190 €**

7.2.2 Tools

The following tools and materials haven been used through the development:

Tools	Cost
Work Station (iMac 27")	1,384 €
iOS Testing Device (iPad mini)	349 €
Android Testing Device (Xiaomi 2s)	219 €
Apple Developer Certificate (1 year)	80 €
Total	**3,032 €**

However, since the development of the project has taken 11 months, we will take into account the amortizations for one year of development:

- The Work Station has a service life of 5 years, this gives a cost of 276.8 € for one year.
- An iPad also has a service life of 5 years, which gives us a cost of 69.8 € for one year.
- A Xiaomi 2s device has a service life of around 3 years (taking into account statistical data on android mobile devices), this gives a cost of 73 € for one year.

Taking this into account we have the following costs:

Materials	Cost
Work Station (iMac 27")	276.8 €
iOS Testing Device (iPad mini)	69.8 €
Android Testing Device (Xiaomi 2s)	73 €
Apple Developer Certificate (1 year)	80 €
Total	**499.6 €**

With this, we have that the final cost of the project would result from the addition of the human resources cost plus the cost of the tools and resources, which makes a total cost of 23,190 € + 499.6 € = 23,689 €, which considering that the project was developed during 11 months, gives us a cost of 2,153 € per month.

7.3 Development with the Engine

When using the engine, a developer frees himself from most of the programming job related to the creation of the video game.
For this reason, we can consider that the job directly related to coding would be almost negligible in most cases, taking just a matter of weeks to fully program the necessary callbacks and configurations. However, we should add a prior period of time for familiarization with the tools that the engine provides.
By using the engine, most of the workload for the development of a game belonging to the genre would lie in the design and art, making it much easier to develop them in shorter periods of time and hence reducing their cost.

8. CONCLUSION

With the development of this project I have learned the main difficulties that come with the development of a game engine, which is not just a program, but a tool that must deliver a friendly and simple environment for its users and, at the same time, give them the possibility of skipping the boundaries of the established so that they can focus on improving the contents and innovating.
In order to do this, it is primordial to set a balance between those parts that are totally controlled by the engine and those that are up to the users to configure in their own way.
With this in mind, I have been forced to think in terms of usability and adaptation, looking for the best programming practices and adopting the most adequate programming patterns to fulfill the goals. By doing this, my knowledge over software design has substantially increased and with it, I have discovered many new possibilities that the C++ language offers to achieve different goals.

Cocos2d-x has allowed me to directly provide the user with a great API for handling the graphics, sounds and animations of the elements in the game. With it, users can configure the visuals of all the elements and establish their animations and sound effects in association to the action they are performing.
In addition, the engine is totally portable between the different target platforms and I have been able to play games developed with it without any notable difference in Android, iOS, Windows and Mac, and it is also compatible with Linux.
Since Cocos2d-x is an always expanding and updating framework, I have always been using the latest version available on their website, updating any time a new version was released.
Although the global experience of working with Cocos2d-x has been very positive, one of the main problems of the framework is its lack of stable documentation that is maintained and updated at the same speed as the source code. This problem is partially attenuated by a very active community of developers, who share their ideas and code on the platform forums, often helping with any problems and questions other users may have.

I started the project using Cocos2d-x.2.2, which at the time was the latest revision, and have ended up using Cocos2d-x.3.2. Although 2.x versions offered compatibility with Blackberry and Marmalade, the newest ones do not. I have used some of the new features of 3.x, but it would be pretty easy to adapt the engine back to these older versions to enable this compatibility. But once again these were not the target platforms and I did not have the means to test the engine on them.

Strategy2D offers an adaptable framework that can be configured to reproduce almost any kind of game pertaining to the turn-based strategy genre with around 400 public functions distributed over the 26 different modules that compose the engine, in addition to all of the Cocos2d-x framework functions.

It is very easy and fast to develop a normal strategy game but, at the same time, it allows the user to go deeper into the development and adapt the callbacks to develop a more complex type of game.

8.1 Work for the future

While the project has fulfilled all the goals of the project, there are parts of it that could be improved or changed in order to offer the user a much more intuitive and user friendly layout.

Another important enhancement would be to provide the users with more control schemes and offer them the means to configure them, since by now there is only one game flow that establishes a preset of interactions between the user and the device.

Although the engine gives the users total control over the callbacks, it would also be a nice improvement to offer a wide range of presets that would make this configuration much easier.

Despite the project was established as the development of an engine for turn-based games, it would also be possible to adapt it to a real-time gameplay without having to make any change into its core and by just making some adjustments into the control schemes and how the AI is handled.

It would also be a good idea to adapt the engine in order to support 3d graphics. With cocos2d-x now starting to feature 3d assets, an adaptation to them would be pretty simple. Another option would be to port it into a different development framework more centered on three-dimensional graphics. Either way, this could be achieved by changing only the View Layer of the engine, and not have any major effects on its core.

Finally, another possibility would be the adaptation of the engine in order to run real-time strategy games as well as turn-based, this would only require small changes in how the turns are handled and in the control scheme for the player.

9. Bibliography

'The rebirth of turn-based strategy games', CNN
http://edition.cnn.com/2012/07/23/tech/gaming-gadgets/rebirth-turn-based-strategy-games/

Game Engine, Wikipedia
http://en.wikipedia.org/wiki/Game_engine

Quake Engine, Wikipedia
http://en.wikipedia.org/wiki/Quake_engine

Unreal Engine, Wikipedia
http://en.wikipedia.org/wiki/Unreal_Engine

SCUMM, Wikipedia
http://en.wikipedia.org/wiki/SCUMM

Maniac Mansion, Wikipedia
http://en.wikipedia.org/wiki/Maniac_Mansion

Infinity Engine, Wikipedia
http://en.wikipedia.org/wiki/Infinity_Engine

Planescape Torment, Wikipedia
http://en.wikipedia.org/wiki/Planescape:_Torment

RPG Maker, Enterbrain
http://www.rpgmakerweb.com/

Unity3d, Unity Technologies
http://unity3d.com/

Unreal Engine 4, Epic Games
https://www.unrealengine.com/

CryEngine, Crytek
http://www.crytek.com/cryengine

Corona SDK, Corona Labs
http://coronalabs.com/products/corona-sdk/

List of Game Engines, Wikipedia
http://en.wikipedia.org/wiki/List_of_game_engines

Turn-based Strategy, Wikipedia
http://en.wikipedia.org/wiki/Turn-based_strategy

Wargaming, Wikipedia
http://en.wikipedia.org/wiki/Wargaming

Turn-based Tactics, Wikipedia
http://en.wikipedia.org/wiki/Turn-based_tactics

Advance Wars, Wikipedia
http://en.wikipedia.org/wiki/Advance_Wars

Tactical Role-Playing Game, Wikipedia
http://en.wikipedia.org/wiki/Tactical_role-playing_game

Fire Emblem, Wikipedia
http://en.wikipedia.org/wiki/Fire_Emblem

Tactics Ogre, Wikipedia
http://en.wikipedia.org/wiki/Tactics_Ogre:_Let_Us_Cling_Together

Wasteland 2, Wikipedia
http://en.wikipedia.org/wiki/Wasteland_2

X-COM Series, Wikipedia
http://en.wikipedia.org/wiki/X-COM

Civilization Series, Wikipedia
http://en.wikipedia.org/wiki/Civilization_(series)

The best 2D Game Engines, Slant
http://www.slant.co/topics/341/~what-are-the-best-2d-game-engines

Amazon offered $600 million for Cocos2d-x, Pocket Gamer
http://www.pocketgamer.biz/asia/news/59204/amazon-offered-us-600-million-for-cocos2d-x-says-chukong-ceo/

Cocos2d-x, Chukong Technologies
http://www.cocos2d-x.org/

Cocos2d-x Wiki, Chukong Technologies
http://www.cocos2d-x.org/wiki

OpenAL, Wikipedia
http://en.wikipedia.org/wiki/OpenAL

OpenAL, openal.org
http://www.openal.org/

OpenGL, Wikipedia
http://en.wikipedia.org/wiki/OpenGL

OpenGL, Khronos Group
https://www.opengl.org/

OpenGL ES, Wikipedia
http://en.wikipedia.org/wiki/OpenGL_ES

OpenGL ES, Khronos Group
https://www.khronos.org/opengles/

Cocos2d-x Online API Reference, Chukong Technologies
http://www.cocos2d-x.org/reference/native-cpp/V3.3rc0/index.html

Stack Overflow
http://stackoverflow.com/

Alphat Blending, Wikipedia
http://en.wikipedia.org/wiki/Alpha_compositing

Dangling Pointer, Wikipedia
http://en.wikipedia.org/wiki/Dangling_pointer

Object Slicing in C++, GeeksForGeeks
http://www.geeksforgeeks.org/object-slicing-in-c/

Path Finding, Wikipedia
http://en.wikipedia.org/wiki/Pathfinding

Dijkstra's Algorithm, Wikipedia
http://en.wikipedia.org/wiki/Dijkstra's_algorithm

Path Finding Demystified, Gabriel Gambetta
http://gabrielgambetta.com/path1.html

C++ FAQ, Bjarne Stroustrup
http://www.stroustrup.com/C++11FAQ.html

The C++ Language (Fourth Edition), Bjarne Stroustrup
Addison-Wesley Professional (May 2013)

Game Engine Architecture, Jason Gregory
A K Peters / CRC Press (June 2009)