A Project Report On

“RoboDef – A Tower Defense Game”

For

Veermata Jijabai Technological Institute

Submitted

in partial fulfillment of

the requirements of the degree of

Master Of Computer Applications

By

Swapnil S. Patil

Roll No: 102010030

Under the guidance of

Prof. Archana G. Pai



(MCA Department )

Veermata Jijabai Technological Institute

Mumbai 400 019

2012-2013

**Declaration of the Student**

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea / data / fact / source in my submission.

I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature of the student

Name : Swapnil S. Patil

Roll No. MCA102010030

Date :

|  |  |
| --- | --- |
| **Certificate** | |
| This is to certify that **Swapnil S.Patil**, a student of **Master of Computer Applications**, has completed the report entitled, **“**RoboDef – A Tower Defense Game**”** to our satisfaction. | |
| (Name & Signature)  **Guide / Supervisor** | (Name & Signature)  **Head of Department** |
| (Name & Signature)  **Co-Guide / Co-Supervisor** | (Name & Signature)  **Director, VJTI** |

**Certificate**

|  |  |
| --- | --- |
| The report, **“**RoboDef – A Tower Defense Game**” s**ubmitted by **Swapnil S.Patil**, is found to be satisfactory and is approved for the Degree of **Master of Computer Applications**. | |
| (Name & Signature)  Supervisor / Guide  Date:  Place: | (Name & Signature)  Examiner  Date:  Place: |
|  |  |

**Acknowledgment**

It gives immense pleasure to present my project on ‘RoboDef – A Tower Defense Game’. This acknowledgement is a small effort to express my gratitude to all those who have assisted me during the course of my project.

I am greatly indebted to express my immense pleasure and sense of gratitude towards my Project Guide **Mrs. Prof. Archana G. Pai** for her constant and valuable encouragement.

|  |
| --- |
| **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Swapnil S. Patil**  **Roll. No: 102010030** |
|  |

**Abstract**

A tower defense game is a simple game that is classified as a real time strategy game. The objective of the game is to stop enemies from reaching a destination by building towers. The game is intended to be easy and fun enough for a non-skilled gamer to enjoy, while still having the depth required to interested the strategic mastermind.

Tower defense games are a unique form of strategy games because one can either

play it in a turn-based manner or as if it were an action game. That is, players can either

place towers between rounds in a relaxed manner, or configure their towers as the enemy

attacks. As a result, tower defense games make for great casual gaming because they can

reach both the person who wants to take their time (who can play more on the turn-based

side of the game) and the skilled player with fast decision skills (who can play more on

the fast action side)

The user is charged with placing and upgrading towers to accomplish this goal. The game is ruled by a simple economy where towers and upgrades cost money and money is received for each enemy that is killed.

**Table of Contents**

**Introduction**

The project taken up by me for the 6th semester project paper involves the creation of a “Tower Defense Game” which can run on the mobile operating system called “Android”. Primary focus is given to the programming aspects of the game rather than the design aspects limited to manipulating game models and level creation.

**Project Description**

The goal of tower defense games is to try to stop enemies from crossing a map by building traps to slow them down and towers which shoot at them as they pass. Enemies and towers usually have varied abilities, costs, and upgrade prices. When an enemy is defeated, the player earns money or points, which are used to buy or upgrade towers, or upgrade the number of money or points that are earned, or even upgrade the rate at which they upgrade. The choice and positioning of the personal towers is the essential strategy of the game. In this tower defense game, Turrets will be used as ‘Towers’ and Enemies will be in the form of ‘Robots’ and ‘Fighter Planes’.

**Features:**

* At least three tower units with varying capabilities.
* At least three offensive units with varying speeds or defences.
* Multiple levels or Scenarios
* Store player progress so player can continue on the same level where they left.

**Problem Statement**

Casual gaming is a genre which has taken off in the last few years. At the same time, the casual game doesn't need to be a shallow gaming experience.

Tower defense games are relatively younger than other game genres, especially compared to other strategy games. Contemporary games are each other’s clone and there are only few original products which distinguish from the others. Thus, a game to be created should distinguish itself from the others. A combination of the implementation of the developments in the game industry and “old school” features which are generally approved by the gamer society should be included in the game.

Contemporary tower defense games are almost always casual games, which are targeted at a mass audience of casual gamers, who do not have much sense of professional gaming and seek to be entertained by simple games in short time game-plays and do not have any sense of commitment to to the game itself. Such games are simple to understand, have basic rules and lack many features that a game can provide; a sense of long-term commitment, depth of background scenario, re-playability. Having personalized profiles, having a character in the game, mixing game dynamics to a meaningful story and game-play are commonly used methods, and this game should use them properly.

Virtually all tower defense games are deployed and played online in web browsers, thus, they can each large group of users but are limited buy the capacities of these users' system and the general constraints of the web based developments in terms of playability and performance. Although such games are on the rise; as professional personal computer owner gamers and users, this product should have the concern of having the best performance as a desktop PC application on every day computers, like good graphics. Although it aims to satisfy PC users, it would be better if this project should also be used to have a web browser game only with little tweaks.

Today's many PC-based games are being converted to be played on game consoles since many players turn to these devices for a period of time.As our game can be implemented on web browser (with some additional tweaks, since it is only a secondary concern), it also should be playable on a popular game console. This problem, however, has higher concern than its playability on the web.

**Game Design Document (GDD)**

A game design document is a written outline of all the game's functionalities that will be featured throughout the gaming experience. Usually written by game designers, these documents come in many shapes, structures and sizes but they all have the same goal: put on paper what the game will be in order to guide its development. These documents will tell the reader what the game is, how it plays and why it's like it is.

It is primarily used in the video game industry to organize efforts within a developer team and it is often attached to the agreement between publisher and developer; the developer has to adhere to the GDD during game development process.

System Design: Design Goals

As in all engineering solutions, the system to be developed should also have prioritized goals, and should optimize its resources to achieve them. Since the game to be developed is a personal computer/mobile game, system should actually “entertain” casual computer users, who do not necessarily have good knowledge (or patience) to deal with difficulties that the system can create. Since this is a class project aimed to be an example of good software engineering, best methods of development should be followed.

*Reliability:*

The system should be reliable. Since reliability is one of the fundamental design goals it must be reached in Tower Defense Game, too. The system should run consistently and no data loss should occur. Even though it is not a life or death situation but a game for fun; it is still frustrating to get different results each time a set of inputs are applied.

*Modifiability:*

In order to provide different gaming experience to players, modifiability is a key factor. With high degree of modifiability, it is granted that developers can add new content easily. It is important for our project since it is a game and requires lots of content and modification of this content with game play testing. For example, when a particular tower is extremely strong that makes game very easy or a level that is extremely hard, developers should be able to modify particular objects to make it balanced.

*High Performance:*

The system distinguishes from other contemporary games by being a stand-alone mobile application, rather than limited application that is supposed work with other programs. Thus, the system should be able to use this advantage and be able to benefit from computer/mobile resources allocated for it. The system will be designed provide rich auditory and visionary experience, by not sacrificing much from the cost. A high performance is especially important for games; since they usually demand more resources to be used than other programs, and users quickly recognize a lack of performance in a game and are usually bothered with this. Since the whole idea is allowing users to have fun; a high performance is a must in a game.

*Good Documentation:*

Documentation is aimed to be well organized and thorough, because it is important for users to have a guidebook or manual before playing the game. Also, it is very crucial to have a detailed documentation of the analysis, design and development processes for other developers that like to continue the project. For these purposes, in each step time and man power is allocated for the documentation of project. As well as necessary and crucial documentation, we spared some time to make a background and a story for the game. It is fun and a quality that is looked by dedicated players to have a manual explaining the backgrounds and properties of items in the game.

*Minimum Number of Errors:*

One of our main design goals was reliability. In order to develop a reliable system, number of errors should be kept at minimum. Errors may result in loss of game data and unreliable game play; therefore error tolerance is extremely low.

System Design: Design Goals

*Ease of Learning:*

The use of options is very explicit. It does have complex structure of options menu. This will enable user to easily learn to handle the game.

*Ease of Use:* The interface of game is very well understood. User can recognize which buttons are used . User taps on the buttons to perform desired action. The design of graphics will be very well represented.

**Trade-Offs:**

Naturally, all optimization problems come with drawbacks. Although we want to minimize them, expected sacrifices are as below

*Cost (High Performance, Reliability):*

The system should be designed as a stand-alone game application, which benefits from high level frameworks and modern technologies. These features, however, are in conflict with the cost-effectiveness of the system. In computer/mobile games, however, it is not unusual to sacrifice a system's cost to its reliability or high performance; since these two are usually more important for gamers than systems cost-gamers prefer to have costly systems in terms of both hardware and software.

*Functionality (Ease of Use, Ease of Learning):*

The system is to be designed to be addressing all user groups, from novice to experts. For this very reason, it is inevitable to sacrifice some possible extra features for the sake of usage and learning ease. Hence balancing the system to be loaded with functions at the same time being simple enough will be an important task.

**Requirements Analysis:**

*Functional Requirements***:**

* The user should be able to start a new game.
* The user should be able to load the last saved game.
* The user should be able to add a new tower to the map during the game.
* The user should be able to upgrade the towers on the map during the game.
* The user should be able to exit the game during the game or between levels.
* The user should be able to start the next level when the current level finishes.

**Non-Functional Requirements:**

*Usability Requirements*:

* Inexperienced player should be able to start a new game within 60 seconds.
* Game mechanics must be adjusted properly for each background and trait options in order to provide balanced game play.
* Difficulty of the game should be adjusted carefully to avoid boringly easy or frustrating hard gameplay experience.

*Implementation Requirements*:

* For developers, adding new item (tower or enemy) should be a trivial process and shouldn’t involve any coding process. System should be able to recognize added content files and responds properly.

*Portability Requirements*:

* Resolution of the game must be suitable for all common mobile and tablet screens types.

*Performance Requirements*:

* Game should be able to run at a rate above 30 frames/second in order to provide smooth frame transitions.
* In game graphics must be clear, well designed and nice looking in order to create nicer atmosphere for players.

**Game Rules**

###### *Level Start Rules:*

* Levels start on a timer before the beginning of the first round. A Player's health is refilled at the beginning of a level.
* The game grid is clear of objects. Paths are visible to the player. The first level for the game starts the player with a default resource bonus to allow initial building of towers.

###### *Level End Rules:*

* A Player wins a level if all rounds are survived.
* A Player loses a level if that player's health is reduced to 0 during a round.

###### *Round / Wave Start Rules:*

* Each round starts with a timer counting down until the round begins and enemies flow.
* Players are encouraged to use this time to place towers effectively.
* The time between rounds decreases as the player progresses through rounds and levels.
* Players can activate a command to begin the round earlier if they wish.

###### *Round / Wave End Rules:*

Each round ends when one of two conditions is reached:

* The Source on each path is empty and no units are on the game grid. This results in a cleared round, allowing for progression into the next.
* The player's health is reduced to 0 by enemies reaching the exit. This results in a loss for the player, who then may restart the level, restart from the beginning, or quit the game.

*Path Rules:*

* The starting (source, spawning) point of a path is predetermined.
* The ending (sink, exiting) point of a path is predetermined.
* The path between the source(s) and the sink(s) is calculated.
* The path is visible to the player at all times.
* Multiple paths may exist on the game grid.
* Branches may fork.

**Game Rules**

*Enemy Rules:*

* Enemies are restricted to moving along the path
* If a fork in the path is encountered the enemy randomly determines which branch is followed.
* Enemy's have health. If that health is reduced to 0, then the enemy is destroyed.
* When an enemy is destroyed the Player earns more resources and points.
* Enemies have a movement score, which determines how quickly that unit moves across the path.
* Enemy's have a strength rating, which determines the amount of health they reduce from the Player upon reaching the exit.
* Special enemies may have additional effects if they reach the exit, such as increasing the speed of the remaining enemies.

*Points Rules*

* As the player reaches point milestones health energy, if depleted, is partially restored.

*Tower Rules*

* Players can place any tower they have the resources for at any time.
* Only one tower can exist in a grid square/placement plane at a time.
* Players can place towers on any designated building grid square/placement plane. Illegal squares are path squares and squares that are blocked.
* Towers have a strength which determines how much damage they deal to enemies.
* Towers have a cool-down time, which determines how much time passes between attacks.
* Towers have a resource cost, which determines how many of each tower the Player can construct.

Technologies and Components

**Unity3D Game Engine:**

Unity3D is an award winning tool for creating interactive 3D applications on multiple platforms.Unity3D consists of a game engine and an Editor. The engine contains software components for the most common and recurring tasks in game development. The topics covered by the engine include sound, graphics, physics and network functions. The engine supports C-Sharp, Boo, and JavaScript for script programming.

**Availability:**

Unity Pro and Pro add-ons for multiplatform publishing can be bought from the Online Store. Also the free version of unity with Basic Add-ons can be downloaded from the store.

**Content Creation:**

Unity supports the creation of almost any 2D or 3D interactive content imaginable:

Games, including: Browser-based MMOGs, First-person shooters, Racing games, Real-time strategy games, Third-person shooters, Roleplaying games, Side-scrollers etc and serious games including: military simulation, medical training, virtual reality, architectural applications, advertising and retail.

**Multiplatform Development**

Unity allows deploying a project on different platforms with minor changes. The presentation and functionality is largely maintained depending on the platform's capabilities. However, there are major differences in some areas like the input mechanisms on different devices.

The following platforms can be targeted with unity:

Mac OSX App, Windows Executable, Linux desktop, Web Browsers, via Unity Web Player, iPhone, iPad, Android phones and tablets, Wii U, PS3 and Xbox 360

Unity3D Game Engine includes these systems but is not limited to:

* Collision
* Particles
* 2D/3D Renderer
* Graphical 3D Scene Designer
* Game Object Hierarchy View

**Advantages of using unity:**

*Business Reasons*:

* Simple and straight forward licensing model: $1500 per developer seat.
* No third-party license agreements beyond a click-wrap.
* No publishing fees or royalties that would create sale revenue deductions.
* No annual or maintenance fees that would create ongoing overhead costs.

*Technical Reasons*:

* Stable & capable graphics and game engine. (Audio, HID, Physics, Asset, Pipeline, Navmesh / Pathfinding, AI plugins, etc.)
* Cross-platform for the 3 major PC operating systems.
* Graphics engine renders well in both the OpenGL and DirectX APIs.
* Team oriented version control server.
* Finished Unity3D products successfully deployed through Steam.
* Custom shader support.
* Platform specific script compilation to native code, 3 Scripting languages.

Backup and Source Control: **GIT**

**Git** is a distributed version control and source code management (SCM) system with an emphasis on speed.Initially designed and developed by Linus Torvalds for Linux kernel development, Git has since been adopted by many other projects.

Every Git working directory is a full-fledged repository with complete history and full version tracking capabilities, not dependent on network access or a central server. Git is free software distributed under the terms of the GNU General Public License version 2.

### Characteristics:

*Strong support for non-linear development:* Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers.

*Distributed development:* Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

*Compatibility with existing systems/protocols:*

Repositories can be published via HTTP, FTP, or a Git protocol over either a plain socket, ssh or HTTP. Git also has a CVS server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories.

*Efficient handling of large projects:*

Very fast and scalable,and performance tests done by Mozilla showed it was an order of magnitude faster than some version control systems, and fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server. In particular, Git does not get slower as the project history grows larger.

*Cryptographic authentication of history:*

The Git history is stored in such a way that the id of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed. The structure is similar to a hash tree, but with additional data at the nodes as well as the leaves.

*Toolkit-based design:*

Git was designed as a set of programs written in C, and a number of shell scripts that provide wrappers around those programs. Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together.

*Pluggable merge strategies:*

As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and manual editing is required.

*Garbage accumulates unless collected:*

Aborting operations or backing out changes will leave useless dangling objects in the database. These are generally a small fraction of the continuously growing history of wanted objects. Git will automatically perform garbage collection when enough loose objects have been created in the repository.

**Why use Android as a platform to launch game:**

|  |  |
| --- | --- |
| **Android** | **iPhone** |
| Simple process for releasing applications means  a guarantee that the game will be released into  the market. | Applications released in the apple store have to  be checked and approved by apple. |
| Open source. Applications gets free rein  assuming permission is given from the user. | Applications are restricted on multiple areas  including running in the background,  communicating with other applications and more |
| Android applications can be developed from any  kind of computer | iPhone applications can only be developed using  a MAC. |
| Android phones are relatively cheaper. | No easy way to acquire iPhones, |
| Game can easily be run and tested without any hindrance or pre requisites. | Even for testing the game on an iphone a developer’s license is required. |

**Asset Breakdown**

**Art Assets**

Assets will need to be low poly as the game will run on a mobile OS as its primary platform. Hence the assets will need good detail as with considerably low polygon counts.

Areas that will need art assets:

* Enemies
* Levels
* UI/Menus
* HUD
* Particle effects

**Text Assets**

There will not need to be many text assets since there is no story, narration, or dialogue in game. Most of what is needed is: names for enemies, names for towers, menu option text, and credits.

Minimal time is required for these.

**Sound Assets**

As per art assets, a variety of sounds can be imported into Unity3D. A quick explanation of music vs. sound: music is the long form sound that is looped; a sound is the short form that is usually only lasts about .25 to .5 seconds. A list of what might be needed:

* Menu/Level Picker music
* Menu/UI sounds
* In-game sounds
  + Tower projectiles
  + Enemy explosion

**Event Driven Response Table**

**Main Menu**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Event | System State | Response |
| E1 | User taps the new game button | Game is at start screen | New game starts. |
| E2 | User taps the continue button | Game is at start screen | Last saved game is resumed |
| E3 | User taps the quit game button or back button | Game is at start screen | The game ends and the window closes |

**Pause Screen**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Event | System State | Response |
| E4 | User taps quit button | Game is at the pause screen | Main menu is loaded |
| E5 | User taps the instructions button | Game is at the pause screen | The instructions page is displayed |
| E6 | User taps the resume button | Game is at the pause screen | The game resumes |

**Tower**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Event | System State | Response |
| E15 | User taps tower button | Game is running | Tower to build is selected |
| E16 | User taps on an Upgrade button | User has enough money, tower has another upgrade. User is at tower upgrade/sell menu. | Tower will be upgraded to next level. Cash will be deducted. |
| E17 | User taps on Sell button | User is at tower upgrade/sell menu. | Tower will be destroyed and cash added. |
| E18 | Enemy enters range | Tower is not aimed at enemy | Tower rotates to enemy |
| E19 | Enemy enters range | Tower is aimed at enemy | Tower shoots enemy |

**Event Driven Response Table**

**Game Play**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Event | System State | Response |
| E7 | User taps on a tower | Game is running | Shows tower upgrade/sell menu |
| E8 | User taps on an Upgrade button | User has enough money, tower has another upgrade. User is at tower upgrade/sell menu. | Tower will be upgraded to next level. Cash will be deducted. |
| E9 | User taps on Sell button | User is at tower upgrade/sell menu. | Tower will be destroyed and cash added. |
| E10 | User taps on a build space | User has a green square attached to click. | Tower is built, cash is deducted. |
| E11 | User hits back button | Game is in play | Pause screen is displayed |
| E12 | An enemy comes within range of a tower | Tower is aimed at enemy | Projectiles are shot, Sound is played, Enemy health is decremented |
| E13 | An enemy reached the end of the path | User lives > 1 | User lives is decremented |
| E14 | An enemy reached the end of the path | User lives = 1 | User lives is decremented to zero and game over image is displayed |

**Enemy**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Event | System State | Response |
| E20 | Enemy creation | Time between waves has elapsed | An appropriate amount of enemies is created |
| E21 | Enemy enters range of a tower | Tower is aimed at Enemy | Enemy is fired upon until it exits tower range |
| E22 | Enemy is hit by tower | Enemy health > Tower Damage | Enemy health is decremented by damage amount. |
| E23 | Enemy is destroyed by tower | Enemy health is less than damage amount and is hit by tower | Enemy is destroyed and sound is played |
| E24 | Enemy scans and decides new path | Game is running, enemy spawned or current path interrupted | Enemy walks on new path |

**Use case Model**

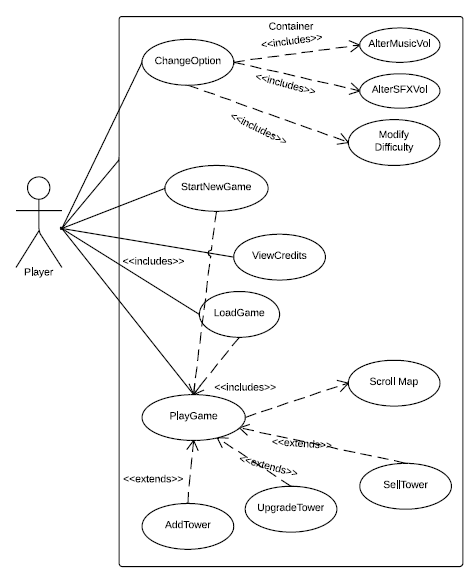


Fig: Use case Diagram for Tower defense game

1. Player turns on System and waits for menu to display.
2. Player taps on ChangeOption .
   1. Taps on Modify difficulty to change game difficulty.
   2. Taps on AlterSFX Volume to change game SoundFX volume.
   3. Taps on AlterMusicVolume to change background music Volume.
3. Player taps on ViewCredits to get info about developers.
4. Player taps on Start New Game to start game from beginning (Level 1) with default settings
5. Player taps on Load Game to load the level corresponding to user progress.
6. When the game begins user can performs functions of:
   1. Placing new Tower.
   2. Upgrading existing tower.
   3. Selling a tower.
   4. Scrolling the map.

**State Diagrams (Dynamic Models)**

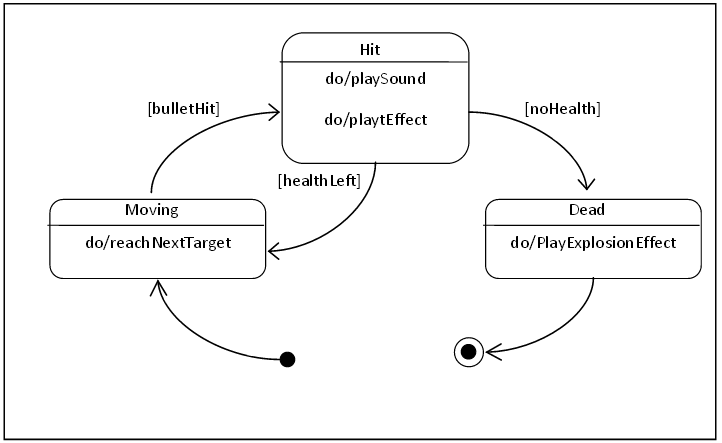


Fig: State Chart diagram for Enemy Object

There are three states of the Enemy object. When Enemy is in Moving state, it updates its position by trying to reaching next target, which is the next turning point in the path. If in Moving state, a bullet hits the Enemy, Enemy passes to Hit state, plays a sound and visual effect and checks if there is any health points left, if any left it returns to Moving state; if not it proceeds to Dead state. In Dead state, it plays an explosion effect and finishes the state machine.

**State Diagrams (Dynamic Models)**

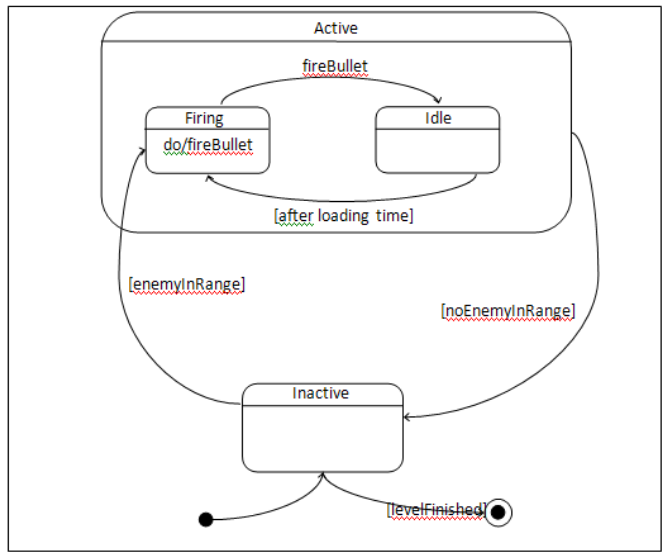


Fig: State Chart diagram for Tower Object

The states of Tower object have been diagrammed. There are two main states of Tower object; the Active and Inactive. When a level started, Tower starts in Inactive state. When there is an enemy in range it passes to the Firing state, which is a sub-state of Active state. In Firing state, Tower object calls fireProjectile() method and, whenever it calls fireProjectile() method, it passes to the Idle state. In Idle state, it waits for a “loading time”, which is predetermined by the properties of the Tower, and returns to the Firing state. Whenever in Firing or Idle state- i.e. in Active state- if there is no enemy in the range, tower returns to the Inactive state. In Inactive state, when the level finishes, Tower exits state machine.

**State Diagrams (Dynamic Models)**

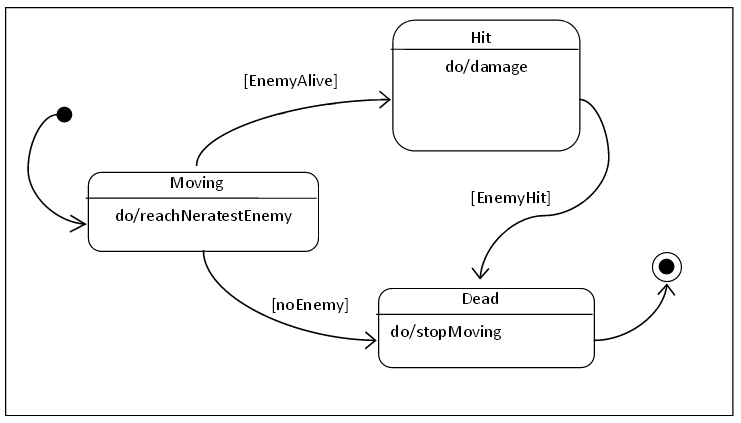


Fig: State Chart diagram for Tower Object

The states of Bullet object are diagrammed. Bullet starts in Moving state, it tries to reach the nearest enemy. When the enemy has health when Bullet reaches the position when enemy is in when the Bullet is fired; there are two conditions possible. If the Enemy is still alive, Bullet passes to Hit state and damages the Enemy, then proceeds to Dead state. If Enemy is not alive –killed by another Bullet-, Bullet jumps to Dead state. Dead state is the final state and Bullet finishes the state machine.

**State Diagrams (Dynamic Models)**

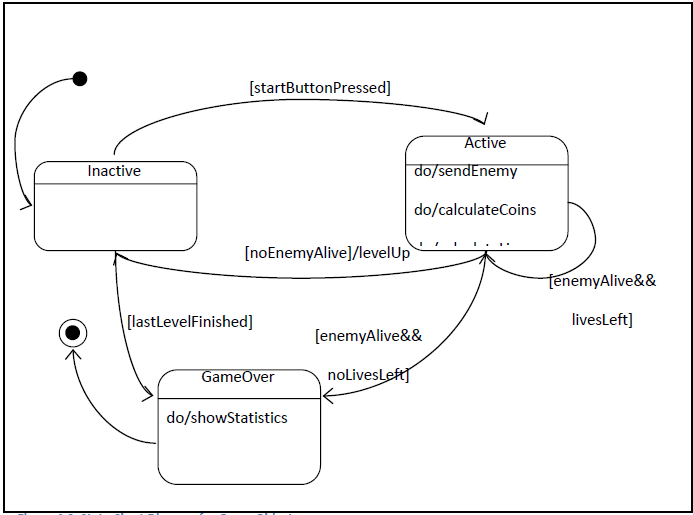


Fig: State Chart diagram for LevelMaster Game Object

The states of the LevelMaster Game object are diagrammed. LevelMaster Game object starts in Inactive state. When start button is pressed the Game passes to Active state. In Active state, Game sends the enemies in pre-defined numbers according to level, calculates the coins and lives. If last enemy is dead and there are lives left in Active state, Game passes to Inactive state and increments level. If there are enemies alive and there are no lives left in Active state or last level is reached in Inactive state, Game passes to GameOver state. In GameOver state Game shows the statistics and finishes the state machine.

**Sub-System Decomposition**

Our system consists of following subsystems:

* InputManagement Subsystem
* Sound Management Subsystem
* Viewer Subsystem
* Game Screen Management Subsystem
* Screen Elements Subsystem
* Game Logic Subsystem
* Logic Management Subsystem
* Data Management Subsystem

**Input Management Subsystem:** This subsystem’s main function is to observe user actions, detect inputs and pass these inputs to the other subsystems. It records touch events such as touch taps, touch slides, pinch etc.

**Sound Management Subsystem:** This system handles all kind of sound effects and music that will be played in the game. When notified, this system is responsible to play appropriate sound effect or music.

**Viewer Subsystem:** Viewer Subsystem’s main concern is to present game model part of the game to the users with graphics. It contains all kind of things that are represented as graphical objects like images, buttons, panels, texts etc. It holds all game screens and manages transitions between them. In order to further specialize this system, it is divided into two subsystem named as Game Screen Management Subsystem and Screen Elements Subsystem.

**Game Screen Management Subsystem:** This system is responsible for managing game screens. It stores current state of the game screen (ex: active screen that is displayed on the screen) and with given inputs decides which game screen is to be displayed.

**Screen Elements Subsystem:** This system serves as a collection of all displayable elements that can be used in typical game screen. Some of these elements are buttons, text fields and input fields and can be added to any game screen to add functionality to these screens. These elements also respond to user inputs (ex: buttons).

**Hardware/Software Mapping:**

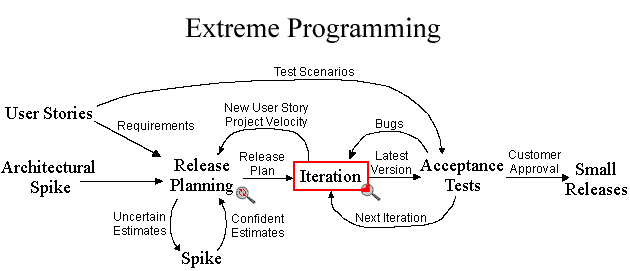
Tower Defense Game is a standalone application, which will not require any kind of web or network system to operate. In order to run Tower Defense Game, any android mobile or tablet device with medium processing and graphics capabilities will suffice.

In terms of memory management, Tower Defense is a medium – heavy weight game, memory will mostly used by game logic which is demanding since all drawing – drawing related operations will be handled by GPU.

I/O demand for project is very low; there is no exhaustive access from hard drive or any other media. Secondary storage access will be used in initialization time and won’t require much access anyway.

Being standalone application, once installed to secondary memory, project subsystems will be mapped to secondary memory, main memory and video card. Before runtime, all software will reside on secondary memory. During runtime, objects will be created and used in memory while data files (txt files for initialization or saving purposes) resides in hard drive. Other than that, graphical context (Textures, animations) will be executed in GPU.

**Development Model adopted:**

****

**Extreme Programming** is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

All the contributors to an XP project sit together, members of one team. This team must include a business representative — the “Customer” — who provides the requirements, sets the priorities, and steers the project. It’s best if the Customer or one of her aides is a real end user who knows the domain and what is needed. The team will of course have programmers. The team may include testers, who help the Customer define the customer acceptance tests. Analysts may serve as helpers to the Customer, helping to define the requirements. There is commonly a coach, who helps the team keep on track, and facilitates the process.

*Spike Solutions*: Spike solutions are created to figure out answers to tough technical or design problems. A spike solution is a very simple program to explore potential solutions. A spike should be built to only addresses the problem under examination and ignore all other concerns. Most spikes are not good enough to keep and are discarded. The goal is reducing the risk of a technical problem or increasing the reliability of a user story's estimate.

*Release Planning*: A release planning meeting is used to create a release plan, which lays out the overall project. The release plan is then used to create iteration plans for each individual iteration. It is important for technical people to make the technical decisions and business people to make the business decisions. Release planning has a set of rules that allows everyone involved with the project to make their own decisions. The rules define a method to negotiate a schedule everyone can commit to.

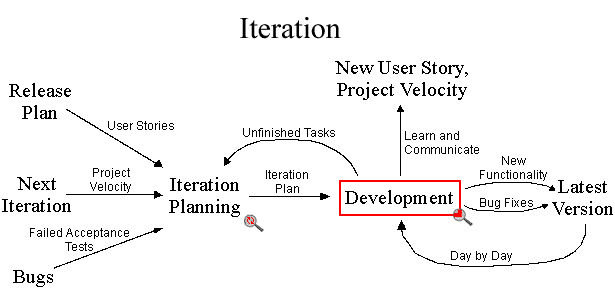
The essence of the release planning meeting is for the development team to estimate each user story in terms of ideal programming weeks.

*Acceptance Tests:* Acceptance tests are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, whatever it takes to ensure the functionality works.   
Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release.

*Project Velocity:* The project velocity (or just velocity) is a measure of how much work is getting done on the project. In order to measure the project velocity, the estimates of the user stories that were finished during the iteration are simply added up. Also added are the estimates of the tasks finished during the iteration. Both of these measurements are used for iteration planning.

This simple mechanism allows developers to recover and clean up after a difficult iteration and averages out estimates. The project velocity goes up by allowing developers to ask the customers for another story when their work is completed early and no clean up tasks remain.

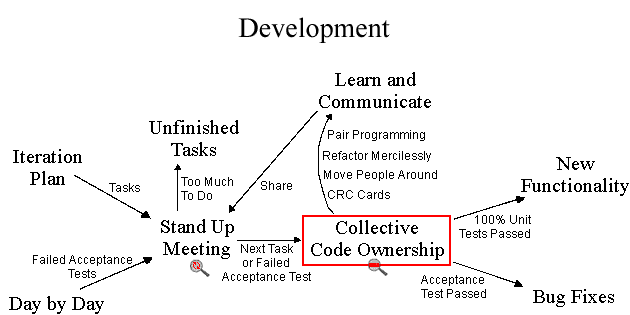
*User Stories:* User stories serve the same purpose as use cases but are not the same. They are used to create time estimates for the release planning meeting. They are also used instead of a large requirements document. User Stories are written by the customers as things that the system needs to do for them. They are similar to usage scenarios, except that they are not limited to describing a user interface. User stories also drive the creation of the acceptance tests.



An iteration planning meeting is called at the beginning of each iteration to produce that iteration's plan of programming tasks. The Release plan, Project velocity, bugs encountered and unfinished tasks act as the input for planning the next iteration. The next iteration plan is then pushed to the development phase. The development phase provides the following output:

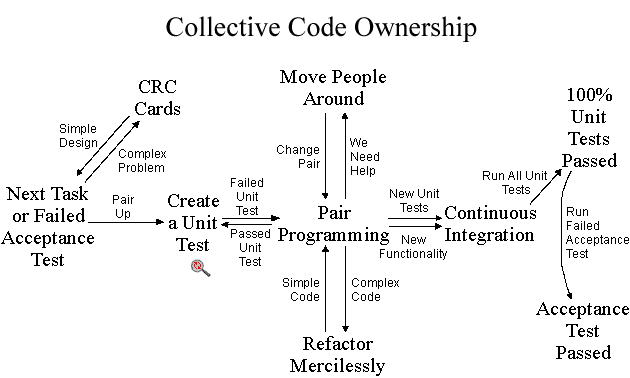
* New functionalities
* Fixes for previously reported bugs
* Determines current Project Velocity
* Points to the next User story to work upon
* Unfinished tasks to be completed in the next iteration

*Code Integration:* Developers should be integrating and committing code into the code repository every few hours, whenever possible. Continuous integration often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be re-used, or what could be shared.

****

Stand up Meeting : Communication among the entire team is the purpose of the stand up meeting. A stand up meeting every morning is used to communicate problems, solutions, and promote team focus. Everyone stands up in a circle to avoid long discussions. It is more efficient to have one short meeting that everyone is required to attend than many meetings with a few developers each.

In general, the completed tasks which failed the Acceptances tests and new tasks are discussed and forwarded for developing in “collective code ownership” form in the everyday Stand Up meeting. If the workload is too much, some tasks are left unfinished to be completed in the next iteration.

****

*Collective code ownership*: On an Extreme Programming project, any pair of programmers can improve any code at any time. This means that all code gets the benefit of many people’s attention, which increases code quality and reduces defects. XP teams follow a common coding standard, so that all the code in the system looks as if it was written by a single — very competent — individual. Collective code ownership involves concepts of pair programming, code refactoring, moving people around into various combinations, CRC cards.

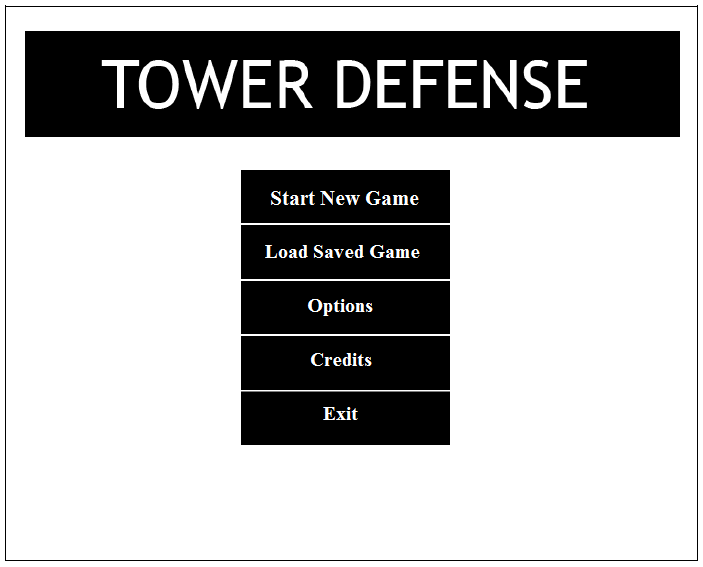
**User Stories**

There are several user stories focused on the basic user actions in the game. I have also produced stories to go along with the major actions of our units so as to better understand their functions. Each Story for the game units outlines that unit's update() method, as this method drives the actions of that unit.

* *User Stories* 
  + New Game: The User opens the game, presses the "New" button and starts a game. Reads from a file to generate a map, which avoids hard-coding map data.
  + Quit Game: The User clicks the "Quit" button and the program closes.
  + Start Game: The User clicks the "Start" button which begins the game, creating enemies.
  + Place Tower: User selects a type of tower and clicks in the map, if they have the resources and it's an open square a Tower is placed there.
    - Actors: User, the System
    - Prerequisites: Game has begun, the map has loaded
      1. User selects a type of Tower
      2. User clicks in an area of the map
         1. The System prompts the User to select a tower type if none has been selected.
         2. The area clicked is a non-build cell, the System sends a message to notify the User that a Tower cannot be built there.
         3. The area clicked is already occupied by a tower, the System sends a message to notify the User that a Tower is already built there.
      3. The System decreases the User resource amount based on the Tower type selected.
         1. The System posts a message to the User that not enough resources are available to build.
         2. The System creates a new Tower object and registers it.
* *Tower Story* 
  + The Tower look for enemies, and if they see one in their range, the fire a bullet at it.
    - Actors: Tower, Weapon, Enemy, System
    - Preconditions: a Tower is on a Map, Enemies are on the map
      1. Tower checks its counter to see if it is eligible to fire.
         1. Decrement counter if counter is not 0.
      2. Tower checks the System for a collection of Enemy units in range.
         1. Stop if no enemies are found.
      3. Tower finds closest Enemy Unit
      4. Tower creates a Weapon unit, sets the Weapon unit on the map and the Enemy unit from step 2 as the target.
      5. Reset the counter.
* *Weapon Story* 
  + Weapons move towards a targeted Enemy, if they collide they hurt that enemy and may kill it.
* *Enemy Stories* 
  + Update Story: Enemies move towards a sink based on their speed and the direction of the path object they are on.
  + Death Story: When an enemy dies it increases the player's points and resources based on its values.
* *Source Story* 
  + Source produces enemies and places them on the map until the source runs out of stock
* *Sink(Player Base) Story* 
  + If an Enemy is on the Sink(Player base) remove the enemy and damage the player based on the enemy removed.

**Screen shots**

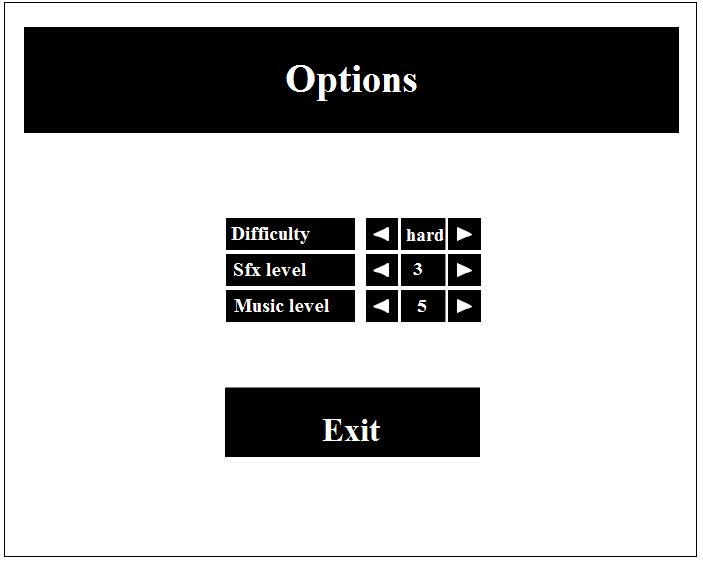
Start Menu

****

**This is the main game menu. From here the player will be able to:**

1. Start playing a game from the beginning with all settings restored to default.
2. Load saved game and start playing from the level where the player left.
3. Access Option menu
4. View Credits
5. Exit Game

Options Menu

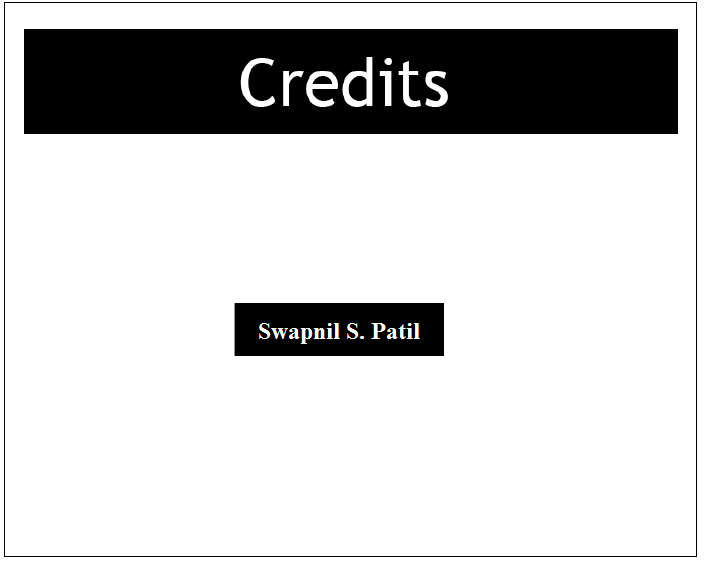


**Options Menu:**

The player can, through this menu alter the difficulty of the game. Three difficulty levels available are:

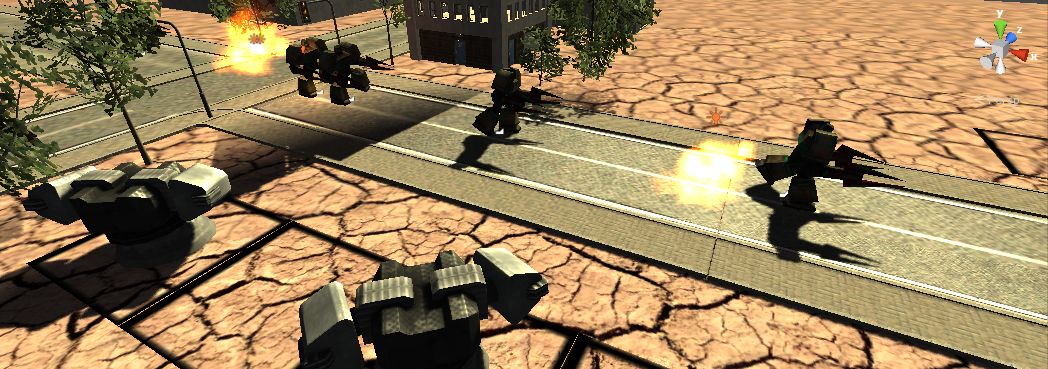
* Easy
* Medium
* Hard

Also the Volume levels of sound effects (Explosions, bullets etc) and background music volume can be adjusted.



They player can see the credits when they tap on the Credits button in the start menu.

In game screenshots

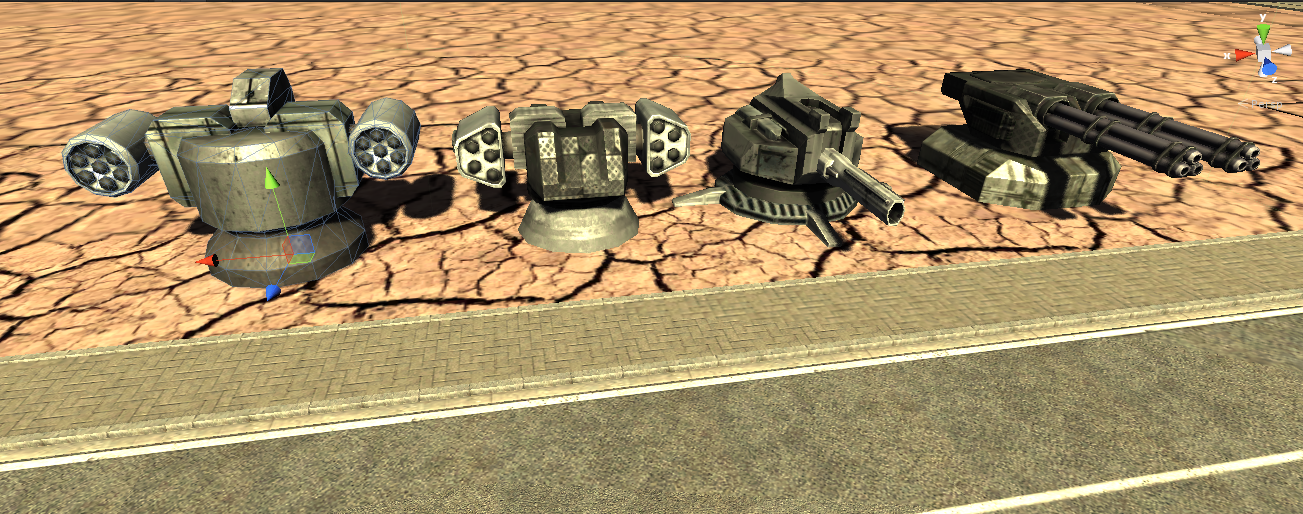


In the above screenshot, the SAM turrets are firing missiles at the enemy (Bots) as they are walking through the turrets’ firing radius’.



In this screenshot, the air units (Fighter planes) are passing over the SAM turrets which then open fire that them. The turrets fire seeking missiles which follow the planes until they hit it or the plane goes out of their range.

Screenshots (Turrets)



All the turrets (Towers) available to the player can be seen above. The two turrets on the right fire projectiles like bullets and can attack only the ground units whereas the two on the left can attack both air and ground units. They fire missiles that follow and lock on the targets. Newer turrets will be added to the arsenal in the later versions of the game.

Screenshot (Enemy Bots)



The enemies (Bots) in the game are seen here. Each bot has his own level of health and walk speed. The bots will be able to attack the towers and destroy them in the later versions of the game. More units can be expected in the future.

Screen Shot (GUI)

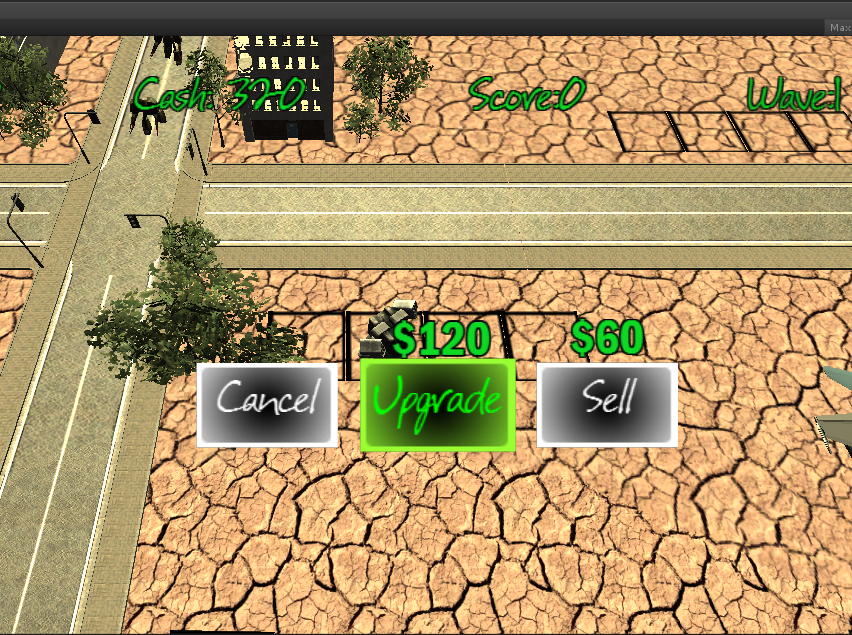


**GUI Components**:

* **Health**: The health count shows the number of enemy units that the player can afford to reach his base. It decreases as the units reach the player’s base. When the count reaches 0, the game level ends and the player loses. He can replay the same level if he wants.
* **Cash**: The amount of money available to the player. The player can spend the money on building towers and upgrading them. Apart from the starting amount, the player earns money by destroying enemy units.
* **Score**: The player scores points by destroying enemies which are then added to the total score.
* **Wave**: Indicates the number of the current wave of enemies.
* **Gatling button**: This button allows building a Gatling gun. The figure in green above the button is the price of the tower.
* **SAM button** : This button enables player to build a SAM turret and the figure above is its cost.

The black empty squares are the build areas or the placement planes. The towers can be placed only on these areas and only if they are empty.

Screen shot (Upgrade Menu)



The upgrade panel opens when a tower is selected, it contains buttons for 3 actions:

* Upgrade Button: Allows the currently selected tower to be upgraded to its next level,
* Sell Button : Sells the tower/ Destroys the game object adding some money to cash count.
* Cancel : Closes the upgrade panel.

Source Code

The source code of the Tower Defense game is provided hereafter.

LevelMaster.js

============================================================================

#pragma strict

static var playerDamage = 0;

var waveActive : boolean = false;

var spawnEnemies : boolean = false;

var upgradePanelOpen : boolean=false;

//Player Variables

var healthCount : int = 10;

var scoreCount : int = 0;

var cashCount : int = 0;

//Wave Specific

var waveLevel : int=0;

var difficultyMultiplier : float = 1.0;

var waveLength : float = 30.0;

var intermissionTime : float = 5.0;

private var waveEndTime : float = 0;

//Enemy

var enemyPrefabs : GameObject[];

var flyerSpawns : Transform;

var groundSpawns : Transform;

private var flyerSpawnPoints : Transform[];

private var groundSpawnPoints : Transform[];

var respawnMinBase : float = 1.0;

var respawnMaxBase : float = 2.0;

private var respawnMin : float = 3.0;

private var respawnMax : float = 10.0;

var respawnInterval : float = 2.5;

var enemyCount : int = 0;

private var lastSpawnTime : float =0;

//Turrets

var turretCosts : int[];

// Gui

var waveText : UILabel;

var healthText : UILabel;

var scoreText : UILabel;

var cashText : UILabel;

var costTexts : UILabel[];

var upgradeText : UILabel;

var upgradeBtn : GameObject;

var saleText : UILabel;

var upgradePanelTweener : TweenPosition;

var placementPlanesRoot : Transform;

var hoverMat : Material;

var placementLayerMask : LayerMask;

private var originalMat : Material;

private var lastHitObj : GameObject;

var onColor : Color;

var offColor : Color;

var allStructures : GameObject[];

var buildBtnGraphics : UISlicedSprite[];

private var structureIndex : int=0;

private var focusedPlane : PlacementPlane;

private var structureToUpgrade : Turrets\_Parent;

private var upgradeStructure : GameObject;

private var upgradeCost : int;

private var saleValue : int;

// Gui

function Start ()

{

structureIndex = 1;

UpdateGUI();

flyerSpawnPoints = new Transform[flyerSpawns.childCount];

var i : int = 0;

for(var theSpawnPoint : Transform in flyerSpawns)

{

flyerSpawnPoints[i] = theSpawnPoint;

i++;

}

groundSpawnPoints = new Transform[groundSpawns.childCount];

var g : int = 0;

for(var theGroundSpawnPoint : Transform in groundSpawns)

{

groundSpawnPoints[g] = theGroundSpawnPoint;

g++;

}

SetNextWave();

StartNewWave();

}

function Update () {

//Gui

var ray = Camera.main.ScreenPointToRay (Input.mousePosition);

var hit : RaycastHit;

if(Physics.Raycast (ray, hit, 1000, placementLayerMask))

{

if(lastHitObj)

{

lastHitObj.renderer.material = originalMat;

}

lastHitObj = hit.collider.gameObject;

originalMat = lastHitObj.renderer.material;

lastHitObj.renderer.material = hoverMat;

}

else

{

if(lastHitObj)

{

lastHitObj.renderer.material = originalMat;

lastHitObj = null;

}

}

if(Input.GetMouseButtonDown(0) && lastHitObj && !upgradePanelOpen)

{

focusedPlane = lastHitObj.GetComponent(PlacementPlane);

if(focusedPlane.isOpen && turretCosts[structureIndex] <=cashCount)

{

var newStructure : GameObject = Instantiate(allStructures[structureIndex], lastHitObj.transform.position, Quaternion.identity);

newStructure.transform.localEulerAngles.y = (Random.Range(0,360));

focusedPlane.myStructure = newStructure;

focusedPlane.isOpen = false;

cashCount = cashCount - turretCosts[structureIndex];

UpdateGUI();

}

else if(focusedPlane.myStructure !=null)

{

ShowUpgradeGUI();

}

}

// ---Gui

//Waves--

if(waveActive)

{

if(Time.time>=waveEndTime)

{

spawnEnemies = false;

if(enemyCount ==0)

{

FinishWave();

}

}

if(spawnEnemies)

{

if(Time.time>(lastSpawnTime + respawnInterval))

{

SpawnNewEnemy();

}

}

}

}

//---Waves

function ShowUpgradeGUI()

{

structureToUpgrade=focusedPlane.myStructure.GetComponent(Turrets\_Parent);

upgradeStructure = structureToUpgrade.myUpgrade;

if(upgradeStructure !=null)

{

upgradePanelOpen = true;

upgradeCost = structureToUpgrade.myUpgradeCost;

var upgradeName =structureToUpgrade.myUpgradeName;

saleValue = structureToUpgrade.mySaleValue;

upgradeText.text ="$"+upgradeCost+"";

saleText.text = "$"+saleValue+"";

CostCheckButton(upgradeBtn, upgradeCost);

upgradePanelTweener.Play(true);

}

if(upgradeStructure ==null)

{

upgradePanelOpen = true;

saleValue = structureToUpgrade.mySaleValue;

upgradeText.text ="$"+upgradeCost+"";

saleText.text = "$"+saleValue+"";

//CostCheckButton(upgradeBtn, upgradeCost);

upgradePanelTweener.Play(true);

upgradeBtn.transform.Find("Label").gameObject.GetComponent(UILabel).color = Color.red;

upgradeBtn.transform.Find("Background").gameObject.GetComponent(UISlicedSprite).color = Color(.5,.5,.5,.5);

upgradeBtn.collider.enabled = false;

}

}

function ConfirmUpgrade()

{

var spawnPos = structureToUpgrade.transform.position;

var spawnRot = structureToUpgrade.transform.rotation;

Destroy(structureToUpgrade.gameObject);

var newStructure : GameObject = Instantiate(upgradeStructure, spawnPos, spawnRot);

focusedPlane.myStructure = newStructure;

focusedPlane.isOpen = false;

cashCount = cashCount - upgradeCost;

UpdateGUI();

upgradePanelTweener.Play(false);

upgradePanelOpen = false;

}

function SellTower()

{

Destroy(structureToUpgrade.gameObject);

cashCount = cashCount + saleValue;

UpdateGUI();

upgradePanelTweener.Play(false);

upgradePanelOpen = false;

focusedPlane.isOpen = true;

}

function CancelUpgrade()

{

upgradePanelTweener.Play(false);

upgradePanelOpen=false;

}

function CostCheckButton(theBtn : GameObject, itemCost : int)

{

if(cashCount < itemCost)

{

theBtn.transform.Find("Label").gameObject.GetComponent(UILabel).color = Color.red;

theBtn.transform.Find("Background").gameObject.GetComponent(UISlicedSprite).color = Color(.5,.5,.5,.5);

theBtn.collider.enabled = false;

}

else

{

theBtn.transform.Find("Label").gameObject.GetComponent(UILabel).color = Color.green;

theBtn.transform.Find("Background").gameObject.GetComponent(UISlicedSprite).color = onColor;

theBtn.collider.enabled = true;

}

}

function SetNextWave()

{

//

waveLevel++;

difficultyMultiplier = ((Mathf.Pow(waveLevel,2))\*.005)+1;

respawnMin = respawnMinBase\*(1/difficultyMultiplier);

respawnMax = respawnMaxBase\*(1/difficultyMultiplier);

}

function StartNewWave()

{

UpdateGUI();

SpawnNewEnemy();

waveEndTime = Time.time + waveLength;

waveActive = true;

spawnEnemies = true;

}

function FinishWave()

{

Debug.Log("NextWave");

waveActive = false;

yield WaitForSeconds(5);

SetNextWave();

StartNewWave();

}

function SpawnNewEnemy()

{

//To choose an enemy from prefabs

var enemyChoice = Random.Range(0,enemyPrefabs.length);

var spawnChoice : int;

if(enemyPrefabs[enemyChoice].tag == "Air Enemy")

{

spawnChoice = Random.Range(0,flyerSpawnPoints.length);

Instantiate(enemyPrefabs[enemyChoice], flyerSpawnPoints[spawnChoice].position, flyerSpawnPoints[spawnChoice].rotation);

}

else

{

spawnChoice = Random.Range(0,groundSpawnPoints.length);

Instantiate(enemyPrefabs[enemyChoice], groundSpawnPoints[spawnChoice].position, groundSpawnPoints[spawnChoice].rotation);

}

enemyCount++;

lastSpawnTime = Time.time;

respawnInterval = Random.Range(respawnMin, respawnMax);

}

//Custom functions

function UpdateGUI()

{

for(var theBtnGraphic : UISlicedSprite in buildBtnGraphics)

{

theBtnGraphic.color = offColor;

}

buildBtnGraphics[structureIndex].color = onColor;

//Update HUD

waveText.text = "Wave:" +waveLevel;

scoreText.text = "Score:" +scoreCount;

healthText.text = "Health:" +healthCount;

cashText.text = "Cash: "+cashCount;

CheckTurretCosts();

}

function SetBuildChoice(btnObj :GameObject)

{

Debug.Log("Hello");

var btnName : String = btnObj.name;

if(btnName == "Btn\_Gatling")

{

structureIndex = 0;

}

else if(btnName =="Btn\_SAM")

{

structureIndex = 1;

}

UpdateGUI();

}

function CheckTurretCosts()

{

for(var i : int = 0; i <allStructures.length; i++)

{

if(turretCosts[i] > cashCount)

{

costTexts[i].color = Color.red;

buildBtnGraphics[i].color = Color(.5,.5,.5,.5);

buildBtnGraphics[i].transform.parent.gameObject.collider.enabled = false;

}

else

{

costTexts[i].color = Color.green;

if(structureIndex ==i)

{

buildBtnGraphics[i].color = onColor;

}

else

{

buildBtnGraphics[i].color = offColor;

}

buildBtnGraphics[i].transform.parent.gameObject.collider.enabled = true;

}

}

}

===============================================================================

PlayerBase.js

===============================================================================

var levelMaster : LevelMaster;

function Start ()

{

levelMaster =GameObject.FindWithTag("LevelMaster").GetComponent(LevelMaster);

}

function OnTriggerEnter ( other : Collider)

{

if(other.gameObject.tag == "Ground Enemy" || other.gameObject.tag == "Air Enemy")

{

Destroy(other.gameObject);

levelMaster.enemyCount --;

levelMaster.healthCount--;

levelMaster.UpdateGUI();

}

}

===============================================================================

AttackBots.js

===============================================================================

class AttackBots extends Enemy\_Parent

{

function Start () {

}

function Update () {

transform.Translate(Vector3.forward\*(forwardSpeed\*Time.deltaTime));

}

}

===============================================================================

Enemy\_Parent.js

===============================================================================

#pragma strict

var myCashValue : int = 50;

var levelMaster : LevelMaster;

var speedRange : Vector2 = Vector2(7.0, 10.0);

var forwardSpeed : float;

var health : float = 100;

var smokeTrail : GameObject;

var explosionEffect : GameObject;

var maxHealth : float = 100.0;

var explosionAudio : AudioClip;

function Awake()

{

levelMaster = GameObject.FindWithTag("LevelMaster").GetComponent(LevelMaster);

maxHealth = health;

forwardSpeed = Random.Range(speedRange.x, speedRange.y);

forwardSpeed = forwardSpeed \* levelMaster.difficultyMultiplier;

health = health \* levelMaster.difficultyMultiplier;

maxHealth = maxHealth \* levelMaster.difficultyMultiplier;

}

function TakeDamage(damageAmount:float)

{

health -=damageAmount;

var healthPercent = health/maxHealth;

if(health <=0)

{

Explode();

return;

}

}

function Explode()

{

audio.Play();

levelMaster.enemyCount--;

levelMaster.cashCount +=myCashValue;

levelMaster.scoreCount += (maxHealth+forwardSpeed\*levelMaster.difficultyMultiplier);

levelMaster.UpdateGUI();

Instantiate(explosionEffect, transform.position, Quaternion.identity);

Destroy(gameObject);

}

function Start () {

}

function Update () {

}

===============================================================================

Attack\_jet.js

===============================================================================

class Attack\_jet extends Enemy\_Parent

{

var height : float;

var heightRange : Vector2 = Vector2(6.0,9.0);

function Start () {

transform.position.y = Random.Range(heightRange.x, heightRange.y);

height = transform.position.y;

}

function Update () {

transform.Translate(Vector3.forward\*(forwardSpeed\*Time.deltaTime));

}

}

===============================================================================

Anim\_Speed.js

===============================================================================

var speedChoice : Enemy\_Parent;

function Start ()

{

speedChoice = GameObject.FindWithTag("Ground Enemy").GetComponent(Enemy\_Parent);

if (speedChoice.forwardSpeed>=4)

{

for (var state : AnimationState in animation) {

state.speed = 2.5;

}

}

else

{

for (var state : AnimationState in animation) {

state.speed = 1.5;

}

}

}

function Update () {

}

===============================================================================

CameraBehavior.js

===============================================================================

// The camera bounds

var mapMinX : int;

var mapMinZ : int;

var mapMaxX : int;

var mapMaxZ : int;

// Zoom limits for the camera

var mapMaxY : int = 9.5;

var mapMinY : int = 4;

var scrollArea = 7; // Defines the distance from the edge of the window that mouse scrolling starts

var scrollSpeed = 22; // Defines how fast the window scrolls

// Translates the camera

function moveMe(myDir, mySpeed) {

switch (myDir)

{

case ("Left") :

myVector = (Vector3(mySpeed,0,0) \* scrollSpeed \* Time.deltaTime);

break;

case ("Right") :

myVector = (Vector3(mySpeed,0,0) \* scrollSpeed \* Time.deltaTime);

break;

case ("Forwards") :

myVector = (Vector3(0,0,mySpeed) \* scrollSpeed \* Time.deltaTime);

break;

case ("Backwards") :

myVector = (Vector3(0,0,mySpeed) \* scrollSpeed \* Time.deltaTime);

break;

case ("Up") :

myVector = (Vector3(0,mySpeed,0));

break;

case ("Down") :

myVector = (Vector3(0,mySpeed,0));

break;

default : Debug.Log("Can't Move.");

}

if (InBounds(myVector))

{transform.Translate(myVector, Space.World);}

}

function Update ()

{

var mPosX = Input.mousePosition.x;

var mPosY = Input.mousePosition.y;

// Do camera movement by mouse position

if (mPosX < scrollArea) {moveMe("Left", -1);}

if (mPosX >= Screen.width-scrollArea) {moveMe("Right", 1);}

if (mPosY >= Screen.height-scrollArea) {moveMe("Forwards", 1);}

if (mPosY < scrollArea) {moveMe("Backwards", -1);}

// Do camera movement by keyboard

if (Input.GetAxis("Horizontal") < 0) {moveMe("Left", Input.GetAxis("Horizontal"));}

if (Input.GetAxis("Horizontal") > 0) {moveMe("Right", Input.GetAxis("Horizontal"));}

if (Input.GetAxis("Vertical") > 0) {moveMe("Forwards", Input.GetAxis("Vertical"));}

if (Input.GetAxis("Vertical") < 0) {moveMe("Backwards", Input.GetAxis("Vertical"));}

// Zoom Camera in or out

if (Input.GetAxis("Mouse ScrollWheel") < 0) {

moveMe("Up", .2);

}

if (Input.GetAxis("Mouse ScrollWheel") > 0) {

moveMe("Down", -.2);

}

}

// Checks to see if the camera would be in bounds after the move

// if not, it brings the camera back to the edge of the bounds

function InBounds (vector : Vector3) : boolean {

var answer : boolean = true;

if ((transform.position.x + vector.x) < mapMinX) {

transform.position.x = mapMinX;

answer = false;

}

if ((transform.position.z + vector.z) < mapMinZ) {

transform.position.z = mapMinZ;

answer = false;

}

if ((transform.position.x + vector.x) > mapMaxX) {

transform.position.x = mapMaxX;

answer = false;

}

if ((transform.position.z + vector.z) > mapMaxZ) {

transform.position.z = mapMaxZ;

answer = false;

}

if ((transform.position.y + vector.y) > mapMaxY) {

transform.position.y = mapMaxY;

answer = false;

}

if ((transform.position.y + vector.y) < mapMinY) {

transform.position.y = mapMinY;

answer = false;

}

return answer;

}

===============================================================================

PlacementPlane.js

===============================================================================

var isOpen : boolean = true;

var myStructure : GameObject;

function Start () {

}

function Update () {

}

===============================================================================

Projectile\_Cannon.js

===============================================================================

#pragma strict

var mySpeed : float = 40;

var myRange : float = 10;

var myDamageAmount : float = 2;

private var myDist : float;

function Start () {

}

function Update () {

transform.Translate(Vector3.forward\*Time.deltaTime\*mySpeed);

myDist += Time.deltaTime\*mySpeed;

if(myDist >= myRange)

Destroy(gameObject);

}

function OnTriggerEnter(other : Collider)

{

if(other.gameObject.tag =="Ground Enemy")

{

Destroy();

other.gameObject.SendMessage("TakeDamage", myDamageAmount, SendMessageOptions.DontRequireReceiver);

}

}

function Destroy()

{

Destroy(gameObject);

}

===============================================================================

Projectile\_Missile.js

===============================================================================

#pragma strict

var myExplosion : GameObject;

var myTarget : Transform;

var myRange : float = 10;

var mySpeed : float = 50;

var myDamageAmount : float = 20;

private var myDist : float;

function Start () {

}

function Update () {

transform.Translate(Vector3.forward \* Time.deltaTime \* mySpeed);

myDist += Time.deltaTime \* mySpeed;

if(myDist >= myRange)

Explode();

if(myTarget)

{

transform.LookAt(myTarget);

}

else

{

Explode();

}

}

function OnTriggerEnter(other : Collider)

{

if(other.gameObject.tag =="Air Enemy" || other.gameObject.tag =="Ground Enemy")

{

Explode();

other.gameObject.SendMessage("TakeDamage", myDamageAmount, SendMessageOptions.DontRequireReceiver);

}

}

function Explode()

{

Instantiate(myExplosion, transform.position, Quaternion.identity);

Destroy(gameObject);

}

===============================================================================

Turret\_Canon.js

===============================================================================

class Turret\_Canon extends Turrets\_Parent

{

var myProjectile : GameObject;

var reloadTime : float =1f;

var turnSpeed : float = 5f;

var firePauseTime : float = 0f;

var muzzleEffect : GameObject;

var myTarget : Transform;

var muzzlePositions : Transform[];

var turretBall : Transform;

var errorAmount : float = .001;

var aim\_Pan : Transform;

private var nextFireTime : float;

private var nextMoveTime : float;

private var desiredRotation : Quaternion;

private var aimError : float;

function Start () {

}

function Update ()

{

if(myTarget)

{

if(Time.time >= nextMoveTime)

{

aim\_Pan.LookAt(myTarget);

//aim\_Pan.eulerAngles = Vector3(0, aim\_Pan.eulerAngles.y,0);

turretBall.rotation = Quaternion.Lerp(turretBall.rotation, aim\_Pan.rotation, Time.deltaTime\*turnSpeed);

//CalculateAimPosition(myTarget.position);

//turretBall.rotation = Quaternion.Lerp(turretBall.rotation, desiredRotation, Time.deltaTime\*turnSpeed);

}

if(Time.time >=nextFireTime)

{

FireProjectile();

}

}

}

function OnTriggerEnter(other : Collider)

{

if(other.gameObject.tag =="Ground Enemy")

{

nextFireTime = Time.time+(reloadTime\*.5);

myTarget = other.gameObject.transform;

}

}

function OnTriggerExit (other:Collider)

{

if(other.gameObject.transform == myTarget)

{

myTarget = null;

}

}

function CalculateAimPosition(targetPos : Vector3)

{

var aimPoint = Vector3(targetPos.x+aimError, targetPos.y+aimError, targetPos.z+aimError);

desiredRotation = Quaternion.LookRotation(aimPoint);

}

function CalculateAimError()

{

aimError = Random.Range(-errorAmount, errorAmount );

}

function FireProjectile()

{

nextFireTime = Time.time + reloadTime;

nextMoveTime = Time.time+firePauseTime;

//CalculateAimError();

for(theMuzzlePos in muzzlePositions)

{

Instantiate(myProjectile, theMuzzlePos.position, theMuzzlePos.rotation);

Instantiate(muzzleEffect, theMuzzlePos.position, theMuzzlePos.rotation);

}

}

}

===============================================================================

Turret\_SAM.js

===============================================================================

class Turret\_Sam extends Turrets\_Parent

{

var myProjectile : GameObject;

var reloadTime : float = 1f;

var turnSpeed : float = 5f;

var firePauseTime : float = .25f;

var errorAmount : float =.001;

var myTarget : Transform;

var muzzlePositions : Transform[];

var pivot\_TiltL : Transform;

var pivot\_TiltR : Transform;

var pivot\_Pan : Transform;

var aim\_Pan : Transform;

var aim\_TiltL : Transform;

var aim\_TiltR : Transform;

var muzzlePosCount : int;

private var nextFireTime: float;

private var desiredRotation : Vector3;

private var aimError : float;

function Start () {

}

function Update () {

if(myTarget)

{

aim\_Pan.LookAt(myTarget);

aim\_Pan.eulerAngles = Vector3(0, aim\_Pan.eulerAngles.y,0);

aim\_TiltL.LookAt(myTarget);

aim\_TiltR.LookAt(myTarget);

pivot\_Pan.rotation = Quaternion.Lerp(pivot\_Pan.rotation, aim\_Pan.rotation, Time.deltaTime\*turnSpeed);

pivot\_TiltL.rotation = Quaternion.Lerp(pivot\_TiltL.rotation, aim\_TiltL.rotation, Time.deltaTime\*turnSpeed);

pivot\_TiltR.rotation = Quaternion.Lerp(pivot\_TiltR.rotation, aim\_TiltR.rotation, Time.deltaTime\*turnSpeed);

if(Time.time >= nextFireTime)

{

FireProjectile();

}

}

}

function OnTriggerEnter(other : Collider)

{

if(other.gameObject.tag == "Air Enemy" || other.gameObject.tag == "Ground Enemy")

{

nextFireTime = Time.time +(reloadTime\*.5);

myTarget = other.gameObject.transform;

}

}

function OnTriggerExit(other : Collider)

{

if(other.gameObject.transform == myTarget)

{

myTarget = null;

}

}

function FireProjectile()

{

nextFireTime = Time.time+reloadTime;

var m : int = Random.Range(0,muzzlePosCount-1);

var newMissile = Instantiate(myProjectile, muzzlePositions[m].position, muzzlePositions[m].rotation);

newMissile.GetComponent(Projectile\_Missile).myTarget = myTarget;

}

}

===============================================================================

Turret\_Parent.js

===============================================================================

var myUpgrade : GameObject;

var myUpgradeCost : int = 0;

var myUpgradeName :String = "";

var mySaleValue : int;

function Start () {

}

function Update () {

}

**Domain Lexicon**

During the designing of a tower defense game, more than one comprehensive domain had to be examined: gaming, game development, computer graphics game engines, user interface, content and logic. These examinations had been done both by doing research on existing systems and techniques, by doing some questionnaire, or from general knowledge. Most of the resulting terms are familiar to gamers, but they still should be reported.

A **Game** is the program to be developed, all of the system, is called a game. It is the resulting artefact user should play/uses, and **Player** refers to the targeted user of the game, the live person itself.

***Game Engine:***

It is the part of the system which does not interact with the user. However, it is the core of the all system. Does all needed calculations, maintenance of graphical, audible and interactive control objects and ultimately changes in other objects. Ideally, it is completely separated from game logic and game content.

**Animation:** Animations are the rapid display of a sequence of images of two-dimensional or three dimensional artwork or model positions in order to illustrate an effect.

**Collision:** A major term in game development, collision refers both the notion of “colliding” agents in the game, and the logical aspect of such collisions. Although it is a major concern of game engines, in tower defense games it is often trivial

**Draw:** Rendering all drawable objects by the game engine, it allows user to see actually all graphical aspects of the game.

**Game Time:** Elapsed time in the game logic. System time is not included; it refers the time considered to be spent in the game, not the real time.

**Grid:** In most systems, not only in games, a screen is separated to little sub-screens, to the smallest units which a user can take action on. It is usually a square, a hexagon or an octagon.

**Light:** A computer graphics term, light refers both the lightning effect on three dimensional objects and the source of light itself. Light allows user to see rendered objects in the three dimensional space.

**Real Time:** A genre of strategy games. In such games, all players (either artificial intelligence or human) play the game at the same time, decide and take action simultaneously.

**Rotation:** A computer graphics term, rotation specifically refers to the action of rotating an image according to a specified origin, to some extent. It is commonly used to illustrate stationary movements in two-dimensional games.

**Scaling:** A computer graphics term, scaling refers re-sizing an image on the screen.

Domain Lexicon

**Shadow:** A computer graphics term, a shadow is partly illuminated or un-illuminated areas in the three dimensional space. The renderer dynamically creates them.

**Sprite:** A computer graphics term, sprites are two-dimensional or three-dimensional images or animations that are integrated into a larger screen. Usually, they consist of a number of images or models which are displayed one after another and create an animated visualization.

**System Time:** Elapsed time in system creates the real time with game time and other environmental variables. System time increases as the calculations of the game increase and force the hardware.

**Texture:** May also considered as a game content term, textures are two dimensional image files that cover certain areas, such as a background or a model.

**Update:** A game development term, update refers to applying all changes in the game logic to the game objects, usually dictated by the state of a game.

**X Width:** A graphical term, defines the X coordinate of a drawable object in the space.

**Y Length:** A graphical term, defines the Y coordinate of a drawable object in the space.

**Z Depth:** A graphical term, defines the Z coordinate of a drawable object in the space.

**Effect:** Effects are all kind of experience enrichment content, either graphical or audible.

***Game Control:***

This part of the system deals with user interaction and its effects on both game engine and game logic

**Building Menu:** Visible and reachable on game-play and preparation screen on a different panel; building menu allows player to browse towers (s) he can build and choose a tower to build.

**Click Area:** A user interface term, click areas are defined to specific buttons and correspond to a certain area, which allows system to decide when a button is clicked.

**Game-play Screen:** A state of a tower defense game where game runs and enemies attack to the player’s character. The most dynamic of a game, this part can be sums up the conventional meaning of playing with the “Preparation Screen”.

**Main Menu:** The first user interactive state of a game. Unlike other computer programs, nearly all computer games start with a main menu and allows user to navigate through states by using large buttons.

**Map Editor:** A sub-system in the game, level editors allows an interface to users add contents (specifically, a map) to an existing game.

Domain Lexicon

**Options:** Reached either from main menu or after starting a play, this menu allows user to change settings of the game. It usually allows user to change graphics settings, sound settings, control settings and game-play settings.

**Preparation Screen:** A state of a tower defense game which is just before game-play settings. In this state, enemies do not attack, and there is no movement. User prepares his actions, and usually can do what s/he can do during game-play screen.

**Tower Info Menu:** Visible and reachable on game-play and preparation screen, tower info menu is visible only when a built tower is selected. It gives information about towers status, values, provides a button to sell it or upgrade it.

***Game Logic:***

With game engine, game content, and game control creates the sum of all the game. Game logic is the model of the game, and defines what game should do, and how it should interact with the user. Do calculations and changes that are related with the rules of the game, however is not interested how in the end they will be presented or implemented in the system. Ideally, game logic is completely separated from game content and game engine.

**Agent:** An agent refers to an object in the game that is capable to take an action, by examining other agents and the environment.

**Attack:** A common term in gaming, attack refers any type of action which aims to do any kind of damage or hindrance. In tower defense games, attacking refers the whole action of aiming a creature in range, shooting a bullet to it, hit or miss action of a bullet, applying damage and damage type to creatures related.

**Attack Range:** Distance, which a tower can aim and shoot to, starts from its center.

**Attack Speed:** Attack speed is a value of an attacking agent, which decides the minimum time between two following attacks

**Building:** The most common and needed player action in tower defense games meaning creating fortifications (usually towers) is that preventing enemy.

**Bullet/Projectile/Missiles:** Although word of bullet mean a single bullet in general gaming usage, tower defense games refers bullets as agents that are sent by the towers to the enemy creature.

**Bullet/Projectile/Missiles Speed:** Bullet speed is a value of an attacking agent, and the bullet agent, which decides how much space a sent bullet will parse in a unit time.

**Continuous Damage:** A damage type which is not common in tower defense games, “continuous damage” type attacks continue to decrease applied creatures' hit point even after it is applied to the target. Fire damages are an example to that.

Domain Lexicon

**Creature:** Also referred as “monster” or “soldier”, creatures are the tools of enemy to win the game in tower defense games. They move through the path, and wish to survive attacks from towers. They have no capacity to attack or to show any sentient resistance.

**Creature Speed:** Creature speed is a value of an enemy creature agent, which decides how fast it can move through the map.

**Damage Type:** Damage types are values that are stored in attacking towers and their bullets, to apply a certain type of damage. Damage types change the applied creatures, any possible hindrance to these creatures, damage that applies to these creatures.

**Enemy:** Opponent of the player. In gaming, it corresponds to all possible opponents and competitions, however tower defense games are single player, with very strict rules on player's and opponents role. Thus, enemy is the opponent of the player, which aims to finish the lives of the player.

**Enemy Base:** A tower defense game term, referring to the starting point of enemy creatures.

**Environment:** A term related to game logic and artificial intelligence, environment is the sum of all objects that are not capable to take an action and is in game logic itself.

**Game Artificial Intelligence:** A sub discipline of artificial intelligence, game AI is very different from academic discipline of artificial intelligence. Even to some, it is wrong to state that artificial intelligence is used in games. Artificial intelligence in games aims to achieve an enjoyable and competitive agent behavior in the game.

**Game Level:** In tower defense games, a game level starts when player finishes preparation screen and ends when game-play screen finishes either with success, or failures. A resulting success allows user to pass a level and start a new one. Usually, all game levels consist of a certain type of enemy.

**Ground:** Remaining parts of the map, excluding paths, home base and enemy base. Consists the majority of the map, and user is allowed to build on such areas.

**Gold/Cash:** Also referred as “coin” or “money”, it stands for the resource itself that is used in the game. It comes from the notion of playing a game in the old ages.

**Hit:** A common term in gaming, a hit means a successful attack. In tower defense games, it is one of possible ending states of a sent bullet; a hit means a bullet has successfully met with its target.

**Hit Damage:** A damage type which is common in tower defense games, “hit damage” type attacks only affect the targeted creature and decrease its hit point.

**Hit Point:** A common term in gaming, hit point is a value that decides how much damage an agent can take before being eliminated. In tower defense games, only enemy creatures have hit points.

Domain Lexicon

**Home Base:** A tower defense game term, referring to the target point of enemy creatures, which is the “home” of the player.

**Importer:** In game systems, objects that are related to the game logic are subject to change, either during development or after release. It is also considered as a good practice of game development to separate game entities from the system as much as possible, allowing variations and rapid modification. An importer deals with such external files that keep needed parameters for game objects. They usually read a file, and allow game logic to create related objects accordingly.

**Interest:** Some tower defense games aim to reward users by not spending many resources and try to motivate others to challenge them to optimize resources. By having an interest rate, the gold player stores in her/his treasury will be on his interest.

**Kill:** A common term in gaming, killing means eliminating a creature of opponent, by decreasing its hit point to zero or below.

**Life:** Interchangeably used to define the notion of “life”, or the number of “lives” a player has. A game starts with a number of lives, and ends when player lost all her/his lives. Extra life can be gained by spending gold or by having specific traits.

**Manhattan Distance:** A term related to game logic and artificial intelligence, Manhattan distance is the shortest path between two points by following only moving on one coordinate at a time.

**Map:** Sum of all grids, where all game runs and opponents compete with each other. A map consists of home base, enemy base and path. Towers and creatures are placed (and move) on the map; however is not part of the map.

**Miss:** A common term in gaming, a hit means an unsuccessful attack. In tower defense games, it is one of possible ending states of a sent bullet. A miss means a bullet has not successfully met with its target.

**Path finding:** Algorithms and methods to define a path for game agents. Part of game artificial intelligence, path finding is a major concern in game development. In tower defense games, however, it is a trivial task.

**Path:** A static way for enemy creatures to move through screen, from enemy base to home base, defined by waypoints. User cannot build towers on paths.

**Profile:** Sometimes referred as “account”, a profile is recordings of a specific player. A player may have more than one profile, and can separate distinct plays from one another.

**Slow Damage:** A damage type which is common in tower defense games, “slow damage” type attacks decrease the speed of applied creatures. Cold damages are a common example to that.

Domain Lexicon

**Splash Damage:** A damage type which is common in tower defense games, “splash damage” type attacks not only affects to aimed target, but other creatures in the range as well.

**Splash Range:** A term related to splash damage, splash range value is the distance that splashed bullet damage can reach at most.

**Tower:** Basic tool that player has, towers are the foundations of a tower defense game. They have a price (usually in terms of gold or coin), needs a clear area to build on, has a range, an attack type, attack speed and damage value. They can be upgraded to increase one or more of these attributes**.**

**Tower Level:** State of a built tower, starting from zero to either infinity or a magic number. As it increases, a tower becomes more effective. Tower levels are increased by upgrading.

**Turn Based:** A genre of strategy games. In such games, players (either artificial intelligence or human) play the game by following turns, one after another. Actions and decisions do not overlap.

**Upgrade:** A player action, which allows him to make more benefit of a tower by spending some resource (gold or coin). An upgraded tower has an increased level, and usually further upgrades are more expansive. By upgrading, tower may simply have increased attack range, speed or damage; or may evolve into something else completely.

**Waypoint:** Waypoints define a set of points which defines a path for the enemy creatures. It starts from enemy base, and is aimed to home base.

**Sell:** A common gaming term, sell refers to the action of sacrificing a possessed agent of the player to gain some resource on it. In tower defense games, selling refers removing a built tower from a map and gaining some degree of gold from this action.

***Game Content:***

Part of a game system, which includes all non-implementation and non-logical (in the end, that is not related to the source code directly) features. Textures, sprites, animations, sounds, music themselves (not their implementation) are the most known examples of game content. In addition, aspects that are related with the game logic, but not the playability, are also fall into this category.