FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Implementation of a cross-platform strategy multiplayer game based on Unity3D

Igor Galochkin

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Implementation of a cross-platform strategy multiplayer game based on Unity3D

Entwicklung eines plattformunabhängigen Strategiespiels mit Unity3D

Author:	Igor Galochkin
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils T. Kannengießer, M.Sc.
Date:	January 15, 2013

I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, 10th of January, 2013 Igor Galochkin

Abstract

The thesis describes development of a multiplayer real-time strategy game "Fantasy Arena" for a range of operating systems and devices with focus on Android- and iOS-powered smartphones and tablets. The game uses Unity3D 4.x Pro game engine. The goal of the project was to provide a playable technical demo implementing the main features of the game.

The thesis consists of 3 chapters. The first chapter provides background information about games development with focus on real-time strategy games and mobile games. The second chapter describes the design of the "Fantasy Arena" game, moving from a description of the game and explanation of game design choices to the listing of elicited requirements and constraints. The third chapter describes the game's implementation, starting with overall system design, then covering some interesting or technically challenging features, and ending with testing techniques. The thesis concludes with a look back at the project's planned and actual timelines, an evaluation of the project's results and an outlook.

Table of Contents

Introduction

The development of computer games has since end of the 2000s been shifting from desktop computers and gaming consoles to mobile phones and tablets with touch-sensitive screens. The reason for this is that smartphones are at the moment of this writing (end of 2012) replacing the older Java-based cell phones worldwide, thus the smartphones user base is rapidly growing.

The number of smartphones is projected to reach 2 billion by 2015 [33]. These devices have powerful graphical capabilities that allow them to run games that are technically comparable to those of mid-late 1990s for desktop computers. Application stores (AppStore for iOS and GooglePlay for Android) allow users to easily download and install these games. Integrated payment systems allow a customer to seamlessly buy games or make in-game purchases. All these factors make smartphones a strategically vital platform for game development studios. Many game programmers have to switch from older platforms like Windows and older game development APIs like DirectX to new mobile platforms.

Unity3D is currently the most widely used game engine for development of games for mobile platforms[1]. It supports almost all platforms on which games are played today: desktops, laptops, gaming consoles, web browsers and smartphones. This thesis describes development of a game "Fantasy Arena" based on Unity3D 4.x Pro game engine.

The market of smartphone games was initially dominated by games for casual players - players who don't or very occasionally play games on a computer and don't want games to be challenging. However, games on mobile platforms started their evolution the same way they did on desktop computers but at a much faster rate. Games are getting more complex, with more content and 3D graphics. As the casual gamers become more experienced in gaming, they ask for more challenging games, closer to those hardcore gamers play on computers. Genres common on desktop games attempt to conquer mobile platforms and adapt to the limitations these platforms impose.

One such genre is that of the Real-time strategy games (RTS) where the player controls armies, builds bases and defensive towers and gathers resources in real time with the goal to capture the whole map and destroy opponents. This genre was very popular in late 1990s, and slowly became a niche market in 2000s and early 2010s. On mobile platforms there is so far only a handful of games in this genre, with varying success.

The game developed for this thesis is an RTS game. It is also a multiplayer game: many players can play together on the same battlefield from their devices. Due to the wide range of platforms the Unity3D game engine supports, the game is also cross-platform: a player with an Android device can play against a player with iPhone and a player with a Windows desktop PC, all on the same virtual battlefield.

[1] In May 2012 53% of mobile game developers were using Unity3D game engine for their games. [34]

Development of a game in the RTS genre, especially a multiplayer one, usually requires an entire game studio with separate departments for art, design, testing, administration, and at least a few programmers. They would be expected to work full-time on the game for 1-3 years. This bachelor thesis, taking only 1 developer and about 4 months full-time obviously could not produce any competitive commercial product. Its goal was to provide a proof of concept: a playable technical demo which can be developed further with involvement of a larger team.

The first chapter of the thesis starts with giving an overview of game programming and the specifics of games as software. It moves on to RTS games and how these games are specific from the point of view of the developer. Then multiplayer games are described with focus on their development challenges. The chapter then continues with presenting the challenges game designers and game programmers face when developing games for mobile devices and explains how these challenges are important for RTS games. This overview should provide enough information to understand the context in which the "Fantasy Arena" game was designed and developed. The first chapter concludes with a short overview of the Unity3D engine.

The second chapter describes the design of the game. It starts with providing the general ideas which where chosen as the starting point in the design. It then follows how these ideas were concretized and broken down into smaller design decisions about various features of the game as the different limitations of the mobile platforms were considered. Then a short overview of the resulting game design is given. The chapter concludes with a list of elicited requirements and limitations on the game's architecture.

The third chapter handles the implementation of the game based on the Unity3D game engine. First, it highlights the overall project structure and the organization of game assets and scripts. Then it describes various enhancements and additional scripts which had to be written to streamline development of the game with Unity3D. All these scripts can be reused in other games. The chapter continues with a look at selected features of the game which were technically challenging or were otherwise interesting in their implementation. In the end of the chapter the testing techniques are described.

The thesis concludes with a look back at the project's planned and actual timelines. The achieved results of the project are evaluated. An outlook is provided with the listing of the features which are planned for further development.

Chapter 1. Background information

This chapter gives an overview of computer games, how they are developed, what the real-time strategy (RTS) genre is and how this genre is evolving to succeed on mobile games market. Common software development challenges are described as well as typical solutions which developers use when developing computer games and in particular, RTS games, games for mobile devices with a touchscreen and multiplayer games. This should provide background information to see which challenges the "Fantasy Arena" project had to face. The chapter concludes with a short description of Unity3D game engine which was chosen for this project.

1.1. Development of computer games

A *computer game* or a *video game* is an interactive application with the main purpose of entertaining the user (player). Initially games were developed by computer scientists primarily to entertain themselves and their colleagues. In late 1970s, with immense increase in computing power, graphical capabilities and mass production of personal computers, games development became an industry. Competition between developers and rapidly increasing expectations of players forced developers to make games as computationally complex and as graphically appealing as was allowed by the average hardware of the end user.

The 1990s saw a vast improvement of the presentation of games: from a palette of just 4-8 colors and pixelated characters in the beginning of the 1990s, evolving into 32bit-colored masterpieces of 2D graphics by the late 1990s. A transition from 2D to 3D graphics quickly followed. Today a commercial game takes 2-3 years development time and 20-50 developers.

In late 1990s, with worldwide adoption of cell phones, a part of development effort of game companies got focused on making small, low budget 2D games for these new devices. These games weren't really successful due to limited distribution possibilities, insufficient user acceptance of payment methods, extreme fragmentation of hardware and low performance of the Java MIDP platform which cell phones used for games.

In 2007 a new device called iPhone by Apple Inc. turned the peripheral niche of mobile games into an extremely profitable space for both small indie developers and large companies. Apple provided a single distribution channel (AppStore) which the user could access directly and easily from the device, a stable standardized development platform, and a quickly growing audience of players who were ready to pay for games.

The iPhone runs an operating system iOS which is based on Apple's operating system Macintosh for laptops and desktop computers. A new term, "smartphone", was coined to describe mobile phones with an approximately 5-inch large touch-sensitive display and more powerful graphical and computational capabilities than a usual mobile phone.

In 2009 a competing operating system Android by Google Inc. was released and by

2011 outgrew iOS in popularity due to being open-source and less restrictive. The number of games developed for smartphones grew at an unprecedented rate during 2009-2012 (Figure 1). At the moment of this writing mobile games are the only growing market of video games while all other markets (desktop games, gaming consoles, web-browser games, games for handheld devices) are either stagnating or shrinking.

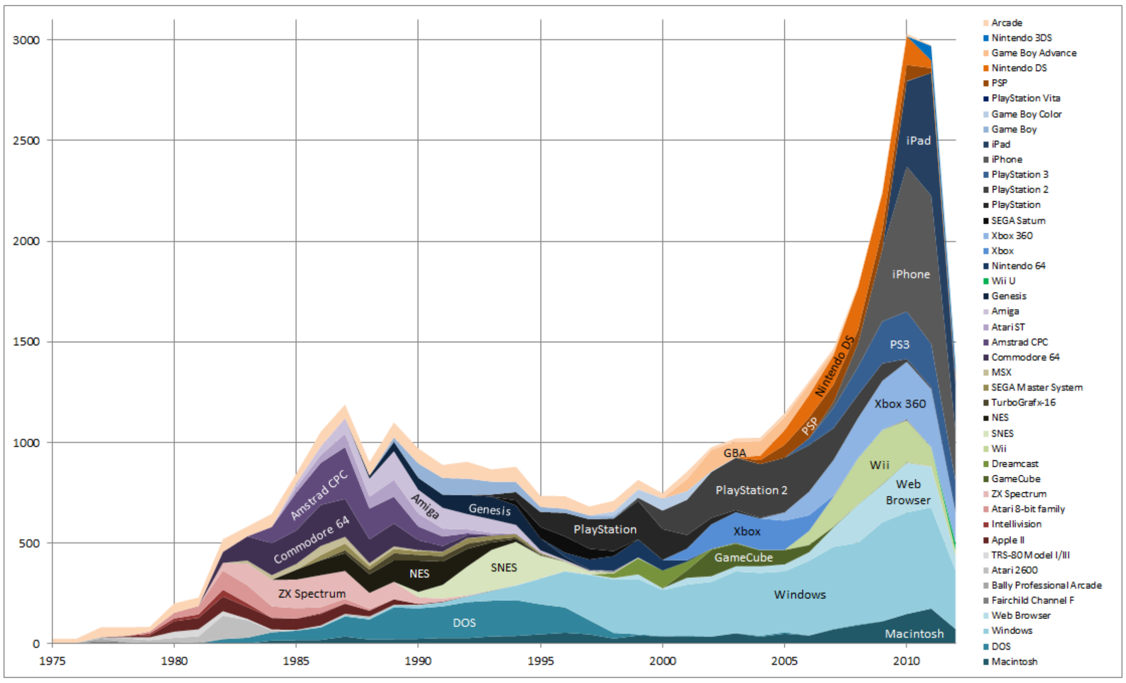


Figure 1. Number of major commercial games released per year on various game platforms (excluding Android) in 1975-2012 [15]

Development of a computer game requires collaboration of developers in many areas including programming, 2D and 3D art, music and sound effects, game design and testing. Programming tasks while developing a game are divided into a few areas including AI & gameplay, 3D graphics, networking, tools and physics.

Initially games were designed by programmers themselves: they were both thinking up the idea of the game and implementing it. This is still the case for indie games and games done by very small teams (2 or 3 developers). Since the 2000s the games have become so large in content that a need manifested to have special team members doing game design only.

Game designers write the *design document* of the game - a large document which in the games industry acts as a functional specification document.

Game designers also fill the game with content: populate levels with monsters, tweak the game rules and parameters of in-game elements and write in-game texts (for example, conversations of characters, tutorials, quest assignments). With *game content* or *game resources* programmers refer to images, sounds, in-game texts, 3D models, scripts and other assets which are specific to the currently developed game and are usually produced by non-programmers.

1.1.1. Common development challenges

Real-time systems with instant response

Games must provide feedback to the player immediately, without any noticeable delay. In most games (with an exception of some turn-based games and puzzles) the situation on the screen changes in real time and the user has to react fast to succeed with the game objective. If his actions are delayed due to lags (the game takes too much time to calculate something, load data or resources, receive update from the game server), the user considers this as an unfair impediment to achieving the game objective, gets frustrated and will likely stop playing.

Solution: before the game is released, it is extensively tested to ensure that the refresh rate of the frame buffer (measured in frames per second or FPS) never drops too low. A drop of FPS to less than 10 will be noticed by the player and perceived as sudden freezing or hiccups. Algorithms used in games must be efficient.

On the other hand, efficient code is often hard to understand and to write. The common practice is thus to first write easy-to-read and maybe inefficient code. Then, after lags or low FPS have been discovered during testing, one would run the game using a *profiler* - a tool inside the development environment which measures how long execution of each function or method in the code takes.

The profiler allows to quickly spot so-called "bottlenecks" in the code - functions or places in code which are executed very often but aren't optimized. Optimizing the game starts with optimizing this frequently called code which usually solves performance issues without wasteful efforts to optimize everything.

In cases when optimization is not possible (e.g. loading resources of the game from the disk can't be done faster than the hardware allows) such long-lasting operations are done in advance. For example, all resources belonging to the same area in the game world are loaded at once while a special loading screen with a progress bar is shown to the player.

If loading screens can't solve the problem or aren't allowed, resource-consuming operations can be done in a separate thread. For example, pathfinding (seeking for a way from one point to another) can be done in a separate thread and notify the main thread after it's done.

Alternatively, special algorithms may be developed which can run in small chunks of time on the main thread, store intermediate results to fetch them later, when algorithm is given time again.

Frequent changes of requirements

Game designers usually only have a vague idea of the game they want to see developed. As creative people, they often do not plan much in advance and start the project with insufficient or hardly any documentation. As soon as parts of the game get implemented by programmers, testers and designers start playing their game and will often find that some features turned out to be not as fun as they expected.

Also, as game comes closer to release, many people in the company begin to like it and, as avid gamers themselves, start getting ideas how the game can be made even better. If this process it not controlled by management, the game may never get released because of endless change requests (so-called "feature-creep").

In case of online games or games for mobile platforms a released game only starts its lifecycle at the moment of release. It keeps getting updated for years, changes happen inevitably due to requests of marketing department, wishes of players, company policy and other factors. Massive multiplayer online games (MMO) evolve constantly with new features and content added every month.

Solution: Since games development is an extreme case of agile development, a higher than usual discipline of coding must be used. Agile development philosophy requires that programmers write clean code - code which remains understandable and nicely organized no matter how much the system changes.

Changes in the requirements (for example, rules of the game) must never be implemented as "quick and dirty" fixes or "hacks". Instead, the architecture of the code should change each time a change is made and look as if it were originally done for the current requirements. System architecture must be modular and extensible.

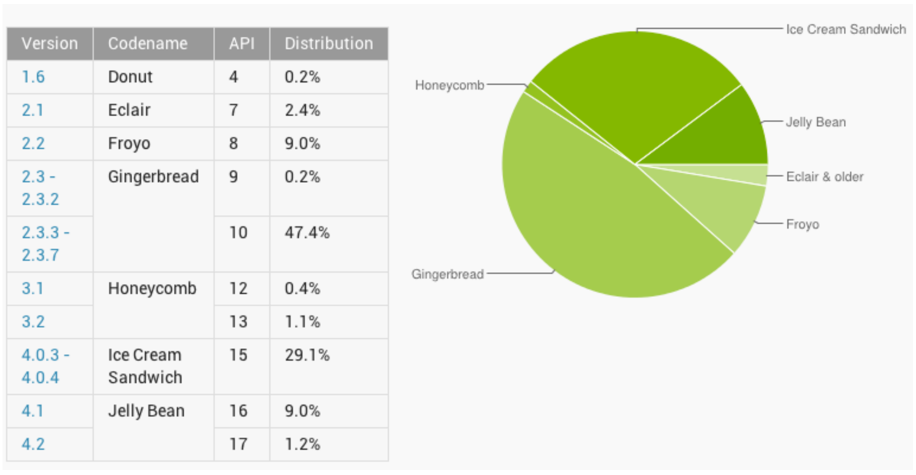
Fragmentation of target platforms

On desktop computers running Windows OS a common problem has been that the developer of a computer game can not test the game on all possible configurations the players of the game may have. With iOS this problem was effectively solved: by end of 2012 there were still just a handful of popular devices running iOS.

With Android however device fragmentation is immense. There are various versions of Android on the market (Figure 2). Also little is standardized as to what screen size, pixel density (Figure 3) and hardware buttons it may have and where those buttons are located as well as what additional features the device may support. Many of the low-budget devices are rushed into market without sufficient testing, which results in unstable performance, crashes, hardware errors which the users sees as bugs of the game he is playing. Errors may be caused by libraries the game uses, like AdMob, GoogleAnalytics or in-app billing.[2]

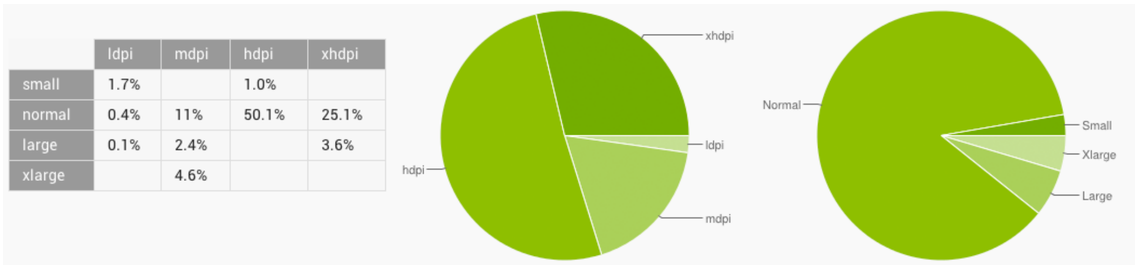
Solution: games are usually tested only on the most widespread devices or configurations. In case new major devices come out, the game is re-tested on them and, if needed, updates or patches are released.

2 Author's experience with publishing apps on Google's Play Store shows that about 80% of all crashes in the field have no relation to the game itself and happen due to bugs in libraries, in hardware or the operating system.



Version	Codename	API	Distribution
1.6	Donut	4	0.2%
2.1	Eclair	7	2.4%
2.2	Froyo	8	9.0%
2.3 - 2.3.2	Gingerbread	9	0.2%
2.3.3 - 2.3.7		10	47.4%
3.1	Honeycomb	12	0.4%
3.2		13	1.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	29.1%
4.1	Jelly Bean	16	9.0%
4.2		17	1.2%

Figure 2. Distribution of Android OS versions on the market, state Jan 3, 2013 [16]



	ldpi	mdpi	hdpi	xhdpi
small	1.7%		1.0%	
normal	0.4%	11%	50.1%	25.1%
large	0.1%	2.4%		3.6%
xlarge		4.6%		

Figure 3. Distribution of screen sizes and densities of Android-powered devices, state Oct 1, 2012 [16].

Vast volume of game resources

Games typically operate with large amounts of images, sounds, configuration files and other data. Such resources are usually produced in parallel with the code. Resources get updated and changed. Changes are usually done by non-programmers and can create bugs in runtime. For example, a missing parameter in an XML file configured by a non-technically-savvy game designer will cause a crash in runtime because some texture will not be loaded. Figure 4 present typical dynamics of game resources on a large commercial game project during its development.

Solution: the game must do extensive runtime checks of any loaded data which gets configured by game designers or artists. If an error is found, the game should show it as a popup or in some other noticeable way with directions how to fix the error.

Version control systems have to be used not only for code, but also for all resources including images, sounds, and configuration files. Non-programmer members of the team have to be taught to name resources consistently, avoid creating multiple copies of them and instead use version control systems.

For very large projects (for example, MMO games) special "commit hooks" may be designed which check resources for validity at the moment a team member attempts to commit them to the repository and disallow commit if a resource breaks game configuration.

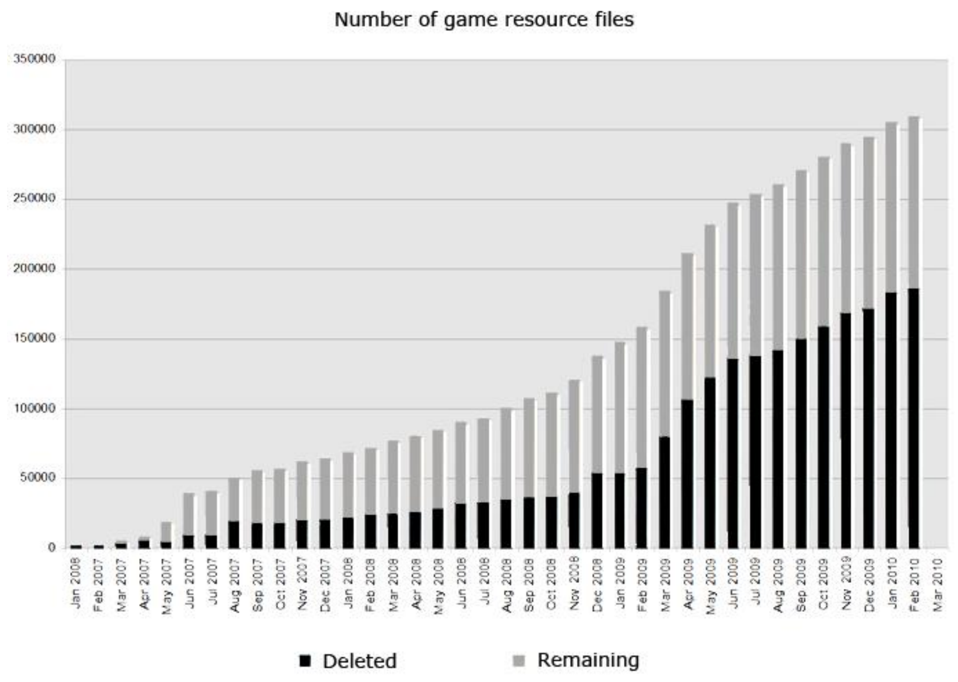


Figure 4. Dynamics of the number of game resources in the MMORG game Allods Online by Nival Interactive during development of the game [17].

1.1.2. Common programming techniques

Game engine

A game engine is a *middleware* (a large and complex software package or library) which implements common functionality used for games.

Powerful commercial game engines are usually cross-platform: they hide the underlying operating system completely from the game developer and provide an application programming interface (API) which is more suited for developing a game.

Game loop

A game application usually has a main loop which keeps repeating as long as the game is not finished. When the user chooses to exit the game, the loop finally finishes and the main method returns control to the operating system. The main game loop typically lies deep inside the game engine. The loop usually does the following steps on each update (Figure 5):

1. read user input
2. update the game world according to user input, update AI and physics in the game
3. render the game world

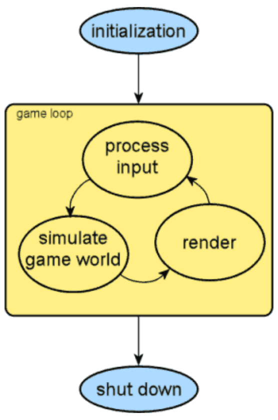


Figure 5. Typical flow of the game (game loop).

Finite state machines

AI of monsters, units or other semi-autonomous entities in the game is quite often implemented as a Finite state machine (FSM). Depending on the current state, the actor would behave in some way specific to that state. Such organization of AI is also easy to understand for players and game designers.

In some games the player can order game actors to switch to some state. For example in World of Warcraft a hunter can order his hound to stay in the defensive mode and only attack enemies which attack the hunter.

Scripting

A game typically has a big executable file which is compiled from a large body of C++ code. Compilation of a game may take up to half an hour even on a powerful development machine, so developers attempt to minimize the need to compile.

On the other hand, game designers, while testing the game and doing small tweaks in rules and other parts of game content, can work much faster if they can immediately see the effect of the changes they make.

To meet this need, programmers attempt to extract all logic and data which can potentially change a lot during development to separate files (scripts) which don't have to be compiled and ideally even don't require a game re-start to apply the changes. Scripts are written in interpretable languages like Lua, Python and JavaScript with simple syntax understandable for non-programmers.

Programmers strive to separate code common to all games from code specific to the currently developed game. They keep generic code in the game engine while game-specific code is produced in form of scripts.

1.2. Real-time strategy games (RTS)

Computer games are classified by their genre. Common genres are action games, puzzles, role-playing games, arcade games, strategy games, and racing simulators. Real-time strategy is one of the popular genres which appeared in the beginning of

1990s, reached it's zenith by the beginning of 2000s and in the recent years has somewhat lost it's popularity.

1.2.1. Game elements

In an RTS game the player acts as a commander of an army with a goal of defeating opponents by destroying their armies and capturing territory. Often the player can gather resources (ore, gold, wood) which are then used to produce new military units (tanks, archers, or fantastic creatures like goblins) and build up a base with defensive structures.

Units under the player's command have a specific type (tanks, archers, goblins) which defines their parameters (for example, speed of movement, range of attack, power of attack and armor). The player of such a game would typically get entertained by analyzing the parameters of available unit types in the game, the layout of the world map, positions of his and enemy structures and defenses and try to find a strategy which would allow him to destroy all enemies or reach other required objective without losing his army.

The main difference of RTS games from classic board games like chess is that the army is controlled in real time. There are no turns or pauses. As soon as a unit receives an order to move somewhere, it will start moving immediately and arrive to the destination after some time. During that time, the player can tend to other units and give them orders. RTS games give a much more realistic, immersive feeling of commanding an army than board games.

To be able to effectively control a large army, the player views the battlefield either from above ("top-down view") or from a high angle. RTS games use an isometric camera (distant objects don't look smaller than closer ones) which can usually be "scrolled" (moved around the map) by just pushing the mouse cursor to the screen edge.

Also, to select a lot of units at, the player would usually drag the mouse diagonally over an area to select all units inside the rectangle based on the drawn diagonal ("*dragbox*"). Double-click on a unit in some games also selects all units of the same type around the clicked unit.

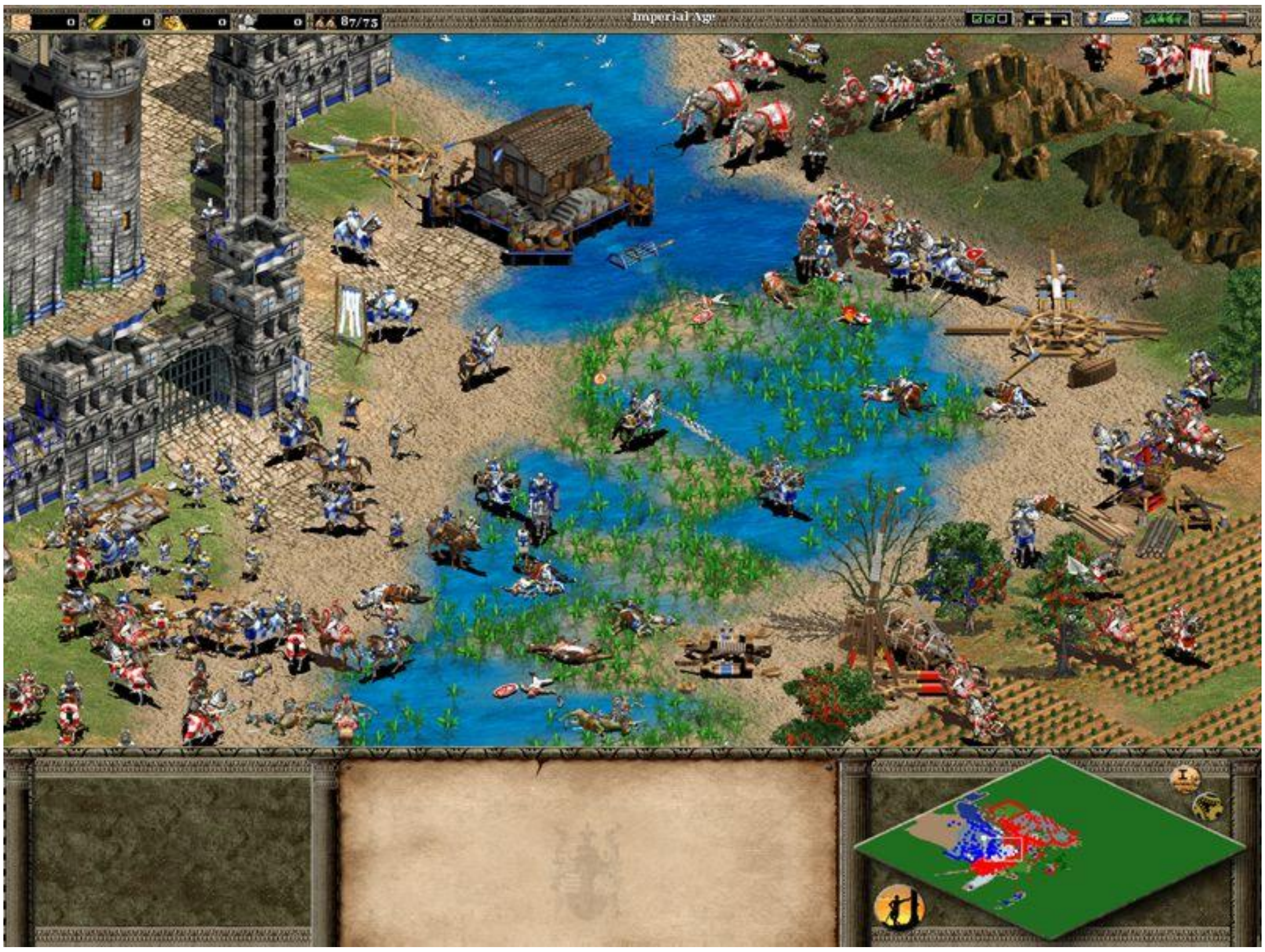


Figure 6. RTS game Age of Empires 2 by Microsoft, 1997 [18].

Figure 6 shows a situation in the RTS game Age of Empires 2 by Microsoft where players command Medieval armies. The player controls blue units and defends his castle to the left. The enemy (red) is attacking with horsemen, camel riders, elephants and trebuchets. The minimap in the bottom-right corner of the screen shows unexplored areas of the world with green, enemy units with red dots and player's units with blue dots. Currently viewed part of the world is shown with small white rectangle.

Real-time strategy games stem from 1992's game “Dune 2” [19] for DOS by Westwood studios and 1990's game “Herzog Zwei” [20] for gaming console Sega Megadrive by TechnoSoft. In late 1990s RTS games were typically played over local area network (LAN) in computer clubs, at home with friends or with random opponents over the Internet. The market of RTS games became saturated in early 2000s because very many titles were released with the same game mechanics, and most of the fans were already bored with the concept.

In the 2000s, games in another genre, the role-playing games (RPG) gained popularity, especially massive multiplayer ones with World of Warcraft [21] by Blizzard Entertainment as the most successful title. Game designers discovered that RPG games keep players engaged for a long time due to the slow build-up of main character's parameters which is common to the genre. The main character typically receives "experience points" (EXP) in such games. After a certain amount of EXP is accumulated, the character gains a level ("levels up"). The higher the level of the character, the more abilities it can perform (cast more spells or perform more types of swings with his sword), the better weapons and armor it can equip, the stronger it is. The player gets a feeling of an unfinished task which stays during the whole game while the main character progresses through the plot until the main objective of the game is complete and the game is finished.

In 2000s, to keep up with changing tastes of players, RTS games started adopting features from role-playing games. The player would now level up his units, some units became more powerful and get carried over to next battles.

1.2.2. Common challenges

Pathfinding

Pathfinding is the task of finding a path for a unit which it should follow to arrive at the requested destination without colliding into other units, buildings, world objects and stepping on impassable terrain. With lots of units moving on the map it becomes resource-intensive to calculate paths for all units simultaneously.

Solution: in some games, when the player issues a movement order to a group of units, these units are placed into a formation. The formation would move as a single unit at the speed of the slowest unit in the group. This reduces the pathfinding task for a lot of units to a task for just one unit, but of a larger size (or the formation may change shape when moving through narrow places).

The player expects the units to start moving immediately after they are given an order. Pathfinding algorithm however may require time to complete. To eliminate visual lag, a special technique may be used: the unit would start moving along a straight line in the direction of the target spot and keep moving in that direction until the pathfinding algorithm has delivered a result. After that the unit moves along the found path if there is one.

Strategy games typically use a grid-based map which allows to easily build a graph and run a pathfinding algorithm on it (for example, *A* algorithm [26]*). For games without such grid, the game designer can put the nodes and the edges of the pathfinding graph manually for each map.

Alternatively the graph can be generated automatically by laying a grid over the terrain and checking which nodes are still outside obstacles.

Also Unity3D engine offers a NavMesh feature which builds the navigation graph by analyzing the geometry of the scene.

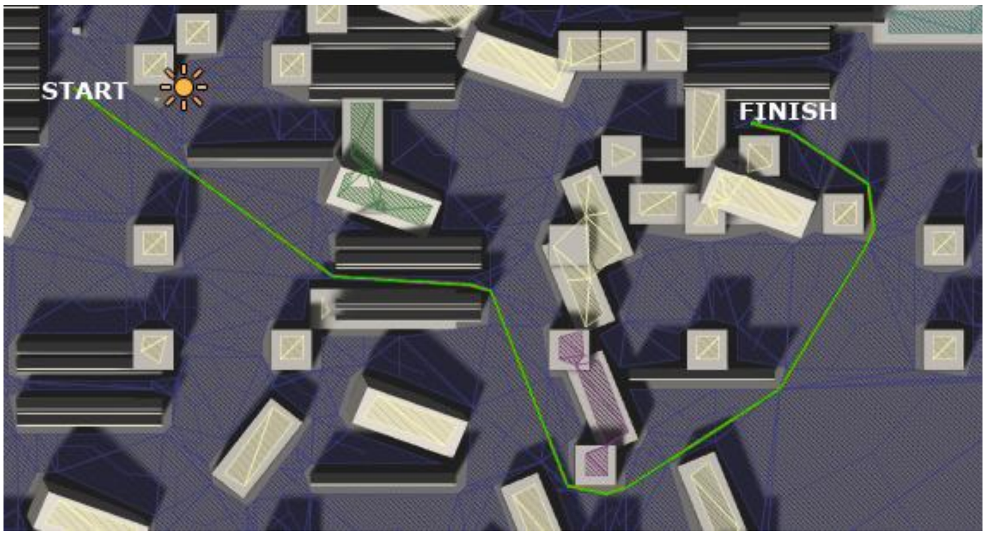


Figure 7. Path produced by the A* algorithm on an arbitrary scene with obstacles using NavMesh in Unity3D game engine with Astar Pathfinding plugin

AI of computer opponents

A strategy game is perceived by the player as a competition to outsmart an opponent which has equal resources. Unfortunately, the rules of an RTS game are usually so complex and change so much during development of the game that building an intelligent opponent which could compete with human intellect becomes impossible.

Solution: to keep the game challenging, the AI-controlled opponents usually get more resources and play by more favorable rules to compensate for their lack of strategic skill. Their AI is typically implemented as a set of heuristics or fixed rules which it follows.

Since AI implemented in such way cannot learn, the player may find its weaknesses by experimentation. This is usually not considered as major flaw because the single-player mode of RTS games is thought of as training for multiplayer battles where AI opponents are either not present or play only minor role.

1.3. Multiplayer games

A multiplayer game differs from a single-player game by the fact that it involves more than one player. Two or more players interact with the same set of data (on the same device or on different devices). Multiplayer games are naturally more engaging due to their social nature, especially if the players are in the same room, can talk to each other, see each other or at least can exchange messages over the in-game chat. However, the complexity of the game increases dramatically as soon as the game becomes a multiplayer one.

1.3.1. Common challenges

Minimizing traffic

Since many users may pay for their Internet connection based on the amount of data they download or upload, it is vital that the amount of data sent over the network by the game is minimal.

Solution: The game state may be synchronized once at game start or as soon as a level or a map is loaded. After that the clients only receive and send incremental updates only about data which has changed. At major events in the game full state synchronization may be still viable to make sure that clients are still in synch. Requirement to minimize traffic makes architectures where each client sends updates to each other client wasteful and forces to use a single server to handle network communication.

In any case, the game should gracefully handle the situation when clients do run out of synch and attempt to perform contradicting or incompatible operations.

Network lag

When two computers send updates of a game state to each other there is an inevitable delay which depends on the quality of the network. This may be less of a problem if a game is played over LAN, but on mobile devices players often connect over mobile 3G or even 2G networks where speed is slower and occasional hiccups in data traffic are possible.

The player would see enemy units moving in abrupt jumps as their position gets updated over the network. If the whole game runs on the server then the player would see the lag even for his own units.

Solution: Clients may attempt to predict the position of moving objects until an update is received from the server. When the update is received the current position of the object is override with the one received from the server. Prediction may be based on the last update or on an averaged movement direction in the last few updates.

To make the correction less abrupt, instead of correcting the position immediately, the client may use interpolation and change the position across a few frames until it becomes the same as received from the server.

Disconnects

During a network game any client can unexpectedly loose Internet connectivity. If this client was also acting as the game server (host), all players will get disconnected and the game lost.

Solution: To minimize the chance of disconnects and also improve performance of the networked game, the clients may negotiate which of them should become server (for example, the one with fastest connection or most powerful hardware).

If the server gets disconnected, the remaining clients can re-vote and select one of them as the new server. The game could also allow a disconnected user to re-join the game after restoring connectivity.

Cheating players

Some players enjoy finding ways to gain an unfair advantage so as to repeatedly and easily defeat other players. While they may enjoy the game more as they do this, the rest of the players either have to cheat as well or just stop playing. This may lead to a collapse of a multiplayer game community, so cheating has to be fought.

Solution: A costly way to fight cheaters is to employ a system administrator who responds to player's reports about detected cheaters and introduces some penalties against them (for example, bans them for some time from the game).

A much cheaper way is to organize the architecture of a networked game in such a way that cheating is either impossible or ineffective. The whole game logic can be calculated by the server running on a dedicated machine in the studio's office and the clients only send input or actions of the players. For any action or input the server may do a sanity check. As soon as a cheat is detected the server apples punishment to the cheaters automatically.

In some games the mechanics can be organized in such a way that even though a player cheats during the game, the outcome is still checked and is discarded if a cheat took place. Unfortunately no such technique can solve the problem perfectly because cheaters may spend incredible efforts and creativity to find bugs and holes in defense to exploit.

Malicious users

If the game has a dedicated server there will be people who would try to bring that server down (for various reason, including pure challenge hackers see in the task or other developers trying to get rid of a competing game). A common type of an attack against a web server is distributed denial-of-service (DDOS) attack. Protection of a web server from attacks is a large topic and is out of scope of this thesis.

1.4. Games on mobile devices with touch screens

Games for mobile phones on touch screens present a unique set of challenges, both from game design and programming point of view.

1.4.1. Game design challenges

Users with low familiarity with computers

Smartphones can be used by people who may have never used a personal computer, for example small kids. The designer has to avoid using computer jargon in the in-game texts and never assume any background knowledge of the user. For example, in a tutorial for an RTS game a word "dragbox" shouldn't be used because it is technical jargon familiar only to hardcore players.

Casual gameplay

Casual games are liked by casual players which are the majority of players on mobile devices, but game designers are typically avid hardcore gamers themselves. To develop casual games means for them to develop games they would never play themselves.

The game designer of a casual game thus cannot rely on his own taste and feeling of what is fun and what creates engagement. Instead, he has to analyze statistics of downloads and in-game purchases, user reviews and results of beta testing.

Short playing sessions

A typical playing session of a smartphone game lasts 10 minutes and takes place in the subway while the user is going to work or back home [12]. This limitation effectively makes traditional genres (where the player would spend a whole evening in front of his PC) impossible to port to mobile platforms unless significant adjustments are made. There are various ways to fit the gameplay into short sessions.

The game may have very small and short levels so that in a typical session the player could only complete just a few of them. This design is used by many arcade games. The positive side is that if the player has more time, he can play through more levels in a row, so the game effectively scales to the time allotment.

The game may be a long and slow simulation where every action takes hours to complete, however requires that the player keeps returning to the game a few times a day for a few minute to do just a few simple actions. This design is used by farming simulators and strategic games focusing on building up castles or military bases. A common monetization technique in such games is to sell to the player ways to accelerate gameplay. Their whole design is built around creating engagement and then interrupting the gameplay which makes the player grow impatient and willing to pay for the speed-up.

A turn-based game may be able to wait for the next action of the player for an arbitrarily long time. Just like when reading a book, the player would interrupt the turn-based game when he gets out of the train to continue it a few days later from the same state he left it in.

Small screen and very large buttons

To fit into the user's pocket, smartphones have to be relatively small. On the other hand, the imprecise touch-based input requires the buttons and other interactive elements to be pretty large. This leads to a requirement to have either very simple graphical interfaces or interfaces with a lot of pages.

Game designers thus struggle to keep the number of buttons and user option as low as possible. A typical start screen of a game contains a big round Start button and just a few buttons critical to marketing and monetization: link to the paid version or in-app billing pages, Facebook and Twitter buttons which post advertising message on user's behalf in those social networks.

Imprecise controls

Imprecise user input makes games where player would have to extensively interact with objects virtually unplayable. For example, a first person shooter game where the player would usually point at enemies with his mouse and left- or right-click to shoot doesn't work well on a touchscreen. To turn the camera, the player would have to drag the finger across the screen, partly covering it and possibly failing to see the enemies under his finger. As soon as the player taps on the enemy to start shooting, the enemy gets covered by the finger and it becomes impossible to see when the enemy drops dead and one should stop shooting.

A typical way to interact with a touch screen is to touch it in a slow pace (1 tap per 2-3 seconds). A game designed for touchscreen devices can't expect the player to react fast to the in-game events which inevitably slows down the game process.

Some games also attempt to turn the touch screen into a game pad by placing virtual buttons in the bottom corners of the screen. This only works well if there is 1, at most 2 buttons on each side because on a flat touchscreen the player can't detect edges of the buttons with his fingers. Fingers can slip over to a wrong button. Instead of buttons it is even better in such case to have the whole screen to be split into large touchable areas which work as buttons.

Unwillingness to read instructions

Patience of players and their willingness to apply effort to understand instructions has decreased over the history of games. In 1980s games were delivered with manuals as separate documents. In 1990s games would include a collection of screens describing the game rules inside the game itself.

In 2000s the game would typically have an interactive tutorial inside. The tutorial not only told the user how to interact with the game but also showed exact places where to click and didn't let him proceed until the required click is done.

On today's mobile phones even an interactive tutorial is hardly tolerated by the user. The game may show instructions screen which the user would be forced to click through before playing, but most players would just skip those instructions without reading as if the instructions were some annoying ads or marketing messages. With so many free games on the stores the users may download them just to have a look and quickly decide if they want to keep the game or not because installing the game takes about as much time as reading its description and browsing the screenshots on the store. If such player is greeted by a lengthy tutorial he has to crunch through, he may get annoyed and just quit the game before even understanding what it is about.

Solution. Some games solve this problem by making the first few levels a tutorial sequence, introducing each new concept or type of action in the game as a separate level. Completion of each tutorial level requires that the introduced concept or action is understood and applied.

Games struggle to keep the player's attention and lure him to have a look at just one more level in hope that at some point the player finally gets to like the game.

For a complex game like an RTS or RPG the first location or the first level of the game may be fully devoted to explaining the gameplay to the user in hope that people who downloaded an RTS or RPG expect it to be complex and are ready to learn the rules by going through a tutorial. Developing a tutorial level requires scripting and a lot of playtesting.

Unwillingness to pay for Android games

At the time of this writing it is common knowledge that games on Android can't earn enough money to cover development costs. Android games are 3-10 times less profitable than iOS games [27, 28] and are either done by hobbyists who don't expect their games to bring any noticeable income or by big studios which already have a successful game on iOS and just wish to make their game better known to gamers so that Android users advertise it to iOS users.

Android's failure to monetize doesn't matter if one develops a game on a fully cross-platform engine like Unity3D where the effort to deliver a build for Android is virtually non-existent. Even if the Android version brings very low income, it's still worth to release it.

Poor discoverability

As of end of 2012, Play Store only offers the following categories of games: Acrade&Action, Brain&Puzzle, Cards&Casino, Casual, Live wallpaper, Racing, Sports games, Widgets [24]. These categories are hardly useful if the developer is releasing a game in one of the genres common on PC. For example, an RTS game should probably be placed into Brain&Puzzle where it will be lost among Tic-tac-toe clones, 3-in-a-line casual games and Tetris remakes. A multiplayer economic simulator of farms in the style of Farmville could go both to Brain&Puzzle and Casual, again to be mixed with completely unrelated games.

Since the store effectively becomes just a big pile, the users cannot rely on any classification. Patient uses would use search for common keywords in hope that games they may like contain such words in the description. But most users will just browse only the games in Staff picks or Editor's choice.

On AppStore for iOS the categories are much better (Figure 8), but the list of apps shown is limited by top 25 titles. To get next 25 titles, a long reload is required, so typical user hardly ever sees apps below top 50 in its category.

For a game designer all these limitations mean that making a niche game is not a viable strategy because the target audience will have trouble discovering it while a lot of other players will download it accidentally, will be disappointed and may leave bad reviews.

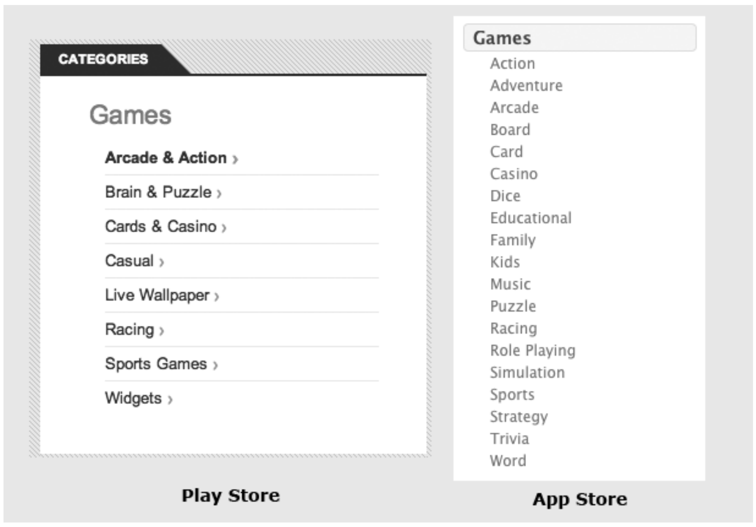


Figure 8. Categories of games on Play Store (Android) and App Store (iOS).

1.4.2. Development challenges

Low processing power of the devices

Mobile devices are generally much less powerful than desktop computers but what's worse is that at the same time there are devices on the market with very different capabilities.

Solution: The game may try to run some short tests at launch to determine the overall performance of the device and load resources of lower quality (like smaller textures) or set smaller target frame rate if the device is weak.

If the game is developed natively for iOS, two versions of resources (normal and high-definition) may be provided. In case of a native game for Android resources for 4 screen pixel densities (low, medium, high and extra-high) can be provided.

Fragmentation of Android platform

Android runs on devices with varying screen sizes and proportions.

Solution: When designing GUI, game designers and artists have to consider at least a few popular screen sizes and design a different version for each. Alternatively, the game may check the size of the screen at launch and apply some scaling factor to all GUI sprites to attempt to achieve uniform presentation across various screens.

Game can be interrupted at any time

Both Android and iOS have a Home hardware button, which brings the currently running app to a paused state and returns the user to the home screen of the operating system. The app has no control of this event and can not forbid it (this is a design decision of Google and Apple to prevent malicious programs from locking the user inside the app). The app only gets a notification that it is going to the background.

A more important factor for a game is that as soon as an app has been paused, the operating system is allowed to silently kill it (with or without notifying the app) for various reasons, e.g. when some other app requires memory and there is not enough free memory unless some other apps are killed (Figure 9).

On iOS, the game may be brought from foreground to background state any time (Figure 10), and then from background to foreground state (Figure 11).

Solution: The game may fully save its state as soon as a notification about the Home button press is received. All updates and repaints must stop, all timers and counters freeze. The game must save any data so as to be able to restore the current state from this data the next time it's launched.

Android and iOS are slightly different in this respect: on iOS the user commonly has no way to exit the app other than by pressing Home button, and apps usually don't have any "Exit" buttons inside them; on Android the user commonly presses Back hardware button to exit the app and Home button to just temporarily go to the home screen.

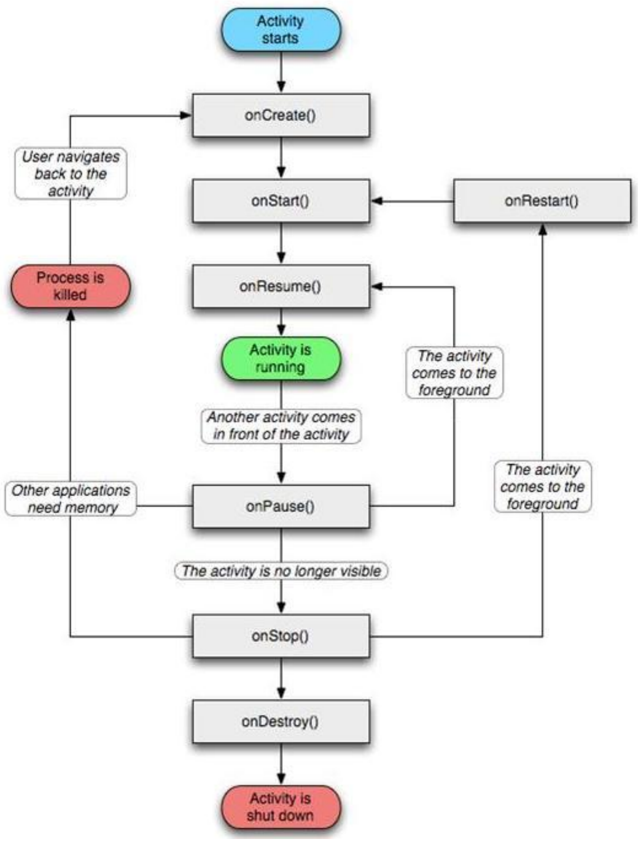

Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Figure 9. Android application lifecycle

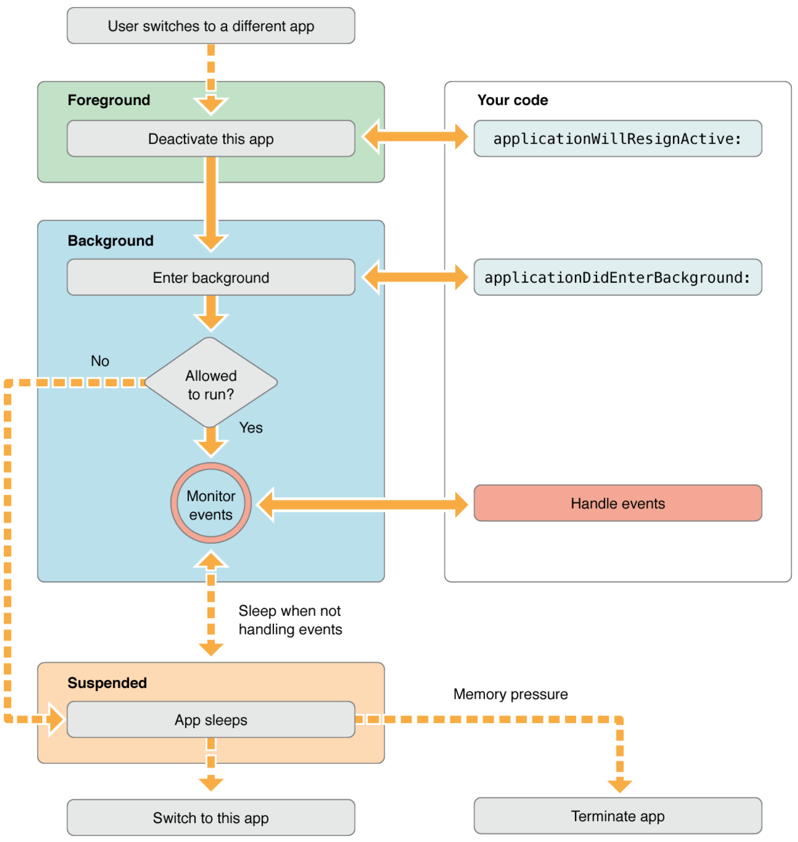


Figure 10. Transitioning of an iOS app from foreground to background [29]

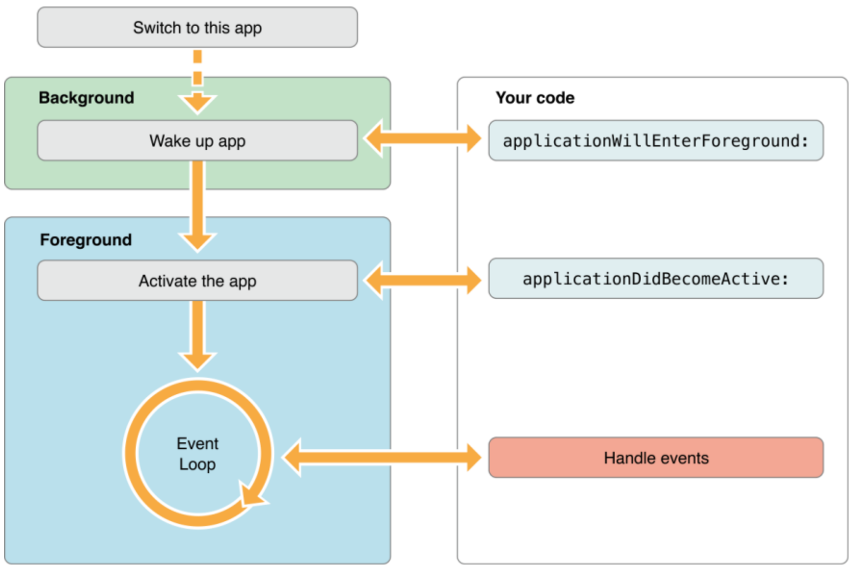


Figure 11. Transitioning of an iOS app from background to foreground [10329]

Limited battery charge

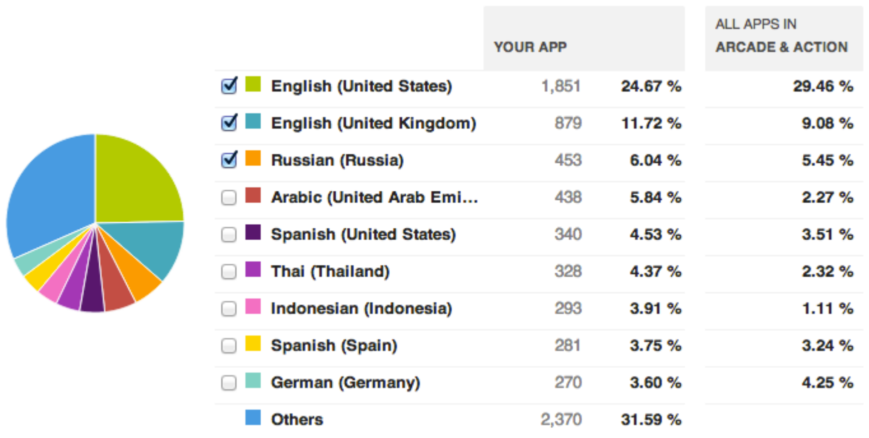
Today's smartphones can be used for between 1 and 3 days without battery recharge. However, battery consumption rises greatly when apps are run, especially such resource-intensive apps as games, even more games with multiplayer, which send data over WiFi or mobile network. Battery life can never be enough, so the games have to use efficient algorithms.

Localization

Smartphones reach very large audiences, also in non-English speaking countries. A larger portion of users than on desktop games doesn't understand English. For example, another game released by the author (Monkey UFO for Android) has only about 35% downloads in English localization (Figure 12). Top 10 languages cover about 70%, while other languages make up about 30% of downloads.

To support many languages, the game should have separate files containing strings only, a file per language, and use string identifiers in the code to look up the string in the currently used language. System of strings lookup is in-built into Android and iOS and requires no extra effort. Game engines offer varying levels of support for localization of games.

It is impossible to cover all languages, so to make the game accessible to people speaking other languages or illiterate people (including small kids who can't read) buttons with icons instead of text labels are used.



			YOUR APP		ALL APPS IN ARCADE & ACTION
☑	🟩	English (United States)	1,851	24.67 %	29.46 %
☑	🟦	English (United Kingdom)	879	11.72 %	9.08 %
☑	🟧	Russian (Russia)	453	6.04 %	5.45 %
☐	🟥	Arabic (United Arab Emi…	438	5.84 %	2.27 %
☐	🟪	Spanish (United States)	340	4.53 %	3.51 %
☐	🟪	Thai (Thailand)	328	4.37 %	2.32 %
☐	🟪	Indonesian (Indonesia)	293	3.91 %	1.11 %
☐	🟨	Spanish (Spain)	281	3.75 %	3.24 %
☐	🟩	German (Germany)	270	3.60 %	4.25 %
	🟦	Others	2,370	31.59 %	

Figure 12. Downloads of localized versions of the arcade Android game Monkey UFO

1.4.3. RTS genre on mobile devices

On mobile platforms of late 2000s (iPhone and later Android and iPad) the RTS genre was virtually non-existent due to following factors:

- modest processing power of mobile devices which did not allow to create visuals comparable to PC games or even just render an army of units on the screen;
- small screen which did not allow a comfortable overview of the map;
- slow and not very responsive touchscreen which did not allow to effectively control armies of units in real time;
- more casual audience on mobile markets which required both simpler gameplay than an RTS typically offers and non-military oriented setting.

Classic RTS ports

An example of an attempt to port the genre without significant modifications to mobile platforms is Starfront Collision from GameLoft or Red Alert from Electronic Arts. Much like in desktop RTS games, the player selects his units by dragging, but two fingers instead of one, to create a *dragbox* and select every unit inside the drawn rectangle. Then he orders the units to move or to attack by tapping on a spot on the map or an enemy unit. A normal drag is used to move the camera around the world.

Despite attempts to adapt these games to the limitations of the mobile platforms both games did not gain much popularity. Their players typically compared these games to desktop games and were disappointed with slow and clumsy controls, poor graphics and overall low-budget gameplay, forgetting that a mobile platform is just too weak and limited to meet their expectations.

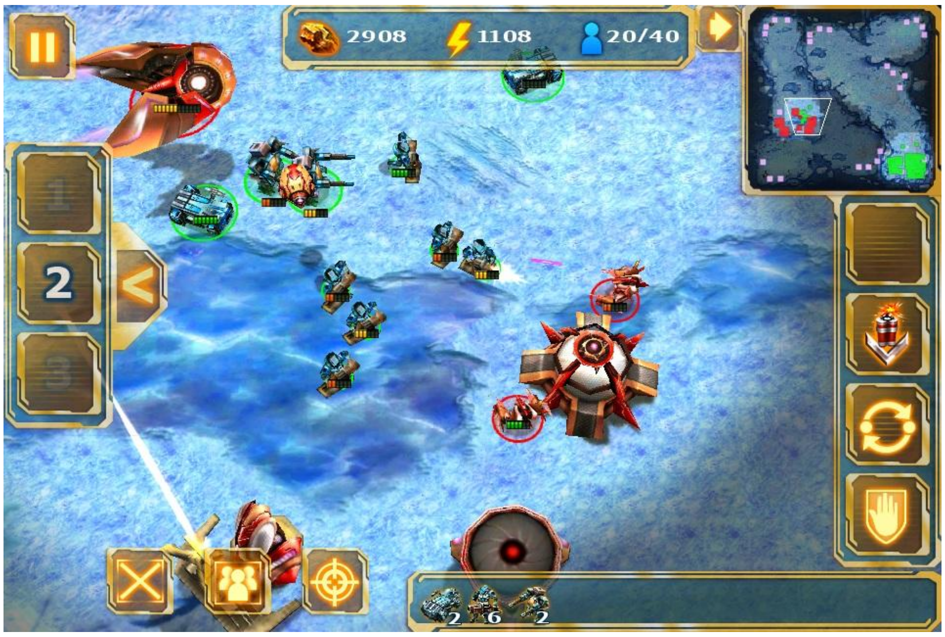


Figure 13. iPhone version of RTS game Starfront Collision by GameLoft, 2012 [25].

Figure 13 presents a game situation in the RTS game Starfront Collision. The player controls the blue units and attacks the red tower and defending units. On the left edge there is a panel to quickly select groups of units. At the top there is a panel showing player's resources. In the top-right corner there is a minimap - a map showing the whole game world at a very small scale. The current position of the player's camera in the world is shown with a white trapezoid, the player's base with green dots and enemy base with red dots. The right panel has buttons with special orders to the selected units. At the bottom there are 3 buttons to change the grouping of units and a horizontal panel showing units in the currently selected group.

Tower defense games

An actual search for keyword “strategy” on AppStore or GooglePlay in 2012 would bring up mostly games in *tower defense* sub-genre of RTS games. In a tower defense game the player places towers equipped with ranged weapons along a winding path.

During the game waves of enemies keep entering the map and move along this path. Towers automatically shoot at the enemies. Enemies do not attack the towers and just keep moving in an attempt to reach the end of the path where the headquarters of the player are situated. The level is won after all waves have been suppressed or lost if too many enemies have reached the player's HQ.

The opponent in the game is obviously not intelligent (because it doesn't change it's behavior no matter what the player does), so tower defense is not really an RTS game but rather a game about optimizing tower building in a set of dynamic but predictable conditions imposed by the environment.

The main factor of the genre's success is that the user rarely interacts with the game to build or upgrade towers and most of the time just watches the towers bring down the waves of enemies which gives him a feeling of power.

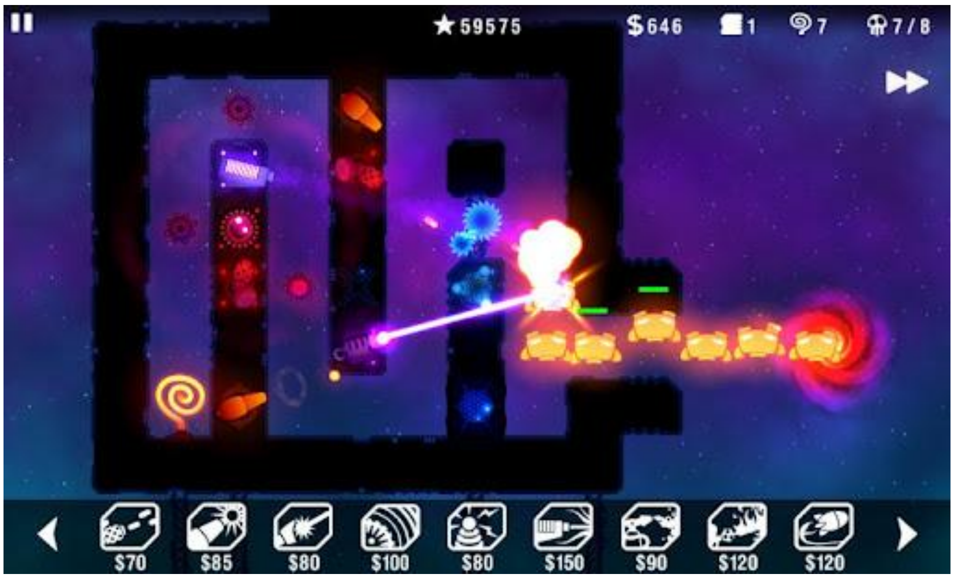


Figure 14. Android version of Radiant Defense game by Hexage LTD, 2102 [24].

Figure 14 shows gameplay of a tower defense game Radiant Defense. The enemies (orange robots) appear form a vortex to the right and enter the tunnel. Orange, blue and violet cannons are placed by the player on the black walls and shoot at the incoming enemies. Orange spiral in the end of the tunnel signifies the player's headquarters which the enemies seek to destroy. At the top of the screen player's resources are shown. At the bottom there is a scrollable horizontal list of towers which the player can build with their respective prices. The player can also place mines on the floor of the tunnel to damage and slow down enemies.

Defense of the Ancients remakes

Another successful sub-genre of RTS games spawns from Defense of the Ancients (DotA) multiplayer mod of Warcraft 3 game by Blizzard Entertainment. In DotA the player only controls the main unit which is assisted by a small number of AI-controlled supportive units.

The game was ported to Flash (League of Legends) and in 2012 to iOS and Android (Legendary Heroes by Maya). These games were somewhat more successful than classic RTS ports because the player only controls a few characters, the map is small and the gameplay is focused on defending static structures.

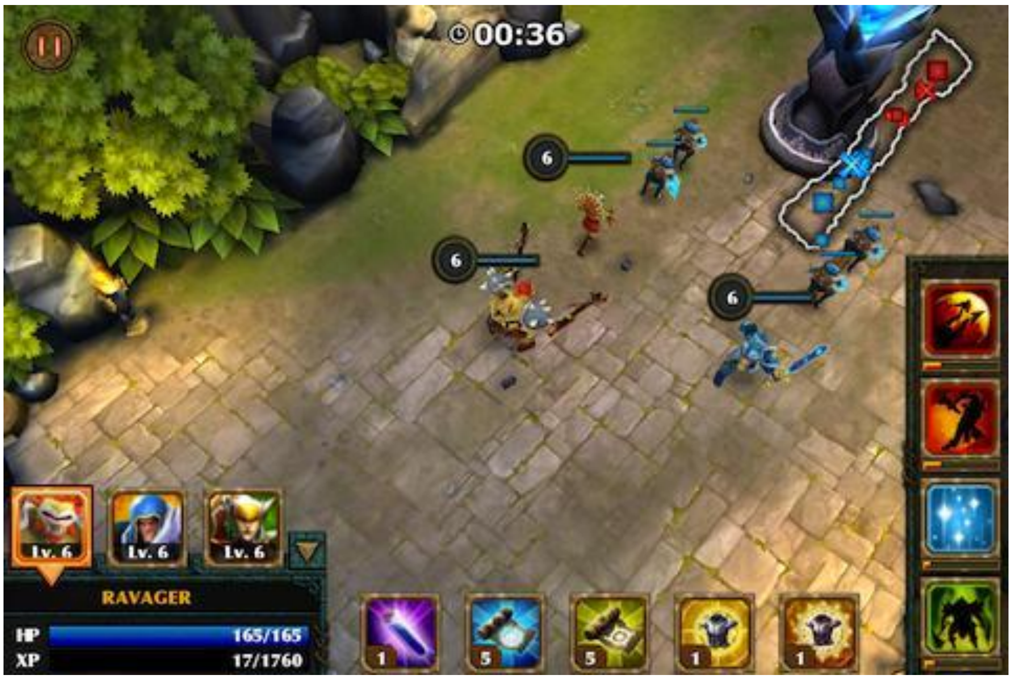


Figure 15. Legendary Heroes for Android and iOS by Maya [24].

On Figure 15 one can see a screenshot of the game Legendary Heroes. The player is moving his 3 heroes with a few small supportive units towards a blue tower. In the bottom-right corner the player can select one of the heroes and see the hero's name, health and gathered experience points. In the bottom there are buttons to use potions and other items in the backpack of the selected hero or special magical effects of the equipped armor. In the bottom-right corner there is a panel with 4 spells the selected hero can cast. In the top-right corner there is a transparent minimap of the world showing positions of the player's units with blue squares, player-controlled towers with blue crosses, enemy units with red squares and enemy towers with red crosses. The timer at the top shows the time remaining till the end of the game round. Pause button in the top-left corner opens the game menu and allows to exit the game or configure options.

Analysis of RTS genre on mobile markets suggests that the potential demand is high: there are a lot of players who are eager to see games in their favorite genre appear on new mobile platforms.

There is a handful of mobile RTS games so far, and these games have made attempts to adapt to mobile platforms from points of view of GUI design, gameplay features and monetization models.

It is very likely, that 2010s decade will see many new RTS games appearing on mobile platforms and new GUI and gameplay standards crystallizing as the genre attempts to migrate to mobile platforms.

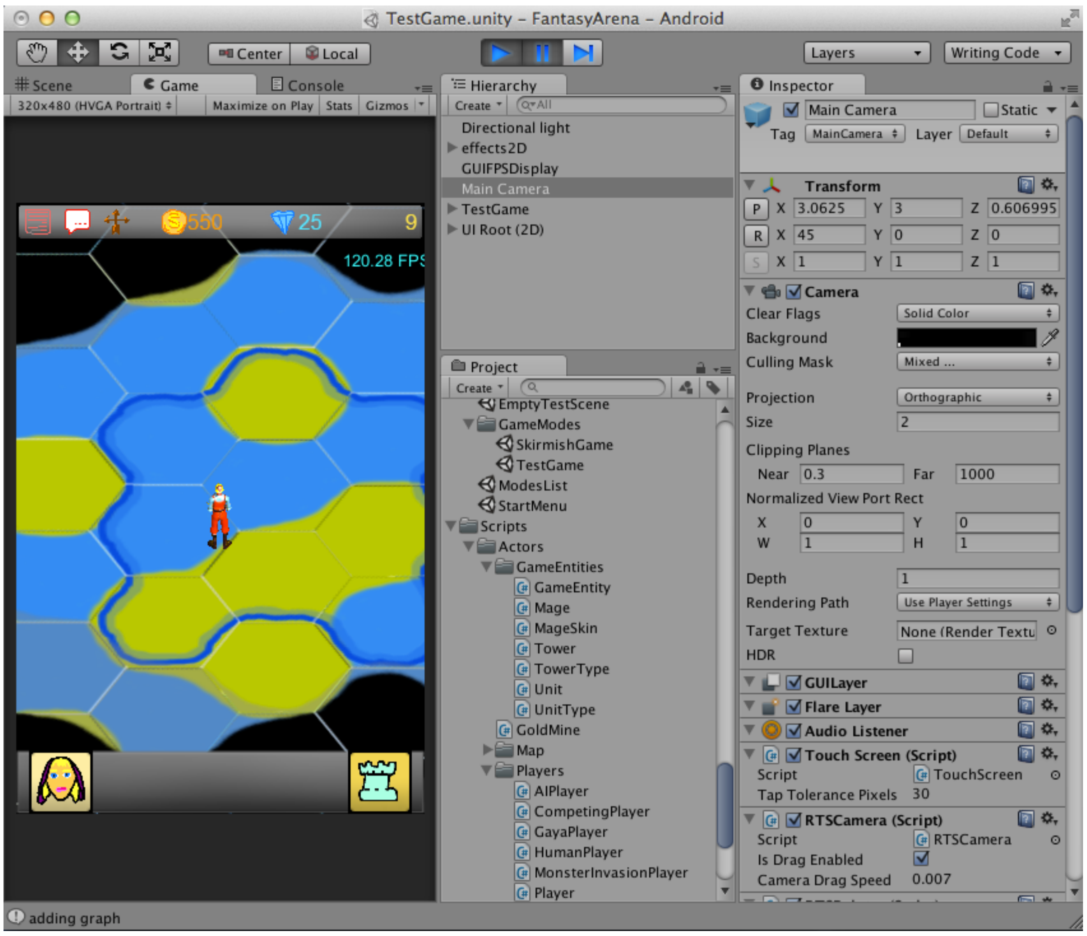
1.5. Unity3D game engine

As of 2012, Unity (or Unity3D) is the most popular game engine used for developing 3D cross-platform mobile games for iOS and Android. Unity is an industry-level engine comparable in capabilities to Unreal game engine, Torque and CryEngine, yet it costs 1-2 orders of magnitude less which makes it popular among small studios and indie developers.

Unity3D can be used by programmers, artists and game designers. It has a game editor with an interface resembling one of 3D modeling packages (3D MAX, Maya, Blender). A game developer populates the game scene with 3D objects (*game objects*), assigns materials to them, places cameras and lights.

To define behaviors of the game objects, the developer creates scripts and assigns them to game objects. For each level of the game the developer creates a separate game scene. Then the developer selects the target platform (for example, iOS), and the Unity3d engine creates a build ready for installation onto that platform. The programmer's role is mostly to write the scripts.

Scripts for Unity3D can be written in C#, JavaScript or Boo (a simple Python-like language). All scripts for this project were written in C# inside MonoDevelop which is included with the Unity3D engine.



Main Camera ☐ Static
Tag MainCamera Layer Default

Transform
P X 3.0625 Y 3 Z 0.606995
R X 45 Y 0 Z 0
S X 1 Y 1 Z 1

✓ Camera
Clear Flags: Solid Color
Background: ■
Culling Mask: Mixed ...
Projection: Orthographic
Size: 2
Clipping Planes: Near 0.3 Far 1000
Normalized View Port Rect: X 0 Y 0 W 1 H 1
Depth: 1
Rendering Path: Use Player Settings
Target Texture: None (Render Textu
HDR: ☐

✓ GUILayer
✓ Flare Layer
✓ Audio Listener
✓ Touch Screen (Script)
Script: TouchScreen
Tap Tolerance Pixels: 30
✓ RTSCamera (Script)
Script: RTSCamera
Is Drag Enabled: ✓
Camera Drag Speed: 0.007

Figure 16. Game editor of the Unity3D game engine

Figure 1 shows a state of the GUI of the game editor of Unity3D engine. The Inspector panel on the right shows game components on the Main Camera game object. One the components is a RTSCamera script which determines the game-specific behavior of the camera. The Hierarchy panel above shows game objects in the scene. The Project panel below shows game assets (scripts, textures, sounds, scenes, materials etc.) of the project. The editor allows to run the game inside an in-built simulator (on the left).

1.5.1. Component-based programming

A game world is represented in Unity3D as a scene populated with *game objects*. Each game object has a *Transform*: a component which determines position, rotation and scale of the object in 3D space. Apart from that, every game object may have a set of other components which define its behavior and properties. Unity3D implements a *component-based programming principle* which uses a concept of components instead of building inheritance hierarchies.

In usual object-oriented programming, classes inherit from other classes to override behaviors, e.g. *Cat* and *Dog* can inherit from *Mammal* and override it's *say()* method so that the cat would say "meow" and the dog say "woof".

If pure hierarchical approach is used, than a game typically ends with having gigantic classes inside which conceptually unrelated methods are tightly coupled. For example, a *Cat* class would be able to say its name, render itself on the screen, collide with other objects, serialize itself into an XML file, find path to a given point on the map, show a tooltip and do a lot of other things. Such class would have hundreds of methods and dozens of variables scattered around the inheritance hierarchy.

In components-based model the game object is just a container instead of base class. It has a list of components, one component per property or behavior, which makes the system very modular and allows to have much smaller and manageable classes (see Figure 17).

If a game object has a *RigidBody* component, it is considered by the engine as a physical body and is processed by the physics engine. For example, forces will be applied to it. It will fall down if gravitation is enabled for it.

If the object has a *Collider* component (various types of colliders are available including simple but imprecise *BoxCollider* and slow but accurate *MeshCollider*), the object will start colliding with other objects with colliders.

Renderer components allow the object to be visible for cameras. *AudioSource* component allows the object to emit sounds and *AudioListener* (typically placed on the main camera of the scene) - to detect sounds.

Most of the scripts the Unity3D programmer writes are classes inheriting from *MonoBehaviour* component. To control the object, such scripts can override a set of predefined virtual methods.

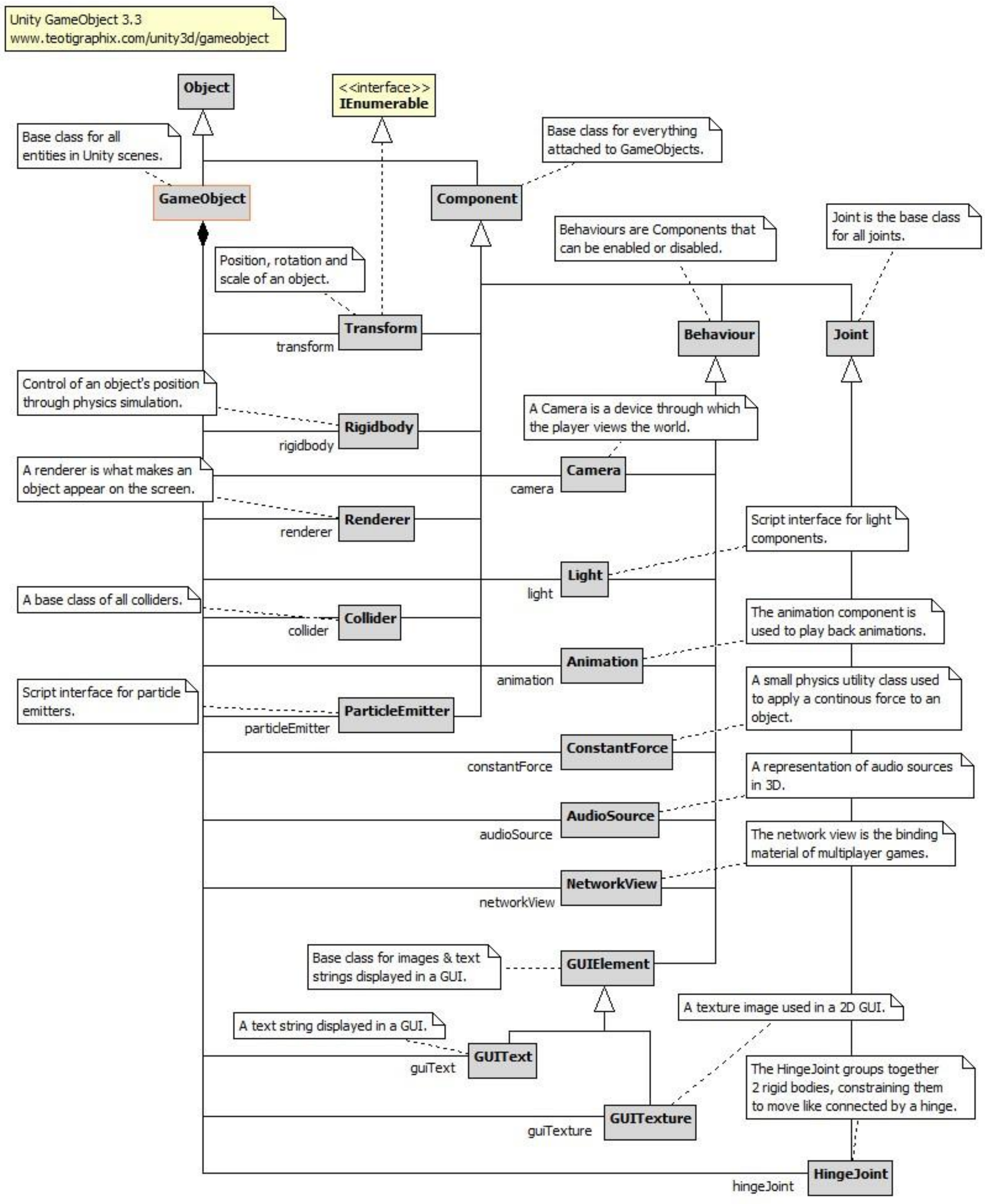

Unity GameObject 3.3
www.teotigraphix.com/unity3d/gameobject
Object
<<interface>>
IEnumerable
Base class for all entities in Unity scenes.
Base class for everything attached to GameObjects.
GameObject
Component
Behaviours are Components that can be enabled or disabled.
Joint is the base class for all joints.
Position, rotation and scale of an object.
Transform
transform
Behaviour
Joint
Control of an object's position through physics simulation.
Rigidbody
rigidbody
A Camera is a device through which the player views the world.
A renderer is what makes an object appear on the screen.
Camera
camera
Renderer
renderer
Script interface for light components.
Light
light
A base class of all colliders.
Collider
collider
The animation component is used to play back animations.
Animation
animation
Script interface for particle emitters.
ParticleEmitter
particleEmitter
A small physics utility class used to apply a continous force to an object.
ConstantForce
constantForce
A representation of audio sources in 3D.
AudioSource
audioSource
The network view is the binding material of multiplayer games.
NetworkView
networkView
Base class for images & text strings displayed in a GUI.
GUIElement
A texture image used in a 2D GUI.
A text string displayed in a GUI.
GUIText
guiText
The HingeJoint groups together 2 rigid bodies, constraining them to move like connected by a hinge.
GUITexture
guiTexture
HingeJoint
hingeJoint

Figure 17. GameObject in Unity3D

1.5.2. Prefabs

A game may not only use game objects which were initially placed in the scene by the game designer but also require to create objects in runtime (for example, in a strategy game place newly produced units). This can be done programmatically by creating a game object and adding all of the required components to it and setting its properties.

For easier instantiation of objects Unity3D allows to save some objects as *Prefabs* (prefabricated objects) which can then be copied into the scene in runtime. Prefabs in Unity3D act as blueprints for objects. Instantiating a game object from a prefab is conceptually very close to instantiating an instance of a class in object-oriented programming.

A common practice is to create a prefab for every object that may be used in the game.

1.5.3. Plugins

Unity3D provides an AssetStore - a portal where developers can purchase game assets (for example, 3D models or textures) as well as plugins (collections of scripts which add functionality to the game editor or allow to implement parts of the currently developed game easier). In the development of "Fantasy Area" the following plugins were used:

- *NGUI* - plugin to create GUI of the game in a better way than the in-built GUI system of Unity3D allows

- *Prime31 social plugin* - plugin to post messages on the player's Facebook wall and on his Twitter feed.

- *Astar Pathfinding Project* - plugin, which efficiently implements the A* pathfinding algorithm on a graph of waypoints.

- *Lumos* - plugin to gather in-game analytics.

Chapter 2. Design of "Fantasy Arena" game

This chapter describes the design of the developed game. It starts with listing the general ideas which where chosen as the starting point in the design. It then follows how these ideas evolved into design decisions about various features of the game as the different limitations of the mobile platforms were considered. Then a short overview of the resulting game design is given. The chapter concludes with a list of elicited requirements and limitations on the game's architecture.

Game design and software design

There is a mixture of meanings of the term "*design*" when it is used in relation to games.

In software engineering *design* means deciding on the architecture of the software, on which components will be needed to achieve the requirements of the system, how these components will be organized and how they interact.

In games industry, *game design* means just the description of the requirements, which is never formulated in a technical language. *Game design* covers design of the system only on its Requirements elicitation stage or at most Application domain model stage.

Development of a game usually starts with writing a *Game design document* by a game designer. Game designers are not software engineers. Software design and implementation decisions starting with the game architecture (Application domain model, Solution domain model) are fully done by the game programmers.

It's the programmers' job to elicit requirements from the game designers and translate generic and technically imprecise description of the proposed game provided by the *game design document* into concrete requirements and constraints.

The design document for "Fantasy Arena" is provided as a separate file together with the source code. In this chapter we first put the hat of a game designer and describe the game idea and features in a non-technical way. Then we put the hat of a software engineer and reformulate the game description in form of requirements and constraints which prepare the software design of the game and its implementation which will be covered in the next chapter.

An important difference of games from other types of software is that they are not aimed at solving a problem of the user or providing him with tools to improve his productivity. Instead, the game *creates* problems for the player. It puts him into a situation which has to be actively changed with the hope that while solving this problem the player will be entertained.

The game designer makes a creative guess what the user may find entertaining and initiates the technical process of creating the game by describing his vision of the system in the design document for the programmer.

While being technically just a piece of complex software, a computer game is by it's nature and meaning to the user rather a piece of art, much closer to a movie or a book, yet an interactive one, more like a toy. So, while software engineering criteria of quality and software engineering techniques can be used to the *implementation* of the game idea, the *idea* itself lies completely outside of software engineering.

For example, we can discuss pros and cons of some GUI layout for an RTS game in terms of usability and ergonomics, but the decision to make a tank unit stronger than the infantry unit is an arbitrary choice of the game designer from the point of view of the programmer.

Game balance

Choices in game design (in its game industry meaning) are usually driven by the concept of "*game balance*" which means that every element of the game is needed and doesn't create *dominating strategies* in the game. In game theory, *dominating strategy* means that some choice of a player is the best no matter what the opponent does.

Suppose, in a strategic game there are 3 unit types. The warrior defeats the barbarian, and the barbarian defeats the archers. All unit types cost the same. Figure 18 shows relationship "defeats" with arrows. The dominating strategy is to produce only warriors.

The time and effort developers spent on adding the barbarian and the archer to the game are essentially in vain because players will only produce warriors and will never see other units.

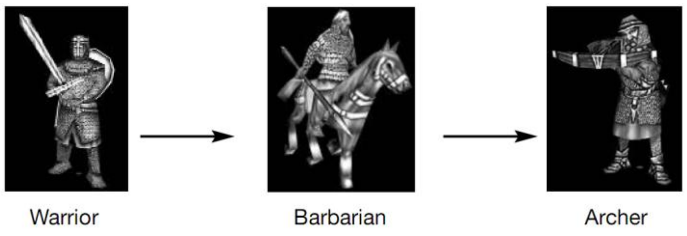


Figure 18. Relationships between units in a game with broken game balance [13].

Figure 19 shows another strategic game where units are balanced according to rock-paper-scissors model. The warrior defeats the barbarian which defeats the archer, and the archer defeats the warrior. The situation in the game would determine which unit is better to produce at the given moment. In such a game all 3 unit types are needed.

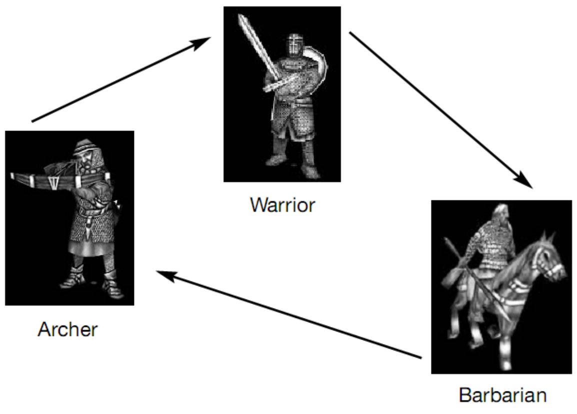


Figure 19. Relationships between units in a balanced game ("rock-paper-scissors") [13]

2.1. Game design decisions

In this section we put on the hat of a game designer - a non-technical person unfamiliar with software engineering and who is actually the client of the game programmer. We describe what game we want to get as it will be seen by players and game reviewers.

2.1.1. Main idea of the game

Strategy games are games about interesting choices [13, p. 60]. The basic creative ideas around which the "Fantasy Arena" game design was based are:

- to combine 3 sub-genres in one game: 1) classic RTS, 2) tower defense and 3) DoTA-style gameplay with strong RPG elements

- have territory expansion, however not in form of castles or bases but small pieces of land; also this expansion should be very dynamic and take place during the battle, not as a result of battles as it is usually done in slow strategic multiplayer games for web-browsers

These ideas can't be justified by anything except a feeling that "this exact combination of game mechanics hasn't been in any games yet".

The fact of having some novel elements while still re-using proved concepts and re-mixing them in interesting ways is a typical approach of game designers when coming up with ideas for a game. The novel elements are added by the game designer based on his aesthetic feeling and a creative guess what could make the game more fun to play for its target audience.

The basic ideas set the major conceptual goals of the design of the game, provide direction when more detailed decisions are made and help the game to keep a single "vision".

2.1.2. Basic features

The basic ideas were further concretized in the following set of basic features (game elements) in the initial design.

DoTA-style features

Like in DoTA, the player controls the main character which has a common set of RPG-style elements: experience points and levels, a selection of spells he can cast and a skill tree.

Initial consideration was also to allow the main character to equip armor, weapons, including magical ones with additional spells and effects, and also carry a backpack like in Legendary Heroes where the character could hold healing potions, magical scrolls and other items. However, this feature was not implemented in the technical demo.

Also a system of *quests* (tasks which a player has to accomplish to get some in-game reward) was considered which could play a big role in single-player mode of the game and maybe even in the multiplayer mode. The quests system was reserved for further development.

The main rationale behind having a main character in the game is that the player should associate himself with this character and get engaged with the game by trying to make this character stronger, level it up and build it's skill set in a unique way which best matches the individual play style of the player.

RTS-style features

Like in classical RTS games, the player controls multiple units directly. Since the game declares itself as an RTS game and not an RPG game, the feature of having directly controllable units comparable in power to the main character was considered as a crucial one. These units must be of different pre-defined types.

Like in RTS games, such units should not gain experience so that the player could loose them in battles and replace them with new ones without hard feelings he could otherwise get if some effort had been put into leveling up those units.

To increase player's engagement and to make the learning curve less steep, the types of units the player can build should get gradually unlocked as the player keeps leveling up his main character.

Tower defense-style features

Like in tower defense games, the game should have a wide selection of defensive towers the player can build. Such towers ideally fit the concept of expanding and defending the territory.

Towers should be a place-and-forget feature: the player should only decide once where and what tower to place, and after that the tower shoots at the enemies fully automatically, even automatically repairs itself when attacked.

The same way as with units, to increase engagement and ease learning, tower types are unlocked gradually.

Territory expansion

Since the player should spend most of the time in the game controlling his main character the concept of expanding territory should be implemented as activity of the main character and not just some impersonal process.

The decision was thus made to give the main character a special spell "Capture tile" which it has to cast at spots on the ground to add pieces of land to its territory.

2.1.3. Limitations

In further design the following constraints of mobile platforms were taken into account and the following decisions about the design of the game were made:

Small touch screen of the device, imprecise touch

These 2 constraints together present a chain of game design problems:

- only a very small part of the map can be viewed at any given time;
- because the units have to be selectable and touches are very imprecise, the units have to be quite large;
- on a 5-inch screen only a rectangle of 6 by 8 tiles (with one unit per tile) can be displayed;
- scrolling is clumsy and slow;
- on a small screen the player will not see the whole area in sight range of his main character and may not notice enemies, loose the game because of that and get frustrated because he will consider this a technical issue which brings unfairness and annoying randomness into the game.

Solution: the game world must be very small and be at most a square of 3 by 3 screens, better just 2 by 2 screens.

In such a small world all units, towers and spells of the mage must have very short range. An absolute limit on sight and attack range for all units, tower and spells is set to 4 tiles, but an average attack range should be just 1-2 tiles.

Also in the beginning of the game the sight range of the main character should be small (just 2 tiles) so that the issue of enemies outside the screen could only arise much later in the game, when the players are more engaged and feel more comfortable with the controls.

Slow rate of touches

The touch screen responds slowly to touches. Also it takes time to move the finger from one place to another on the screen as well as just to see what happened on the place which the player just touched. Because of this the player can't effectively control more than 2-3 entities at once, and also the game can't count on fast reaction of the player.

Solution: At the start of the game, the player only has the main character under his control. To build (summon) more units, the player has to first unlock "summoning slots".

Each summoning slot allows to have one more unit on the field. On high levels the player should still not have more than 4 summoning slots so that in total he would have to control 5 entities (the main character plus the 4 summoned units).

Also, the main character's skills should be organized in such a way that it is a perfectly valid strategy to not unlock many or even any summoning slots. Players who aren't dexterous enough to control many entities at once could still enjoy the game by only building static towers or just doing all battles with the main character alone.

Furthermore, the summoned units must have a set of relatively complex AI modes which the player could toggle so that the units could act autonomously.

For example, the units must be able to follow and assist other units or the main character - this way the player would be able to lead the attack by just casting spells on the enemy (the units in "assist" AI mode will attack the same target as the main character).

Also, each unit must be able to defend a tile and attack enemies in range. Healing units must be able to automatically heal friendly units nearby.

To even further reduce micromanagement the units should be able to select the enemy target according to some smart policy (for example, first attack damaged enemies so that they go down faster or first attack the enemies with highest attack power and weakest armor). Such policy could be selected by the player from a list of policies, each optimized for a different tactical situation.

The battles should be slow so that the player could react adequately to the changing situation: the units have to move slowly, they should be able to withstand many hits before they die.

For aesthetic reasons the projectiles of the ranged attacks should still fly at a usual speed common in games (fast but noticeable).

To make the game more static, the player-controlled character cannot move outside his territory. Other units still can move anywhere because they will be mostly used by players who feel comfortable with touch screen.

Short playing sessions

Since a typical playing session is about 10 minutes on a mobile device, a battle in a real-time strategy game should fit into this time limit. In case of a multiplayer game the full scenario of finding human opponents over the Internet, launching the game, playing it and viewing the results should fit into 10 minutes.

It would be nice if the game could also scale to other time frames between as short as 5 minutes and as long as 2 hours, so that the game could cover the use case of a player playing on an iPad from home.

Solution: Resource gathering which is so common in RTS games, has to be brought to an absolute minimum. Also, there have to be no buildings in the game except towers.

Base build-up is a common initial stage in RTS games and takes between 5 and 30 minutes, but it can't be afforded by a 10-minutes long game on a mobile device. So, a decision was made to have only 1 type of collectible resource (gold) which is used to summon units, build towers and unlock skills.

To decrease micromanagement of resource gathering, gold is gained in regular ticks for just having gold mines inside the player's territory.

In RTS games main characters typically have mana which is spent on casting spells and gets gradually restored by just waiting or drinking potions. Again, in a 10 minute game the player would have no time to wait for mana to restore, so mana is not present in the game.

Instead, to limit the player's spell-casting rate, only the common RPG system of *spell cool-downs* is used (Figure 20). Each time the player casts a spell, he can't cast the same spell again for the following few seconds.

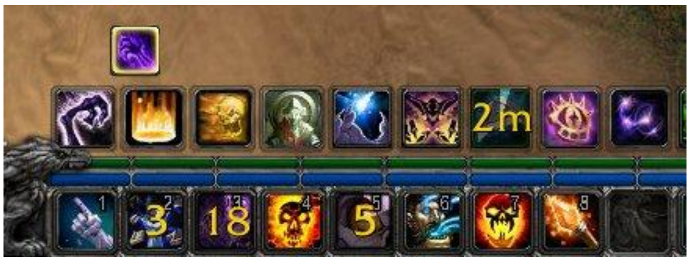


Figure 20. Cool-downs of spells shown as numbers over the spell icon in online RPG game World of Warcraft

Since the cool-downs are usually relatively short the player could still cast spells too fast by just casting different spells.

In some games like World of Warcraft this problem is solved by introducing a *global cool-down* of 1.5 seconds in addition to individual cool-downs of spells: after any spell has been cast, no other spell can be cast until it's own cool-down and the global cool-down have passed.

In other games (for example, Guild Wars 2) the problem is solved by limiting the number of spells the player can select for casting: there are only 5 spell buttons and if the player has more than 5 learnt spells, he has to decide which of them to put on the quick spells bar (the rest will be inaccessible).

For "Fantasy Arena" the second approach was used because it also solves the problem of presenting all the available spells on the small screen: only 5 quick spells are shown and the rest can be found in a special spells dialog which covers the whole screen and is supposed to be viewed rarely.

RTS games typically use lobby screen to start multiplayer sessions. Some players declare that they will host a game, wait until enough players have joined and then start. To minimize time of searching online opponents, in addition to the lobby system the game should also have a Quick start option.

When the player opts to quickly start a game, the multiplayer game must get assembled in less that a minute and fully automatically. The auto-matching feature should take the levels of players into account and attempt to match player of similar levels.

Strategy games usually end only after a single party (alliance or a single player) has defeated all other competitors. If the parties are roughly equal in strength, the game make take very long time or even come to a boring draw/stalemate situation.

To avoid this and ensure that the session fits into 10 minutes, there is a hard limit of 10 minutes on the battle. If by the end of this time there are still more than 1 competitors in the game, the game is won by the party with higher score. Score is given in 1-minute ticks for the size of the territory under player's control.

To make the game session scalable, the players should be able to host a game with a custom rule on the game duration, ranging between 5 minutes and 2 hours.

Players who can devote more time to the game can join the session of optimal duration for their case.

Unwillingness to read instructions

Players will drop the game if learning curve is too steep, the instructions are too long or the game rules are too complex overall.

Solution. The game should have a single training map with an interactive tutorial. Also to make the game easier to learn, only very few abilities should be accessible to the player at start. At level 1 the player's character has only 1 attacking spell (fireball) and 1 spell to capture tiles. It can't summon any units or build towers.

The player starts the game by capturing tiles around him to grow the territory and gets experience points for this. Since "Capture tile" spell is the first one the player would use it should be used automatically on the tile if the player attempts to move there. After a few captured tiles the character gains a level, and new abilities become available: the weakest unit and 1 summoning slot, another attacking spell in addition to the fireball and the weakest tower type.

So as not to overwhelm the player, the full skills tree is never shown and instead only the skills which he can unlock at the current time are displayed. Also, to make it easier to the player to concentrate on one of the modes of gameplay (tower building, controlling units in RTS style or casting spells in an RPG style) the skills are organized into separate trees which should even be shown on separate dialogs (units-related, towers-related, spells-related plus another tree for generic all-purpose spells).

If the game rules are not presented all at once but instead get unveiled as the player progresses from one battle to another, some players may get engaged by just the will to explore all the spells, the units types and towers and see how they interact in the game.

This learning process should last for months as the player gains higher and higher levels at an ever slower rate, but also as new spells, units and towers are introduced into the game in later updates. Ideally the game should have no end, but the time required to gain a level becomes so immense at higher levels that players effectively stop progressing and only the most dedicated ones (the "elite" players) have a character a few levels above the rest.

2.1.4. Commercial requirements

Monetization of the game is outside of scope of this thesis, yet in considering design of the game in the games industry it plays central role.

The following features crucial to monetization have been considered:

- additional skins and possibly special items or units which the player can only access by paying real money. They should not break the game balance but still affect gameplay.

- there can be a way to get in-game gold for real money but to protect the game balance the rate at which the player can get gold for money must be limited so that the game did not degrade into a "pay-to-win" scam. For example, the player would buy crystals for money, which can be converted to gold inside the game. However, only 1 crystal per hour can be converted.

- the free version of the game should show ads in the place where accidental click is most likely: the bottom of the screen, if the game is played with one hand in vertical device orientation. A paid Pro version of the game without ads should be available.

Marketing of a mobile game is a serious issue given the low discoverability of apps on today's app stores. Games may attempt to utilize the ubiquity of social networks and convince the players to advertise the game to their friends, make the game "viral", so that news about it spread across the social network like a virus.

To achieve this, games nowadays have a Facebook and Twitter buttons which place an advertising message about the game on the user's Facebook wall or Twitter feed. These can be not only messages generally praising the game but also reports that the player has gained a level or succeeded in achieving something in the game.

Players may not like that games post spam on their behalf, so games have to reward the player, for example by turning off ads as soon as the player has used the Facebook button to tell about the game to his friends.

When selecting the *setting* of the game (its overall artistic theme) the following variants were considered:

- fantasy - magical world with Medieval technologies, based on Germanic mythology and books by Tolkien ("The Lord of the Rings"). The cause of the conflict can be ancient evil gods or orcs attacking humans and elves. Or this can be just a strife of magic orders for power.

- post-apocalyptic - world after an allegedly happened nuclear war where survivors have come out of the bunkers and joined different groups based on their political agenda.

- sci-fi - the conflict takes place in space or on a distant planet with different races of aliens involved.

The fantasy setting was selected since it is considered to be the type of setting which attracts the greatest audience and appeals both to hardcore and casual players.

Based on this choice, the title "Fantasy Arena" was selected to stress the game's competitive multiplayer nature and its fantasy theme.

2.2. Game description

In the previous section we covered the main features of the game and the reasons why they were selected. To ensure a better understanding of the game before we go on to the elicited requirements, this section provides a shortened version of the game design document. Parts already mentioned in the previous session are skipped or covered without much detail. In this section we are still wearing the hat of a game designer and not of a software engineer.

2.2.1. Game mechanics

World

The game is played on a square 8x8, 16x16 or 32x32 grid of hex tiles. Each tile is assigned some type of terrain which defines if player character can move to this tile, if the tile can be captured, what towers can be built on the tile and what units can move through it. High-level mages can cast a spell to change terrain type of the tile.

There is a Fog-of-war system (Figure 21). Tiles can be visible, explored or unexplored. Visible tiles are those in sight of player's character or units. Explored are those which were in sight previously but aren't now, or are visible/explored for other players in the same team. Initially the whole world is unexplored.

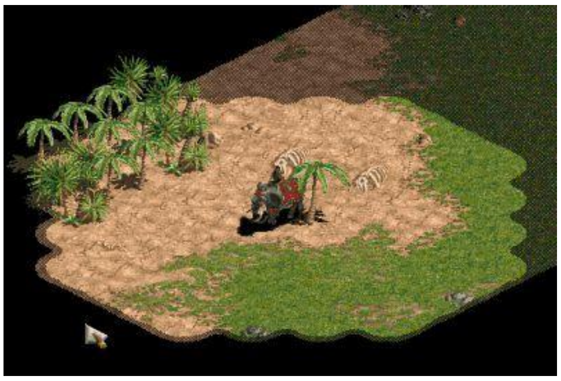


Figure 21. Fog of war system: visible, revealed and unrevealed areas around the elephant unit in RTS game Age of Empires 2.

Game entities

The player is represented in the game by a mage. Before the battle starts, the players can make teams. Apart from everyone-for-himself typical setups can be 2vs2, 3vs3, 4vs4, 2vs2vs2 and so on, including asymmetrical ones. Time limit of the game is specified by the players at game start, it can be 5, 10, 20, 30, 60 or 120 minutes.

If the time limit is reached and the game is still not finished, the team with the highest score wins. Score is gained in ticks of 1 minute, 1 point per each tile in player's territory. The mage can use special "Capture tile" spell to add tiles to their territory. Players are randomly placed on the map with a small piece of territory which has a gold mine on it.

Mages can summon units and towers for gold. The mage can only move and place towers inside his territory. Mages, units and towers have *hitpoints* (lives) which they slowly restore with time. If a mage dies, it resurrects after a 30 second delay at the nearest empty space inside his territory.

If no such space exists or no tiles belong to the player, the character gets a new start at a different place of the world map with small territory including a gold mine. If no place is available, the player loses the game.

Mages, units and towers can perform abilities (for example, cast spells, shoot arrows or attack in melee). Abilities have cool-downs.

Skills and experience

To cast spells, summon units or build towers, the mage first has to unlock the corresponding skill from a skill tree. Unlocking costs gold and skill points which are gained when mage levels up. Levels are gained by accumulating experience points which are given for killing enemies and capturing tiles.

There are 4 separate skill trees: spells-related tree, units-related tree, towers-related tree and a special tree for miscellaneous general-purpose skills.

Economy

In-game gold is produces by gold mines inside player's territory.

To prevent snowball effect (when a mistake in the first minute of the game dooms the player to failure), each new mine added to the player's territory brings less gold and the player can use no more than 5 gold mines (any extra mines will not bring any gold).

For killing enemies, 20% of their price in gold is given as a reward.

Physics

There are different types of damage to which all creatures and towers in the world have different resistances (percent of damage dampened). Damage of attacks in the game is defined as a range between min and max value.

When damage is inflicted, a value is randomly chosen between min and max, then part of it is dampened by the target's resistance to that type of damage, and the resulting value is subtracted from the target's lives. All attacks always hit their targets, the projectiles used to graphically represent spells and ranged attacks fly like homing missiles and always reach their target.

Any object in the world can have a stack of “Effects” on it which can be of limited duration and expire. Such effect can, for example, inflict damage to the object and slow it down.

Some game objects may emit auras which apply an effect to everything around as long at it is in the range of the aura. For example, a silence aura of Magehunter unit will disallow mages around it to cast spells.

Game entity control

The player can select towers and order them to shoot at specific targets but normally towers shoot at enemies in range automatically. Towers can be repaired for gold or sold.

Towers can also be upgraded. Upgrades are only applied to a single tower and present a binary tree with 2 choices available at any time and 3 depth levels making in total 8 variations of each tower at fully upgraded state.

Units can be controlled in RTS-style by tapping on a spot on the map to move to or an enemy to attack. All units and towers have just 1 ability which they apply to the target. Healing units have beneficial abilities which can only be applied to friendly targets.

Combat and healing units have different sets of AI modes (behaviors) which the player can choose for the unit.

There are no transport units which could host other units, also units can't enter towers.

Neutral armies

The whole map is initially populated with random neutral units and towers which players have to clear off the map while expanding.

On large maps with many players and time limits of 60 or 120 minutes to liven up the game occasionally monster invasions happen. At a random place which still doesn't belong to any player a portal opens and a large army of powerful neutral units is spawned to attack closest human players.

Among such neutral invasions there can be special mage-like units which Capture territory of player's back to neutral territory.

2.2.2. Game control and GUI

Camera

RTS camera is controlled by dragging the finger across the screen. A special zoom in/out feature can be implemented and level-of-detail technique used when the camera is high above ground.

Camera movement is limited by the borders of the map.

Camera cannot be rotated or tilted.

Orders

All orders in the game require a full tap.

Tap on a friendly unit, mage or tower selects them, a tap on an enemy target gives an attack order, a tap on an empty spot on the map - a move order or in the special case when mage is selected an a tile outside the territory is tapped - an order to capture the tile.

When the player touches the screen (touch down) over any object in the game (unit, mage or tower) and waits for a short time, while the finger is still touching the object, a semi-transparent dialog is shown with quick info about the object including it's attack range as a red circle around the object.

GUI

The in-game GUI consists of two horizontal action bars at the bottom of the screen and a thin bar at the top of the screen. Only portrait orientation is supported.

The lower action bar always contains the button of the main character to the left, then up to 4 buttons for selecting or summoning units (summoning slots) and on the right edge the button to build towers.

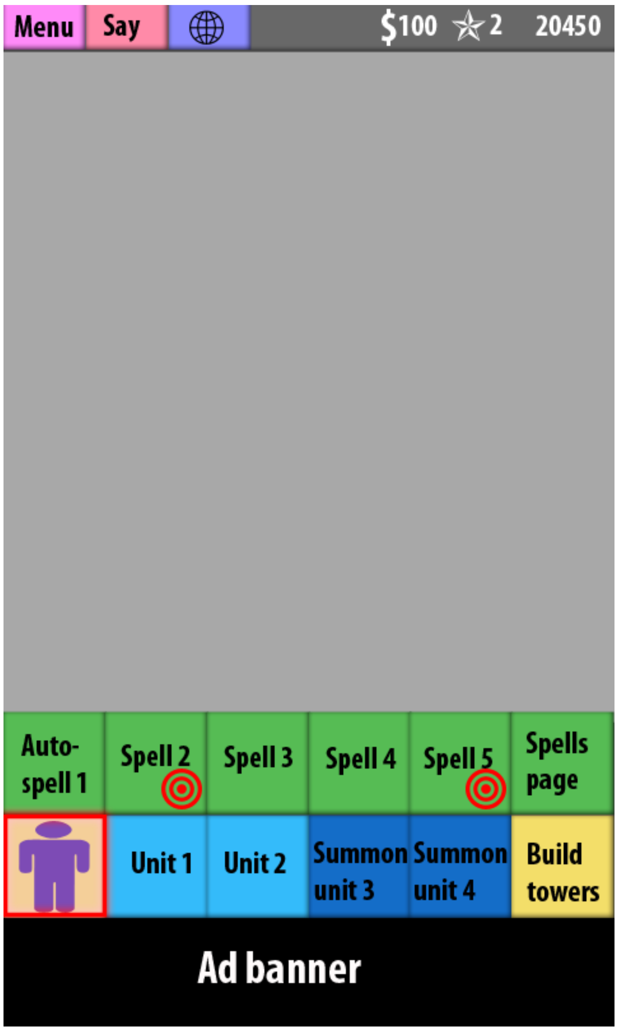


Figure 22. In-game GUI layout

The contents of the upper action bar depend on the current state of the GUI.

Quick spells

While a mage is selected, the upper action bar shows 5 quick ("memorized") spells of the mage of which the leftmost one is always the one used for auto-attack and is cast automatically. On the right there is a button opening a window with all spells known to the mage ("Spellbook").

Icons of the spells can be dragged from this window to the quick spells bar (there is a 5-second cool-down on this action as the mage "memorizes" the spell) and drag one spell over another to swap their positions on the bar.

Spells which can be cast at a spot on the ground have a red scope icon. To cast a specific spell, the player taps on the button of the spell and then on the target object or the spot on the ground.

If the mage is already attacking something with his auto-attack spell, the spell will be cast at the same target. Cool-downs of spells are shown as partial gray overlay over the spell's icon.

Unit orders

While a unit is selected, the upper action shows a button for the main ability of the unit and buttons for various AI modes of the unit.

To the right there is an Unsummon button which kills the unit, partially restores it's cost and frees up the summoning slot. If a healing unit is selected, the player should tap the main ability button (for example, Heal) and then tap on a friendly target to start healing. Only 1 unit can be selected at any time.

Unit summoning

Tap on a summoning slot which has no unit triggers unit summoning GUI state and the upper action bar is filled with 5 buttons for summoning units. Tap on any such button summons the unit.

On the right edge of the summoning bar there is a button to open the Units window. From the units dialog the player can drag unit types to the unit summoning bar and swap units in the summoning bar by dragging icons over each other. Over the unit type icons in the summoning bar the price of the unit in gold is shown.

Tower building

Tap on the Build towers button shows a quick list of up to 5 tower types to build in the upper action bar.

The rightmost button in the bar opens the Towers window. The player can drag tower types from the window to the quick bar and swap tower types in the bar by dragging them to a different slot. Tap on a tower type in the quick bar selects the tower type and the player has to tap on a spot in the game world to place the tower. Over the tower type button it's price in gold is shown.

Tower orders

While a tower is selected, it's main ability button is shown on the left of the upper action bar. Other buttons are 2 possible upgrades of the tower, repair it for gold, auto-repair toggle and a Remove button which destroys the tower and refunds part of it's cost.

Skill windows

Windows for known towers, units and spells have similar layout. The user can switch between pages using buttons at the bottom. To the right of the icon a short description of the spell, unit or tower is shown including their price.

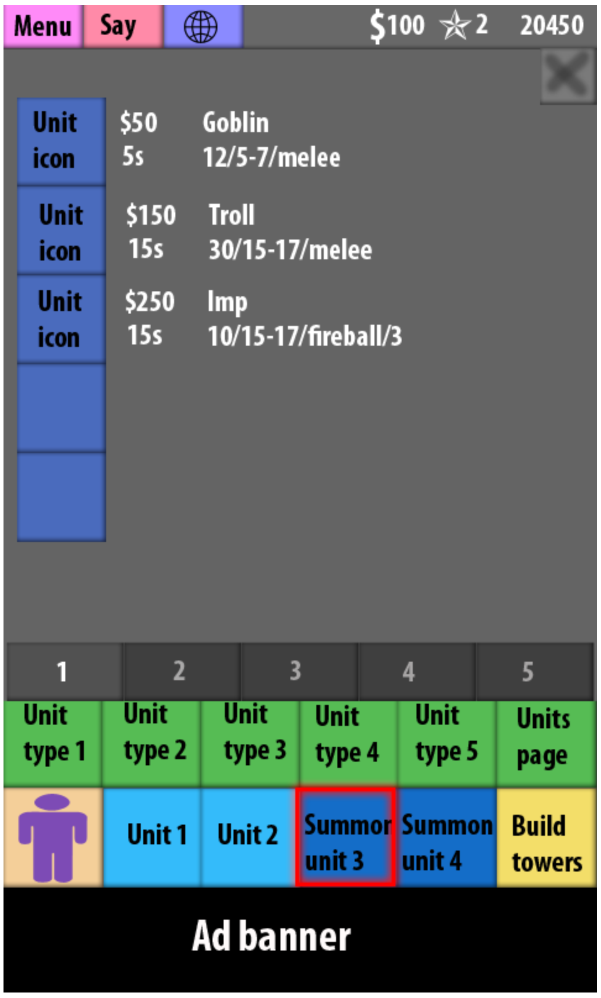


Figure 23. Layout of the units window

If the player has just gained 1 or more levels, a special symbol is shown over his character's portrait. When the player taps on the portrait, the skill selection screen is opened. It's similar to the skill pages screen, except that in the list only the skills are shown which the player can pick in the selected category. There are 4 categories (toggled by the buttons at the bottom of the page): Units, Spells, Towers, Other. Tap on the icon of the skill unlocks that skill.

The panel at the top of the screen shows resources and score of the player and 3 buttons: Menu, Say and World.

Communication of players in the game happens through a chat with 3 channels: global chat, team chat and private messages. Messages of other players pop in the top-left corner of the screen. Messages are formatted depending on the relation of their author to the player. To say something, the player presses a Say button at the top of the screen.

Menu button opens a system menu where the player can opt to exit the game, surrender, and configure options. World button opens battle statistics screen where the player can assess his current standing in the game compared to other players and also send gold to allies.

Presentation of game entities

Units, towers and mages are represented in the game by animated 3D models. Units have 4 animations: moving, attacking (or healing), dying, being summoned (appears in the game world). Towers have 3 animations: attacking, being destroyed, being built (appears in the game world). Mages have 4 animations: moving, casting a spell (for very different types of spells different animations may be used), dying, being resurrected (reappearing in the game world after having been killed).

Over each mage there is an overlay bar with the name of the mage and it's level. The mages can have different 3D models (skins) to represent them in the game with it's own icon and set of animations.

For any damaged objects (having less than maximum number of hitpoints) a health bar is shown above the object. The length of the bar represents the percentage of hitpoints remaining. The color of the bar ranges from green to yellow to red depending on health status.

2.3. Requirements elicitation

In this section we finally put on the hat of a software engineer and proceed with eliciting the requirements of the system based on the description of the game. This list only includes major requirements which could also be broken down to lots of smaller requirements since the developed game is actually quite complex. To fully understand the meaning of the requirements one has to also be familiar with the game description.

2.3.1. Functional requirements

- The player must be able to host a multiplayer game, set time limit and teams structure, join a game, select the team he wants to play for and start the game
- In the beginning of the game session a random map of hex tiles of different types must be generated, mages get randomly placed and starting territory with gold mines assigned. The map must be populated with random neutral units and towers.
- The player must be able to move the camera in 4 directions around the game world.
- The player must accumulate gold for owning gold mines and score for controlling territory.
- The player must be able to summon new units into available summoning slots and place new towers.
- The player must be able to move his mage and units around the map, fog of war must be shown and updated accordingly. Units and the mage must not enter impassable terrain or attempt to enter occupied tiles.

- The player must be able to capture tiles with his mage
- The player must be able to learn new skills after gaining a level, which must unlock new unit types, tower types and spells.
- The player must be able to cast spells with his mage (with system of cool-downs applied).
- The player must be able to issue an attack order, and the selected unit, tower or mage must starting using their default ability against the target.
- The player must only be able to quickly use max 5 spells from his spellbook and define which spells that would be
- After one of the team has defeated all opponents the game must end making this team the winner. If the time limit expires before this happens, the game must end and decide the winner based on the score.
- The player must be able to send and receive messages over the in-game chat.
- The player's character must carry over his accumulated levels and skills to the next battles.

It is important to note here that some of the features described in the design document actually do not present requirements but rather a suggestion how some feature should be implemented.

For example, the wish that the player should be able to place some spells on the quick spells bar by dragging spells from the spells window is not a requirement but a description of the proposed implementation of the GUI system. This system is aimed at meeting the functional requirement "The player must only be able to quickly use max 5 spells from his spellbook and define which spells that would be". During the development and testing of the game this system can be changed with high probability.

2.3.2. Non-functional requirements

- The game must sustain high refresh rate, on average of at least 30 frames per second (FPS) and never slower than 10 FPS on all officially supported devices (Android 2.2 or higher, iPhone3GS or higher).
- The size of the game build for Android must not exceed 50MB, for iOS 100MB.
- The game must be playable with one hand.
- The game must provide uniform GUI across all supported mobile platforms.
- The game must gracefully handle a disconnect of any of the game participants and hand over control of his units and mage to the AI.

- The movements of the units and mages must look smooth even on a slow network with high latency and packet loss.
- The game must gracefully handle the situation of conflicting updates if clients get slightly out of synch.
- The game must prevent (or at least do it's best to prevent) cheating of clients.
- The game must allow up to 8 players to participate in the game session.
- The game must resolve common NAT and firewall issues, except from very strict conditions of corporate networks.
- The game must be implemented in a modular fashion allowing constant further development and changes after initial release, especially in the following: types of units and towers, spells, mage skins.
- Progress of players must be stored and backed up on a server.
- The game must turn off sound immediately if the player has been using headphones and unplugged them (use case: the user plays at work during a meeting and accidentally pulls the headphones so that they get unplugged).

2.3.3. Constraints

- Units must be controlled in RTS-style familiar to players: single tap on a friendly unit selects the unit, single tap on an occupied spot issues a move order and a single tap on the enemy issues an attack order
- The game must use Unity3D game engine given the present level of skills of the author with this engine acquired with previous projects and the investment into the Pro version and commercial plugins.
- The game should not use any copyrighted content or content requiring a license (characters, setting).
- The game should contain only mild graphical violence and get rated "Everyone" by ESRB [32].

2.3.4. Low priority requirements

Low priority features are “nice to have”. They do not have to be included in the final deliverable but may be added in later updates to improve player experience.

- The player should be able to equip weapons and armor on his mage, put items into the backpack of the mage and use the items (e.g. healing potions).
- The player should be able to order a unit to enter one of the AI modes. For each mode, the unit should demonstrate corresponding behavior.

- For spells, attacks and other events corresponding animations and particle effect should be used.

- The game should represent mages, units, towers and other in-game objects with professional animated 3D models.

- All notable events in the game must be accompanied by professional sound effects.

- During the whole playing experience and in all game screens professional background music of appropriate style should be looped.

- The player should be able to zoom in/out the camera

- The user should be able to post on Facebook or Twitter about the game directly from inside the game.

- The player should be able to buy in-game currency for real money using Apple's and Google's in-app billing systems.

- The player should be able to start a game with random opponents quickly without hosting or joining using a matchmaking server.

- The game should be able to run long game sessions with very large maps and more than 24 players connected simultaneously (MMO mode). The AI should take over control of units of players which went offline and give control back to the players as soon as they come online.

- To optimize performance the game should automatically detect the capabilities of the hardware on the current device and automatically adjust graphical settings to achieve required FPS without too much reducing rendering quality.

- The game should contain an offline single-player mode with a set of missions where the player faces only AI opponents.

- On bigger maps the game should spawn monster invasions

- Before the game starts, the host should be determined automatically by comparing network quality and device performance. If the host later disconnects, the new host should be determined with a re-vote and the game continued without interruption.

Chapter 3. Implementation of "Fantasy Arena" game

This chapter describes implementation of the actual game, starting with overall architecture, then covering some of the interesting and technically challenging features and concluding with the testing techniques used. In this chapter we are wearing the hat of a game programmer and start our work on the basis of requirements and constraints elicited from the game design document in the previous chapter.

3.1. Overall project structure

3.1.1. Project assets organization

Unity3D game engine enforces a certain way of game architecture and requires that all in-game assets (e.g. scripts, textures, models) are placed into *Assets* folder inside the main folder of the project. Inside *Assets*, the programmer is free to organize his files in any way.

For each file inside *Assets* folder or one of its subfolders Unity3D editor automatically creates a metadata file with the same name as the original file and *.meta* extension. Each time the editor is brought to the front it rescans the whole *Assets* folder hierarchy and updates the metadata files.

Each time a script is edited in MonoDevelop or any other external text editor, Unit3D game editor automatically detects the changes and recompiles the scripts.

Unity3D provides some helpful tools inside the GUI of the editor which however only work if the scripts are written in compliance with the editor's conventions. For example, all C# classes must remain in the same namespace so that they could be shown in Component/Scripts menu item and the game designer could easily add the script to a game object.

The result is that most of the scripts are all visible to each other and the only way to organize them is to place them into different folders.

Also, if some subclass of *MonoBehaviour* (base class for most in-game scripts) has public variables, such variables can be edited by the game designer from inside the game editor, in the Inspector panel. This feature breaks OOP principle of encapsulation but provides an easier way to configure parameters of scripts.

Inside Fantasy Arena project, the game assets are organized in the following way: the folders immediately inside Assets represent the type of asset, then inside those folders assets of the same type may be classified into folders according to the module of game where they belong. Also, folders with assets belonging to plugins are placed into Plugins subfolder or a separate folder.

The structure of the project (expressed in the structure of folders) is thus the following:

- *Scripts* - contains all scripts except from those coming inside plugins. The structure of subfolders reflects game modules and is similar to that inside Prefabs folder.

- *Prefabs* - contains prefabricated objects which can be copied into the game in runtime. The structure of subfolders reflects game modules and is similar to that inside Scripts folder.

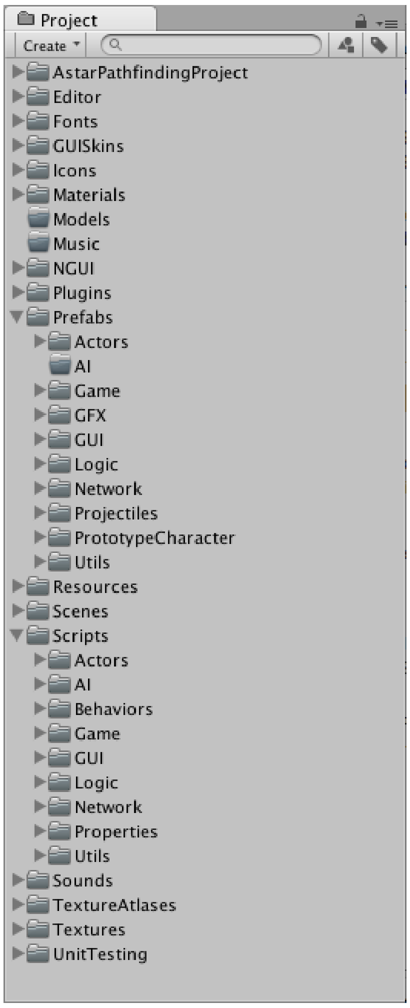


Figure 24. Project assets structure

- *Editor* - contains files of plugins which affect the functionality and GUI of the Unit3D editor

- *Fonts* - contains TTF fonts as well as all bitmap fonts for NGUI plugin

- *GUISkins* - contains a single GUISkin (not needed if NGUI plugin is used consistently for GUI)

- *Icons* - contains the icons of the game of various size, which are used when the game is built for various platforms

- *Materials* - contains materials which are used to apply textures to 3D models

- *Models* - contains animated 3D models

- *Music* - contains music files

- *Resources* - contains 2 prefabs: Assets and Rules (see Representing global data).

- *Scenes* - contains game scenes including TestScene which was used during development to test features.

- *Sounds* - contains all in-game sound effects

- *Texture Atlases* - contains GUI2D atlas utilized by NGUI plugin for this game. More texture atlases can be added e.g. to optimize materials.

- *Textures* - contains images (textures)

- *UnitTesting* - contains test scenes and scripts used for unit testing of in-game scripts.

3.1.2. Architecture of the scripts system

To achieve maximal readability of the code as well as modularity and extensibility of the system, *clean code* [8] programming conventions were used. In particular the following rules were set and consistently enforced:

- Script files must be small (less than 200 lines of code, usually just 20-50 lines).
- Functions must be small (never be longer than 7-10 lines, usually just 3-5 lines). Functions should do one thing which the name of the function must unambiguously describe.
- Classes should be small and have only one reason to change. They should relate to one concept which is unambiguously described by the name of the class. Classes must have a small number of methods and fields. Most methods of a class must use most of the fields. This allows to achieve high cohesion within components and low coupling between components.
- Components must know about other components only exactly as much as they require and not more.

To meet these principles, scripts of Fantasy Arena game have the following structure reflected by subfolders:

- *Actors* - contains scripts which configure composition and generic characteristics of the main objects used in the game: *Mage*, *Unit*, *Tower*, *Map*, *Tile*, *Player*, *Team*, *GoldMine*. As much as possible these scripts strive to contain no custom logic and only act as "middlemen". They contain links to other scripts, mostly from *Behaviors* and *Properties* folders.
- *Behaviors* - contains basic behaviors (like *MoveStraightLine*). None of these scripts knows of any script from *Actors* folder. These scripts are agnostic to the current game and can be re-used in other games. Each such script implements a concept which is described as a verb: the object "does" something. *Behavior* scripts usually do not contain any data which other scripts would need to access.
- *Properties* - contains components which together construct the game logic but are still agnostic to the game actors so as to keep modularity as high as possible. For example, *WithLives* component can be assigned to any game object, and after that this object can sustain damage and report death when lives reach zero. *Leveling* component allows the object to gain experience points and level up when required amount of experience points is reached. *EarnsSkillPointsOnLevelUp* is another component which depends on *Leveling* component and gives skill points to the object as soon as it levels up. Scripts in this categories represent persistent properties of objects and can be described using "is a" or "has a" expressions. They often contain data which is accessed or even modified by other scripts from *Properties* or *Behaviors* folders. These scripts can also be reused in other games if such games contain similar pieces of game logic.
- *AI* - contains AI-related scripts, including AI modes of units, global strategic AI of AI players and AI or scripted events specific to levels in single-player campaigns.

- *Game* - contains scripts which relate to the game as a whole or control the overall flow of the game. *GameControl* subfolder contains scripts for RTS camera and issuing orders to units, mages and towers. *GameModes* subfolder contains various game modes including planned ones and the *TestGame* mode which sets up a test scene to demonstrate and test game features.

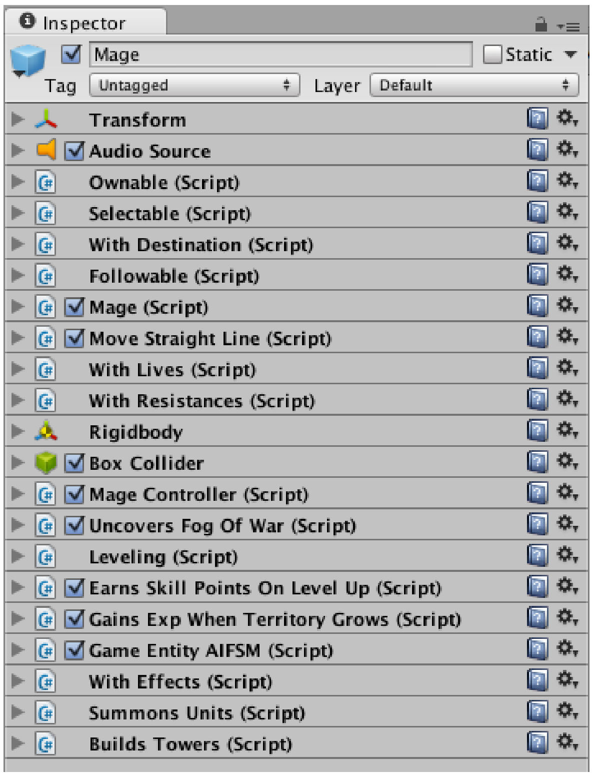
- *GUI* - contains scripts which are assigned to buttons, panels and other objects in the GUI system based on NGUI plugin.

- *Logic* - contains scripts related to abstract concepts in the game which aren't represented by any 3D models: abilities, spells, skills, effects and auras, damage and resistances, game economy.

- *Utils* - contains utility scripts which aren't specific to the currently developed game (or even games in general). These scripts are sorted into subfolders: AI, Audio, Generic, GFX, GUI, IO, Math, Memory and other. Utilities are intended to be a fully reusable collection of source file which grows and provides more and more functionality across projects.

- *Network* - contains scripts implementing network communication in multiplayer mode.

The resulting system of quests consists of about 200 C# source files. High modularity of the system leads to having fewer prefabs, each having many scripts attached. For example, the prefab of a mage has 21 components of which 17 are scripts (Figure 25).



```
Inspector
☑ Mage                                   ☐ Static
Tag  Untagged            Layer  Default
   Transform
☑  Audio Source
   Ownable (Script)
   Selectable (Script)
   With Destination (Script)
   Followable (Script)
☑  Mage (Script)
☑  Move Straight Line (Script)
   With Lives (Script)
   With Resistances (Script)
   Rigidbody
☑  Box Collider
☑  Mage Controller (Script)
☑  Uncovers Fog Of War (Script)
   Leveling (Script)
☑  Earns Skill Points On Level Up (Script)
☑  Gains Exp When Territory Grows (Script)
☑  Game Entity AIFSM (Script)
   With Effects (Script)
   Summons Units (Script)
   Builds Towers (Script)
```

Figure 25. Components of mage prefab (17 of which are scripts)

The *Mage* component is of *Actor* type and mostly manages other scripts. The only *Behavior*-type script of the *Mage* is *MoveStraightLine* which allows the mage to move to a destination. More *Behavior*-type scripts can be added to the mage in runtime, for example, a behavior to cast a spell. The rest of the scripts are all of *Property* type and define all the various characteristics and properties the mage has. Many of them are shared with units or towers, for example, *WithResistance* script which defines resistance of the mage to various types of damage.

3.2. Unity3D customization

Complex real-world projects developed with Unit3D reveal shortcomings, inefficiencies and poor design choices in the engine. This forces programmers to find workarounds or write custom scripts which partially replace or enhance the in-built functionality. In this section we describe specific techniques which we had to use to compensate some of the Unity3D's shortcomings.

3.2.1. Object pooling

A normal way to work with game objects in Unity3D is to instantiate them, use them and destroy them when they are no more needed. Unfortunately on mobile platforms (Android and iOS) instantiation and destruction become expensive operations, and if more than a few objects are instantiated per second, the frame rate drops significantly.

For example, our testing using the Unity3D's in-built profiler has shown that an attempt to instantiate 20 floating text messages as separate game objects in the same tick is not noticeable when the game in launched in the in-built player inside Unity3D or as Windows or Mac application. However, this causes a nearly 2-second freeze of the game on Android device HTC Desire with Android 2.2 and about 1-second delay on iPhone3GS. Instantiation of a single object can thus take between 1/20 and 1/10 of a second, which alone would take more time than is allotted for a single tick.

A possible solution could be to spread instantiation of objects across many updates so as to instantiate only a few objects per update and let the frame rate drop only slightly. This however is only feasible for non-critical operations like showing a floating text message. If instantiation is critical to the game logic, its delay will cause a hiccup in the whole game flow and also would significantly complicate the game, making many operations asynchronous.

A better solution is *object-pooling*. A special container object is placed in the scene which instantiates a large number of objects of various types when the scene is loaded and the user is patiently waiting for the level to start. All such objects are initially in a deactivated state: they don't get rendered (because their *Rendering* component is disabled) and the physics and collision engines don't consider them (because the *RigidBody* and *Collision* components are disabled).

As soon as some script requires to instantiate a game object of some type, it instead asks the pool to give an instance of that object. The pool then just activates one of the saved objects and returns a reference to it.

It's the caller's responsibility to later notify the pool that the object is no more in use - the pool then deactivates the object again and makes it available for other callers.

The following methods implemented objects pool functionality:

- *GameObject* ***GetObjectInstance****(GameObject prefab, Vector3 position, Quaternion rotation)*
- *void* ***FreeInstance****(GameObject obj)*
- *void* ***PreallocateObjectInstances****(GameObject prefab, int count, GameObject parent)*
- *void* ***Drain****()*

While the game is starting, *PreallocateObjectInstances* should be called for all types of objects (prefabs) which need to be pooled. During the game *GetObjectInstance* and *FreeInstance* are used instead of in-built *Instantiate* and *Destroy* methods. Finally, in the end of the game *Drain* must be called to clear memory.

The pool also ensures that the objects receive proper callbacks (*Awake*, *OnEnabled*, *Start*, *OnDisabled*) methods they would normally receive. The objects don't have to know that the object pooling technique is applied to them. To them, their lifecycle looks as if they were instantiated and destroyed like Unit3D system requires.

3.2.2. Representing global data

Unity3D was initially designed as a game engine for first-person shooter (FPS) games where the player controls an invisible "ego" character (a soldier) in a 3D world. The main character runs, jumps, shoots at enemies, and interacts with some objects like buttons, levers, weapon crates and healing pills. The paradigm of game objects with some simple scripts which are triggered when the player interacts with them inside the game is perfectly suited for such kind of gameplay.

With strategic games however there is no such thing as "player interacting with a game object" because the player is only represented in the game by a camera and the actual objects are units which have complex behaviors.

Also, strategic games inherently require storing of a lot of data which changes during the game. The player gathers resources and spends them, the units can be assigned different orders including such complex ones as patrolling an area along a set of waypoints, units in some strategic games can be grouped, and the group would act as a single entity.

Finally, in any game there is a lot of global data which defines the game rules. In case of a strategic game such rules would include large tables of parameters of unit types. In RPG games rules would define parameters of spells and how those spells become available to the player's character as it gains levels (so-called *skills tree*).

Unity3D examples and tutorials on the Internet and in books usually create a single *Game* script which is placed on the main camera object. The game rules and parameters are stored there.

This may work for tiny example projects, however even in small games with more than one game scene this leads to data duplication. The same parameters and rules of the game as well as the same links to assets (textures, sounds, music and other) have to be copied from one scene to another.

A next step is to make the main camera a prefabricated object and configure the rules and required global assets of the game once on it so that in every game scene the rules are changed automatically. This still however only remains a valid approach for very small games with at most a few dozens of parameters and globally used assets.

In Fantasy Arena the global data is stored in dummy prefabs with scripts on them which mostly contain configuration data. In the example of data validation a special Goblin prefab stores everything about the Goblin unit type (Figure 38).

The finally released game may contain up to 50 unit types, spell types, tower types and skills, and for each of them a separate dummy prefab is needed. To make all this data accessible globally a special prefab *Rules* is used. It is placed into *Assets*/*Resources* subfolder. The *Rules.cs* script on this prefab contains arrays of spells, unit types, towers types, skills and other parameters which the game designer has to set up manually once. *Rules* class is a singleton.

In a similar fashion, globally used assets (for example, a sound of button click) are all linked by the *Assets.cs* script which is placed on the *Assets/Resources/Assets.prefab* prefabricated object. It is also a singleton. The links to globally used assets have to be set up manually once.

Assets/Resources folder is a special one for Unity3D editor. Normally, assets are only added to the game build if they are referenced by other assets in at least one of the game scenes added to the build. Any assets inside the folder Assets/Resources are always added.

These two prefabs contain global data which should always be accessible and from any game scene, so placing them in Resources ensures that all rules and all globally used assets are always present in the game.

3.2.3. Messaging

In games, frequently a need arises for some objects to send data to other objects. For example, a flying fireball has at some point to tell its target that it has finally reached it and now some lives must be subtracted.

The easiest way to implement this would be to let the fireball call a method on the target directly. This however quickly leads to tightly coupled and rigid designs where every component must know about many other components.

In component-based programming paradigm there are at least 3 ways to exchange data between components [7]:

- Modify the host object's state to store temporary data
- Call methods of another component directly
- Send messages to the object and let it's components handle them

.NET platform also offers additional alternative:

- Fire events on the sender and handle them on the receiver

Modifying the host object's state

A component writes to a variable inside the game object and another component reads out the variable.

Since Unity3D doesn't allow extending *GameObject* and adding any extra variables to it, we stored such variables in scripts from *Assets/Scripts/Properties* subfolder.

For example, to implement movement of units in formations the following set of scripts is used: *FollowInFixedFormation*, *WithDestination* and *MoveStraightLine*.

FollowInFixedFormation only keeps track of the followed object and sets the new destination in the *WithDestination* script.

MoveStraightLine keeps checking the *WithDestination* script and adjusts the direction of movement accordingly.

WithDestination is of course required by both *FollowInFixedFormation* and *MoveStraightLine* scripts. It acts as a place of data exchange and allows flexibility on both ends. In fact, there are other ways to follow (not at fixed shift but in a circle or swinging around the followed object, or follow at a fixed distance without regard for the angle) - all such ways to follow, like *FollowInFixedFormation* inherit from abstract *FollowAbstract* class.

Also there are various ways to move, and all such ways, like *MoveStraightLine*, inherit from *MoveAbstract*. To achieve required follow behavior, the game designer has to assign *WithDestination* and some version of follow and move to the game object.

Calling methods on components directly

We used this mostly for cases when the other component is required by the calling component and it is marked so by the *[RequireComponent]* attribute.

The rationale here is that if some component declares that it cannot work unless another component is present, it also expects this component to provide some public interface. Using this interface must be safe.

Sending messages

Unity3D allows to send a message to a game object with *SendMessage(methodName)* method where the name of the method must be passed as a string parameter. One of the scripts on the game object which has such a method will execute the method.

One can also enforce that the message is actually processed by some script by passing an additional flag *RequireReceiver*. If no script can handle the message, an exception will be thrown.

Unity3D also offers a similar method *BroadcastMessage(methodName)* which sends the message not only to the game object itself, but also to all game objects attached to it's transform as children in the game scene, to their children and so on. Also, the message will be processed by all scripts which can process it, not just one.

An obvious problem with these both methods is that the name of the method is passed as string. In case refactoring is done and methods are renamed, the whole system with sending messages breaks down, but no errors will be reported in compile time and maybe even in runtime.

For this reason, sending message was used as little as possible, only in places where it couldn't be avoided.

Firing events

In some cases messages or data exchange can be viewed as events or callbacks - messages which are called rarely (maybe even once) after the state of something has changed.

In C# this system is implemented in form of events and delegates. The C# implementation allows the sender to be completely agnostic to who will receive the message, if at all.

This implementation is conceptually more advanced than the one used in other languages or frameworks. For example, in Java, when programming natively for Android, one defines interfaces and may let classes implement these interfaces so that they could be used for processing the message. A button can be given an anonymous class which implements the *View.OnClickListener* interface, so that when the user clicks the button, the *OnClick* method of this object is called. In Java thus only 1 object can be notified. If many objects were to listen to the button click, the button would need to have a list of listeners and notify all of them.

In C# this mechanism is in-built and has different, more compact syntax. A handler can be added for the event with += operator and removed with -= operator. When the event is fired, all handlers are called (in random order).

In Fantasy Arena the mechanism of events and handlers was used extensively to reduce coupling between modules.

For example, all GUI-related scripts handle events which happen inside game actors to update displayed data, but the game actors themselves don't know about the existence of the GUI scripts. The actors only fire events letting anyone interested to receive the updates.

Changes in the GUI system would thus hardly require any changes in the game logic scripts.

3.2.4. Controlling animation states

Behaviors of game actors sometimes require that the 3D model which represents this actor in the game shows an animation (for example, the mage should have a special animation for casting spells).

Unity3D offers a handy *Animation* component which allows switching between different animations the model has. When a 3D model is imported into the editor, its animations are automatically recognized and listed, and a prefab is created with the model, *MeshRenderer* and *Animation* components attached. The game designer then proceeds with assigning scripts and other components to the prefab.

A problem which arises later is that if the current 3D model has to be replaced with a new one, the editor creates a new prefab for it instead of just switching the existing model with the new one. In the new prefab, the scripts and other components must be assigned anew, corresponding values re-set and, worst of all, any references to the old game object must be replaced with references to the new object.

Each time a 3D modeler sends a new version of a 3D model to the game designer, the whole game object must be reconfigured.

The solution is to attach game objects which were created automatically from 3D models as children to dummy game objects which should act as the main object and contain all the scripts, *Collider*, *RigidBody* and other components.

Sometimes 3D models may consist of many parts and present a hierarchy of objects initially. The standard model of an engineer (which Unity3D provides as a standard asset for learning the engine and for usage until production models are available; this model was used as a stub for all units and mages) - consists of 2 meshes (the engineer himself and a wrench he is holding in hands) attached to a parent object which contains the Animation component. The angry bot model from the demo Angry Bots game which is also delivered for learning has a *Shadow* and *mine_bot* children objects, the *Animation* component is in *mine_bot* child and not in the parent object.

From the point of view of the programmer who writes the scripts, this variety of ways how the 3D model can be structured presents an issue. The script may require a reference to the *Animation* component so as to be able to tell it to switch to a different animation, but the *Animation* component may be in a child object, or even a few levels below in hierarchy. It is not obvious where this component may reside and the location may change if models are replaced.

A solution could be to create a public variable of type *Animation* in the script which needs to access the animation so that the game designer had to establish the link manually by dragging the part of the model with the *Animation* component over the link.

This would solve little: each time a model is updated, the designer would have to look though all game objects using it, look if any scripts on them reference the model and update the links if needed.

The solution we finally came to was to make an abstract class *ControlAnimationInAttachedModel* and let all scripts which need to change animations of the host game object inherit from it. For example, *MoveStraightLine* would need to switch the unit's animation from "idle" to "walk".

This class provides the subclasses a protected method *SwitchToAnimation(animationName)*. It takes care that the actual model and its animation component are found in one of the child objects.

This system doesn't cover the case when the model may have 2 or more separately animated components (for example, in MMORPG games the upper part of the character is animated separately from the legs, so it can jump and cast a spell simultaneously). But this use case isn't present in the game since mages aren't allowed to move while casting.

Yet there still remains a problem with animations which can't be solved easily. In Unity3D toggling animations can be done only using the name of the animation as a string.

If the 3D modeler changes the name of the animation and the programmer doesn't know about it, the error will manifest itself only in runtime: the character will just not switch to some animation state. This may be a rare state or a very fast, hard to notice state, so the error may stay undetected for a long time.

3.3. Selected game features

In this section some of technically challenging or otherwise interesting features of the game implementation are described.

3.3.1. Processing touches

In RTS games the units are controlled by tapping on them to select them and then tapping on a spot on the map to order them to move there or an enemy to attack. In such model the script which handles the touches would only have to save what unit is currently selected. The whole logic would fit into a few nested if-else statements.

This system is however complicated in Fantasy Arena by the fact that the mage can also cast various spells by first selecting a spell and then tapping at the target. It can also place towers at the tapped spot after the required tower type has been.

Implementing this extended logic within the same script would require that the script processing user's taps stores the currently selected spell or the currently selected tower.

If an interface has to be developed for setting waypoints of patrol route for a unit in patrol AI mode, the same script would have to know that a unit is selected and currently it's AI patrol mode is being configured, and how many waypoints and where have already been placed (to disallow placing 2 points at the same spot).

To make things even more complicated, Unity3D handles mouse clicks and touches on touch screens in a slightly different way, and we want to have a fully cross-platform game running on both desktop and mobile platforms.

It becomes obvious that handling taps may require at least a few scripts. In Fantasy Arena the following scripts handle touch events:

- *TouchScreen* utility script does low level processing and provides a generic interface for reading out touch event or mouse clicks (which for the game should be indistinguishable). It fires *OnTouchDownEvent*, *OnDragEvent*, *OnTouchUpEvent*, *OnTapEvent* and *OnTouchCanceledEvent* events when corresponding states of the touchscreen or the mouse are detected.

- *RTSCamera* script handles *OnDragEvent* of the *TouchScreen* and drags the camera, also taking care that it never moves off the map.

- *RTSPointer* script handles all the events of *TouchScreen* and uses them to control the game. To make the control as modular as possible, *RTSPointer* itself is just a finite state machine (FSM) which can switch between different states. It is up to the state to provide custom logic of handling user touch input. Each state also knows only as much about the rest of the game as is needed for it to process the touches. All states subclass abstract *RTSPointer.State* class. States in the game so far are shown in Table 1, more can be added later.

Table 1. States of RTSPointer

State	**Description**
NothingSelected	lets to select friendly units, towers and the mage
IssuingContextOrders	processes the usual case when the second tap would issue a contextual order
PlacingTower	allows to place the tower after the tower type has been selected from the towers bar.
TargetingAbility	allows to order the actor to apply the selected ability on the target.

The mage, various types of units and towers may differ in what they do in reaction to touches in *IssuingContextOrders*.

For example, a second touch on the empty spot means nothing for a tower since it cannot move. For a unit it means a movement order. For a mage the second tap may mean an order to capture a tile if it is a tile outside territory. Instead of doing all these complex checks inside the *RTSPointer*, the touches are passed further to a corresponding subclass of *GameEntityController* class which the selected actor has

(*MageController*, *UnitController*, or *TowerController*).

3.3.2. Orders queue

If the player orders the mage to capture a tile but it's outside of the range of *Capture tile* spell, the mage should first move up to the spot from which it could cast the spell and then cast it.

This and other cases lead to a requirement that game actors should not only be able to just execute the current order (e.g move somewhere) but also have some memory where a whole sequence of orders can be stored.

In Fantasy Arena every *GameEntityController* (a component on a mage, unit or tower which handles controlling of these actors by the player) internally has a queue of orders. As soon as the currently executed order is completed the next order in the queue is taken. While an order is executed, the AI of the game entity is turned off.

As soon as the whole queue is finished, the game entity returns to default AI mode.

All orders inherit from abstract *Order* class. Currently used orders are *MoveOrder* and *PerformAbilityOrder*, more can be added later.

Orders can visualize themselves to provide a clue for the player on what the actor is currently doing. For example, while executing a movement order, a marker is displayed over the target spot.

Orders queue makes it easy to implement a more precise control of movement of unit common in RTS games when a series of movement orders is queued.

3.3.3. Map generation and presentation

Unity3D comes with a powerful terrain editor with brushes to paint heights, place trees and vegetation, draw semi-transparent textures. This editor creates a *Terrain* object for the game scene. The in-built *NavMesh* component in Unity3D 4.x Pro can generate graphs for pathfinding on *Terrain* without much extra effort required.

Unfortunately these powerful tools cannot be applied in 2 cases:

- for massive multiplayer online games with gigantic maps. Unity3D only supports a single Terrain object per game scene. Some game developers have attempted to use multiple terrain objects but the welds between neighboring terrain objects aren't tidy enough for a commercial game. This has been one of the main reasons why Unity3D isn't often used for MMORPG games.

- if random terrain has to be generated at level start. Obviously, the whole terrain object is pre-configured by the 3D modeler and is stored in the game build as an asset. Like textures or sounds, it cannot be modified in runtime.

Another important shortcoming is that *Terrain* object is not implemented efficiently enough in Unity3D and causes significant drop of frame rate, especially on mobile devices.

A general practice for mobile games is thus to use common 3D models for terrain and resort to un-built Terrain object only if extremely detailed terrain is required (which is rarely the case in games).

In Fantasy Arena the maps have to be generated randomly, so *Terrain* could not be used. Instead, a custom Hex-grid based map system was implemented which attempts to build a visually pleasing map from a set of hex-shaped 3D models for each type of terrain.

Hex grid is often used by strategic games because it doesn't require the game designer to create special rules for diagonal movement, which may be used in games with a square grid.

A hex grid also allows to much easier show a realistic, nearly circle-shaped sight or attack range of units.

On a square grid basically 3 policies can be used when calculating range between tiles: Manhattan-block distance (*x*+*y*), max distance (*max{x, y}*) and Euclidian distance (*sqrt(x*x + y*y)*).

Using Manhattan-block distance results in positional games where units are more static and actually can't help each other much in battle because their range is too limited.

Using max distance results in dynamic games where there are more possible moves and the situation is less predictable (for example, chess). In both cases the range of the units doesn't look realistic: it's either a diamond or a square.

Using Euclidian distance produces realistic-looking ranges but the rules are then not very easily understood by players (for simplicity, a diagonal of the square may be considered equal 1.5 and not sqrt(2) of it's side).

Hex grid provides a tradeoff between complexity of the grid for the player and realism.

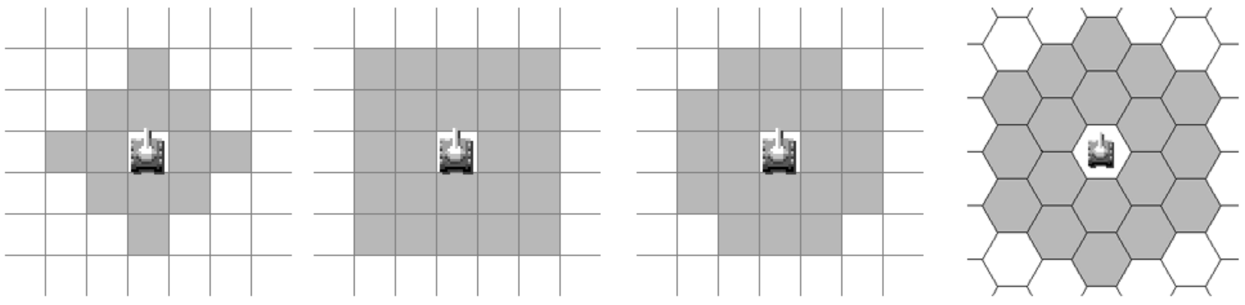


Figure 26. Tiles in attack range of 2 on different maps: (from left to right) square grid with Manhattan-block distance, max distance, Euclidian distance and a hex grid.

Hex grid can be represented in memory in many ways, but the most efficient one is to represent it in a 2-dimensional array as if it were a square grid, but with odd columns shifted down by 1/2 of a tile. Obviously, the presentations with odd columns shifted upwards by 1/2 of a tile or with rows shifted instead of columns are all equivalent.

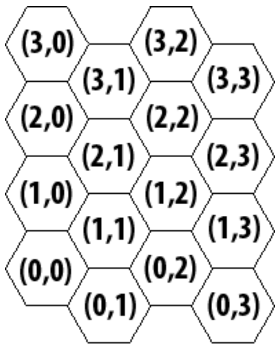


Figure 27. Presentation of a hex grid as a 4x4 2-dimensional array

Algorithms on a hex grid then differ from similar algorithms on a square grid mostly by the need to consider if the current tile is in an odd or even column and depending on that skip the diagonal neighbors above or below (neighbors which would be in the 2-dimensional array if it represented a square grid).

To improve the presentation of the map, an artistic requirement to the game was that tiles of the map visually connect to each other if they are of the same type.

For example, a tile with type "water" must look like a small round lake it it's fully surrounded by non-water tiles but rather as a coast when to the left of it there are ground tiles and to the right are water tiles.

Obviously, each tile type must then have more than one 3D model, and the exact model to be used for displaying the tile must be selected depending on which neighbors the tile has.

The algorithm to select the proper model for the tile depending on the tile's neighbors is implemented by the *HexField* class in *Utils/Math*. It works in the following way:

- for each tile it is recorded which neighbors are of the same type as the tile and which differ;
- each variant of the 3D model of the tile is taken and for each of it's 6 rotations (0 grad, 60 grad, 120 grad, 180 grad, 240 drag, 300 grad) it is checked if the model fits the current neighbors of the tile;
- the variant that fits is rotated as needed and is instantiated in the game to represent the tile.

On a hex field each tile can have 2^6 = 64 possible configurations of neighbors (2 states: "neighbor of the same type as the tile" and "neighbor of a different type" and 6 neighbors in total). 14 variants of the 3D model of the tile are needed to fit all 64 possible configurations.

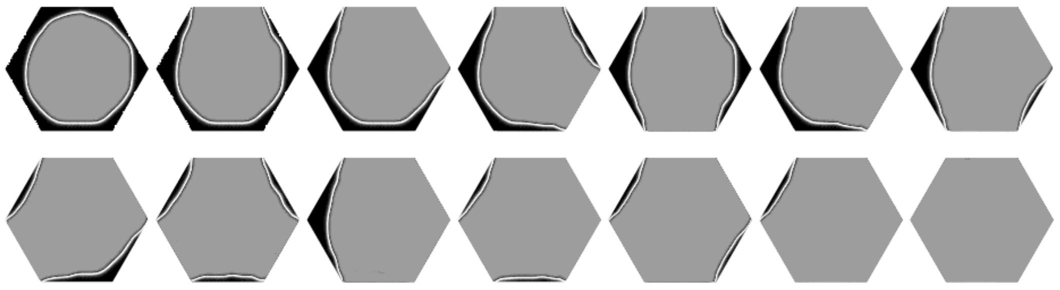


Figure 28. Required variants of a water tile to fit any configuration of neighboring tiles on a hex grid.

Every time the type of a tile changes, not only its own but also models of it's neighbors must be updated.

The same system is used to properly visualize territory of players and fog-of-war.

3.3.4. Pathfinding

Pathfinding in games is a task of finding a path from one point on the map to another without colliding into obstacles or stepping into impassable terrain. The most well-known algorithms for pathfinding are depth-first-search (DFS), breadth-first-search (BFS), Dijkstra's algorithm and A*.

In games mostly A* is used and in rare cases Dijkstra's algorithm. Dijkstra's algorithm is, for example, useful if not just 1 path to 1 target spot has to be calculated but all paths to all tiles on the map need to be pre-cached for fast lookup later - this only works if the graph never changes.

All these algorithms work on graphs, so before pathfinding can be done, a graph must be constructed for the terrain in such a way that every edge is a straight line and doesn't collide with any objects or terrain.

After the graph has been constructed, the pathfinding task reduces to the *single-source shortest path problem on a graph with non-negative edge costs*.

For Fantasy Arena game, the hex grid represents a graph, if each tile is considered as a graph's node and it's relation to the neighboring tiles as edges.

Dijkstra's algorithm

Dijkstra's algorithm (Figure 29) keeps a priority queue of nodes called the *search frontier*. On each iteration, the algorithm takes the first node in the search frontier for processing: it looks at all adjacent nodes of the processed node, takes those that haven't already been processed and adds them to the search frontier (if some of these nodes are already on the search frontier, their position in the priority queue may change).

The nodes in the search frontier are sorted by their distance from the starting node (where unit is initially located). Thus, Dijkstra's algorithm processes nodes in the order of how close they are to the starting node, always taking for processing the closest unprocessed node.

The algorithm terminates successfully when the destination node is finally getting processed. If after lots of iterations the search frontier gets empty and the destination node still hasn't been found, this means that the algorithm has processed all reachable parts of the graph and there is no path from the starting node to the destination node.

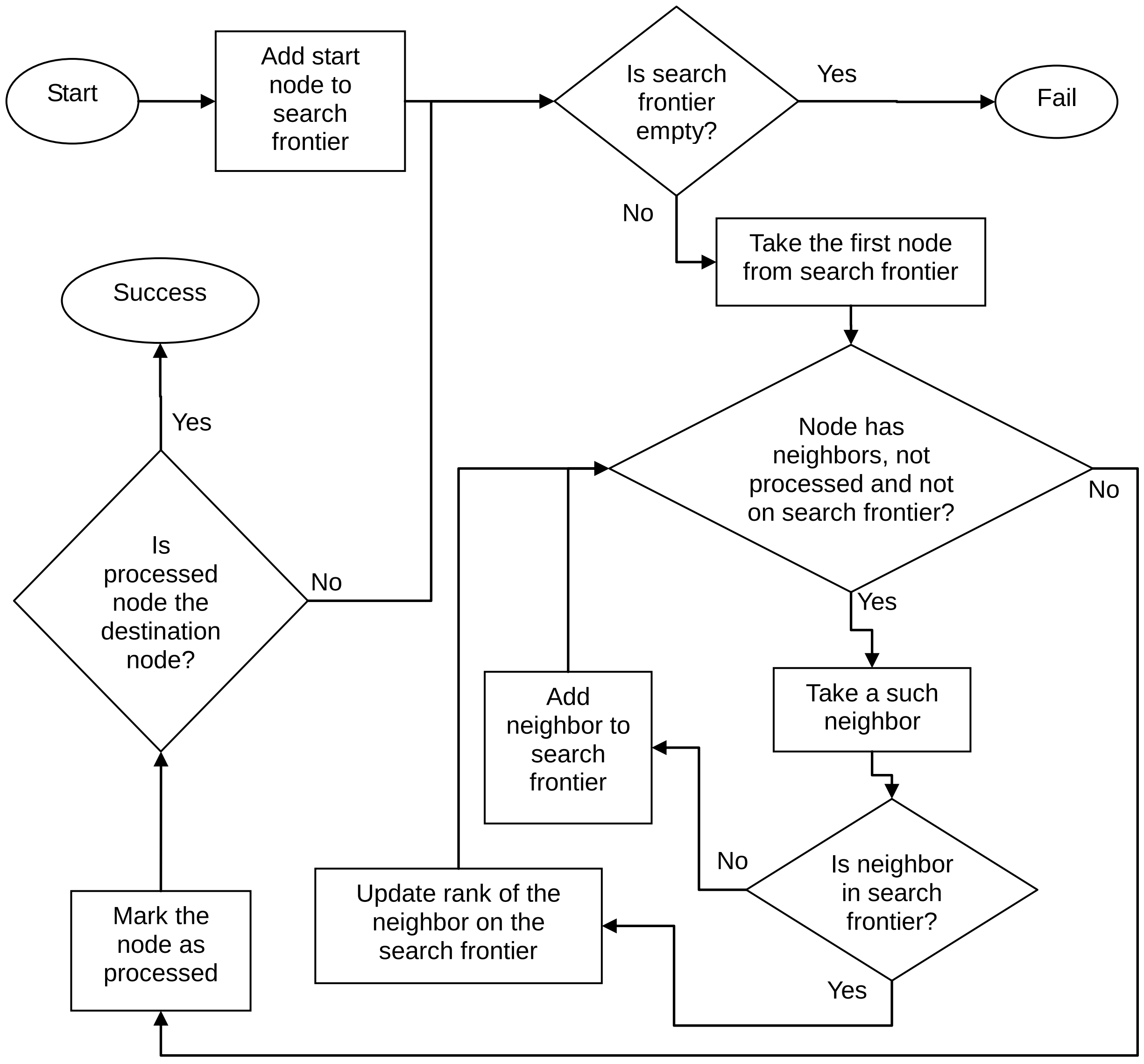

Start
Add start node to search frontier
Is search frontier empty?
Yes
Fail
No
Take the first node from search frontier
Success
Yes
Node has neighbors, not processed and not on search frontier?
No
Is processed node the destination node?
No
Yes
Add neighbor to search frontier
Take a such neighbor
No
Is neighbor in search frontier?
Update rank of the neighbor on the search frontier
Mark the node as processed
Yes

Figure 29. Flow chart of the Dijkstra's algorithm

Dijkstra's algorithm in it's original implementation has complexity of $O(|V|^2)$ where V - number of nodes. Implementation by Fredman & Tarjan [4] uses a Fibonacci heap and runs in $O(|E| + |V| \log |V|)$ where E - number of edges.

The described algorithm of course only checks if there is a path and how long the shortest path is. To get the path itself, the nodes have to store their "parent" (neighbor from which the walker would come to this node). The link to the parent may get changed if a better path to the node has been found and the position of the node in the priority queue is getting changed.

After the destination node is processed by the algorithm, the exact path (a list of nodes) can be constructed by moving back from the destination node to the starting node using the chain of links to parent.

A* algorithm

A* algorithm differs from Dijkstra's only by the way the nodes are sorted on the search frontier.

The problem with Dijkstra's algorithm is that it "radiates" away from the starting node in all directions until it finally stumbles on the destination node. However, in games the graphs are usually planar and the costs of edges roughly correspond to the Euclidian distance between the positions of the nodes in space.

Intuitively it seems more logical to first process nodes which lie somehow in the direction towards the destination node. A* implements this intuition by sorting the nodes in the search frontier by how close they are to the starting node *plus* an estimate of how close it is to the destination node (heuristic).

The distance to the destination node can be, for example, estimated by the length of a straight line connecting the node and the destination node (Euclidian distance).

A* works much better than Dijkstra's algorithm on open maps with few obstacles. On a maze-like map where the shortest path winds across the whole map, their performance converges.

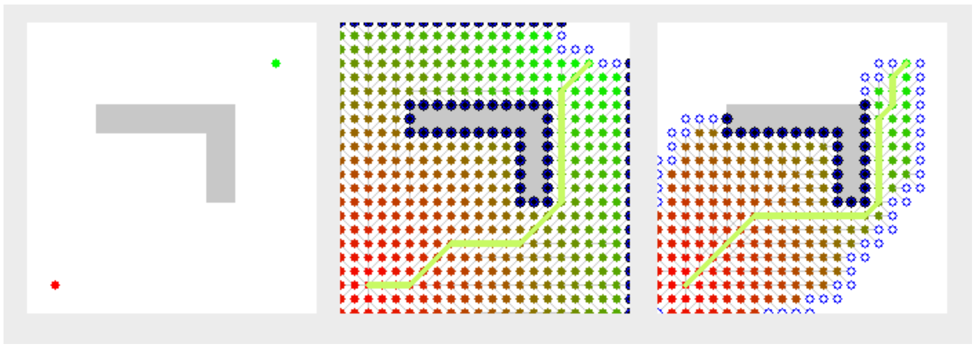


Figure 30. Comparison of performance of Dijkstra's (center) and A* (right)

An example in Figure 30 shows that A* algorithm has to process fewer nodes than Dijstra's to find the shortest path on a square grid with one obstacle from the red dot at the bottom left to the green dot at the top right.

For Unity3D engine, there is a popular plugin *AStar Pathfinding project* [22] which takes arbitrary geometry of the map, generates a graph and stores it in the scene so that it can be used in runtime by the units to find shortest paths. This plugin is very efficient and can use multithreading so that pathfinding could be done across many frames and did not block a single update.

Update of the graph in runtime is also supported to cover the use case when a connection between two neighbor nodes gets blocked by another unit.

Unfortunately, the plugin doesn't work with maps generated in runtime because it requires an already existing graph in the scene when the level is loaded. Generation of the graph must be done in the editor, while the game scene is designed. In Fantasy Arena each map is generated randomly from tiles when the battle starts.

The plugin still supports importing of custom type of graph, so a custom graph generator was developed which produces a point graph for the AStar Pathfinding plugin from a randomly generated hex grid based map.

3.3.5. Modular implementation of abilities

In Fantasy Arena, units, mages and towers can attack enemies or heal allies with abilities. Some of the abilities have quite complex mechanics. For example, a Frostbolt spell should not only inflict frost damage to the enemy, but also slow it down for a few seconds. Some abilities may require different animations than others.

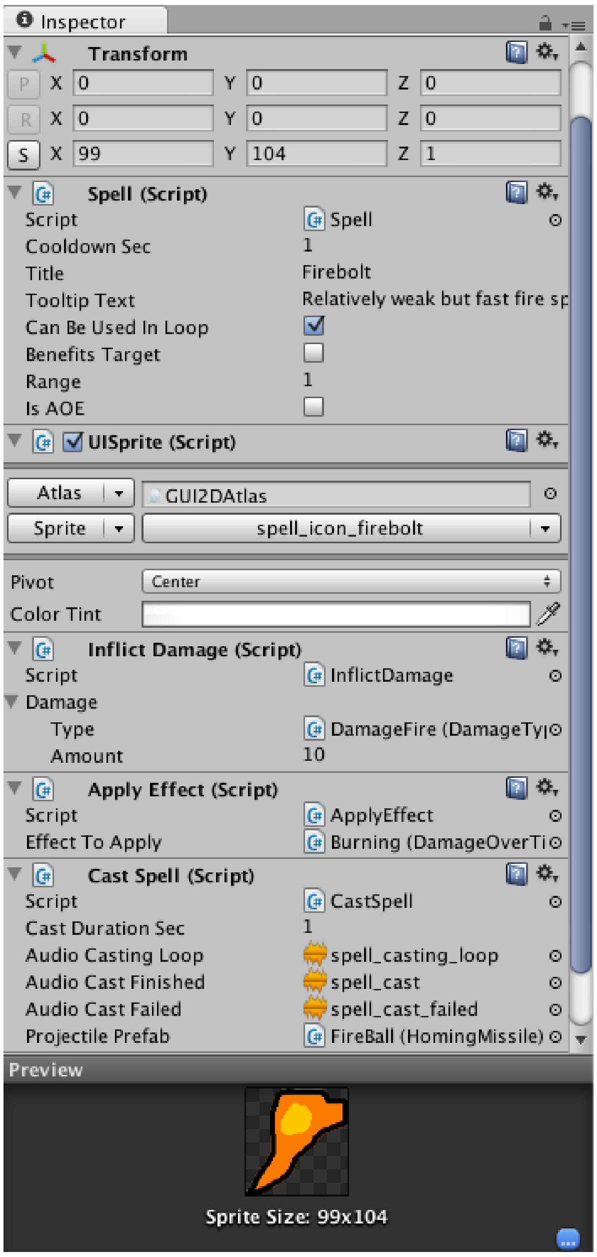
To achieve high modularity, abilities are implemented as prefabricated objects which have 3 kinds of scripts on them.

- an *Ability* or it's subclass (for example, *Spell*) where general parameters are stored about the ability: it's cool-down duration, it's title and description, it's range and if it's a beneficial ability (can be applied to friendly targets).

- a subclass of *PerformAbility* (*CastSpell*, *ChannelSpell*, *AttackInMelee*, *ShootProjectile*) which defines how the ability is performed. For example, for casting spells the duration of the cast is defined, the sounds which are played while the spell is cast, after it was cast or interrupted and the 3D model of the projectile which will fly at the target.

- an arbitrary number of scripts subclassing *AbilityResult* (*InflictDamage*, *DrainLife*, *CaptureTile*, *ApplyEffect* or other) which define what happens to the target after the ability has been performed on it. The *ApplyEffect* must specify what effect it applies. After the ability has been performed, this effect will be instantiated on the target. All effects must subclass *Effect* class. For example, *DamageOverTimeEffect* inflicts damage at regular intervals, *HealOverTimeEffect* heals the target periodically, *SlowEffect* slows the target's movement. After an effect expires, it is removed from the host.

Figure 31 shows the interface to edit parameters of the *Firebolt* spell. The spell is a prefab with 4 scripts on it. The Spell script sets the name of the spell to "Filebolt", cool-down to 1 second and allows the spell to be used for autoattack. *InflictDamage* script defines that upon reaching target the spell inflicts 10 points of fire damage to it. *ApplyEffect* scripts also defines that *Burning* effect will be applied to the target. *CastSpell* script defines the sound effects played during and after casting. It also sets duration of the cast to 1 second. When the spell is cast, a homing missile with *FireBall* model will be created and will fly towards the target. Finally, the spell has a *UISprite* component which defines the icon which will represent this spell in various places of the GUI.



Inspector

Transform

	X	Y	Z
P	0	0	0
R	0	0	0
S	99	104	1

Spell (Script)

Script	Spell
Cooldown Sec	1
Title	Firebolt
Tooltip Text	Relatively weak but fast fire sp
Can Be Used In Loop	✓
Benefits Target	☐
Range	1
Is AOE	☐

UISprite (Script)

Atlas	GUI2DAtlas
Sprite	spell_icon_firebolt
Pivot	Center
Color Tint	

Inflict Damage (Script)

Script	InflictDamage
Damage	
Type	DamageFire (DamageTy
Amount	10

Apply Effect (Script)

Script	ApplyEffect
Effect To Apply	Burning (DamageOverTi

Cast Spell (Script)

Script	CastSpell
Cast Duration Sec	1
Audio Casting Loop	spell_casting_loop
Audio Cast Finished	spell_cast
Audio Cast Failed	spell_cast_failed
Projectile Prefab	FireBall (HomingMissile)

Preview

Sprite Size: 99x104

Figure 31. Parameters of the Firebolt spell, configurable through Inspector panel

3.3.6. Game actor AI modes

As the game requirements specify, units and towers must be able to behave in a meaningful way when they aren't controlled by the player. Also, the player should be able to set the AI mode of units. AI modes will also be used by the AI-controlled opponents.

It is likely that later in development new AI modes will be added, existing ones changed or even removed.

To achieve this kind of modularity, the AI of units is implemented as a Finite state machine (FSM). The current AI state determines the actual behavior.

All states must subclass abstract *GameEntityAIState* class and is free to override

OnEnter(), *OnExit*() and *Update*() methods. *OnEnter* is called when the unit enters the state, *OnExit* - just before it leaves the state and Update is called every tick as long as the unit remains in that state.

For the technical demo only *Pursue*, *AttackStationary* and *StandGround* AI modes have been implemented.

- In *Pursue* mode (Figure 32), the unit keeps attacking the designated target. If the target moves away (comes out of range) the unit will attempt to move closer to get it back into range and keep attacking. The pursue AI mode it turned on automatically when the player gives a unit a contextual order to attack by tapping on the enemy.

- In *AttackStationary* mode the unit would attack only the target selected by the player as long as it is in range. If the target escapes from range, the unit doesn't move and instead switches to default AI mode.

- *StandGround* is the default AI mode for all combat units, towers and the mage. After the unit, mage or tower cannot further stay in its current AI mode (the target is dead or escaped or something else prevents to continue with the AI logic of the mode), they switch to *StandGround*. In this mode, the unit doesn't move and only attacks enemy targets in range. The unit selects the targets itself and can switch them automatically depending on the current tactical situation.

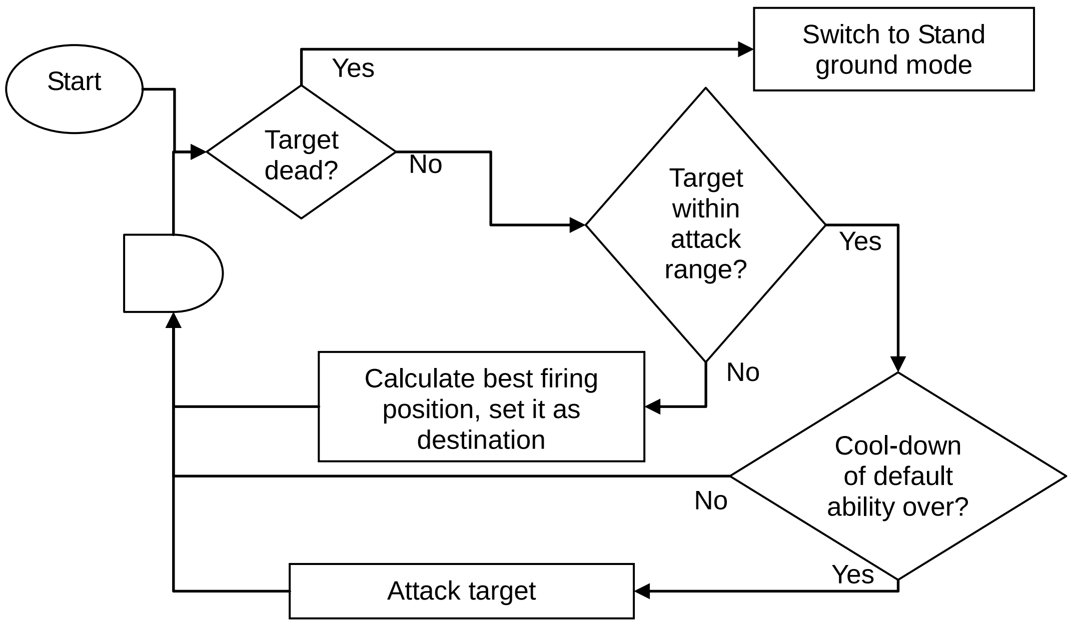

Start
Yes
Switch to Stand ground mode
Target dead?
No
Target within attack range?
Yes
No
Calculate best firing position, set it as destination
Cool-down of default ability over?
No
Yes
Attack target

Figure 32. Flow chart of the unit's behavior in Pursue mode

These AI modes as well as many other modes require that the actor can continuously apply its default ability, also taking into account the cool-down of this ability. This functionality is implemented in the abstract *StateLoopingDefaultAbility* class.

After the ability has been performed (for example, a spell has been cast), its

PerformAbility component fires *OnPerformingAbilityCompletedEvent*. The state handles the event and schedules to re-apply the ability to the target as soon as the cool-down expires.

Automatic selection of targets in *StandGround* and other modes is an interesting topic and is also important for AI-controlled opponents. To select targets, some kind of a *CombatTargetChoicePolicy* is used by the AI mode. For example, *StandGround* by default uses *ChooseEnemyWithLowestLives* policy.

Healing units will use some kind of a *HealingTargetChoicePolicy* in their AI modes for healing. All such policies must subclass abstract *TargetChoicePolicy* class and implement *ChooseTargetFor(actor)* method, which finds the best target for the given unit.

Selecting a target first requires that the policy finds out which units are in range of the unit.

The simple way is just to look though all units in the game scene and check if the squared Euclidian distance to them is smaller or equal than the squared range of the attack. Squares are used to avoid calculating the expensive square root operation.

For very large maps with many opponents this may become a performance issue. To resolve it, clustering may be used. The map is split into regions, and each region keeps track which units are inside it. When checking for enemies in range, the unit would only have to consider its own region and maybe the neighbor regions. Implementation of this feature was reserved for later development.

3.3.7. GUI

GUIText component

Unity3D at its initial release provided a single way to build 2D graphical user interface (GUI) for games. The game designer would use small textured polygons which always face the camera. If their scale and texture coordinates are properly adjusted to achieve perfect match of the size of pixels of the image rendered by the camera and the pixels of the texture, such polygons looked like 2D sprites to the player.

Unity3D provides a *GUITexture* component to render a single texture on the screen as if it were a sprite. It also provides a *GUIText* component to render paragraphs of text using TTF fonts.

This GUI system obviously isn't sufficient for any game because at the very least it lacks buttons. In Fantasy Arena *GUIText* is only used to display FPS counter in the corner of the screen, and *GUITexture* objects are used to display floating messages though there are still some issues with the scaling on font sizes across various devices.

Declarative GUI system

In Unity 2.x an alternative GUI system was added which currently exists together with the old one.

Any script in the game (inheriting from *MonoBehaviour*) can override the *OnGUI()* method which is called every time the GUI is rendered. Inside this method, the script defines the layout of the GUI elements in a declarative way.

On each repaint the whole GUI is reconstructed and can theoretically change once per frame. Unity3D pre-defines a set of commonly used GUI elements: Button, Label, Box and other. To display them, the game must call static methods of in-built *GUI* class like *DrawTexture()*, *Button()*, *Label()*, passing the parameters of these GUI elements (like the text on the button or the texture used for the button's background) on each repaint.

To achieve uniform look of all GUI elements, the game can instead create a *GUISkin* and pass it as a parameter to all buttons and labels.

Overall, our attempt to use the GUI system of Unity 2.x in Fantasy Arena has shown that it has several major issues which make it unusable for any real project:

- The size of the declared GUI elements must be given in pixels and they do not scale for the various screen sizes. One can implement the scaling programmatically by measuring the size of the screen at game start, assuming from that the required scale of the GUI (for example, x1.0, x1.5 or x2.0) and then using one of the custom styles in the GUISkin depending on the current scale. It gets more complicated with fonts because fonts in Unity3D are imported with fixed size (e.g. 14) and can't be scaled. This means that for labels and buttons one of the three font sizes has to be selected in runtime depending on the current GUI scale.

- An important quirk of the OnGUI() method is that the GUI elements drawn in it actually don't block the taps of the user. Even if the tap was on a button, and the button has processed the tap, the tap would still fall though to the game and for example order the unit to move to some spot. To prevent this, a workaround has to be used. A special static boolean variable *wasClickOnGUI* is set to *false* before every frame, and if any button is clicked, in addition to processing the click a statement *wasClickOnGUI = true;* must be added. Inside the game, the camera would first check if the tap was on the GUI, and if so, ignore it. Of course, in a large game the programmer may forget to set the flag for every button. We've found ourselves hunting for all button click processing methods and inserting missing *wasClickOnGUI = true* statements.

- There is no way to define the Z-ordering of GUI elements for processing the taps. For example, if one attempts to show a few dialogs overlaying each other, a tap on an OK button may instead trigger a button in one of the dialogs below. This forces the programmer to keep track of what is currently shown and ignore invalid tap events. Overall, such system is very error-prone, rigid and tedious to write.

- The most important shortcoming is that GUI methods are very slow on mobile devices. Our testing using an Android 2.2 HTC Desire device has shown that a game which runs without any GUI elements at 30 FPS, bogs down to 10-15

FPS as soon as a 5-10 buttons or GUI textures and 5-10 labels are added. On iPhone the frame rate drop is not as significant but still noticeable.

Poor support of the GUI in Unity3D caused creation of several plugins which provide alternative GUI systems.

As of end of 2012, the most popular ones are NGUI and EZ GUI. NGUI is generally considered as an easier to learn and overall a slightly better plugin that EZ GUI.[3] NGUI is also cheaper and has a larger community.

For these reasons we switched from the in-built GUI system to the one offered by the NGUI plugin.

NGUI plugin

NGUI presents GUI as a hierarchy of dummy game objects which may contain various components and child objects to achieve required behavior and presentation.

For example, a button is a dummy game object with a few NGUI-provided scripts to control its animations when it is pressed or hovered over. It can contain one or more text labels as child objects as well as sprites for background of the button or an icon next to the label.

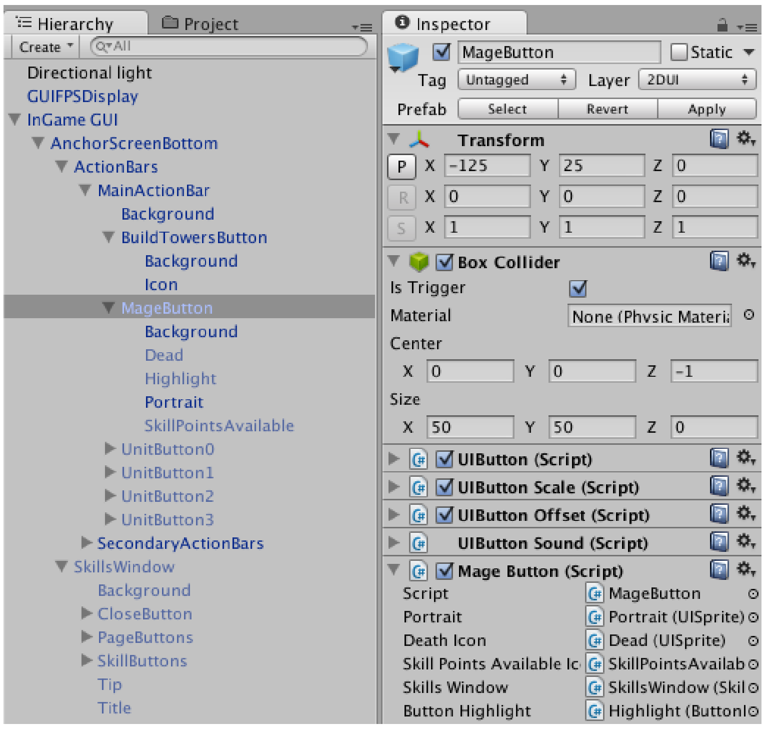
Any GUI elements which should block the taps should have a *Collider* component.

The Z-ordering of the GUI elements can be set explicitly.

Figure 33 shows the structure of the GUI elements in Fantasy Arena. Currently highlighted is the *MageButton* - the button which selects the player's mage. Apart from a *BoxCollider* to detect taps and 4 common NGUI scripts to animate the button it contains a custom script *MageButton.cs* which handles the *OnClick* message to process clicks on the button. *MageButton.cs* has references to other elements of the GUI (mage's portrait, death icon, icon for available skill points and other) which allow the button to manipulate those objects.

For example, the button listens to the *OnDeathEvent* event of the mage and shows the death icon in the handler of this event.

3 See comparision of NGUI and EZ GUI: http://blog.heyworks.com/choosing-gui-framework-for-your-unity3d-project-ezgui-vs-ngui-part-i/, accessed on Dec 26, 2012.



```
Hierarchy                      Project
Create ▾  All
Directional light
GUIFPSDisplay
▼ InGame GUI
  ▼ AnchorScreenBottom
    ▼ ActionBars
      ▼ MainActionBar
          Background
        ▼ BuildTowersButton
            Background
            Icon
        ▼ MageButton
            Background
            Dead
            Highlight
            Portrait
            SkillPointsAvailable
        ▶ UnitButton0
        ▶ UnitButton1
        ▶ UnitButton2
        ▶ UnitButton3
      ▶ SecondaryActionBars
    ▼ SkillsWindow
        Background
      ▶ CloseButton
      ▶ PageButtons
      ▶ SkillButtons
        Tip
        Title
```

```
Inspector
☑ MageButton                         ☐ Static
Tag  Untagged        Layer  2DUI
Prefab   Select   Revert   Apply
▼ Transform
P  X -125   Y 25   Z 0
R  X 0      Y 0    Z 0
S  X 1      Y 1    Z 1
▼ ☑ Box Collider
Is Trigger   ☑
Material     None (Physic Materi
Center
  X 0    Y 0    Z -1
Size
  X 50   Y 50   Z 0
▶ ☑ UIButton (Script)
▶ ☑ UIButton Scale (Script)
▶ ☑ UIButton Offset (Script)
▶   UIButton Sound (Script)
▼ ☑ Mage Button (Script)
  Script                   MageButton
  Portrait                 Portrait (UISprite)
  Death Icon               Dead (UISprite)
  Skill Points Available Ic SkillPointsAvailab
  Skills Window            SkillsWindow (Skil
  Button Highlight         Highlight (ButtonI
```

Figure 33. Hierarchical structure of elements in the GUI system based on NGUI plugin

Texture atlases

A common problem when working with textures in Unity3D is that many small textures saved as separate files may not be stored very efficiently, take more memory and get loaded slower than one big texture.

Usually in games textures and sprites are combined into large *atlases*. A separate configuration file is stored to allow the game to draw the correct region of the atlas where the sprite is stored from the atlas in runtime.

Some game engines provide this feature natively, but Unity3D doesn't. NGUI plugin however provides a widget to generate texture atlases and consistently uses atlases instead of separate textures.

Fonts

NGUI uses an alternative system of fonts. A BMP font together with the configuration file must be imported manually, however no extra effort is required to display fonts of various sizes. The whole text label can be scaled as any game object, using its *Transform* component.

Also, text labels in NGUI allow to change color of font for a substring of the text which is quite often used in games to provide additional information to the player (for example, displaying the name of a difficult task in red or the name of a powerful epic piece of armor in violet).

3.3.8. Data serialization (game save/load)

Unity3D provides no native support for saving or loading game state, so each time the game is closed its state is lost. Unity3D still allows to save data persistently using *PlayerPrefs* class and its *SetInt*, *SetFloat*, *SetString* methods (which on Android platforms use *SharedPreferences* to store the values and on iOS platform *NSUserDefaults*). However, preferences are intended rather for storing options of the game, for example, if sound should be on or off or which level of difficulty should be used across the game.

A usual way to store/restore state is to serialize whole objects into XML files so as to later deserialize them from those files. Since this is such a common operation, frameworks of high-level languages like C# or Java provide in-built support for serialization, for example in C# the *XmlSerializer* class inside *System.Xml.Serialization* namespace can store object's state in XML format unless this object is marked as non-serializable.

Unfortunately Unity3D's in-built *MonoBehavior* class from which most scripts inherit is marked as non-serializable, so XML serialization cannot be used on the classes comprising the scripts.

Instead, the script may contain another class which just mirrors all the fields of the main class in the script and copies all fields from the main class to this "data carrier" class to pass it to the XML serializer. When the game is loaded, the script would then restore its state from the provided "data carrier" object.

This system is of course by no means perfect. A lot of boilerplate code has to be written. If some variables are added later to the scripts, it's easy to forget to add them to the parameters passing routine.

Restoring the scripts using *XMLSerializer* also doesn't solve the problem of how the whole hierarchy of game objects has to be reconstructed when the game is loaded (if such hierarchy is created in runtime and not just comes pre-configured in the loaded game scene). In this case the *Game* or other script which controls the game on a high level would need to implement it's own XML serialization and deserialization routines.

Serialization is required, for example, at the very start of a multiplayer session. The host generates the initial game state and sends it to clients.

Serialization is also needed to load single-player campaigns.

3.3.9. Multiplayer mode architecture

Unity3D uses RakNet engine [35] for networking, but wraps it's functionality in a high-level networking API.

Most of the functionality is accessible as static methods of *Network* class, which provides methods to create a game server, connect to a server, test connection, turn on security and query current network state.

Network events (for example, when a player has connected or disconnected) are delivered to all scripts in the scene as messages.

For synchronizing game state between players Unity3D offers 2 basic mechanisms:

- *Remote procedure calls.* The caller specifies the name of the method, the parameters (which can only be of in-built types like *int*, *float*, *string*) and the receiver (everyone, everyone except him, server only or a specific player). The method will be called on all specified receivers.

- *Network views*. A special component is attached to the game object. The game designer specifies which component should be watched by the view (for example, it's *Transform* component which defines position, rotation and scale of the object in space). Unity3D makes sure that the watched component synchronizes its state across all participants of the game. The updates are only sent incrementally if there are any changes in the state of the component. Network views are typically used for updating positions of players in first-person shooter games.

The game server can be run on one of the devices or as a separate application on a dedicated machine. There are two basic ways how the server of networking game can be organized:

- *Non-authoritative server*. Clients send updates of their state to everyone else and accept updates of the state of others from everyone.

- *Authoritative server*. Clients send updates only to the server. The server first checks updates for validity and then sends the update further to clients including the client who sent the update initially.

When deciding for network architecture of Fantasy Arena, we considered 3 variants which differ by how the players send updates of game state to each other and where the server is run.

Non-authoritative server without a dedicated machine

The server is started on one of the devices. Game objects just have network views which broadcast their movement and updates of state of other components to everyone. Other changes are broadcast with remote procedure calls to everyone.

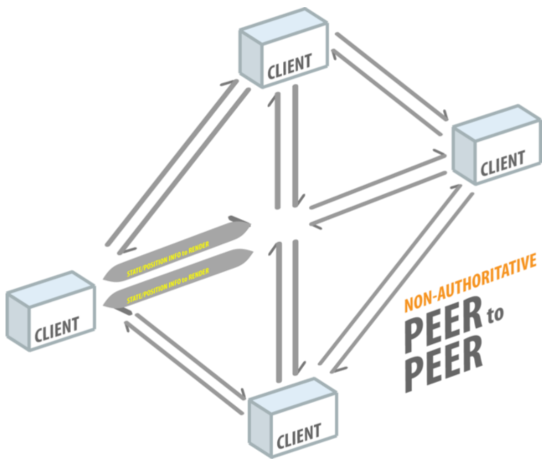


Figure 34. Non-authoritative game server architecture (peer-to-peer) [23]

Advantages:

- easiest (almost trivial) to implement: one just has to add network views to most of the in-game objects;
- no network lag will be experienced for player's own units.

Disadvantages:

- cheaters can build their own clients which, for example, get more gold per tick. This may eventually destroy the community of the game. The problem can be mitigated if submitted scores are checked, however only big cheating can be caught that way;
- a lot of useless updates will be sent, received and discarded (for example, about position of invisible enemy units deep in fog or war). This great increase in network traffic is alone a big enough disadvantage to make this networking architecture unacceptable;
- clients are more likely to run out of synch and get into a situation where their updates contradict each other.

Authoritative server on a dedicated machine

Clients only send player actions to the server. The server processes everything and sends updates of positions of game objects with network views and other events with remote procedure calls. Any command coming from client is checked for validity.

The server is run on a dedicated machine (e.g. VPS - virtual private server). The server creates a map for the game where all clients are placed. The server itself has no rendering.

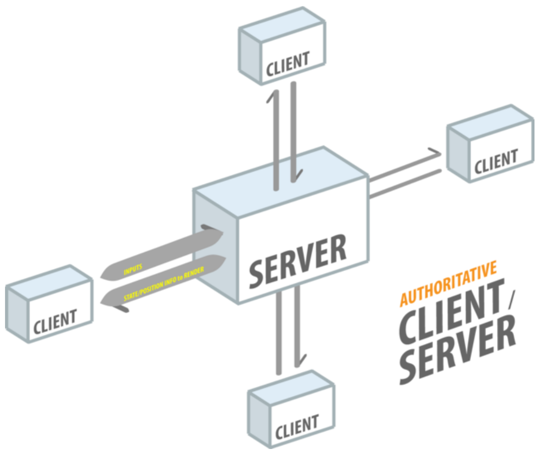


Figure 35. Authoritative game server architecture [23]

Advantages:

- much less hacking is possible on the client side;
- a dedicated server is more stable and more powerful than any mobile device;
- a server with a database would be needed anyway for storing global score so having a dedicated server allows for a more integrated solution (unless some third-party leader boards are used like Apple's Game Center [40]);
- more naturally scaled up to MMO mode;
- can enforce updates to the latest client version by forbidding older clients to connect to the server.

Disadvantages:

- network lag requires use of client *prediction* technique. The game updates itself in response to player's actions as if it were on the server until some critical event like losing lives, capturing a tile or dying which are only accepted from server updates. Updates from server eventually override any local updates. To smooth correction the clients can use interpolation between old position to the corrected one within a short time interval;
- a dedicated server costs money to run. Costs increase as the game becomes more popular and more players attempt to play the game simultaneously. More powerful servers and more dedicated machines need to be installed.
- the server has to be protected from DDOS and other attacks which may require an additional server to just shield the game server. In any case, the costs increase dramatically as the game becomes more popular and more security has to be implemented.
- (specific to Fantasy Arena) since every map is unique and generated randomly, the server can't run many games on the same map. Unity3D can

only run a single scene at any given time, which means that a dedicated server could only host 1 game at a time. A possible solution could be to create multiple maps, sparsely distributed in the scene and keep track which players play on which map. However, this system would only allow at most 50-100 players in the same game scene, because Unity3D only allows 32 network view groups and the network group ID is usually unique for every player or an alliance of players. This disadvantage effectively makes the solution with a dedicated server unacceptable. The limit of 32 network views is actually one of the many limitations which don't let Unity3D scale up well for MMO games.

Authoritative server without a dedicated machine

One of the devices runs the server and receives all updates from other devices, checks them for validity and then sends back only what's needed to each client. This is the classical architecture used in multiplayer RTS games when such games are played over LAN.

Advantages:

- server costs nothing to run, even if the game is very popular;
- relatively easier to add to a single-player game since no separate server application has to be written. Instead, only pieces of network-related code must be added.

Disadvantages:

- cheaters can host games with unfair rules. This can be somewhat negated if submitted scores are checked, however only big cheating can be caught that way;
- multiplayer games run on a user's device are more likely to get interrupted than those run on a dedicated server because devices are weaker and are more likely to loose connectivity. This problem is however not critical since the game sessions will be pretty short (mostly 5-10 minutes) and sudden disconnect will not cause much stress to players. A bigger game design problem is that the hosting player may be motivated to cancel the game if he sees that he is going to loose it. To combat this, special techniques can be used to let game continue even if the host gets disconnected. As soon as clients detect that the server got disconnected, a new server is started on one of the clients, clients connect to the new server and the game continues;
- the host will often drain device resources. The game will run slowly for everyone if the host has a weak device. To solve this, the game should automatically select the most appropriate device on which to host the game among the participants. Before the game is started, the clients can send to each other evaluation of their hardware performance and network connection quality;
- the players will experience even a higher network lag than with a dedicated server since the host would have to use his network connection to process all

network packets for all other clients. The host's network connection thus becomes a bottleneck. To handle this, prediction technique must be used by clients. Also, the device with the best network connection should be selected for hosting the game. ;

- high usage of host's battery may lead to few players wanting to host. This is a game design problem which can only be solved by making the selection of the host automatic. Also, while the host is selected, the battery state of the devices should be taken into account.

- in LAN-oriented RTS games players have to arrange the gaming sessions in advance and make sure that their friends are in the same local network. Unity3D however provides a *Master server* which lets the game to register the server globally and a *NAT punchthrough* technique which allows servers with private IP addresses still be accessible from outside of the local network (this however may not work on strict corporate networks).

Analysis of the requirements of the game leaves the variant with an authoritative server running on one of the player's devices as the only acceptable solution.

Multiplayer use cases

In multiplayer RTS games players typically can search for servers in the Internet or in their LAN and then connect to them. Some servers may require passwords. This system is hard to understand for casual players on mobile devices. Therefore for Fantasy Arena the interface for finding human opponents was based on the typical use cases for network play:

- *Use case 1. New player*. The player starts the game for the first time. This player must not be able to access the network functionality and instead get directed to the series of tutorial single-player levels where basic concepts of the game are described and only some very weak AI-controlled opponents are present. At first launch of the game, the tutorial sequence must start automatically.

- *Use case 2. Casual player.* Has a short time interval (5-10 minutes) in which he would play a game against some random opponents over the Internet. It doesn't matter much for him what exact game and against whom he would play. The only important factor is that this player has a tight time limit and doesn't want to spend any efforts on starting the game. For this player, the game provides a *Quick multiplayer* mode, where the player just selects what kind of teams setup he wants (1vs1, 2vs2, 3vs3 or other) and his time limit (for example, 10 minutes). The player presses start and waits for a short time until the game is assembled and started automatically. The game first connects to the Unity3D master server and checks if there are any games currently waiting for more players to join, with the same setup of teams. If there are any, the one with least still required number of players is selected. If connection to that server fails for any reason, the game tries to connect to another host. If none are found or all connections failed, a new game is hosted automatically. After the required number of players is assembled for the game, players are randomly assigned into teams and the game is started.

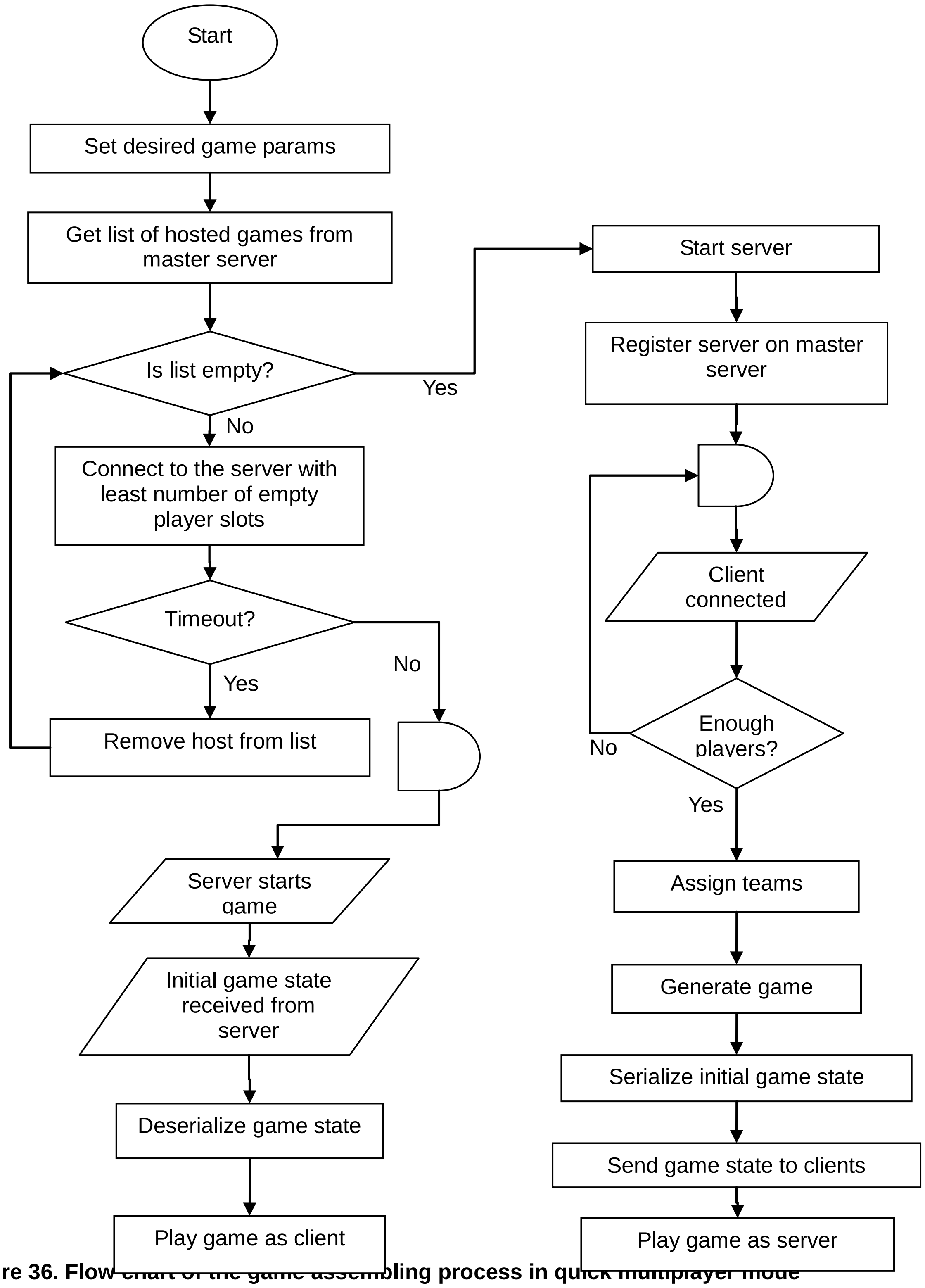


Figure 36. Flow chart of the game assembling process in quick multiplayer mode

- *Use case 3. Hardcore player.* This player wants to play with his friends and has exact plans about which teams and which players have to compete in the game. He also may devote a longer time to the game. For such players a classic Multiplayer mode is provided. The player can either choose to host or join a game. Typically, one of the friends with the best device will host a game, set players limit, time limit and password. Then his friends choose to join a game, find his game in the *game lobby*, enter the password and are forwarded to the teams setup screen. There they select for which team they want to play (there are no restrictions, so, for example, 2 weak players may choose to play together against 1 stronger player). All players press the "Ready" button, and finally the hosting player presses "Start" button to launch the game. While inside a game lobby or a team setup screen, the players can send messages to each other over the in-game chat.

3.3.10. Persistent music player

Playing background music is a trivial task in Unity3D: the AudioSource for the music should be set on the main camera and the Loop flag checked. A problem arises if the music has to be played across different game scenes.

For example, the main menu of the game may be implemented as separate game scene, then the options screen as another scene, the list of levels as another and so on. When the game switches between scenes, the previous scene is destroyed and the new one constructed. Any objects which have been in the previous scene are destroyed as well. Even if the same music is placed on camera objects in all the different scenes, the music will get interrupted and will restart each time the user navigates between game scenes.

To solve this issue Unity3D provides a method *DontDestroyOnLoad* to mark game objects which should not be destroyed together with the game scene. A utility class *PersistentMusicPlayer* was implemented to keep looping background music. To finally stop the loop, *StopLooping* method must be called which just destroys the player.

3.4. Testing techniques

3.4.1. Test-driven development

Test-driven development (TDD) is a commonly used methodology to reduce amount of bugs in software and decrease the costs of finding bugs. The process of writing code according to TDD consists of short cycles:

- before a method is written, a test is created which would determine if the method returns correct results for all principally different inputs;

- the test is run to make sure that it fails (because the tested method is not implemented yet and returns default value);

- the method is written;

- the test is run again. If it still fails, the errors in the method's implementation are fixed immediately. The method is considered complete as soon as the test succeeds.

This methodology is best suited for writing libraries and other reusable parts of code, especially those which take some data as input, process it and return some value as result.

For games, it is used while writing parts of games engine: rendering, physics, audio, script interpreter and other parts. Most of this however already comes implemented in Unity3D engine and doesn't require testing.

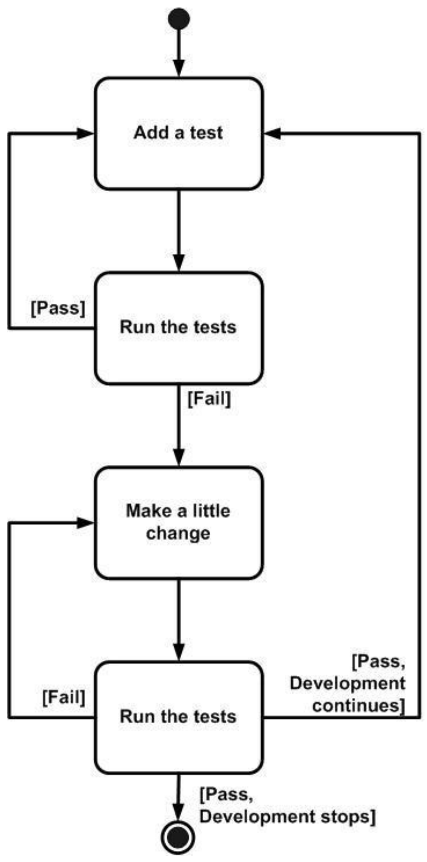


Figure 37. Development flow chart in Test driven development paradigm [30]

While writing game scripts, the scripts are usually highly interdependent, and a lot of objects and settings of the game are global, so it is very hard if not impossible to test pieces of functionality in an isolated way. Also, the *MonoBehavior* class from which most Unity3D scripts inherit is not testable.

Although unit testing is not popular while developing Unity3D games, there are a few plugins and solutions for that:

- *TestStar* plugin [37] - a commercial plugin which extends the Unity3D editor with a panel to create tests and a panel with test results. Its main advantage is that it reloads the whole scene for each test. This ensures fixed conditions of the test. The plugin makes it easier to create test scenes and populate them with prefabs and game objects.

- *UUnit* [38] - provides a *UUintTestCase* class from which all custom tests must inherit. To run all tests, a game object with *UUnitTestRunner* script must be attached.

- *SharpUnit* [39] - a further development of UUnit which also allows the tests to expect exceptions. One can run the tests for scripts inside Unity3D Assets folder without launching Unity3D editor using a SharpUnit.dll.

- *Uniject* [36] - testing framework for Unity3D C# scripts which provides its own set of classes to mimic *MonoBehaviour* and is run completely outside of the engine.

In Fantasy Arena project we used *SharpUnit* plugin to test a few complex classes which do not inherit from *MonoBehaviour*, for example *HexField* - a class which makes sure that neighbor tiles of the same type in a hex grid are properly connected to each other.

TestStar plugin will be used in further development of the game for testing behavior of game objects in cases where no human evaluation is required, for example, testing if a game level was initialized successfully.

3.4.2. Test game scenes

A more common way to test game scripts is to create separate test scenes for pieces of game mechanics (test cases).[4]

Such test scenes are usually tested manually. The programmer just runs the scene, performs a series of actions as if he was the player and makes sure that no errors are written to the console and the events happening in the scene are exactly what he expects.

Testing manually proves to be more efficient because in games it is often much simpler to just run the game and see if "everything is ok" that to attempt to predict what may go wrong and write test cases for this. Many things in games can only be evaluated by a human, for example, if jumping animations looks realistic enough.

For each reported bug, the programmer may create a separate test scene to simulate the conditions in which the bug occurs.

At first the game was developed in a single-player mode only. To test basic functionality of the game, a test game scene was created. In this scene, the game generated a random map of a small size (8x8), placed 2 players (1 human-controlled and 1 AI-controlled) and gave each player a mage and a small piece of territory. The human player also received a single unit and a tower.

Development was organized in a series of tasks. The completion of a task would usually mean that the user now can perform a longer use case. In current technical demo, the single-player test scene allows to perform the following high-level use case without any errors (steps of the use case were implemented and tested from top to bottom):

4 See discussion of the topic on Unity3D forums here: http://forum.unity3d.com/threads/35901-Test-driven-development, http://forum.unity3d.com/threads/38818-Unit-testing-of-code-written-for-Unity (accessed on Jan 5, 2013)

- As soon as the test game is started, a random map is generated and 3D models of tiles are correctly selected to ensure that neighbor tiles of the same type are properly connected to each other. The player can drag the camera around the map.

- 2 players (one human player and one AI player) are created. At a random place of the map a mage is given to each player. Around the human-controlled mage a few tiles are given as territory. For of war is properly opened for the mage.

- The player selects the mage by clicking on him, orders him to move inside his territory by tapping on a tile or order to convert a tile by tapping on a tile outside the territory. Fog of war is updated accordingly.

- For each converted tile the mage receives experience points. After enough points have been accumulated, the mage levels up, a level-up icon is shown over his portrait.

- Player opens the level-up window and selects new spells, units or tower types as well as unlocks additional summoning slots. New spells appear on the quick spells bar. New units of unlocked types can now be summoned. New towers of unlocked types can be placed.

- The player selects one of the summoned units and orders it to move or attack the enemy mage. If the player moves the unit close to the enemy mage, both the unit and the enemy mage start attacking each other with their default abilities automatically. If the unit dies first, a new unit can be summoned into the freed slot.

For multiplayer mode, no specific test scenes were done due to time constraints. Development was organized as a series of milestones which allow the user to perform the following high-level use case:

- The player starts two instances of the game (on any combination of PC, Mac, Android or iOS devices, also two instances on the same computer).

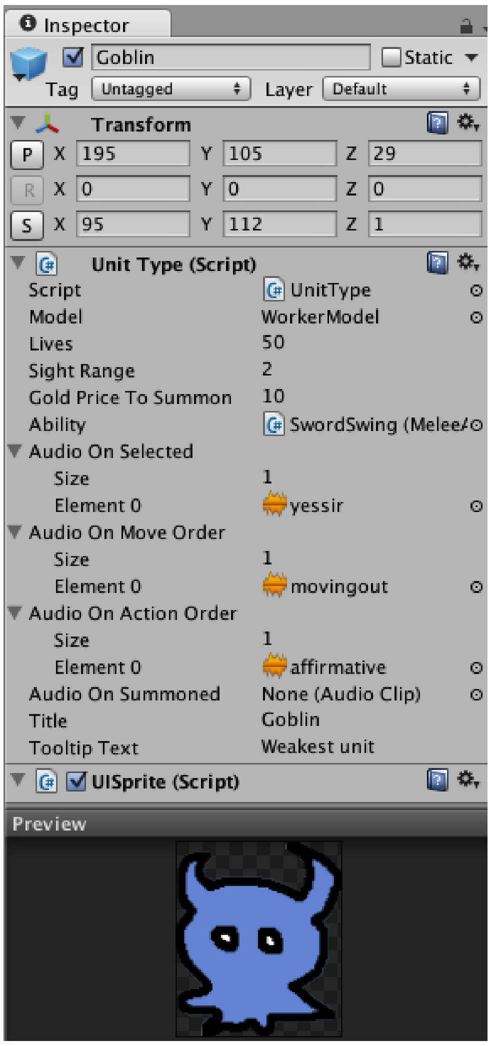
- In one of the instances the player starts quick multiplayer game and waits a few seconds to let the game host a server and register it on Unity3D's master server. After that the player starts quick multiplayer in the second instance of the game. The instances find each other, and a multiplayer game is started.

- A random map of appropriate size for the selected teams setup and session duration is generated on the server, human players and their mages are randomly placed and some territory is given. The game state is sent to the clients, clients properly display the initial game state.

3.4.3. Data validation

Unity3D editor allows game designers to configure game objects using drag and drop. For example, to put a script or other component on a game object, one has to just drag that component over the game object.

If some script has public variables, such variables become visible and editable in the editor. If a public variable is of a subclass of *Component* or a *GameObject* (it is a reference to another component or game object) or if it's an asset, the reference can be established by dragging the component, the game object or the asset over the variable.



Inspector

Goblin | Static

Tag: Untagged | Layer: Default

Transform

P	X	195	Y	105	Z	29
R	X	0	Y	0	Z	0
S	X	95	Y	112	Z	1

Unit Type (Script)

Script	UnitType
Model	WorkerModel
Lives	50
Sight Range	2
Gold Price To Summon	10
Ability	SwordSwing (Melee
Audio On Selected	
Size	1
Element 0	yessir
Audio On Move Order	
Size	1
Element 0	movingout
Audio On Action Order	
Size	1
Element 0	affirmative
Audio On Summoned	None (Audio Clip)
Title	Goblin
Tooltip Text	Weakest unit

UISprite (Script)

Preview

Figure 38. Manually configured parameters of the Goblin unit type

For example, a prefab for the Goblin unit type (Figure 38) has a script *UnitType.cs*, which contains parameters of the unit type. Game designers can edit how many lives the goblin has (50), its sight range (2), price in gold (10), title ("Goblin") and tooltip text ("Weakest unit"). Other parameters are links. *Model* parameter links to a game object *WorkerModel*. *Ability* parameter links to a prefab named *SwordSwing* which has a script of type *MeleeAttack* on it. *Audio On Selected* is an array containing 1 element - an audio file *yessir.mp3*. In the same fashion, *Audio On Move Order* and *Audio On Action* are arrays with 1 elements.

However, *Audio On Summoned* is not linked to anything and the editor shows *None* with the type of the link expected (Audio Clip asset). If the script attempts to access *Audio On Summoned* (for example, to play the sound) a runtime exception will be thrown.

If the game is run in the in-built player, the player can stop execution at that point if option *"Error pause"* is checked in *Console* panel. Otherwise an error message is printed in the console of the in-built player. On the user's device no error will be visible in runtime, so such error it likely to stay unnoticed by the testers.

The need to drag and drop files and components to establish links presents a management problem. Game designers may just forget to establish some links, because a game can grow very large and complex and there are thousands of links that must be checked.

Moreover, during development game resources get added, removed or renamed. If the asset is first removed (e.g. by just deleting it from the folder in a file browser), the editor notices it and sets all references which previously pointed to it to *None*. Even if the asset is restored later, the references remain *None*.

In a similar way, if the asset is renamed outside the editor (e.g. by using a file browser or if another team member renamed it and we received an update from the SVN repository), the links to this asset get broken as well.

To check if any links got broken, the game designer would have to go through all prefabs and all game objects in all scenes, select them with the mouse, and look through all of it's components in the Inspector panel to see if any of the links became *None*.

In fact, in many cases links actually can be none, for example in the case of a Goblin unit an empty link may mean that no sound should be played when the goblin is summoned.

The game designer who does the checking would thus need to know for each of the links if they are allowed to be null or not. In the end, the fear to break links may lead to a situation where the members of the team are afraid to rename anything or move assets at all.

Poor naming or organization of assets into subfolders may have arisen in the early stages of the project when it was not yet clear which features and assets will be present in the game. It will not be fixed as the number of assets grows. The project eventually turns into a big unmanageable pile.

The solution to this problem we used in Fantasy Arena project was to make every script check all links inside it for null as soon as the script is initialized.

Every script in the game has to subclass custom *CheckedMonoBehaviour* which extends the in-built *MonoBehaviour* class and ensures that all the checks are done. Special attributes *[CanBeNull]* and *[CanBeAddedInRuntime]* disable the links checking for a single link or for the whole script.

Similar checks are done for uninitialized numeric and string values as well as arrays. Unless an *int* or *float* variable has a custom *[CanBeZero]* attribute, an error is reported if it's zero (in games most parameters that a game designer would configure are never zero).

Unless a string variable has a custom [*CanBeEmpty]* attribute, it must consist of at least 1 character because strings usually stand for names, tooltips or descriptions and they are never empty.

Arrays must have at least 1 element unless they have a custom *[CanBeEmpty]* attribute and they aren't allowed to contain null elements unless they have a custom *[CanHaveNullElements]* attribute.

Introduction of this system of resource checking allowed us to find dozens of broken links across the project at the first run. It later proved, that after introduction of any new feature into the game there always were links or values which we forgot to set up manually - they would otherwise be found much later, maybe even after release.

Working with prefabs presents a problem for validating data: Unity3D never calls any common initialization methods on a prefab because it is actually not a game object but instead just an asset, like a texture.

To ensure that the game rules are still checked for validity at game launch, the Rules script, at the first time it is accessed, runs through all referenced scripts and calls the initialization method of *CheckedMonoBehaviour*.

To guarantee that the error checking reaches all prefabs in the game, the *CheckedMonoBehaviour*'s initialization method recursively broadcasts a message to all referenced *CheckedMonoBehaviour*'s (to avoid infinite loops the initialization method immediately returns if it has already been called before).

3.4.4. Linking required components

Some components in the game may require that the game object also has some other components so that this component could work.

For example, *GainsExpWhenTerritoryGrows* script ensures that the object gains experience points when the player controlling this object (the owner) gains territory. Obviously, this script will only work if the object on which it is placed can receive experience points in general (not only for expanding territory but for anything else like killing enemies), so this script requires that the object also has a *Leveling* script.

In runtime the *GainsExpWhenTerritoryGrows* script needs to access the *Leveing* script to tell it that now some experience points must be added. To get a reference to the *Leveling* script it will use the in-built method *GetComponent<Leveling>()* or a more robust in-built method *GetComponent(typeof(Leveling))* which will find the component even if it's not exactly a *Leveling* class but some subclass of it.

The problem which may happen here is that the game designer has forgot to assign *Leveling* script to the object. The returned reference will be null and raise a NullPointerException as soon as the script attempts to call any methods of *Leveling*.

Thus, to write secure code, the programmer should always check for null after calling GetComponent and write an error message if the component is missing which introduces a lot of boilerplate code.

Unity3D attempts to handle this problem by introducing a *[RequireComponent]* attribute for scripts. If some script declares with such attribute that some other scripts must be on the same game object for it to work, at the moment the script is assigned to the game object, all the required scripts are added automatically as well.

Unfortunately this attribute is not sufficient for real-world projects. *[RequireComponent]* only adds missing components when the component is dragged over the game object in the editor. However it doesn't do any checks later.

Components may be missing because there was some refactoring done on the scripts and their structure and dependancies have changed after some of them were already added to the game objects (this happened very often during development of Fantasy Arena due to "*continuous refactoring*" principle [3].

Checking for null of the return value of *GetComponent* method is secure, but some calls may happen very rarely or very late in the game, so the error may stay hidden for a long time and even get into the final release.

The solution we came to was to limit the usage of *GetComponent*. The links to required components on the game game objects are initialized at object creation and checked for null. In combination with object pooling this ensures that checking for missing components is done for most of the objects and very early.

To reduce the amount of boilerplate code needed to initialize references to other components, the *CheckedMonoBehaviour* uses C# reflection and initializes all references to required components automatically. It even initializes arrays of links to required components.

Conclusion

The goal of this thesis was to develop a technical demo demonstrating main features of a cross-platform multiplayer real-time strategy game "Fantasy Arena". The Unity3D 4.x Pro game engine was selected as the basis for the game.

The thesis itself provides the project overview. It starts with some background information about games development and presents peculiarities of strategic games and mobile games from the point of view of a game developer. It then goes on to describe the design of the game, providing some short justification for game design decisions taken during development. Then the thesis describes overall architecture of the game, coves some of the technically challenging or otherwise interesting features that have been implemented and concludes with the testing techniques used in development.

Development of a game in the real-time strategy genre is a complex task. It is especially complex in the case of a multiplayer game. During initial planning of the project this was taken into account, and only a modest feature set was included into the project's timeline which spanned across 4 months: from September 15, 2012 till January 15, 2013. It was also planned that the demo would be delivered with stubs for graphical and audio assets.

The actual development of the project allowed implementing about 2/3 of the initially planned features. The resulting product is still able to demonstrate the basic functionality of the game. The following table displays the planned and actual time it took to implement major tasks.

Table 2. Planned and actual development schedule

Task	**Planned**	**Actual**
Set up Unity3D development environment	15-16 Sep	15 Sep
Project plan and functional specification	17-21 Sep	15 Sep
Design document, define game setting	22-23 Sep	16-18 Sep
Develop basic game architecture: controlling RTS camera, fog of war, selecting and moving units, system of skills, abilities, levels, gaining experience.	24-30 Sep	19 Sep - 1 Nov
Moving units and pathfinding Basic type of attacks: melee, usual projectile, projectile following target. Basic unit behaviors: follow, attack, defend. Implement defensive towers	1-15 Oct	15 Oct - 14 Jan
Develop basic in-game GUI (use stub images), menus, dialogs	16-17 Oct	20 Sep - 10 Nov
Finalize the single-player part and ensure all basic game elements work for the case of a sole player playing on the map, for the	18-22 Oct	11 Nov - 15 Dec

minimal case of 2 unit types, 2 spells and 2 tower types		
Develop architecture of the network game using Unity3d Network class, implement lobby, starting, saving, restoring multiplayer game, handling of disconnects. The game is now playable in multiplayer mode.	23 Oct - 07 Nov	16 Dec - 14 Jan
Implement all unit types described in the design document and their abilities, abilities of the main character	08-20 Nov	Canceled
Finalize implementation of GUI screens including lobby, skills screen, units, towers and spells screens	21-25 Nov	1 Nov - 14 Jan
Particle effects, shaders for any graphical effects, final polishing, bug fixes. The game is fully playable in multiplayer mode	26 Nov - 15 Dec	Canceled
Writing thesis, finalizing documentation	16 Dec - 14 Jan	16 Dec - 10 Jan

Two major factors caused a significant slowdown in the game's implementation:

- a change in programming methodology about a month into the project;
- transition from in-built GUI system of Unity3D to NGUI plugin.

The scripts for the game were initially written without much regard to the size of classes or methods. Even relatively simple modules tended to grow into large, hard-to-read masses of code, typically accumulated inside gigantic "god"-classes with multiple responsibilities and abundant comments which also tended to become outdated.

After a few weeks of development it became clear that the project of such large scale as a multiplayer strategy game may not progress in such way and will inevitably grind to a crawl. The code would become very hard to modify and extend later, as more features have to be added to the game or existing functionality changed.

Therefore, a transition to "Clean code" methodology was made [8]. The code was extensively refactored, methods and classes split into many. Abundant comments in the code were largely replaced with small methods with self-descriptive names.

A few weeks were spent on refactoring. The speed of writing new code also decreased in the following few months. Yet, even after 4 months of development and with about 200 classes, the project is still organized well enough to continue development.

Initially the game's GUI was based on the in-built declarative GUI system of the Unity3D engine. However, after the first attempts to test the game on a mobile device, we found out that frame rate drops too much when GUI elements are rendered.

It turned out that this is a known issue of the engine, and that most game developers use commercial plugins (NGUI or EZ GUI) instead of the in-built system. About a week was spent on learning the NGUI plugin and rewriting the GUI from scratch.

Overall, it was underestimated how much time it takes to learn new technologies. For example, about 2 weeks were planned to learn the networking architecture of Unity3D and to implement the whole multiplayer functionality. In reality, at least 1 week was required to just understand the networking module of Unity3D and evaluate the possible multiplayer implementations of the game.

The documentation about the networking module of Unity3D on the official website is poorly organized and none of the books about Unity3D currently on the market has a chapter about networking. The only documentation we could finally find on the Internet was a tutorial with a few examples in Javascript.

Also, the actual implementation of features often did not follow the pattern laid out in the plan. While writing the plan, we had little knowledge on how these features are going to be implemented, so only some general estimations of the required time were made.

During actual development, it turned out that some features were dependent on other features some of which were not even on the plan. Features which were initially planned to be done in a sequential order were done in parallel and stayed in the "in-progress" state nearly till the very end of the project.

Some features like particle effects or implementing all unit and tower types as well as all spells and all unit AI behaviors were canceled. In fact, by the end of the project, it became clear how much effort a full game of such scale may require to get released. Namely, 1-2 years of full time development by a game studio with at least 2-3 programmers.

Outlook

Since the game was initially planned as just a technical demo, its further development would consist of implementing the rest of the features and replacing stub textures, models and sound files with the real ones.

The first important step in further development would be to adapt the existing features of the game to the multiplayer mode and ensure that all actions in the game and all in-game events are properly synchronized across players.

Since the networking architecture with authoritative server is used, the server must do extensive checking of any data received from the clients to prevent cheating. Network functionality must be placed into separate scripts and must be modular. Ideally the existing scripts (which work for the single player case) should not even have to know about the networking-related scripts.

The next step would be to make a more detailed game design of the units, towers, spells and other pieces of game content. In particular, the tables about all parameters of these game elements must be made. After that it would be possible to start filling the game with actual content.

After the list of required units, towers and other elements is created, a game artist and a 3D modeler should start creating assets for the game: textures, animated models, sprites for the GUI and so on. A sound effects engineer should go through the list of the required sound effects for the spell casting and many other in-game events and create them. A few professional actors may be required to record funny speech for the goblins and other monsters so that they could react to player's commands.

On the programming side, many further features would have to be implemented including:

- game lobby with an ability to create a multiplayer game with customized teams of players;
- spells which affect an area on the ground instead of just 1 target;
- auras, more effects and more types of spells;
- the rest of the required AI behaviors of the units;
- strategic AI of AI players, so that the AI could command his mage and an army of units and compete with human players.

Bibliography

1. Blackman S. Beginning 3D Game Development with Unity: All-in-one, multi-platform game development. Apress; 1 edition (May 25, 2011)

2. Creighton R.H. Unity3D Game Development by Example Beginner's Guide. Packt Publishing (September 24, 2010)

3. Fowler M. et al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional; 1 edition (July 8, 1999)

4. Fredman, Michael Lawrence; Tarjan, Robert E. (1984). "Fibonacci heaps and their uses in improved network optimization algorithms". 25th Annual Symposium on Foundations of Computer Science (IEEE): 338–346. doi:10.1109/SFCS.1984.715934.

5. Goldstone W. Unity 3.x Game Development Essentials. Packt Publishing; 2 edition (December 20, 2011)

6. Hergaarden M. Unity networking; the Zero to Hero guilde. 29-10-2009 (part of Unity3D plugin Ultimate Networking Project, http://u3d.as/content/m2h/ultimate-networking-project/1ut)

7. Nystrom R. Game programming patterns. http://gameprogrammingpatterns.com, accessed on Jan 9, 2013.

8. Martin R.C. Clean Code: A Handbook of Agile Software Craftsmanship; Prentice Hall; 1 edition (August 11, 2008)

9. Martin R.C. The Clean Coder: A Code of Conduct for Professional Programmers (Robert C. Martin Series). Prentice Hall; 1 edition (May 23, 2011)

10. Menard M. Game Development with Unity. Course Technology PTR; 1 edition (January 19, 2011)

11. Murray J.W. Game Development for iOS with Unity3D. A K Peters/CRC Press; 1 edition (July 26, 2012)

12. Patro A. et al. The anatomy of a large mobile massively multiplayer game. MobiGames '12, August 13, 2012, Helsinki, Finland. http://pages.cs.wisc.edu/~shravan/anatomy.pdf, accessed on Jan 06, 2013.

13. Rollins A., Morris D. Game architecture and design. New Riders Publishing (2004).

14. Wittayabundit J. Unity 3 Game Development Hotshot. Packt Publishing, August 26, 2011.

15. Juul J. A visual history of genres and platforms. http://www.jesperjuul.net/ludologist/, accessed on Jan 8, 2013.

16. Google Inc. Android developer portal, dashboard. Platform versions, screen sizes and densities. http://developer.android.com/about/dashboards/index.html, accessed on Jan 8, 2013.

17. Чибриков В. Методы контроля целостности ресурсной системы в процессе разработки игры. http://dtf.ru/articles/read.php?id=60539, accessed on Jan 8, 2013.

18. Springfiles.com. Age of Empires 2 game screenshots. http://springfiles.com/games/strategy/real-time-strategy/age-empires-2-age-kings, accessed on Jan 7, 2013.

19. MobyGames. Dune 2 page. http://www.mobygames.com/game/dune-ii-the-building-of-a-dynasty, accessed on Dec 21, 2012.

20. MobyGames. Herzog Zwei page. http://www.mobygames.com/game/herzog-zwei, accessed on Dec 21, 2012.

21. Activision Blizzard. World of Warcraft official website (US version). http://www.worldofwarcraft.com, accessed on Nov 6, 2012.

22. Granberg A. A* Pathfinding plugin documentation. http://www.arongranberg.com/astar/docs/, accessed on Nov 15, 2012.

23. Christensen R. Efficient Network Design for Large Multi-user Applications and Multi-player Game Systems for the Web. May 13, 2009. http://research.drawlabs.com/en/2009/efficient-network-design-games-and-applications/paper/, accessed on Jan 8, 2013.

24. Google Inc. PlayStore. https://play.google.com/store/apps, accessed on Jan 9, 2013.

25. Apple Inc. AppStore. http://www.apple.com/itunes/, accessed on Jan 9, 2013.

26. Lester P. A* pathfinding for beginners (online tutorial). July 18, 2005. http://www.policyalmanac.org/games/aStarTutorial.htm, accessed on Oct 14, 2012.

27. http://www.tomsguide.com/us/iOS-Apps-Google-Android-Apple,news-16371.html

28. http://www.wired.co.uk/news/archive/2012-10/31/android-games

29. Apple Inc. iOS developer library. http://developer.apple.com/library/ios, accessed on Jan 2, 2013.

30. Ambler S. W. Introduction to test-driven development (TDD).

http://www.agiledata.org/essays/tdd.html, accessed on Jan 9, 2013.

31. Unity3D forum. http://forum.unity3d.com, accessed on Jan 5, 2013.

32. ESRB official website. http://www.esrb.org/ratings/ratings_guide.jsp, accessed on Dec 23, 2012.

33. Yahoo! Finance. Mobile gaming market sees opportunity for major growth as numbers of smartphones expected to reach 2 billion by 2015. http://finance.yahoo.com/news/mobile-gaming-market-sees-opportunity-122000213.html, accessed on Jan 9, 2013.

34. Gamasutra. Mobile game developer survey leans heavily towards iOS, Unity. http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity.php, accessed on Jan 9, 2013.

35. Jenkins Software LLC. RakNet engine documentation. http://www.jenkinssoftware.com/, accessed on Jan 9, 2013.

36. Outline Games. On testability and Unity3D. http://outlinegames.com/2012/08/29/on-testability/, accessed on Jan 9, 2013.

37. Unity3D Asset Store. TestStar plugin description and introduction video. http://u3d.as/content/eye3ware/test-star/2dB, accessed on Jan 9, 2013.

38. Unify Community Wiki. Uunit page. http://wiki.unity3d.com/index.php?title=Uunit, accessed on Jan 9, 2013.

39. Unify Community Wiki. SharpUnit page. http://wiki.unity3d.com/index.php?title=SharpUnit, accessed on Jan 9, 2013.

40. Apple Inc. Game Center. http:///www.apple.com/game-center, accessed on Jan 10, 2013.

List of figures

List of tables