SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

MOBILE GAME PROTOTYPE FOR SAVONIA

Game Development with Unity

provided by Theseus
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE

TEKIJÄ: Sini-Maaret Mustonen

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Sini-Maaret Mustonen			
Title of Thesis Mobile Game Prototype for Savonia			
Date	8 October 2015	Pages/Appendices	28/0
Supervisor(s) Mr. Jussi Koistinen, Lecturer, Mr. Sami Lahti, Lecturer			
Client Organisation /Partners Savonia University of Applied Sciences			

Abstract

The subject of this thesis was the development of a game prototype for Savonia University of Applied Sciences. In addition to the primary goal of creating a prototype suitable for further development into a full mobile game intended for marketing use, the aim of the project was to explore the process of game development and learn more of the tools and techniques used in it.

The needed features were determined in the designing phase at the beginning of the project and divided into two independent theses, of which the second was written by Milla Koivisto. At the same stage the process of learning the use of the tools and techniques selected for the project, including the Unity game development platform, Git version control, and Visual Studio programming environment, was also initiated. The next step was to begin the practical work of programming the game.

The resulting prototype created in the project implements all of the features planned for it, and does it in a manner conducive to the intended future development.

Keywords
Unity, Game, Version Control, Git

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala
Tekniikan ja liikenteen ala

Koulutusohjelma
Tietotekniikan koulutusohjelma

Työn tekijä(t)
Sini-Maaret Mustonen

Työn nimi
Mobiili peliprototyyppi Savonialle

Päiväys	8.10.2015	Sivumäärä/Liitteet	28/0

Ohjaaja(t)
lehtori Jussi Koistinen, lehtori Sami Lahti

Toimeksiantaja/Yhteistyökumppani(t)
Savonia-ammattikorkeakoulu

Tiivistelmä

Opinnäytetyön aiheena oli peliprototyypin kehitys Savonia-ammattikorkeakoululle. Tavoitteena projektissa oli tutustua pelinkehitysprosessiin sekä siinä käytettäviin työkaluihin ja luoda prototyyppi, josta voitaisiin tulevaisuudessa jatkokehittää mobiilipeli Savonian markkinointikäyttöön.

Projektin suunnitteluvaiheessa määritellyt toiminnallisuudet jaettiin kahteen erilliseen opinnäytetyöhön, joista toisen teki Milla Koivisto. Suunnittelun ohessa valittiin käytettävät työkalut ja tekniikat, kuten Unity-pelinkehitysalusta, Git-versionhallinta sekä Visual Studio-ohjelmointiympäristö, ja aloitettiin niiden käytön opiskelu. Tämän jälkeen siirryttiin varsinaiseen ohjelmointityöhön ja samalla jatkettiin tekniikoiden ja työvälineiden tuntemuksen syventämistä.

Lopputuloksena valmistui prototyyppi, joka sisältää kaikki määritellyt toiminnot ja josta kehitystä voidaan helposti jatkaa suunnitellusti julkaistavaksi peliksi.

Avainsanat
Unity, Peli, Versionhallinta, Git

CONTENTS

TERMS 5
1 INTRODUCTION 6
2 TOOLS 7
2.1 Unity 7
2.1.1 Editor 8
2.1.2 Assets 8
2.1.3 GameObjects and Components 9
2.1.4 Prefabs 9
2.1.5 MonoBehaviour 9
2.1.6 ScriptableObject 12
2.2 Git 13
2.2.1 Using Version Control with Unity 14
2.2.2 Workflow 14
2.3 Visual Studio 2015 15
3 DEVELOPMENT 16
3.1 User Interface 16
3.2 Items and Inventory 20
3.3 Save System 21
3.4 Player Controls 23
3.5 Dependency Management 24
3.6 2D Sprite Rendering 24
4 SUMMARY 26
REFERENCES 27

TERMS

API

Application Programming Interface, a set rules on how to interact with the tools or services provided by the software underlying the interface.

Frame

The individual images that make up a video, each displayed for a fraction of a second. In the context of games a frame can also refer to a single loop through the collection of functions that process and modify the game state, and render the frame to the player.

Rendering

The process of programmatically generating an image from a collection of 2D or 3D objects. In games this process is performed multiple times per second to create the stream of images, or frames, which gives the player the illusion of an interactive virtual environment.

Sprite

A bitmap image that represents a single entity such as a creature or a piece of scenery.

1 INTRODUCTION

The client for this thesis project was Savonia University of Applied Sciences, represented by Petteri Alanko, the communications manager. The main objective was to create a prototype of a mobile game that, if further developed into a finished product, could be used for marketing purposes. The basic idea was to create a game that could increase interest in potential new students in an entertaining manner that does not feel like an advertisement. The original concept was a role-playing game in which the player progresses through a story by doing quests, defeating enemies and conversing with other characters. In addition to the traditional combat centered gameplay mini-games and puzzle-like elements were considered for creating a more diverse experience, but were later dropped to simplify the design. Game development is not a part of the curriculum in Savonia, and exploring this process and learning some of the tools available for it were a secondary but important goal for the project.

The development began in an earlier group project for the same client and was continued further with a two programmer team, both focusing on separate areas to allow the project to be divided into a pair of distinct theses. This thesis concentrates on the development of the user interface, items and inventory, saving system, and player controls. The second thesis by Milla Koivisto covers the characters, quests, and the dialogue system.

2 TOOLS

2.1 Unity

Unity is a development platform for both 3D and 2D games with support for over 20 deployment platforms. Its major features include an animation system, Box2D and Nvidia PhysX physics engines, a particle system, graphics, audio, and scripting. The application programming interface (API) is based on Mono, an open source implementation of Microsoft's .NET framework, and it supports C#, JavaScript based UnityScript and Python-like Boo as programming languages. Previously some of its non-essential features were limited to the paid Pro version, but a new pricing model in the Unity 5 release brought all engine features to the users of the Personal Edition which is available free of charge. (Unity Technologies a. 2015-10-05)

With its 45% market share and 4.5 million registered developers Unity has a large community contributing resources ranging from articles, wikis, tutorials and support forums to free and paid assets in the official Asset Store (Unity Technologies b. 2015-10-05). This community support along with good documentation, free license, multi-platform publishing and the variety of features were the reasons Unity was chosen for this project. Another factor in its favor was the ability to use C#, a language already familiar to the programmers in the project.

The rest of this chapter outlines the features of the engine that are the most prominent from a programming perspective, starting with the Unity editor. Then follow subsections that describe some of the essential parts of the scripting API.

2.1.1 Editor

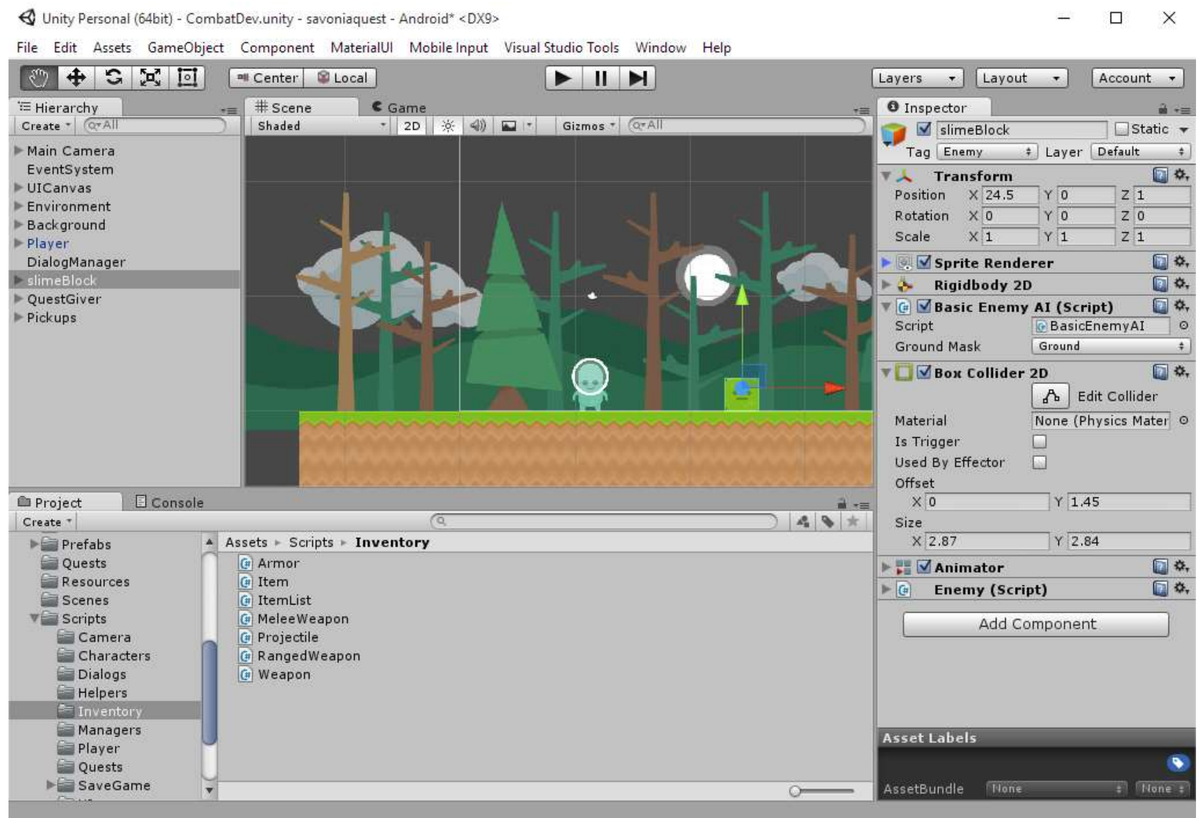

Image 1 Unity editor

A large part of development is done with the Unity editor, as it is the tool used for managing the various parts that make up a game and composing them into levels. The editor consists of multiple windows, the most important ones being the scene, hierarchy, inspector and project views, shown in Image 1. A scene is a container for all the objects that form a single game level, and its visual elements are displayed in the scene view. The hierarchy lists in a tree-view the objects in the currently active scene; the properties of these objects can be viewed and edited in the inspector. The project view displays the contents of the assets folder, which in Unity is where the created project files are stored. In addition to the included views the editor can be extended with custom windows and inspectors to support the needs of each project. (Unity Technologies c. 2015-11-06)

2.1.2 Assets

Assets are the content files used to create a game, such as 3D models, images, sounds and scripts. Many of the asset types have import settings that modify how the source files are translated into the game, for example an image file can be split into multiple sprites, the compression format of an audio file changed or a 3D model scaled to a different size. Assets are brought into the game by either adding them to a scene from the editor or by loading them programmatically at runtime. (Unity Technologies d. 2015-11-02)

2.1.3 GameObjects and Components

GameObject is a fundamental object in Unity that represents game entities such as characters, props and environments. Every object in a game is a GameObject, but by itself it does not accomplish much as it is simply a container for the components that define the actual functionality. Lights, cameras, terrains and particle systems are all examples of built-in components, and many of the engine features are accessed by adding the needed components to GameObjects. The transform component is the most prominent one, as all GameObjects have one to determine their position, rotation and scale in the game world. Scripting is also done through the component system by programming custom components that implement the desired new behaviors. (Unity Technologies e. 2015-11-02)

2.1.4 Prefabs

Prefab is an asset type that allows a GameObject, its components and their properties to be saved. A prefab can be used as a template to create a new GameObject as an instance of that prefab, and if the asset is later modified the changes are reflected in all instances linked to that prefab. Properties and components of prefab instances can be individually overridden, and from the inspector these changes can be applied to the asset or the instance reverted back to the prefab values. Defining reused objects as prefabs allows changes to be made in one location instead of having to go through all copies, possibly in multiple scenes, and modify them one by one. Another important use-case for prefabs is instantiating complex GameObjects at runtime; creating a new GameObject, adding the needed components, and defining their properties takes multiple lines of code while a prefab can be instantiated with one, and the prefab to be instantiated can be quickly switched without the need to modify the script. (Unity Technologies f. 2015-11-02)

2.1.5 MonoBehaviour

MonoBehaviour is the base class for components and therefore the foundation of scripting in Unity. A custom component is created by defining it in a class that inherits MonoBehaviour, and when the component class, also referred as a script, is attached to a GameObject a new script instance is created to operate as the individual component. The built-in components function in a similar manner, only with the difference that the users of the engine cannot modify the underlying component classes. (Unity Technologies g. 2015-11-02)

For code to execute it needs to be called from the program’s main loop, and in Unity the connection between internal and external code is made through the component system. Event functions in the engine classes are used to execute code in predetermined order at runtime by calling the functions in the objects active in the current scene. In practice this means implementing an inherited even function, for example to move a GameObject every frame a component would define the Update function and in it manipulate the position. All custom code does not necessarily have to be contained within components, but for it to execute a piece of code defined in a non-Unity class does need to be called from an event function at some point. (Unity Technologies g. 2015-11-02)

With the game logic distributed amongst multiple component classes and their instances, it is often necessary to know the execution order of the event functions (Image 2). Because of how Unity internally manages its objects, using constructors in MonoBehaviour derived classes is heavily discouraged and may cause unpredictable behavior. Instead, component initialization is handled in specific event functions. Every enabled script instance in the scene receives a call to Awake when it is being loaded; as the order of calling it in multiple objects is random, the function should be used only to set the internal state of the component. After all objects in the scene have received the Awake event and therefore have been initialized, Start is called, and it can be used to set up references and pass data between objects. Both Awake and Start get called only once in the lifetime of a script instance, while OnEnable occurs between the two during the initialization phase but may be called again if the GameObject is disabled and then re-enabled. (Unity Technologies h. 2015-10-05)

An initialized script instance will continuously cycle through a series of events until the instance is either disabled or destroyed, the most frequently used being the three update functions. The first of them is FixedUpdate which is called right before the internal physics update. As the physics simulation needs to run at constant speed regardless of the frame rate, the related events repeat on a fixed time step and thus may occur more or less frequently than frame updates. Usually the main location for game logic is Update, ran once every frame. The final update event before scene rendering is LateUpdate, which can be used to execute code that needs other game logic to be finalized before it. Several other event functions are available and occur in their corresponding points in the execution order as visualized below in Image 2, albeit some only situationally. (Unity Technologies h. 2015-10-05)

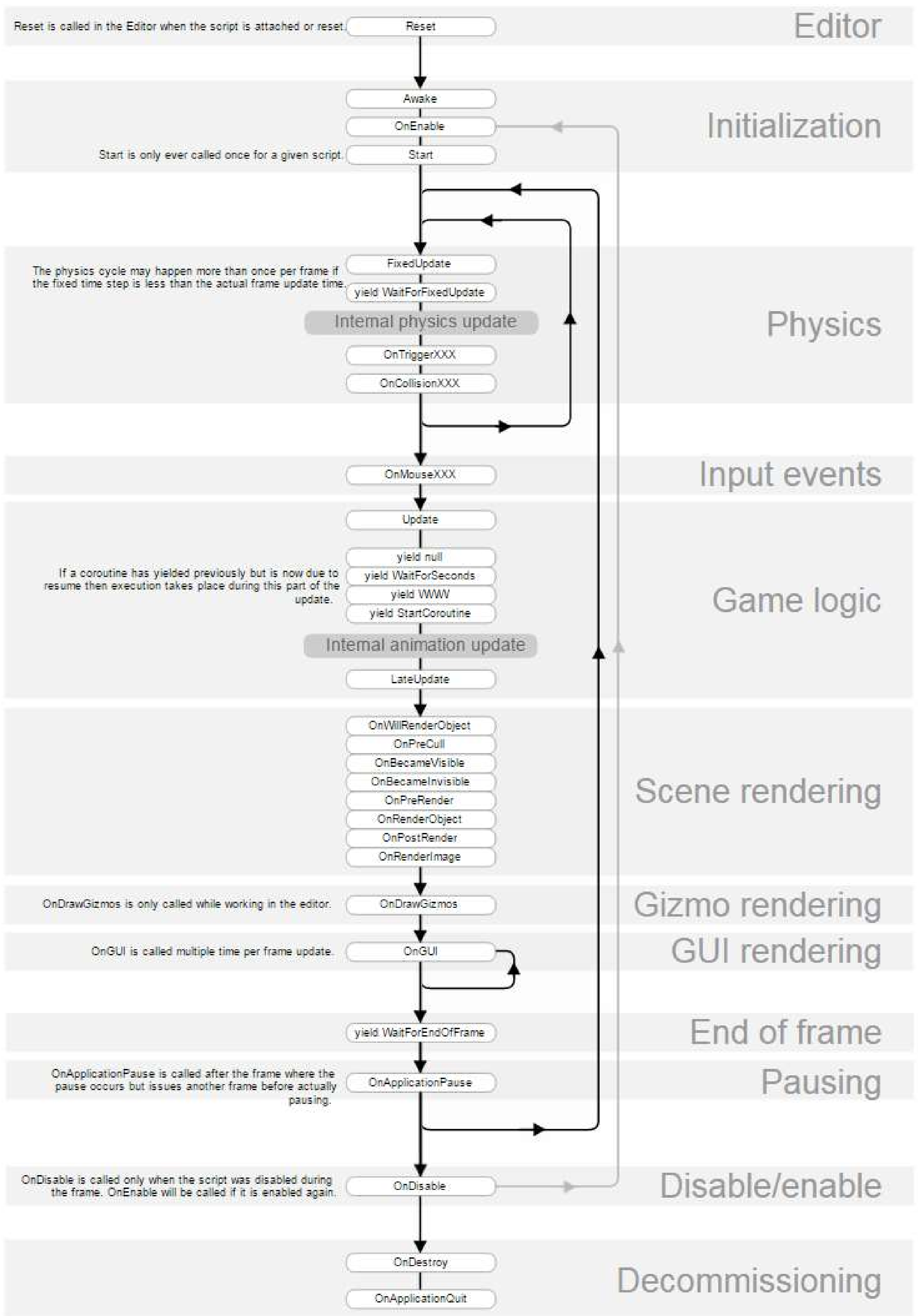

Image 2 Execution order of Unity event functions (Unity Technologies h. 2015-10-05)

2.1.6 ScriptableObject

ScriptableObject is a Unity class for storing data independently of script instances. When information is stored in a component script every instance of that component will receive its own copy of the data even if the contents are the same, which can then raise memory usage to problematic levels if larger quantities are stored this way. In contrast ScriptableObject instances are stored separately and used by reference, allowing a set of data to be accessed any number of times but placed in the memory only once. Another use case for ScriptableObject is creating pluggable data sets. Multiple instances with different variations of the contained data can be created and then used interchangeably. The inventory system detailed later in chapter 3.2 will demonstrate this usage. (Unity Technologies i. 2015-11-02)

As with MonoBehaviour, to make use of ScriptableObject a class needs to inherit it. The instances of that class can be saved as asset files, each storing an instance and its data. The assets are managed in the editor, and like other Unity objects they can be assigned to component variables. A drawback to this approach of using asset files is the inability to save changes or new instances at runtime; the pre-existing objects are initialized from the asset files and changes made during play can not be reflected back to the files, and although new instances from the ScriptableObject classes can be created dynamically it is not possible to store them as assets. (Unity Technologies i. 2015-11-02)

For most of the project Unity did not contain a user interface for creating ScriptableObject assets, and instead this functionality needed to be implemented by extending the editor. In a recent update a simple menu interface was added for creating such assets, paired with a new attribute in the scripting API for marking ScriptableObject classes that should be listed in the menu. Custom extensions can still be used to add more features to the process, for example to set the initial values of the object or to create the asset in a specific folder. (Unity Technologies i. 2015-11-02)

2.2 Git

A version control system (VCS), sometimes also called source or revision control, is a system for recording changes made to a set of files. Tracking file history allows users to find when a change was made, by whom, and why, and to restore the project to a previous state. Multiple concurrent lines of development can diverge into independent subprojects, referred as branches, that do not affect each other, and later the changes from them can be merged back to the main project. (O'Sullivan 2009.)

The data structure storing the history of changes is called a repository, and in its simplest form it may simply be a series of copies of the tracked files. Manually managed solutions or VCS tools that maintain a database of the changes only locally can be appropriate for single developer use, but for projects with multiple collaborators such individual databases are hard to manage. A centralized VCS stores the repository on a single server, from where the developers check out a current snapshot of the files. This allows for more administrative control and makes it easier to see what each project member is working on. A significant downside is that server issues can stop anyone from saving their changes to the VCS. If data corruption occurs in the repository, for example in the case of a hard drive failure, both centralized and local VCSs will lose the entire history unless backups have been made. A distributed VCS is an alternative solution that uses peer-to-peer approach to minimize this risk. Checking out the repository mirrors the whole version history, and in the case of a failure it can be restored from any mirrored copy. (Chacon & Straub 2014, 27 - 31.)

Git is a distributed version control system created in 2005 for the Linux Kernel project. This large and highly collaborative project called for a fast, simple and fully distributed VCS with support for non-linear development. Git was created to meet these needs, and over the years it has grown to a mature, easy to use, feature rich system and it is one of the most used VCSs today. (Chacon & Straub 2014, 31.)

Multiple services offering VCS repository hosting are available, and for git the most used one is GitHub; with over 27 million repositories it is the largest host of code in the world (GitHub, Inc 2015). GitHub offers free hosting for public repositories, which makes it a popular choice for open-source development. BitBucket is another similar service, and it was chosen for this project as it provides free hosting for private repositories as well.

2.2.1 Using Version Control with Unity

A Unity project contains a variety of files and folders, of which only the Assets and ProjectSettings folders and their contents need to be tracked when using a VCS, as the rest are automatically generated. Internally used metadata such as unique identifiers and import settings are stored as separate files beside the assets, and it is recommended to set these meta files to be visible to ensure they get added to the repository. Setting asset serialization mode to text changes the engine's own asset types to be saved in a text format, which allows changed lines to be viewed in the version history and merges to be performed in case of conflicts. (Unity Technologies j. 2015-10-05)

Large binary files such as images, audio and 3D models common in game projects can rapidly increase the size of a repository, especially if changes are frequently committed. This slows down operations to the remote repository, and the use of space can become a problem if a service with limited storage is used for hosting. Other solutions for asset development collaboration in addition to VCS can be used to alleviate this issue, for example storing work-in-progress art and media on a separate file hosting service and only adding the final versions to the repository. (Paolucci 2014.)

2.2.2 Workflow

The secondary goal of the project included attaining better understanding and more experience with Git, and for this reason the repository structure was designed in a manner more typical of open-source projects that need more authoritative control for working with untrusted collaborators. The main repository contains the official copy of the project, and new commits are added to it with pull requests that need to be approved before merging. This extra review step helps to minimize the risk of introducing breaking changes to the central repository. Each developer works in their own personal copy of the main repository, and can freely experiment with both Git and Unity without the fear of data loss.

The individual copy of the main repository is known as a fork, a term used for server-side clones in Git hosting services such as GitHub and BitBucket. The workflow begins with syncing the fork if new commits have been added to the main repository, and then creating a local clone of the fork on the developers' computer or pulling added commits to an existing one. The project files can then be edited with Unity, Visual Studio and other tools. After working on the project the next step is to prepare the modified files for commit by adding them to the staging area. Performing a commit then saves the staged changes and a message describing them. The use of staging area allows larger batches of work to be divided into multiple commits as unstaged changes are not affected when commits are made. Splitting changes to smaller related sets and writing clear commit messages make it easier to find specific alterations to the code, and thus it is advisable to commit frequently when the details are still fresh in memory. New commits can then be pushed to the fork in BitBucket to make them available for the developer on other computers, and a pull request can be created from the web interface to send the commits to be reviewed and added to the main repository.

2.3 Visual Studio 2015

Visual Studio is an integrated development environment (IDE) created by Microsoft that provides tools for application development for Windows, Android, iOS, web and cloud platforms. Supported programming languages include C#, Visual Basic, F#, C++, Python and JavaScript. For game development the free Visual Studio Tools for Unity plugin adds integration and debugging support for the Unity engine. (Microsoft 2015-10-05)

3 DEVELOPMENT

The client had very few requirements for the project, and freedom was given to design and implement the game as the project members saw fit as long as it was kept appropriate for the intended use. The design was based on the role-playing game (RPG) genre because of the project members' familiarity with such games, and for the clear framework of common features that have been established in the history of the genre. In addition to a technical specification a brief outline of the plot was written to assist in detailing the exact features and resources that needed to be created during the project. The planned game features were divided into two distinct sections that separate theses could be based on. The second thesis by Milla Koivisto focuses on characters, quests and the dialogue system.

For producing the visual elements two design students were recruited to the project. Unfortunately because of issues in communication and project management, all of the required graphical resources were not created, and the incomplete set of images could not be used as visually consistent replacements for the missing pieces were not easily available. Therefore the graphics for the game were selected from Kenney Game Assets, a free collection of resources released with a creative commons license to the public domain (Kenney 2015).

This thesis describes the development of the user interface, items and inventory, saving system, and player controls. These systems are not only connected to each other, but also to the other areas of the game implemented separately by the second project member. For this reason the effects of a single change could easily propagate to other systems, increasing the importance of modularity and collaboration between the developers.

3.1 User Interface

At the start of the project the only user interface (UI) system included in Unity was a relatively old, purely script based framework, and in fact the same one that the editor UI and its extensions are built with. Although suitable tool for the editor usage, the system was rather inelegant and inefficient for in-game UI creation. Researching the approaches taken by other developers led to numerous recommendations for the use of third-party tools sold in the Asset Store, but for the small needs of the project the price of a more robust framework was not considered worth the investment. The first version of the UI worked well enough for testing when the game was ran on PC, but on mobile devices the usability was poor because the UI elements and fonts did not scale according to the screen size. As fixing this issue would have taken considerable effort and the exact needs of other interconnected systems were not completely established at the time, the rest of the UI development was left to a later stage in the project. This decision ended up saving unnecessary work as later an update added a new and improved UI system. Some consideration was of course needed before discarding the pre-existing work, but testing the new system clearly demonstrated its advantages, and the built-in scaling features, visual editing and familiar component based structure easily led to the decision to rebuild the UI using it.

The old framework caused tight coupling between the creation of purely visual elements and other UI behaviors, and the effects were further compounded by the decision to abstract the UI code behind a single class in attempt to ease maintenance and usage, necessitating other classes using the UI to be aware of and adapted to the specific implementation. Separating responsibilities was greatly simplified by the new system that split the visualization elements to their own components. Manipulating the properties of the UI elements, data handling and other behaviors could then be divided to smaller more reusable scripts without complicating component interactions. Especially beneficial addition was the event system; it allows both built-in UI components and scripts to receive events such as the user clicking or touching an element, and assign actions to be triggered in response. Most component functions can be used as event callbacks, and neither the calling nor the receiving object needs to be aware of each other. As the callbacks can be set in the editor simple functionality, for example hiding or showing parts of the UI and tabbed windows, can be easily created without any code.

Image 3 Gameplay user interface

Because of the smaller size of mobile screens that can be partially obstructed by touch interactions, the UI used during gameplay was designed to be unobtrusive and minimal. The interface pictured in Image 3 contains buttons for opening the player information window and game menu in the top left and right corners respectively, and between them an element for displaying short notifications from other systems. The notifications run on top of a queue with a delay between each shown text so that if new information is added in quick succession the player will have time to read the messages. After displaying all queued notifications the text returns to a default message such as the description of an ongoing quest. On the bottom left are buttons for movement control, further explained in chapter 3.4.

The menu shown in Image 4, opened from the gameplay view, contains buttons for saving, loading and quitting the game, and returning from the menu. A modified version of the window is used to inform the player of game over if the character dies, with the resume and save game options replaced with a single button for starting a new game.

Image 4 Game menu

Image 5 Player character information view

In the screenshot shown in Image 5 is pictured the player information window and the first panel of the tabbed interface, player statistics. It allows the player to view basic information about their character such as name, location, and attributes that affect actions taken in the game.

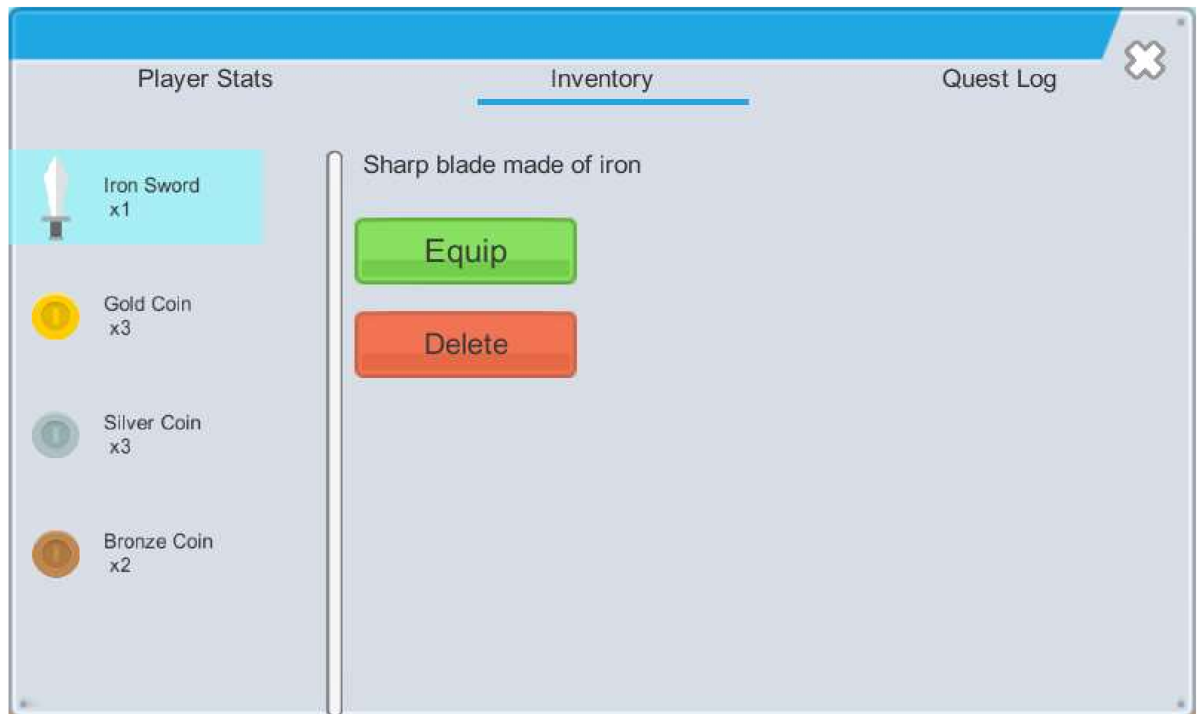

Image 6 Inventory panel

The next tab shown in Image 6 contains the inventory panel that displays the items carried by the player character. The scrollable list of items on the left side of the window is dynamically generated utilizing a collection of UI objects saved to a prefab. The GameObjects for the icon and text are nested inside another object that has components for automatically scaling each entry on the list and making them selectable. When the prefab is instantiated in code the image, name, and count are inserted to the appropriate component variables, and the callback for the selection event is set to a function that will change the contents of the details view on the right.

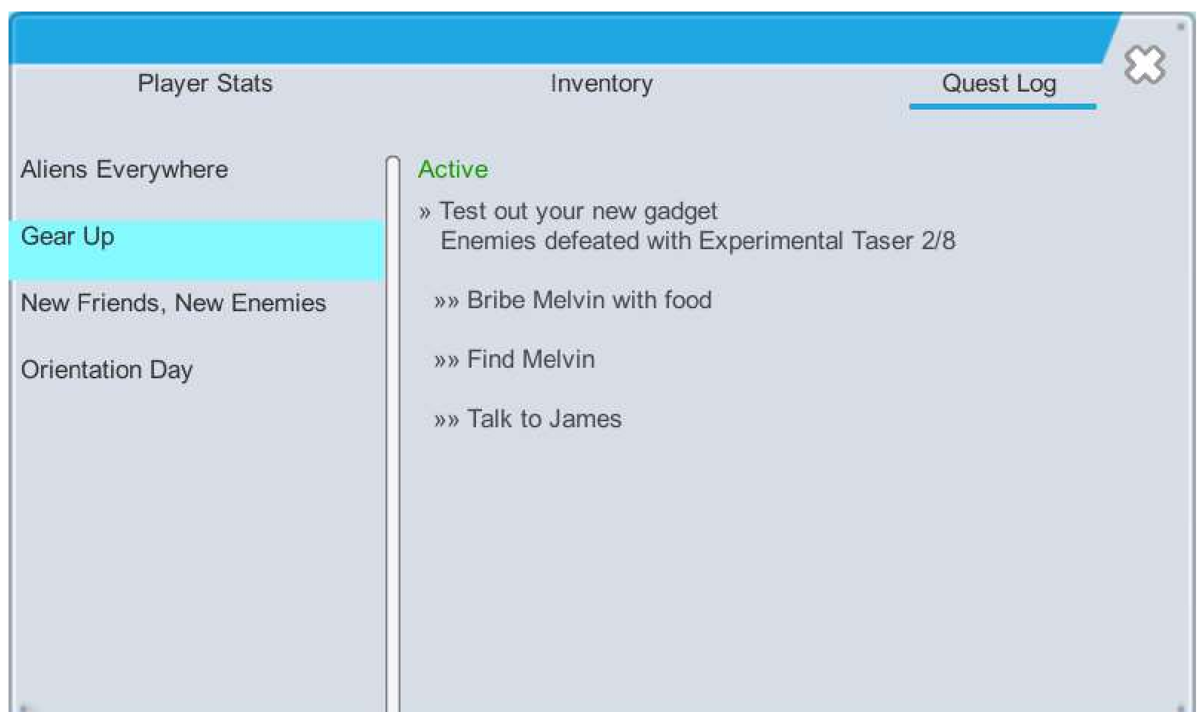

Image 7 Quest log interface

The last tab of the window pictured in Image 7 contains the quest log. As with the inventory the contents are added at runtime, and in this case the used data is handled by the quest system detailed in the thesis written by Milla Koivisto. Another simple window not pictured here is used for the quest dialogs and conversations between the player and non-player characters, also dynamically creating needed text and button objects based on information received from the corresponding systems.

3.2 Items and Inventory

Traditionally an integral part of RPGs are the various items the characters gain and use during the game, used in quests for collectibles and rewards, as consumables to add temporary benefits, and as usable gear that increases the attributes of the user or gives them new abilities. When designing the system an important factor was the method of data storage. For large amounts of information a database might seem like an obvious choice, but for a small project the need for a remote server or a third-party embedded solution combined with the rigid structure of a relational database was an unnecessary complication. Storing the items in XML or JSON files was considered as these formats are well supported, can be edited with any text editor, and could be implemented with the included C# libraries. In the end the built-in ScriptableObject class was chosen for its simplicity, flexibility and automatic handling of file operations.

As the designed items had both common properties and ones used only by a subset, they were divided into a hierarchy of item types. The base ScriptableObject class for items defines the variables shared by all types such as icon shown in the UI, name, and description, and many scripts use this base type for item references to allow new types to be freely added and by default compatible with most existing systems. Other types then inherit this class and add the features used by items of that type and its subtypes. The individual items are created as ScriptableObject asset files.

In addition to variables for data storage the ScriptableObject classes can of course contain methods, and this feature is used for actions connected to specific items. The weapon class for example includes a virtual method for performing an attack, called by scripts attached to the player and other characters. The attack method is implemented by subclasses for specific types such as ranged and melee weapons, allowing scripts to handle weapons in a more generalized manner though a common interface. The data contained in a ScriptableObject can include prefab references which can then be utilized in the class methods. The ranged weapons for instance use pre-defined GameObjects for the projectiles launched on attack.

Collections of items owned by game entities, known as inventories, are stored in lists abstracted behind a class that internally handles additions and removals. Each item in an inventory is paired with a count, and the class method for adding an item either inserts a new item to the list or increments the count of one already exists in the list, or the reverse for removals. The inventory class is not a component, but instead a regular class that can be used in scripts that manage lists of

items. To allow the inventories to be modified in the editor an extension was added, and for the use as variables stored in components it was created as a property drawer. In comparison to a custom inspector that can completely re-define how a component information is displayed, a property drawer only modifies the area assigned for a single variable. The default inspector will use the new property drawer for inventory fields, and custom inspectors can make use of it as well. Below in Image 8 the property drawer for an inventory is shown along with two other component variables in the default inspector.

Image 8 Custom property drawer in an inspector window

3.3 Save System

The ability to stop playing and continue later without losing progress is essential in most games. To create this functionality all the data necessary for restoring the game state has to be saved into a location where it will persist after the game has been closed. It may seem obvious that what is needed is to take a snapshot of all game data and save it for full restoration, but in practice it is not quite as straightforward. Storing objects indiscriminately can lead to large save files that take a long time to save and load, and a more feasible strategy is to deliberately choose the data which needs to be saved. The overall state of the game consists of smaller substates contained in components, and an individual state may be calculable from a limited number of values thus eliminating the need to save the whole object.

A central part of the saving feature is serialization, a system that converts game data into the chosen storage format and extracts it back from it. Serializers for various formats are available both in the scripting API and as third-party libraries. Using a human-readable text format such as XML or JSON makes for larger save files because of the needed syntax elements and allows the files to be easily modified outside of the game. For smaller files and cheating prevention binary format can be used; binary files can be edited as well but the increased level of difficulty discourages most users sufficiently. (Kirst 2014.)

Unity objects and types such as GameObjects, components and vectors bring some complications into the equation as they are not serializable. To save them the data needs to be transferred into another type such as a custom class that is marked serializable, and when loaded the data needs to

be inserted back. Additionally, they do not have stable identifiers that stay the same between sessions. When the game state is loaded, data needs to be restored to specific objects, references to Unity objects rebuilt, and previously initialized prefabs re-created, and for these operations unique identifiers are often necessary. (Meijer 2014.)

A simple solution for saving the game would be to create a single class for storing the needed data and then serialize an instance of that class to a file. Before saving the data would need to be fetched from other components and inserted to the storage object, and after deserializing the file the values would need to be restored back to the original component variables to load the game. The downside of this approach is the fact that every piece of information needs to be handled individually in the saving and loading scripts, which can negatively impact maintainability.

The system created for this project uses a more complicated strategy of collecting the data dynamically. A custom component is attached to each GameObject that contains information needing to be saved. The component defines a unique identifier for the GameObject and selects which of its other components will be stored, and with a customized inspector these selections can be modified in the editor as shown in Image 9. Another script maintains a list of GameObject references and their associated identifiers in the active scene for saving and loading.

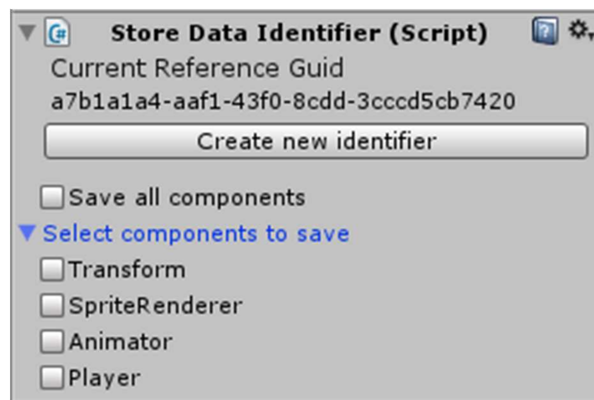

Image 9 Store Data Identifier component viewed in the inspector

Serialization is handled with protobuf-net, a C# .NET implementation of Google’s protocol buffers serialization format and library. The protocol buffers binary format is small and fast to serialize, and the library has an additional advantage of making it simpler to work around the limitations of Unity types. Support for vectors and other struct types for example can be added in a single line of code per type, while more complex types can be serialized using surrogate classes. A surrogate defines an alternative representation of a type and the methods used for transforming data between objects of the original and surrogate types. When the protobuf-net serializer is set to use a surrogate for a type it handles the transitions between objects automatically.

To create a save the saving script iterates through the aforementioned list of GameObjects and identifiers, stores the identifier and a list of the components marked for saving from each of them in a serializable object, and finally serializes the list of objects to a file with the protobuf-net serializer.

Similarly the loading operation goes over each object deserialized from a file, finds the GameObject correlating to the stored identifier, and loads each component. Specifically, values are restored to component variables one by one as the pre-existing instance of the component in the GameObject cannot simply be replaced with the newly loaded copy because of restrictions in the scripting API.

3.4 Player Controls

Image 10 Player interface

In Image 10 can be seen two arrow buttons used for player movement. The area responding to touch input is larger than the images, but it is not emphasized more so that the view of the game is not obstructed. The controls respond to long presses as well as single taps, and allow the player to drag their finger from one button to the other to change movement direction without stopping the touch in between. A custom UI event component is used to detect any touches in the control area and send the event details to the assigned callbacks. Another component on the UI GameObject receives the input information and assigns a movement direction to a variable depending on the location of the touch. To allow both touch and keyboard control a manager script is used to handle both forms of input, and to determine the actual movement it accesses the aforementioned movement direction variable in addition to querying a built-in input system. Finally a component on the player GameObject moves the character whenever it reads a non-zero value from the input manager, while simultaneously adjusting the sprite animations to match the movement.

A second player interaction visible in Image 10 is the talk button above the blue character. These action buttons connect other game systems to the UI and are displayed situationally. For instance the button decorated with an exclamation mark icon is presented when the player is in the vicinity of a character offering a new quest. The area where the action can be performed is specified with a trigger component. The boundaries of solid objects in the physics simulation are defined with collider components, and scripts can use MonoBehaviour event functions to detect when such

objects collide with each other or move inside a trigger. The button in this example is set visible when the player enters the trigger area attached to the other character, and hidden when they exit.

3.5 Dependency Management

Several systems and classes in the game have dependencies between them, and if not properly managed this can lead to decrease in code readability, quality, reusability, and maintainability. This is of course true in all software development, but lack of experience in complex projects with such a large scope caused this matter to be somewhat overlooked until it became more prominent. A rudimentary method for dependency management grew organically as over time the most significant dependencies were moved into dedicated manager components. Disadvantage of this method was the necessity to add the managers into each scene individually, which opened up more opportunities for human errors. To minimize issues the managers were changed into singletons, but instead of using static classes they were still kept as components to retain access to the inherited MonoBehaviour functionality. The only difference from the usual is that on initialization a singleton component adds a GameObject containing an instance of itself into the scene if one does not exist. The singleton pattern guarantees access in every scene, while lazy initialization ensures that manager objects are not created unless needed but still allows them to be added to a scene in the editor for level specific configuration.

3.6 2D Sprite Rendering

The 2D features of Unity have been greatly improved during its lifetime, but as a mostly 3D focused engine some issues can be found when working with 2D graphics. A problem encountered in this project was transparent pixels appearing on the edges of sprites, illustrated below (Image 11).

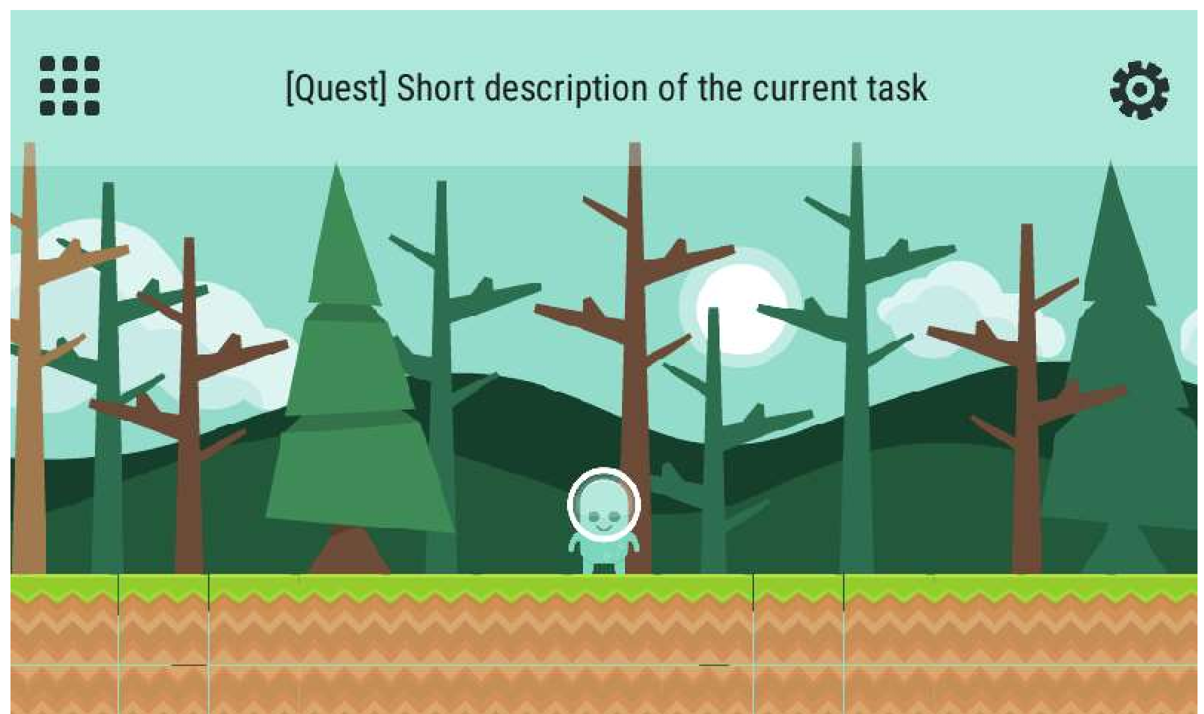

Image 11 Transparent pixels appear between tiled ground images

2D graphics often use smaller stylized images, and with modern screen resolutions the sprites need to be scaled to sizes larger than the original images. To keep the desired look the sprites need to be precisely scaled and positioned so that one pixel in the source image is rendered as a whole number of pixels on the screen, otherwise they might look blurry or have other inconsistencies such as misaligned or transparent pixels. Unity editor allows for very precise positioning of objects in a scene, but when working with sprites this precision can be a hindrance. Two sprites in the editor might appear to be right next to each other, when in reality there is a small gap between them that is only visible when the scene is rendered. As the coordinate system uses floating point numbers there is also the possibility of rounding errors when determining the exact position of an object for rendering. Using sprites that are one unit wide in Unity allows easy and precise positioning as the location coordinates can then be defined in whole numbers. This can be accomplished by importing the sprites to Unity with pixels per unit set correctly in relation to the source image resolution. Another potential cause for issues is the texture filtering mode of images; the bilinear and trilinear filtering modes used to create smooth textures for 3D models may cause sprites to be blurred, and point filtering should be used instead to maintain the intentionally pixelated look.

In addition to the sizing and import settings of sprite images the orthographic size of the camera also needs to be set so that the correct pixel ratio is achieved. This can be done programmatically by calculating the aspect ratio from the screen resolution and sprite pixel size, which works correctly as long as the vertical resolution of the screen is a power of two. For majority of mobile devices this can be assumed to be true, but it is possible that on other platforms the game might be displayed in a window not consistent with standard screen sizes which would necessitate additional checks and fixes.

4 SUMMARY

The goal of creating a game prototype was met well in regard to the planned features as the needed background systems were all implemented during the project. Where the end result is somewhat lacking is playable content; the levels used during development to test the game systems did not get polished quite enough to form a cohesive play experience.

Although it was a rather programming centric project, in hindsight more focus should have been given to content creation. The decision to create a roleplaying game was not the optimal choice for inexperienced developers; while the features expected of games of the genre are indeed well established, they are numerous, complex, and interconnected. The large scope made the project more challenging than it had to be, and the reliance on content such as quests, items, and character abilities to create gameplay was definitely a drawback for a small team consisting only of programmers.

Likely the most significant lesson learned from the project was the importance of good planning and project management. The project was divided only to a few broad tasks, which made it hard to track progress and to keep focus on the central aspects of each task. Personally I devoted too much time on code quality, and usability features, and while they are important for maintainability and further development, these details did not need quite as much emphasis in the prototype stage. This could have been kept better in check with smaller milestones and more frequent deadlines, which would have additionally made the schedule easier to follow.

Overall, this project was interesting and highly educational, and the continuous growth in my skills and experience was clearly noticeable especially towards the end. Several game systems went through large rewrites, and over time previously complex scripts became much simpler as more elegant solutions were found. Moreover, the variety of programming problems gave plentiful opportunities to learn in practice about a wide array of subjects besides familiarizing myself with the selected tools.

For future development, the feature I see as most in need of improvement is the interactivity between game characters, more precisely the combat actions. Currently they are tied to the items, a solution that initially seemed simple but in practice turned out to be somewhat inconvenient. Creating a separate system for the character abilities would make it easier to add new ones, and allow usage as both character specific actions and as item effects. Other sections of the prototype are at a better stage when it comes to extendibility, and the intention is to continue development towards a release ready game.

REFERENCES

CHACON, Scott & STRAUB, Ben 2014. Pro Git. 2nd ed. Apress. [Referenced 2015-09-14]. Available at: https://git-scm.com/book/en/v2

GITHUB, Inc 2015. Field-tested tools for any project. [Referenced 2015-10-5]. Available at: https://github.com/features

KENNEY 2015. Kenney Game Assets. [Referenced 2015-10-4]. Available at: http://kenney.nl/projects/kga

KIRST, Martin 2014. Persistence in Unity 1 - Saving Game Data via Serialization. [Referenced 2015-11-6]. Available at: http://www.frozenfragments.com/en/persistence-in-unity-1/

MEIJER, Lucas 2014. Serialization in Unity. [Referenced 2015-11-6]. Available at: http://blogs.unity3d.com/2014/06/24/serialization-in-unity/

MICROSOFT 2015. Game Development with Visual Studio. [Referenced 2015-10-5]. Available at: http://www.visualstudio.com/features/game-development-vs

O'SULLIVAN, Bryan 2009. Making Sense of Revision-control Systems. [Referenced 2015-09-14]. Available at: http://queue.acm.org/detail.cfm?id=1595636

PAOLUCCI, Nicola 2014. How to Handle Big Repositories with Git. [Referenced 2015-11-6]. Available at: http://blogs.atlassian.com/2014/05/handle-big-repositories-git/

UNITY TECHNOLOGIES 2015a. A Feature-Rich and Highly Flexible Editor. [Referenced 2015-10-5]. Available at: http://unity3d.com/unity/editor

UNITY TECHNOLOGIES 2015d. Asset Import and Creation. [Referenced 2015-11-2]. Available at: http://docs.unity3d.com/Manual/AssetImportandCreation.html

UNITY TECHNOLOGIES 2015g. Creating and Using Scripts. [Referenced 2015-11-2]. Available at: http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

UNITY TECHNOLOGIES 2015h. Execution Order of Event Functions. [Referenced 2015-10-5]. Available at: http://docs.unity3d.com/Manual/ExecutionOrder.html

UNITY TECHNOLOGIES 2015e. GameObjects. [Referenced 2015-11-2]. Available at: http://docs.unity3d.com/Manual/GameObjects.html

UNITY TECHNOLOGIES 2015c. Learning the Interface. [Referenced 2015-11-6]. Available at: http://docs.unity3d.com/Manual/LearningtheInterface.html

UNITY TECHNOLOGIES 2015f. Prefabs. [Referenced 2015-11-2]. Available at: http://docs.unity3d.com/Manual/Prefabs.html

UNITY TECHNOLOGIES 2015i. ScriptableObject. [Referenced 2015-11-2]. Available at: http://docs.unity3d.com/Manual/class-ScriptableObject.html

UNITY TECHNOLOGIES 2015b. The Leading Global Game Industry Software. [Referenced 2015-10-5]. Available at: http://unity3d.com/public-relations

UNITY TECHNOLOGIES 2015j. Using External Version Control Systems with Unity. [Referenced 2015-10-5]. Available at:
http://docs.unity3d.com/Manual/ExternalVersionControlSystemSupport.html