KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme In Business Information Technology

Taavi Saarelainen
Miika Pakarinen

2D GAME DEVELOPMENT WITH UNITY 3D
CASE STUDY: ICEMARE

Thesis
December 2013

Karelia UNIVERSITY OF APPLIED SCIENCES	**THESIS** **December 2013** Degree Programme In Business Information Technology Karjalankatu 3 80220 JOENSUU FINLAND p. 013 260 600

Author(s)
Taavi Saarelainen, Miika Pakarinen

Title
2D Game Development With Unity 3D

Commissioned by
-

Abstract

The aim of the thesis is to study the functionality of third party 2D frameworks with the Unity game engine. The thesis discusses the advantages and disadvantages these frameworks have. A series of performance tests were executed on these frameworks to investigate their ability to render 2D graphics.

A game prototype was created in support of this thesis where one of the examined 2D frameworks was utilized. Common 2D mechanics were applied in the prototype development, which are inspected both in practice and theory. The prototype utilizes general 2D game mechanics and fluid physics simulation provided by Fluvio-plugin.

All the chosen frameworks included a similar set of core tools for applying and managing 2D graphics but differed in the number of special tools. Noticeable differences in quality were found between the frameworks. The performance test results were consistent on average but substantial differences appeared when rendering large amounts of graphic objects.

2D frameworks come with various features and qualities. A 2D framework should be chosen based on the requirements of the project and any special tools required.

Language English	Pages 67 Appendices 5 Pages of Appendices 12

Keywords

game development, fluid dynamics, Unity, 2D game, 2D framework, 2D Toolkit, Ex2D, Uni2D, Orthello, Futile, Fluvio

	OPINNÄYTETYÖ **Joulukuu 2013** Tietojenkäsittelyn koulutusohjelma Karjalankatu 3 80220 JOENSUU p. 013 260 600
Tekijä(t) Taavi Saarelainen, Miika Pakarinen	
Nimeke 2D Game Development With Unity 3D Toimeksiantaja -	

Tiivistelmä

Opinnäytetyön tarkoitus on tutkia kolmannen osapuolen 2D-sovelluskehysten toiminnallisuutta Unity-pelimoottorilla. Opinnäytetyössä selvitetään näiden sovelluskehysten hyviä puolia sekä ongelmakohtia. Sovelluskehyksille tehtiin lisäksi sarja suorituskykytestejä, joissa tarkasteltiin niiden kykyä piirtää 2D-grafiikkaa.

Opinnäytetyön tueksi toteutettiin peliprototyyppi, johon valittiin yksi tarkastelluista sovelluskehyksistä. Prototyypin kehityksessä käytettiin yleisiä 2D-mekaniikkoja, joita tarkasteltiin sekä käytännön että teorian tasolla. Peliprototyyppiin luotiin myös yleisesti käytettyjä 2D-pelimekaniikkoja ja siinä hyödynnettiin nestefysiikkaa mallintavaa Fluvio-liitännäistä.

Kaikki valitut sovelluskehykset sisälsivät samankaltaiset perustyökalut 2D-grafiikan tuottamiseen ja hallintaan, mutta erosivat erityistyökalujen määrässä. Laadullisesti sovelluskehyksissä nähtiin huomattavia eroja. Ajetuissa suorituskykytesteissä tulokset olivat pääpiirteittäin samankaltaisia, mutta isojakin eroja ilmeni piirrettäessä suuria määriä grafiikkaobjekteja.

Sovelluskehyksillä on erilaisia ominaisuuksia. Sovelluskehys tuleekin valita projektin vaatimusten ja tarvittavien erikoistyökalujen mukaan.

Kieli englanti	Sivuja 67 Liitteet 5 Liitesivumäärä 12

Asiasanat

pelikehitys, nestefysiikka, Unity, 2D-peli, 2D-sovelluskehys, 2D Toolkit, Ex2D, Uni2D, Orthello, Futile, Fluvio

CONTENTS

PREFACE

This thesis was created out of interest towards creating 2D games with the Unity game engine. We wish to provide other amateur developers with insight on the subject and help their efforts while developing 2D content with Unity. The content of this thesis is primarily meant for people who are already familiar with Unity and game development in general.

We thank the following people for participating in the making of the 2D game prototype that later on was used as a case study in support of this thesis.

Antti Kovanto, Antti Närvänen, Jani Eronen, Juho-Pekka Pirskanen, Simo Riikonen, Joonas Rauha and Teemu Kokkonen.

Special thanks go to our teacher and mentor Anssi Gröhn, who gave us guidance throughout the project and eventually oversaw creation of this thesis.

We would also equally like to thank our families and everyone involved in the making of this thesis and all those who gave their support one way or another.

1 INTRODUCTION

This thesis covers basic concepts behind creating a 2D game with Unity[1] game engine. The authors of the thesis are Miika Pakarinen and Taavi Saarelainen, both specialising in game programming and being especially interested in enhancing their grasp on 2D development. The need for making this thesis arose solely from personal and professional pursuit to understand the different development possibilities available for creating 2D games with Unity.

The case study is a prototype of a 2D platformer[2] called Icemare. Icemare is a game project made in collaboration with several fellow students from our degree programme. Icemare is a simplistic 2D platformer in sprite graphic environment. It has features common to the genre such as jumping over obstacles and climbing ledges and ropes. We also utilize fluid dynamics in the prototype in form of a water gun which is meant for defeating foes and solving puzzles.

Unity included 2D development tools in the update 4.3 (Goldstone, 2013). Unfortunately the update did not make it in time to be included in this thesis; therefore only third party 2D frameworks[3] for Unity are covered. It remains to be seen how the current frameworks will compete against Unity’s native 2D tools, but it is quite certain that they will continue to exist nevertheless and concentrate on those matters that the native support does not do or does inadequately.

The lack of 2D support has generated many challenges for developers to tackle before a productive development environment can be established. The very first issue has been choosing between the multiple different 2D frameworks available for Unity. Choosing the 2D framework most suitable for a project is no trivial task and requires extensive research. One of the goals of this thesis is to ease this process.

[1] Appendix 1: Unity.
[2] Appendix 1: Platformer.
[3] Appendix 1: 2D Framework.

The content of this thesis can be divided roughly in three sections. First is the theoretical part about Unity and 2D development in general for those not too familiar with the subject. Why Unity was the engine chosen for our project is also discussed.

The thesis then moves on to various 2D frameworks available for Unity, where they and their features are examined. Five different frameworks were chosen for this study. The frameworks were chosen by two criteria; they had to be established with a notable user base and they had to include a collection of tools for handling sprites. A few already existing framework comparisons (Prime31 2013, Müller 2012, Jarcas Studios 2013, Pau 2013) were used to help decide which frameworks would be worth exploring. The frameworks also undergo performance tests to evaluate their capabilities rendering 2D sprite graphics.

The last part is about the case study which focuses on the development process of the Icemare game prototype. This part covers some background about the theory behind 2D development techniques utilized in the prototype and also goes through the tools and plugins[4] we used during the project.

The questions this study aims to answer are the following. How do the chosen 2D frameworks compare feature wise and what are their obvious pros and cons? What kind of performance do they offer compared to each other when rendering sprite graphics? Which framework would best suit a traditional side-scrolling[5] 2D platformer similar to the case study prototype?

[4] Appendix 1: Plugin.
[5] Appendix 1: Side-scrolling.

2 UNITY 3D ENGINE

Unity is a video game development environment used for developing games on multiple platforms. It is used to develop games for PC, Mac, Linux, current generation consoles and most popular mobile operating systems. It is also the most widely adopted engine used today for mobile development (Gamasutra 2012). This is not surprising since Unity is often praised for being the best engine for rapid development (Andrews 2013).

One of Unity's beauties lies in the possibility in cross-platform[6] deployment without the need to rewrite your code base. A handful of system specific tweaks are enough. Unity comes in two versions for PC; free and pro version. The choice for this study was the free version of Unity since purchasing the pro version for a non-commercial project would have been rather costly for a student project and no problems were seen in realising the study without Unity's pro features. The free version of Unity does not come with some of the excellent features of Unity Pro (appendix 2) but still manages to offer a very good choice for many amateur projects and even some cases professional.

However, Unity Pro would be the definite choice for most serious projects with the arsenal of important features it provides. Some of these features include optimization and profiling tools, state of the art graphic options and support for external version control to mention a few.

Whether you are using the free or pro version; it is good to remember that you can build your own tools if needed or resort to using third party plugins and programs. You can find a variety of editor extensions and other tools from the Unity asset store[7] accessible through Unity editor. Most items found in the asset store are available for a moderate fee but there are plenty of free downloads as well. Why Unity was chosen for this study is discussed in the following chapter.

[6] Appendix 1: Cross-platform.
[7] Appendix 1: Unity asset store.

2.1 Why Unity?

A 3D game engine is not necessarily the first thing to come in mind when thinking about developing a 2D game. The case study was initially meant to be developed into a flash[8] game, using Apache Flex[9] and FlashDevelop[10]. This at the time felt like a good choice since somehow flash had become somewhat synonymous with smaller amateur 2D platformers and was what everyone involved in the project thought should be the tool of choice at the time.

It took a while for the project team to even realize that Unity could be used for the prototype even though Unity was the only engine that all the team members had previous experience from. This was probably because all the work anyone had done with it was mostly in 3D. It was somewhere around the first week of the project where the development environment had already been established with flash when someone from the team said; “Could we not do all this easier in Unity?”.

This raised the question if Unity would be a good choice. Unity is a 3D engine and therefore a 2D game made with it would essentially be 2.5D which by definition can mean either 2D that fakes 3D or 3D that fakes 2D. For clarification whenever 2.5D is mentioned in this thesis, it refers to the latter. What this means is that a 3D world can be created and simply be restricted into to a 2D plane. This is something that comes with advantages as well as disadvantages.

Starting from the negatives one of such would be, that the game probably cannot be optimized as well as a real 2D game could be. Still the advantages outweigh the disadvantages by far especially since Unity comes with a visual editor that greatly enhances the workflow. For example, having the possibility to mix 3D effects in otherwise 2D world, can create some really good looking scenarios.

[8] Appendix 1: Flash.
[9] Appendix 1: Apache Flex.
[10] Appendix 1: FlashDevelop.

A rough comparison (table 1) was composed to help make the final decision between Flash and Unity. The project team appreciated how easy Unity was to learn and how effective it was for rapid development with one button platform deployment. Most of the faults seen in Unity such as the lack of proper GUI-tools[11] were viewed as solvable by using third party plugins if needed. The biggest issue with Flash was essentially the code-only development approach compared to having a visual editor and thus having to do more manual labour.

Unity Free		**Flex / FlashDevelop**	
Pros	**Cons**	**Pros**	**Cons**
Free	Difficult source code management	Free and open source	More difficult to learn
Instant deployment on most platforms	No tools for sprite graphics	Easy to deploy on web	Not universally supported (iOS)
Comes with a visual editor	Bad tools for building in-game GUI's	Highly Customizable	No visual editor
Easy to use and learn	Expensive if you want pro features		Works only through code
Built-in physics engine	No native 2D support		

Table 1. Comparison of Unity 3D and Flex / FlashDevelop.

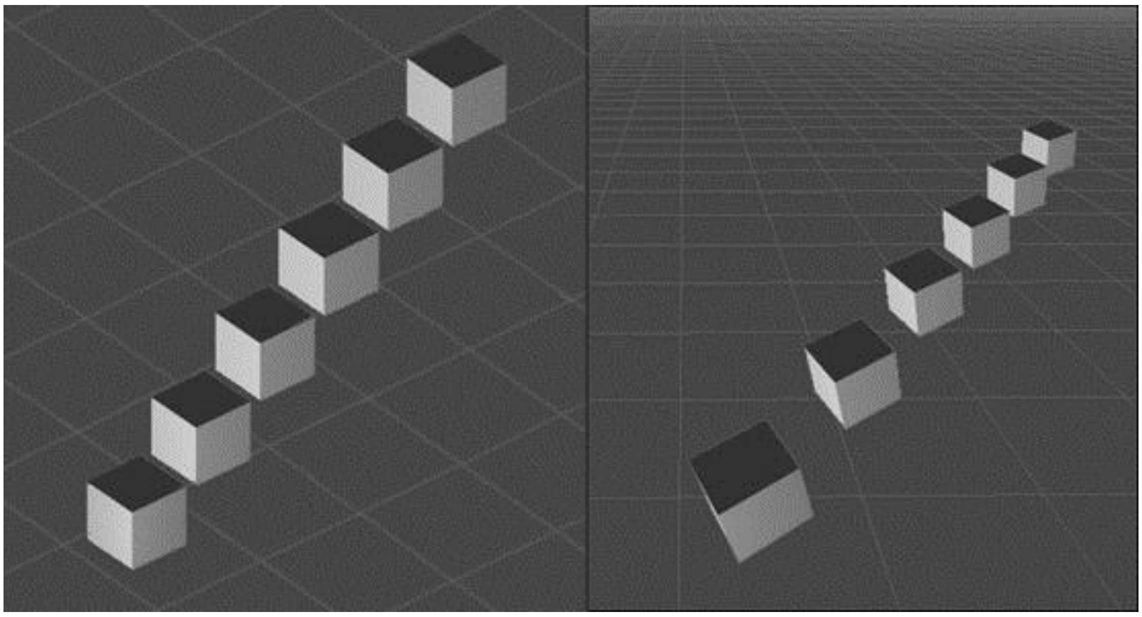
2.2 2D Development

The trick to creating 2D games in a 3D environment is to create the illusion that everything is in two dimensions. In reality everything is created in 3D space but with a fixed camera angle. This is achieved by an orthographic[12] perspective shown in picture 1. In other words; something commonly referred to as 2.5D is being created, which means limiting all or parts of itself into an orthographic

[11] Appendix 1: GUI.
[12] Appendix 1: Orthographic.

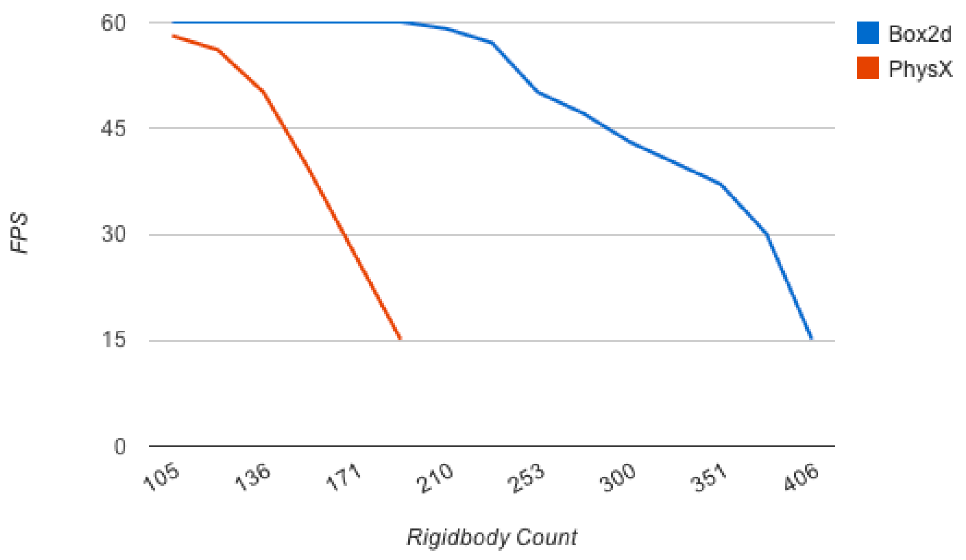
perspective. In modern games the latter is used more often due the updated look and feeling it gives over the traditional 2D games.

In orthographic projection it does not matter from which angle the viewer sees the picture as it always looks the same. This happens because the projecting lines are always at the right angle to the plane of the projection, no matter if it is viewed from right of the viewspace or left of the viewspace. It will also disconnect the distance relation so the game object will always appear to be as close even if it the depth of the object would change.


Picture 1. Orthographic and perspective view of the same objects.

Both strict 2D and 2.5D have advantages and disadvantages although the transition from one to the other can be considered trivial if using a 3D engine. Gameplay wise there is not necessarily much difference between them, so the greatest differences usually lie in the artwork. Using 2D pixel art style sprites for everything can be costly as creating animations and lighting for each scenario would require extra work. Using 3D objects on the other hand requires modelling but allows the re-use of animations and the use of real time lightning and physics. The use of a 3D engine for a 2D game allows things that would normally be beyond for a typical 2D game such as dynamic physics or 3D animation among other things. (François 2011.)

However, it is worth remembering that using a fully fletched 3D physics engine comes with certain drawbacks no matter how easy and convenient Unity's built-in NVIDIA physX[13] engine sounds like. This shows especially in performance where an optimized Cocos2D engine[14] running with Box2D[15] physics out performs Unity and physX by far in amount of rigidbodies measured in frames per second as presented in picture 2 below.



Picture 2. PhysX versus Box2D. (Picture: O'Brien 2011).

Even though the possibility of using real 3D objects and animations in the prototype exists; using 2D sprites and repetition of sprites for creating animations was consider a better option. This is the main reason for needing a 2D framework for handling 2D sprites since Unity was not very well designed to do that natively, before update 4.3, but luckily Unity community introduces multiple frameworks to aid the development of 2D content.

[13] Appendix 1: NVIDIA PhysX.
[14] Appendix 1: Cocos2D.
[15] Appendix 1: Box2D.

3 2D FRAMEWORKS

Third party 2D frameworks available for Unity ease the development of 2D games and provide developers with the right kind of tools for tasks like using sprite images, using texture atlases (explained in chapter 4.1), animating sprites and much more. Without a proper framework doing otherwise trivial tasks can be quite awkward and difficult with just Unity’s own toolset. 2D frameworks can make 2D development quite smooth by providing you with professional quality tools.

Another way the 2D frameworks improve workflow is by reducing the amount of code you need to write or even remove the need completely depending what it is you want to achieve. Many 2D frameworks allow you to hook directly in to their source code if you need modify their functionality.

3.1 Free frameworks

The frameworks picked for comparison were chosen by their popularity and by how much they appealed to the needs of the prototype project. Frameworks that only do very basic sprite functionality were not included. Futile was chosen as it stood out as the sole code-only framework with no editor side features. Orthello, on the other hand, was chosen as it offered wide array of features compared to most free frameworks.

Futile is an open source 2D framework designed to work completely from code and has no real use for the Unity editor. Futile is described to be very similar compared to using Cocos2D or Flash. The creator himself, Matt Rix (2012), stated the following: “Think of it as a gateway drug. Maybe after they try this for a while they'll want to dive in and do more stuff in the editor“. According to him it might be attractive to introduce programming oriented people to the Unity engine by code and let them decide if they want to dig in to the graphical editor or not.

Obviously the fact alone that there is practically nothing on the editor side makes Futile fundamentally different from the GUI-based 2D frameworks which speed your workflow by giving you quick and simple tools for dealing with sprites and other 2D content. In Futile's case you could argue that the workflow is slow and tedious in comparison but maybe there are things to benefit from as well such as having more flexibility within the framework since the source code is mostly overridable.

The performance of Futile is very good when rendering large amounts of sprite objects on the screen. Futile performed best among the tested 2D frameworks in the performance testing concluded in chapter 3.5.

Futile is currently under development but is promised to work nevertheless. Also there is no documentation available which can and will make Futile a little hard to get into, leaving only some tutorial videos as the only guide. Fortunately the programming logic is easy to understand and well structured, making everything easier after learning the basics.

The first thing to do with Futile is to simply create a blank game object[16] and drag the Futile-script in it. That is all that is required on the editor side of Unity for initializing Futile. After this, everything is done in custom made scripts where Futile can be implemented. The first script for initialization shown below has to be added under the same object as the Futile-script.

The code in listing 1 is initialized quite similarly to many other environments like Flash. Futile-parameters object is created first which handles orientation, resolution, point of origin and other similar things. Then Futile is simply initialized with that object (Lines 6-9).

Images and text in Futile are handled with texture atlases. Atlasing programs such as TexturePacker are needed to make atlases from your sprite images and BMFfont for generating atlases of symbols such as letters and numerals

[16] Appendix 1: Game object.

(Refer to 4.2 Tools). Atlases are then loaded into atlasManager-class which contains everything that can be rendered. Sprites are made with FSprite-objects and labels with Flabel-objects. Finally these are added to the stage to be drawn (lines 11-17).

```
 1   public class DrawSprites : MonoBehaviour
 2   {
 3         public FSprite sprite;
 4         void Start()
 5         {
 6               FutileParams fparams = new FutileParams(true,true,false,false);
 7               fparams.AddResolutionLevel(680.0f,1.0f,1.0f,"");
 8               fparams.origin = new Vector2(0.5f,0.5f);
 9               Futile.instance.Init(fparams);
10
11               Futile.atlasManager.LoadAtlas("Sprite_Atlas");
12               sprite = new FSprite("Image_In_Atlas");
13               Futile.stage.AddChild(sprite);
14
15               Futile.atlasManager.LoadFont("Arial","Arial.png","Assets/Arial");
16               Flabel label = new Flabel("Arial","Hello World");
17               Futile.stage.AddChild(label);
18         }
19   }
```

Listing 1. Basic Futile setup.

Pros of Futile:

- Very flexible, everything can be overwritten
- Excellent performance
- Programming logic is well structured and easy to follow

Cons of Futile:

- No documentation
- Hard to get into
- Forces texture atlases and does not provide tools for making them
- Still in alpha

Orthello comes in a free version as well as a commercial pro version. Pro features are not discussed therefore anything said here does not reflect the pro version or its features. Orthello has built-in animation tools, sprite containers, custom physics properties, sound support and many more other features that supplement 2D game development with Unity.

Orthello provides an easy way to add sprites and animations to the Unity scene and lets them interact with each other with custom made collision or user-actions. Adding these elements is easy by just drag and dropping them from the Orthello root folder in to the scene and configuring their properties from the editor.

Orthello also supports creating objects during runtime through code by using Orthello’s OT-class which allows you to hook into Orthello’s functionality. Orthello is built around using prefabs[17] that can be dragged into the game scene from the Orthello root folder. These prefabs hold ready-made options that can be used to configure their settings.

Free version of Orthello does not come with texture atlasing tools so using an external atlas generator is required. The framework does however offer some configuration options for imported texture atlases.

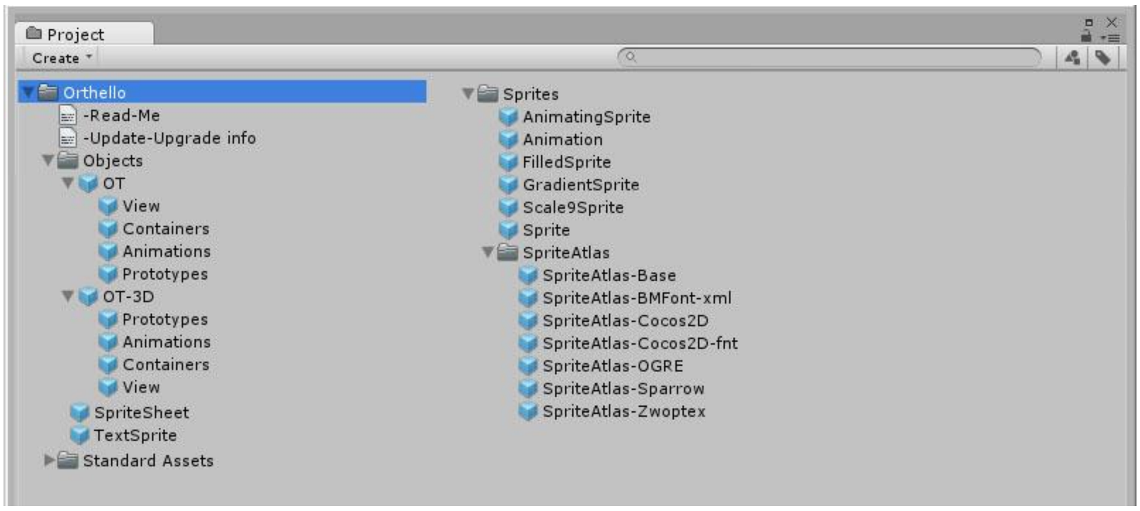
Orthello’s performance was found to be somewhat sluggish when compared to the other 2D frameworks in chapter 3.5. Orthello especially struggles with high sprite counts. This is also reflected in the Unity editor as the editor performance gets worse as you keep adding new sprites.

To start working with Orthello the main system object OT-prefab is first dragged from the Orthello root folder into the editor hierarchy. It needs to be the first Orthello object in the scene. After this other prefabs like sprites or atlases can be added into the scene.

Orthello offers a multitude of different prefabs as shown in picture 3 that can be used for different purposes. These prefabs include several types of sprite objects such as filled sprites which are for repeating texture patterns and scale9

[17] Appendix 1: Prefab.

sprites for rescaling windows or panels to mention a couple. In addition there are different prefab types of atlas files.



Picture 3. Orthello root folder.

Pros of Orthello:

- Drag and drop scene building
- Many prefabs for different object types
- Works well through code

Cons of Orthello:

- No texture atlasing tools
- Poor editor performance
- Poor in-game performance with high sprite counts

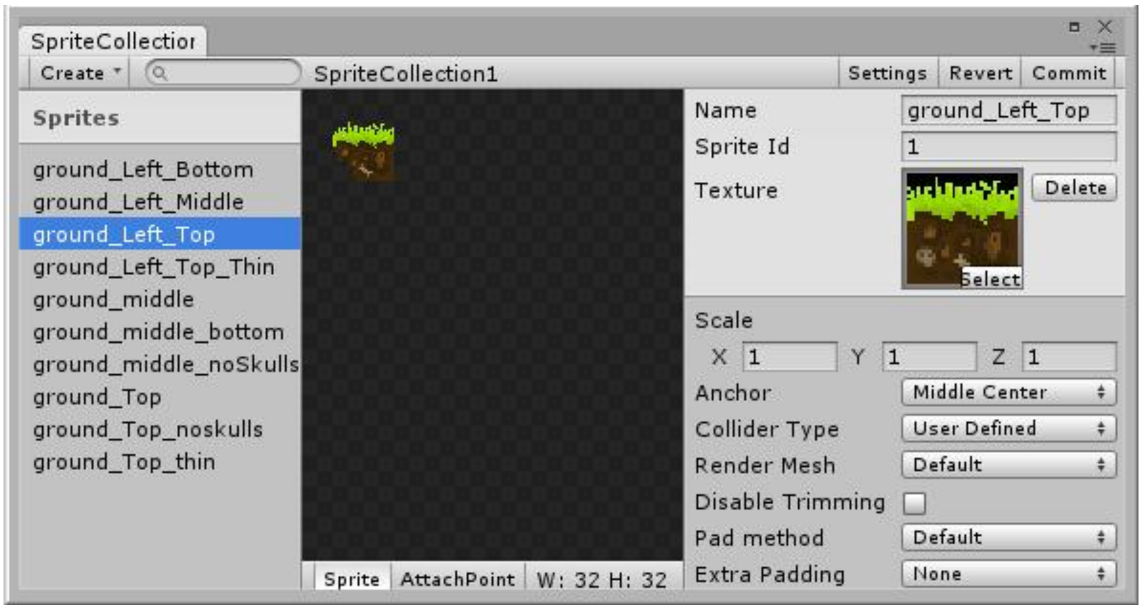
3.2 Commercial frameworks

The commercial frameworks comparison includes the most feature heavy frameworks currently available, excluding Orthello’s pro version. The chosen frameworks are very similar to each other in many aspects but each having some unique features and qualities.

2D Toolkit provides a 2D sprite and text system which are integrated in to the Unity environment. 2D toolkit focuses on mobile performance and workflow efficiency which makes it good for rapid prototyping and fast development cycles. Considering the tools it offers for 65€, it seems to offer everything needed for 2D development, excluding image processing tools.

2D Toolkit has a built-in tile map editor that allows painting them directly into the scene. The framework also comes with set of UI components that complete the lack of good native GUI-tools in Unity. Unfortunately 2D Toolkit does not have any kind of trial option so buying it might be a small leap of faith.

2D Toolkit is used by creating a 2D Toolkit game object into the scene hierarchy. It can be a basic sprite with, without animation or it could have multiple other settings. The framework uses tk2dCamera game object, instead of Unity’s basic camera game object that has 2D specific configurations available such as pixels per meter and orthographic size settings. What makes this toolkit unique is that sprite settings may be decided separately inside texture atlases making it versatile for not forcing to use multiple atlases for different setups. 2D Toolkit allows you to import atlases as well as generate sprite collections by using its own tool into a prefab as shown in the picture 4 below.



SpriteCollection1

Sprites

ground_Left_Bottom
ground_Left_Middle
ground_Left_Top
ground_Left_Top_Thin
ground_middle
ground_middle_bottom
ground_middle_noSkulls
ground_Top
ground_Top_noskulls
ground_Top_thin

W: 32 H: 32

Name	ground_Left_Top
Sprite Id	1
Texture	
Scale	X 1 Y 1 Z 1
Anchor	Middle Center
Collider Type	User Defined
Render Mesh	Default
Disable Trimming	☐
Pad method	Default
Extra Padding	None

Picture 4. 2D Toolkit’s texture atlas generator.

Pros of 2D Toolkit:

- Built-in sprite atlas generator
- Built-in tile map editor
- Built-in UI tools

Cons of 2D Toolkit:

- No trial available

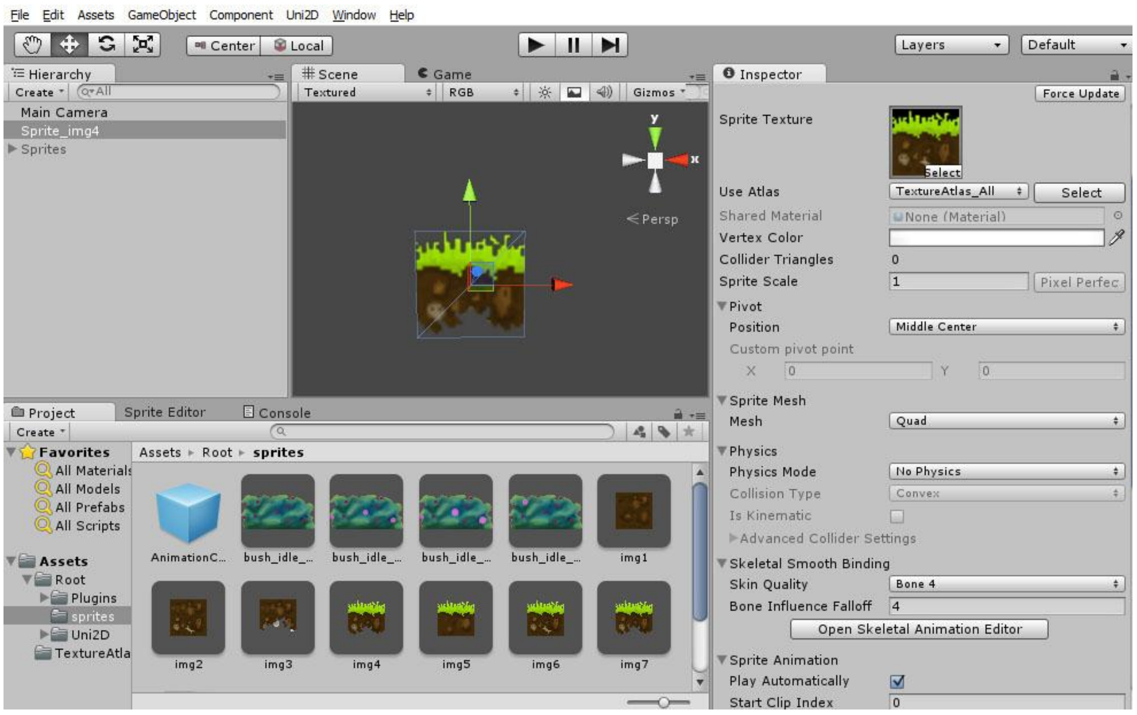
Uni2D is a visual 2D tool extension for Unity like the rest but concentrates more on providing a smooth effortless workflow. This is well represented by Uni2D's slick user interface and one click actions for multi-atlasing and setting up object physics.

One seat licence is priced at 90€ making Uni2D a bit more expensive than the other 2D frameworks. There is also no trial available for Uni2D to test whether it is worth it or not before buying.

Uni2D promotes itself with some really fancy features that no other framework provides such as skeletal animation[18] tool. Uni2D's sprite animation tool is good as well; you simply choose the amount of frames you want and assign them with sprites.

Setting up Uni2D is straightforward. First Uni2D package file is imported into the project. Any image from this point forward that is brought into your assets is automatically associated with Uni2D and works as a sprite. All you need to do is to drag them into the scene. All the needed options like physics are found under the sprite object as shown in picture 5 below.

[18] Appendix 1: Skeletal animation.



Sprite Texture	
Use Atlas	TextureAtlas_All
Shared Material	None (Material)
Vertex Color	
Collider Triangles	0
Sprite Scale	1
Pivot	
Position	Middle Center
Custom pivot point	X 0 Y 0
Sprite Mesh	
Mesh	Quad
Physics	
Physics Mode	No Physics
Collision Type	Convex
Is Kinematic	☐
Advanced Collider Settings	
Skeletal Smooth Binding	
Skin Quality	Bone 4
Bone Influence Falloff	4
Sprite Animation	
Play Automatically	✓
Start Clip Index	0

Picture 5. Uni2D sprite options.

Uni2D also has its own texture atlasing tool which is very easy to use. Atlas object is created through the Uni2D toolbar and sprites can simply be dragged into it. Every sprite object also has a handy setting where you can choose which texture atlas to use.

Pros of Uni2D:

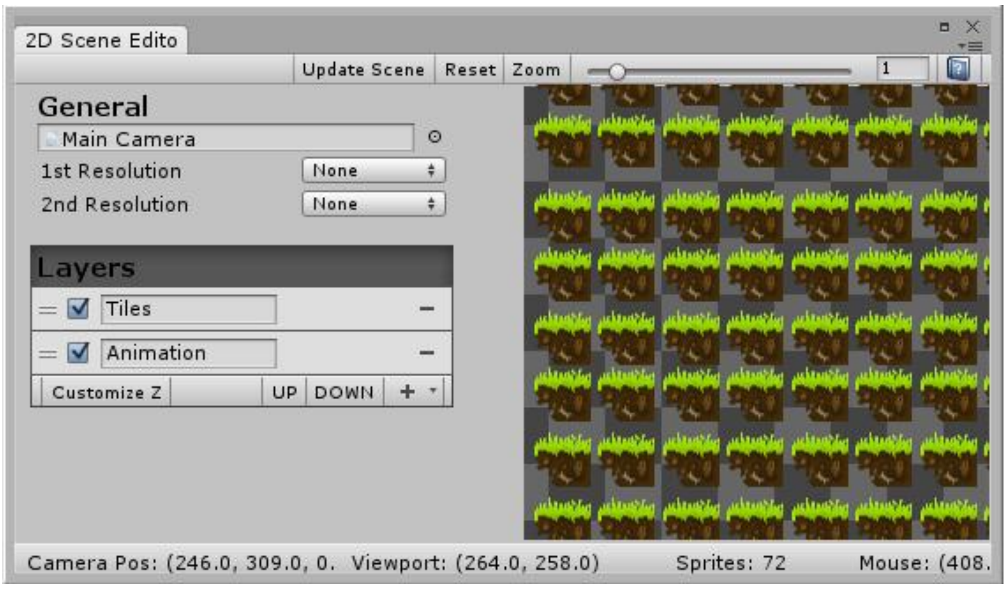
- Good sprite animation tools
- Skeletal animation tools
- Simplistic UI and one click actions
- Texture atlasing tools

Cons of Uni2D:

- The most expensive 2D framework at 90€ per seat license.
- No trial available

Ex2D is a 2D framework that focuses on editor integration. Ex2D comes with editor classes that run inside Unity, which allows users to configure how Ex2D looks like inside the Unity editor. The editor classes are atlas editor, font editor, sprite animation editor, tile map editor and a 2D skinning editor. (Pau 2013.)

Ex2D also comes with a 2D scene editor which allows the developers to work the game world layout in another window which is shown in picture 6 below.



Picture 6. Ex2D external 2D scene editor.

The framework reduces heap allocation[19] and draw calls[20], allowing for maintaining high performance on mobile devices. Ex2D credits its good performance to its custom sprite batching which ignores Unity's own sprite batching mechanisms. Ex2D utilizes sprite method (mentioned in 4.1) layering where it adds all the sprites on different layers which are then rendered in a specific order. This allows Ex2D to render all the sprites on the same layer from the same texture atlas, making the sprite batching process smoother. (exDev Team 2013.)

Ex2D is used by adding Ex-sprite objects from Ex2D menu in the Unity toolbar. The sprite can then be filled with texture info and other configurations from Ex-sprite script. There are no other Ex2D specific game objects. Sprite animations and other new components can be created from the basic sprite as well.

Pros of Ex2D:

- Has texture atlasing tools
- Editor integration / view tabs

[19] Appendix 1: Heap allocation.
[20] Appendix 1: Draw call.

- External scene editor

Cons of Ex2D:

- Lacking documentation

3.3 Framework feature comparison

Table 2 below demonstrates feature differences between Orthello, Ex2D, Futile, 2D Toolkit and Uni2D frameworks. Table 2 does not represent full toolset available for these frameworks but focuses on listing the basic 2D tools.

Tool	**Orthello**	**Ex2D**	**Futile**	**2D Toolkit**	**Uni2D**
Sprite animations	✓	✓	✓	✓	✓
Sprites	✓	✓	✓	✓	✓
Filled sprites	✓	✓	✓*	✓	✓
Gradient sprites	✓		✓*		✓
Scale 9 sprites	✓		✓*		✓
Clipped sprites	✓	✓	✓*	✓	✓
Tilemaps	✓	✓	✓*	✓	✓
Text based sprites	✓	✓	✓*	✓	✓
Polygon Sprites	✓		✓*	✓	✓
Atlasing tools		✓	✓*	✓	✓
Visual editor	✓	✓		✓	✓
Accessible source code			✓		✓

✓* Can be done by coding it yourself

Table 2. Framework feature comparison.

3.4 Performance testing

Performance tests were completed with different amounts of sprites rendered on the scene which were added to the game scene from the Unity editor and in Futile through scripts. All the tests were done with an Android device instead of PC since the amount of sprites that an average computer can process precedes what Unity itself can. The tests were done with sprites ranging from a thousand static sprites to several thousand including animated sprites as well.

The test platform was a Samsung Galaxy S3 smartphone running Android Jelly Bean version 4.1.2. Galaxy S3 has a quad-core 1.4GHz CPU, Mali-400 MP GPU and 1 GB of RAM. Galaxy S3 was chosen due to its large consumer base (Harris 2013) and above average computing capabilities for a smartphone. Choosing more device platforms such as iOS or Blackberry for the tests would have given wider results but was not possible within the scope of this thesis.

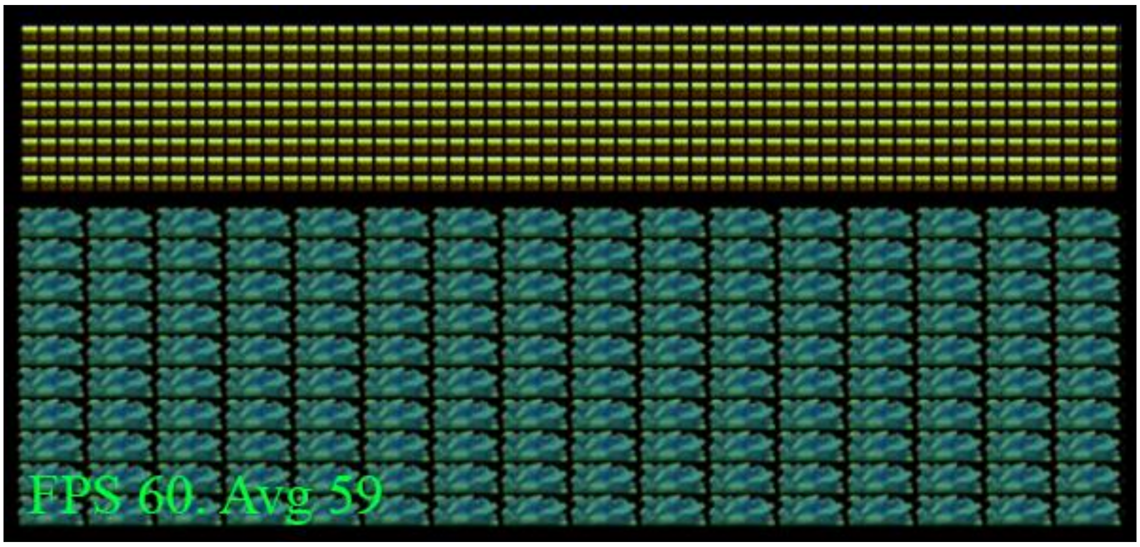
The tests were constructed in empty game scenes following the general guidelines provided by the frameworks in the most similar fashion possible. Each scene would contain a framework system object or objects if needed. Sprites were added into the scene from atlases and multiplied by copying them. The same base project was used in every test. In table 3 below you can see how the tests were composed.

Rule	**Static Sprites**	**Static and animated sprites**
Static sprites 32x32 pixels	✓	✓
Animated sprites 128x64 pixels		✓
Default android build settings	✓	✓
All sprites children of the same game object	✓	✓
Downwards movement for 20 seconds	✓	✓
1000 static sprites	✓	
3000 static sprites	✓	
5000 static sprites	✓	
100 static sprites, 100 animated sprites		✓
300 static sprites, 300 animated sprites		✓
500 static sprites, 350 animated sprites		✓
1000 static sprites, 500 animated sprites		✓
2000 static sprites, 1000 animates sprites		✓

Table 3. Sprite settings for the tests.

Two ways of benchmarking were used. The first test included only static sprites from 1000 to 5000 sprites. The second one had static sprites as well as animated sprites ranging from 1000 to 2000 static sprites and 100 to 1000 animated sprites. The animated sprites were 4 frames each chancing every update. In both tests the sprites were made to move downwards by a script that either used Unity's Translate[21] method or in Futile by updating the objects y-axis value directly through code.

The frame rate was measured with a custom script which counts frames per second over a defined interval. The script is fairly accurate and the error margin is less than 2 frames per second. It should be noted that android limits frame rate to a maximum value of 60 so any results above that are neglected in these tests. The frame rate script was left to run 20 seconds on each setup to get a reliable average. All the Unity and build related settings were left on default settings. An example of a test is shown in the picture 7 below.



Picture 7. Static sprites on top and animated sprites below.

3.5 Test results

The frame rates measured are presented in tables 4 and 5 as well as in charts 1 and 2. There is an estimated error margin of two frames per second. The results are further discussed in the following chapter 3.6.

[21] Appendix 1: Translate.

Static sprite performance

Static sprite count	Orthello FPS	Ex2D FPS	Futile FPS	2D Toolkit FPS	Uni2D
1000	60	59	59	60	60
3000	36	29	59	24	25
5000	0*	19	52	14	14

*Could not test: the framework crashed Unity.
Table 4. Static sprite performance.

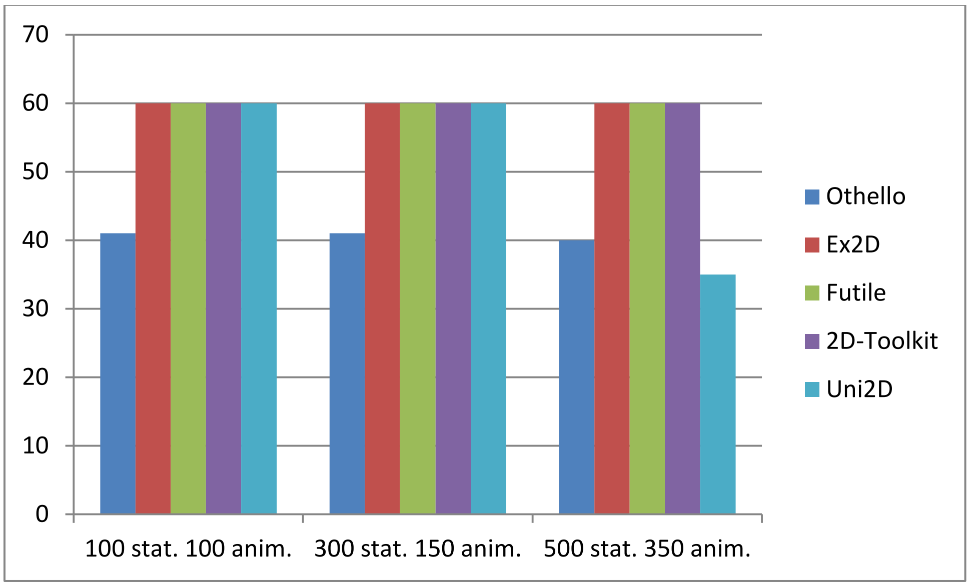


Chart 1. Static sprite performance.

Static and animated sprite performance

Static sprite count	Animated sprite count	Orthello FPS	Ex2D FPS	Futile FPS	2D Toolkit FPS	Uni2D
100	100	41	60	59	60	60
300	150	41	60	59	60	60
500	350	40	60	59	59	35
1000	500	28	55	59	33	23
2000	1000	10	26	59	10	11

Table 5. Static and animated sprite performance.

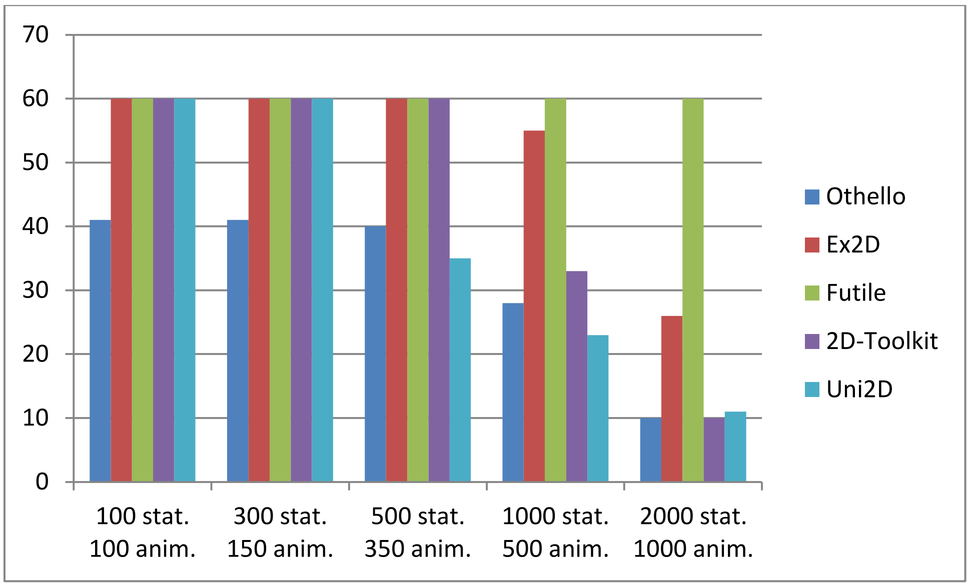


Chart 2. Static and animated sprite performance.

3.6 Test analysis

Futile blew the competition out of the water as far as performance by sprite count goes. Futile stayed within the 59 frames per second in each test, only budging a bit with 5000 static sprites down to 52 frames per second. Testing with Futile was concluded with an extra test of 3000 animated sprites and the frame rate still did not drop noticeably. Any further testing would have been pointless as the sprite count was already much more than what could be expected to be rendered on a screen in mobile 2D game scene. If performance is what is needed the most in a project, and development through programming is not a problem, then Futile is by far the best choice.

Orthello seemed to perform well within average frame rates giving highest ranking score between the visual editors with the tested static sprite count of 3000. Even though the static sprite frame rate performed best in the first two tests; Unity failed to build the project for the last test. Orthello's in-editor performance was also borderline frozen after the 3000 sprites.

With animated sprites added Orthello performed much worse than its competitors. The highest animation frame rates we were able to get were with 100 static and 100 animated sprites which gave us 41 frames per second which was almost 20 frames lower than with the other frameworks. Oddly the performance was comparable with the second last test but again dropped dramatically in the final test.

Ex2D did not seem to perform well with static sprite testing as there was noticeable stuttering through all the tests. The game scene was visibly choppy and frame rate was not constant. Still Ex2D managed to deliver the highest frame rate after Futile with the 5000 sprites tested.

On the other hand Ex2D performed stably with animated sprites involved. Frame rate kept up without any throttling until it dropped with 2000 static sprites and 1000 animated sprites down to 29, which was still higher than what the other frameworks achieved. Unity editor itself worked smoothly even when sprite count was up at 5000 sprites which is very praiseworthy.

2D Toolkit performed well on average throughout the testing only slightly losing to Ex2D and Uni2D with static sprites. Between these tests the average did not seem to drop down until the halfway where animation count went to 500. 2D Toolkit also performed all around good inside Unity editor and there was no visible lag with huge numbers of sprites involved simultaneously.

Uni2D performed well and consistently throughout the tests. The frame rate was a constant 60 with lower sprite amounts but dropped similarly compared to the other visual frameworks with 3000 and 5000 sprite counts. However, in comparison, the frame rate dropped earlier with the animated sprites and only managed a frame rate of 35 with 350 sprites whereas 2D Toolkit and Ex2D pulled solid 60. Nevertheless Uni2D did redeem itself a little by having a higher frame rate than Ex2D and Orthello with the highest animated sprite count. Editor side performance was also good with all the tests although some lag was present but nothing work hindering.

Summary

Futile outperforms every other framework tested here but it is only useable with user written code. Orthello gave mid-range performance but throttled heavily with any animated sprite amount and failed to build with 5000 sprites. Also the editor side performance with Orthello quickly became unbearable after some 500 sprites becoming close to impossible to use after 3000. With Ex2D frame rates did not seem to budge when animations were present and it gave the best averages compared to the other visual frameworks. There was some visual stuttering involved with a few tests and frame rates were slightly jumpy. 2D Toolkit's frame rates were mediocre throughout the tests, but it was the first one which had a dip with static sprite performance. Uni2D performed similarly to Ex2D and 2D Toolkit but struggled a bit with mid-range animated sprite counts.

All the frameworks benchmarked demonstrated strengths and weaknesses in at least one area. In the end the differences of performance between Ex2D, 2D Toolkit and Uni2D are rather small. Orthello managed to perform similarly in some tests but failed badly in others. Futile's performance was obviously the best but is not exactly comparable as it does not use the visual editor.

4 PROTOTYPE: CASE

This chapter discusses basic 2D game development theory elements involved in the making of the game prototype and what developers may be dealing with when creating 2D games. The same things are also gone through in practice by showing examples of how they were implemented in the prototype.

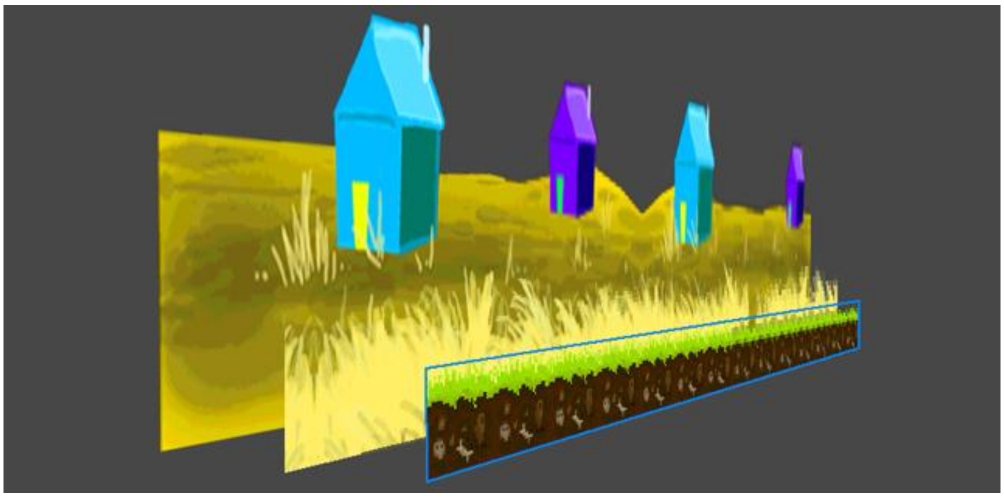
The prototype itself is a fairly straightforward classic 2D platformer. The concept revolves around the use of fluid physics and simulation of water. With the water the player is meant to solve puzzles that interact with the water particles. Other features include basic movement for a 2D platformer, mouse cursor aiming as well as climbable swinging ropes. Any integral tools that were required during the project are also covered.

4.1 Theory behind the designs

Parallax scrolling is a special technique commonly used in video game and flat image based graphics. This method allows the developers to create pseudo 3D or an illusion of depth to the game scene making the scene look more engrossing. (Wyatt 2007). What this technique does is that it makes the background game objects to move at different rates independent from each other (Dane 1988).

For example, if the foremost layer where the player is rendered moves forwards at a constant of 1.0 while the background layer moves at the speed of 0.8, making it slightly slower. In other words the further something is in the distance, the less it appears to move. (Stahl 2013.)

Parallax scrolling may be achieved multiple ways and here are the 4 most commonly used methods: the layer method, the sprite method, the repeating pattern method and the raster method. The layer method focuses on adding the objects onto different layers as shown in picture 8 below. The layers have their own movement constant for vertical and horizontal axes which causes the perception of depth.


Picture 8. Parallax layers in Unity editor.

The sprite method uses pseudo layers which are on the same layer but the sprite draw order or movement could be decided beforehand. This method works more or less just as the layer method. (Wikipedia 2013.)

The repeating pattern method repeats a background image, a sprite for example, infinitely. The image has two stages how it is displayed. The image has always two graphical objects which are swapped always in front of each other, making the game consume less memory instead of looping the background many times. (Wikipedia 2013.)

The raster method is what has been used in older video games that were dedicated for TV or similar handheld systems. This method creates an illusion of depth by displaying the pixels of an object from top to bottom order with a slight delay. (Wikipedia 2013.)

Unity allows you to choose from these techniques as it makes most of them possible. The layer method was used to achieve parallax scrolling in the prototype.

Sprites in relation to modern video games, usually refers to two dimensional partially transparent image objects. Sprites are always placed sideways or two dimensionally towards the camera so that the image is always at the correct angle. They can be rotated on their z-axis but rarely on x- or y-axes as that would break the illusion. Sprites can also be scaled to simulate depth for the sprite. Sprites can be either static meaning they do not change but can move or animated sprites where the sprite actually consists of multiple different images that are rendered one after another.

In this case project all the images you see, except backgrounds, are made from sprite images. Even larger entities like the ground is made from repeating 32x32 sized textures.

Texture atlases are large collections of different textures placed on the same image file. Texture atlases always come with a data file that contains information about the atlas such as the texture names, their locations coordinates in the file and their size. (Ivanov 2006.) Texture atlases have several advantages over using sprite sheets. For one you will be able to call the sprite from the atlas by its name or index number so you do not need to know where it is located on the sprite sheet. You can also have sprites of many different sizes in the same atlas as knowing their size becomes will not be trivial.

There are even more advantages in using texture atlases, especially when looking for performance in 2D games that are rendered with 3D engines like Unity or DirectX. It is highly recommended trying out texture atlases instead of sprite sheets to see if it helps out. (Freeman 2013.)

Sprite sheets unlike texture atlases do not have a data file and all the image locations need to be manually defined upon calling the sprites. Sprite sheets as well as atlases are commonly built from power of two sprite tiles which means that sprite size is multiplier of power of two so the sprite could be of 8, 16, 32, 64 pixels wide or tall and so forth.

Variations may occur as sprite object width and height does not necessarily need to be the same making it possible to create 16 pixels wide and 32 pixels high objects. In 2D development this approach is typical and most of all the game engines perform better when all graphics are provided in size of power of two as non-power of two textures consumes more memory.

The graphics look more pure when this rule is used since there is no need to stretch the graphics to fit a cube game object, in example. Also there is no need to think if the graphics object is fully covered. It also helps to create the world in different sizes as sprites and game objects can be scaled freely without distortion.

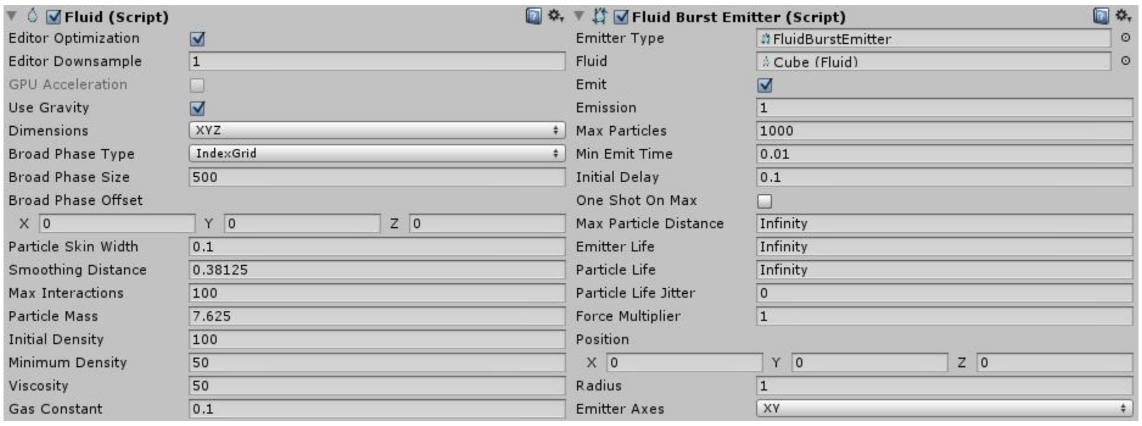
Water has always been tricky subject in game development. In 2D games the developer has to make a choice how realistic he wants to have his water as performance will be largely affected by it.

The method for simulating water we will discuss here is about using a Fluvio particle emitter[22] (chapter 4.2 for examples) which emits small particles that collide by the use of ridigbodies[23] on the particles. In Unity using this method is quite performance costly as having hundreds or more of colliding rigidbodies will make quite the dent on frame rate.

Another very performance wise important issue is the type of shader[24] these particles use. There are tens of different stock shaders Unity provides, thus using flashy looking shaders may not be the best idea even though it might look good. Fluvio also comes with a handful of custom shaders for making good looking fluids.

Making the particles look like water is the easy part. Making the particles act like water is a bit harder. Fluvio provides a long list of settings to tamper with and finding the right balance can be a tedious process of trial and error.

There are settings like: mass, density, viscosity, force, damping, velocity and smoothing distance to mention some. There is no point trying to give a recipe for a good mixture for these as it will be scenario specific how they need to be setup. Most of the available settings are shown in the picture 9 below.



✓ Fluid (Script)		✓ Fluid Burst Emitter (Script)	
Editor Optimization	✓	Emitter Type	FluidBurstEmitter
Editor Downsample	1	Fluid	Cube (Fluid)
GPU Acceleration	☐	Emit	✓
Use Gravity	✓	Emission	1
Dimensions	XYZ	Max Particles	1000
Broad Phase Type	IndexGrid	Min Emit Time	0.01
Broad Phase Size	500	Initial Delay	0.1
Broad Phase Offset		One Shot On Max	☐
X 0 Y 0 Z 0		Max Particle Distance	Infinity
Particle Skin Width	0.1	Emitter Life	Infinity
Smoothing Distance	0.38125	Particle Life	Infinity
Max Interactions	100	Particle Life Jitter	0
Particle Mass	7.625	Force Multiplier	1
Initial Density	100	Position	
Minimum Density	50	X 0 Y 0 Z 0	
Viscosity	50	Radius	1
Gas Constant	0.1	Emitter Axes	XY

Picture 9. Fluvio particle settings.

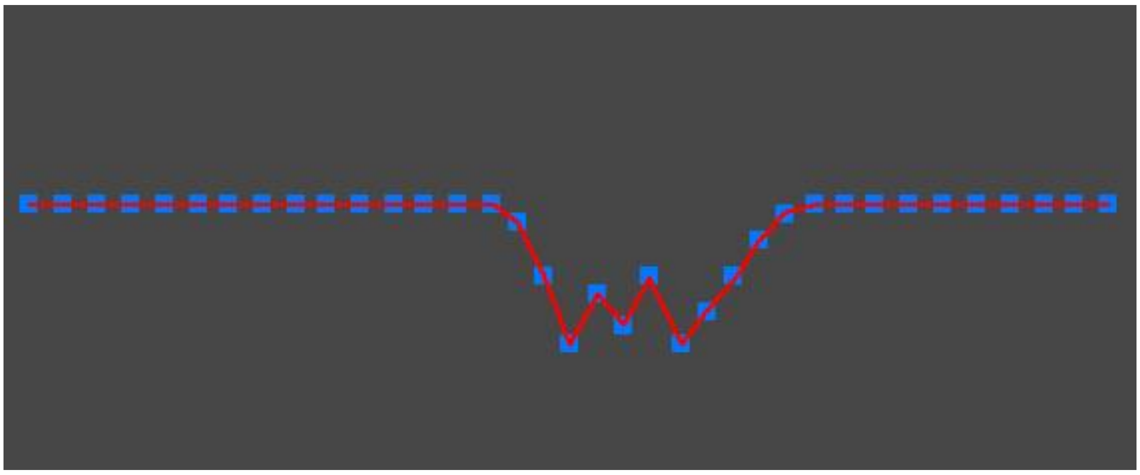
For optimization in the prototype; larger particles and the type of shaders that made the particles look like one larger entity when together were chosen in

[22] Appendix 1: Particle emitter.
[23] Appendix 1: Rigibody.
[24] Appendix 1: Shader.

order to reduce the amount of particles needed for something that looks like liquid. An in-game resource type of management was also included in the prototype that limits the amount of particles rendered simultaneously.

Faking the water in certain points was also considered but it did not actually make into the prototype. There would be no point keeping hundreds of particles in the game scene, if for example the player had to fill a container or a pool with particles to make it look like it is full, as that would be a memory and performance consuming process. A technique described below could be used for that.

Merging the game objects together is used in 2D water surface which consists of objects in horizontal line at fixed points as shown in picture 10. The points are able to move a certain amount in vertical direction causing them to drag some of the attached points with them, making it look like something hit the water surface. The dots are trying to maintain their position in a certain line causing them to reset their position where they were before the object impacted with it.



Picture 10. Waterline stretching connection demonstration.

If the game prototype had had a larger project scope, implementing this method to the game would have been a good idea as memory consumption would otherwise get out of hand really quickly.

Climbing ropes and ladders can be made in several ways in Unity. There are probably as many different solutions as there are actual implementations.

Climbable ladders are quite easy in theory. The first colliders are needed both on the player and the area of the ladder that will trigger upon collision or when the player presses a button while the trigger is being called. At this point it a good to disable whatever movement script is attached to the player and latch the player in the ladder for example by updating the player’s transformation[25] with the ladders transformation. Then activate another script for climbing specific movement that simply enables movement on the y-axis. Detaching could be done by the press of a button as well that in turn de-actives the climbing script and re-enables normal movement.

One of the features done for the prototype was rope climbing. Both the rope and the climbing part took quite a while get to work since Unity does not give any out of the box tools for either of them so using ones imagination becomes vital for coming up with a good solution. One viable solution for creating a rope is to use several capsule colliders that act as segments of a rope, something like pieces of an iron chain for example. These segments are then connected to each other via characterJoints in order to simulate the ragdoll like behaviour of a rope.

The movement of the rope also needs to be restricted into 2D plane otherwise it will swing in all directions, which may cause the character to move outside of the platform boundaries and fall off the map or cause other kinds of bugs.

4.2 Tools

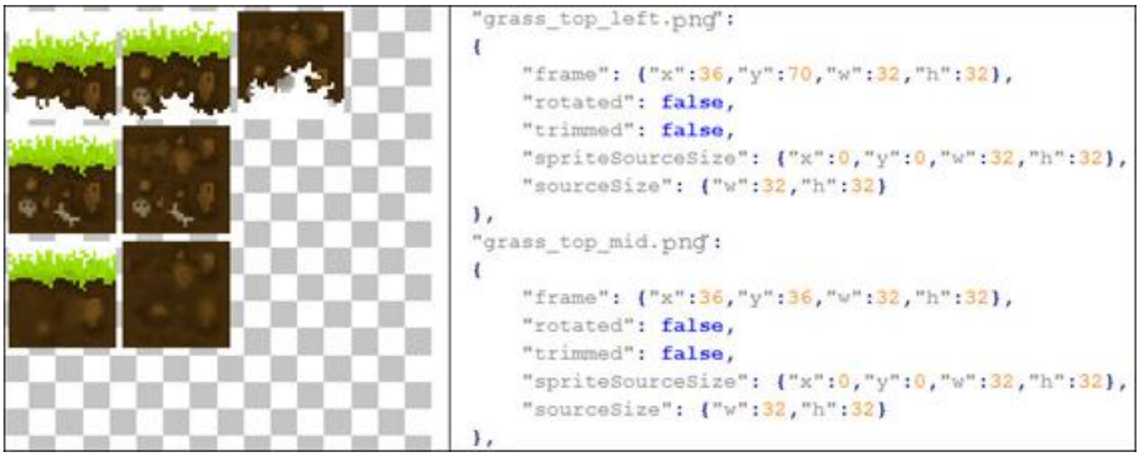
Choosing the right tools is one of the most important issues in a software project, especially when creating a video game. This is because once they have been chosen it is possible that they cannot be changed in the middle of the project, even if the need arises. Now this obviously does not apply to all of the tools at your disposal but for example changing the game engine would probably mean redoing most of your work.

[25] Appendix 1: Transform.

A good choice is to go with the tools used before as knowing how they work saves time from learning new environments and creates an air of expertise. Still it is always a good idea to try and get out of the comfort zone and see what other solutions there might be available since there is always new and better software coming out which could smooth the workflow.

In this project the first tool chosen was Unity (chapter 2). It was the most important decision to make first since it lays down the foundation for the need of other tools in the project as everything needs to revolve around the game engine. This chapter will mainly go over the tools needed for handling and utilizing 2D graphics.

TexturePacker is a texture packing tool for compressing separate images into texture atlases which are collections of images in a single file. There is a small sprite sheet of grass and dirt tiles and a snippet of the atlas data file that contains data for that sheet in the picture 11 below.



```
"grass_top_left.png":
{
    "frame": {"x":36,"y":70,"w":32,"h":32},
    "rotated": false,
    "trimmed": false,
    "spriteSourceSize": {"x":0,"y":0,"w":32,"h":32},
    "sourceSize": {"w":32,"h":32}
},
"grass_top_mid.png":
{
    "frame": {"x":36,"y":36,"w":32,"h":32},
    "rotated": false,
    "trimmed": false,
    "spriteSourceSize": {"x":0,"y":0,"w":32,"h":32},
    "sourceSize": {"w":32,"h":32}
},
```

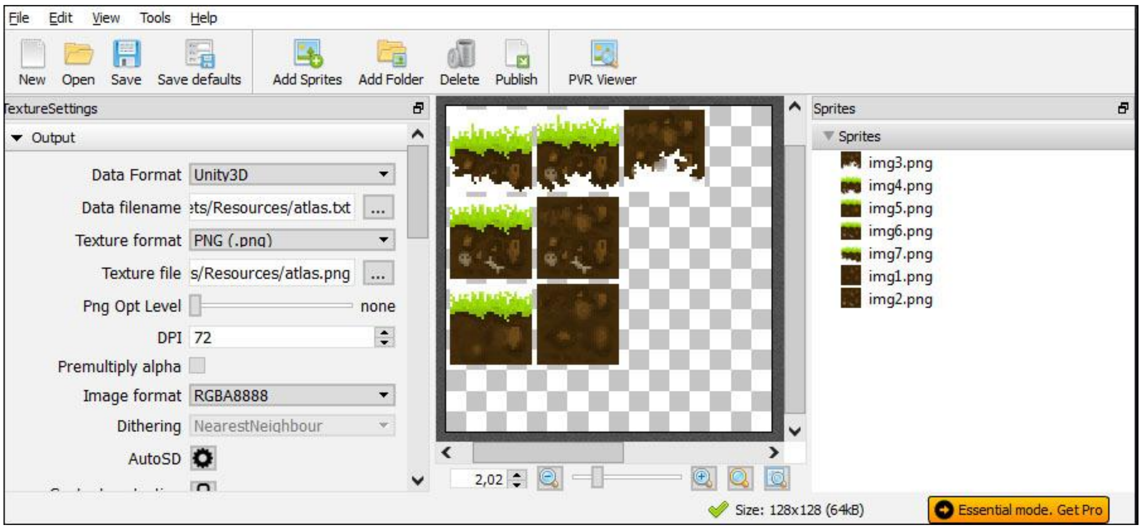
Picture 11. Sprite sheet and atlas data file for the sheet.

Not all of the frameworks introduced in chapter 3 for Unity require a third party atlas generator as Ex2D, 2D Toolkit and Uni2D introduce a possibility for creating atlases inside the frameworks themselves. However Orthello, which was used for the prototype, requires a sprite atlas generator as does Futile.

Creating atlases with TexturePacker is quite straightforward even though there is plenty of room for configuration too. Settings like trim, crop, rotate, scale and smooth among other options are available.

Textures can be simply dropped into TexturePacker in the section which contains .png images in picture 12 below. This will automatically arrange the textures into a sprite sheet with basic settings.

There are some base settings that should be taken in account. Data format should be set to Unity3D. Data filename is the atlas file that is loaded in to Unity and it has to be .txt format as that is the only format Unity can handle. Texture file is the actual sprite sheet and you can choose any image format that Unity supports.



Data Format	Unity3D
Data filename	ets/Resources/atlas.txt
Texture format	PNG (.png)
Texture file	s/Resources/atlas.png
Png Opt Level	none
DPI	72
Premultiply alpha	☐
Image format	RGBA8888
Dithering	NearestNeighbour
AutoSD	

img3.png
img4.png
img5.png
img6.png
img7.png
img1.png
img2.png

Size: 128x128 (64kB)

Picture 12. TexturePacker GUI showing an atlas.

As far as the rest of the settings go there is nothing that necessarily has to be forced. Most of the settings affect the geometry of the sprite sheet. Now all there is left is to hit the “Publish” button and it is all ready for use. Some of the options like image optimization or algorithms are not allowed in the free version of TexturePacker but a notification is given if such features are used when trying to publish the texture atlas. At this point it is very framework specific how to utilize the atlas.

Revision Control

Managing the project source is crucial for any software development team. There is a limit on how long people care to keep moving files from one person to another via flash drive or email. Even though such methods might work to some extent with two or few people, the advantages of using version control far out-weight the opposite. There are plenty of version control programs to deal with this issue and even several online services that provide you free hosting for the project.

Revision control is not necessarily required for sharing things such as the 2D frameworks as they themselves do no chance but sharing the things created with them like sprites, animation and scenes would be a good idea.

Using revision control with Unity can be a little tricky. The problem lies with the sheer quantity of binary-files that Unity creates for objects in the project and obviously binary-files are something that cannot be merged with revision control software. In other words, Unity-scenes would break, if two people were to make simultaneous changes that would affect the same binary files.

Luckily there are several solutions and workarounds for this problem. One rather laborious solution is to simply to track what people are doing and make sure they do not touch the same game objects.

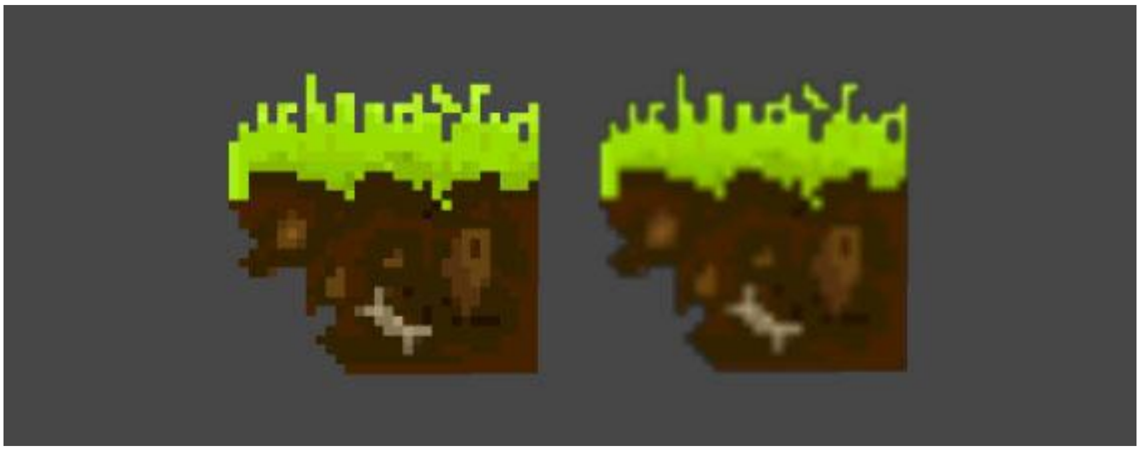
Setting up the actual version control to work with Unity is fairly straightforward. Enabling meta-files in the Unity editor settings should be the first thing to do. Now every asset will identify itself with a specific meta-file and all the changes made to these assets are written in these meta-files in binary format.

Certain folders and files should also be ignored from the version control's end. These are Temp and Library folders as well as files ending with .pibd, .sln, .userprefs, .csprog and .orig. These folders and files are ignored since they are not needed and most of the time they are generated on Unity's side when the game is imported or compiled. It saves space from the repository download and upload wise.

Orthello was the 2D framework chosen for the case project. The decision at the time was made based on how appealing Orthello looked on paper. Orthello also had broad documentation which made adopting it easy. There was no proper source of reference about what framework would be the best and some of the frameworks discussed in this thesis were not yet released. This is one of the reasons for writing this thesis. Even though Orthello may not have been the best framework to choose from, the project requirements back then included that the framework had to be free. Orthello is still the only 2D framework around, that offers an arsenal of tools and is free.

Working with Orthello was straightforward as everything else except scripts and atlas generation was automated. Learning how to create sprites or animations took no time at all and building the scene was actually quick after learning the basics. Setting up Orthello is covered earlier in chapter 3.1.

Orthello itself does not require that much configuring from the user but there are certainly some configurations that should be made inside the Unity editor for textures to make everything work for sprites. For example graphics settings in texture atlases should be set to point -mode as otherwise anti-aliasing will smoothen the pixels of the sprites. This setting does not fit a game that utilizes sprite graphics and prefers pixel perfect[26] quality as seen in the picture 13 below where right sprite presents aliased mode and left point -mode.


Picture 13. Filter mode difference between point and trilinear modes.

Fluvio is a tool for simulation of fluid dynamics. Fluvio is made for Unity by Thinksquirrel Software LLC and is meant for simulating real-time fluid dynamics both in 2D and 3D environments. It can be used but not limited for simulating accurate liquids, gasses, fire, volumetric smoke and fluid interactions.

[26] Appendix 1: Pixel perfect.

(Thinksquirrel Software LLC 2013.) Fluvio can be purchased from the Unity asset store for a moderate fee.

The projectile shooting covered in the next chapter is achieved with the Fluvio particle emitter which is quite simple to setup. A fluid object can be created from the Fluvio toolbar which contains a script for handling everything related to the particles such as the quantity and shape. The actual emitter is found under the fluid object in the scene hierarchy.

The emitter has a script that in turn contains everything emitter related like how fast particles are emitted and at what velocity. There are plenty of options for tweaking the particles to be whatever you need from realistic waterfall to a couple of bouncing particle balls. What the particles look like is largely set by shaders, which come in many forms. Unfortunately some of the fancier shaders needed for more realistic fluid simulations are not available with the free version of Unity.

It is also noteworthy that Fluvio comes in a couple of different editions and only the most expensive one of these contains access to source code. This means that tinkering with the particle or emitter functionality can be tricky. Luckily most of the functions the source code contains are public and can be hooked into from custom made scripts. For example shooting particles at command via key-press requires a script that calls methods or variables from the emitter script.

4.3 Creating the prototype

Design cycle started in December 2012, months before the actual prototyping phase started. The game prototype was designed to investigate how water physics would work in a 2D game environment by creating a small map with some kind of a physics puzzle inside it. Aim was to make a new kind of 2D puzzle mechanic with the help of real time fluids. Testing something new with the prototype was one of the corner stones of the project as it would have a large impact on how interesting the game would be to others.

Concept of the game is classically generic where the environment is created with simple sprite graphics. Player can climb obstacles and ropes and jump over life threatening pits. Enemies would be quite basic as well with simplistic path finding and player position tracking. Their attacks could be countered with a water gun and they would be defeated by soaking them or by knocking over interactive environment such as crates.

This chapter demonstrates how to create some of the features that were made in the prototype. The examples are shown in short code snippets or found from the appendices.

Resource Management

A pretty user interface was not in the project scope or priorities but something was still needed for displaying basic game economy[27] to the player even if just for testing purposes. In this case game economy stands for player's health and ammunition.

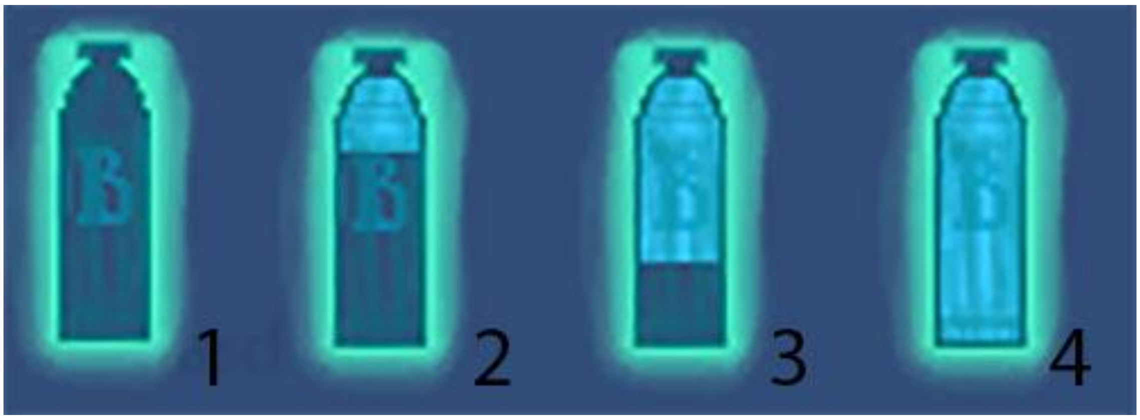
Creating a bar for player's current amount of usable ammunition was quite simple. The same code can also be for also utilized for displaying player's health or anything where you need a bar that simply gets smaller and bigger as you consume a resource. Listing 2 below gives an example of how to create such a bar.

Two textures (picture 14) were created, one for a resizable bar and one for a fixed container that holds that bar (Lines 3 and 4). This was done without the help of 2D frameworks since Unity's own GUI-tools offer an excellent way for simple texture clipping[28] with the help of Unity's GUI methods and Rect structure. The bars are drawn in groups and then resized during runtime by adjusting newBarHeight variable as the defaultWaterAmount runs down (Lines 16-26). The defaultWaterAmount variable also needs to be updated according to how much resources the player has used reflects to the defaultWaterAmount for any effect to take place.

[27] Appendix 1: Game economy.
[28] Appendix 1: Texture clipping.

```
public class DrawWaterBottle : MonoBehaviour
{
    public Texture2D waterBar;
    public Texture2D waterBottle;
    float waterBarMaxWidth;
    float waterBarMaxHeight;

void Start ()
{
    waterBarMaxWidth = waterBar.width;
    waterBarMaxHeight = waterBar.height;
}

void OnGUI()
{
    float newBarWidth = (defaultWaterAmount/100) * waterBarMaxWidth;
    float newBarHeight = (defaultWaterAmount/100) *waterBarMaxHeight;

    GUI.BeginGroup(new Rect(5,6, waterBottle.width, newBarHeight));
    GUI.DrawTexture(new Rect(0,0, waterBar.width, waterBar.height),
         waterBar);
    GUI.EndGroup();
    GUI.BeginGroup(new Rect(5,6, waterBottle.width,waterBottle.height));
    GUI.DrawTexture(new Rect(0,0,waterBottle.width,waterBottle.height),
         waterBottle);
    GUI.EndGroup();
}
```

Listing 2. Example code for drawing a resource bar.



Picture 14. Water bottle and water textures blending together.

Movement for a 2D platformer

There are plenty of ways to go around doing the movement for a 2D character but in this case something precise was required since the “feel and touch” of controlling the player in a platformer environment is crucial. Something precise and predictable is good. Knowing exactly where a jump will land can mean the difference between game over and accomplishment. For such cases Unity provides the CharacterController[29] component.

The character controller is essentially a capsule collider attached in the player object. There is some controversy whether or not one should use Unity’s character controller. It is unaffected by rigidbody physics and outside forces unless specifically told so, meaning that any physical properties need to be manually coded. Collisions with other objects that do use rigidbodies have to be handled separately as rigidbody is needed to handle collision triggers. If you need rigidbody physics then character controller is probably not the best way to go about controlling the player.

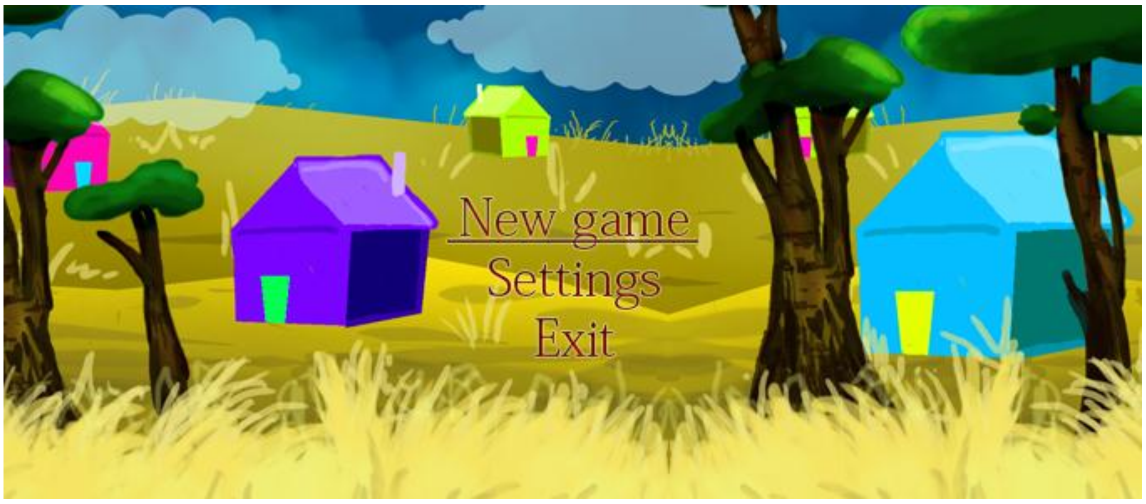
For this prototype the character controller was suitable as the movement is pretty simplistic. One way to utilize the character controller is shown in a movement script in appendix 4. The script is trimmed to a very basic layout that can be used as a reference for building your own script.

Menus

The main menu in picture 15 was put together with the Orthello framework by creating normal sprites to the scene and inserting scripts in them that utilize Unity’s GetMouseButtonDown method which detects mouse click. The new game button would simply call Unity’s Application.LoadLevel method which in turn loads the corresponding game scene.

As Unity is a 3D engine, the user interface could have been 2.5D, which is a common way for highlighting certain items on screen in 2D games. Another possibility provided by Unity is to create a user interface by script using Unity’s own GUI functions but working with the Unity GUI-tools is difficult and the functionality is generally cumbersome.

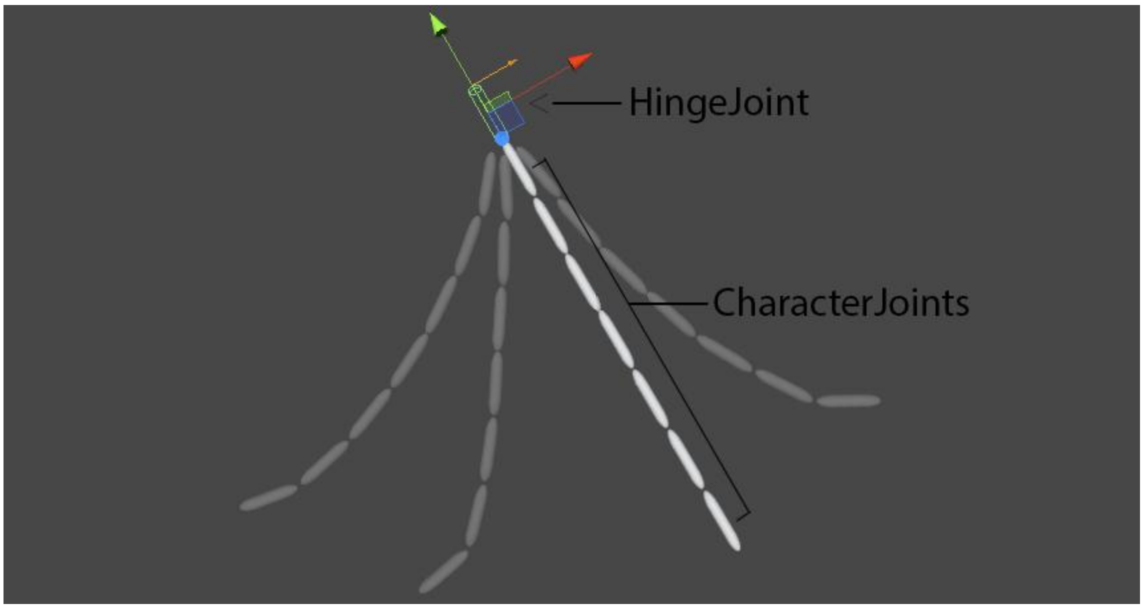
[29] Appendix 1: CharacterController.



Picture 15. Simple sprite menu.

Rope and Climbing

The top most segment of the rope has a hingeJoint[30] attached to for creating a hinge like movement between it and the next segment. This is what creates the swinging motion of the rope. Rest of the rope is just characterJoints[31] linked together as shown in picture 16 below.



Picture 16. Components of the rope.

An additional script containing an update method was attached into each segment. The scripts constantly update ropes z-axis's local position well as the

[30] Appendix 1: HingeJoint.
[31] Appendix 1: CharacterJoint.

x and y euler-angles[32] to 0 in order to keep it strictly on the two dimensional plane.

The climbing logic is rather simple. Every rope segment is a trigger and contains a script that will invoke the climbing script which in turn is attached to the player. When the script is invoked, it stores data from collision; the segment and the rope. These are used to manipulate player's position during the climb. The script is presented as pseudo code because a full example would be too long and it would not necessarily work straight out of the box. The pseudo code shown in appendix 5 is meant for a base to give ideas about how to go around building scripts for climbing ropes.

Shooting

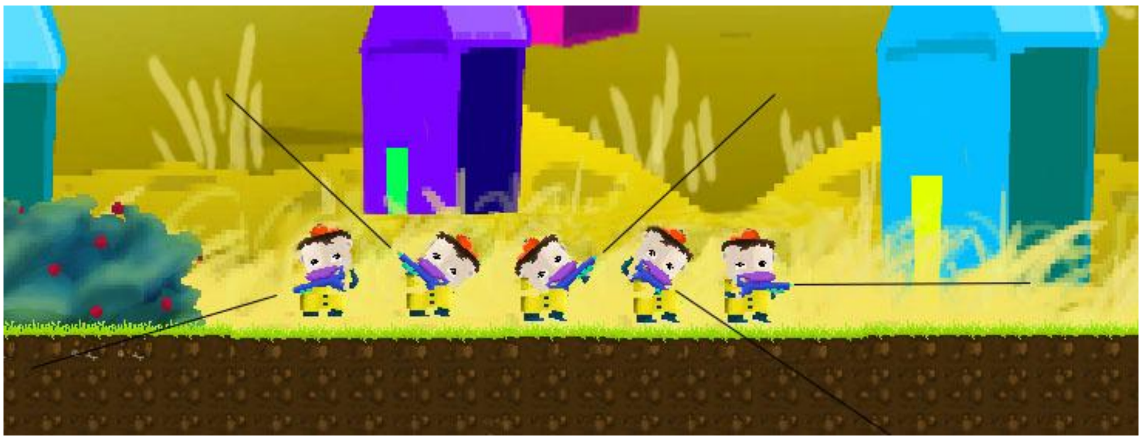
The weapon aiming mechanism in listing 3 was designed to work so that player can instantly aim and fire in any direction with a movement of a mouse much like in picture 17. Unity's LookAt method (Line 11) can be taken advantage to achieve this, which is one of the many convenient functions that Unity's scripting API provides. The x and y mouse coordinates (Lines 16-31) have to be inverted and the weapon objects rotated around its y-axis by 90 degrees for correct 2D orientation (Line 13). This simple logic gives a nice 360 degree rotation for the weapon object following mouse movement. The script works as it is when attached to a game object such as a weapon texture.

```
1   public class Aiming : MonoBehaviour
2   {
3       public float speedRot = 1.0f;
4       private float mouseX;
5       private float mouseY;
6
7       void Update ()
8       {
9           mouseX = Input.mousePosition.x - Screen.width / 2;
10          mouseY = Input.mousePosition.y - Screen.height / 2;
11          transform.LookAt    (speedRot    *    new    Vector3
12          (invertValueX(mouseX), invertValueY(mouseY),0));
13          transform.Rotate(0,90,0);
14      }
15
```

[32] Appendix 1: Euler-angle.

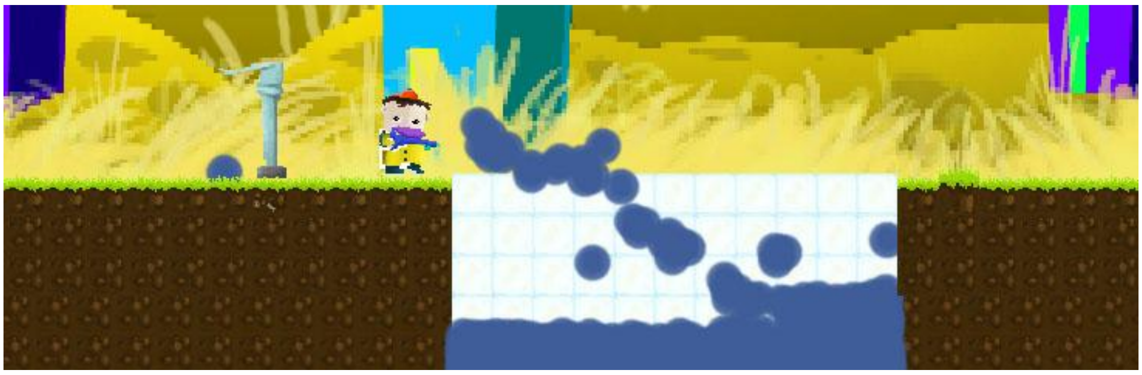
```
        public float invertValueX(float mouseX)
        {
            if (mouseX < 0)
            mouseX = System.Math.Abs(mouseX);
            else if (mouseX > 0)
            mouseX = -System.Math.Abs(mouseX);
            return mouseX;
        }

            public float invertValueY(float mouseY)
            {
            if (mouseY < 0)
            mouseY = System.Math.Abs(mouseY);
            else if (mouseY > 0)
            mouseY = -System.Math.Abs(mouseY);
            return mouseY;
        }
    }
```

Listing 3. Cursor aiming for 2D platformer.



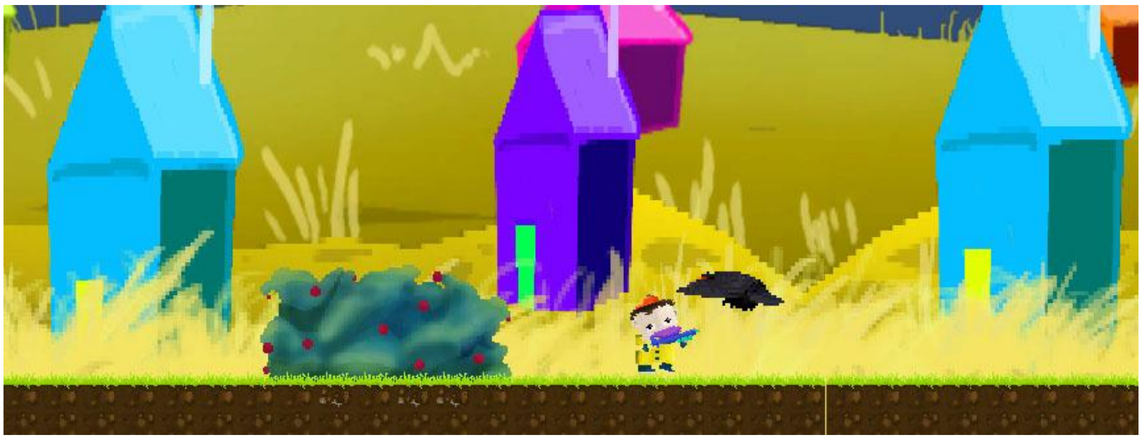
Picture 17. Example collage of how the aiming roughly follows the cursor.

As mentioned in 4.1 the actual shooting shown in picture 18, is done with the Fluvio plugin. With 2D, the z-axis has to be disabled for the emitter object as the fired particles would otherwise fall off the platform but fortunately Fluvio has built-in options to limit the axes. The particles are very hard on performance depending on what settings are used. The prototype had pretty elementary looking particles with simple shaders and crude collision in order to keep up decent frame rates. Around 500 simultaneous particles appeared to be limit. This performance issue could be fixed with the trick mentioned in chapter 4.1, by creating a water surface system that could be lifted up via script when water particle hits its surface but it did not fit the project scope.


Picture 18. Shooting particles into a pool (about 500 particles).

Enemy AI

One simple AI was introduced in the prototype. The enemy prediction was planned to be accurately as our game was intended to focus on hard enemies alongside with the puzzles. The enemy, a flying crow shown in picture 19, had a couple requirements for attacking the player. First it needs to check whether the player is running below it in certain area of effect. This is done by placing two collider objects near each other. If the player is walking between them, the attacking commences.


Picture 19. Crow AI attacking the player.

Second, if the crow hits the player it will return back to its patrol route and a cooldown begins, before another attack can start. If the enemy crow does not hit the player it goes back to the patrolling route and resumes on attempting to attack the player without a cooldown being initiated.

The movement of the crow is mostly done with Unity's own MoveTowards[33] method which moves a selected object transform towards another. Some unpredictability is added by using a fixed modifier for how much the crow will miss the exact position of the player. The AI logic is included in the appendix 3 which is code originally written by our peer Simo Riikonen.

Dealing with 2D space makes creating AI's much easier as only x and y coordinates need to be dealt with and in most cases the z-axis can be ignored. This allows creating fixed paths along just the x and y axes while not worrying will or won't the player walk through there. It also negates the need for complex pathfinding algorithms such as the commonly used A* search algorithm[34] even though using A* or similar would be advisable if more unpredictable movement patterns for the AI are needed.

5 CONCLUSION

In this thesis we set out to examine various different frameworks currently available for Unity. We examined and compared their basic functionality and features. Most importantly we ran several performance tests to see their capability handling sprites with the tools they provided. The frameworks selected were Futile, Orthello Uni2D, Ex2D and 2D Toolkit.

The frameworks were chosen by two criteria; they had to be well established with a notable user base and include a collection of different tools for handling sprites. The first criteria was established by following the developer community and surveying what tools other developers were using as well as what they were recommending to use. The second criteria was settled by comparing the frameworks and disqualifying those that only managed very basic sprite handling.

[33] Appendix 1: MoveTowards.
[34] Appendix 1: A* algorithm.

We also covered theory and practice behind the techniques used in the case study. This included elementary sprite handling in 2D games and texture atlasing as well as explaining the theory behind the important prototype features including parallax scrolling, a fluid particle system and classic 2D platformer mechanics like climbing ropes and obstacles.

The theory is also heavily linked to the tools utilized with the project. Most crucially we explain how atlasing works with TexturePacker atlasing software and how water particle mechanics can be implemented with the Fluvio particle emitter plugin. TexturePacker was chosen because it offered all the tools we required for basic atlasing and because it had a free version available. Fluvio on the other hand at the time we chose it, was the only capable particle emitter that did what we needed for the prototype.

5.1 Results

Here are the answers to the research questions laid at the beginning of this thesis, which included comparing their features, advantages as well as disadvantages and their performance.

How do the frameworks compare feature wise and what are their obvious pros and cons?

We came to the conclusion that here is no definite choice between these frameworks since each of them comes with unique features and areas where they strive to cater for specific audiences, while having the same core package containing basic sprite tools listed in table 2.

What is important is to evaluate these frameworks on individual project basis to find the most suitable features. We can, however, provide some basic scenarios where these frameworks might serve their purpose best.

Futile is a framework which we do not see applicable in larger projects. Managing your project from code only will immediately exclude any non-programmers from building the project directly. Artists and designers would not be able to participate in building game scenes which would hinder your teams work flow. We mostly see Futile for programming enthusiasts with “do it yourself” approach on game development. Futile is a little hard to evaluate feature wise due its currently lacking documentation but we do not see any obstacles in realizing any functionality as the framework is quite manipulable. It does come with basic classes for sprites and using texture atlases to mention the minimum requirements but for example leaves creating animations to a more abstract solution for the user to figure out.

Orthello was the framework we chose to carry out our prototype project with. Orthello offers a free version with slightly limited features compared to the commercial version. This is excellent for someone who wishes to test Orthello before making the purchase if better features are needed. Other frameworks do not offer such opportunity. Orthello is very easy to approach. It had a very good documentation and good example projects available. Orthello has a heavy prefab approach meaning that each different sprite or other object provided by Orthello is a different kind of a prefab. There are quite a few of them which might feel slightly overwhelming at first. The biggest issues we had with Orthello were performance problems within the Unity editor itself which rendered scene building difficult a times. Orthello can be used from both code and the editor which creates a good balance between the two. We find that Orthello is very good framework for most projects where you need many different ways to manipulate sprites but do not require extreme amounts of sprites within the scenes as that quite easily leads to performance issues.

2D Toolkit is the most popular item in entire Unity asset store which itself stands as a testament to its quality and it does deliver a very professional air about it. 2D Toolkit has wide range of tools provided and if they are not enough you can modify them as the source code is included in the price. Out of all the frameworks tested here we feel that 2D Toolkit is the most versatile and suitable for most 2D application we can think of. 2D Toolkit also delivers tools for building graphical user interfaces which is a commodity that no other framework delivers as well as very powerful atlasing tools. We recommend 2D Toolkit for any serious projects where you need state of the art 2D tools.

Ex2D comes with internal 2D Scene Editor which is not provided by other frameworks. It simplifies the workflow apart from Unity scene editing and it provides information that would otherwise be hidden such like camera position viewports and how many sprites are currently drawn on the area camera is cast on. It also removes extra space from the game scene. Ex2D scene editor introduces simplified built-in layering system reducing the amount of otherwise manual work needed for that. As with 2D Toolkit we feel that Ex2D manages to deliver a very professional package and as such is applicable for most projects.

Uni2D stands out with the easiest work flow. Everything is very simple, minimalistic and intuitive. We managed to set up and create our test scene without relying on any documentation or tutorials. You can simply drag and drop images into the scene and they automatically become game objects. Uni2D tries it best to be different as well. It is the only framework that has skeletal animation support which can be a very important feature for many projects. The sprite animation tools are also the best we tested. We recommend Uni2D for any animation and art heavy projects.

What kind of performance do they offer compared to each other when rendering sprite graphics?

We batted several 2D frameworks against each other to see how they compare. Performance wise the results were enlightening as it is quite important to any developer and especially to someone who targets mobile devices to know how it will run.

Choosing a popular Android device Samsung Galaxy III was a very good choice as it will best cater to android developers. Obviously deploying the tests on multiple different mobile devices and operating systems would have given more definite results but this was the best we could do with the resources available.

Performance results in chapter 3.5 were surprisingly constant although there were clear differences in some of the test areas. While dealing with static sprites only Futile performed clearly better that the rest of the frameworks. Otherwise the visual frameworks pulled quite similar results excluding the highest test count where Unity did not manage to build Orthello's test project. Whether the fault is with Unity or Orthello remains unclear.

Test results analysed in chapter 3.6 did not remain quite as consistent when we included animated sprites into the equation. Again Futile's unmatched performance overwhelmed the visual frameworks. Ex2D managed also visibly better than the rest of its competitors. 2D Toolkit's frame rates were slightly better than Uni2D's in a couple of tests but evened out with the highest sprite counts. Orthello on the other hand was significantly weaker with the lowest animated sprite counts but also improved its performance during the last tests to match the other visual frameworks.

We conclude that Futile is the best choice if performance needed is beyond the capabilities of what Uni2D, Ex2D, 2D Toolkit or Orthello managed to offer in these tests. However, Futile's fundamentally different development approach will not make it a very attractive tool for most people. The rest of the other frameworks performed quite similarly with a few exceptions. In the end the choice between these frameworks is best done by deciding which framework offers you the best features and work flow.

Which framework would best suit a traditional side-scrolling 2D platformer similar to the case study prototype?

To begin with we need to say that all of the frameworks we tested would have been capable for delivering our case study prototype. We chose Orthello but the decision was made long before we started to properly explore other options. Orthello was able to finish the job but caused problems with in-editor performance. This alone is enough to say that Orthello was not the best choice available. Futile would not have a very good choice either as we needed visual editor for artists and mixing in other plugins which would have been difficult to get working with Futile. This leaves us with Ex2D, Uni2D and 2D Toolkit and we can say with confidence that we would have been very happy with any of these three. Uni2D would have granted us some artistic liberties we initially hoped for in terms of animation and it would have been a simplistic enough for artists to use as well. 2D Toolkit would have provided us with better tools for creating GUI objects which we had to create with Unity’s own tools.

5.2 Development possibilities

There are a few matters we would have wanted to do but were unable to accomplish due to lack of resources and time. The biggest regret is that we did not have time to include Unity’s own native 2D tools which were released just before the completion of this thesis and could not fit in the timetable at that point. Including these would have given interesting comparison tests about how third party frameworks compare against native support and what are their differences. Most importantly it would have answered the question: how viable are the frameworks at this point.

Moreover, we would have liked to make our performance tests more versatile by including multiple mobile devices and operating systems in order to create some contrast. We were not able to achieve this since we did not have access to more devices.

As far as the prototype goes we are currently finished with it. We will, however, re-evaluate which framework to use if we continue work with it. In such case we would most likely choose Uni2D or 2D Toolkit based on our current experience and preference. We would also need to consider Unity’s native 2D support which we have not been able to test at this point.

5.3 Hindsight

It is always easy to find wisdom in hindsight but that is how we learn to avoid making the same mistakes again. We were all pretty green when we started working on the game prototype so nothing we did was backed by real professional experience. That is why this project was first and foremost a learning process for us all who participated. Everything from general software development know-how to project management and the work process itself taught us much about game development. Especially about how important it is to have a versatile team. We had to deal with a very monotonous group of programmers.

Mistakes were made and although we addressed most of them, some were unfortunately irreversible. The biggest mistake we made and the exact mistake we wish this thesis might help someone to avoid was choosing the wrong 2D framework to develop our prototype with. Orthello probably was not the best framework for our project even though it initially seemed like a good choice. Obviously if we were to continue the prototype into production then it would be changed but nevertheless, it would mean more work.

We would have also wanted to include more frameworks in the tests but were unable to do so for the reasons already mentioned. The same reasons apply why the tests were only done on android.

REFERENCES

1. Andrews, J. Unity 3D Game Development Proves Effective. 2007 [Referred 20.5.2013] Available at: http://blog.zco.com/unity-3d-game-development-proves-effective/

2. Dane, E. Lucas' new epic fantasy game. New Strait Times. [Web document]. Sep. 1, 1988. P. 26. Available at: http://news.google.com/newspapers?id=drgTAAAAIBAJ&sjid=S5ADAAAAIBAJ&pg=3478,303305&dq=parallax+scrolling

3. ExDev Team. ex2D. 2013. Available at: http://u3d.as/content/ex-dev-team/ex2d/2eJ

4. ExDev Team. How Ex2D Renders. 2013. Available at http://ex-dev.com/ex2d/docs/how-ex2d-renders/

5. François, B. 2D Game Prototyping In Unity 3D: Orthographic Projection. [Blog]. 2011. [Referred 20.5.2013]. Available at: http://www.previewlabs.com/2d-game-prototyping-in-unity3d-orthographic-projection/

6. Freeman, J. Advantages of texture atlases over sprite sheets. [Web Journal]. 2013. Available at: http://jessefreeman.com/game-dev/advantages-of-texture-atlases-over-sprite-sheets/

7. Gamasutra. Game Developer Survey Leans Heavily Toward iOS. [Web Journal]. 2012. [Referred 20.5.2013] Available at: http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity.php.

8. Goldstone, W. Unity Native 2D Tools. [Blog]. 2013. [Referenced 11.11.2012]. Available at: http://blogs.unity3d.com/2013/08/28/unity-native-2d-tools/

9. Ivanov, I. Practical Texture Atlases. Gamasutra. [Web journal]. 2006. Available at: http://www.gamasutra.com/view/feature/2530/

10. Jarcas Studios. An overview of 2D solutions for the Unity game engine. [Blog]. 2013. Available at: http://www.jarcas.com/studios/?p=134

11. Müller, P. Comparison of 2D frameworks for Unity 3D. [Blog]. 2012. Available at: http://www.thesecretpie.com/2012/07/comparison-of-2d-frameworks-for-unity-3d.html

12. O'Brien, J. Physics Engine Platformer Terrible Idea. [Blog]. 2013. [Referenced 25.8.2013]. Available at: http://tech.flyclops.com/tag/cocos2d

13. Pau, E. The best 2D extension tools for Unity3D. 2013. [Web Journal]. Available at: http://www.mossman.es/the-best-2d-extension-tools-for-unity3d/

14. Prime31. Unity 2D framework comparison. 2013. Available at: https://gist.github.com/prime31/3333887

15. Quintans, D. Game UI By Example: A Crash Course in the Good and the Bad. 2013. [Web Journal]. Available at: Gamehttp://gamedevelopment.tutsplus.com/tutorials/game-ui-by-example-a-crash-course-in-the-good-and-bad--gamedev-3943

16. Rix, M. Futile – An open source code-centric 2D framework. [forum]. 2012. Available at: forum.unity3d.com/threads/147246-Futile-An-open-source-code-centric-2D-framework

17. Stahl, T. Chronology of the History of Video Games. [Web journal]. 2013. Available at: http://www.thocp.net/software/games/golden_age.htm

18. Thinksquirrel. Fluvio. 2013. Available at: https://thinksquirrel.com/products/fluvio

19. Wikipedia. Parallax Scrolling. [Wiki]. 2013. [Referred 23.11.2013]. Available at: http://en.wikipedia.org/wiki/Parallax_scrolling

20. Wyatt, P. /Flash/the art of parallax scrolling. .net. 2007. Available at: http://mos.futurenet.com/pdf/net/NET165_tut_flash.pdf

Glossary

1. Unity	Unity is a cross-platform game development environment for 2D and 3D content.
2. Platformer	Platformer or a platform game is a video game genre where classically player jumps between platforms and obstacles.
3. 2D Framework	Third party 2D frameworks for Unity offer tools for delivering 2D graphics with Unity.
4. Plugin	Usually a third party tool that can be integrated with Unity.
5. Side-scrolling	A type of game where the game scene is projected from the side revealing only the immediate area. The revealed area traditionally moves to right along with the player exposing more ground while simultaneously hiding the left side.
6. Cross-platform	Software that can be run on two or more different platforms.
7. Unity asset store	Marketplace for third party Unity assets.
8. Flash	Flash is a development environment by Adobe primarily meant for creating multimedia applications for the web but can be equally used to make animations and games as well.
9. Apache Flex	Software development kit for adobe flash.
10. FlashDevelop	Integrated development environment for adobe flash.
11. GUI	Short for graphical user interface.
12. Orthographic	Projecting a three dimensional object in two dimensions.
13. NVIDIA PhysX	Physics engine used by Unity.
14. Cocos2D	Cross platform open source 2D game engine.

15. Box2D	Cross platform open source 2D game engine.
16. Game object	An object in a game that represents one individual object
17. Prefab	An asset that is a reusable game object stored in the project tab.
18. Skeletal animation	An animation method which simulates bone structure. Bone structures are connected making the animation more realistic.
19. Heap allocation	Memory management method.
20. Draw call	How many materials are being rendered on the screen.
21. Translate	This method moves a game object transform in the direction of translation.
22. Particle emitter	Particle system which creates small particles.
23. Rigidbody	Controls object position through physics.
24. Shader	Lightning and shading of a game object. Used for lighting or special effects.
25. Transform	Unity component which contains object position, rotation and scale.
26. Pixel perfect	Level of pixel rendering quality. Each pixel is rendered.
27. Game economy	Game resources: player health, ammo, money et cetera.
28. Texture clipping	Technique for cutting or hiding a part of an image.
29. CharacterController	Unity movement constraint.

30. HingeJoint	Simulates an actual hinge. Used to attach game objects to each other.
31. CharacterJoint	Similar to HingeJoint but simulates a ball-socket joint.
32. Euler-angles	Used to describe the orientation of a rigidbody
33. MoveTowards	Unity method for moving an object in a straight line towards another point
34. A* algorithm	Pathfinding algorithm for finding a route between travellable points.

Unity Licence Comparison Unity 2013

Source: http://unity3d.com/unity/licenses

General	**Free**	**Pro**
Physics	✓	✓
NavMeshes, path-finding, and crowd Simulation4	✓	✓
Multiplayer Networking with RakNet	✓	✓
LOD support		✓
Audio (3D Positional and Classic Stereo)	✓	✓
Audio Filter		✓
Video Playback and Streaming1,2		✓
Fully Fledged Streaming with Asset Bundles		✓
May be licensed and used by companies or incorporated entities that had a turnover in excess of US$100,000 in their last fiscal year.		✓
Animation		✓
Mecanim	✓	✓
Mecanim: IK Rigs		✓
Mecanim: Sync Layers & Additional Curves		✓
Deployment		✓
One-Click Deployment	✓	✓
Web Browser Integration	✓	✓
Custom Splash Screen		✓
Build Size Stripping		
Graphics		
Low-Level Rendering Access	✓	✓
Dynamic Fonts with markup	✓	✓
Shuriken Particle System	✓	✓
3D Texture Support		✓
Realtime Directional Shadows	✓	✓
Realtime Spot/Point and soft shadows		✓
HDR, tone mapping		✓
Light Probes		✓
Optimized Graphics	✓	✓
Shaders (Built-in and Custom)	✓	✓
Lightmapping	✓	✓
Lightmapping with Global Illumination and area lights		✓
Dynamic Batching	✓	✓
Static Batching		✓
Terrains (Vast, Densely Foliaged Landscapes)	✓	✓
Render-to-Texture Effects		✓

Full-Screen Post-Processing Effects		✓
Occlusion Culling		✓
Deferred Rendering		✓
Full multi-screen support (AirPlay)		
Stencil Buffer Access		✓
GPU Skinning		✓
Code		
Navmesh: Dynamic Obstacles and Priority		✓
Webplayer debugging	✓	✓
.NET Based Scripting With C#, JavaScript, and Boo	✓	✓
Access to Web Data through WWW Functions	✓	✓
Open an URL in the User's Browser	✓	✓
.NET Socket Support	✓	✓
Native Code Plugins Support		✓
Inspector GUI for custom classes	✓	✓
Editor		
Integrated Editor	✓	✓
Instantaneous, Automatic Asset Importing	✓	✓
Integrated Animation Editor	✓	✓
Profiler and GPU profiling3		✓
External Version Control Support	✓	✓
Script Access to Asset Pipeline		✓
Dark Skin		✓

Enemy AI pseudocode examples

```
public class MonsterReturn : MonoBehaviour
{
    Return()
    {
        // Wait _returnDelay to finish
        _returnDelay —= Time.deltaTime;

        // As long as there is no return point. Find one.
        IF returnPointExists EQUALS FALSE
        {
            _ReturnSmoother's position = halfway point between _Waypoint1 and _Waypoint2
            _DistanceToWp1 = Get distance to _Haypoint1
            _DistanceToWp2 = Get distance to _Waypoint2

            // If Waypoint2 is further away.
            IF _DistanceToWp2 > _DistanceToWp1
                // Waypoint2 is the next target
                MovementScript.SetActiveDirection(FALSE)
            ELSE
                // Waypointl is the next target
                MovementScript.SetActiveDirection(TRUE)

            _returnPointExists = TRUE
        }

        // If we are done waiting.
        IF _returnDelay < O
        {
            IF MovementScript>().Direction == TRUE
                Move _ReturnSmoother towards _Waypoint1

            ELSE
                Move _ReturnSmoother towards _Waypoint2

            // Make enemy move torwards return smoother
            _ReturnPoint = _ReturnSmoother's position
            Move towards _ReturnPoint

            // If the enemy has reached the ReturnPoint
            IF position EQUALS _ReturnPosition
            {
                RESET _ReturnDelay
                MonsterAttack().ResetAttackCooldown()
                _returnPointExists = FALSE
                MonsterBehaviour.SetActiveBehaviour(MOVE)
            }
        }
    }
    SetWayPoints(Wp1, Wp2)
```

```
        {
        _Haypoint1 = Wpl;
        _Haypoint2 = Wp2;
        }
}
```

```
public class MonsterReturn : MonoBehaviour
{
      Return()
      {
            // Wait _returnDelay to finish
            _returnDelay —= Time.deltaTime;

            // As long as there is no return point. Find one.
            IF returnPointExists EQUALS FALSE
            {
                  _ReturnSmoother's position = halfway point between
                  _Waypoint1 and _Waypoint2
                  _DistanceToWp1 = Get distance to _Haypoint1
                  _DistanceToWp2 = Get distance to _Waypoint2

                  // If Waypoint2 is further away.
                  IF _DistanceToWp2 > _DistanceToWp1
                        // Waypoint2 is the next target
                        MovementScript.SetActiveDirection(FALSE)
                  ELSE
                        // Waypointl is the next target
                        MovementScript.SetActiveDirection(TRUE)

                  _returnPointExists = TRUE
            }

            // If we are done waiting.
            IF _returnDelay < O
            {
                  IF MovementScript>().Direction == TRUE
                        Move _ReturnSmoother towards _Waypoint1

                  ELSE
                        Move _ReturnSmoother towards _Waypoint2

                  // Make enemy move torwards return smoother
                  _ReturnPoint = _ReturnSmoother's position
                  Move towards _ReturnPoint

                  // If the enemy has reached the ReturnPoint
                  IF position EQUALS _ReturnPosition
                  {
                        RESET _ReturnDelay
```

```
                    MonsterAttack().ResetAttackCooldown()
                    _returnPointExists = FALSE
                    MonsterBehaviour.SetActiveBehaviour(MOVE)
                }
            }
        }
        SetWayPoints(Wp1, Wp2)
        {
        _Haypoint1 = Wpl;
        _Haypoint2 = Wp2;
        }
}
```

```
public class MonsterMovement : MonoBehaviour
{
        Move(Waypoint1,Waypoint2)
        {
            IF _Direction
                MOVE TOWARDS Waypoint1
            IF !_Direction
                MOVE TOWARDS Waypoint2
        }

        // When colliding with wp1 or wp2
        OnCollisionEnter(Collision)
        {
            IF Colliding with wp1
                _Direction = TRUE
            IF Colliding with wp2
                _Direction = FALSE
        }

        SetActiveDirection(Boolean)
        {
            Set _Direction
        }
}
```

```
public class MonsterBehaviour : MonoBehaviour
{
        SetBehaviour(behaviour)
        {
            Set _ActiveBehaviour
        }

// Update behaviour
        Update()
        {
              IF Current behavior EQUALS Move
              {
                    MonsterMovement.Move(Waypoint1,Waypoint2)
                    // Target areas are target objects set in editor
                    MonsterAttack.PrepareAttack(TargetArea1,TargetArea2)
              }

              IF _ ActiveBehaviour == Attack
                    MonsterAttack.Attack()
              IF ActiveBehaviour == Return
                     MonsterReturn.Return()
        }
}
```

Movement script for a 2D platformer

```
public class Movement : MonoBehaviour
{
     public float still = 0.1f;
     public float gravity = 21f;
     public float jumpSpeed = Sf;
     public float moveSpeed = 10f;
     public float maxSpeed = 20f;
     public float fallSpeed;
     public Vector3 vectorMove;
     public CharacterController CharacterController;

     void Awake()
     {
          CharacterController = gameObject.GetComponent
          ("CharacterController") as CharacterController;}
     }

     void Update()
     {
          Controls();
          Jump();
          Move();
     }

     public void Controls()
     {
          fallSpeed = vectorMove.y;
          vectorMove = Vector3.zero;

          if (Input.GetAxis ("Horizontal") > still ll Input.GetAxis ("Horizontal")
               < - still)
             vectorMove += new Vector3 (Input.GetAxis("Horizontal"), 0, 0);
     }

     public void Move()
     {
          vectorMove = transform.TransformDirection(vectorMove);
              vectorMove *= moveSpeed;
          vectorMove = new Vector3(vectorMove.x, fallSpeed,
               vectorMove.z);
          Gravity();
          CharacterController.Move(vectorMove * Time.deltaTime);
     }

```

```
   public void Jump()
    {
        if (Input.GetButton("Jump"))
        {
            if (CharacterController.isGrounded)
                fallSpeed = jumpSpeed;
        }
    }

    public void Gravity()
    {
        if (vectorMove.y > —maxSpeed)
            vectorMove = new Vector3 (vectorMove.x, (vectorMove.y
                gravity * Time.deltaTime), vectorMove.z);
        if (CharacterController.isGrounded && vectorMove.y < -1)
            vectorMove = new Vector3 (vectorMove.x, (-1),
                vectorMove.z);
     }
}
```

Rope climbing pseudocode example

```
public class ClimbRope : MonoBehaviour
{
        FixedUpdate()
        {
            IF keyUP && not lowest rope segment
            {
                  // So we don't fall of the rope
                  Disable gravity for player

                  // Position of the next segment upwards
                  Get the position of the next segment from the current rope
                  and segment

                  // Move player towards the next segment (upwards)
                  Transform players local position towards the next segment
                  with MoveTowards()
            }
            ELSE IF keyUP && not upmost rope segment
            {
                  // So we don't fall of the rope
                  Disable gravity for player

                  // Position of the next segment downwards
                  Get the position of the next segment from the current rope
                  and segment

                  // Move player towards the next segment (downwards)
                  Transform players local position towards the next segment
                  with MoveTowards()
            }
            // Jumping off the rope
            ELSE IF keyRIGHT OR keyLEFT
            {
                  RE-enable player movement script
                  RE-enable player gravity
                  RE-enable player kinematic
            }
            ELSE
                  Keep transform on player position with current segment's
                  position
        }
}
```