jamk.fi

TEACHING UNITY3D IN GAME PROGRAMMING MODULE

Paavo Nelimarkka

Bachelor's Thesis
May 2014

Degree Programme in Media Engineering
School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
NELIMARKKA, Paavo	Bachelor´s Thesis	5.5.2014
	Pages 104	Language English
		Permission for web publication (X)

Title
Teaching Unity3D in Game Programming Module

Degree Programme

Media Engineering

Tutor(s)
NIEMI, Kari

Assigned by
HUOTARI, Jouni. JAMK University of Applied Sciences

Abstract

The objective of this bachelor's thesis was to provide a good source for JAMK University of Applied Sciences to teach the basics of Unity3D game engine in the Game Programming module. The aim is to cover what features and aspects are crucial for beginners and how to organize the course.

The thesis discusses the game industry, game production and game development with game engines. The game engines in educational context and some educational source material are also covered. Experiences from the Game Programming module in spring 2014 were examined and ideas for development of the course were created on that basis.

The main part of the thesis Is the source material created for educational purposes including scheduling of the contact lessons, demonstrative lecturing resources, tutorials and assignments. This part provides all the absolutely crucial aspects, features and functionalities of the Unity3D for the student to create a simple working game.

The conclusion of the thesis includes what the author learned, how the teaching was as an experience and how to develop his skills in future.

Keywords

Game development, Unity3D, educational, C#

Miscellaneous

Appendices 1-11

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

KUVAILULEHTI

Tekijä(t) Nelimarkka, Paavo	Julkaisun laji Opinnäytetyö	Päivämäärä 5.5.2014
	Sivumäärä 104	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
Teaching Unity3D in Game Programming Module

Koulutusohjelma
Mediatekniikka

Työn ohjaaja(t)
NIEMI, Kari

Toimeksiantaja(t)
HUOTARI, Jouni. JAMK University of Applied Sciences

Tiivistelmä

Opinnäytetyön tavoitteena oli tuottaa Jyväskylän Ammattikorkeakoululle hyvä ohjeistus ja tietolähde Game Programming valinnaismoduulin perustaksi. Tarkoituksena oli selvittää mitkä ominaisuudet ja toiminnot ovat kaikkein tärkeimpiä perusteiden opetuksessa ja kuinka opetus tulisi toteuttaa.

Opinnäytetyön selvitysosassa käsitellään pelialaa, pelien tuotantoa, pelien kehittämistä pelimoottorilla, pelimoottoreita opetusympäristössä ja olemassaolevaa opetusta. Työssä selvitetään myös Game Programming valinnaismoduulin historiaa, kevään 2014 toteutusta ja kuinka sitä voidaan kehittää.

Työn pääosa on ohjeistus ja tietolähde Unity3D:n opettamiseen. Ohjeistuksessa käsitellään tuntijakoa, aikataulutusta ja se sisältää myös luentomateriaalia, tutoriaaleja ja tehtäviä. Ohjeistus käsittelee Unityn välttämättömimmät ominaisuudet ja toiminnot joita opiskelija tarvitsee luodakseen yksinkertaisen ja toimivan pelin.

Työn pohdintaosassa käydään läpi mitä kirjoittaja oppi, millainen kokemus pelinkehityksen opettaminen oli ja kuinka kehittää taitoja tulevaisuutta ajatellen.

Avainsanat (asiasanat)
Pelinkehitys, Unity3D, opetusmateriaali, C#

Muut tiedot

Liitteet 1-11

Contents

Figures 3

Terms and abbreviations 6

Preface 7

1 Introduction 8

2 Game industry 10

2.1 Globally 10

2.2 In Finland 11

3 Game development and production 12

3.1 Game production 12

3.2 Game production phases 12

3.3 Roles of game development 14

3.4 Unity3D 15

3.4.1 Introduction 15
3.4.2 Unity3D as a game engine 16
3.4.3 Example usage 17
3.4.4 Unity3D influences in game development 18

4 Teaching game development with Unity3D 21

4.1 Benefits 21

4.2 Education & materials 22

4.2.1 NHTV IGAD 22
4.2.2 Beginning 3D Game Development with Unity 4 22
4.2.3 Holistic Game Development with Unity 24

5 Experiences in the game programming module at JAMK 25

5.1 Short history 25

5.2 Implementation 25

5.2.1 Dividing the content 25
5.2.2 Bordering 27
5.2.3 Lecturing 27
5.2.4 Exercises 27
5.2.5 Online working 28

5.3 Developing the course 28

6 Lessons, tutorials and assignments 29

6.1 Literature and other sources 29

6.2 What the beginner should know 29

6.3 Contact lessons 29

6.4 Assignments 30

6.5 Interface 30

6.5.1 Crucial points 37
6.5.2 Tutorials and assignments 38
6.6 Assets, GameObjects and Components 38

6.6.1 Assets 38
6.6.2 GameObjects 39
6.6.3 Components 40
6.6.4 Prefabs 41
6.6.5 Tutorials and assignments 41
6.6.6 Crucial points 42
6.7 Scripting 42

6.7.1 Languages 42
6.7.2 Game Loop 43
6.7.3 Update() loop 45
6.7.4 FixedUpdate() loop 45
6.7.5 Testing the game loop 46
6.7.6 Time.deltaTime 48
6.7.7 Instantiate 48
6.7.8 Tutorials and assignments 49
6.7.9 Crucial points 49
6.8 Physics 50

6.8.1 Rigidbody 50
6.8.2 Physic materials 51
6.8.3 Colliders 53
6.8.4 Character controller 56
6.8.5 Tutorials and assignments 57
6.8.6 Crucial points 57
6.9 Rendering 57

6.9.1 Camera 57
6.9.2 Skybox 60
6.9.3 Materials and shaders 60
6.9.4 Lights 64
6.9.5 Lightmapping 70
6.9.6 Tutorials and assignments 72
6.9.7 Crucial points 72

7 Conclusion 73

7.1 The results 73
7.2 Teaching as an experience 74
7.3 Developing my skills 74

References 76

APPENDICES 79
Appendix 1 79
Appendix 2 85
Appendix 3 89
Appendix 4 90
Appendix 5 91
Appendix 6 93
Appendix 7 100
Appendix 8 101
Appendix 9 102
Appendix 10 103
Appendix 11 104

Figures

Figure 1 - Gamer statistics from ESA study.

Figure 2 - Chart about the turnout of Finnish game industry in last 5 years.

Figure 3 - Representation of game production cycles.

Figure 4 - Representation of the four main roles of a game development team.

Figure 5 - Unity logo.

Figure 6 - Unity project.

Figure 7 - Heartstone logo.

Figure 8 - NHTV logo.

Figure 9 - Beginning 3D Game Development with Unity 4.

Figure 10 - Holistic Game Development with Unity.

Figure 11 – Unity’s default interface.

Figure 12 – Top toolbar.

Figure 13 – GameObject manipulation tools

Figure 14 – Playback buttons

Figure 15 – Layer and Layout menus.

Figure 16 – Hierarchy view.

Figure 17 – Scene view.

Figure 18 – Inspector view.

Figure 19 – Project view.

Figure 20 – Project view in an empty project.

Figure 21 – Game project directory

Figure 22 – Hierarchy view folder structure.

Figure 23 – Transform component.

Figure 24 – Components.

Figure 25 – Unity’s main game loop.

Figure 26 – The Update() loop.

Figure 27 – The FixedUpdate() loop.

Figure 28 – ExecutionOrder.cs

Figure 29 – Execution order in console.

Figure 30 – Time.deltaTime in use.

Figure 31 – Instantiation of a GameObject.

Figure 32 – Rigbody component.

Figure 33 – Physic material.

Figure 34 – Box collider.

Figure 35 – Sphere collider.

Figure 36 – Capsule collider.

Figure 37 – Mesh collider.

Figure 38 – Character controller component.

Figure 39 – Camera component.

Figure 40 – Skybox.

Figure 41 – Vertex Lit.

Figure 42 – Diffuse shader.

Figure 43 – Specular shader.

Figure 44 – Bumped Diffuse shader.

Figure 45 – Bumped Specular shader.

Figure 46 – Directional light.

Figure 47 – Directional light properties.

Figure 48 – Point light.

Figure 49 – Point light properties.

Figure 50 – Spot light.

Figure 51 – Spot light properties.

Figure 52 – Bake tab.

Figure 53 – Baked scene.

Terms and abbreviations

Alpha, Beta = Phases of game publishing. Developers can publish the game unfinished to a certain group of people before publishing the retail version of the game.

Bug = Bug is an error or a fault on the programming code which usually produces unexpected and unwanted results when the program is in run

ECTs = Standard for comparing study attainment or performance

IGAD = NHTV International Game Architecture and Design, game development oriented school in Breda, Netherlands.

Indie = Used to describe the process of making games with small teams and without a notable financial support

JAMK = University of Applied Sciences in Jyväskylä

JavaScript, C# and Boo = Programming languages

NHTV = Breda University of Applied Sciences, an university in Breda, Netherlands

Unity = Unity3D game engine

Preface

I have been playing and making games since I was a child. The first games I made were text-based adventure games I made with HTML or with QBASIC. Soon after that I got familiar with Game Maker game development platform where I also had my first collaborative projects with my friend. After that I have been involved in various game projects.

I studied in Breda, Netherlands where I completed my exchange term studying in NHTV International Game Architecture and Design, which is a school based on game development, graphics, programming and production. I got a really good basic understanding of the Unity3D game engine there.

There we had a big half year game project in a group of 8 students where we made a game called Z-Gee. It was a zero gravity adventure/puzzle game and we got positive feedback and a good grade for it. My role there was to be the lead level designer, and I also got to test and polish the metrics and mechanics of the game. This experience might have been a kick starter for my actual game development career.

After I got home from Netherlands, Jouni Huotari, principal lecturer at JAMK University of Applied Sciences, asked me to tell about my experiences over there and if I was interested to help in the Game Programming module next spring.

Now it is spring 2014 and I have been teaching Unity3D in Game Programming module here at JAMK University of Applied Sciences. This has been a completely new experience for me since I have never taught anything.

I’ve now got more involved with the core points and features of Unity3D. But what features should beginners know? How should they be taught? At this point it was clear that I wanted to write my bachelor’s thesis about this subject.

1 Introduction

This bachelor's thesis discusses Unity3D game engine in game development and in teaching environment and the teaching and the implementation of the Game Programming module in JAMK University of Applied Sciences.

Unity3D is a game engine which is greatly used now, especially in 'Indie' game development since it can be used cheaply or even free. Despite being made for game development, it can be used for many other purposes as well, e.g. for example visualizing architecture in 3D environment, using Oculus Rift (virtual reality glasses) to experience an artificial 3D reality.

It is also an ideal engine to teach game development process since it speeds it up greatly and does not require a team with so many programmers. This is a way for a team can get familiar with different roles and processes of a typical game project.

JAMK has an optional course module called Game Programming which is lectured in IT-Dynamo at Piippukatu 2. The course aims to teach students the basics of game programming and it is worth of 15 ECT. This course has lately gone through some major changes, which are covered in this thesis. One of these changes is including the Unity3D in the course plan.

Jouni Huotari, a principal lecturer represents JAMK University of Applied Sciences and is the issuer of this bachelor's thesis. He is the organizer of the Game Programming module at JAMK. He set a goal that this bachelor's thesis would benefit JAMK in future in developing the Game Programming module and especially in teaching the Unity3D.

This bachelor's thesis discusses the Unity3D in game development, what beginners should know about it and how it could be taught. There are also chapters about game programming in general, the Game Programming module at JAMK University of Applied Sciences, its development opportunities and experiences in teaching Unity there.

The goal was to have a solid source of knowledge and experiences for JAMK to base the Game Programming module in future, for the author of the thesis to market his skills, to benefit him on his career and all the other thinkable purposes.

2 Game industry

2.1 Globally

Game Industry has been the fastest growing branch of entertainment industry through 21st century. Game industry is already bigger than music industry, and it might be catching movie industry soon. (Industry Info 2014)

This year the global market revenues are estimated to grow to 70.4 billion dollars which represent an increase of 6%. This year the number of gamers is estimated to go over 1.2 billion. Smart phone and tablet games grow 35% this year easily outpacing any other studied segment. (Global Games Market Report Infographics 2013)

The Entertainment Software Association published a study they carried out on people of the United States about their relationship with computer and video games. Some findings can be seen below (Figure 1).

Figure 1. Gamer statistics from ESA study

(Essential facts about the computer and video game industry 2013)

IGDA, International Game Developers Association is the largest non-profit membership organization for game developers worldwide. It is a network for collaborative projects and communities the main goals of which are to develop game industry, advance the careers and to enhance the lives of game developers. (International Game Developers Association website)

2.2 In Finland

The game industry in Finland has been growing a great deal from year 2000 to this day. Because of the small market here in Finland, games have become a great export for Finland. (Industry Info 2014)

As can be seen in the chart below, the turnover for Finnish game industry has more than doubled in one single year. (Figure 2)

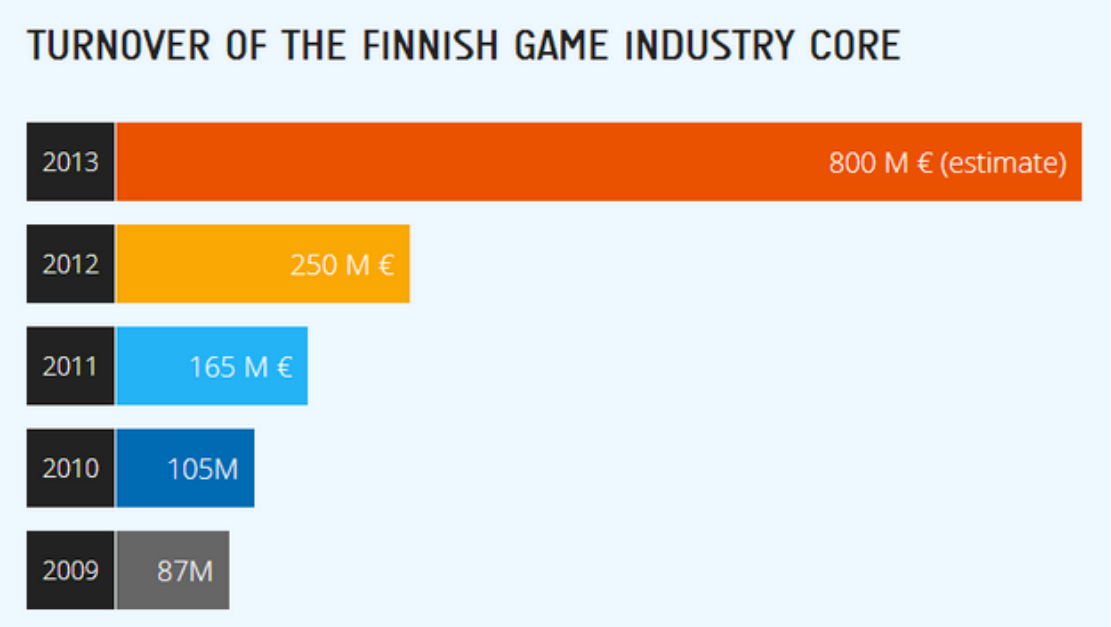


Figure 2. Turnout of Finnish game industry in last 5 years. (Industry Info 2014)

In 2012 there were approximately 150 companies producing games in Finland. The best known companies from Finland are Rovio (Angry Birds) and Supercell (Clash of Clans). The total value of Finnish game industry in 2011 was 270 million euros (344 million dollars). Compound annual growth rate is 25.7% which is significantly faster than the global game market. If the growth continues this way, it is estimated that the turnover of Finnish game industry in 2020 will be 1.49 billion euros.

In 2012 there were more than 1,500 employees in the game industry in Finland and the average salary in 2012 was 3590 euros. (The Game Industry in Finland 2012)

3 Game development and production

3.1 Game production

Game development is a process where a group of various assets is crafted to a playable game. These assets can be code, graphics, audio, script, etc. Because of the amount of assets and the work the game usually needs, games are not developed alone but in groups. This group can be a small group of indie developers, a massive game company or anything between.

In addition to the traditional game development, there is a phenomenon called indie game movement. Indie is a term to describe a game development process executed with a small team without any notable funding. Usually indie developers get their funding via crowd funding.

3.2 Game production phases

Game production process itself can be divided into four phases: pre-production, production, testing and post-production. Some games have multiple production cycles instead of one. (Chandler 2014. 4)

The figure below represents the cycles and the phases of game production (Figure 3).

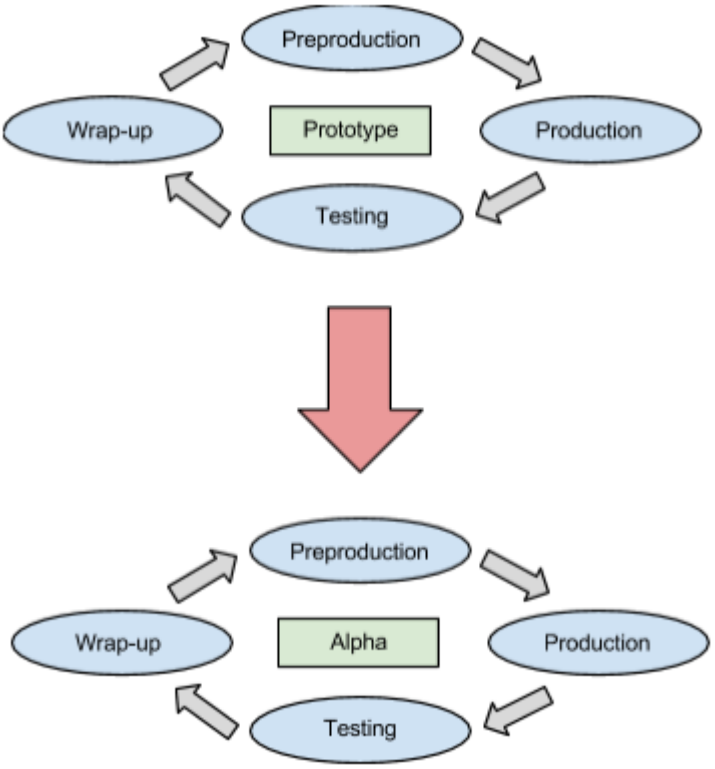


Figure 3. Representation of game production cycles (Chandler 2014. 4.)

Pre-production is a critical phase which defines what the game is, how long it will take to make it, how many employees will be needed and the costs. As a rule of thumb, pre-production should require 10-25% of the total development time. Pre-production can also be broken down to a concept, game requirements and a game plan, which is a roadmap for finishing the game. (Chandler 2014. 5-6)

In the production phase the team can begin work on making the assets and code for the game. Although the production should go smoothly based on the planning in pre-production phase, there are usually some new features and assets that need to be added which were not planned. The main focus here is on content, code, tracking progress and completing tasks. (Chandler 2014. 9-10)

Testing phase is a phase where a game will be fully tested so that it works completely and without bugs. The testing phase can already be started in the

production phase, after a feature or asset is published in the build. In a beta phase, developers will be fixing the bugs found while testing. (Chandler 2014. 12-13)

The post-production is also called a wrap-up of the game. This is where the team can make points and notes for the projects in future and review the pros and cons of the project. The phase usually consists of learning from experience and archiving the plan. (Chandler 2014. 15)

3.3 Roles of game development

In typical game development a group or team is divided into various roles (Figure 4). There are numerous different possible roles depending on the project itself, however, the most usual roles can be placed under managers, designers, programmers and artists.

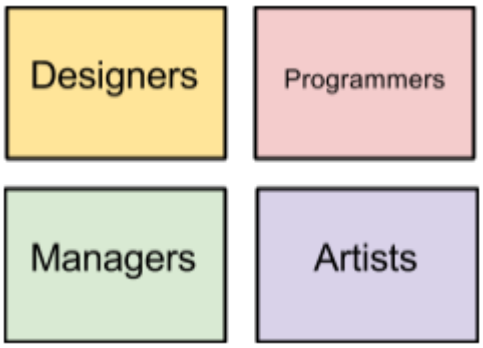


Figure 4. Representation of four main roles of a game development team.

Lead designer comes up with the game idea and develops it with other designers towards a full game by designing the mechanics, content, visual style etc. After the prototype they work closely with artists and programmers. There can be many types of designers, e.g. a level designer, game mechanic designer and visual art style designer.

Managers manage teams and the project itself. Project management in game development can be similar to actual software development processes. Their

job is to listen to the team, give tasks for them and report what has been accomplished. Producers can be listed in this group.

Programmers program the game by forming game mechanics and interaction with all the assets. The first task for them to do is to take the game design document and do the prototype of the game. After the prototype they listen to the designers and develop the game further.

Artists make game assets like graphics, music, scripts, sound effects, 3D models etc. Artists work closely with designers so they can produce assets which go along with the designed style.

3.4 Unity3D

3.4.1 Introduction

"Unity is a game development ecosystem: a powerful rendering engine fully integrated with a complete set of intuitive tools and rapid workflows to create interactive 3D and 2D content; easy multiplatform publishing; thousands of quality, ready-made assets in the Asset Store and a knowledge-sharing community." (Unity3d.com website 2014)

Figure 5. Unity logo (Unity Logo 2014)

At this point Unity3D has over 2.5 million users worldwide. Unity has an office here in Finland and it is located in Helsinki. (Public relations)

Unity has two main versions: Unity Free and Unity Pro. Unity Free is free to use as long as a company does not exceed 100,000$ annual gross revenues or budget. Unity Pro costs 1500$ and has more features than Free. Pro license is for one user only. There are also some add-ons for different

platforms which may cost extra. If more users are wanted, a team license can be purchased, which is cheaper than buying many single-user licenses. (Unity3D License Agreement 2013)

Unity has some competitors, for example Unreal Engine and CryEngine.

3.4.2 Unity3D as a game engine

As a game engine Unity3D is somewhat different from other game engines. It has a really simple user interface which is easy to get to know even for a complete beginner. It has many integrated tools so developers might not need that many different programs.

Here is a screenshot of a Unity project (Figure 6).

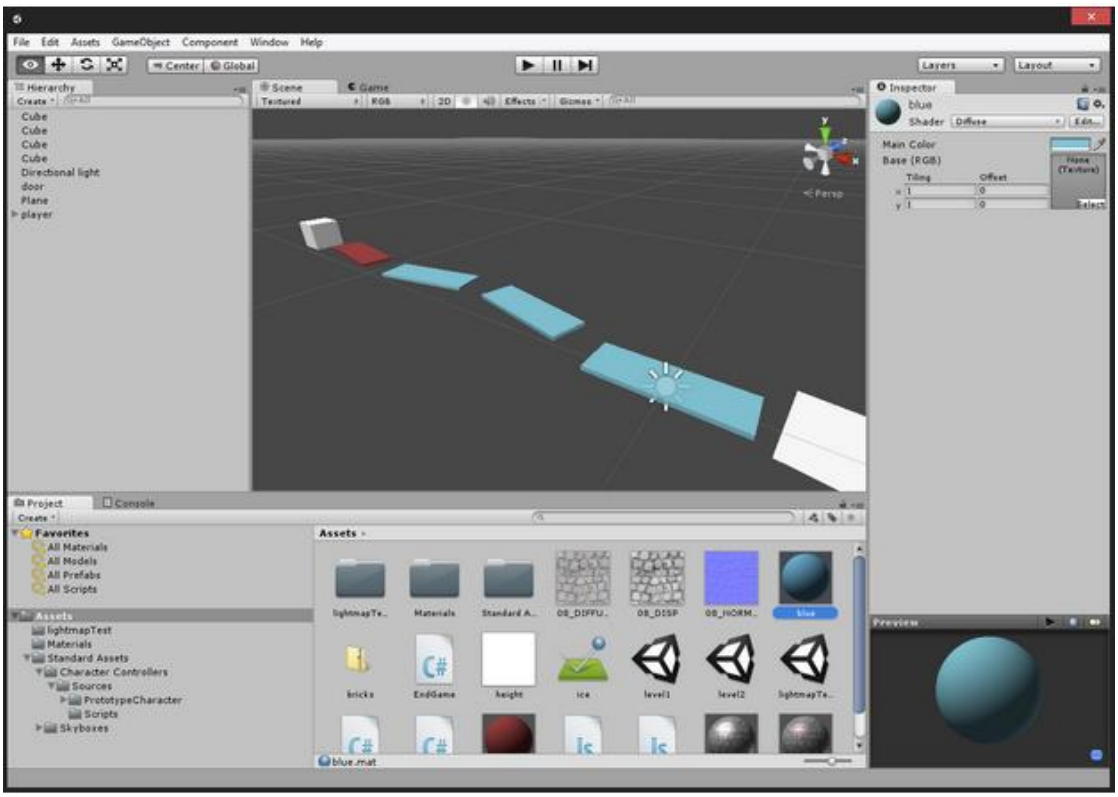


Figure 6. Unity project

Unity has an asset store, a place where the community can publish their assets, plugins or whatever game developers might need. This is a great way to outsource workload depending on your team.

Scripting in Unity is uses JavaScript, C# or Boo for scripting and it comes with an integrated script editor called MonoDevelop; also user favourite editor can be used. It is advised to use C# because of its object-oriented nature. There are also some visual scripting tools in the asset store for those who really do not want to write any code.

Unity Pro also has an extensive support for native code plugins, thus parts of the project can be written using C++, Objective C, etc. This can be used to extend the game mechanics or to extend Unity's features.

For 3D graphics a 3D modeling software is needed. Unity can import some formats, however, other formats might need the modeling software to be installed to get the models imported. Unity can usually import more information from the files than just the mesh, e.g. height maps, animations, and textures can be imported.

Unity has a tool called Mecanim for making and adjusting animations for 3D models also in Unity. In addition Unity has a tool called Shuriken for creating diverse particle effects using colors, shapes, meshes etc. For terrains Unity has its own terrain editor.

Unity has great documentation on their website and the community forums are a useful place to get help and suggestions from other developers. Unity also provides a set of really useful video tutorials on their website with script references for every script language listed above.

3.4.3 Example usage

Unity can be used for many different styles of games, or even for things which are not game related. With Unity both 2D and 3D games can be developed, although it originally was meant for 3D games only.

One issue where Unity3D performs really well is multiplatform publishing. At this point Unity can publish games to:

- Windows
- Mac

- Linux
- Android
- iOS
- Windows Phone 8
- Blackberry
- PS3
- Xbox360
- Wii
- Windows Store Apps
- Unity web player

(Multiplatform)

Unity is as good in over-network multiplayer games as it is in single player games. Unity script library has an integrated server class for simple network server creation.

Unity can also be used to make website plugins to visualize 3D-models. If there is for example a web shop and a 3D model of a product, Unity web player plugin can be published for the customer to view the product in 3D space. Unity is also used in industrial visualizing of architecture and mechanical objects.

Visually Unity is not as good as some of its competitors. When developing a game with as good graphics and visuals as possible, some of Unity's competing engines might perform better.

3.4.4 Unity3D influences in game development

"Unity is a great choice for small studios and indie developers and with its large user base and community it allows everyone from newbies to seasoned developers to get help and share information quickly", says Sue Blackman in her book. (Blackman, 2013. xxvi)

Unity with other easy low-budget game developing solutions has had a great effect on game development. These entry-level engines have made it possible to develop games in smaller teams with much smaller budgets.

In the big game companies there can be enough employees so they can make their own game engine and produce almost everything they need for their games, but in smaller groups this is not possible. This is why Unity has gained such great popularity among the Indie developers.

With Unity time, workload and expenses can be saved compared to making the engine alone.

Among the Indie developers, it is very common to use as much free and open source assets as is possible and to outsource as much work as can be outsourced. Unity community helps here. There is much free or cheap content on the asset store. If there is a problem the solution can probably be found from Unity community forums or documentation.

PlayRaven is a good example of one of these groups. PlayRaven is a freshly founded game company with five founding members. They pursue to outsource as much as they can. If the team has, for example a graphic designer and he/she has not any work to do in the early stages of development and still has to paid that is wasted money. If graphics are outsourced, the developers can concentrate on core mechanics for example and buy the graphics just when they are needed. (Seppänen, 2014)

Unity is one of these free or cheap to use game development platforms which provide users with a good set of tools, great community and loads of assets to use allowing these small Indie teams to exist in the first place. The easy user interface and great community is why Unity is very popular among the game development beginners.

Heartstone (Figure 7) is Blizzard’s new cross-platform card game which is developed with Unity. (Wawro, 2014)

Figure 7. Heartstone logo. (Heartstone Logo 2013)

Blizzard chose Unity for development of their new multiplatform card game called Heartstone. Some of the reasons why they chose Unity was:

- Good for a small team
- Established toolset
- Easy multiplatform debugging, less duplicate work
- Multiplatform flexibility for future

(Wawro, 2014)

4 Teaching game development with Unity3D

4.1 Benefits

Game developing is not all about programming. When courses about game development are designed a particular set of constraints is needed and one of them is time. For example, in the Game Programming course module there is a package of 15 ECTS credits, which means 375-450 hours of work. In this time students are given a basic understanding of game development as a business, some programming, game-based math, and of course, a game project.

If the students had made their own game engine in this distribution of course subjects, there could have been no way students could have had a real game project, which is where Unity becomes crucial.

Instead of making students to program a game engine, they can be offered a real game project experience. Students can take actual roles in game development, designer, manager, programmer, artist to name a few. This gives students a real experience of game developing instead of just programming a game engine. Not everyone is a programmer or wants to be one.

Another good feature within Unity and other easy-to-use game engines is that when the process is eased up the developers can concentrate on what matters the most, the game itself. Making a game is a complex process which requires good ideas and design. The game should be fun. With Unity or other easy-to-use platforms there is no need to waste that much time on programming problems or on a massive set of unattached tools.

This kind of approach is also more realistic when thinking of the industry. Not everyone will be a part of a big game company. Most people will work with smaller companies and others will start their own companies.

4.2 Education & materials

4.2.1 NHTV IGAD

NHTV Breda University of Applied Sciences is a university in Netherlands with a programme called International Game Architecture & Design. (Figure 8 represents the NHTV's logo)

Figure 8. NHTV logo (NHTV website 2014)

In this programme students can specialize themselves in Game Design & Production, 3D Art, Programming or Indie Game Development. (International Game Architecture and Design programme brochure, 2013. 4)

Unity is taught by a lecturer, Jeremiah van Oosten. Students have a three-hour lecture once or twice a week. After the lecture the students are expected to do a tutorial related to the lecture as a homework. The lectures are completely synchronized with the Game Lab project, so when the students learn something in Unity class they can apply this information immediately to their projects. There is also a test at the end of the course.

4.2.2 Beginning 3D Game Development with Unity 4

Beginning 3D Game Development with Unity 4 is an educational book about Unity game development. Sue Blackman is the author of the book and she

works in southern California. She is a 3D artist and interactive applications author and instructor. (Blackman 2013. xix)

The cover of the Beginning 3D Game Development with Unity 4 (Figure 9).

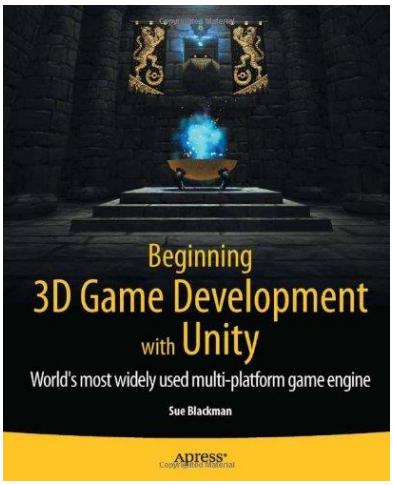


Figure 9. Beginning 3D Game Development with Unity 4 (unity3Dbooks.com website 2014)

In the beginning of the book there is an introduction to game development. However, the rest of the book is one large tutorial which guides the readers through Unity's features by helping them to develop a full playable adventure game. The book introduces all main features of Unity to the readers and with these skills they are able to create various styles of games, not just an adventure game.

Because of the tutorial-oriented nature of the book, finding specific information can be challenging. There could also have been a chapter about Unity itself. The author uses JavaScript language, which might be a friendlier language for beginners; however, it lacks the true object-oriented nature. Then again, scripting in Unity is usually more logic than syntax-based.

This book could be recommended to a person who knows what Unity is and wants to learn the core features Unity by himself. It might not be a good book for a person who is an advanced level user of Unity and already knows about game development and production.

4.2.3 Holistic Game Development with Unity

This book is divided into 8 chapters. The first one is theory of art and programming, and the rest of the chapters deal with the main mechanics of a complete game product. In every mechanic field there is an introduction to Unity's features concerning the subject, and some tutorials and practices. The author uses JavaScript in her examples.

The cover of Holistic Game Development with Unity (Figure 10).

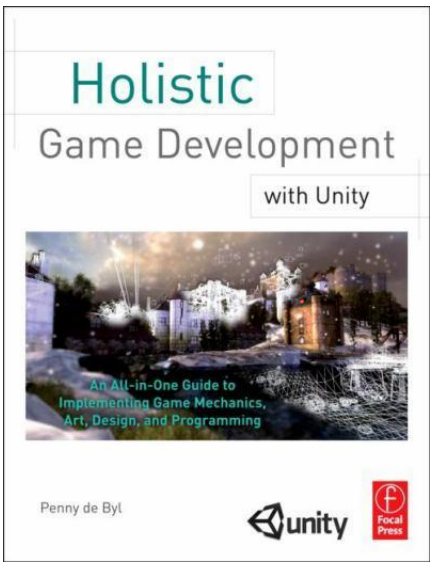


Figure 10. Holistic Game Development with Unity (unity3Dbooks.com website 2014)

This is an educational book about Unity game development. The author is called Penny de Byl. This book is information based and has a lot of smaller tutorials to present the mechanics and introduce readers to Unity's tools.

This book does not cover game development or production at all. It is more based on the tools of Unity. Compared to the Beginning 3D Game Development with Unity 4 book, this book might be more in-depth about how Unity works as a program.

5 Experiences in the game programming module at JAMK

5.1 Short history

Game Programming module is an optional course module in JAMK University of Applied Sciences. It has a history of being more programming oriented where the students usually made their own game engine with raw C++.

Jani Immonen is the CTO of Star-Arcade and has been here teaching C++ game engine and OpenGL programming. He has been teaching the game programming module by himself previously.

Spring 2013 was the first year of Unity being taught in JAMK. Viope was the main learning source and service provider for the Unity learning.

5.2 Implementation

It was decided to leave Viopes service out of the course content since the experience with it last year was not that good and we found it to be a bit expensive for the content it provided. Here some money was saved to spend for example Unity Pro license. One of the new sources of learning material and tutorials was 3DGep which is a blog of lecturer Jeremiah van Oosten from NHTV IGAD. (Jeremiah van Oosten 2014)

This blog provides lecturing material and some tutorials which are free to use. There are some facts on the blog which are outdated, however, mostly it is still usable and beneficial. There are also few pro license features.

5.2.1 Dividing the content

The content did not have to be divided since it was already divided into the 3Dgep blog. Five most critical subjects were chosen which had to be taught well and for which a full lesson was reserved. These five chapters will provide everything the student needs to know to create a simple game.

Those five subjects are:

- Introduction to Unity
- Assets, GameObjects and Components
- Scripting in Unity
- Physics in Unity
- Rendering, Shaders and Lightmapping

Introduction to Unity was a lesson about Unity’s interface. How is the layout divided and what do the different parts of the layout display do? Where does the user find the most used features and how to navigate in the project folders?

Assets, GameObjects and Components is a lesson where these essential features are introduced. What are they and how they can be used to achieve functionality in the project?

Scripting in Unity was a lesson dedicated to programming. The students were expected to have at least some experience in object oriented programming, so it was possible to go straight in to the scripting in Unity.

Physics in Unity was a lesson dedicated to Unity’s physic engine called Rigidbody. Students got to make objects with physic engine component attached to it and to modify its values from the Unity user interface and from code.

Rendering, Shaders and Lightmapping was a lesson where Unity’s visual rendering features were taught. Students got to try different shaders and to see the effect on a 3D model. They also got to learn the difference between dynamic lighting and baked lighting.

5.2.2 Bordering

All the previously mentioned five subjects are considered to be very crucial when developing a simple game and that is why they are emphasized. These topics concern features that are crucial if not mandatory in almost every imaginable game project.

Shuriken particle effect tool, terrain editor, GUI elements and networking were subjects which were left to be dealt with only if needed. There was a small tutorial for Shuriken and GUI but there was no need for terrain editor or networking.

There could also have been a lesson about the built in 2D engine and components meant especially for a 2D project.

5.2.3 Lecturing

Lessons were organized to have two parts: The lecture part and the tutorial part. In the lecture part the subjects were lectured and demonstrated with the projector. The lecture part usually took almost half of the lesson.

After the lecture part and the coffee break there was the tutorial part where the students got to do a one to a few tutorials following teacher's example from the screen. After these tutorials there was usually time to make exercises.

5.2.4 Exercises

After the lecture and the tutorial part the students could stay in the class and do exercises or they could leave and do them at home also. An example exercise can be found in Appendix 1.

There were usually one to few exercises after the tutorials. All of the exercises were not mandatory. There were three exercises which were mandatory and had to be returned to the course's online learning environment called Optima. The students had to pass all these three exercises in order to pass the Unity part of the course.

These exercises had questions about a subjects of Unity and students were asked to find information and summarize the subject. Questions concerned features considered to be crucial. These exercises can be seen in appendices 4, 7 and 11.

5.2.5 Online working

Optima was used as the main working environment where the students were supposed to return their exercises and where they could find all the information they needed.

5.3 Developing the course

Some negative feedback was given concerning the authors skills as a teacher, which could be somewhat expected and the students feedback will help JAMK to develop the course further.

The course can be developed in many ways; however, few points stand out in the feedback.

Teaching should be more tutorial and practice oriented than lecturing. 3DGep is also a good base source but it is not enough. More tutorials and exercises are needed and at least some of them should be more advanced for those students already familiar with Unity.

Some of the guest lecturers could have been more beneficial with better timing considering the game projects.

Changing from JavaScript to C# on the go was somewhat challenging. Luckily the scripting is more logic than syntax based so it was not that hard to translate the scripts from JavaScript to C#. The course could also be developed by getting some good C# scripting reference material.

There could also be a lesson concerning more advanced scripting. An experienced guest lecturer could be a good choice.

6 Lessons, tutorials and assignments

6.1 Literature and other sources

As mentioned earlier, the main source for learning material was the 3DGep blog written by lecturer Jeremiah van Oosten from NHTV. There was material for lecturing and few tutorials per subject. (Jeremiah Van Oosten)

Unity3D also has plenty of video tutorials on their website and they were also used to create tutorials and exercises. Unity3D Script Reference pages were used additionally for a great deal of the scripting lessons. (Unity website 2014)

The two books mentioned earlier were also used: Beginning 3D Game Development with Unity 4 by Sue Blackman and Holistic Game Development with Unity by Penny de Byl. They were not used that much since they both used JavaScript as the scripting language.

6.2 What the beginner should know

All five chapters listed earlier contain almost everything that is absolutely crucial and needs to be taught to a beginner. With the features, components and mechanics described in these chapters the student is able to develop a game.

These five main subjects are covered in this chapter.

Particle effects, terrains, networking etc. are issues which should be taught after these main chapters if there is time and need for it.

6.3 Contact lessons

The author suggests that each one of the following chapters is to be covered in one contact lesson. This way the basics would be covered in five contact lessons which can be implemented intensively in one week or can be

distributed for longer time period. One contact lesson could be implemented as follows:

Contact lesson = 4*45min

- 2*45min of orientating demonstration lecture about the subject and tutorial done together
- 15min Coffee break
- 2*45min of independent work on tutorials and assignments

This model was used in the module this spring and it worked fine.

The next chapters of the thesis include the orientating and demonstrative part of the lesson and the tutorials and assignments related to the subject are included as appendices which will be referred in the end.

6.4 Assignments

Students should be given some assignments for the second part of the contact lesson which they can do individually. These can also be done at home. Tutorials are good exercises; however, also essay-type written assignments about the most crucial features should be given.

There are some example tutorials and assignments included in the appendices. Appendices 4, 7, 9 and 11 give more detailed examples about the mandatory written assignments.

6.5 Interface

Unity is known for having a simple and beginner friendly user interface. Some of the tools and features can be similar as in 3D modeling software. Like in many other software the same action can be done or a same feature found in many different places on the interface, thus it can be used as found best.

First, the student must get familiar with the user interface and to be able to find a tool or feature he/she wants to use. The author breaks the user interface into smaller sections now and describes their most used features.

Here is what the default layout of Unity's interface looks like (Figure 11).

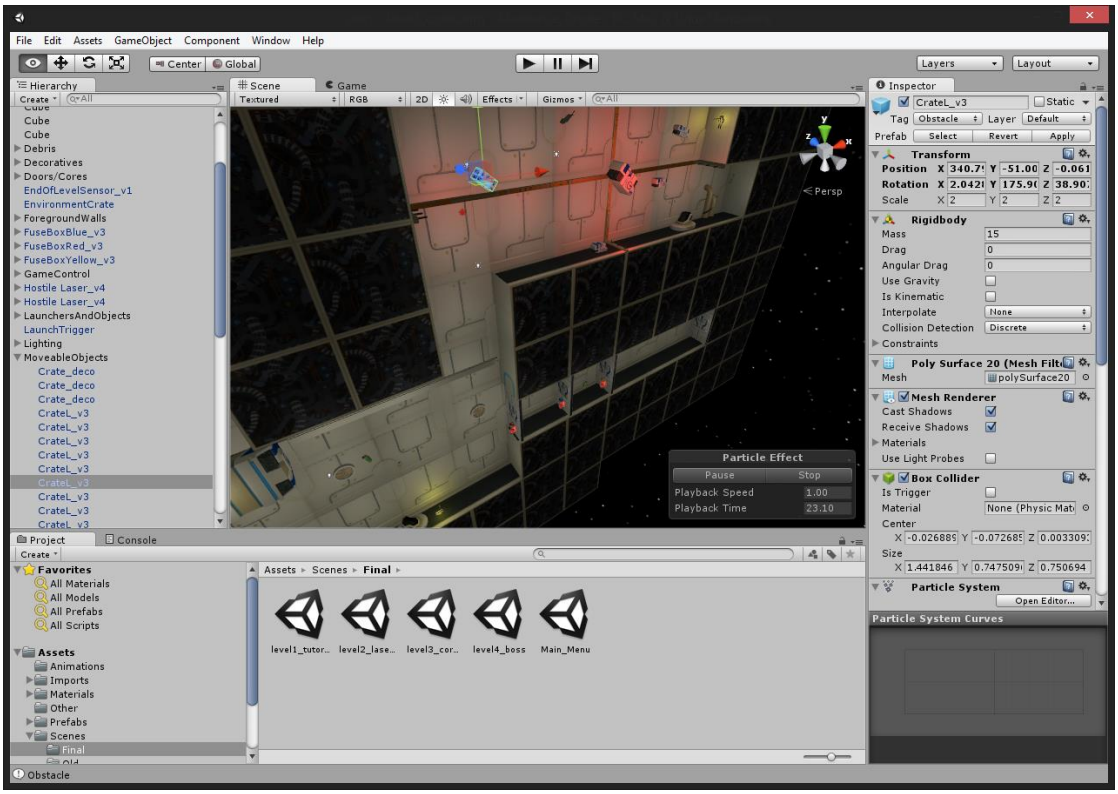


Figure 11. Unity's default interface.

In the top row there is the usual top toolbar which has some familiar looking menus like 'File', 'Edit' and 'Help' (Figure 12).

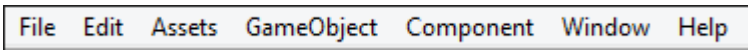


Figure 12. Top toolbar.

New, save or open scene/project can be found from 'File' menu**.** In Unity there can be one project and one scene opened at the time. Scenes are like levels or rooms in the project so the user can have many scenes in the project but only one opened in the scene view at the time. The build feature and the build settings can also be found from this menu. In the 'Edit' menu there are the

typical copy, paste, cut, duplicate and delete features familiar from many other software. ‘Frame selected’, ‘Lock view to selected’, ‘Find’ and ‘Select all’ are used to find and handle objects in the scene view. There are also playback actions and different kind of settings.

From the ‘Assets’ dropdown menu the user can create or manage assets or create and import packages. Packages are sort of collection of assets. Unity comes with some packages on its own, but packages can also be made or downloaded from the ‘Asset Store’.

The ‘GameObject’ and the ‘Components’ menu contains actions related to GameObjects and Components.

In the ‘Window’ menu there are all the possible windows and tools in Unity interface. Unity interface can be modified from the ‘Layout’ dropdown menu.

In the ‘Help’ menu there are the typical help features. They mostly take the user to their documentation in their website. There are also information about the Unity installation, users Unity license and bug report feature.

The second row of the interface (Figure 13) contains tools related to the active scene the user is working on.

Figure 13. GameObject manipulation tools.

This set of tools contain actions to manipulate GameObjects position, rotation and scale. The hotkeys for these actions are Q, W, E and R. The student should get used to the hot-keys, since it is much faster that way.

The first button with the eye icon (Q) is the view mode. With this it is possible to move in the scene and select objects safely without moving, rotating or scaling them.

The second button (W) is the position tool. With this tool objects in the scene can be moved in X, Y or Z axis. GameObjects can be moved a certain amount

by adjusting the snap settings from the ‘Edit’ menu. When the user have determined the distance, the object can be moved in steps by holding the CTRL button when moving an object.

The third button (E) is the rotation tool. When it is selected the GameObject can be rotated by X, Y and Z axis.

The fourth button (R) is the scale tool. When it is selected the user can scale the object in X, Y, Z or in all axes.

Position, rotation and scale factors can always be adjusted precisely from the ‘Inspector view’.

The ‘Center’ button is to choose if the object is manipulated from the center point or from its pivot point. From the ‘Global’ button the user can choose if user wants to see the axes from the scenes viewpoint or from the GameObjects viewpoint. This is very useful if there are objects which are rotated.

In the middle of the second row there are these tools (Figure 14).

Figure 14. Playback buttons.

These tools are the typical playback tools. The first button plays the game in the game window, and by pressing it again it stops the game. The second button pauses the game, and variables and other things can be adjusted on the go. When the game is stopped these variables will be set back. The last button is the step button. The game can be iterated by one physic step at the time.

The last part of the second row looks like this (Figure 15).

Figure 15. Layer and layout menus.

The ‘Layers’ dropdown menu contains the layers of the scene. Scene can be divided into different layers and objects can be assigned per layer. The user can freeze or hide a layer. New layers can be made and old ones be deleted from the ‘Edit layers’ window.

The ‘Layout’ menu contains the interface layouts. The user can choose few predefined layouts or make a new layout.

The next part of the interface is called the ‘Hierarchy view’ (Figure 16).

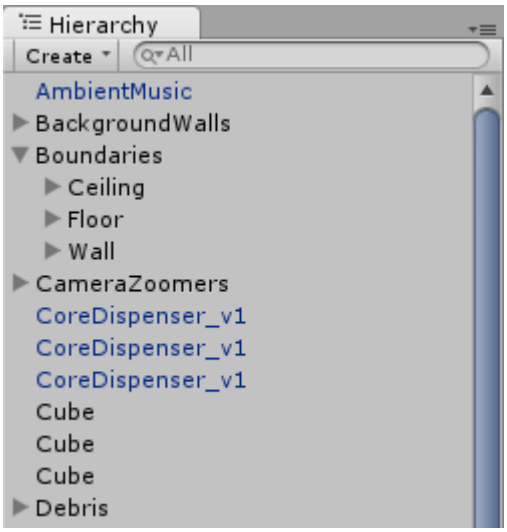


Figure 16. Hierarchy view.

The Hierarchy view displays all the GameObjects in the scene. In this view the user can see the structure of the GameObject and its contents allowing him/her to select only certain parts of the GameObject for example.

There is this ‘Create’ button which allows the user to create GameObjects in top left corner. It is almost the same as the GameObject menu in the top bar, although it is not possible to make an empty GameObject from there.

In top right corner there is a small menu from where the user can close the window, maximize it or add a tab.

There is also a search area which allows searching GameObjects from the scene.

The next part in the middle of the layout is called the ‘Scene view’ (Figure 17).

Figure 17. Scene view

Scenes are the levels/rooms of the project. There can be only one scene open per time. The Scene view displays the scene and its GameObjects visually.

The user can navigate in the scene in many ways. By pressing the right button of the mouse the user can move in the scene with W, A, S and D keys like in many first person video games. By pressing shift here it will move faster. If the user wants to find an object from the scene he/she can select the object in the hierarchy view and press F. It is also possible to zoom with the mouse scroll.

By running the game it will be displayed on scene view also, but in a Game tab.

The first dropdown menu on the left is called the draw mode. It is possible to change the way scene is displayed on the scene view from here. There are options like Textured or Wireframe draw mode for example.

The next menu is called the Render Mode. There are some useful options how to render the scene in the scene view.

Next there are buttons to change to the 2D view, to disable lighting and to disable audio. The effect menu allows the user to disable certain effects from the scene view.

The next part is called the ‘Inspector view’ (Figure 18).

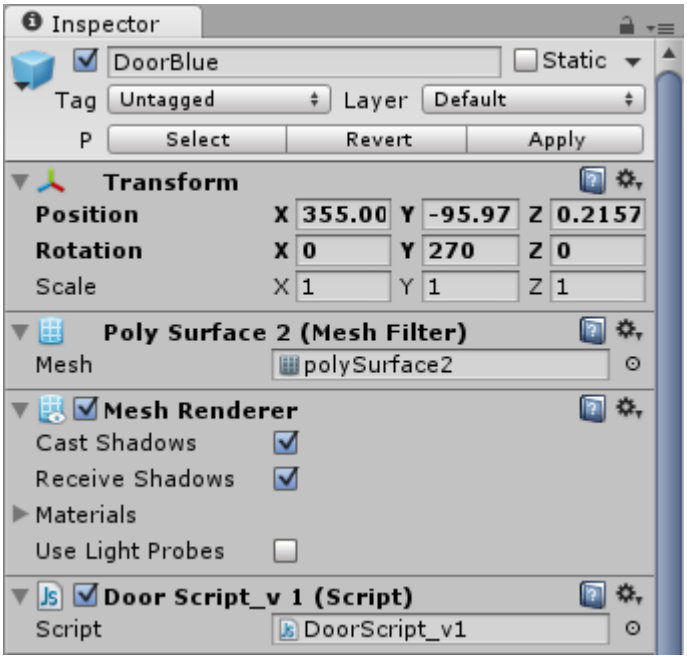


Figure 18. The Inspector view

Inspector view lets the user to view all the components and their attributes of a GameObject. From here it is possible to add or delete components and modify all the values and variables.

The first area in the top is the name of the GameObject.

Below is the ‘Tag’ and the ‘Layer’ dropdown menus. GameObjects can be given a tag which helps to refer to the object or all the objects which have the same tag. The user can also assign a GameObject to a certain layer from here.

Below are all the components and their attributes and values. From here the user can add, delete or disable components. The user can also arrange the components so the most modified component can moved up and can be modified faster. Components values can be reset from the top right corner of the component. There is also the book icon which is a link to component's reference documentation on Unity's website.

In the bottom there is a preview image of the GameObject.

The next section is called 'Project view' (Figure 19).

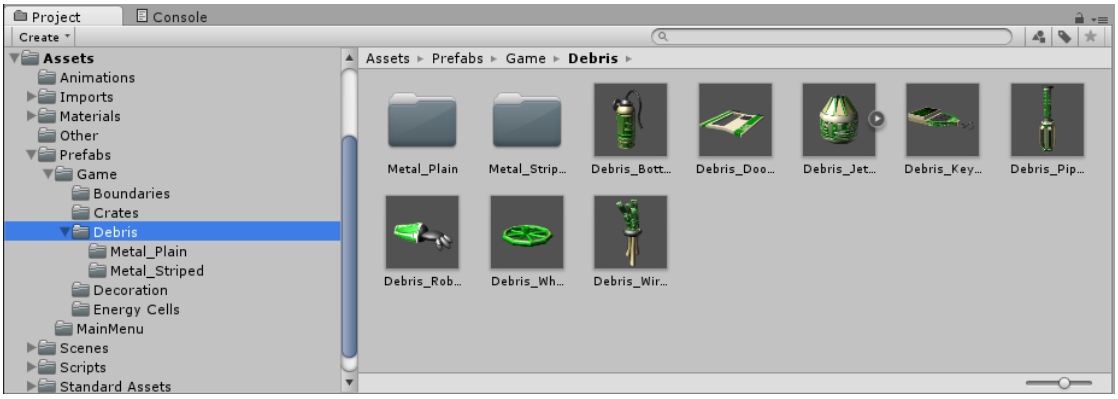


Figure 19. The Project view.

The Project view is a file manager which contains the 'Assets' folder of the actual game project directory on the computer. When making a change in the folder with the file manager of the operating system, the change will also update in the Project view. It is possible to create folders and GameObjects or do a plenty of other actions in the file manager by clicking the right mouse button.

6.5.1 Crucial points

The student should get familiar with the user interface and understand the different parts of it. Some features can be found in various places of the interface so the student should find the most efficient way of working in Unity. Students should also learn to navigate well in the Scene view and use the basic manipulation tools. Hotkeys for the Position, Rotation and Scale are

really beneficial to get used to and students should be recommended to use those (Q, W, E and R).

6.5.2 Tutorials and assignments

Appendix 1 discusses the tutorial on how to create a new project and navigate in Unity.

6.6 Assets, GameObjects and Components

6.6.1 Assets

The assets in Unity are imported or created resources. For example 3D model, sound clip, texture image or script can be assets. Empty GameObjects are also a good start to create any other objects.

In an empty project the asset folder is empty. The asset folder in the project view should look like the image below (Figure 20).

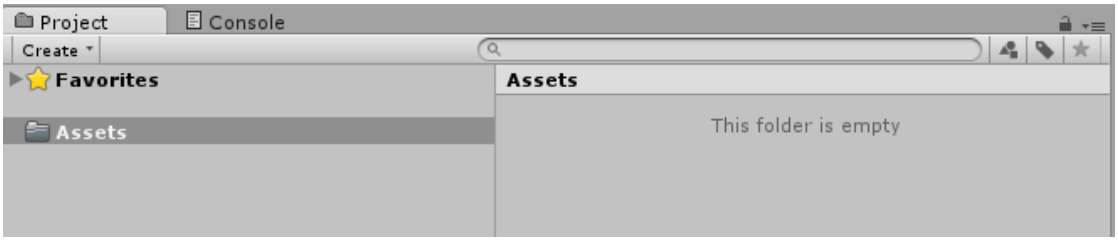


Figure 20. Project view in an empty project.

The asset folder can also be found in the game's project directory (See the figure 21). Library, ProjectSettings and Temp folders are not displayed in Unity's Project View.

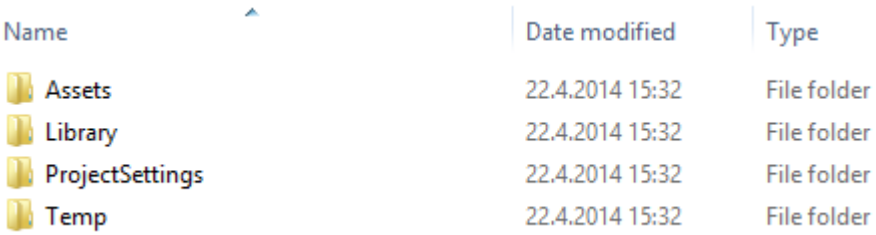


Figure 21. Game project directory.

Created or imported assets can be seen in the asset folder. It is advised to make subfolders for different kinds of assets.

Assets can be imported by clicking the right mouse button in the project view and then by clicking ‘Import new asset’.

6.6.2 GameObjects

Everything displayed in the scene and hierarchy view must be a GameObject. GameObject has the following properties in it.

- Name
- Tag
- Layer
- Static

The user can name the GameObject as he/she likes. There are some good naming conventions to be considered since the user may need to refer to the GameObject in code.

Tags are really powerful when referring to objects in code. Also searching the objects per tag may help in bigger projects. It is advised to give a tag to almost every GameObject there is in the project.

GameObjects can be assigned to different layers.

Static property is usually for GameObjects with meshes and which are static in the scene. This is used for lightmapping for example.

Empty GameObjects can be created by clicking the “Create Empty” option from the GameObject dropdown menu in the top. Empty GameObjects can be used to create a folder structure in the hierarchy view like in the image below (Figure 22).

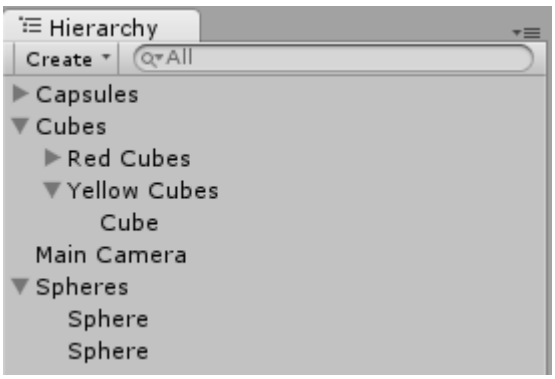


Figure 22. Hierarchy view folder structure.

There are also pre-made GameObjects to be used if wanted.

6.6.3 Components

Every GameObject has at least a Transform component which displays the position, rotation and a scale of the GameObject (Figure 23 represents the Inspector component).

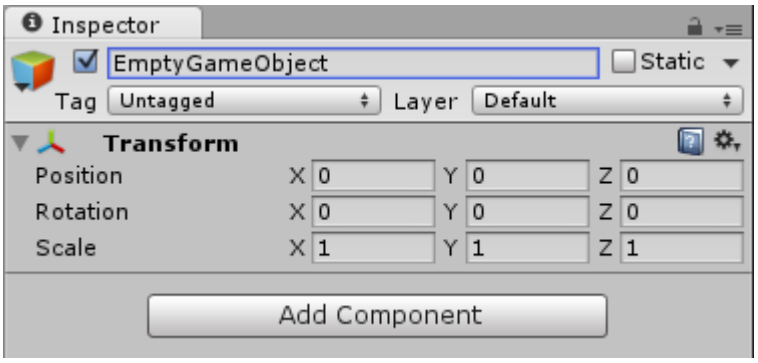


Figure 23. Transform component.

There are many different components which add various kind of functionality to GameObjects, e.g. mesh, effect, physics and sound components.

The image below (Figure 24) represents a GameObject with transform, Mesh Filter, Box Collider, Mesh Renderer and a Rigidbody physic component.

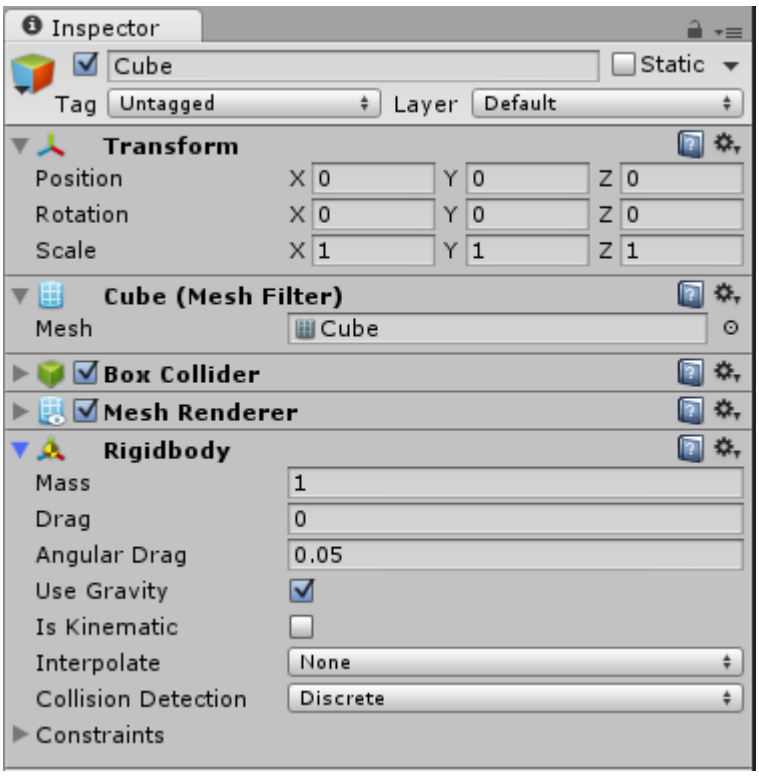


Figure 24. Components.

6.6.4 Prefabs

Prefab is really crucial type of GameObject and asset in Unity. In object-oriented viewpoint prefab can be considered as an object the user can make instances of. When a change to a prefab is made the change will be applied for the instances also. When GameObject is dragged from the Hierarchy view to Project views asset folder it becomes a prefab. Prefab is stored like any other asset.

6.6.5 Tutorials and assignments

See the appendix 2 for a tutorial how to make a character for a game.

See the appendix 3 for an assignment about GameObjects and components.

See the appendix 4 for a mandatory written assignment.

6.6.6 Crucial points

The student should understand how the following features operate and differ from others:

- Asset
- GameObject
- Component
- Prefab

The student should know that simple resources like texture images or sound clips and prefabs made from complex GameObjects containing many assets in it are both called assets. Students should know in what way GameObject, prefab and instance differ. Students should get familiar with the basic properties of the GameObject and the Transform default component.

These features are really crucial and the student cannot develop a game without them, so it is very beneficial to understand them well.

6.7 Scripting

6.7.1 Languages

Scripting in Unity is used for programming game behavior or mechanics. There are three possible scripting languages which are JavaScript, C# and Boo.

In this bachelor’s thesis the author is to using C# for examples. Some of the benefits of C# compared to JavaScript are:

- Visual Studio IDE
- No dynamic variable creation
- JavaScript documentation misleading with Unity

- Less confusion with compiler

JavaScript is the other scripting language which is widely used. In Unity the traditional JavaScript is a bit modified and it can also sometimes be called UnityScript. Some of its benefits compared to C# are:

- Simple syntax
- More code examples in the internet

The Boo is not very used language in Unity and there are not that much code examples or tutorials for it.

It is the developer's own decision which language he/she wants to use. Both languages perform as well as long as good code is written.

6.7.2 Game Loop

Unity executes the code within certain loops and they have a certain order (See the figure 25). The main game loop might look something like this:

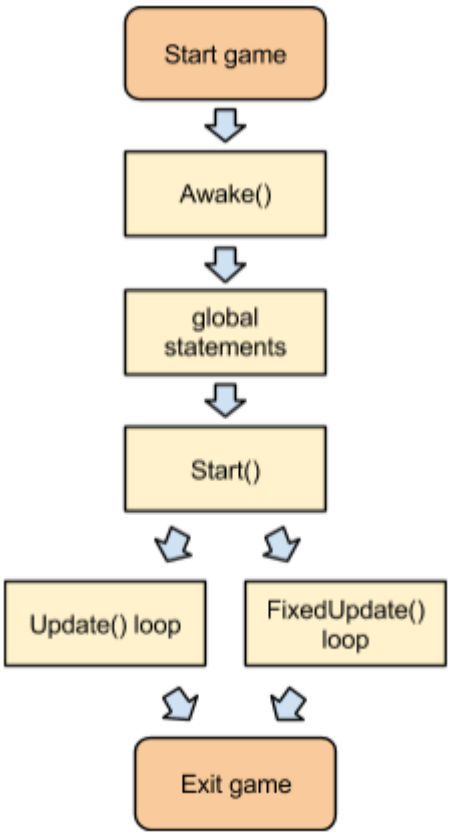


Figure 25. Unity's main game loop. (Jeremiah van Oosten, Scripting in Unity)

Awake() is the first function which is called when the game has started and its only called once. Awake() is a good place to set references between scripts. (Unity Scripting Reference 2014)

All the statements in the global scope will be executed after Awake() function and before Start() function.

Start() function will be called when all the GameObjects from the scene are loaded, right before the first update function. Start() function is also called only once. Because the Start() function is executed after all the GameObjects are loaded, it is a best place to make queries or references to GameObjects. (Jeremiah van Oosten, Scripting in Unity)

After Start() function, Unity will execute an update loop. There are two update loops.

6.7.3 Update() loop

The Update() loop looks like this (Figure 26):

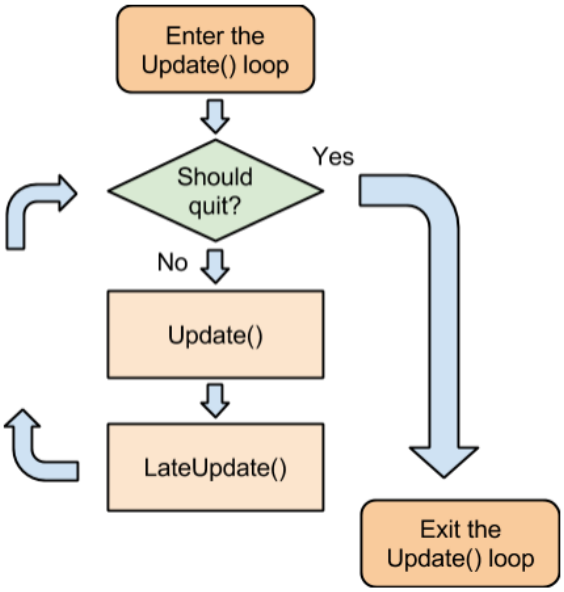


Figure 26. The Update() loop. (Jeremiah van Oosten, Scripting in Unity)

This loop is executed as many times per second as the computer can process it. This means that various computer perform differently.

When the loop is called the first thing it does is check if the loop should still be executed. If it should, it continues the loop by going in to the Update().

LateUpdate() is performed always after the Update(). This helps the user to manage in which order scripts are executed. Here the user can also check if some or all objects are updated before doing something else.

6.7.4 FixedUpdate() loop

FixedUpdate() loop looks like this (Figure 27):

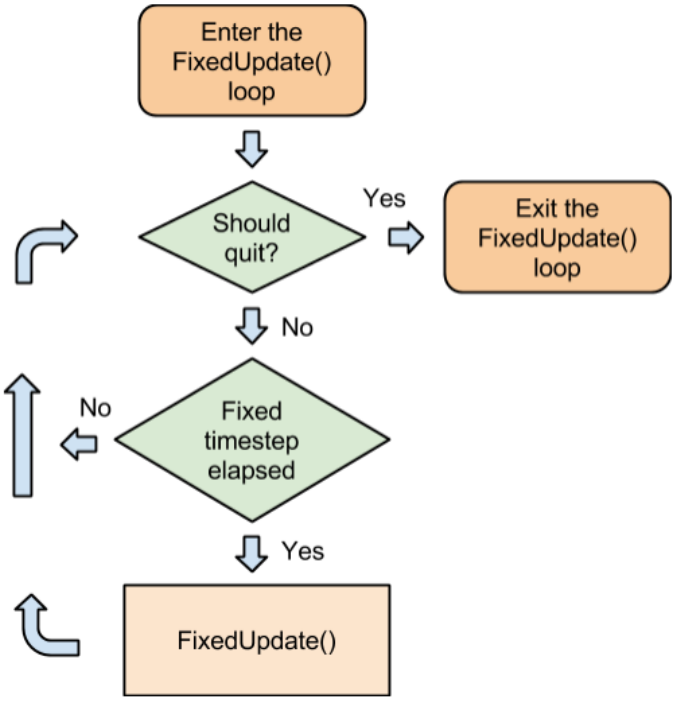


Figure 27. The FixedUpdate() loop. (Jeremiah van Oosten, Scripting in Unity)

FixedUpdate() loop executes in fixed time steps. This way the computers processing power is not relevant anymore. It is advised to use FixedUpdate() loop for everything related to physics for example.

The execution is almost the same as in the Update() loop, but there is a checkup if the fixed time step is elapsed.

6.7.5 Testing the game loop

The game loops execution order can be tested with the following script (Figure 28):

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class ExecutionOrder : MonoBehaviour {
5
6     void Awake () {
7         Debug.Log ("Awake()");
8     }
9
10    void Start () {
11        Debug.Log ("Start()");
12    }
13
14    void Update () {
15        Debug.Log ("Update()");
16    }
17
18    void FixedUpdate () {
19        Debug.Log ("FixedUpdate()");
20    }
21
22    void LateUpdate () {
23        Debug.Log ("LateUpdate()");
24    }
25
26 }
```

Figure 28. ExecutionOrder.cs

By saving this script, making an empty GameObject in the scene, including this script in it, playing the game and opening the console this (Figure 29) should be seen in the console as a result.

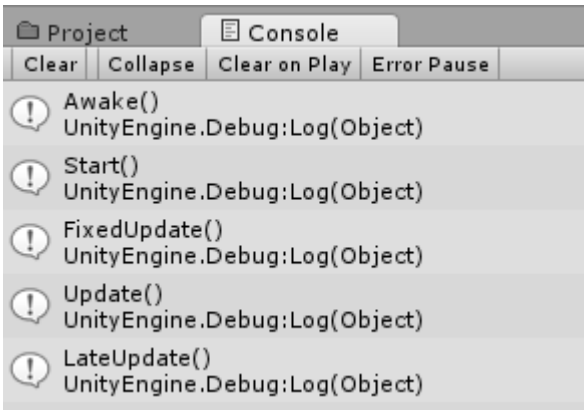


Figure 29. Execution order in console.

This is how the first 5 log entries in the console should look like. After these entries there should only be FixedUpdate(), Update() and LateUpdate() in various order depending of the computers processing power.

When the student is familiar with the game loop, he/she can arrange his code in the right order to get the wanted functionality easier and with less errors or bugs.

6.7.6 Time.deltaTime

Time.deltaTime is a function which should be used when the userwants to make something frame rate independent. If multiplied with Time.deltaTime the code will mean ‘do x 10 times per second’ instead of ‘do x 10 times per frame’.

Here’s an example (Figure 30):

```
void Update () {
    float translation = Time.deltaTime * 10;
    transform.Translate(0, 0, translation);
}
```

Figure 30. Time.deltaTime in use.

With this script attached to a GameObject the GameObject will move in z axis 10 units per second. If the Time.deltaTime would be removed, the GameObject would move 10 units per frame.

When something in code is related to frame rate the computer with more processing power will perform this faster. This is unwanted in some cases, especially in everything physic related.

(Unity Scripting Reference 2014)

6.7.7 Instantiate

Instantiate is a function which simply creates an instance of a prefab in the scene. Here’s an example of instantiation of a GameObject (Figure 31):

```
void Update () {
    Instantiate(prefab, new Vector3(0, 0, 0), Quaternion.identity) as Transform;
}
```

Figure 31. Instantiation of a GameObject.

This script instantiates a prefab, gives it position ‘x:0, y:0, z:0’ and no rotation.

Instantiation is very crucial in development and the students will get familiar with it in their projects.

6.7.8 Tutorials and assignments

See the appendix 5 for a “hello world” tutorial.

See the appendix 6 for a wall breaker tutorial.

See the appendix 7 for a mandatory written assignment.

6.7.9 Crucial points

The student should understand:

- In what order are the functions executed in the main game loop
- What are the differences between Update() and FixedUpdate() loops and those loops should be used
- What does the Time.deltaTime function do and when it should be used
- What does the Instantiate function do

Student should understand what is the right loop to include a certain function or a line of code, so that he/she will have minimal amount of errors and bugs and has less things to debug afterwards. Choosing the right loop can also make the game more efficient and easier to process.

The differences and benefits between functionality based on time steps of functionality based on frame rate are really crucial when developing with Unity. The student should know when something needs to be based on frame rate and when on time steps.

Time.deltaTime and Instantiate are crucial functions which will be used really often.

6.8 Physics

6.8.1 Rigidbody

Rigidbody is the physic engine in Unity. Everything that is about to be physic simulated with needs to have a rigidbody and a collider.

This is what the Rigidbody component looks like in the Inspector view (Figure 32):

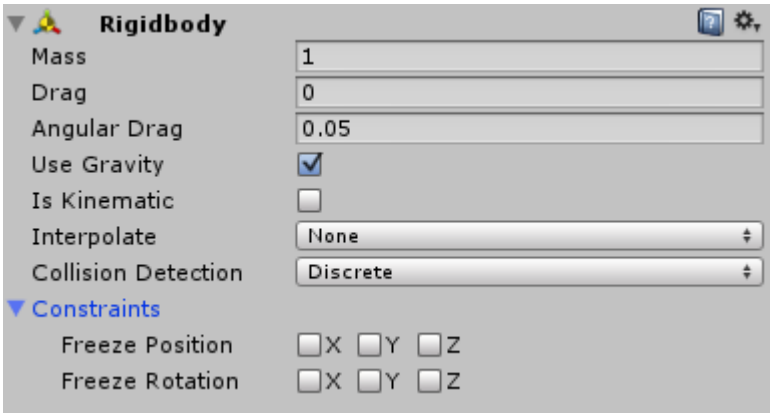


Figure 32. Rigibody component.

Mass is the mass of the GameObject. The mass should be adjusted to be relative to the size of the GameObject.

Drag controls how much the object velocity is reduced due the air resistance.

Angular drag reduces the angular velocity of the object

Use gravity just simply means if the GameObject should be affected by the gravity or not

Is Kinematic should be on if the user wants to move the GameObject with transform function. Kinematic rigidbodies are not affected by physics and only collide with other non-kinematic Rigidbody Colliders.

Interpolate smooths the movement. There are three options:

- None = No smoothing
- Interpolate = Smoothing based on the previous frame
- Extrapolate = Smoothing based on the estimated position of the next frame

Collision Detection has three different modes for detection:

- Discreet = Simplest detection which checks nearby colliders every frame. Can have some problems with small and fast moving objects.
- Continuous = Continuous detection against objects with static colliders and rigidbody colliders with continuous detection is set on. With other colliders it will perform discreet detection.
- Continuous Dynamic = Continuous collision detection against objects with continuous and continuous dynamic colliders and also with static colliders. For other type colliders it will use discreet detection. This should be used only for objects which move really fast.

Constraints = The object's position or rotation can be freezed in the selected axes.

(Jeremiah van Oosten, Physics in Unity)

6.8.2 Physic materials

Physic materials are assets that are used together with collider components. Default physic material looks like this (Figure 33):

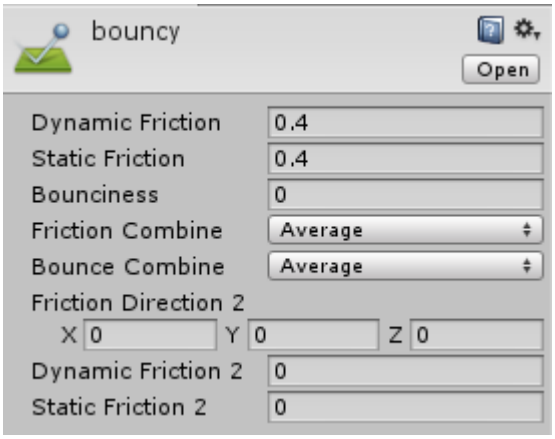


Figure 33. Physic material.

Dynamic Friction is the friction of moving object.

Static Friction is the friction of static object.

Bounciness is how much the object bounces when it collides with another collider.

Friction and **Bounce Combine** has four modes:

- Average = The average friction/bounce of two colliders.
- Multiply = Applied friction/bounce of both colliders.
- Minimum = The minimum friction/bounce of either of the colliders.
- Maximum = The maximum friction/bounce of either of the colliders.

Friction Direction 2 can be used to simulate surfaces that slide easier on one direction

Dynamic Friction 2 or **Static Friction 2** are the amount of friction in the direction of the selected axes of **Friction Direction 2.**

(Jeremiah van Oosten, Physics in Unity)

6.8.3 Colliders

GameObjects need colliders to Rigidbody to perform collision simulation. There are various kind of colliders for different purposes.

Trigger Colliders are colliders with isTrigger option checked. These colliders can be used to trigger events when another collider collides with them. They will start the ‘onTriggerEnter’, ‘onTriggerStay’ and ‘onTriggerExit’ events in the GameObject where the isTrigger is checked.

There can be a Rigidbody component in a GameObject where isTrigger is set to get the collider to be simulated with gravity, but it won’t collide with other colliders.

These triggers should be used for example for doors which should be opened when the player approaches them or launch other events in the game.

Static Colliders are colliders with no trigger or rigidbody. Rigidbody colliders will collide with static colliders, but they cannot move static colliders. Static colliders should be used for example walls and floors.

Rigidbody colliders are colliders with rigidbody physic component. They will collide with other rigidbody colliders and static colliders, and when entered in trigger collider area they will get the ‘onTriggerEnter’, ‘onTriggerStay’ and ‘onTriggerExit’ events.

Kinematic Rigidbody Collider is a colldier with isKinematic flag set on. These colliders will not collide with other colliders and they must be moved with Transform components values.

They can collide with rigidbody colliders and move them, but they cannot be moved with other rigidbody colliders themselves. They will not collide with static colliders.

Box Collider (Figure 34) is a collider which is box shaped. The properties of box collider are:

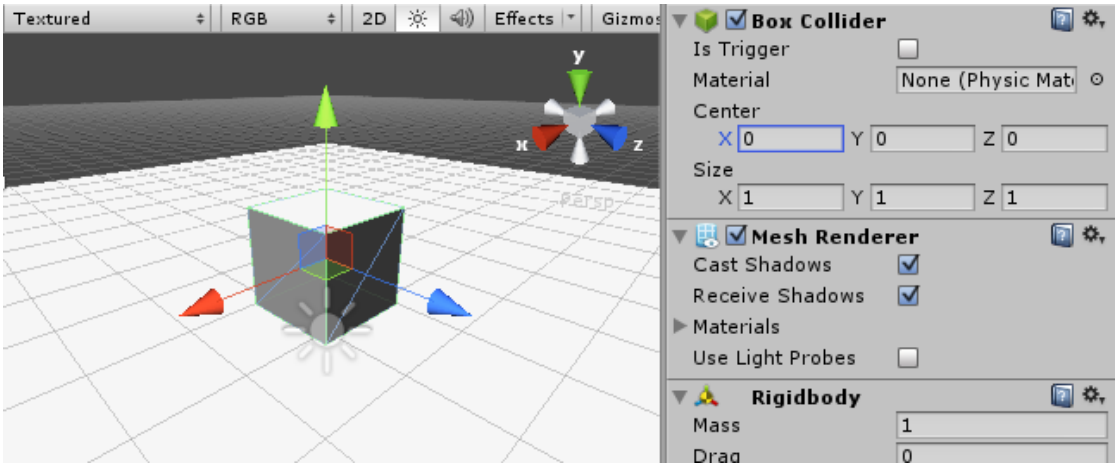


Figure 34. Box collider.

Is Trigger is the flag if the collider is meant to be a trigger or just a normal collider.

- **Material** is the physic material attached to the collider.
- With **Center** the collider's position can be adjusted.
- With **Size** the size of the collider can be adjusted.

Box collider is typical choice for triggers.

Sphere collider looks like this (Figure 35):

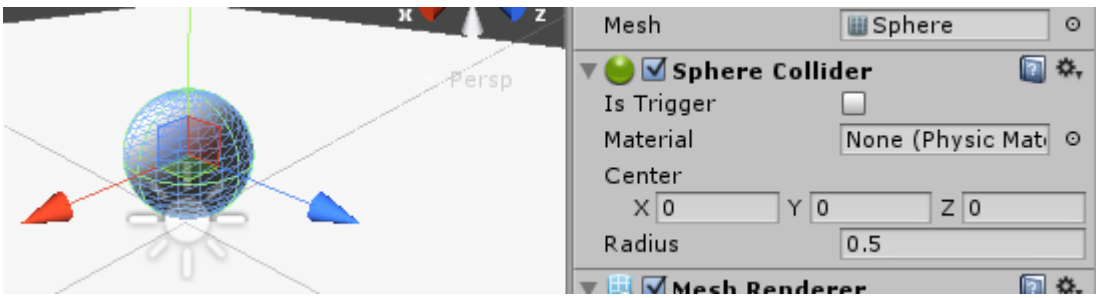


Figure 35. Sphere collider.

It has almost the same properties as the box collider, but the size is determined by the radius.

Capsule Collider (Figure 36) is a collider shaped like a capsule and it is used for the bones of animated character or for the whole character itself. Capsule collider looks like this in the inspector view:

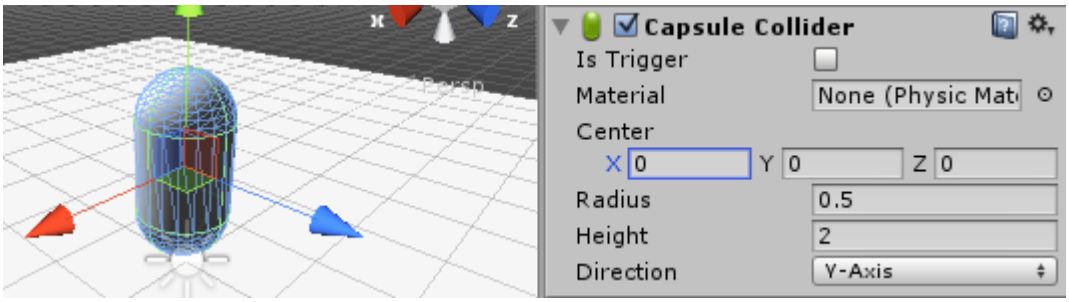


Figure 36. Capsule collider.

Capsule collider also has all the typical properties listed earlier but the height and direction should also be selected.

Compound Colliders are normal primitive colliders combined together with parent-child hierarchy. This way the user sometimes does not need to use a complex mesh collider and can save some processing power.

Mesh Colliders are colliders which use the mesh of the 3D model to create its shape. These colliders can be really expensive in the terms of processing power, they should only be used when the user absolutely must. (Figure 37 represents a mesh collider and it's properties)

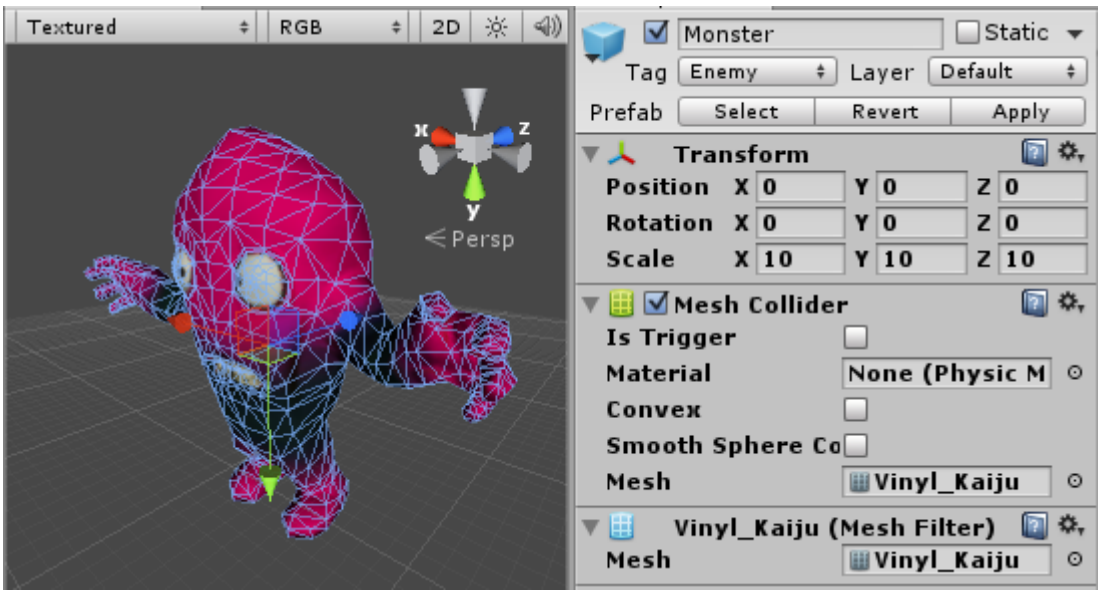


Figure 37. Mesh collider.

Is Trigger is the flag if the collider is meant to be a trigger or just a normal collider.

Material is the physic material for the collider.

Convex is a simplified version of the mesh and which collides with other mesh colliders.

Smooth Sphere Collision will smooth the edges of the mesh. This can be used for example on terrains.

Mesh is the selected mesh which shape the collider takes.

(Jeremiah van Oosten, Physics in Unity)

6.8.4 Character controller

Character controller is a controller which is not affected by physics. It is usually used with first person or third person characters. If the user wants to apply forces when the character collides with another GameObject he/she has to do it with the 'onControllerHit()' function. Character controller's collider is normally capsule shaped. (See the figure 38 for Character controllers properties)

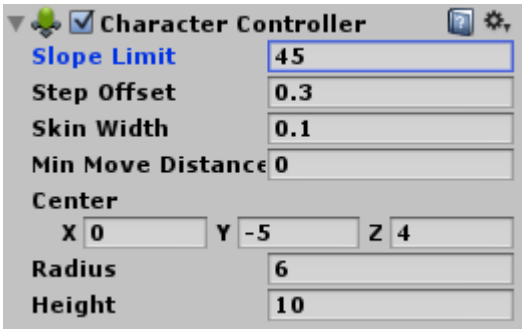


Figure 38. Character controller component.

Slope Limit is the highest angle the character can climb up.

Step Offset is the maximum height that character can step over.

Skin Width should be increased when the character is getting stuck too easily.

Min Move Distance is the minimum distance the character will move.

With the **Center** properties the user can fix the capsule colliders position for the character.

Radius and **Height** are for adjusting the size of the capsule collider.

Unity has two example characters which are in the Character Controller default package.

If the character has to be affected by physics, the character controller should not be used. Heavily realistic physics in characters might not be fun to play at all.

(Unity Script Reference 2014)

6.8.5 Tutorials and assignments

See the appendix 8 for an example assignment.

See the appendix 9 for mandatory written assignment.

6.8.6 Crucial points

The student should understand the Rigidbody and its properties, how do different kinds colliders and especially how do trigger collider's work, where physic materials are used and when to use Character Controller.

6.9 Rendering

Unity has a bunch of rendering components and features in it. In this chapter the author is going to tell about cameras, shaders, lighting and light mapping.

6.9.1 Camera

The camera is a component which captures the view from the scene to be drawn in the screen. There has to be at least one camera per scene. When

new scene is created, there is always the Main Camera component. (See the figure 39 for camera components properties)

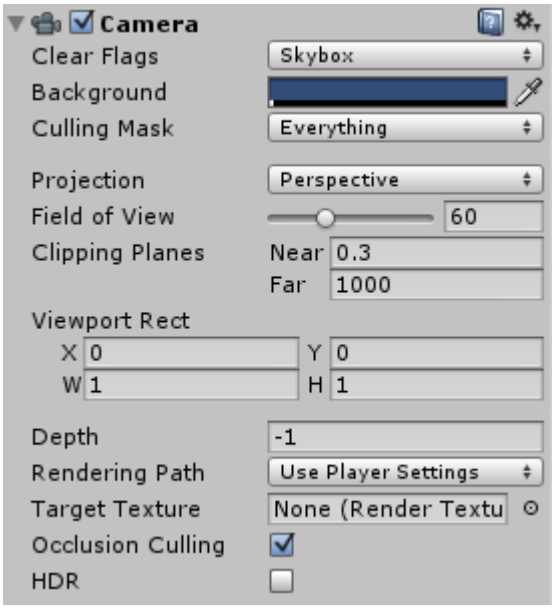


Figure 39. Camera component.

Clear Flags has three options

- Skybox – This will use the Skybox to clear the scene before drawing any objects. Skybox Material properties can be found from the Render Settings.
- Solid Color – This will use the selected solid color to clear the scene.
- Depth Only – This does not clear the color buffer, only the depth buffer.
- Do not Clear – Depth and color buffer will not be cleared.

Background is the background color and will be rendered if the solid color flag is chosen or if there's no skybox set.

Culling Mask is used to select which layers are rendered to the camera.

Projection has two values

- Perspective – This view type simulates the human sight. Used for 3D games.

- Orthographic – This view type renders the object near the camera and the same object far away the same size. It has no depth. This is commonly used for 2D, isometric and top-down games.

Field of View is the angle of the camera projection. The default is 60 degrees. This property is only for the Perspective projection.

Size means the area where the objects are rendered in the Orthographic projection.

Clipping Planes are the range of the camera.

- Near – This means how close the object can be to the camera so it will still be rendered.

- Far – This means how far the object can be to the camera so that it will still be rendered.

Normalized View Port Rect has the properties to adjust where the camera component will render the view in the scene. This can be used for mini-maps or for split screen games. It has properties **Z** and **Y** to adjust the place of the view port and properties **W** and **H** to adjust the size of the port.

Depth property determines in which order the camera components will render.

Rendering Path has three options for rendering method (also Deferred Lighting in Unity Pro):

- Use Player Settings – Camera will use the settings set in Player Settings.

- Vertex Lit – Per vertex lighting instead of per pixel.

- Forward – This is the default method.

6.9.2 Skybox

Skybox draws an environment far away around the player. The user can assign a default skybox from Rendering Settings or add a skybox component in a camera. Unity has some default skyboxes in the ‘Skyboxes’ package. (Figure 40 shows the skybox in the background)

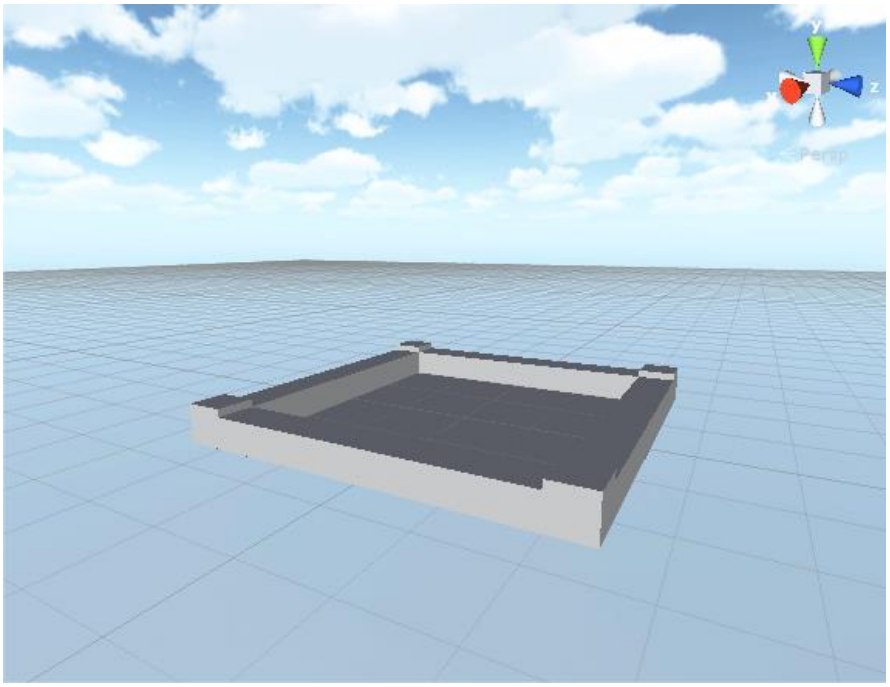


Figure 40. Skybox.

6.9.3 Materials and shaders

Shaders are code which h tells the Unity how different materials should be rendered. Unity has a set of built-in shaders and the most used ones will be covered next.

Vertex Lit is a shader type which performs fast but doesn’t perform any lighting effects like normal mapping or shadows. Should be used on objects which doesn’t need to have that good lighting effects. (See the figure 41 for vertex lit properties)

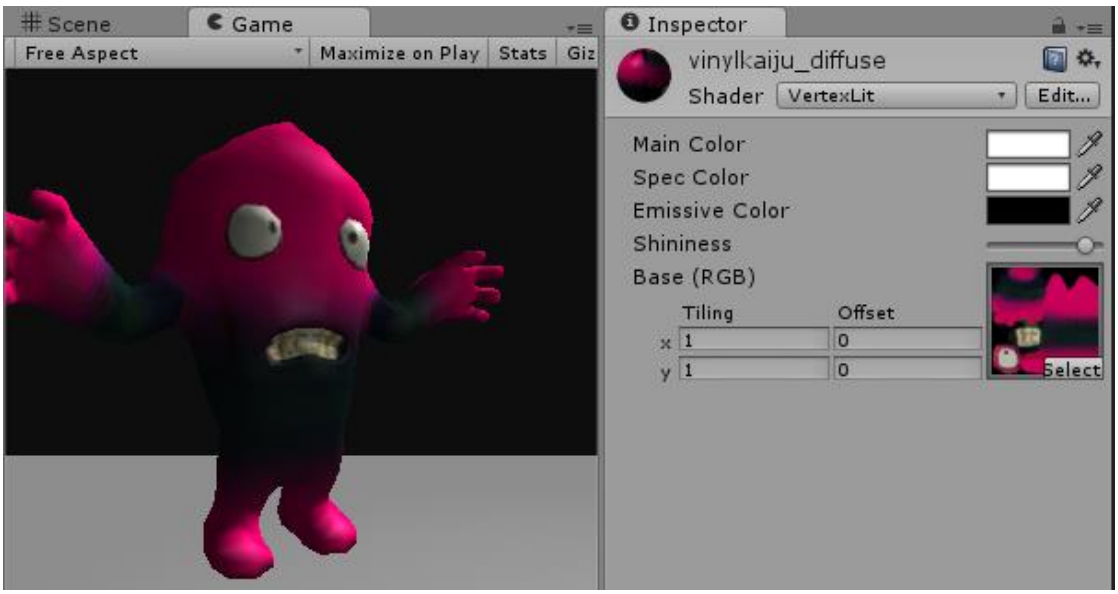


Figure 41. Vertex Lit.

Vertex Lit (and many other shader) has following properties:

Main Color can be used to darken the texture by selecting a darker shade of gray.

Spec Color is the color of specular highlight.

Increasing **Shininess** the specular highlight gets weaker.

Base is the base texture image used on the material. With the **Tiling** and the **Offset** the user can tile the image and move it.

Diffuse shader is per-pixel lighted and it does not have specular highlight. (See the figure 42 for diffuse shader properties)

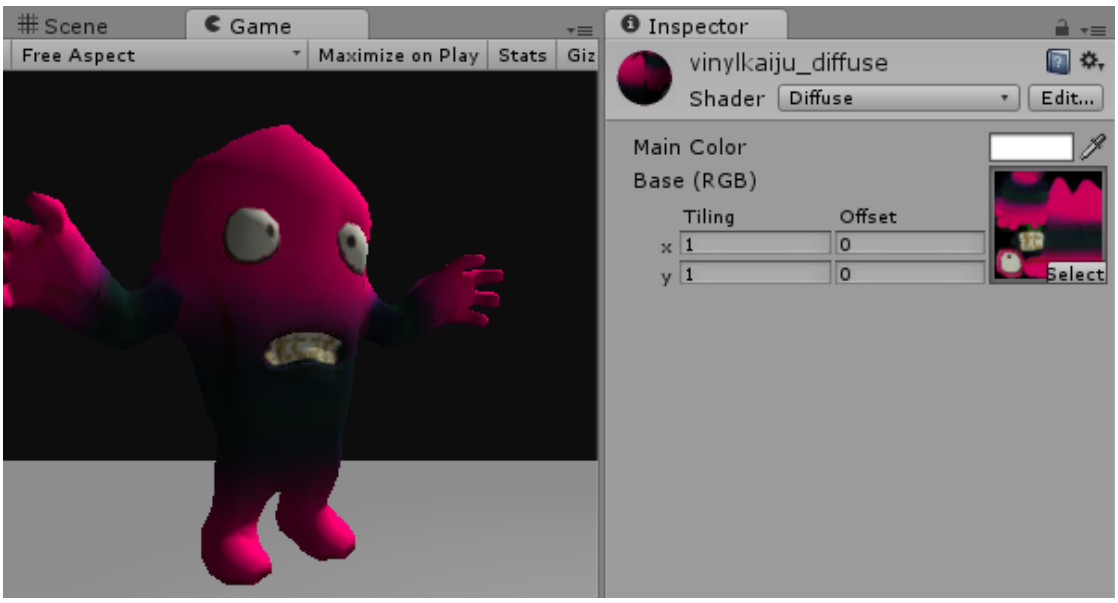


Figure 42. Diffuse shader.

Diffuse shader has same properties as the Vertex Lit, but lacks the specular highlight properties.

Specular shader adds the specular highlight properties. (See the figure 43 for specular shader properties)

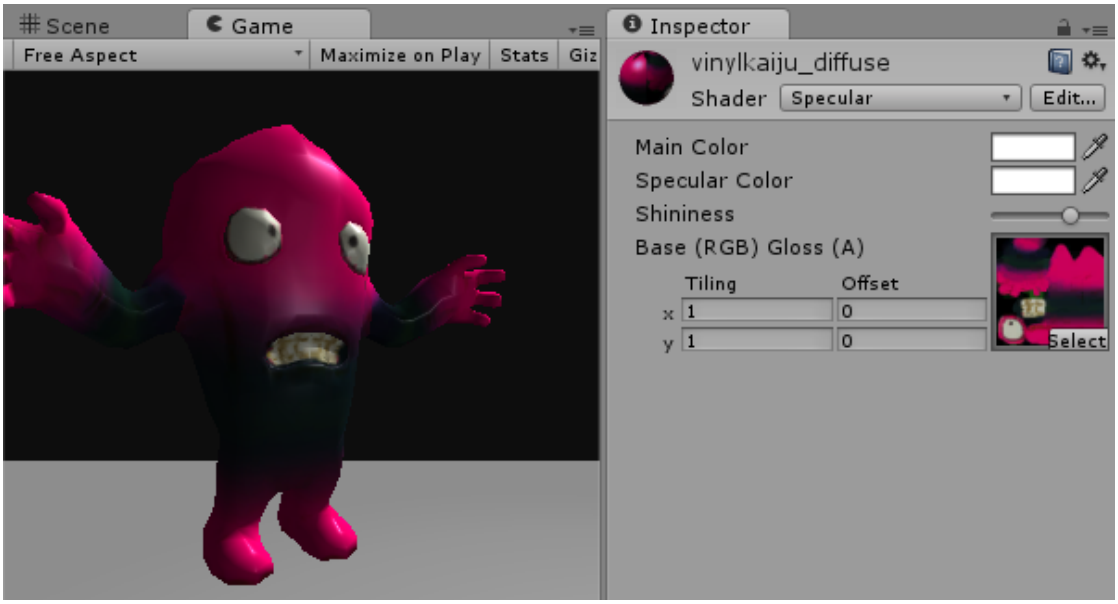


Figure 43. Specular shader.

Bumped Diffuse adds the normal map property. Normal maps are used to create more detail on the object without any new vertices. Bumped Diffuse shader does not have specular highlight. (See the figure 44 for bumped diffuse shader properties)

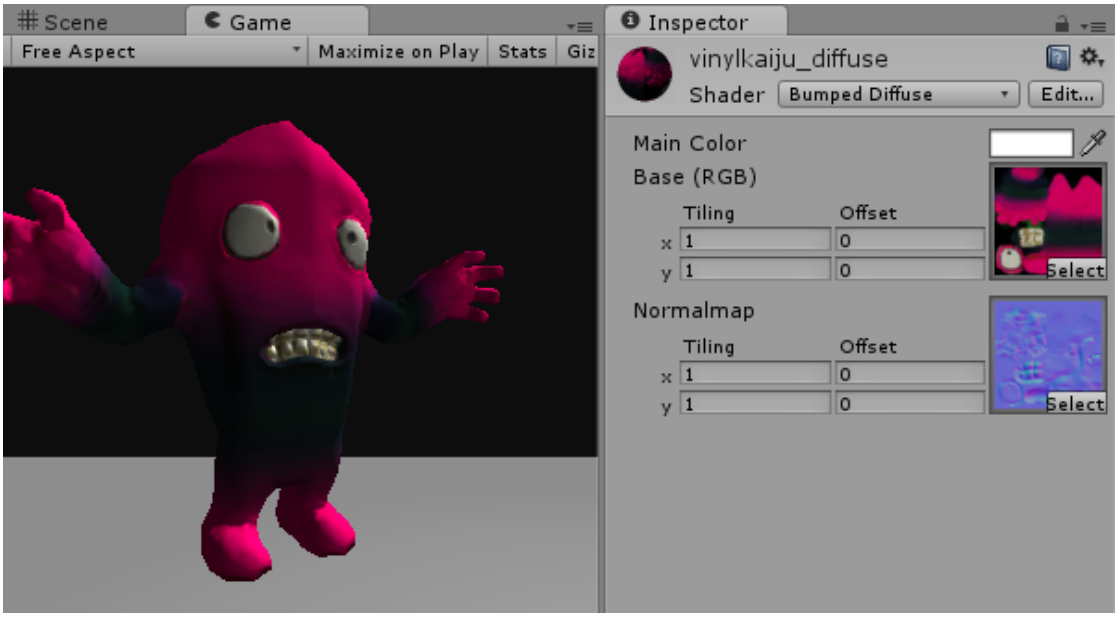


Figure 44. Bumped Diffuse shader.

Tiling and **Offset** adjustments can also be done for the normal map.

Bumped Specular shader adds the specular highlight properties. (See the figure 45 for bumped specular shader properties)

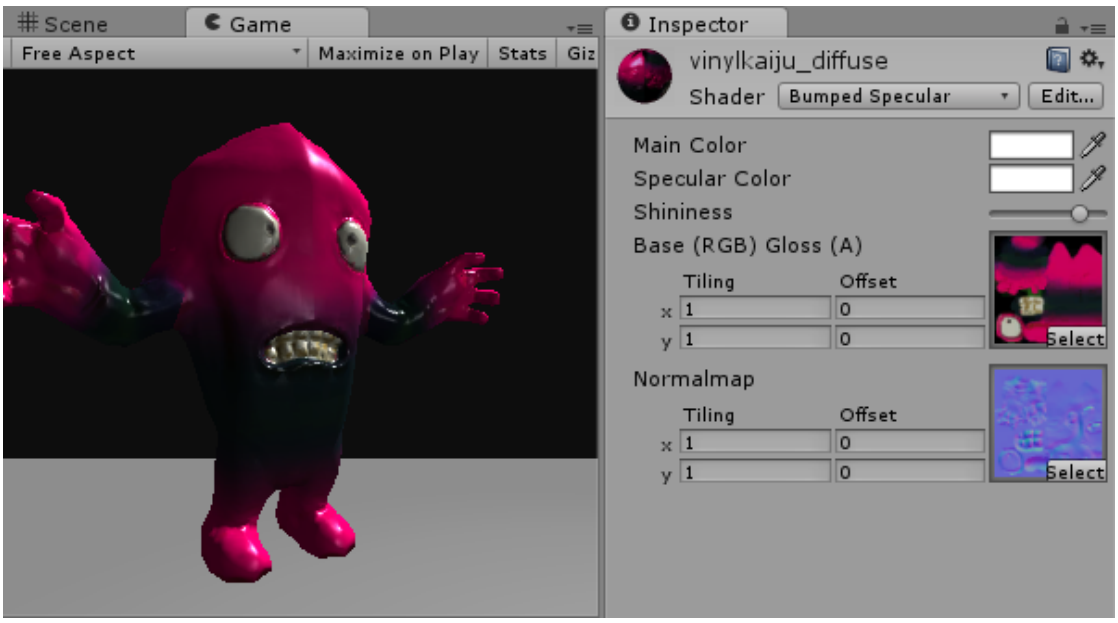


Figure 45. Bumped Specular shader.

Parallax Diffuse shader adds the heightmap property. The shader uses the alpha channel of the texture to determine the height of certain parts of the texture. Usually used for example brick wall textures to highlight the bricks.

Parallax Specular is the same as the diffuse, but with added specular highlight properties.

With **Decal** shader the user can add decals in textures. This can be used for example to make a scratch on a wall or something similar.

Diffuse Detail shader adds some detail properties on the material. The detail will fade in once the player get closer to the object. Can be used for hiding rendering artifacts at close range.

There are still several more shaders, for example Transparent, Self-Illuminating and Reflective shaders, but those are more advanced shaders and will need another contact lesson to fully cover.

6.9.4 Lights

The scene needs light so that the player can see all the GameObjects and other assets. In the Unity Free there are three light types:

- Directional light
- Point light
- Spot light

Directional Light is a typical choice of light for outdoors because it simulates the light of sun. The directional light can be positioned anywhere since its position will be ignored. The direction of the light can be changed.

This light type is the only type that will produce shadows with the default rendering mode Unity Free supports. Dynamic shadows are Pro license feature. In Unity Free shadows can be produces from other light types by light mapping. (Figure 46 shows how the directional light affects on character)

Figure 46. Directrional light.

Directional lights properties (Figure 47):

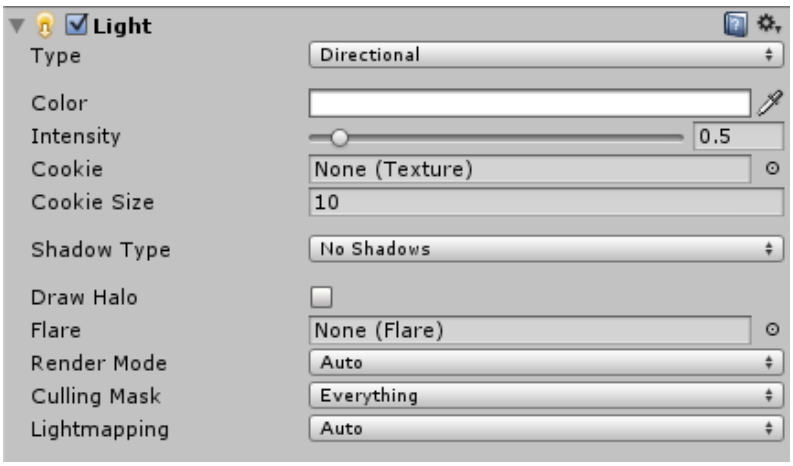


Figure 47. Directional light properties.

Type is the light type. Directional, point or spot.

Color is simply the color of the light.

Intensity is how bright the light is.

Cookies are textures which are used to determine the brightness of light. Values are taken from the textures alpha channel.

Cookie Size scales the size of cookie texture.

If using Unity Free **Shadow Types** only render with the directional light. Shadow types are **Soft Shadows** and **Hard shadows.** There are some options for shadows:

- Strength = How dark/light the shadow is
- Resolution = The resolution of the shadow texture.
- Bias = this can be used to remove artifacts.
- Softness = This property is only available if the soft shadows are selected.
- Softness Fade = This fades the soft edges of the shadows based on distance to the camera.

Draw Halo this option draws a halo based on the position of light. This should be used with Point lights and Spot lights.

Flare is the lens flare effect which is seen in real life cameras. Flare texture can be assigned from here.

Render mode is how the light is applied to objects. It has three different values:

- Auto = Quality is determined by the proximity and intensity of the light relative to the objects.
- Important = Uses per-pixel render shaders.
- Not Important = Uses per-vertex render shaders.

Culling Mask toggle layers which will be affected by the light.

Lightmapping determines how the lights are baked. It has three options:

- Auto – Uses realtime lighting when the objects are near the camera and baked mapping when the objects are far away. Requires dual lighmaps to be baked in the scene.
- Realtime Only – Lighting information are not baked from this light
- Baked Only – Not simulated at run time, but only used to bake the lightmap.

Point Lights light the scene in every direction from their position, so there is no option for direction but the positioning is really important. (Figure 48 shows how the point light affects on character)

Figure 48. Point Light.

Point light properties (Figure 49):

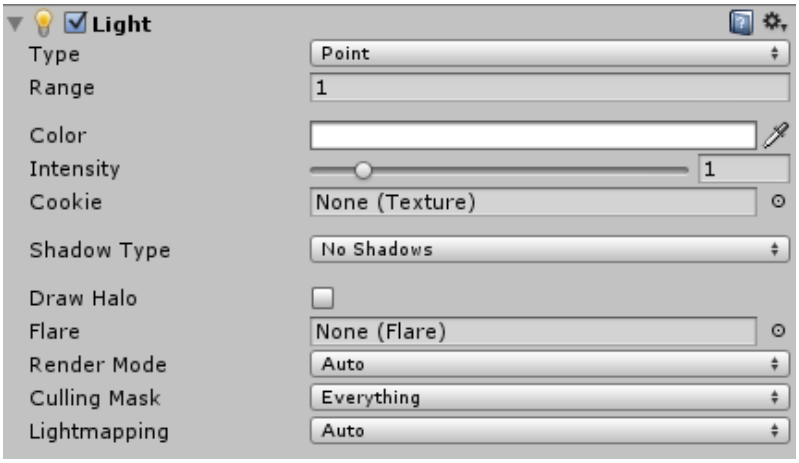


Figure 49. Point light properties.

Range determines the radius of the point light.

Spot Lights are lights commonly used for flashlight, headlights or street lamps. (Figure 50 showsh how the spot light affects on character)

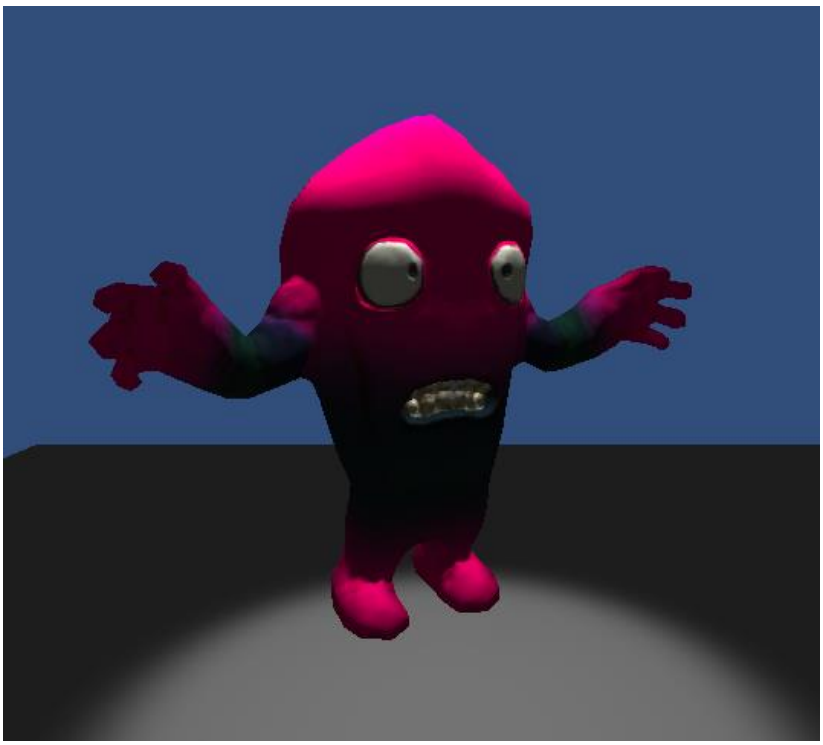


Figure 50. Spot Light.

Spot light properties (Figure 51):

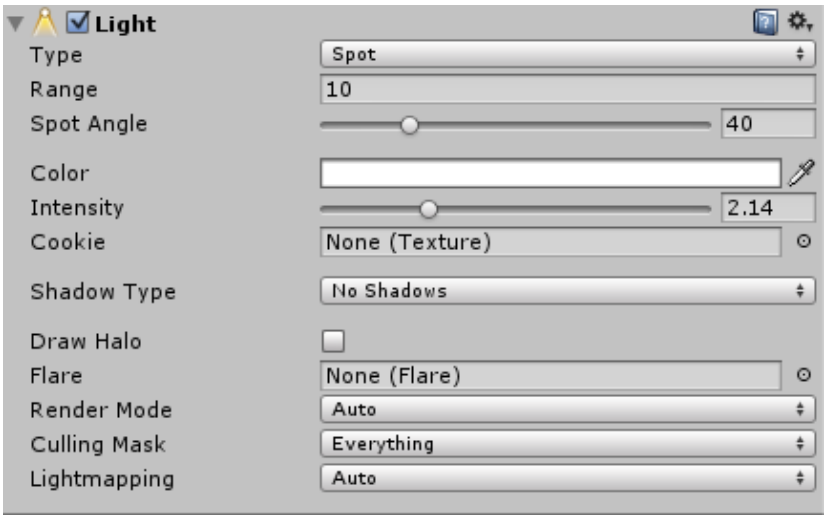


Figure 51. Spot light properties.

Range determines the distance of the light

Spot Angle determines the width of the spot and the angle of the light “cane” created.

(Jeremiah van Oosten, Rendering and Special Effects in unity 3.5)

6.9.5 Lightmapping

Lightmapping is a process where all the lighting information is baked on the textures. This is something that should be done for static objects which does not need a real time shadows to save processing power. The process of making the lightmaps in the textures are called baking.

Lightmapping settings can be found from the lightmapping window (Click Window from the top menu and then Lightmapping). Only objects with Static property set on can be baked. All the objects and lights to be baked must be selected from the Objects tab in the lightmapping window. The bake tab has the settings for the baking (See the figure 52 for the settings).

Figure 52. Bake tab.

Mode has the options **Single Lightmaps, Dual Lightmaps** and **Directional Lightmaps.**

Single Lightmaps bakes only the far lightmaps.

Dual Lightmaps bake both, the far and near lightmaps.

Directional Lightmaps are for directional lights.

Quality has two presets for quality: High and Low. High is better looking but the low is faster.

Bounces are the light bounces in the global illumination simulation. The more there are the more smoother the result is.

Ambient Occlusion determines the amount of ambient occlusion to be baked in the lightmap.

LOD Surface Distance. LOD groups are Unity Pro features.

Lock Atlas toggles the automatic atlasing.

Resolution is the resolution of the lightmap in texels.

Padding is the space between objects in the atlas.

(Unity Manual)

The baking may take a while depending on the computers processing power.

Here is a scene where all the lights are baked in the textures and there is no real-time lights in the scene (Figure 53).

Figure 53. Baked scene.

6.9.6 Tutorials and assignments

See the appendix 10 for an example assignment.

See the appendix 11 for mandatory written assignment.

6.9.7 Crucial points

Students should understand the difference between the perspective and orthographic camera modes, how to use different kinds of shaders and their properties to make good looking objects and to understand the difference between real time lighting and lightmapping.

7 Conclusion

7.1 The results

The most essential part of this thesis is a guide for JAMK University of Applied Sciences how to organize the teaching of Unity3D in the Game Programming module and provides demonstrative lecturing material and gives some examples of possible tutorials and assignments. It should help the one who organizes the Game Programming module, but also the one who will be teaching Unity3D in it.

It covers only the absolutely crucial features and aspects of Unity3D.

There are some issues to mention about using the Unity. I would suggest the students to be taught to use hot keys as much as they can. The basic manipulation tools found under Q, W, E and R keys will be used really often and selecting with mouse is slow. There are usually many ways to do a certain task in the interface and I would suggest to let the student do what he/she founds the most efficient. There are also some good naming conventions in the UI and also in scripting, but I did not want to discuss them since they can be a bit controversial.

The scripting part requires the students to have some knowledge about programming but since the scripting is so big part of game development it is only a scratch on the surface.

There are some problems in sticking with Unity. Unity might be a good engine to be considered at this moment, but it might not be so in future. Unreal Engine and CryEngine both lowered their prices a while ago and are now good competitors for Unity in the Indie market. Also the lack of Unity Pro’s features can sometimes be seen in the results. This is why I suggest to have at least one or two Pro licenses in use.

If the guide should be developed further I would do so by adding more tutorials, since the students find them a great way of learning. The scripting is really crucial part of the game development, so it would be the first thing I

would develop further. The guide could also provide information about subjects which are not that crucial. For example effects, terrains, UI and networking.

The chapter about the Game Programming module can also be valuable since it has some ideas for developing the module based on the course feedback. I think taking the Unity or any other engine in the course was absolutely a great decision since now the course can be more interesting for those who like to contribute in game development in other ways than programming.

The research part of the thesis can be seen valuable for those who have little to none knowledge about game industry, production, development or engines. I found the Heartstone to be a great example since Blizzard is really known game company and Heartstone is a differs a bit from their bigger game titles. I was a bit surprised they had used Unity for the game.

The appendices can be used as exercises or assignments like they are, but I would maybe develop them to be bigger or at least have more of them. Mandatory written assignments can be used as such and I would suggest that the students are expected to do some research and maybe use some references for answering those questions.

7.2 Teaching as an experience

The Game Programming module in spring 2014 was the first time ever for me to teach any subject. It was an interesting and challenging experience. I now see clearly what can be done do make the course and my teaching better.

Teaching itself was fun and I would gladly be developing my skills as a teacher further. I am also motivated to develop the course and offer the students the best education possible concerning the subject.

7.3 Developing my skills

I should develop my skills with Unity further, especially advanced level scripting. I also should get more familiar with other game engines because we

cannot be sure if there will be a reason to change the engine to other one in the future.

As a teacher I should be able to provide better instructions and deadlines for assignments. I should forget how I was taught and adapt to students needs regarding the educational solutions. I also shouldn’t stick to just a few sources I find proficient, but instead I should always find more ways and more perspective on the subject.

I didn’t have to teach in English this time, but I really should prepare myself to do so in future.

References

Blackman, S. 2013. Beginning 3D Game Development with Unity 4. Second edition. New York. Apress.

Beginning 3D Game Development with Unity 4 cover. Picture from Unity3Dbooks website. Accessed on 11.5.2014. Retrieved from: http://unity3dbooks.com/wp-content/uploads/2013/02/51y4x59CDAL1-300x368.jpg

de Byl, P. 2012. Holistic Game Development with Unity. USA. Focal Press.

Chandler H.M. 2014. The Game Production Handbook. Third Edition. Burlington, USA. Jones & Barlett Learning.

Character Controller. Page from Unity Scripting Reference. Accessed on 1.5.2014. Retrieved from: http://docs.unity3d.com/Documentation/Components/class-CharacterController.html

Essential facts about the computer and video game industry 2013. PDF document from Entertainment Software Association's (ESA) website. Retrieved from: http://www.theesa.com/facts/pdfs/esa_ef_2013.pdf

Global Games Market Report Infographics 2013. Page on Newzoo's website. Accessed on 20.3.2014. Retrieved from: http://www.newzoo.com/infographics/global-games-market-report-infographics

Heartstone Logo.png. Heartstone Wiki. Accessed on 1.5.2014. Retrieved from: http://hearthstone.wikia.com/wiki/File:Hearthstone_Logo.png

Holistic Game Development with Unity. Picture from Unity3DBooks.com website. Accessed on 11.5.2014. Retrieved from: http://unity3dbooks.com/wp-content/uploads/2013/02/51hWQFeRB1L1-300x390.jpg

Industry Info 2014. Page on Neogames website. Accessed on 12.3.2014. Retrieved from: http://www.neogames.fi/en/industry-info/

International Game Architecture and Design programme brochure 2013. PDF document from NHTV website. Retrieved from: http://www.nhtv.nl/fileadmin/user_upload/Documenten/PDF/Brochures/NHTV_Games.pdf

International Game Developers Association's website. Referred 20.3.2014. Retrieved from: https://www.igda.org/?page=about

Jeremiah van Oosten. 3DGEP blog. Accessed on 13.5.2014. Retrieved from: http://3dgep.com/

Jeremiah van Oosten 2012. Physics in Unity. Accessed on 30.4.2014. Retrieved from: http://3dgep.com/?p=4027

Jeremiah van Oosten 2012. Rendering and Special Effects in Unity 3.5. Accessed on 3.5.2014. Retrieved from: http://3dgep.com/?p=3856

Jeremiah van Oosten 2012. Scripting in Unity. Accessed on 25.4.2014. Retrieved from: http://3dgep.com/?p=3474

The Game Industry in Finland 2012. PDF document from TEKES website. Retrieved from: http://www.tekes.fi/Julkaisut/the_game_industry_of_finland.pdf

License Agreement 2013. Licence Agreement from Unity's website. Retrieved from: http://unity3d.com/company/legal/eula

Lightmapping In-Depth 2013. Page from Unity Manual. Retrieved from: http://docs.unity3d.com/Documentation/Manual/LightmappingInDepth.html

Multiplatform. Page from Unity's website. Accessed on 6.3.2014. Retrieved from: http://unity3d.com/unity/multiplatform

NHTV logo. Logo from NHTV's website. Accessed on 11.5.2014. Retrieved from: http://www.nhtv.nl/ENG/about-nhtv/press/nhtv-logos.html

Public relations. Page from Unity's website. Accessed on 5.3.2014. Retrieved from: https://unity3d.com/company/public-relations

Time.deltaTime. Page from Unity Scripting Reference. Accessed on 29.4.2014. Retrieved from: http://docs.unity3d.com/Documentation/ScriptReference/Time-deltaTime.html

Unity. Page from Unity's website. Referred 5.3.2014. Retrieved from: http://unity3d.com/unity

Unity Website. Referred 13.5.2014. Retrieved from: http://unity3d.com

Unity Logo. Logo from Unity's website. Referred 11.5.2014. Retrieved from: http://unity3d.com/profiles/unity3d/themes/unity/images/company/brand/logos/primary/unity.jpg

Wawro, A 2014. Heartstone heralds new challenges for Blizzard on Mobile. Interview on GamaSutra's website. Retrieved from: http://www.gamasutra.com/view/news/214930/QA_Hearthstone_heralds_new_challenges_for_Blizzard_on_mobile.php

APPENDICES

Appendix 1

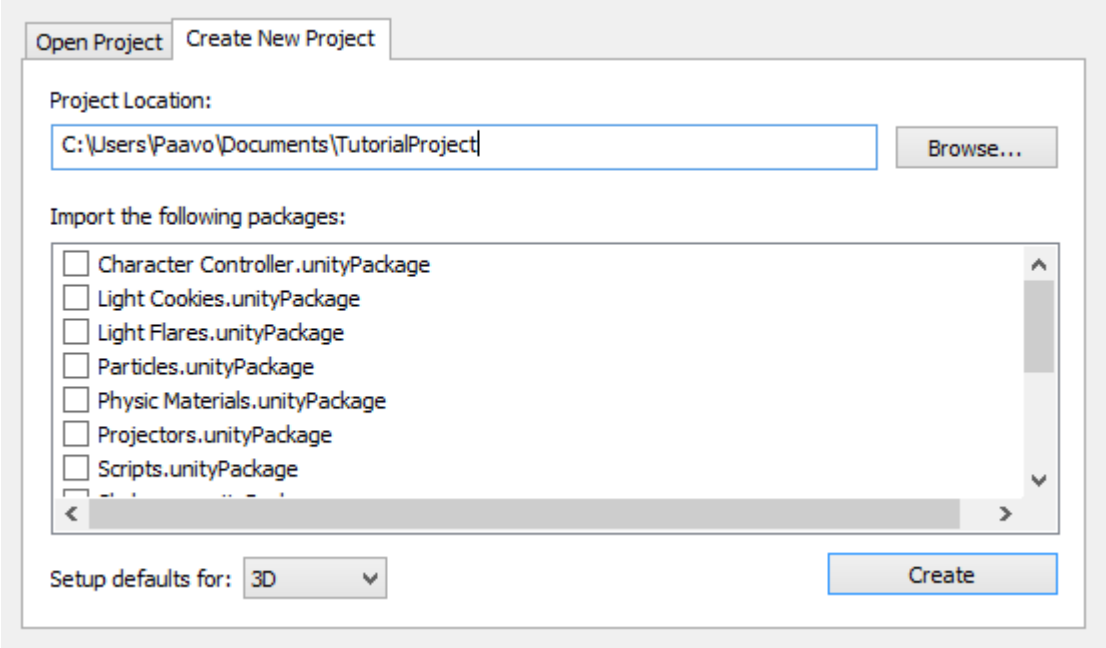
For contact lesson: Interface

In this tutorial we are going to learn how to make a new project, what does different windows include and how to use Unity's navigation, position, rotation and scale tools.

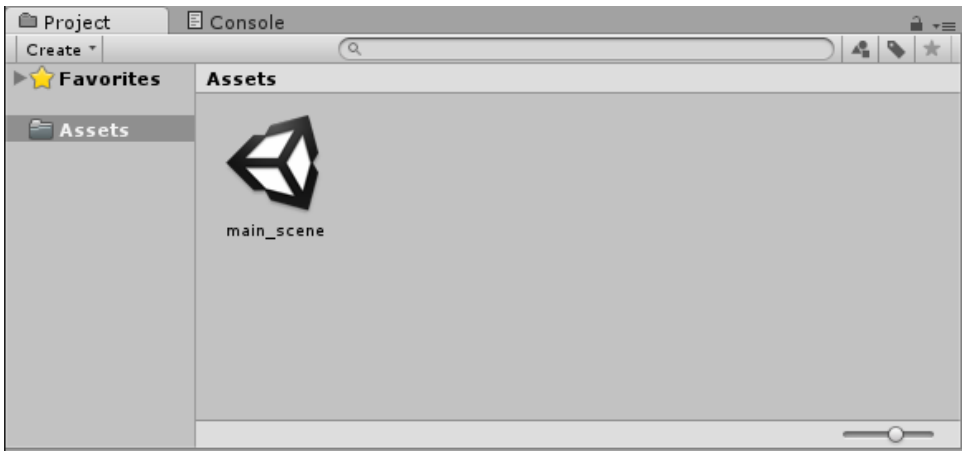
1) Start Unity and click File -> New Project

2) Select the location of the project and name it

3) You can import a default package from here but you can also import them later. Let's not import anything this time. Let the 3D defaults be selected and then press create.

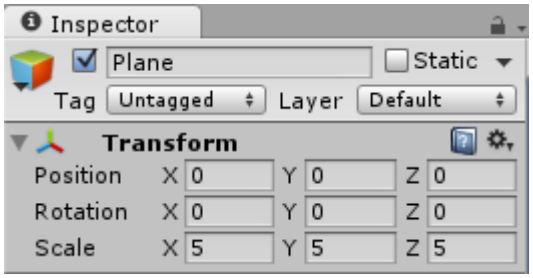


4) Now we have an empty project. Let's save the scene we are starting to work on. Click 'File -> Save Scene As' and name it as you like (MainScene for example). The scene should now display in our Project view in the Assets folder. This view includes all our assets. You will learn more about assets in the next tutorial.

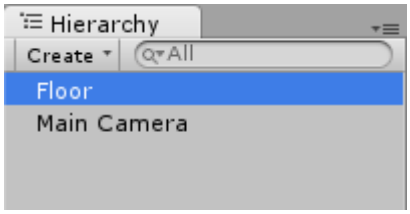


5) Now let's start adding visible things in our scene by adding a plane. You can add a plane in to your scene from 'GameObject -> Create Other -> Plane' or from the Hierarchy view by clickin the Create and then Plane.

6) Select the plane you just made from the Scene view or from the Hierarchy view. You can make the plane to be exactly in the center of the scene by manipulating the Transform component from the Inspector view. Set the values as you see in the image below.



7) You can rename the plane from the Hierarchy view by selecting the plane and clicking its name. Name it a Floor for example.



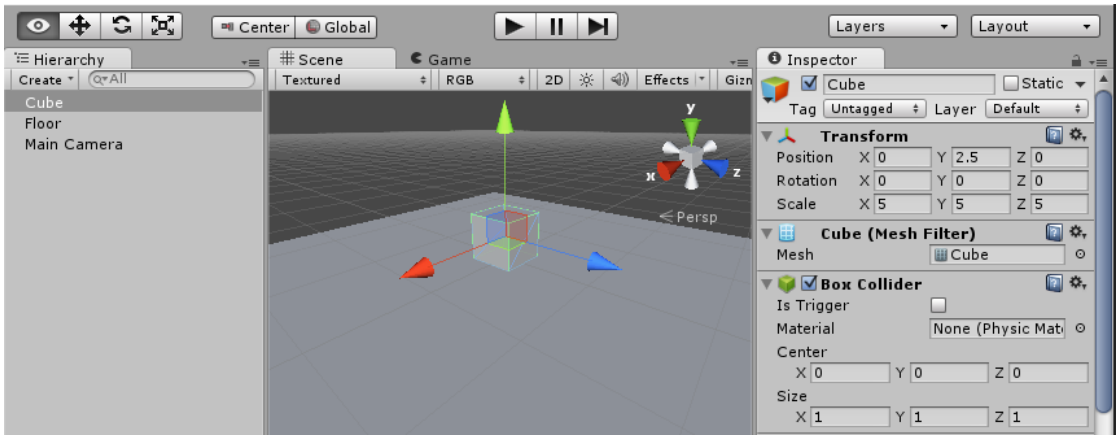
8) Create a cube. You can do it from the GameObject menu or from the Hierarchy views Create menu.

9) Change the scale of the cube from Transform component to

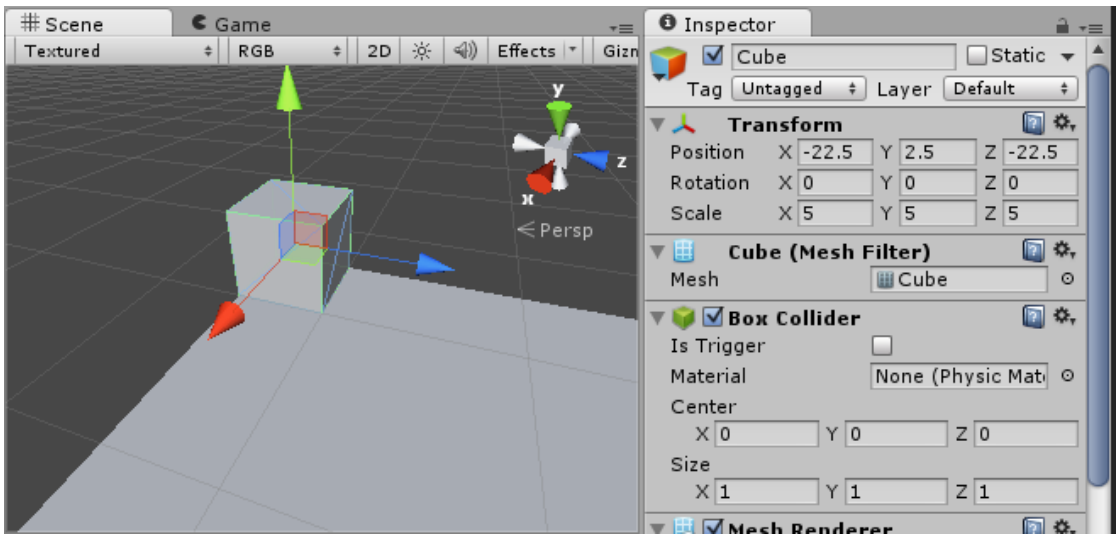
Scale:

- X: 5
- Y: 5
- Z: 5

Notice how the half of the cube is below the plane. You can fix this by changing the cubes Y position to 2.5. Your scene should look like in the image below.



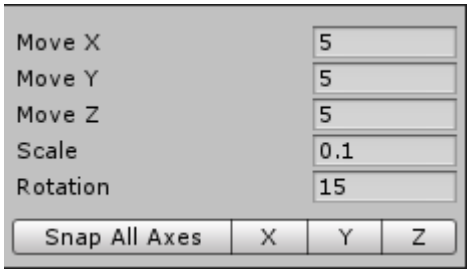
10) Now take the Position tool (W) from the upper left corner of the interface. Use it to move the cube near the corner. When the cube is near the corner, you can just change the position from the Transform component so that the cube is exactly in the corner. You can see the exact values from the image below.



11) Now we're going to make three more cubes so that each corner has a cube. Right click the cube in the Inspector view and click Duplicate. You can do this also by pressing CTRL + D.

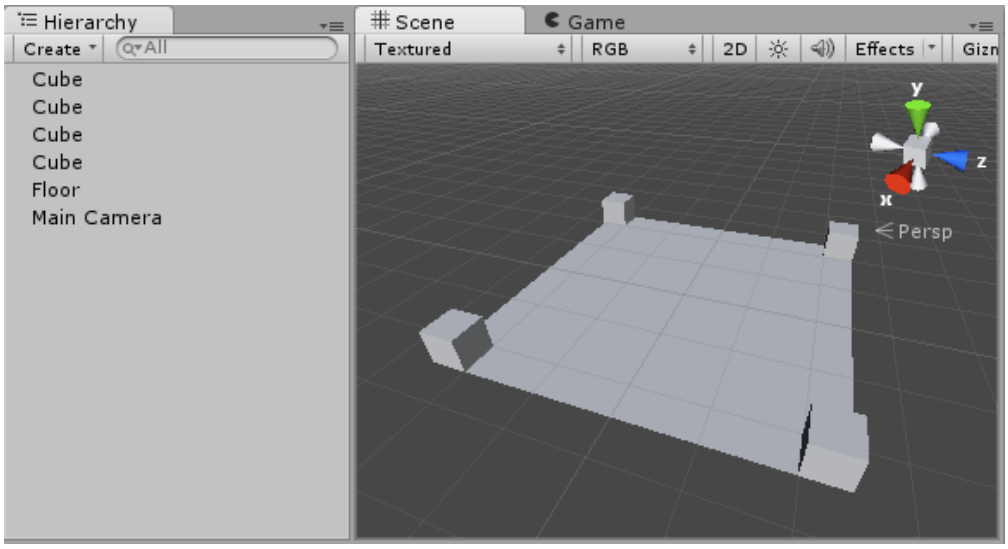
12) There is a new cube in the Inspector view now, but you cannot see it in the scene view. That's because the cube is in the same position as the cube you just duplicated it from. You could change its position with the Position tool or by changing the values from Transform component but let's use snapping this time.

Click the Edit menu and then click the last option called Snap Settings. Change the move X, Y and Z values to 5.

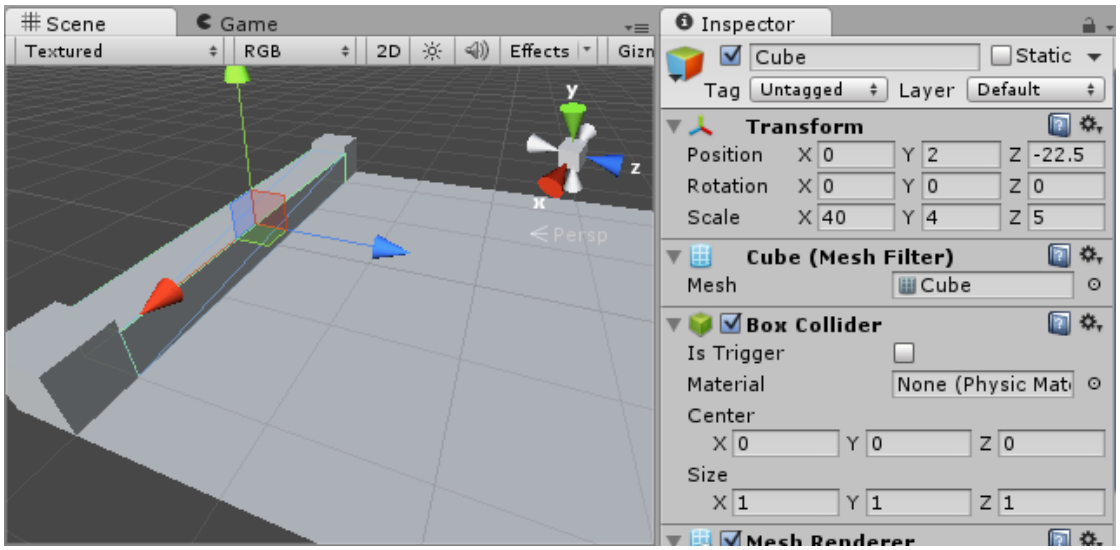


Select the Position moving tool from the upper left corner (W) and press CTRL and move the cube. Notice how the cube now moves 5 units per step.

Move the cube to another corner, duplicate it and repeat this so you have one cube in every corner. The scene should look like in the image below.



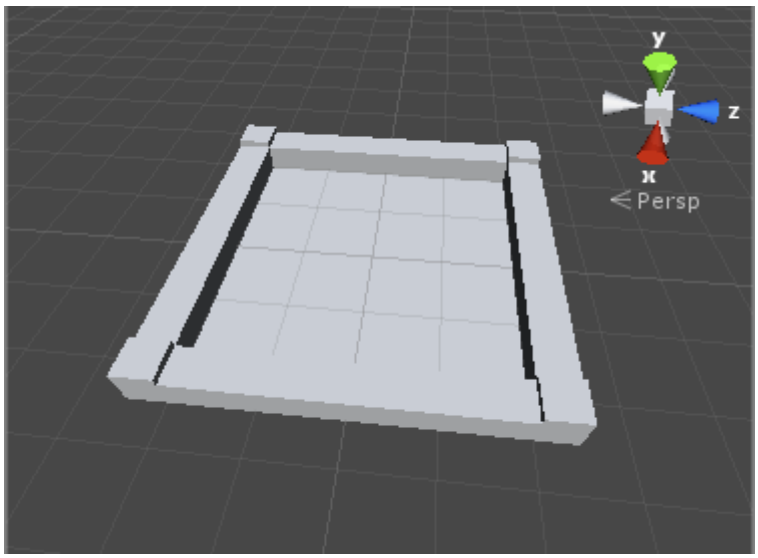
13) Make a new cube. Modify the values from the Transform component like in the image below:



Notice how you need to modify the Y position from 2.5 to 2 because the Y scale is now one unit smaller.

14) Open the snap settings again and set the Rotation to 45. Now duplicate the cube, choose the Rotation tool (E), press CTRL and rotate the cube 90 degrees. The cube should now rotate in 45

degree steps. Place the cube between two other corner cubes, duplicate and repeat so you have a scene like in the image below.



Now we have a new project and a scene with a “sandbox” in it. We can use this project and scene in our next tutorials too.

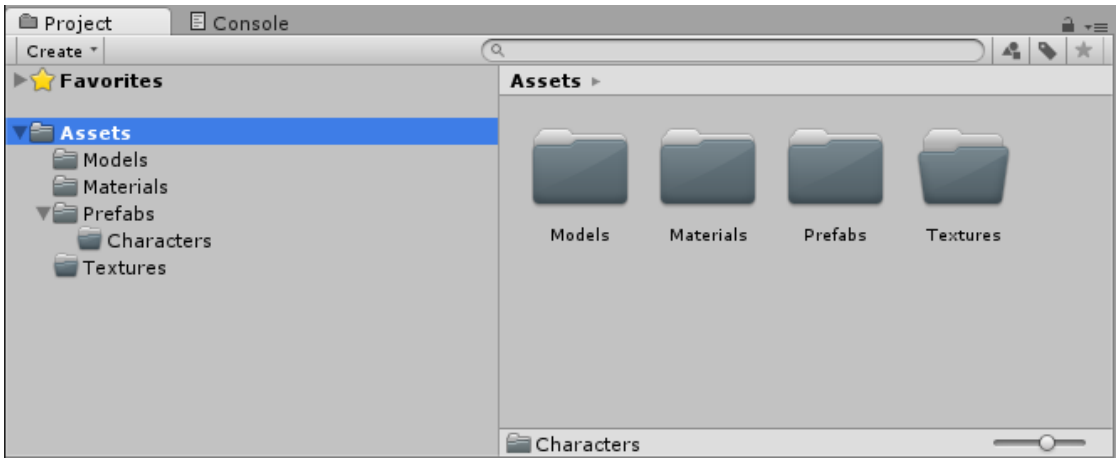
Appendix 2

For contact lesson: Assets, GameObjects and Components

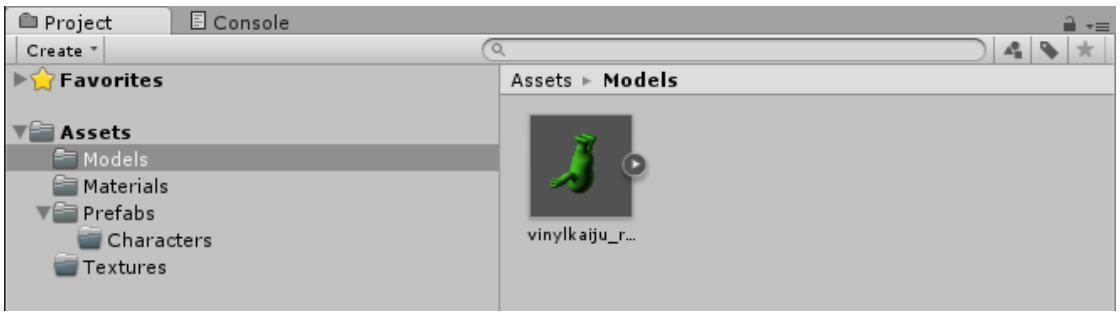
In this tutorial we are going to make a simple character for our game. We will learn how to import an asset, create a GameObject, add components and make a prefab of our character.

Use the project we made in the last tutorial or create a new one.

1) First let's make some subfolders in our asset directory. Make folder for characters, models, materials, textures and scripts.

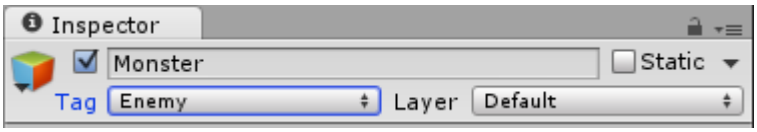


2) Click the project folder with right mouse button and click 'Import New Asset'. Now select the monster.fbx(???) file. Move your model and the green material to correct folders if they aren't already.

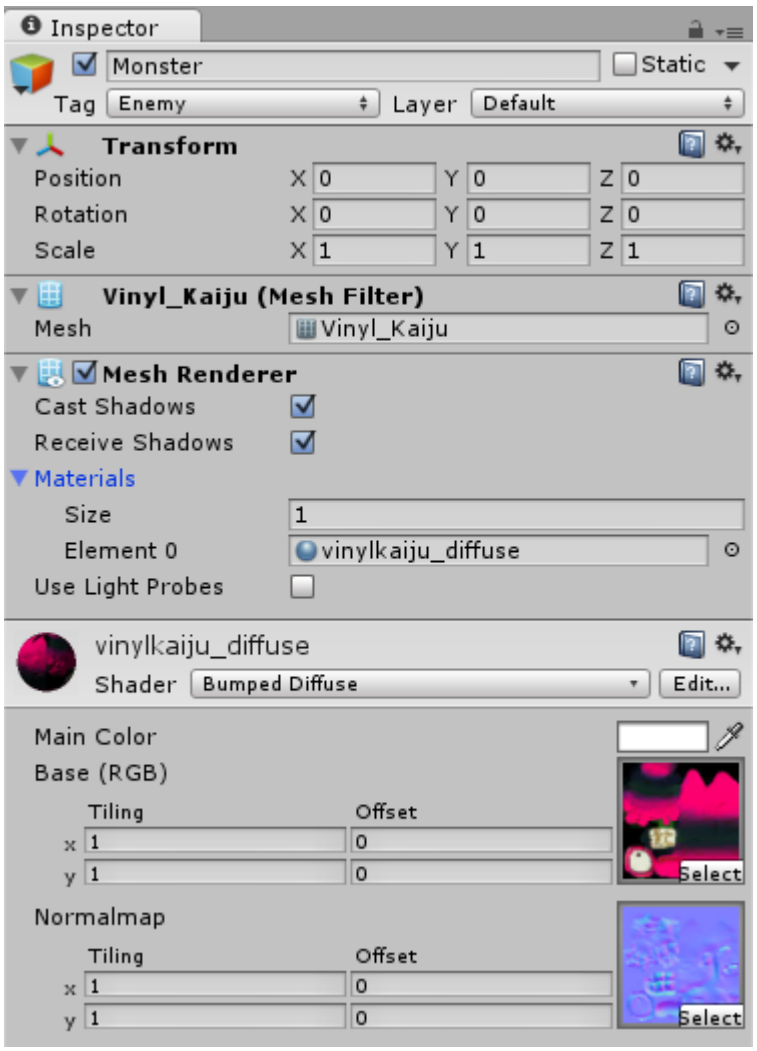


3) Now import the monster_diffuse.png and the monster_normal.png to the Textures folder.

4) Create an empty GameObject. Let's call it Monster and give it a tag. This character will be an enemy character in the game, so let's tag it 'Enemy'. Open the Tag dropdown menu and choose 'Add Tag'. This opens the tag editor. Write 'Enemy' in the element field and press enemy. Now return to the GameObject and select the 'Enemy' tag. Now we should have an empty GameObject called Monster.



5) Select the Monster and add two components, Mesh Filter and Mesh Renderer. In the mesh filter component, add the monster mesh you imported earlier. In the mesh renderer component select the bumped diffuse shader and place the monster_diffuse and monster_normal maps like in the picture. There are lot of different options here, but we're going to learn more about those later.

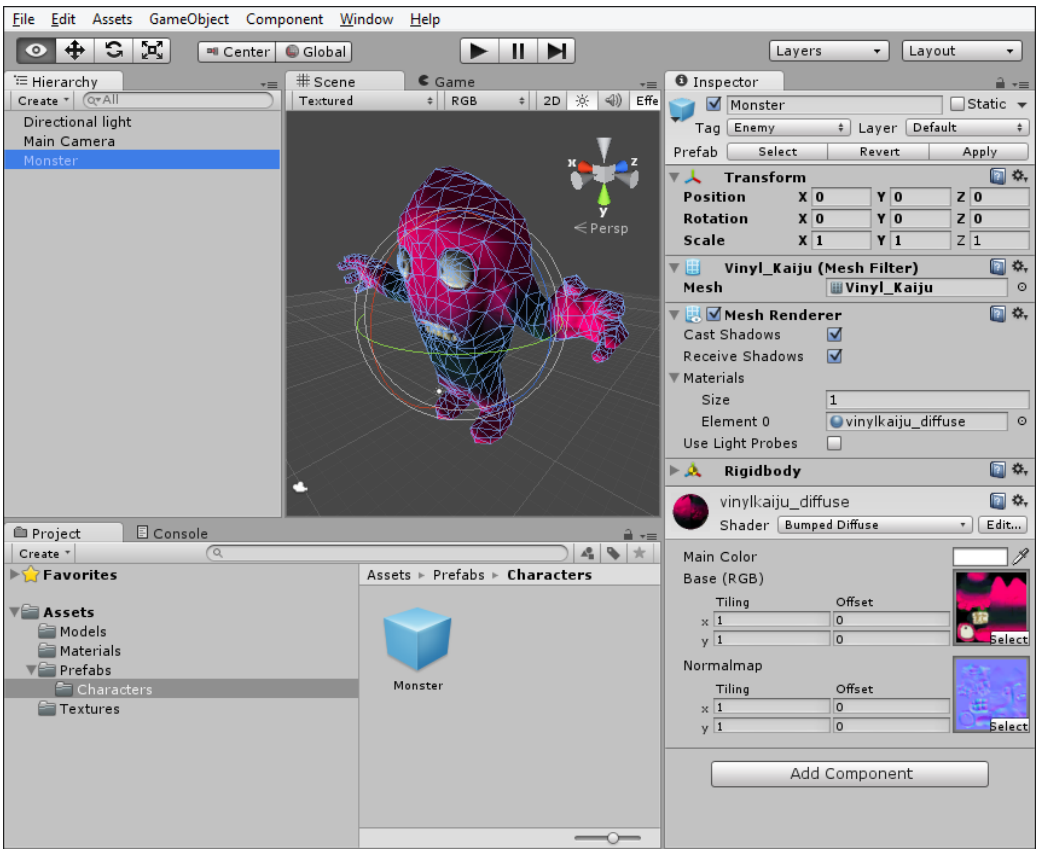


6) Add another component called Rigidbody. It is a physic component. There are lot of options here too but we'll learn more about them later. Let's just let this component be like it is.

7) Now we have a monster GameObject and we want to make it a prefab. You can just drag the Monster GameObject from the Hierarchy view to the Project view in the Characters folder. The Monster GameObjects name in the Hierarchy view should now be blue, indicating that it is now an instance of the prefab we made.

Extra exercise: Use empty GameObjects and tidy up the project

We're ready! The project should look something like this now.


File Edit Assets GameObject Component Window Help
Center Global
Layers Layout
Hierarchy
Create All
Directional light
Main Camera
Monster
Scene Game
Textured RGB 2D
Persp
Inspector
Monster
Static
Tag Enemy Layer Default
Prefab Select Revert Apply
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Vinyl_Kaiju (Mesh Filter)
Mesh Vinyl_Kaiju
Mesh Renderer
Cast Shadows
Receive Shadows
Materials
Size 1
Element 0 vinylkaiju_diffuse
Use Light Probes
Rigidbody
vinylkaiju_diffuse
Shader Bumped Diffuse Edit...
Main Color
Base (RGB)
Tiling Offset
Select
Normalmap
Tiling Offset
Select
Add Component
Project Console
Create
Favorites
Assets
Models
Materials
Prefabs
Characters
Textures
Assets ▸ Prefabs ▸ Characters
Monster

Appendix 3

For contact lesson: Assets, GameObjects and Components

Import at least one asset per type. You can use your own materials too if you want.

- 3D model
- Texture
- Sound
- Text

Now make new Gameobjects. Use the materials you imported, try all of these components and mess with the variables in inspector window:

- Mesh renderer
- Camera
- Lights
- Physics
- Colliders

Then make a scene where you use all of these elements.

Provide an example scene

Appendix 4

For contact lesson: Assets, GameObjects and Components

Describe:

- What is ’prefab’?
- GameObject and Prefab. What are the similarities and the differences between these two?
- Asset and Prefab. What are the similarities and the differences between these two?

Appendix 5

For contact lesson: Scripting

Making a script in Unity is simple. The user can simply right click on the project view and create a C# script. The script should appear in the project view and the user can name it.

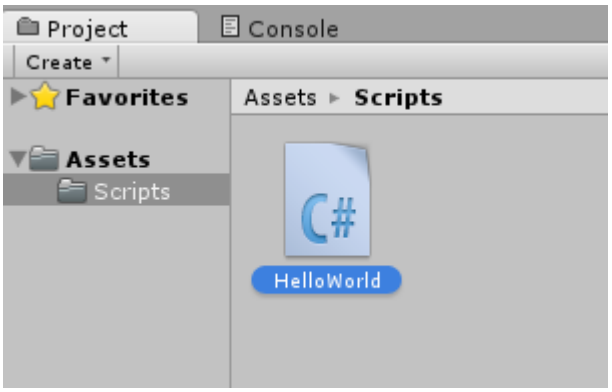


Figure 26. Script in the Project View

The following script creates a simple “hello world” alert to the console.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class HelloWorld : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
8         Debug.Log ("Hello world!");
9     }
10
11    // Update is called once per frame
12    void Update () {
13
14    }
15 }
16
```

Figure 27. HelloWorld.cs

Now the script should be saved and the IDE can be closed. Now the user should have a working script file in the project.

Now the script should be attached to the scene. Create an empty GameObject and name it GameLogic (or something similar). Now drag the script file on the GameObject in the hierarchy view.

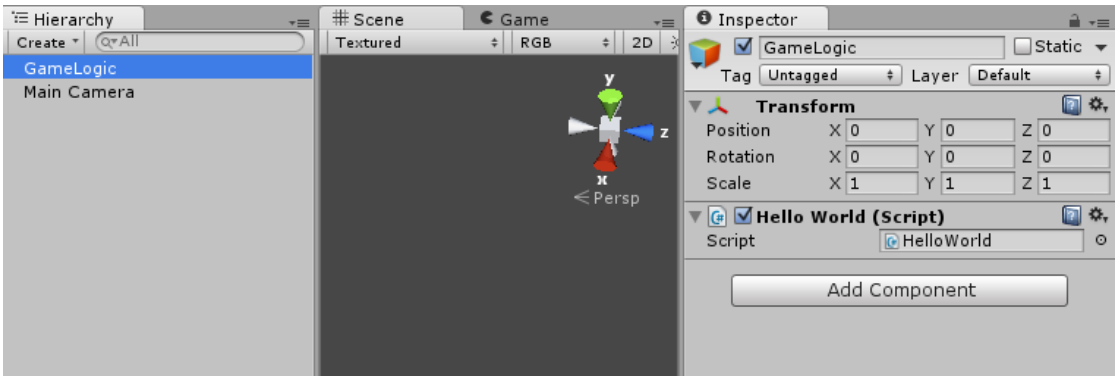


Figure 28. GameLogic GameObject containing the script.

Now you should see the script attached to the Gameobject in the Inspector View. If you run the game now and open the Console from the Project View tab you should see this.

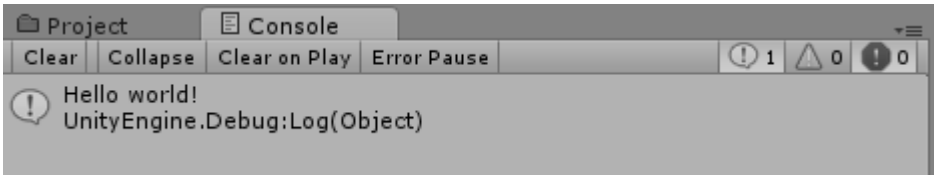


Figure 29. HelloWorld.cs functionality.

This is the simplest way to create and run scripts in your game.

Appendix 6

For contact lesson: Scripting

Let's make an example game where we use cannonballs to break a wall. I this tutorial you will learn some basics of scripting and instantiating prefabs.

1. Start a new project or use the previous project. If you choose to use the previous project, make a new empty scene.

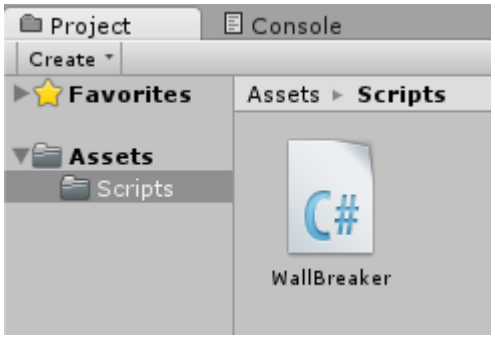
2. Create a plane.

   Position: X = 0, Y = 0, Z = 0

   Scale: X = 2, Y = 1, Z: 2

3. Create a directional light. You do not need to adjust anything, just create it and leave it be

4. Now make a new folder in your Project view and call it "Scripts". Then make a new C# script there and name it 'WallBreaker'



5. Now open the script in MonoDevelop. Let's start by making five public variables:

   - brickPrefab = a variable for the brick prefab we're going to make later.

   - ballPrefab = a variable for the ball prefab we're going to make later

- numBricksX = a variable for the amount of bricks in X axis
- numBricksY = a variable for the amount of bricks in Y axis
- forceMultiplier = a variable for the force applied to the ball we're going to shoot

```
using UnityEngine;
using System.Collections;

public class WallBreaker : MonoBehaviour {

    public GameObject brickPrefab;
    public GameObject ballPrefab;

    public int numBricksX = 5;
    public int numBricksY = 5;

    public float forceMultiplier = 100;

    void Start () {

    }

    void Update () {

    }
}
```

6. Now let's make some private variables which does not need to be displayed and modified from the Inspector view in Unity.
    - X, Y, Z = variables for positions in X, Y and Z axis
    - brickOrientation = a variable for bricks orientation also known as rotation values
    - ray = a variable for ray
    - ball = a variable for an instance of the ballPrecfab

```
private float X;
private float Y;
private float Z;

private Quaternion brickOrientation;

private Ray ray;
private GameObject ball;
```

7. Now let's make sure that the cursor is visible in our game. We can use the Awake() loop for this.

```
void Awake () {
    Screen.showCursor = true;
}
```

8. Now let's start making the functionality. Write all of this inside the Start() loop:

```
void Start () {

    if ( brickPrefab != null )
    {
        var brickSize = brickPrefab.renderer.bounds.size;

        X = brickPrefab.transform.position.x + brickSize.x;
        Y = brickPrefab.transform.position.y;
        Z = brickPrefab.transform.position.z;

        brickOrientation = brickPrefab.transform.rotation;

        for( var i = 0; i < numBricksY; ++i )
        {
            for ( var j = 0; j < numBricksX; ++j )
            {
                if ( i == 0 && j == ( numBricksX - 1 ) ) break;
                Instantiate(brickPrefab, new Vector3(X, Y, Z), brickOrientation);
                X += brickSize.x;
            }
            X = brickPrefab.transform.position.x;
            Y += brickSize.y;
        }
    }

}
```

The first if condition checks if the ballPrefab variable has a prefab selected in it.

Then the X, Y and Z variables are given their right values. Notice how the bricks size is applied to the X variable.

brickOrientation variable is given its value from the rotation values of the brickprefab.

The next for-loop will create the wall with the instances of the brick prefab.

9. Then we're going to the Update() loop. Write this code inside it:

```
void Update () {

    if ( Input.anyKeyDown && ballPrefab != null )
    {
        ray = Camera.main.ScreenPointToRay( Input.mousePosition );
        ball = Instantiate( ballPrefab, ray.origin, Quaternion.identity ) as GameObject;
        ball.rigidbody.AddForce( ray.direction * forceMultiplier );
    }
}
```

The first if condition checks if there any button pressed down and if the ballPrefab variable is set.

Ray variable is now set with the vector from the cameras view to the mouse position.

The ball variable is now set with the instance of the ball prefab.

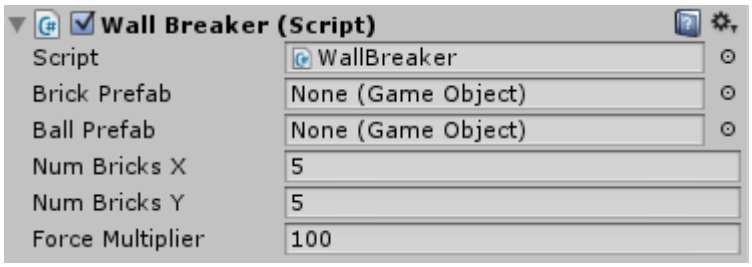
Then the instance gets force in the direction of the ray vector.

10. Now the script is ready and it should look like this:

```
using UnityEngine;
using System.Collections;

public class WallBreaker : MonoBehaviour {

    public GameObject brickPrefab;
    public GameObject ballPrefab;

    public int numBricksX = 5;
    public int numBricksY = 5;

    public float forceMultiplier = 100;

    private float X;
    private float Y;
    private float Z;

    private Quaternion brickOrientation;

    private Ray ray;
    private GameObject ball;

    void Awake () {
        Screen.showCursor = true;
    }

    void Start () {

        if ( brickPrefab != null )
        {
            var brickSize = brickPrefab.renderer.bounds.size;

            X = brickPrefab.transform.position.x + brickSize.x;
            Y = brickPrefab.transform.position.y;
            Z = brickPrefab.transform.position.z;

            brickOrientation = brickPrefab.transform.rotation;

            for( var i = 0; i < numBricksY; ++i )
            {
                for ( var j = 0; j < numBricksX; ++j )
                {
                    if ( i == 0 && j == ( numBricksX - 1 ) ) break;
                    Instantiate(brickPrefab, new Vector3(X, Y, Z), brickOrientation);
                    X += brickSize.x;
                }
                X = brickPrefab.transform.position.x;
                Y += brickSize.y;
            }
        }

    }

    void Update () {

        if ( Input.anyKeyDown && ballPrefab != null )
        {
            ray = Camera.main.ScreenPointToRay( Input.mousePosition );
            ball = Instantiate( ballPrefab, ray.origin, Quaternion.identity ) as GameObject;
            ball.rigidbody.AddForce( ray.direction * forceMultiplier );
        }
    }
}
```

Save it and get back to Unity.

11. Now that we have the script ready we need to add it in the scene. Create a new empty GameObject, name it GameLogic for example and drag the script on it.



Now you should see the script in the inspector window. Notice how the public variables are now adjustable.

12. Now we need to create prefabs for the brick and the ball.

Create a cube and give it scale of: X = 1, Y = 0.5, Z = 0.5. Also give it position of: X = 0, Y = 0.25, Z = 0.

Now add a Rigidbody physic component to the cube.

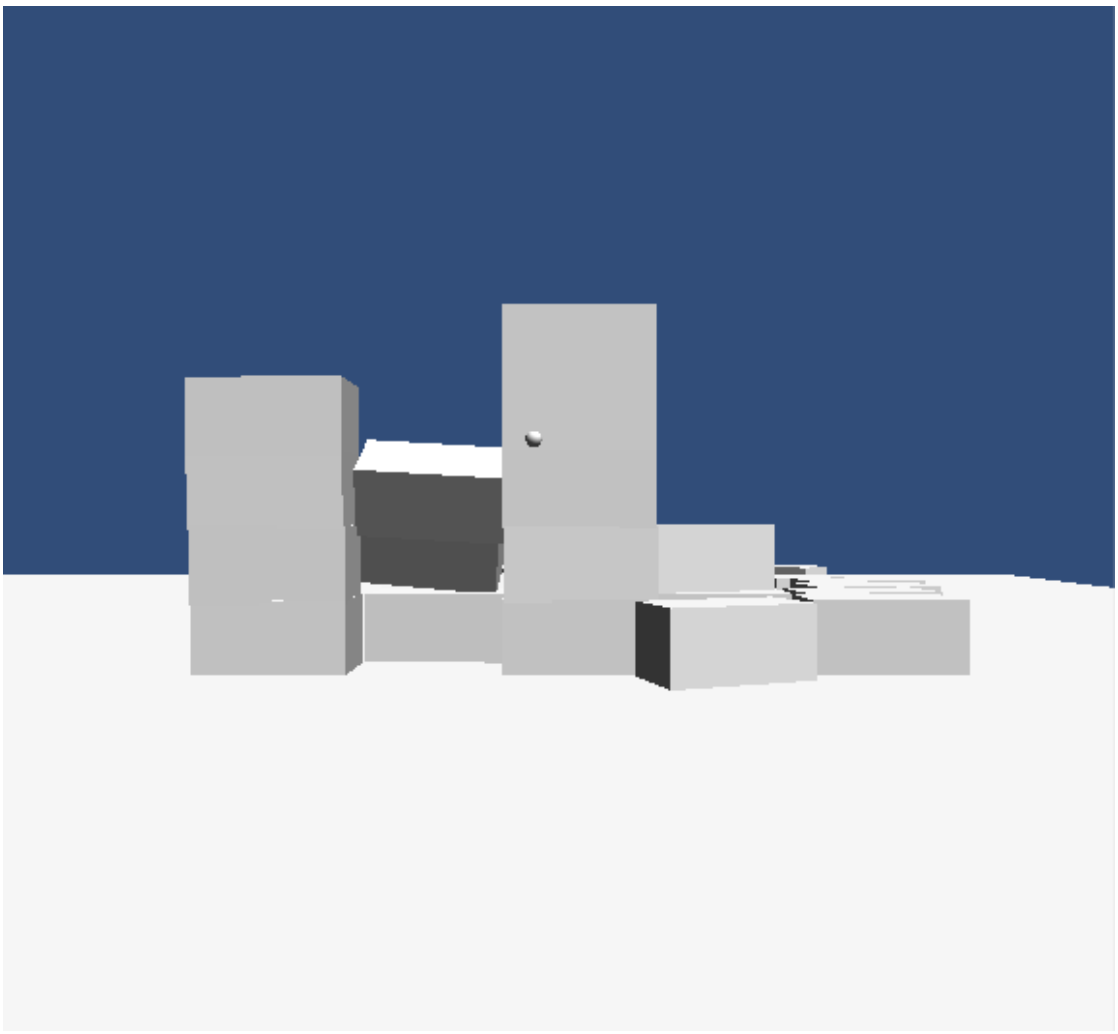
Now just drag the cube from Hierarchy view to the Project view and you have a prefab of the brick. Remember to save your scene!

13. Now create a sphere and add the Rigidbody physic component to it. Drag the sphere in your project view to create a prefab of the cannonball.

Delete the cube and ball from the scene because you only need to have the prefabs in your assets.

14. Now when you add the prefabs to the script in the Inspector view, you should be able to play the game. You may need to adjust the position of the main camera and try a bigger value for the forceMultiplier variable.

The game should look like this:



In this tutorial you learned how to make variables and how you can adjust them in Inspector view, how to instantiate objects and how to use game loops.

Appendix 7

For contact lesson: Scripting

Describe:

- What are the differences between Update() and FixedUpdate() loops?
- What does ‘Time.deltaTime’ do?
- Instantiate?
- Enumeration? (shortly)
- Quaternion? (shortly)

Appendix 8

For contact lesson: Physics

Make a simple platformer where you will use:

- Rigidbodies
- Physic Materials
- Colliders
- Character controller
- Scripting (onCollisionEnter/onTriggerEnter functions, Application.LoadLevel, etc..)

Play around with character controller inspector view. Make steep platforms and see if you can climb them or not.

Appendix 9

For contact lesson: Physics

Describe and give an example usage.

- The difference between rigidbody and kinematic rigidbody.
- The difference between rigidbody and physic material.

When would you use the following collider and why? Give an example.

- Cube collider
- Mesh collider
- Compound collider
- Static collider?
- Trigger collider?

Appendix 10

For contact lesson: Rendering

Use the platformer scene we made last time, or create a new scene. Fill the scene with various objects and then practice making new materials with different shaders and assign them to the objects.

Practice the lighting using both dynamic and baked lights in your scene.

In the end you can play with the camera’s properties.

Appendix 11

For contact lesson: Physics

Describe and give an example when would you use them:

- What are 'perspective' and 'orthographic' render modes and how do they differ?
- What is dynamic lighting, when is it used and how does it differ from baked lights?
- What is baked lighting, when is it used and how does it differ from dynamic lights?