FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Ressourcen zur Unterstützung beim Erlernen von “Unity” zur Entwicklung von 3D-Anwendungen

MAGISTERARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Alexander Wagner
Matrikelnummer 0227425

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Ass.Prof. Dipl.-Ing. Dr.techn. Monika Di Angelo

Mitwirkung
Ass.Prof. Dipl.-Ing. Dr.techn. Peter Ferschin

Wien, 31.08.2011

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Resources to support students in learning the 3D-application development tool “Unity”

MAGISTERARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Alexander Wagner
Matrikelnummer 0227425

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Ass.Prof. Dipl.-Ing. Dr.techn. Monika Di Angelo

Mitwirkung
Ass.Prof. Dipl.-Ing. Dr.techn. Peter Ferschin

Wien, 31.08.2011

Erklärung zur Verfassung der Arbeit

Alexander Wagner
Guttmannstraße 14
A-2540 Bad Vöslau

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Bad Vöslau, 31.08.2011

Acknowledgments

I want to thank Professor Monika Di Angelo and Professor Peter Ferschin for guiding me through my master thesis and allowing me to do research in their university courses.

I also thank the students from these courses who provided me with feedback and insights in the challenges one has to face when beginning to learn a new tool.

Last but not least, I want to gratefully thank my parents for supporting me in my studies.

Abstract

Interactive 3D-applications are software programs that can react immediately to input from its user and display the results of these inputs with three-dimensional graphics in real time. Typical application areas for 3D-programs are multimedia-games, software for visualizing architecture or interactive learning software. The development of interactive 3D-applications is a very challenging area of software development, because it requires a high level of knowledge and implies lots of complexities.

These complexities can significantly be reduced through the usage of appropriate tools. Such a tool for developing interactive 3D-applications is "Unity". It supports to create 3D-applications by providing a graphical user interface and a built-in 3D-engine that hides the mathematical complexities of computer graphics from the user. Because of its great usability and the enormous speed one can produce results, it is a great tool to use for teaching the basics of 3D-application development to beginners. But despite these simplifications the development of interactive 3D-applications with Unity is still not a trivial task.

The aim of this work was to create useful resources that facilitate to learn the usage of Unity for beginners. The target audience consisted primarily of students in computer science (focused on game development) and architecture (focused on architectural applications). The analysis of the special requirements of these groups was conducted through qualitative research in two different university courses.

With this background instructions and tutorials have been created that describe the various aspects of working with Unity that are important for beginners. In addition to descriptions of the tool itself and how to use it, aspects of collaboration and team work in a development project within university context are discussed.

All resources should be made easy available through a central repository. An evaluation was conducted to determine the best way in which this repository should be realized.

Abstract (German)

Interaktive 3D-Anwendungen sind Programme, die unmittelbar auf die Eingaben ihres Benutzers reagieren und die Ergebnisse dieser Eingaben mittels dreidimensionaler Grafiken in Echtzeit darstellen können. Typische Anwendungsgebiete von 3D-Programmen sind multimediale Spielsoftware, Software zur Erkundung architektureller Entwürfe oder interaktive Lernsoftware. Die Entwicklung von interaktiven 3D-Anwendungen ist ein besonders anspruchsvolles Gebiet der Softwareentwicklung, da es ein hohes Maß an Vorwissen benötigt.

Dieses notwendige Vorwissen kann durch den Einsatz geeigneter Tools deutlich reduziert werden. Ein solches Tool zur Entwicklung interaktiver 3D-Anwendungen ist „Unity“. Es erlaubt die einfache Erstellung eines 3D-Projekts mittels graphischem Interface und enthält eine integrierte 3D-Engine, die die mathematischen Komplexitäten der Computergraphik vor dem Nutzer verbirgt. Aufgrund seiner leichten Zugänglichkeit und der enormen Geschwindigkeit, mit der man sichtbare Ergebnisse erzielen kann, eignet es sich hervorragend, um die Grundlagen von 3D-Anwendungs- und Spieleentwicklung zu lehren. Doch trotz dieser Vereinfachungen bleibt die Entwicklung von interaktiven 3D-Anwendungen auch mit Unity ein sehr komplexes Gebiet.

Ziel dieser Arbeit war, sinnvolle Ressourcen zu erstellen, die Anfängern den Einstieg in den Umgang mit Unity erleichtern und diesen beschleunigen sollen. Die betreffende Zielgruppe bestand primär aus StudentInnen der Informatik (mit Fokus auf Spieleentwicklung) und der Architektur (mit Fokus auf architekturelle 3D-Anwendungen). Die Erhebung der besonderen Ansprüche dieser Zielgruppen erfolgte mittels qualitativer Untersuchungen bei zwei universitären Lehrveranstaltungen.

Auf dieser Basis wurden Anleitungen bzw. Tutorials erstellt, die die verschiedenen Aspekte der Arbeit mit Unity beschreiben, die für Anfänger besonders wichtig sind. Neben Beschreibungen des Tools an sich und des Umgangs mit diesem werden darin auch Aspekte der Zusammenarbeit im Team bei einem Entwicklungsprojekt im Universitätskontext erörtert.

Alle erstellten und gesammelten Ressourcen sollten durch eine zentrale Sammelstelle schnell und einfach verfügbar gemacht werden. Eine Evaluierung wurde durchgeführt, um die günstigste Art und Weise zu bestimmen, auf die diese Sammelstelle umgesetzt werden soll.

Table of Contents

1 Introduction

Technology in the field of 3D-software has evolved in a way that allows the rendering of countless objects in real-time, while still being able to perform background processes. The software can react to user input in a split second, creating the impression of real interactivity.
Because of these advances interactive 3D-applications today are being used as a medium in plenty of different areas such as:

- Interactive 3D-software can be used to educate children in a fun and exciting way.
- It can serve as an interface to ease the control of a complex device, for example a drill machine or a surgical instrument.
- Interactive 3D-applications can provide help when planning the construction of a building.
- They can also suit pure entertainment purposes by simulating an interactive gaming scenario for the player.

These are just some examples that shall help to get a picture of the countless possibilities interactive 3D-applications can be used for.

Nevertheless, the development of such a software program is anything but trivial. To achieve the illusion of three dimensions on a two-dimensional display, a mathematical model that represents the current scene has to be projected onto the screen. The colors and textures are influenced by light sources, gravity and physics may have to be applied. The changes in the scene from the current frame to the next frame need to be calculated, user input processed. And all changes have to be applied machine-independent (meaning that an object must not cover more distance in the same amount of time on a faster machine).
Implementing a framework that handles all these issues (a so-called “3D-engine”) is a task that can only be performed by experts in the field of computer graphics. However there are lots of other stakeholders who might be interested in the development of 3D-applications, but do not provide this high level of expertise in computer graphics: Game designers, persons specialized in building science, or just in another area of computer science, teachers,…

Luckily there are prefabricated tools that can handle many (but not all!) of the tasks one would have to fulfill by himself when developing a 3D-application from scratch. This makes the development much easier and accessible to a wider range of people.
But even though such tools can hide many of the complexities that need to be conquered when dealing with 3D computer graphics, it is still quite a challenging task to develop 3D-software using these instruments.

This master thesis deals with a popular tool used to develop 3D-applications named “Unity” [Unity]. The problems beginners have when learning Unity are analyzed and resources for teaching it to students are provided.

1.1 Interactive 3D-applications

In this context, an interactive 3D-application is defined as a piece of computer software that is able to render three-dimensional objects in real-time onto a screen, recognize some kind of user input (though this is not always the case), and process it to apply changes to the objects in the scene.
To create the illusion of motion, this rendering process has to happen multiple times per second, usually at above 16 to 18 times [Wikipedia, “Bewegte Bilder”]. The number of times the screen is rendered is called the “frame rate”, or the “frames per second”. A smooth impression of motion can be experienced at a frame rate of 25, even better at 60 frames per second.

A preliminary question everyone who plans to develop a 3D-application should think about are the advantages that come with the 3D-effect for his specific purpose. These advantages are being exchanged for a probably more complex development process and the need for higher system requirements at the end user’s machine. So if three dimensions are not absolutely necessary for an application to fulfill its purpose, one should consider developing a 2D-application instead.

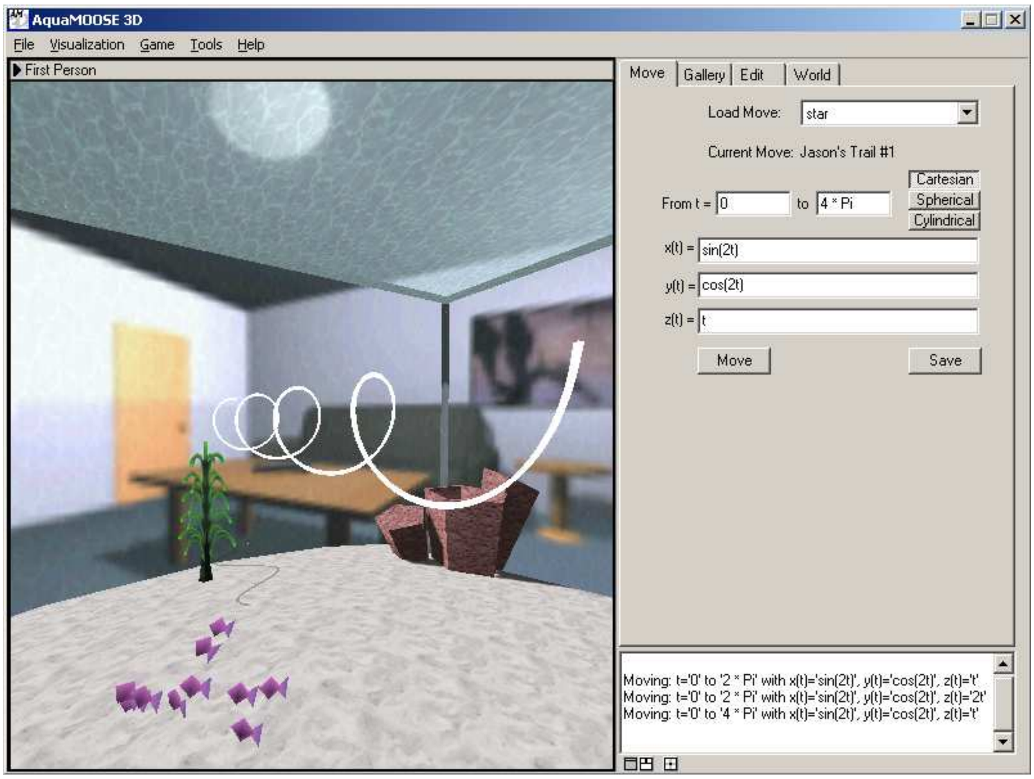
But of course there are certain advantages when it comes to 3D:

- In the majority of cases, 3D-applications are able to resemble real objects in a more natural way than it can be done with 2D-visualizations, also allowing the realistic simulation of physical behavior.

- 3D-applications easily allow the viewing of objects from different perspectives.
- Compared to two dimensions, a three-dimensional environment adds “depth”, therefore providing the developer (and consequently the user) with more degrees of freedom and opportunities. This can also provide a more realistic and intense experience.

By making clever usage of these advantages, one can produce software of high quality. The following examples show what can be done with interactive 3D-applications:

AquaMOOSE 3D is an environment that was investigated by Elliott, Adams and Bruckman [Elliot et. al., 2002]. It was designed to support the teaching of 3D-mathematics, a topic which is very well suited by nature for real-time 3D-visualizations. Elliott et. al. write: “*AquaMOOSE 3D is a graphical environment designed to support free exploration of three-dimensional math concepts. Motion in AquaMOOSE can be specified mathematically, using parametric equations. For example, swimming in a sine wave in x and a cosine in y creates a spiral.*” [Elliot et. al., 2002]



Load Move: star

Current Move: Jason's Trail #1

From t = 0 to 4 * Pi

x(t) = sin(2t)

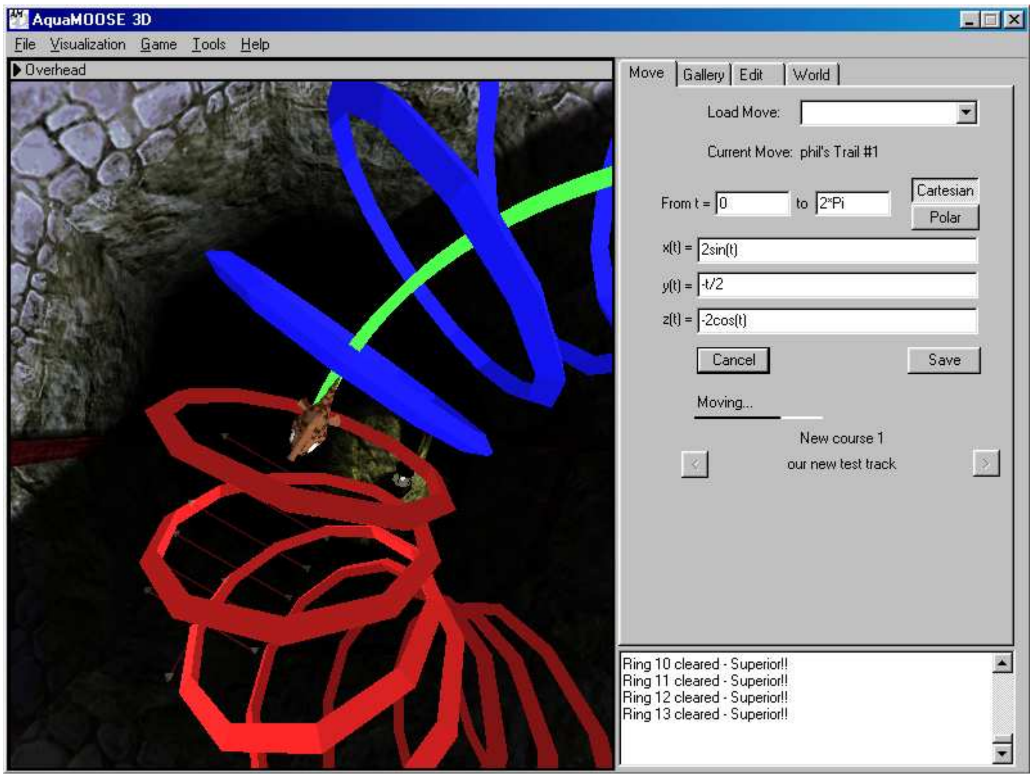
y(t) = cos(2t)

z(t) = t

```
Moving: t='0' to '2 * Pi' with x(t)='sin(2t)', y(t)='cos(2t)', z(t)='t'
Moving: t='0' to '2 * Pi' with x(t)='sin(2t)', y(t)='cos(2t)', z(t)='2t'
Moving: t='0' to '4 * Pi' with x(t)='sin(2t)', y(t)='cos(2t)', z(t)='t'
```

Figure 1: The AquaMOOSE 3D environment (figure from [Elliot et. al., 2002])

They also write about a game they provided for the environment, where the student (or the "player") must define a mathematical function that passes through a certain set of rings:



Load Move:

Current Move: phil's Trail #1

From t = 0 to 2*Pi

x(t) = 2sin(t)

y(t) = -t/2

z(t) = -2cos(t)

Moving...

New course 1

our new test track

Ring 10 cleared - Superior!!
Ring 11 cleared - Superior!!
Ring 12 cleared - Superior!!
Ring 13 cleared - Superior!!

Figure 2: AquaMOOSE 3D - ring game (figure from [Elliot et. al., 2002])

Though Elliott et. al. could not observe that the visual learning method had a significant impact onto the students' test results, they show what can be done with interactive 3D-applications in the area of education.

Another example that shows the advantages of three dimensions is presented by Tong Lu et. al. [Lu et. al., 2005]: They present a method to **reconstruct 3D-models** from 2D-drawings of architectural buildings. They argue that in the field of architecture, many designs are being done via 2D-plans, because of the complexities of editing models in 3D-space. But 3D-models may prove helpful for certain purposes: "*However, for various construction applications, such as quantity surveying, inventory, construction, and visualization, it is necessary to convert the widely used 2D architectural drawings to accurate 3D models.*" [Lu et. al., 2005]

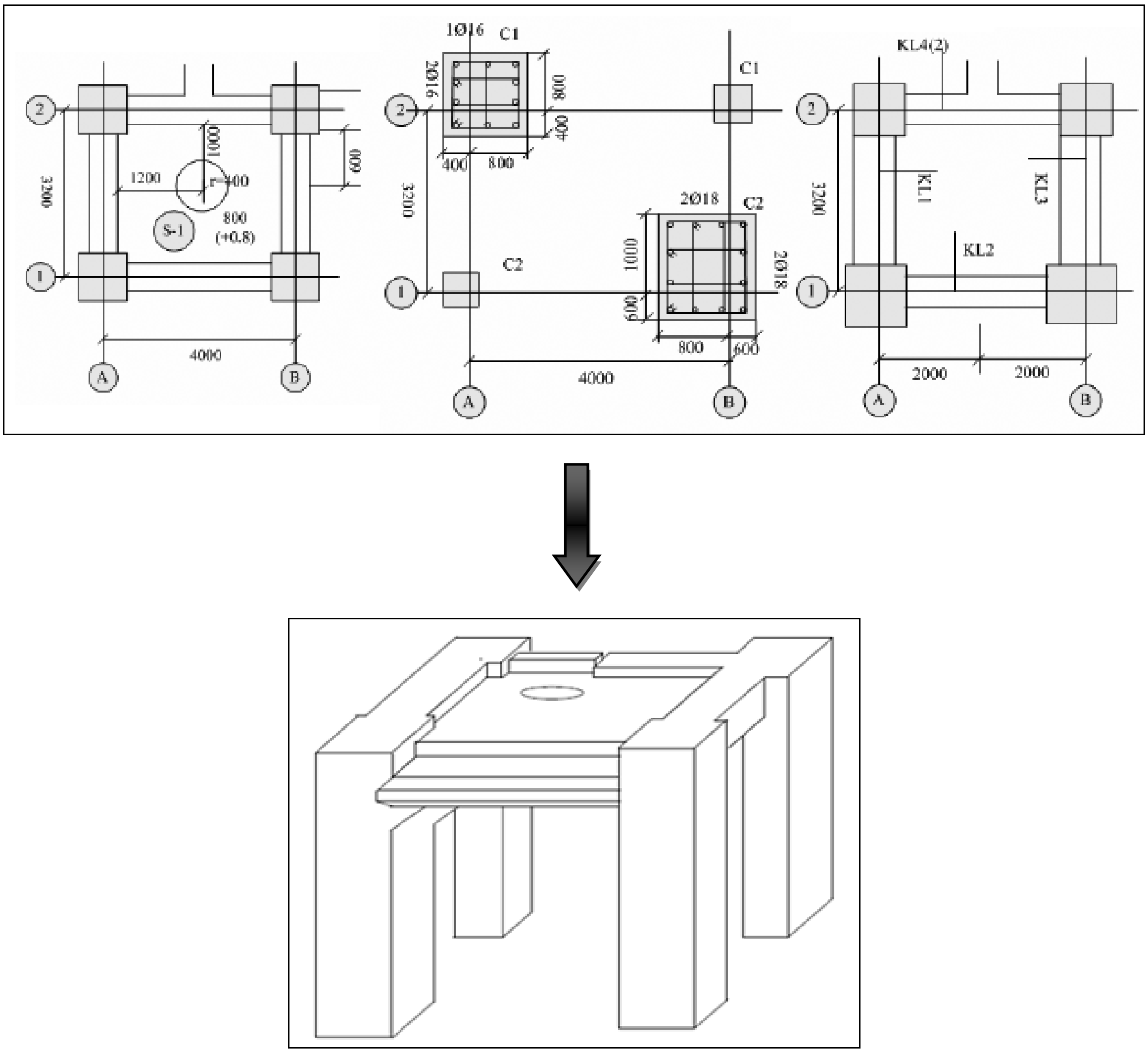


Figure 3: Reconstructing a 3D-model from 2D-plans (figures from [Lu et. al., 2005])

After conversion, such a 3D-model could be used to allow an interactive walkthrough in the building that is planned, allowing a more realistic experience than just by studying a “flat” plan.

Of course one of the most popular applications for interactive 3D-software are **computer games** (also: “video games”). Some products of the modern gaming industry generate higher revenues than blockbuster movies. The constantly increasing hardware performance of computers can be owed to a large extent to the steep rise of requirements for games.
Modern video games have reached a level of realism that has never been seen before. Though currently there is a trend to simpler games that can be played “on the go” (caused especially by the spread of smartphones and the acquisition of new customer groups), realistic games would not be possible without 3D-technology.

There are countless different types of games available. The genres vary from action-oriented games to strategy and puzzle games. But the genre that was primarily responsible for evolving the usage of three dimensions in games and making it popular are “first person shooters” (commonly also called: “ego-shooters”). The name comes from the fact that the player plays the game from the point of view of the virtual character he is controlling. So, analog to the real world, the player just sees the hands of the character (and items they are carrying) – at least when there is no mirror in his view. This causes the player to better identify himself with the character and can intensify the gaming experience. The term “shooter” refers to the fact that in many of those (often violent) games the player’s goal is to shoot enemies with different kinds of weapons – though there are games from first person perspective with other goals as well.

One of the first big cornerstones in the history of first person shooters was “Doom” from 1993, in which the player took on the role of a soldier on a space station and had to fight hordes of evil aliens. Another important example is “Half-Life” from 1998, which added to the standard first-person action a movie-like storyline. Its sequel from 2004 furthermore improved the experience and was one of the first big games to make usage of a physics engine to simulate the realistic behavior of objects. The following screenshots give an impression of the rapid evolution of computer games:

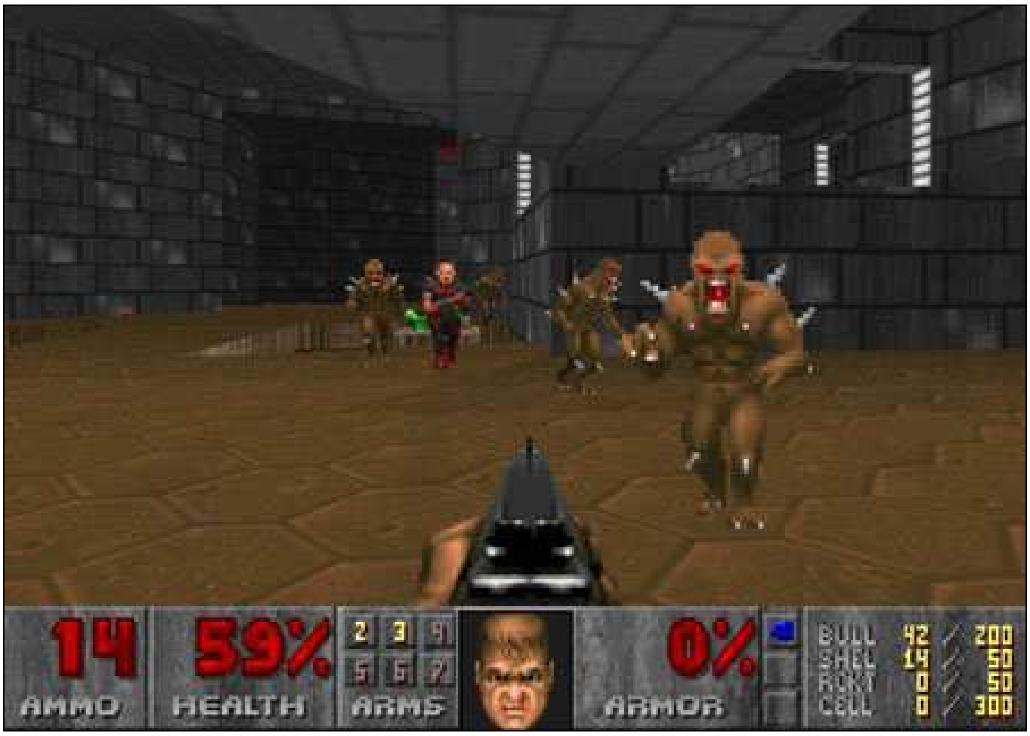


Figure 4: Screenshot of the game "Doom" from 1993 (image from [PC Games, 2010])

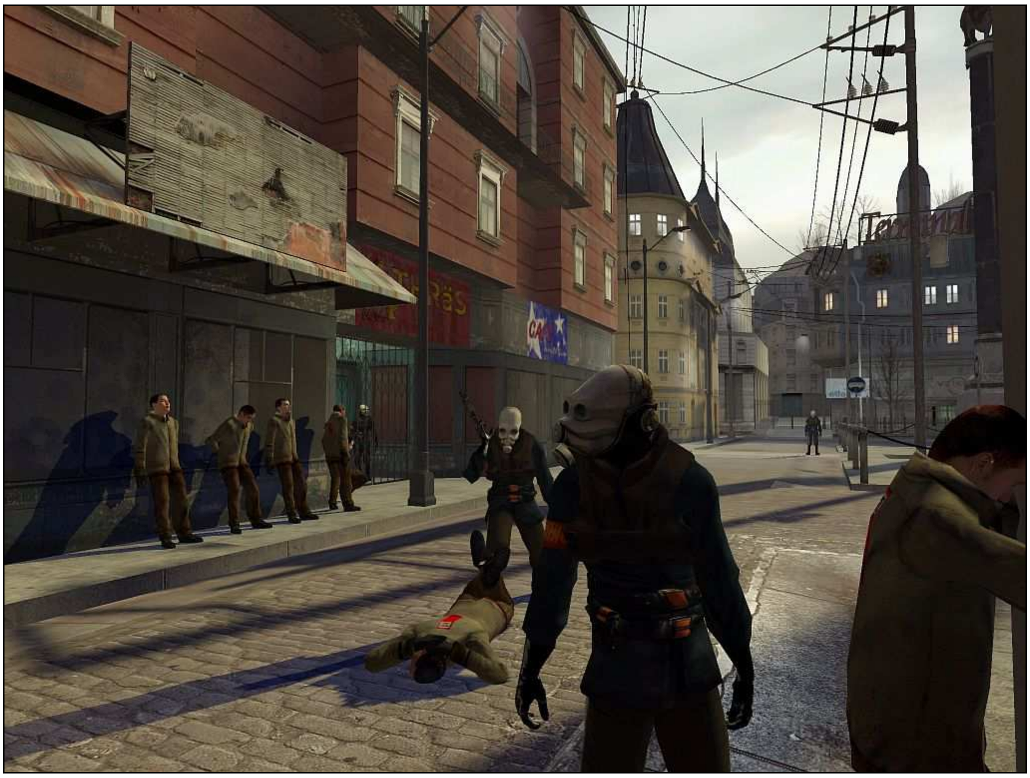


Figure 5: Screenshot of the game “Half-Life 2” from 2004 (image from [golem.de, 2004])

The given examples demonstrate three ways of using interactive 3D-applications for different purposes. Of course there are countless other possibilities that cannot be mentioned here as well.

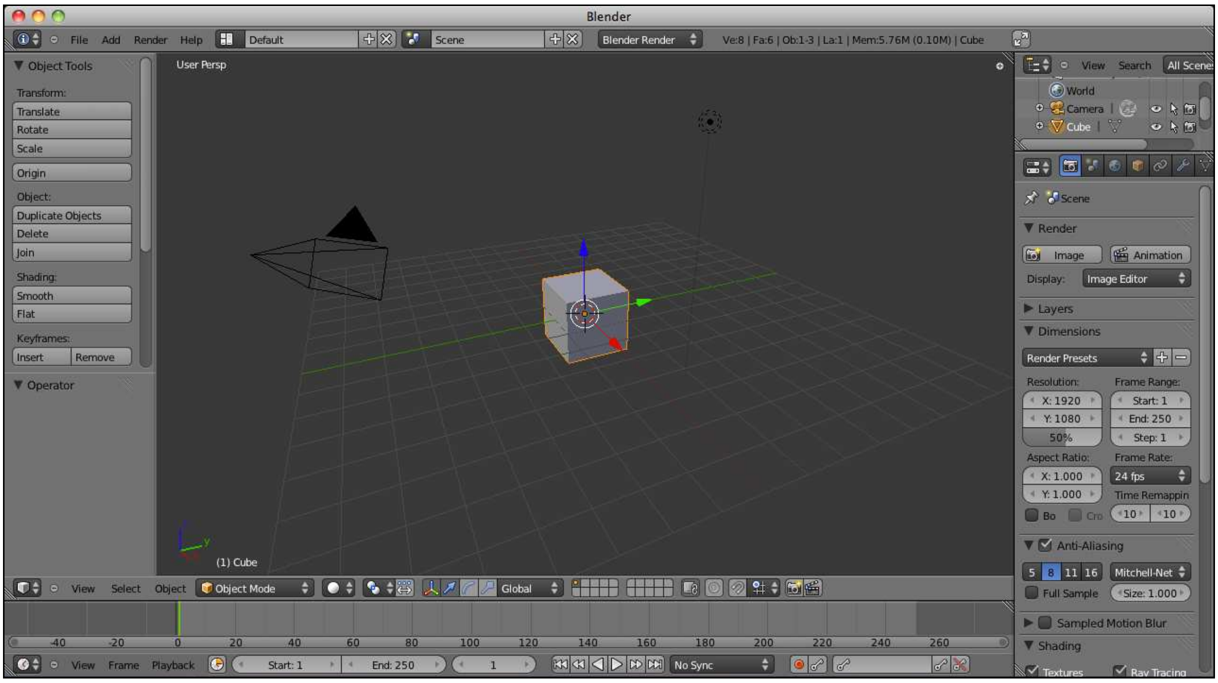
1.2 Development of interactive 3D-applications

Developing a high-quality interactive 3D-application requires expert knowledge, the suitable set of tools and proficient collaboration within a team. To realize such a 3D-application project in a professional way, it is necessary to implement a **structured development process** like it is commonly done in any other software project.

Regarding the tools, there are lots of different software programs available that help in developing a 3D-application and can take over many tasks:

Of course all the objects in a 3D-application have to be defined. The computer must know exactly how they look like, their shape viewed from all angles and what’s their surface like.

Luckily it is not necessary to calculate these objects using mathematical equations, it can be done much easier using **3D-modeling software**. With these tools one can design 3D-objects with a graphical user interface, and instantly preview the created results. Professional software does not only allow to design the shape of objects, but also includes functionality for animation, rendering and texturing. One of the most popular 3D-modeling tools is “Blender”. Due to the fact that it is free open source software, it is very popular among students and 3D-modeling hobbyists.



Display: Image Editor

Resolution: X: 1920 Y: 1080 50%

Frame Range: Start: 1 End: 250 Step: 1

Aspect Ratio: X: 1.000 Y: 1.000

Frame Rate: 24 fps

Time Remapping: 10 10

Anti-Aliasing: 5 8 11 16 Mitchell-Net Size: 1.000

Start: 1 End: 250 1

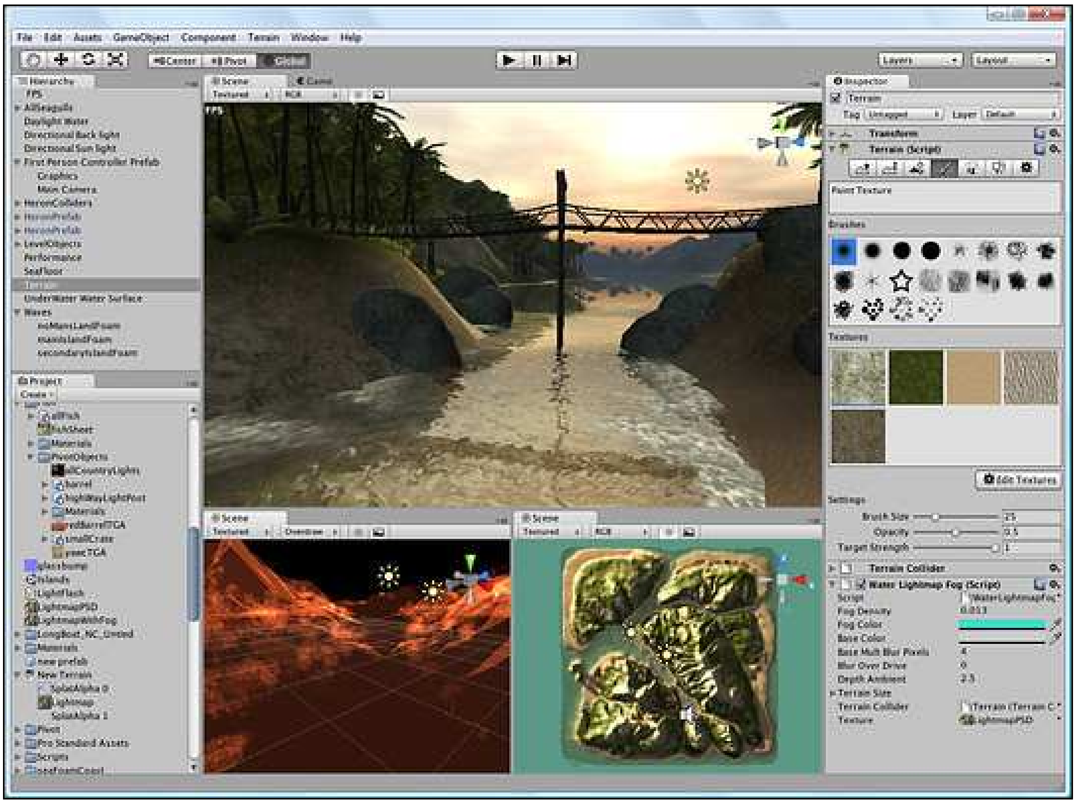
Figure 6: Screenshot from "Blender" (figure from [Wikipedia, “Blender”])

Another important asset when developing 3D-software are so-called “engines”:

A **“graphics engine”** is computer software (though it can also be implemented in hardware) that is able to render given objects to the screen. For video game development one can also make use of **“game engines”**, which do not only render 3D-objects, but can also handle sound, game logic, menus and other things. There are lots of different engines ready to be used, which can greatly improve quality and reduce development time compared to starting a project from scratch.

Also there are tools available which combine a whole game engine with a so-called “authoring tool”. This means that there is not only a raw programming framework for usage in one’s own code, but also a graphical user interface integrated in a software package that can be used to define the environment and place the objects in it. They often also graphically support the user with built-in functions for designing animation, sound and scripting the game logic. It is a whole software package that helps the user with several important tasks.

One of these software tools is called **“Unity”** [Unity]:



Brush Size 25
Opacity 0.5
Target Strength 1
Fog Density 0.013
Base Mult Blur Pixels 4
Blur Over Drive 0
Depth Ambient 2.5

Figure 7: Screenshot from "Unity" (figure from [Unity, “What’s New in Unity 2.5”])

Unity provides its own game engine and allows to utilize it via a graphical user interface. The intuitive tool allows to produce first results very quickly and therefore is very well suited for beginners in the area of interactive 3D-application development. Due to the fact that there is a free version available for download (which can be upgraded to the “Pro” version with additional features for charge), it has recently become very popular among hobbyists and students. But also professional studios increasingly make use of it.

Unity can be used for the development of 3D- and also 2D-applications. In the industry the main purpose it is being used for is the development of video games, though any other kind of application can be developed as well.

Another important advantage of Unity is that it allows true “multiplatform development”. This means that the developer has to implement his application only once, and Unity is able to publish it (at least within the range of its facilities) for different platforms. This feature has

become very important especially due to the increase in mobile phone hardware performance, which allows to play high quality games on the go.
Unity supports the following platforms for publishing of applications:

- PC & Mac
- Internet Browser (needs a plug-in on the client-side)
- iOS (Apple's operating system for mobile devices like the "iPhone")
- Android (Google's operating system for mobile devices)
- Nintendo Wii
- Xbox 360 and Playstation 3

More details on Unity follow later in this master thesis. Unity is the main topic of this thesis, which provides methods and resources to teach the software tool to students.

Another important aspect when it comes to developing applications that are greatly influenced by visuals like 3D-applications is the interdependence between **design and development**.
The designers are responsible for defining the whole concept of the application. For example what's the scenario of the game, how the player can interact with the application or who are the characters he will meet. The design can also include visual concepts for the application, e.g. what it shall look like. Graphical 3D-artists are responsible for realizing the design into 3D-models.
These design tasks are separated from the implementation tasks done by the developers. The developers need the design to exactly know how to do the programming and bring all different resources together into one application.
Of course in small teams it is possible that designers also do development and vice versa. In a single-person team one has to carry out all development and design tasks by oneself.
In a large team with lots of different responsibilities, a process that defines the workflow between designers and developers is absolutely necessary to produce high quality results. Unity provides several functionalities that support this workflow.

1.3 Problems for students who learn to develop interactive 3D-applications

Due to the complexities of 3D-application development, it is not trivial to teach students in this area. Therefore the typical problems that occur while learning and the requirements for teaching need to be analyzed.

This master thesis focuses on university students trying to learn Unity. The students studied for this thesis consist of two different groups:

- Students of **computer science**. This group primarily focuses on applications in the area of video games or other programs that are interactive to a high degree. Often their strengths are development and programming.
- Students of **architecture and building science**. This group is mainly interested in applications that visualize buildings and interior with less user interaction. Frequently their strengths are design and modeling.

Due to the fact that there were not enough students to conduct an empirical study, the chosen method for research was **qualitative studying**. This provided a deep insight into the individual problems of each student.

For this master thesis the students of two university courses were surveyed:

The first course was a **48 hours game development contest**. It was conducted for students of computer science or architecture and was split up into two parts: In the first part the students attended five workshops were they learned the basics of Unity and game development in general. They were introduced into technical topics as well as into issues regarding the development process. These workshops were necessary because the course did not presume any previous knowledge in game development from its participants.

The second part of the course was the contest itself. To provide an appropriate atmosphere for game development, it was conducted at castle “Waldenfels” in Upper Austria [Waldenfels]. The students were divided into groups of 2-3 persons and had to accomplish the task of developing a functional game within 48 hours. The only requirements for the game were that it shall embed the 3D-model of castle Waldenfels into the game (which was provided by the professors) and be innovative in some way. The tight schedule furthermore intensified the experience of working on a project with a close deadline.

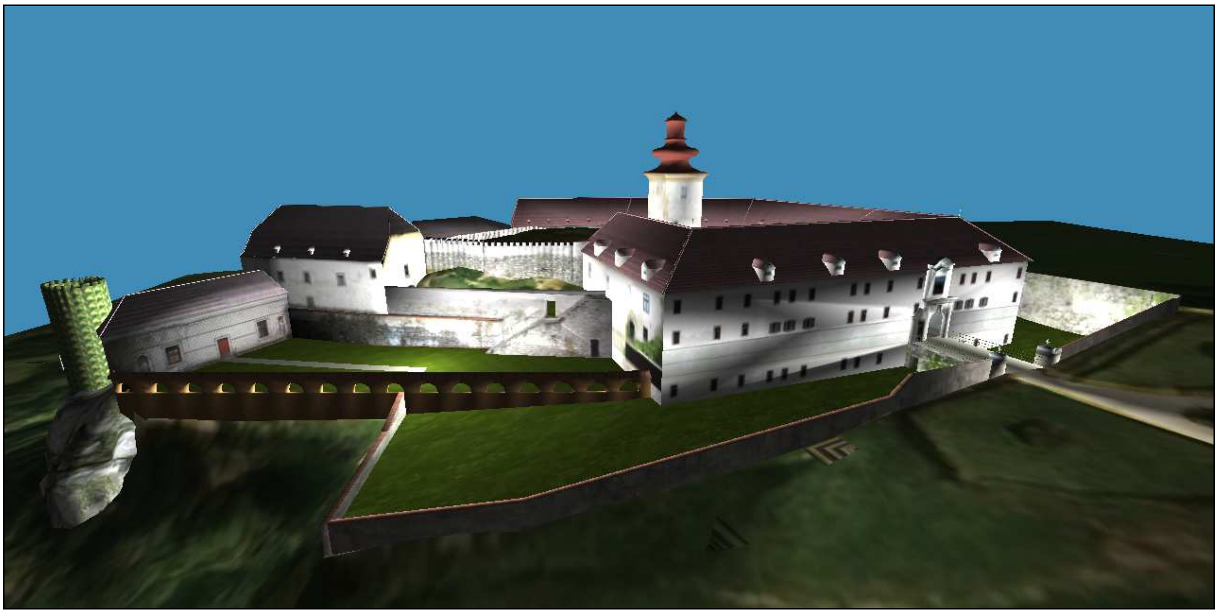


Figure 8: Screenshot of the 3D-model from castle "Waldenfels" that was provided for the 48 hours game development contest

Several measures were taken regarding the course to gain insights for this master thesis:
First and foremost I attended the course by myself and joined a group in developing a game. This way I could not only be very close to the other students of the course, but also get my own insights into the problems beginners have when they start to work on a Unity project on their own. Secondly, I had contact with the students of the other groups and the professors as well during the whole course. Furthermore each student had to fill out a feedback form at the end of the course, in which they explained the problems they had and what could be done any better. This made sure the opinion of each individual student could be obtained.
The end presentations of the different groups were concluded with a review of each student (including the rating of several aspects), which also was an additional source for knowledge. The gained insights follow at the end of this chapter in more detail.

The second course used to gain knowledge for this master thesis was called **“Dynamic Simulation and Visualization”**. This course was conducted primarily for students of architecture and building science. They were split up into small groups of 2-3 persons. Each group had to accomplish one of two tasks: One of those tasks focused on the 3D-modeling of a Balinese village, the other (again) on castle “Waldenfels”. Each group was assigned one special part of these models, so for example one group had to model a Balinese temple, while another one modeled a traditional Balinese house. The idea was that the students can share the models they created with each other, so each group could build an entire village (or the whole castle) using all the parts.

But because the course was about dynamic visualization, it was not enough to just do the modeling. Each group furthermore had to present a concept of an interactive application they had to implement using the 3D-models. These concepts were limited only by their creativity, and included for example obstacle courses or mysterious adventures.

The concepts had to be realized by importing the created 3D-models into Unity, where everything was put together and all application logic had to be defined. Most students had no previous knowledge about Unity, so they were introduced into the software in several tutorials. The professor demonstrated how he accomplished different tasks using Unity, and the students had to reproduce these results on their own computers. They were given the opportunity to ask questions and get help with their problems anytime.

To gain insights for this master thesis, I also attended this course in the role of an assistant and directly supported the students in their work with Unity. This way I could learn about the individual challenges the students had to face and see the most common mistakes they make. Furthermore the way how the professors taught the students was analyzed to gain insights on how to (and how not to) design learning resources for beginners. The direct work with the individual students was a very valuable source of knowledge for this thesis.

The following insights on students' problems and the teaching process were the most important ones that could be gained during the qualitative studies. They were considered when creating the learning resources for beginners with Unity:

- One of the biggest problems during the game contest was **synchronization of work results**. Different persons were working on the same project with the same files. Although the groups arranged work in a way that everyone focused on different parts of the project, it could not be avoided that these parts crossed each other at some point. This is a typical problem: A large project with many different files and lots of dependencies between them has to be manipulated by several persons at the same time. In addition many of the artifacts created during the game development contest were quite large in file size, which further complicated the usage of automatic synchronization tools, because the synchronization process could take very long.

  The method that was finally preferred by the groups was to manually interchange the necessary data by using an USB flash drive. This method of course was very cumbersome and easily caused inconsistent project states. Furthermore it did not support any kind of version management.

- Proper **teamwork and collaboration** in general was a big issue: How shall the project folder be organized? How can work be split up into different parts? How can these parts be merged? What kind of process shall be used during development?

- Another issue that caused many problems during the game contest was the complicated usage of so-called "**prefabs**". These are special artifacts in Unity which can be defined once and then reused at several places. The important part is that when modifying the original source prefab, changes will be applied to all "clones", but this does not happen when just editing a clone. Furthermore there are some steps one has to perform correctly for this to work. It caused much confusion when changes were not applied to any object or sometimes to all objects.

- Though many students in the game contest already were experienced with programming, the **scripting** in Unity works a bit different than other development systems. So it was not only difficult to write a functional script, but also to structure it in a way that allows easy modifications and reuse. Furthermore there are often many possibilities to implement a functionality: When a trigger shall be activated with the collision of two objects, one could either attach a script to object one and call the trigger on the colliding object or vice versa. Though both possibilities may work, it makes sense to be aware of both of them and consistently just use one of them during a project.

  An important concept regarding the functions of scripts in Unity that must be taught to students is the navigation through objects and components in the whole hierarchy. This is necessary to be able to manipulate a different object from within another object, which is often required to accomplish certain tasks.

- Some games developed in the game contest could have been improved a lot if it would have been shown how to accomplish some tasks that are often needed by **using already existing solutions**. One group for example tried to develop a pathfinding system on its own ("pathfinding" is the logic behind objects that directs them from a starting point to an arbitrary destination by automatically avoiding obstacles). The pathfinding system did not only take up large amounts of time, it also did not work well. If the students would have known how to use a free pathfinding system that's available online, they would have saved large amounts of efforts. But the time schedule was too tight to learn how to use the system instantly. Other examples are the usage of an animation library or a framework for explosions.

- In the course "Dynamic Simulation and Visualization" many students had problems with the **importing of their 3D-models** into Unity. This was probably caused by the fact that the students were using several modeling tools in different versions. Here it is probably important to tell students in advance which tools are supported in which versions and can therefore be used.

- Because Unity was all new to the students of the course, they also had problems with the basics: It was difficult for them to **navigate in 3D-space** and position the objects correctly. This hindered their work tremendously.

  This problem also relates to **issues with scaling**: Many students encountered difficulties in defining the right proportions for the dimensions of the different objects and the player. This caused problems like objects that could not be seen because they were too far away or actions that were not triggered because the player-character was too small. It furthermore complicated the testing process because of very large distances that had to be passed and the slow navigation through a gigantic 3D-space. It could be observed that many students had environments that were far too large. It is probably a good idea to start with a small environment on purpose and increase its dimensions step by step when necessary.

- Another big issue for the students was the **process of publishing** the project that was developed to make it playable on other machines. To accomplish this, one has to define exactly what parts of the projects shall be published and the target system's type (like PC, Mac, Android,…).

  The second step also caused some problems: When Unity generated all necessary artifacts in the publishing process, it is important to migrate all parts of the project that are needed to the other machine. It often happened that students migrated the executable *.exe file, but not the artifacts it depends on. The result was that the project could not be played on the other machine.

- Many students had difficulties in applying one of the most fundamental functions for an interactive 3D-application: The **controls for the user** (in Unity the related component is called “character controller”). In many student projects, the character played by the user fell through the ground, passed through walls or other objects he should not walk through, or simply moved either too slow or too fast.

  Though the standard character controller in Unity is able to walk and look in all directions, and even jump to a defined height, many students could have needed a controller that is able to fly through the scene, to be able to easily view the environment from all angles. Such a character controller was implemented for this master thesis.

- Regarding the lessons on **script programming**: It was observed that it was very important for beginners to be introduced into scripting only with very short examples of code. These examples shall only include the concepts that shall be taught at the very moment. When writing a piece of code to accomplish a certain task it is often the case that other functions that are necessary indirectly have to be included as well. These “side-functions” shall be reduced as far as possible in examples for beginners.

  Another important aspect noted was the necessity of presenting the code examples for a long period of time to the students.

  Many of them follow this pattern: At first, they try to analyze the code when it is shown on the screen and listen to the professor’s explanations. Afterwards they try to transcript the code onto their own machine (or onto paper). Doing the transcription while listening hinders the understanding of the explanations. So it is probably better to do any transcription afterwards, but teachers should be aware of that and show the scripts for a long period of time.

  Another option is to prepare the scripts ready for usage and deliver them to the students directly. But this method has a significant drawback: There is a learning effect in the process of writing the scripts by oneself, even if it is just a direct transcription. In addition, because in programming one has to define every single symbol correctly (which is a new experience for some beginners), many students make mistakes in this process and so get the chance to learn from them.

  Also it is not only important to show them examples they can adopt, but show the students how to help themselves: So the professor very often used the “Scripting

Reference", an online documentation that describes all programming functions. This way the students were demonstrated directly how to find definitions they don't know instantly.

- A challenge all beginners with Unity have to face is the difference between **"collisions" and "triggers"**. Both can cause certain actions when two objects clash with each other, but while collisions initiate physical interactions, triggers shall only activate developer-defined events. Also there are some subtle differences in the configuration of both of them. So it is often the case that beginners define a collision but expect a trigger to happen and vice versa.

- It was also noted that the professors often used the concept of animations to demonstrate example actions that could happen. This has the advantage of better visibility than only printing a "Hello World!" message to the output console. Furthermore the students are better introduced into the definition of animations. But the drawback is that this method makes demonstrative examples more complicated.

- Furthermore one issue was observed that probably could be improved when teaching Unity: Occasionally when the professor created examples to demonstrate certain functions of Unity, these examples seemed a bit "random". For instance instead of designing an object that resembles a door he made a cube and told the students this cube should represent a "door" for now.

  Of course it makes no sense to model every detail of the whole environment correctly if one just wants to demonstrate how to open a door. But it makes sense to model it to an extent that makes it easier to relate it to the finished concepts. This way students won't get confused through an additional abstraction layer and can focus on the ideas that shall be taught at the moment. Although this might not always be possible when teaching several independent units that deal with completely different topics.

2 Approach to facilitate the learning process

This chapter shall give an overview of the approach that was chosen to support students in learning Unity. This approach is based on three pillars:

- Help regarding the technical aspects of using Unity.
- Help related to the collaborative aspects of working with Unity in a team.
- A location that centrally provides all help resources.

The approach of how to realize these three aspects is now explained in more detail. The actual outcomes can be found in successive chapters.

2.1 Unity introduction tutorials

One of the most important things for beginners of Unity or any other new technology are resources they can work through to gain the necessary knowledge. These resources are called "tutorials".

In the majority of cases they consist of text combined with images, but also tutorials that can be watched as a video in motion are common in the internet era. Generally they can address beginners as well as intermediate and advanced students of the topic they deal with. Tutorials for example can contain theoretical information, step-by-step instructions to reproduce or exercises the student must solve.

It is important to know what can and what cannot be achieved with tutorials: The usage of tutorials is a very-well suited way to gain the basic knowledge on a topic that is new to the student. This basic knowledge is essential to be able to start working with a tool. Without this, work can only be done following the principle of trial and error, which is incredibly ineffective.

But the constraints of what can be learned through tutorials shall also be mentioned here: A solid knowledge that can be used to solve complex problems in real-life project settings can only be achieved through exercise and experience with such problems. Tutorials can be the starting point for further learning processes, but they will not provide a high degree of deep understanding.

In the context of this master thesis, several tutorials for Unity have been created for the target audience described in chapter 1.3. The tutorials were created with consideration of certain aspects:

- Due to the audience consisting of students with different levels of knowledge, it was not possible to assume any previous knowledge. For example it cannot be expected that all students already possessed basic programming skills.
  Therefore the whole body of knowledge that shall be taught to the students was split up into several "modules". Each module shall contain a unit of knowledge that can be worked through. Furthermore the modules are labeled with some kind of metadata, like "goals", "prerequisites" and "difficulty". This way students can easily identify if a module is appropriate for their current level.
  It is important that **each module can be used as a resource on its own** (at least when the reader meets the defined prerequisites), so students don't have to work through other resources when they just need information about one certain topic. This intended independence between different modules has a consequence: It does not allow the usage of incremental examples. That means that there must not be a module based on resources created in another module, because a student who just wants to take the latter one would be missing something.
  Despite this independence of different modules it should also be possible for students to work through them in a linear fashion. Fanny Klett states that "*The flexible access to hypermedia information in various depth always holds a disorientation problem.*" [Klett, 2001]. Hence she emphasizes the importance of structuring the existing information. Therefore the modules created for this master thesis are provided with a recommended structure of how to work them through.

- The tutorials include a **high degree of practice**. Best results can be achieved if students who work through the modules reproduce the instructions on their own. The tutorials are designed in a way that this can be done in small incremental steps, so students do not lose the thread. The examples themselves are tangible and focused on the current topic, without too much side information. The practical examples are always provided with the necessary theoretical background.

- The tutorials mainly consist of text (including source code listings), but are intertwined with **demonstrative images** related to the current situation. Videos could not be produced due to the necessary efforts, though they could provide a valuable resource and may be added in future work.

2.2 Guidelines for team collaboration in Unity projects

Due to the complexities and efforts necessary to develop interactive 3D-applications, many projects are performed within a team rather than by single persons. Of course this also adds new challenges that need to be handled: In a team it is not only important that every member has the technical skills needed in the development process. It is also necessary that the collaboration itself is organized in an effective way. Work has to be clearly divided among the team members, development goals must be scheduled, work results have to be handed over to other persons depending on them in time, the whole project must be synchronized and of course there have to be ways of providing a steady communication flow between the different members of the team.

To ease this collaboration process for beginners learning Unity, this master thesis contains guidelines that provide possible solutions of how to deal with certain aspects of team work in chapter 4. These guidelines contain:

- Approaches for structuring a Unity project
- Approaches for synchronizing work results with other team members
- A recommended development process for a small team of beginners
- Considerations regarding quality assurance

2.3 Repository for resources

Another goal of this master thesis was to set up a centrally provided repository that can contain all resources produced in this context and also is a place that can be used to save resources created in future work. It shall ease the access for students to the tutorials, guidelines and other assets that were created to help them.

The acceptance among students to make use of the created resources greatly depended on the usability of the technology utilized for the repository. Hence an evaluation was performed in which different tools and technologies were compared regarding their applicability for the given purpose. Of course the chosen set could not be exhaustive, but it contained the most important technologies with significant representatives.

The tools were analyzed regarding the following aspects:

- **Possibilities for structuring and finding resources**

  A repository that probably will contain a large number of resources must allow to structure them in a meaningful way that eases their maintenance and also the search for desired resources. Possible ways of structuring content are: Hierarchical structures, marking content with tags or keywords, categorization of content or chronological structures. Another important option for finding resources is the possibility for doing a full-text search.

- **Visual design and usability**

  Because the repository shall be accessible to a large group of people, it must be easy to use without having to spend much time to get familiar with the system itself. This would dramatically reduce the acceptance of the repository among students.

  The best interface for this purpose uses elements that are well-established in the web-context and so can be used without lots of instructions. Furthermore the technology that shall be used must be modifiable in its visual design, so the interface can be related to the university context.

- **Efforts for maintenance, installation and administration**

  Not only the usability of the technology for the frontend-user is an important aspect, but also for the backend-users who need to install the tools correctly, insert new content and perform administrative tasks. Because in the university context probably no one can concentrate only on these activities, it is important that they can be performed easily with low efforts.

- **Possibilities for (pre-)viewing content**

  When searching through the repository it is important for the user to preview the contents in the repository before obtaining them. It is not feasible for the students to download each 3D-model only to be able to see it. There must be some kind of preview possible that allows to get a first glimpse of the resources before deciding what to download.

  For other resources like tutorials there must be a variety of options to write text in different formats and combine it with other media like images or weblinks.

- **User management and permission system**

  Because possibly a larger group of people – students in courses, their colleagues, anonymous access over the web – will have access to the repository, it is important to be able to precisely define the permissions for different groups. In general it can be said that the contents shall be readable for many users, but modifiable only for a few users. Administrative tasks of course have to be performed by a separate group.

- **Possibilities for collaboration**

  Though it shall not be the primary purpose for the repository, it might be useful to provide some means for collaboration among students. Examples for that could be issue tracking for open tasks, scheduling of goals to achieve or tools for communication like messaging or forums. These tools could help students in organizing their projects for Unity, but they are not as important as other aspects when evaluating technologies for usage as repository.

- **Version history of content**

  A feature that is not absolutely necessary but useful in certain situations is the ability of automatically saving the contents in each new version. This provides the option to be able to undo changes that have been done by accident, or compare different versions of the same resource to be able to analyze the differences.

Now follows a list of tools and technologies with the evaluation results regarding the different aspects mentioned.

2.3.1 Trac

Trac [Trac] is an open source configuration and project management system. It is web-based with the main features of a wiki (a software system that allows the collaborative editing of document pages), issue tracking and tight integration with the version control system Subversion. Trac is mainly used for software development projects, but due to the wiki and the version control system it could be used for other purposes as well. It was chosen as representative for tools focusing on collaboration aspects.

Possibilities for structuring and finding resources:

Trac's wiki allows hierarchical structures, a plug-in makes it also possible to mark resources with tags. A browser allows to easily search the file-repository. A "timeline" shows recent changes in a chronological view. Full-text search is another option for finding resources.

Visual design and usability:

The graphical interface is unspectacular. It is a standard design adequate for project management, but not very suitable for game development. However the interface can be modified through the usage of themes.

The usage of Trac for the end-user should not cause any bigger problems.

Efforts for maintenance, installation and administration:

The usage of the wiki, which is a very stiff structure, could cause high efforts for the authors. It is not easy to construct a useful hierarchy in the wiki, and it cannot be changed in a facile way afterwards. This probably hinders the creation of content a lot.

Installation is also quite complex due to the need to install several necessary software packages. The webserver needs to be configured correctly with Subversion support and the software needs a database connection. For general administration background knowledge is absolutely required to solve certain tasks.

Possibilities for (pre-)viewing content:

Due to the fully-functional wiki that is provided with Trac, there are lots of options of how to create content. It can embed text in many formats combined with images and other resources. However any preview for content must be added manually.

User management and permission system:

Due to its usage in the context of project management, Trac has a very flexible and elaborate system for permissions and user management.

Possibilities for collaboration:

Collaboration is the main case Trac is used for, so it clearly is its greatest strength. Trac has a sophisticated system for issue tracking, the wiki can be edited in a collaborative way and Subversion allows professional distributed work on files.

Version history of content:

Trac automatically saves each version of wiki pages that have been edited, and Subversion allows professional version control for file hierarchies.

Miscellaneous

One of Trac’s main features is a very tight cross-linking of all resources: For example wiki pages can be related to issues or to files in Subversion. Furthermore Trac can be extended with plug-ins.

2.3.2 MediaWiki

MediaWiki [MediaWiki] is one of the most popular tools that can be used as a wiki-system. The popular online encyclopedia “Wikipedia” uses MediaWiki as underlying engine.

The focus of a wiki-system lies in textual descriptions, but other media like images, videos or sounds can be integrated as well. Also all kinds of files can be attached to pages. This flexibility was the reason why also a pure wiki system was chosen for evaluation.

Possibilities for structuring and finding resources:

Hierarchical structuring of content is possible through the usage of categories. However the main method of finding resources in a wiki is probably through full-text search. A chronological view shows the recent changes.

Visual design and usability:

An advantage of MediaWiki is the look and feel that is familiar to most people because of Wikipedia’s popularity. It is a functional interface that can be customized to individual needs.

Efforts for maintenance, installation and administration:

MediaWiki has similar disadvantages as Trac when it comes to creation of content: A wiki system causes high efforts because everything has to be stored within pages. Therefore changes in the structure also cause lots of changes in the documents. Modifications that have to be applied can become very tedious.

Possibilities for (pre-)viewing content:

Due to the flexible creation of pages content can be designed in all combinations of text and other media. However it has to be done manually.

User management and permission system:

Can be configured through the usage of predefined user groups.

Possibilities for collaboration:

Collaboration is limited to the collaborative editing of wiki-pages. This is useful for the distributed work on documentation and other textual resources, but not for other purposes.

Version history of content:

Wiki-pages and uploaded files are being versioned.

2.3.3 WordPress

WordPress [WordPress] is the most popular software that is being used for so-called “blogs”. A blog is a website that focuses on chronological publication of content. So-called “bloggers” maintain such a blog and periodically fill it with new stories, pictures, videos or other media. Most blogs are supervised only by a few or even one person. Examples for bloggers are persons writing about traveling experiences, reviewing new technologies or publishing their progress on a project.

WordPress was chosen as a candidate for the asset repository because it can be used very flexibly for all kinds of purposes.

Possibilities for structuring and finding resources:

WordPress offers the whole range of possibilities for structuring resources: Besides the chronological structure of content which is essential in every blog and can show the most recent entries as well as the entries for a specific month, it also provides support for categories. It furthermore differentiates categories, which can be structured hierarchically, from tags. So every resource can be assigned to one or more categories and be marked with several tags as well. The existing tags can be viewed in a “tag-cloud”, that visually shows the most popular tags. Full-text search is another option.

Visual design and usability:

WordPress offers a standard design which can be modified completely through the usage of templates. The community also offers lots of predefined layouts. Due to the popularity of blogging the general interface of the system should be familiar to most web-savvy persons. In general it is very easy to handle.

Efforts for maintenance, installation and administration:

Due to the fact that WordPress is quite an advanced technology, the system can also be used by authors very well. WordPress is intended to be utilized by everyone and not only by experts. Thanks to this fact the whole administration and creation of content can be done very conveniently. Installation also is a very straightforward process.

Changes to the content or its structure can be performed without problems, even afterwards. This is a consequence of the fact that each unit of content (each “post”) has its own identifier. When referencing this identifier it does not matter in which category the post is located. So hyperlinks to this identifier remain stable after changes.

Possibilities for (pre-)viewing content:

WordPress offers all possibilities for creation of content: Text, images and other media in interleaved combinations. Posts can be previewed by the end-users, but this preview only shows the first lines of text. Other forms of preview would have to be done manually for every post, an automated preview is not possible in standard WordPress for files like 3D-models. Though this maybe could be added with a plug-in.

User management and permission system:

WordPress offers an easy to use permission system, where predefined roles like “administrator”, “author” or “subscriber” can be assigned to certain users. In general it is possible to define the permissions for certain posts or even the whole blog. It is not a highly sophisticated permission system, but provides the most important functionalities.

Possibilities for collaboration:

Collaboration probably is a weak point in WordPress. It is limited to users who are registered and therefore can modify the content in a distributed way. However a wiki offers more options for the collaborative modification of document pages.

WordPress in the standard package does not provide any features like issue tracking or other support for project management.

Version history of content:

WordPress is able to save each version of a post separately. It also offers a way to compare the different revisions of a content page.

A notable detail: WordPress regularly performs an autosave while content is being modified to provide a backup when problems occur.

Miscellaneous

The system can be extended through the use of plug-ins. Due to the popularity of WordPress there is a large community which already provides thousands of plug-ins for countless use-cases that are not covered in the standard package.

2.3.4 Kohive

Kohive [Kohive] can be called a "shared-desktop" platform. The cloud-based service can be accessed with a web-browser after free registration. It provides a virtual workspace resembling the desktop of an operating system, called a "hive". Every user can create multiple hives and also invite other users into their own hives. This allows to work on resources in a collaborative way.

Though it might seem exotic to use Kohive for an asset repository, it was decided to investigate its suitability in more detail. At the evaluation time the system was in "beta" stage.

Possibilities for structuring and finding resources:

Even though Kohive allows to upload any kind of file to a hive, it revealed the great drawback of not being able to hierarchically structure them.

Files can be marked with tags and provided with descriptions. A certain area shows the most recent changes in the hive, but without any options for filtering. Full-text search can also be applied.

Visual design and usability:

Because of the well-known desktop metaphor usage works quite fine. However possible adaptations to the design are limited to changing the background of the desktop.

Efforts for maintenance, installation and administration:

Files can be added quite easily, but this is a consequence of the fact that there are almost no options for structuring them. The system shows all files in a simple list.

Installation is not necessary, because the system resides on an external server. Administration is easy but only offers very few options.

Possibilities for (pre-)viewing content:

There are no possibilities for previewing files, which is a major drawback. Also Kohive does not allow to combine images with text and other media in arbitrary combinations. This is only possible indirectly through the upload of files that contain the intended content.

User management and permission system:

Apparently Kohive does not allow the configuration of permissions for other users. Therefore the system is not appropriate for giving access to a large number of people.

Possibilities for collaboration:

Kohive offers some features for collaboration: It provides a task management feature, possibilities for chatting and sharing content. However these features are basic and not highly sophisticated.

Version history of content:

No possibilities for versioning of content could be found.

Miscellaneous

Due to the beta stage the system does not work completely stable, several errors were encountered. Furthermore the fact that Kohive is a service offered by another company bears some consequences: Though it eases installation and administration, it creates **dependency**. If the company cannot maintain reliability, the whole service might be offline. Or even worse: The company stops business and does not offer the service anymore at all.

Another consequence is that all data resides on an external server. This certainly can be a breeding ground for legal issues and loss of data.

2.3.5 ResourceSpace

ResourceSpace [ResourceSpace] is open source software and was chosen as representative among the group of "digital asset management" systems. This kind of software has the main purpose of maintaining a large number of digital resources in a structured form and providing them to end-users. For example these systems are perfectly suitable for implementing a large database of photographs for a company in the context of journalism. So it seems that digital asset management is an important option to investigate for this evaluation.

Possibilities for structuring and finding resources:

ResourceSpace is quite complex when it comes to structuring digital resources. Even though the system offers the possibility to assign assets into a hierarchical tree of categories, this option is quite cumbersome and not very handy. The guide says that the developers wanted to avoid a search through a large tree, so this has been done on purpose.

Instead, ResourceSpace offers a pretty comprehensive system for assigning metadata to assets. The allowed and obligatory metadata can be configured very exactly for each kind of resource. This system's possibilities go far beyond simple tagging.

A full-text search is able to search through all metadata. ResourceSpace is even able to automatically show related assets to certain resources. A "recent" area also displays the newest assets.

ResourceSpace also offers the possibility to group certain assets to so-called "collections". They subsume a defined set of assets under a specific name. These collections can be predefined and made publicly available to all users, or made private and only be available to the user who defined the collection.

Visual design and usability:

ResourceSpace has a very modern look and feel that appears very professional. The system generally is usable very well for the end-user, though it requires some work to get familiar with the many functions it offers.

Efforts for maintenance, installation and administration:

Installation requires some tricky adjustments in the server configuration, but it is feasible. In the beginning there are lots of efforts necessary to set up the complex metadata system, which should be done very carefully because it affects the overall value of the further work with the system.

ResourceSpace offers lots of support functions to ease the maintenance and the import of new content. However the extensive metadata system could probably cause high efforts when it needs to be changed after some time.

Possibilities for (pre-)viewing content:

One of the most useful functions ResourceSpace offers is the ability to automatically create preview thumbnails for lots of supported file formats. On the other hand, the system is not that flexible when it comes to freely combine text with other media. This could cause problems when trying to add tutorials or other text-based resources.

User management and permission system:

The definition of permissions works a bit cumbersome, but generally it offers a large number of options for configuration.

Possibilities for collaboration:

Collaboration is limited to the distributed import and editing of assets. Thanks to the metadata system, the results of this process can be controlled very well. But in general there are no additional features for project management.

Version history of content:

No functionalities regarding version control could be found.

Miscellaneous

ResourceSpace also offers lots of features concerning the generation of statistics and creation of reports.

2.3.6 Razuna

Razuna [Razuna] is another open source representative in the group of digital asset management software. Due to ResourceSpace's unusual system of structuring resources, it was decided to also add a more traditional candidate for evaluation.

Possibilities for structuring and finding resources:

Resources can be structured hierarchically very easy by using folders. Keywords can be added to every asset. There is no "news" section, but the assets can be sorted by date.

A full-text search that can be parameterized comprehensively is also available. However during tests the search seemed to "miss" some of the keywords defined in the assets and did not show them in the results.

Visual design and usability:

The interface is unspectacular, but generally fine to use. However many bugs in the GUI were encountered that complicated the usage of the system.

Efforts for maintenance, installation and administration:

Installation can be performed very easily when using a ready-to-use package provided from the website. If this package cannot be used, setting up the system becomes quite more difficult and requires a Tomcat server with several additional packages to be installed.

In general adding content and maintaining it should be working quite fine with Razuna. But during local tests many errors and unexpected behaviors of the system were encountered. It is very probable that this causes high efforts for administration.

Possibilities for (pre-)viewing content:

Just like other digital asset management systems, Razuna is able to automatically create preview thumbnails for uploaded assets. All assets can be described with text or other definitions. However it is not possible to flexibly intertwine text with images or other media, which would be necessary for tutorials.

User management and permission system:

Razuna does not offer a sophisticated permission system. Users can be added to defined groups, and these groups can be granted read or read / write access to specific asset folders. This system should solve the most common use-cases, but probably is insufficient in more complex situations.

Possibilities for collaboration:

Collaboration is limited to distributed work on assets. There are no other features like issue tracking or web-based communication.

Version history of content:

Assets are being versioned, so older versions still can be accessed when uploading a new version.

Miscellaneous

After testing, the impression was that Razuna is not a technically mature product yet. An extraordinary number of errors and unexpected system behaviors were encountered. Furthermore the wiki that should be the documentation for Razuna is very incomplete, which additionally complicates usage.

2.3.7 Result of evaluation

Several discussions based on the evaluation insights were held to choose the appropriate tool for an asset repository. In the end it was decided to use **WordPress**. The other candidates were discarded due to the following reasons:

- The evaluation showed that a wiki would probably cause high efforts for maintaining the contents, especially when changes need to be made. This dropped out Trac and MediaWiki.
- Kohive is not able to provide contents in an appropriate form. Its permission system makes it inadequate to be used in a broad university context. Furthermore due to the “beta” stadium and the external service it could be very unreliable in the future.
- Razuna was a promising candidate, but could not live up to high quality standards. ResourceSpace was the second best system after WordPress. Finally it was not chosen because of its lacking capabilities to intertwine text with other media and the high efforts to set it up correctly.

The following arguments support the use of WordPress:

- It is easy to install, administer and to be utilized by the end-user.
- It offers all important possibilities for structuring and finding resources.
- Changes in the structure do not cause high efforts.
- Content can be defined very flexible which is important when mixing textual and other media-based resources.
- Most other features that are important for the purpose like a permission system, version control and a customizable interface are also provided. There is a high chance that features that are not provided within the standard package can be implemented with plug-ins.
- A large community is behind the system which makes it a very reliable tool for the future and ensures documentation and support.

3 Tutorials for students learning Unity

This section of the master thesis contains the resources that were created to teach Unity to beginners. The knowledge has been split up into separate modules. Each module contains information to a certain topic. This allows to work through a specific unit without the need to read the preceding modules (except it is necessary due to missing knowledge).

The modules have been provided with descriptive data like goal, prerequisites and difficulty. Furthermore they have been arranged into an order that resembles the recommended order of working through the modules. The section starts with an overview of the tutorials and their relationships.

3.1 Overview

Though each module can be used by itself, altogether they form a course that can be used to get the basic knowledge in how to work with Unity. They have been structured like illustrated in the following diagram:

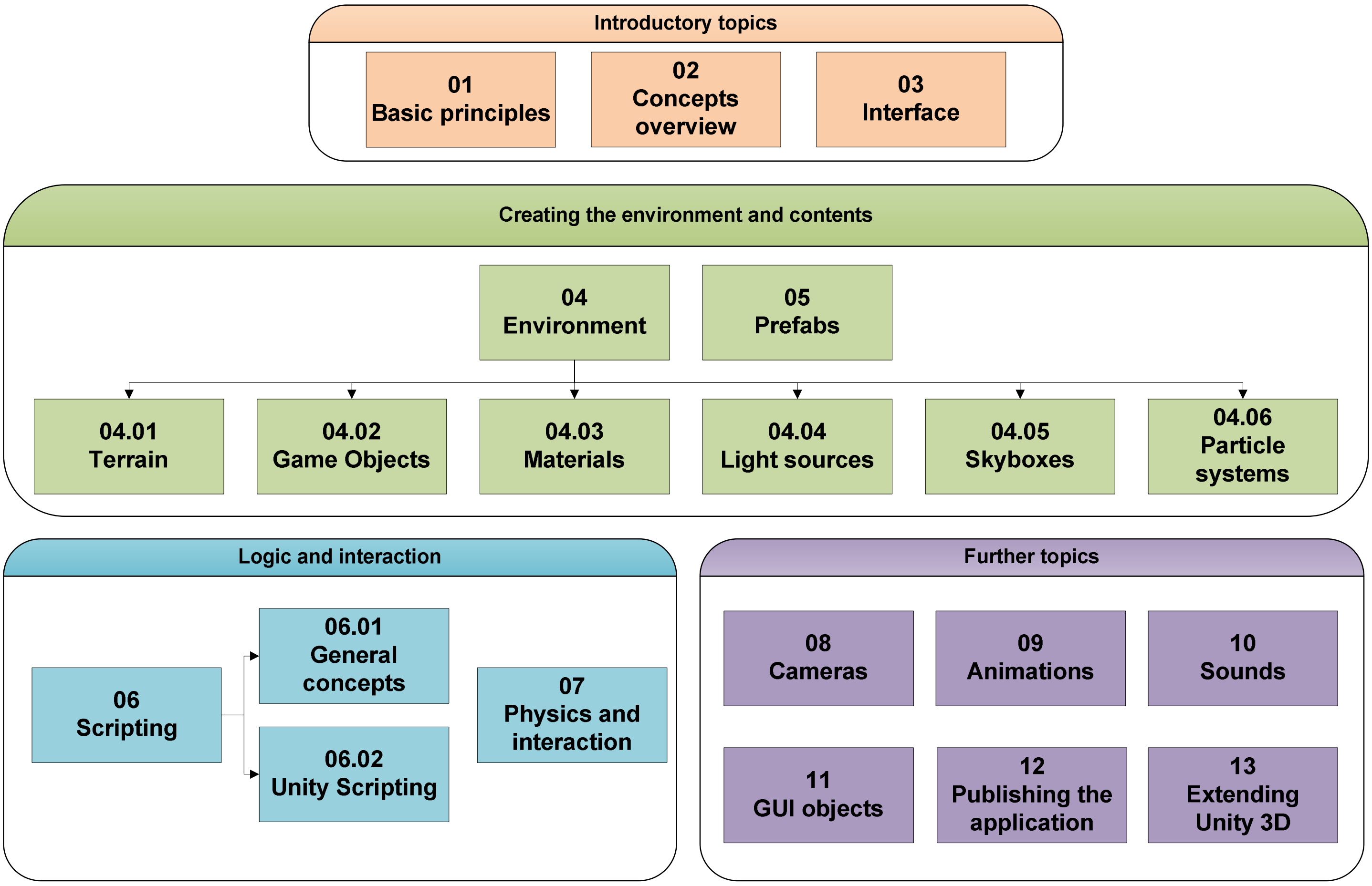


Figure 9: Overview of the different modules about Unity

3.2 MODULE 01: Basic principles of 3D-applications

Goal: Understanding the most fundamental concepts regarding 3D-applications in general

Difficulty: Beginner

Prerequisites: None

3D-applications are able to create the illusion of three dimensions on a computer. But how is this possible with a screen that is flat and does not provide any depths?

The secret lies in the concept of **projection**. In general a projection is the process of mapping something onto another surface. In a cinema the information of the movie that shall be shown is projected onto the big screen. Similarly the information the computer calculates for the current scene of the 3D-application is projected onto the flat computer screen.

To perform this projection, the computer needs to convert all information it stores in three dimensions into a two-dimensional system. The following figure gives an idea about the concept of projection:

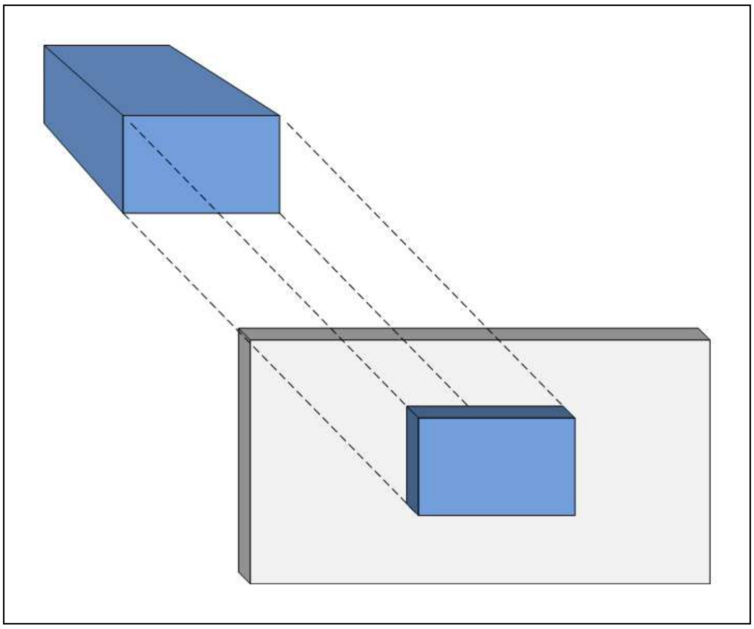


Figure 10: Projecting a three-dimensional object onto a two-dimensional screen

There are different ways of how to perform the projection, some are more complicated (and therefore often have a more realistic result) than others. But luckily, when working with Unity you do not have to care about projection, this is already handled by the Unity game engine. However it is useful knowledge that helps in understanding other concepts.

Another topic that is fundamental when working with 3D-applications is the **coordinate system**. Coordinates are being used to define the positions of objects or other points of interest. While in a standard two-dimensional coordinate system there is an x-axis to describe the horizontal position and an y-axis for the vertical location, in a 3D-application there also is a z-axis that adds depths. Coordinates are defined in numeric value pairs (for two dimensions) or triples (for three dimensions). The following diagram shows two objects in two different coordinate planes:

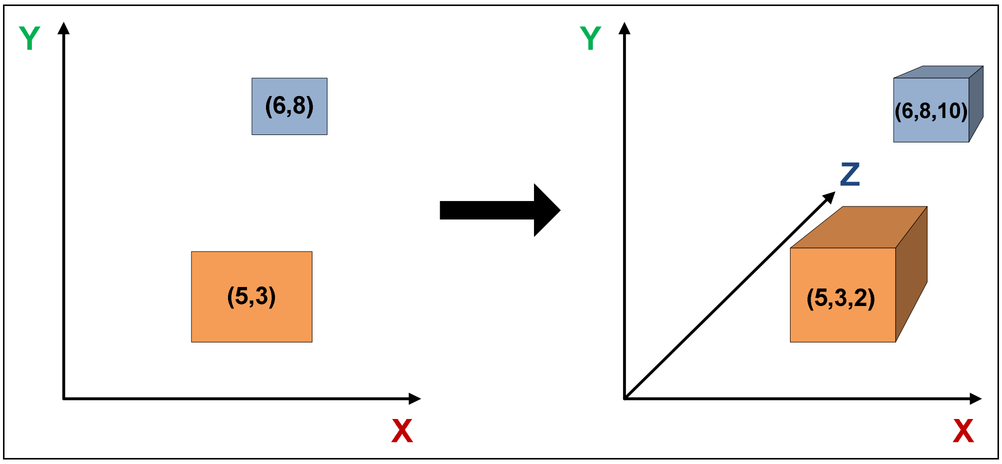


Figure 11: Coordinate systems with different dimensions

The first coordinate system only knows two dimensions. Coordinates are stated as "(x,y)" pairs, so the first object has the x-value of 5 and the y-value of 3. The blue object has the coordinates (6,8).

The second coordinate system adds depths by using the z-axis. The object positions now are defined with triples: Values for x and y remain the same, but now the orange object also has a z-value of 2 and the blue object of 10. This means, that the blue object is not only positioned above the other, but also behind it.

It is a common practice to state positions as "(x,y,z)" coordinates (in this order!). Furthermore many 3D-applications add colors to mark the different axes: **Red** for x, **green** for y and **blue** for z.

The bottom left corner, the point where the coordinate system "starts" is called the origin. It always has the coordinates (0,0) or (0,0,0). It's also possible to locate objects to the left or below the origin, which causes the coordinate values to become negative.

Another important concept are **vectors**: While a point simply is the representation of a specific position in the coordinate plane, a vector contains information about length and direction. Confusing might be the fact that vectors are also defined by using numeric values for x, y and z, so they look quite similar to positions. But they are used in different contexts: For example you have to use vectors when calculating the distance between two objects, or to define the direction the player moves to. In Unity there are lots of operations that need a vector as input, so it is important in many circumstances. Fortunately Unity provides lot of supporting functions that help us in dealing with vectors.

When working with coordinate systems it is necessary to mention that 3D-applications use a specific technique to simplify some issues: The conversion of **world space into local space**. World space (or also called “global space”) just means the same like mentioned before: There is one coordinate plane that is the reference for the definition of the coordinates of all other objects.

When using local space, the objects can be equipped with their “own” coordinate systems, which usually originate in the center of these objects. This way it is much easier to state relative positions like “5 units below this object”, which would be a simple statement like “(0, -5, 0)”. In world space one would have to calculate the position with absolute values referring the origin. Unity provides functions to automatically convert between local and global space. The following diagram illustrates the concept. It shows a cube first in global coordinates, and then within its own local space (in which the object itself of course has the position “(0,0,0)”):

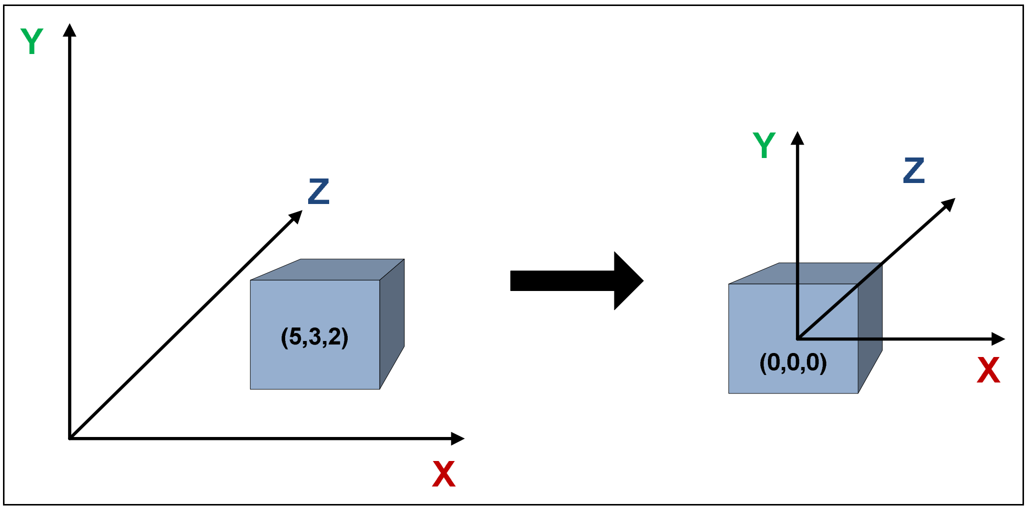


Figure 12: Global and local space

In 3D-applications all three-dimensional objects are made up using two-dimensional **polygons**. A polygon is a defined geometric plane, like for example a triangle, rectangle or arbitrary other structures. Though this might seem simple, it is possible to create highly detailed objects by combining lots of polygons. But the more polygons an object consists of, the more work has to be done by the machine to render it correctly. So the number of polygons is an important aspect one should keep in mind when developing applications with Unity.

However polygons all by themselves could not make up a detailed object, they furthermore need to be provided with realistic surfaces. A car that just has one color cannot resemble a real car, it needs surfaces in different color tones, reflections and transparent parts. To achieve this, object surfaces can be equipped with 2D-images, so-called **“textures”**.

These textures can make an object look quite realistic, but do not provide any effects. For example the reflection from a surface changes with the point of view, which cannot be achieved with a graphic that is static. Therefore exist so-called **“shaders”**.

Shaders are written in a special language which makes them look quite similar to programming code. Using shaders a developer can create various effects like reflections, transparency or color effects. Unity provides a lot of built-in shaders that can be used for the most common use-cases. Shaders and textures can be combined and parameterized in so-called **“materials”**.

The final topic that shall be mentioned here regarding the basics of 3D-applications is the **manipulation of objects**. In most 3D-applications there are three fundamental attributes that are defined for every object: Position, rotation and scale.

- The position of an object is its location in the environment. By changing the position an object can be moved forward, downward or in other directions (thus around all axes). The operation of manipulating the position of an object is also called “translation”.
- The rotation of an object is its angle relative to the environment. Changing the rotation means to turn it around the x, y or z-axis. This way the direction a character is looking can be changed, or an item turned upside down. The operation is simply called “rotation”. Values are given mostly in degrees.
- The scale of an object are its proportions in size relative to its original size. A scale factor of 1 means no change in dimensions, a scale factor of 2 that the object’s size has doubled. The scale also can be changed around all axes separately. By not changing the scale proportional around all axes equally an object will be stretched. The operation is called “scaling”.

The following figure illustrates the different operations:

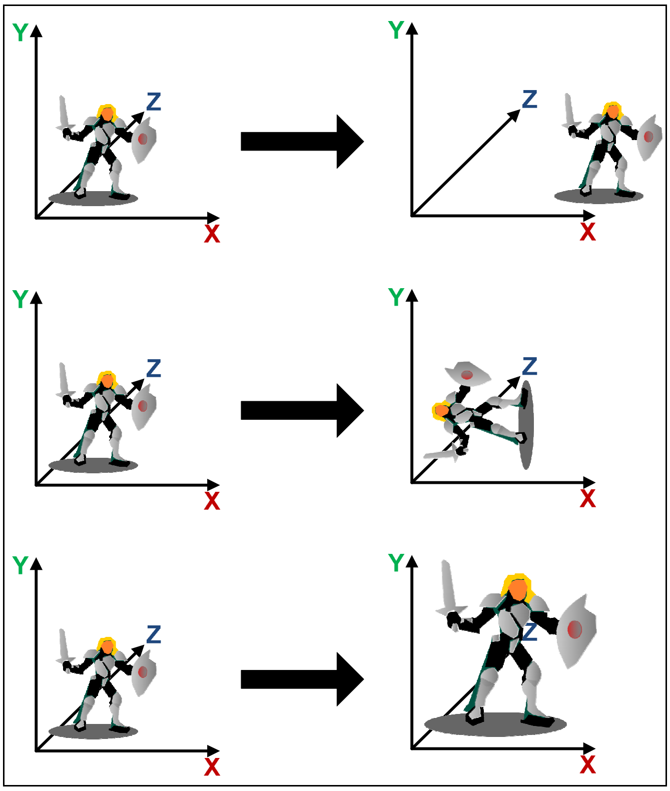


Figure 13: The basic operations translation, rotation and scaling

3.3 MODULE 02: Overview of the fundamental concepts in Unity

Goal: Get an insight into the basic concepts of Unity and what they can be used for, without any detailed knowledge yet

Difficulty: Beginner

Prerequisites: None

While there are certain concepts that are similar in all 3D-applications, some are special when working with Unity. This module shall give an overview of those concepts to catch a first glimpse without getting into any details. They are handled in all its particulars in later modules.

- **"Asset"** is a term that is used very broadly in Unity. An asset can be anything that is being used for a Unity project, for example textures, 3D-models, programming scripts or background images. In general all files that are located within the current project folder and can be used in the application are called assets.
- **"GameObjects"** are objects that are actually used in the current environment. They can be positioned, rotated, scaled and manipulated in many other ways. The difference to an asset is as follows: An asset could be a 3D-model file of a monster that resides

within the folder of the project on the computer's file system. When using this 3D-model in the dungeon that is currently designed, the asset becomes a game object. When manipulating this game object, the original file is not changed. Furthermore the one single file could be used to create a whole horde of monsters.

There are some predefined game objects in Unity that can be used right from the start without any asset, for example geometric bodies like cubes, spheres or cylinders. Also there are special objects that provide extra functionalities, like for example cameras, light sources or menu objects.

- Any game object can consist of one or more **"components"**. A component is part of a game object and describes either attributes or provides functionalities for this object. For example every game object has the component "transform", which can be used to manipulate the position, rotation and scale of an object. Another example is the "RigidBody" component that adds realistic physical behavior to an object.
- A **"scene"** in Unity subsumes the current environment with all contents in it that are being designed for an application. This includes all game objects with their current positions and attributes. An application created with Unity can consist of one or more scenes. The developer can exactly define in which situation the player will jump to another scene. So a scene suits very well to be used as a "level" in a game.
- **"Scripts"** are files that contain programming code. They need to be used to define the whole logic of the application. This means to handle issues like for example: What happens when the player character touches the enemy? How can the character be controlled by the user? Where are the highscores being saved?

  Scripts need to be written in a programming language and must strictly meet certain formal constraints. Unity supports usage of three different languages: "JavaScript", "C#" ("C-Sharp") or "Boo" (a variant of the "Python" programming language). The different languages cannot be mixed in a single file, but different files can use different languages in the same project.
- So-called **"Prefabs"** are one of the most important concepts in Unity at all. You have already seen the difference between assets and game objects. An asset is a file that can be used many times in different situations of the project. Utilization in a scene turns them into a game object.

  But what if modifications are made to the game object (not to the asset-file!), which could make sense to be reused later?

For example: The 3D-model file of a character is imported into the current scene. There it becomes a game object, and the developer equips the character with a weapon and applies different textures to him to make him fit into the current level. If the developer now wants to reuse this whole character with weapon and textures in a different scene, he would have to redo all the steps mentioned before. But by using prefabs, he can save the textured character with his weapon in its current state. The developer can save the whole package as a prefab (with a significant name like “armed hero”), and use this prefab in multiple locations (by creating so-called “instances”). What further reduces efforts is the possibility that changes to the prefab (for example the developer decides to give the character another type of weapon) can automatically be applied to all instances of the prefab in all scenes. But it is also possible to make single instances differ from the others (for example there could be a level where the character has no weapon at all). It is very important when working with Unity to understand the concept of prefabs and apply it in a meaningful way.

3.4 MODULE 03: The interface of Unity

Goal: Get an overview of the basic elements in the interface of Unity and learn to navigate through 3D-space

Difficulty: Beginner

Prerequisites: None

When opening a new project (it does not matter for the moment which assets had been imported), the application looks similar to this (without the numbers):

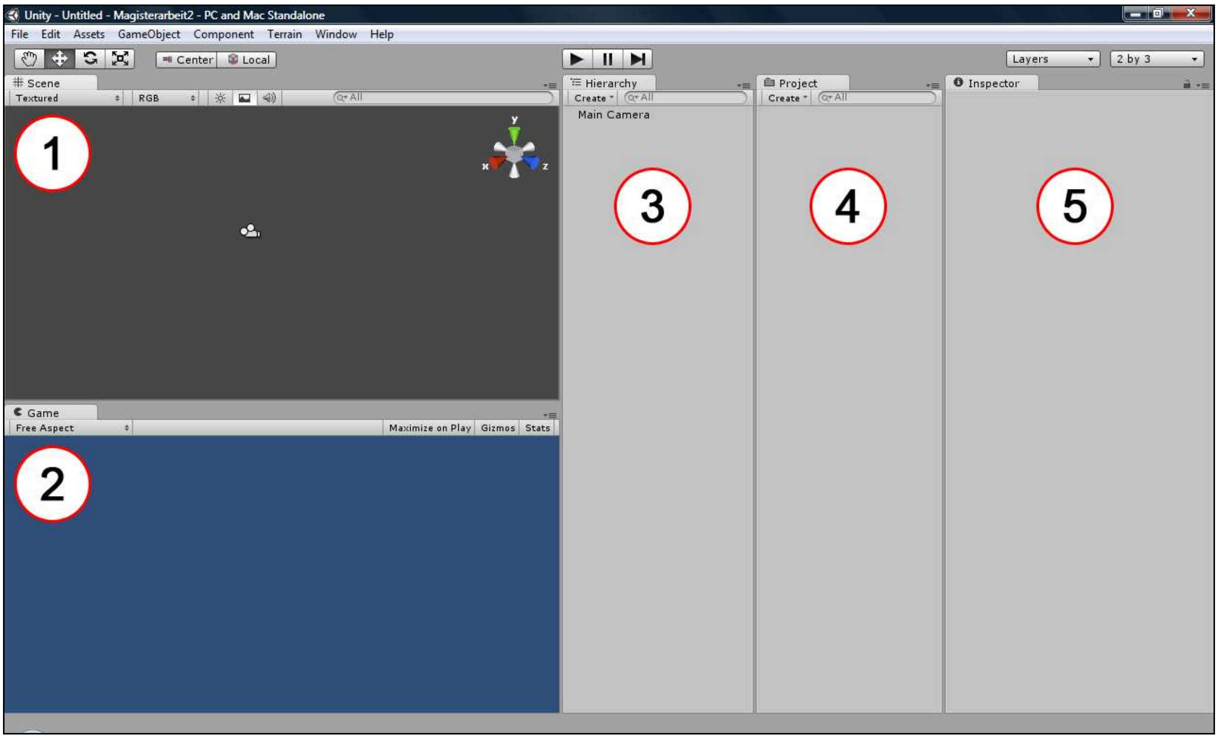


Figure 14: The interface of Unity

It is important to know that one can freely customize the layout's interface. To get a GUI that resembles the screen shown above you have to choose "Window" → "Layouts" → "2 by 3" in the menu. There are other layout configurations possible too, but this one keeps everything important on the screen at once.

The numbers in the screenshot mark the following important areas of the interface:

1. The scene view: This is the "building site" of your application. Here all current objects of the scene can be seen and directly modified.
2. The game view: This is a direct preview of what will be shown when starting the current scene. It has the point of view of the main camera, just like the user has when starting the application. It furthermore is used to preview the game in action after pressing the "Play" button on top of the interface.
3. The hierarchy: Here all game objects of the current scene are listed. In this list all objects can easily be selected, even if they cannot be found in the scene view. In a new project there is only the main camera in the hierarchy, but every other new object will be listed here as well.
4. The project folder: Here all assets that have been imported into the current project are shown. This view resembles the files that lie within the folder of the current project. These assets can be used in all scenes of the application to develop. In fact even the

scenes themselves are files in this folder. At the moment this view is empty (at least if you have not chosen to import assets when creating the project), but will be filled with assets in later modules.

5. Inspector: The inspector gives information and allows modification of the currently selected game object or asset (it is used for both types). Currently the inspector is empty, but if you click on the main camera in the hierarchy view, it will be filled with lots of information. These contents can be very different and depend on the selected object. In the inspector one can analyze and modify the different components of the selected game object. The different components are separated with horizontal rules.

Navigation

One of the most important faculties when working with Unity is to be able to navigate in 3D-space efficiently. This is absolutely necessary when designing an environment with objects in the scene view.

Because it has no demonstrative effect to navigate through an empty space, you need to fill it with a sample object. To do so, please click in the menu on top on “GameObject” → “Create Other” → “Cube”. What happens is that a grey cube will appear in the scene view (and also in the hierarchy). You will now try to navigate around the cube by using different types of movements. Please make sure to **start these movements with the mouse cursor within the scene view**, otherwise you might encounter undesired effects.

If you lose sight of the cube during these experiments and do not find it again, you can click onto the name “Cube” in the hierarchy, move the mouse cursor over the scene view (important!), and press the key “F”. Then the cube will be centered again.

- By holding the right mouse button and moving the cursor around, the scene view rotates around its own axis. It might happen that you lose sight of the cube and you must find it again by properly moving the mouse.
- By holding the middle mouse button the view of the scene can be moved left, right or up- and downwards. If you have no middle mouse button, you can also achieve this by activating the hand-symbol in the topleft corner and holding the left mouse button over the scene view.
- You can zoom in and out by simply scrolling the mouse wheel. This can also be achieved by holding “Alt”, then the right mouse button and moving the mouse. The

latter method has the advantage that you can additionally hold “Shift” to accelerate the zooming speed (which can be useful in large scenes).

- Now hold “Alt”, then hold the left mouse button and move the mouse: This has the effect that the view is rotated again, but this time not around its own axis, but around the axes of the world it shows. This technique can be used to view objects from all angles. Try to see all sides of the cube by using this method.

Please try out all these techniques and learn to know them. To have more objects to see when moving around you can add other game objects in the menu “GameObject” → “Create Other”, like for example a plane, cylinder or sphere.

Another helpful tool to mention is the so-called “scene gizmo”. It is the small object in the upper right corner of the scene view:

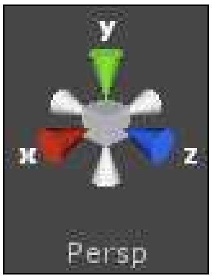


Figure 15: The "scene gizmo"

Visual objects that are part of the interface are called “gizmos” in Unity. The scene gizmo can be used to view the current scene from certain points of view. If you click one of the handles of the scene gizmo (for example the red x-handle), the view changes and shows the scene from a certain side. This can be very useful when trying to accurately position objects.

Please also note that activating one of the handles changes the perspective to “isometric” mode, which causes all objects to remain their size relative to other objects, no matter how far away they are (normally objects that are farther away seem smaller).

To switch back to normal view just click the center (formed like a cube) of the gizmo.

3.5 MODULE 04: Modeling the environment

Goal: Learn how to design the environment of the application, including terrain and objects

Difficulty: Beginner

Prerequisites: Ability to navigate in the scene view, Unity's interface (module 03), basic concepts of 3D-applications (module 01)

This module is separated into several submodules. Each submodule can be worked through on its own.

3.5.1 MODULE 04.01: Modeling the terrain

Most applications need some ground the user's character can walk on or fly over. The simplest way to create a floor for the character is to create a plane. To do so, simply click in the menu "GameObject" → "Create Other" → "Plane". This creates a large, flat object that can serve as provisional ground. You can use this method when experimenting with Unity or building a prototype where you don't need any detailed terrain. Please note that a plane only has one side, so it is invisible when trying to view it from the bottom.

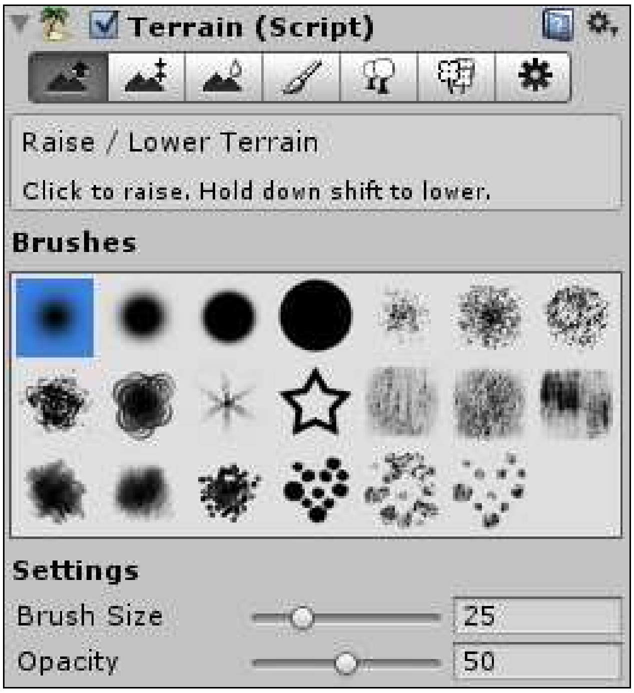
In other circumstances you probably want a more realistic landscape that can be explored by the user. Unity provides a terrain editor that helps in creating such an environment. But before you get into terrain modeling you should import Unity's terrain assets package (if you have not done this already). To do so, please click "Assets" → "Import Package" → "Terrain Assets" in the menu. Make sure all boxes are checked in the following dialog and click "Import". When the process has finished, the project view has been filled with several files and folders that can be used for modeling the environment.

First you need to create a terrain. You can do so by clicking "Terrain" → "Create Terrain" in the menu. This creates a large, flat plane without any details yet.

An important dialog can be shown by clicking "Terrain" → "Set Resolution". Here the options "Terrain Width" and "Terrain Length" are mainly of interest for us. They state the dimensions of the terrain and are both set to 2000 at the moment. You can decrease these values a bit because 2000 is very large for the beginning. Also interesting might be the option

"Terrain Height", which states the maximum height the terrain can have (important for example when modeling mountains). The other options can be ignored for now.

Now select the "Terrain" in the hierarchy. This will cause the inspector view to display certain attributes and tools that can be used. Primarily of interest for us is the component titled "Terrain (Script)":



☑ Terrain (Script)

Raise / Lower Terrain

Click to raise. Hold down shift to lower.

Brushes

Settings

Brush Size 25

Opacity 50

Figure 16: The terrain editor

Before designing the terrain, you might want to change the layout by selecting "Window" → "Layouts" → "Tall" in the menu. This way there is more space for the scene view which is most important when designing the environment.

The modeling of terrain in Unity works similar to drawing a picture in painting applications. But instead of plotting colors onto the screen, you can "paint" mountains, valleys or trees. This can be done by clicking the different icons. The first three tools are used for defining the structure of the terrain:

- By activating the -icon, your mouse cursor becomes a "brush". Try moving the cursor over the terrain while holding the left mouse button. You will notice that many small mountain peaks will appear. The more you move the mouse around, the higher the mountains will become. You can choose various shapes for your "mountain-brush" to achieve different results. Furthermore it is possible to specify the size of the brush and configure its opacity (which is the strength of the brush). By default this tool raises the height of the terrain. But by holding the “Shift”-key while using it, the terrain can be lowered again.

- The -symbol works similar to the tool mentioned before, but it additionally lets you configure the maximum height to which the terrain shall be raised. It has the effect that mountains you create will be flattened exactly at the defined height. By holding shift and left-clicking into an area, the current height of that area will be set as the current maximum height. This tool is especially useful when modeling plateaus.
- The -icon activates a brush that smoothes the height of the terrain. This means that it turns rough height transitions like sharp peaks and chasms into smooth formations.

With the mentioned tools it is possible to shape the terrain as desired, but to make it realistic it furthermore has to be provided with details. To achieve this, the following tools need to be used:

- The selection of the -icon allows to paint textures onto the terrain’s surface. Textures that are used for terrain are called “splats” in Unity.
  Now you can make usage of the “Terrain Assets” package imported previously. To do so, you must click the “Edit Textures…” button. In the following dialog, there is a line called “Splat”, with the value “None (Texture 2D)”. This means that no texture has been selected to serve as “splat”. To the right there is a small circle symbol. By clicking it, you will see the available textures of the current project folder. Select the “Grass (Hill)” texture by double-clicking it and click the “Add” button. This has the effect that the “Grass (Hill)” texture appears in the terrain editor, and also is painted onto the whole terrain that exists until now. But this is done only for the first texture that is being added to the palette (so be careful which texture to add first!). Now repeat this process and add the “Grass&Rock” and the “Cliff (Layered Rock)” textures.
  You should now have three different textures in the palette. Select the “Cliff (Layered Rock)” texture and paint it onto the mountains of your terrain by pointing at them and holding the left mouse button. Again you can configure brush size and opacity. By modifying the target strength you can furthermore set the transparency level of a texture to mix it with other textures.

If you encounter the effect that your textures look very much like repeating patterns, choose the regarding texture by double clicking it and increase the values of “Tile Size X” and “Tile Size Y”. Make sure both have the same value (for now). A texture is being painted on a surface by repeating small image-“tiles” a lot of times. By increasing the tile size, these tiles become larger and therefore are not repeated as much as before (with the drawback that they are not as detailed at close range).

- The -symbol allows to place trees onto the terrain. The same way as before for textures, you first need to add trees to your palette. Choose the “Palm” and any other trees you might like to add. When now “painting” the terrain you will see that the palm trees will appear on the ground. There are several options to configure this process: “Tree Density” states the number of trees placed per click. “Color Variation” makes the trees look slightly different by changing their colors. “Tree Width” and “Tree Height” change the dimensions of the trees (with respect to the original asset’s size) and can be provided with a random factor to make it more realistic.

  To delete trees you have painted, simply hold the “Shift”-key when dragging the mouse over the trees.
- The -icon allows to paint ground details like for example stones, flowers or other plants. For instance you can add grass to your terrain by clicking “Edit Details…” → “Add Grass Texture” and then choose the “Grass2” image. Painting details onto the surface works similar to the “Place Trees” tool.
- By activating the last icon in the row several settings for the terrain can be configured. These affect the level of detail the terrain is rendered, but they are not important for the moment.

3.5.2 MODULE 04.02: Creating Game Objects

When you have created a terrain (it does not matter if it is a detailed environment or just a flat plane for this tutorial), you can get on to the next step: Placing game objects.

A game object can be any object in the environment: A building to explore, a monster that needs to be fought or a vehicle the player can ride. In fact even the terrain itself and the user’s character are game objects.

There are different types of game objects: There are standard objects provided with Unity, and imported game objects modeled with another application. Furthermore there are game objects that have special functionalities like for example light sources. These will be covered in later modules in more detail.

Now create some of the standard game objects that represent the most basic geometric bodies. They can be found in the menu at “GameObject” → “Create Other”. Please create at least one for each of the following objects: Cube, Sphere, Capsule, Cylinder and Plane.

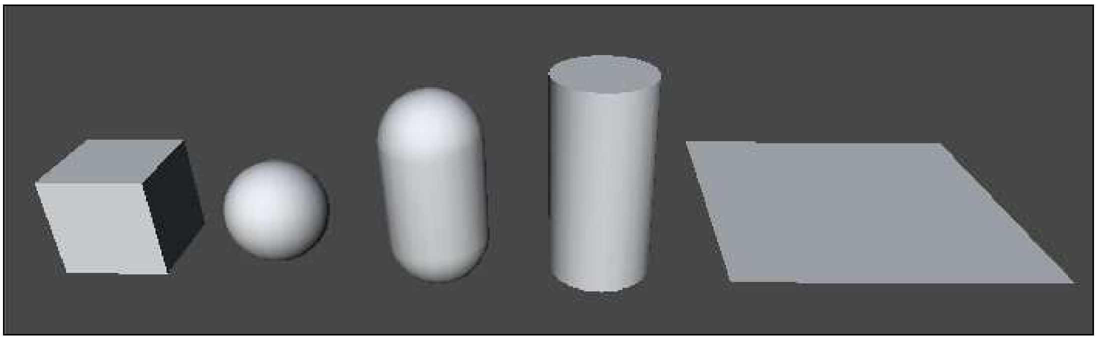


Figure 17: The basic game objects

You can now manipulate the objects you created by selecting one of the icons on the topleft corner:

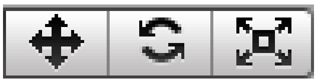


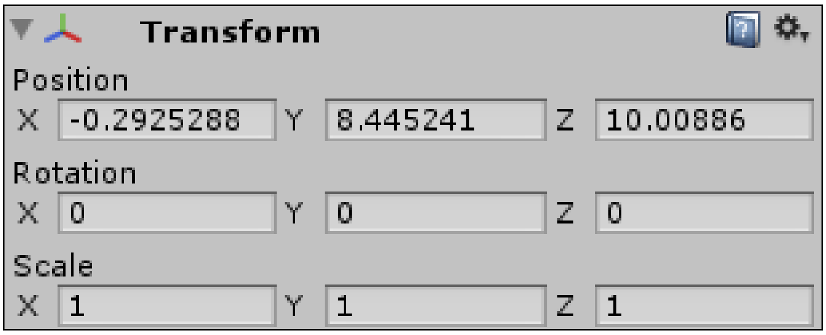
Figure 18: The basic tools for object manipulation

By activating these three icons it is possible to do the basic operations described in module 01: Translation, rotation and scaling.

Activate the first icon and click on one of the objects you created, for example the cube. You will see that the cube will be highlighted and three arrows originate from the object in three different colors: Red, green and blue. This is another “gizmo” that helps in manipulating the object. There are several methods of how to translate (move) the cube now:

- In the center of the object (at the origin of the three arrows) you see a small square. You can left-click the square and drag the object around in the scene view. This method of translating an object is quite simple, but on the other side very imprecise. The reason is that movements of the mouse on flat ground have to be converted into three-dimensional positions. This is not possible without guessing at least one value of the three coordinate axes. To improve this behavior, it is better to choose one side as point of view from the scene gizmo. After that, the cube can be dragged around along two axes precisely.

- Another option that works very well is to click on one of the three arrows, hold the left mouse button and move the cursor. This way the object is only moved along the axis that is represented by the arrow. This method is very useful for precise positioning, but can be a bit tedious when moving the object diagonally (where positions along two or three axes need to be manipulated). By holding the “Ctrl”-key while moving the object, it will be moved in single units.
- Finally the position of an object can also be stated with exact numeric values by using the “Transform” component in the inspector. Use this method when moving the object with the mouse is not precise enough for your purpose:



Transform

Position
X -0.2925288 Y 8.445241 Z 10.00886

Rotation
X 0 Y 0 Z 0

Scale
X 1 Y 1 Z 1

Figure 19: The "Transform" component

The top row of the component is titled “Position” and shows the exact values of the position at x, y and z axis. You can manipulate the values by just typing a new one into the box. Furthermore it is possible to move the cursor over the “X”, “Y” or “Z” letter, click the left mouse button, hold it and move the mouse. The cursor changes its shape and the values can be modified with movements. This way of modifying a box with a numeric value is generally very common in the interface of Unity.

By activating the second icon you can see that the gizmo changes to different circles that contain the game object. In general there are three different gizmos for the basic manipulation operations:

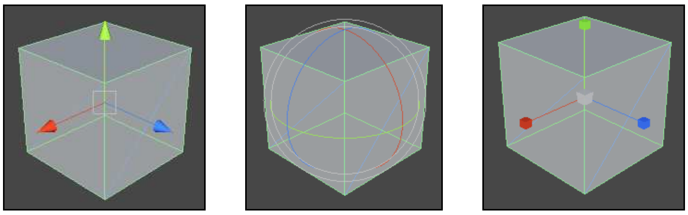


Figure 20: Different "gizmos" for manipulation of objects

The second icon activates “rotation mode”. It works pretty similar to translation but instead of moving the cube around it will be rotated along different axes. Each colored circle will rotate the cube around a specific axis. Furthermore there exists an additional outer area (made up of two grey circles). Rotation along this area is done in relation to the current point of view of the scene.

As with the translation, rotation can also be manipulated using the transform component in the inspector. Values are stated in degrees, so note that a value of zero will show the same rotation as values of 360, 720,… et cetera. It makes sense to let those values not become unnecessary big.

The third icon allows to scale an object. Again this works similar to the other manipulation operations. Note that modifying just one of the axes will either stretch or narrow the cube. To proportionally scale an object click on the center of the gizmo (which is formed like a cube) and then drag the object to the intended size.

You might want to give an appropriate name to an object. To do so, click the object twice with some time in between (like a very slow double-click) and type in the name. It is also possible to select the object and press the “F2”-key.

An important concept when working with game objects is called **“parenting”**. To see this concept in action, create two game objects, like for example two cubes. Now drag one of those cubes in the hierarchy view (not in the scene view!) onto the other.

You will notice that one of the cubes will disappear in the hierarchy view and a grey triangle can be seen. Clicking this triangle will bring back the second cube, but its name is indented now. This indicates that it is now the “child” of the other object, which itself is called the “parent”. Furthermore you might have noticed that the position values of the child cube have changed in the inspector. The reason is that child objects do not have an absolute position referencing the origin of the whole environment, but their position is now relative to their parent's. So a position like for example (5,0,0) means that the child cube is five units to the right of its parent, no matter where the parent currently is located. When moving the parent both objects are moved, but the child remains its position relative to the parent. The same goes for rotation and scaling, both attributes are now given in relation to the parent. So a scale value of two means double the parent's size. If you want to remove the child from its parent, simply drag the child into the grey area of the hierarchy, where there are no other game objects.

This technique is often used to group objects that relate to each other, for example a character and the weapon he is holding.

But sometimes you want related game objects to be grouped, but remain on the same hierarchy level. To achieve this, **empty game objects** can be used.

Please click "GameObject" → "Create Empty" in the menu. A line called "GameObject" will appear in the hierarchy. But you won't be able to see any new object in the scene view. This is because the new game object is "empty", and therefore cannot be seen. But you can still use it as a parent for other objects. To do so, drag two other game objects (that have no parent yet) onto the empty "GameObject". Both will become children now and remain their position, rotation and scale relative to their parent. Be aware of the fact that although the empty game object cannot be seen in the scene view, its transform component still has values for the different attributes. These are the base for the children's attributes.

Importing 3D-models

Until now you just used predefined game objects in the scene. But of course it is difficult to design detailed objects like humans, cars or buildings using the simple shapes of cubes, spheres or cylinders. Unity might be a great tool, but in fact there are applications that are much better suited for performing the detailed modeling of objects. Luckily Unity can import the results of other applications. The following 3D-modeling applications are supported by Unity:

- Cinema 4D
- Blender
- Maya
- 3D Studio Max
- Carara
- Cheetah 3D

- Lightwave
- XSI

Usage of modeling software won't be covered in any details here, in fact it is a course on its own. But in general these applications can export the models and everything packaged with it (like meshes, textures and animations) into the so-called "fbx" file format, which can be used in Unity.

Unity even automatically identifies changes made to the file in other applications, so you can use your favorite 3D-modeler at the same time as working with Unity. You just need to save all exported results into Unity's project folder. A full list of compatible applications can be found at http://unity3d.com/unity/features/asset-importing.

To see an "fbx"-file in action you can import the terrain assets. If they are not already in the project folder, click on "Assets" → "Import Package" → "Terrain Assets". If all boxes are checked in the following dialog, click "Import".

When finished, expand the folders "Standard Assets" / "Terrain Assets" / "Trees Ambient-Occlusion" / "Palm" in the project view. In this folder there is an asset called "Palm" with a cube and small document symbol beside. This is a "fbx"-file that defines the model of a palm tree. By clicking it, the inspector shows all relevant information of the imported file, like information to materials or animations. An important option to configure is the "Scale Factor" from the "(FBXImporter)" component. This setting defines how much the model shall be scaled within Unity in relation to its original size in the modeling application. If you happen to import an object which is gigantic or too small you can tweak this setting.

But for now just drag the palm asset from the project folder into the scene view. You will see that a detailed tree will appear in the scene. Now you can manipulate the palm just like any other object and modify its position, rotation or scale. In general advanced 3D-applications make lots of usage of 3D-models that were designed with professional modeling software.

3.5.3 MODULE 04.03: Materials, Textures & Shaders

You might have noticed that imported 3D-models are often highly detailed, while standard game objects created from within Unity like cubes or spheres all appear just grey. The reason is that 3D-models from other applications are usually already provided with materials, while there are none defined by default for game objects created within Unity. But it is possible to apply materials, textures and also shaders to objects within Unity.

To repeat the difference between materials, shaders and textures:

- Textures are graphics that can be placed onto object surfaces. But because they are only static images, they do not provide any dynamic effects.
- Shaders are definitions of how to render the surfaces of objects (with possible textures on them). In fact they are small programs and are able to create effects like reflections, color toning or transparency.
- Materials unite the different concepts into a single assets. A material can be provided with textures, shaders or other necessary assets. Furthermore different parameters can be configured for the material. Which parameters are available depends on the shader that has been chosen for the material.

You will now see these concepts in action: Create a plane object by clicking “GameObject” → “Create Other” → “Plane”. Now create a new material asset that can be added onto the surface of the plane. Assets are listed in the project folder view, so you need to click onto the “Create” button on top of the asset view (beside the search field). Choose “Create” → “Material”.

An asset will appear in the list called “New Material”. You now have a material, but it is not assigned to any object yet. To do this, you just need to drag the “New Material” asset onto the “Plane” object in the hierarchy. Clicking the “Plane” object now shows the “New Material” in the inspector view. The material itself can now be edited either by selecting the “Plane” object or by selecting the “New Material” asset in the project view, both methods work.

The “New Material” does not show much information yet. The shader is “Diffuse”, which is a built-in default shader. A diffuse shader spreads reflected light very much, in contrary to specular reflection which creates a glossy effect. The following figure illustrates the difference. While the diffuse surface reflects the light evenly, the specular surface highlights the light rays in the direct emerging angle:

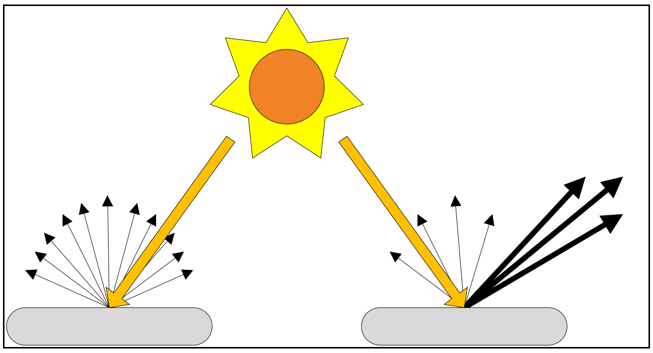


Figure 21: Diffuse and specular light reflection on surfaces

The first obvious thing that can be changed in the “New Material” is the “Main Color”. By default this is white, but by changing it you will see that it is simple to equip the object with a different base color tone. Any changes made to the material can instantly be seen on the object in the scene view.

Now apply a texture to the plane. To do so, you have to click on the “Select” button inside the grey square with caption “None (Texture2D)”. Select the “Cliff (Layered Rock)” texture by double-clicking it (you need to import the “Terrain Assets” for that texture, please do so if not done already). You will see that the plane is now covered with the cliff texture. You may also want to change the “Tiling X” and “Tiling Y” values, which state the number of times the texture image (called “tile”) is repeated on the surface. A value that’s too small might give a washed-out look to the texture at close-range, while a value that’s too high will give the impression of repeating images. A value like “3x3” is probably okay for this purpose.

There are other built-in shaders as well that can be used: Switch the shader of the material from “Diffuse” to “Specular” and have a look at the plane from different angles. You will see that it now seems to be a bit glossy. You can also modify the color of the reflection effect with the option “Specular Color”. The “Shininess” modifies the range of the effect.

Other interesting shaders are the “Bumped” (either diffuse or specular) shaders. These shaders are able to make a texture (that is actually flat) seem to have raised or lowered parts. This effect makes a surface look very realistic because it appears to be three-dimensional.

In order to use bumped shaders effectively it is necessary to provide them with a so-called “Normalmap”. A normalmap is a grayscale image that defines the way the shader shall render the bump effects on the texture. It has to be defined by the developer, but can be done in parallel to creating a texture. “Bumpiness” is an advanced technique and not easy to get done in a realistic way.

There are lots of other built-in shaders that can be used by the developers and cannot be covered here anymore.

Please see the website at http://unity3d.com/support/documentation/Manual/Materials.html for an overview of the different types of materials and shaders.

3.5.4 MODULE 04.04: Light sources

Every scene in a 3D-application usually needs objects that emanate light to make the environment visible to the player. These objects are called “light sources”. They can be created with the menu “GameObject” → “Create Other”. There are three different types of light sources:

- **Directional Lights** are objects that shine on everything in the scene, within infinite range. In fact although directional lights have a position in their transform component, it does not matter at all where to place these light sources. The only relevant transform attributes are the rotational values, that define the angle of the light incidence. Directional lights are most often used as the main source of light in a scene. They simulate the sunlight, so the majority of outdoor scenes that take place during daytime need a directional light. One of these light sources is often enough, but you might want your objects to be equally lit from all sides, then it can make sense to add multiple directional lights with different angles.
- **Point Lights** are light sources that emanate from a single point into all directions. For these types of light the position is very important, because it becomes the center of light emission. But because light is equally emitted into all directions, the rotation does not matter for point lights. Their inspector shows another attribute that is of importance: The range. It defines the distance the light is being emitted, and is illustrated with the yellow circles around the object in the scene view when selecting it. These types of light sources suit very well to realistically simulate light bulbs, lamps or fires.
- **Spotlights** are the last type of light source. As the name already suggests, they shine from a specific point into a particular direction (like real spotlights on stage do). So it is important to define both the position and the direction as intended. For spotlights it is possible to define the distance it shines with the option “Range”. Furthermore the option “Radius” makes it possible to configure the angle the light is emitted: It allows

to model a small ray of light, or a broad emission that covers lot of ground. Spotlights are ideally well suited to simulate objects that work like flashlights.

All three types of light sources have two other configuration options that are of interest here: They let the developer define the color of the light and its intensity. The default color is white, but you maybe want a fire for example to emit a yellowish colored light. By modifying the intensity it is possible to make the light of the source brighter.

Try out all different types of light sources. You will notice that after creating the first light source, the scene view will disable its “standard” lightening, that helped in making the first objects and terrain.

Lightening helps to make scenes more atmospheric, but it is not an easy task. Start with a directional light source so the majority of the scene is well-lit. Try to highlight points of interest with additional light sources. Of course you need further light emitters for indoor environments and objects like lamps, fires or car headlights. Preview your results from different angles and often see what they look like when trying out the game.

3.5.5 MODULE 04.05: Skyboxes

When starting a project from scratch you might have noticed that the background in the game view (or the preview of the game that starts when pressing the “Play” button on top) is just blue, without any other details. In an actual application you probably want something like blue sky with clouds, a sunset or starry sky.

This can be achieved in Unity through the usage of a so-called “skybox”. A skybox is made with a “cubemap”, which means that six textures are put together in the form of a cube. The whole environment then is put into this “cube”. By using seamless textures the player cannot see that he is inside a cube but perceives the textures at the horizon. Similar to reality, the skybox is something that can be seen but the player can never actually get to it.

To apply a skybox to your project you might first want to import the standard set of skyboxes delivered with Unity. To do so, please click “Assets” → “Import Package” → “Skyboxes”. When all boxes are checked in the following dialog, click “Import”.

It is very easy to define a skybox for the current scene. In the top menu you need to click on “Edit” → “Render Settings”. No dialog does appear now, instead the inspector shows different configuration options. There you need to click on the small circle beside the option

called “Skybox Material”. Choose one of the skybox materials (for example one of the “Sunny Skyboxes”) by double-clicking it. You will notice that your scene view immediately becomes provided with a beautiful sky.
The standard assets contain different skyboxes for different types of weather and also for nightskies, so you may want to try them out and see what they look like. If you do not want to see the skybox while working on your scene you can deactivate it by clicking the
🖼 -icon on top of the scene view (note that this will not deactivate the skybox in your final application, but only when working on it).

If the standard assets do not provide an appropriate skybox for your purpose, it is also possible to make one on your own. To achieve this, you have to perform 5 steps:

1. You need 6 images that represent the sky you want to display. Each one of those images should correspond to one of the sides of the cube. Make sure that adjacent images intertwine seamlessly. Then import those textures into your project’s asset folder (you might want to put them into a new folder to group them together).
2. The next step is to select each one of the textures and change the option “Wrap Mode” from “Repeat” to “Clamp”. This is necessary to display them correctly in the skybox.
3. Now you need to create a new material. Click the “Create” button on top of the project folder view and choose “Material”.
4. Select the material you just created and change the shader in the inspector view: Choose “RenderFX” → “Skybox”.
5. The inspector now shows 6 different slots, one for each side of the cube: “Front (+Z)”; “Back (-Z)”, “Left (+X)”, “Right (-X)”, “Up (+Y)” and “Down (-Y)”. Click through the different “Select” buttons of each side and choose the corresponding textures you imported before.

You might want to assign the material a significant name. Then you can apply it as skybox to your scene in the same way that was described before.

3.5.6 MODULE 04.06: Particle systems

Unity has a built-in type of game object that can make the environment of an application much more realistic and dynamic: Particle systems.

These are special objects that are able to emit lots of small 2D images, called particles. There are plenty of configuration options for them, so particle systems can be used for several different purposes like for example explosions, fire, smoke, waterfalls or magic effects.

Figure 22: Simulating fire with a particle system

You might want to have a look at the different particle systems that come with Unity's standard assets. To import them, just click "Assets" → "Import Package" → "Particles", check all boxes in the following dialog and click on "Import". You can now see different objects in the folder "Standard Assets" / "Particles" (in corresponding subfolders). To add one of those particles to your current scene, you just need to drag the object (like for example "Fire1" or "Flame") into the scene view. You will notice that the particle effect is being animated right away in the scene view. But this is only the case if the particle system is the currently selected object. If you unselect it, the animation stops.
Take a look at the different particle systems that come with Unity to get an impression of what's possible with these objects before you will build your own particle system.

It is important to know that a particle system consists of three different components:

- The **Particle Emitter** is responsible for creating the particles. It is possible for example to configure the area where they are generated, their size and the number of particles. There are two types of particle emitters: Ellipsoid and mesh particle emitters. The type defines the outer bounds of the area where the particles are being emitted.
- The **Particle Animator** is the component that changes particles over time, therefore "animates" them. The most important options are different colors that can be applied to make the particles change their color tones, or also rotation effects.

- The **Particle Renderer** defines the details of how each particle will be drawn onto the screen. It can apply materials to particles that make it possible to use textures and shaders for the particles.

Each of these components has several options that can be configured in the inspector. Not all of them are analyzed here, but the most important ones. You might want to create a particle system by yourself and see the effects of changes in these options in action:

Ellipsoid Particle Emitter

- **Emit:** If not activated, no particles will be generated by the object.
- **Min / Max Size:** Defines the size the generated particles will have. By defining different values for min and max particles will have a random size in between the values. Small particles could be used for dust or smoke, large ones for simulating clouds or fog.
- **Min / Max Energy:** Defines the min / max time span a particle will be shown until it disappears. It has to be defined in seconds.
- **Min / Max Emission:** This option states the min / max number of particles that will be created every second.
- **World / Local / Rnd Velocity:** These options can be used to configure the speed of the particles in the directions along x, y and z. Negative values are possible. The differences between the world and local options are the coordinate system in which the values are given – for the first option in world coordinates and for the second option in coordinates relative to the emitters local system. The “Rnd” option allows to apply a random velocity – that is between the value defined and its negative magnitude – in each direction to the particles.
- **Tangent Velocity:** Allows to define the speed of the particles along different axes across the outer bounds of the emitter.
- **Angular / Rnd Angular Velocity:** Allows to generate the particles with a “spinning” speed, making them rotate around their own axis. The “Rnd” option allows a spin in a random direction.
- **Rnd Rotation:** If this option is enabled, particles are already created in a random angle.
- **Simulate in Worldspace:** If enabled, the generated particles will not move with any translation applied to the emitter itself. You can easily try this out in the scene view:

Activate this option and move the particle system game object. You will notice that the particles already generated won't be moved as well, although fresh particles will appear at the emitter's new position. Now try the same with the option deactivated: You'll notice that all particles "stick" with the particle emitter now.

- **One Shot:** If "One Shot" is activated, all particles appear at once (and also disappear at once). This might be useful for effects like explosions, splashes or shots, but probably not for things like smoke, dust or snow.
- **Ellipsoid:** Defines the size of the area where the particles are being generated.
- **MinEmitterRange:** Defines an area in the center of the particle system where no particles are being created.

An alternative to the "Ellipsoid Particle Emitter" is the "Mesh Particle Emitter". Here the area where the particles are being generated can be defined freely with a given mesh.

Particle Animator

- **Color:** Probably the most important option for the "Particle Animator" component is the ability to animate the particles with different colors. There are five different slots for colors in the inspector. By clicking onto the color box you can change each one. Furthermore it is possible to modify the transparency level by changing the value of "A" (which stands for "Alpha"), 255 means "no transparency". If the checkbox "Does Animate Color?" is activated, each particle will change their colors along the 5 given colors. It is very important to choose suitable colors when creating particle effects.
- **World / Local Rotation Axis:** Allows to define an axis the particles will rotate around while moving. This axis can be defined in local or in global space. Please note that this effect will only take effect if velocity (or force) was applied to them.
- **Size Grow:** As the name already suggests, it makes particles grow in size over time.
- **Force / Rnd Force:** Force can be applied along all axes to the particles. "Rnd Force" allows to state a random factor. Force makes the particles move in the defined directions. The difference to velocity: Force also accelerates the particles, while velocity defines a constant speed.
- **Autodestruct:** Allows to automatically destroy the particle system game object when it stopped emitting. This is useful to save processing time, because particle systems can require lots of hardware resources.

Particle Renderer

- **Materials:** The Particle Renderer can define one or more materials that will be displayed as particles. If more than one material is defined, they will be combined.
- **Stretch Articles:** Defines the way of how to render the particles. The standard value is "Billboard", which means that the particles will always face the user of the application, no matter from which point of view he looks at them. There are other values possible too that cause different orientations and behavior.

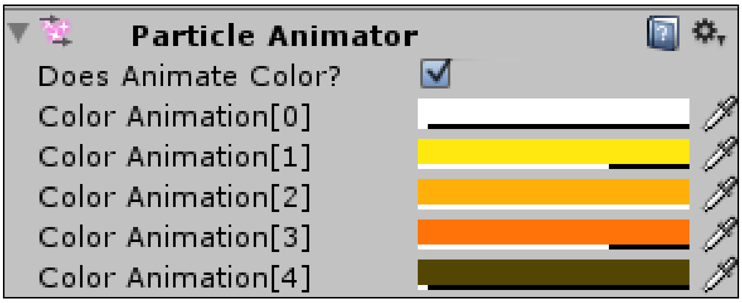
Example: "Starting" a fire

To see how particle systems can be used in applications you will now try to model a small fire (like in Figure 22) by using one of these game objects. So first create a standard particle system by clicking onto "GameObject" → "Create Other" → "Particle System".

First of all the particles need to be larger in size. So change the value of "Min" and "Max Size" to 0.5. The energy defines the lifespan of the particles, and therefore will determine the height of the fire. Set both values to 2 for now. Because you want a dense fire set "Min" and "Max Emission" to 80.

It makes a fire appear more realistic if it seems to burn upwards. So you'll make the particles move along the y-axis: Set the "Local Velocity" of "Y" to 0.6. Furthermore the "fire" burns in a very wide area until now, so change the "Ellipsoid": Set x, y and z to (0.5, 0.3, 0.5).

The general shape of the fire seems to be okay, but next comes one of the most important things to configure: The colors. Now it is only white, but a fire needs color tones with yellow and orange. Try to set the colors to similar values as shown in the following screenshot:



Particle Animator

Does Animate Color? ☑

Color Animation[0] ⬜

Color Animation[1] 🟨

Color Animation[2] 🟧

Color Animation[3] 🟧

Color Animation[4] 🟫

Figure 23: Animating the colors of the particles

When you have finished animating the colors, your particle system should resemble a small fire. Of course there are always many ways of how to simulate an effect. You can tweak the current fire to get different effects: Try to make it wider to burn on a larger area. You could also change the velocity values to simulate the effect of blowing wind. Finally it is a good idea to make a second particle system above the fire that simulates smoke.

3.6 MODULE 05: Unity prefabs

Goal: Learn to know and use the concept of Unity prefabs. Furthermore make use of the "First Person Controller" and preview the application.

Difficulty: Intermediate

Prerequisites: Basic concepts of Unity (module 02), ability to navigate in the scene view (module 03), ability to model a simple environment, parenting, game objects (module 04.02)

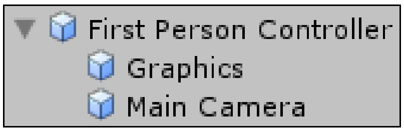
So-called "Prefabs" are one of the fundamental concepts in Unity. Prefabs allow to make game objects reusable, which otherwise would only be possible for assets. In fact prefabs are game objects converted into a special type of asset.

When working with prefabs one must differentiate between original **source prefabs** and **prefab instances**. A source prefab is an asset in the project folder view. It is not present in any scene of the application. To use the prefab in the application, you need to create an "instance" of the source prefab, which is just a kind of copy. Multiple instances can be created from the same source prefab, and they can be used in one or several different scenes. Very important is the fact that all instances are linked to the source prefab. This means that modifications only need to be applied exactly once instead of once for every instance.

These issues will be dealt with later in more detail. First try to instantiate a prefab that's already provided with Unity. To do so, please click on "Assets" → "Import Package" → "Character Controller". Make sure all boxes are checked in the following dialog and click the button "Import".

When the process is finished a new folder called "Standard Assets" / "Character Controllers" has been created. It contains different assets, especially interesting for now is the "First Person Controller".

In general "Character Controllers" are game objects in Unity that represent the user of an application and can be controlled by him using his keyboard, mouse or other forms of input. The "First Person Controller" is a Unity built-in type of prefab that can be controlled by the player, and shows the environment through a first-person perspective (meaning that the player does not see the character that represents him, he directly “sees through his eyes”). Click on the small grey arrow beside the “First Person Controller” and you will see some more objects:



- First Person Controller
  - Graphics
  - Main Camera

Figure 24: The "First Person Controller" prefab in the project folder view

The cube icon beside an asset signifies that it is a prefab (not to be confused with 3D-model assets which have a similar icon!). Every prefab can consist of one or more game objects. You can see that this prefab consists of three objects: An empty game object called “First Person Controller” that is the parent of two other objects: “Graphics” and “Main Camera”.

Now “instantiate” (which means “creating an instance”) the first person controller. Make a small scene with an object that can be used as ground, for example by creating a plane.

It is very easy to get the prefab into the current scene: Just drag it from the assets view into the scene view over the plane you created.

Important: Drag the “First Person Controller”, and not the “Graphics” or the “Main Camera” object into the scene view, otherwise the objects will be disassembled!

Now you will see an object like shown in the following figure:

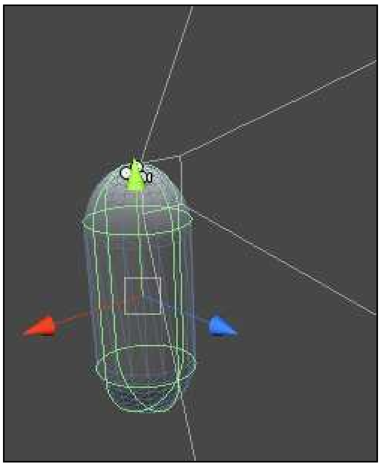


Figure 25: The "First Person Controller" prefab in the scene view

The empty parent object groups two different child objects together:

- The **“Graphics”** is an capsule object that represents the bounds of the player character. It does not have to be any more detailed than it is, because the player can’t see the capsule when playing the game.
- The **“Main Camera”** is attached on top of the capsule and are the “eyes” of the user in the application. The application will show exactly what the camera sees. In the scene view the area that is covered by the camera’s field of vision is marked by grey lines forming a trapezoid.

The objects of the prefab also contain all necessary script components that are necessary to react to user input and move the object accordingly. This means that the prefab is ready for usage. Make sure the graphics object of the first person controller is slightly above solid ground (e.g. the plane). If it is not, it can fall through.

In Unity it is always possible to preview the current application in action without the need to start the whole building process (which would take longer). To do so, you just need to click the ▶ -button in the upper area of the interface. You can now control the character in the game view. Move it using the arrow keys on the keyboard, turn it and change the direction it is looking by moving the mouse. By clicking the ⏸ -button you can pause the current preview. By clicking the ▶ -button again you can stop the current preview. The third button will not be used for now.

Note that it is possible to modify your objects in the inspector or the scene view during preview mode, for example to change the position of an object. **But any changes applied during preview mode will be lost when exiting preview!**

It is very important to bear this in mind, because it often happens that changes are made in preview mode by mistake and are lost afterwards.

After you have used a prefab provided by Unity, you will now try to create your own and see the particularities when working with prefabs. To do so, you will create a simple pillar and convert it into a prefab so it can be reused.

First create an empty game object and rename it to “Pillar”. Then create a cylinder and a sphere game object. Put both of them as children of the empty pillar object. Set the position of the cylinder to (0, -0.5, 0) and the position of the sphere to (0, 1, 0) – remember that these positions are not absolute, but in relation to the empty parent object.

You now have built a primitive object that could be used as a kind of pillar (with some imagination). If you need another pillar object, you would have to redo this process to build another one.

But luckily you can take advantage of Unity’s prefab mechanism: To do so, click in the menu “Assets” → “Create” → “Prefab”. A new asset called “New Prefab” will appear in the project folder view. The grey cube signifies that it is empty yet, but can be filled with an object. Rename the “New Prefab” to “Pillar”, and then drag the “Pillar” from the hierarchy view onto the empty prefab you created. Make sure to drag the parent object (and not the child sphere or cylinder!) onto the prefab. The grey cube icon will become blue now, which means that the prefab is no longer empty. Furthermore you might notice that the pillar in the hierarchy view is now written in blue. This indicates that the instance is “connected” to the source prefab.

If you would delete the pillar object in the scene view now, you could easily rebuild it with the prefab, which has all necessary information stored. Now build a scene that contains 4 pillars in a row. You can add a new pillar by just dragging the prefab into the scene view (like done with the “First Person Controller”).

Next select the “Sphere” object under the “Pillar” prefab in the **asset view** (not in the hierarchy or scene view!). If you want pillars with higher peaks, you would have to change the scale factor. Change the scale factor of Y to 2. You will see that all four pillars in the scene view instantly become higher. The reason is that all instances are connected to the source prefab. All changes applied to the source are automatically applied to all instances.

After that change the scale back to the value 1 (after clicking on the sphere asset).

What if you just want one of the pillars to be higher than the others? Select one of the pillar-spheres in the hierarchy view, and set its y-scale factor to 2. You will see that this time, only one of the four objects changed. It is possible to modify the attributes of one of the instances without affecting the source (and the other instances).

Now the three buttons on top of the selected instance become interesting: **“Select”**, **“Revert”** and **“Apply”.** By clicking the “Select” button the source prefab of an instance is selected in the asset view. This is a handy feature, but you don’t need it for now.

By clicking the “Revert” button all changes made to the current instance are undone, and its state becomes the state of the source prefab. Make sure the sphere that is higher than the others is still selected, and click the “Revert” button. You will see that it becomes the normal size just like the others.

Change the y-scale of one of the spheres back to 2. Now click the “Apply” button: All other spheres will have become the same size now too. The reason is that by clicking “Apply”, all changes made to the current object are copied to the source prefab. And because all changes to the source are automatically applied to all instances, all other pillars in the scene increase their height as well.

Be aware of the fact that these **modifications are always processed per object**. This means that changes made to the child sphere won’t be applied to the source when selecting the parent or the cylinder object and clicking on “Apply”. It is necessary to select the sphere itself.

It is also possible to modify other attributes than the scale of the transform. Select a sphere and click the “Is Trigger” checkbox of the “Sphere Collider” component (its use will be dealt with in later modules). You will see that the “Is Trigger” font now becomes bold. This indicates that the value is different from the original source prefab. You can click the “Revert” button, or right-click onto “Is Trigger” and select “Revert Value to Prefab” to restore the original values. The latter method allows to restore single attributes without affecting others.

All changes done until now to the instances retain the connection to the source prefab. Only if they are connected they can automatically adapt changes made to the source. There are two types of modifications that cause objects to lose connection to their source prefab:

- Adding or removing a component to one of the objects of a prefab.
- Adding or removing a whole child object.

If the user tries to perform one of these actions, Unity displays a warning message: “This action will lose the prefab connection.” The user then has to confirm his decision, which makes sure the connection is not getting lost by accident.

But even if the connection was lost, it can be restored. Right-click on one of the pillar-spheres in the hierarchy and choose “Delete”. Choose “Continue” in the following dialog. One of the pillars will now have lost its sphere. It furthermore is no longer blue in the hierarchy view. But the inspector now shows different buttons: “Select”, “Reconnect” and “Apply”. If you click “Reconnect”, the sphere will appear again and the object will become blue in the hierarchy. If you delete the sphere again and press “Apply”, all spheres will disappear.

"Apply" restores the connection by applying all changes (in this case the removal of an object) to the source. So all other instances lose the sphere as well. Note that **this cannot be undone with "Edit" → "Undo"**, because it directly changed the asset file!

The following diagrams illustrate the different states of source prefabs and their instances:

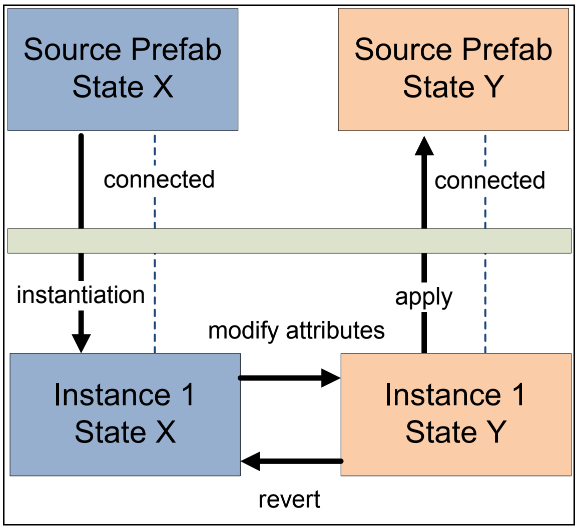


Figure 26: Modifying, reverting and applying the attributes of a prefab instance

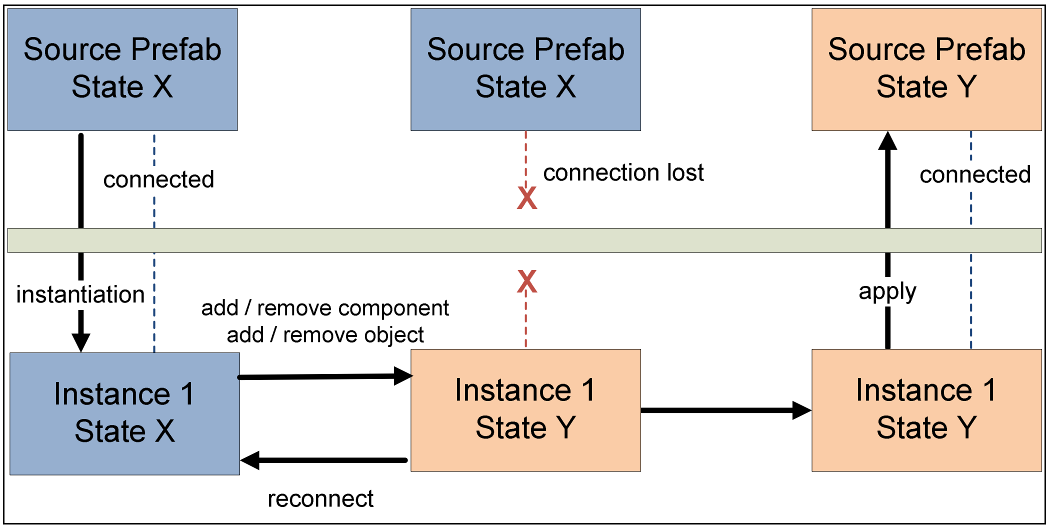


Figure 27: Losing and restoring the connection between an instance and its source prefab

Though prefabs are one of the most important and useful concepts offered by Unity, they are also a very common error source. The dependencies between the source prefabs and their instances can easily cause confusion.

Make sure to be very careful when modifying prefabs:

- Usually it's best to modify the source prefab and not an instance, unless you deliberately want to have an instance to be different than the others!
- Only use the apply, revert and reconnect buttons when you really know the effect it will have! Do not click them thoughtlessly!

3.7 MODULE 06: Script programming

Goal: Learn the basics of script programming in general and specifically for Unity

Difficulty: Intermediate / Advanced

Prerequisites: Basic knowledge about game objects (module 04.02)

Writing script-programs for Unity (also called "scripting") is certainly one of the most important tasks when developing 3D-applications. Only through scripting it is possible to precisely define the logic of an application, e.g. what happens under what circumstances.

This module is separated into two parts: The first one is about programming concepts in general that do not apply to Unity in particular. The second part describes scripting with Unity and brings examples of how to achieve different tasks. You can skip the first part if you are already familiar with programming in general, but make sure to read the second part.

3.7.1 MODULE 06.01: Basic concepts of programming

Unity is a tool that combines new ideas with traditional concepts that have proven of value over a long period of time. When it comes to programming, Unity allows to use well-known concepts, but tweaks them in a way to make them fit to the rest of the tool. This way persons who have already done some scripting can utilize their knowledge and still take advantage of Unity's benefits.

But first of all: Why do you need programming? Why is it necessary to write complex scripts at all? The reason is simple: It is an absolute must to exactly tell the computer how to handle different situations. The software does not know that pressing the forward key should move the player character forward 5 meters by your intention. A priori it is not clear that contact between the player character and the enemy should reduce the health bar by 10%. Such actions are part of the so-called "application logic". Though Unity helps in many ways, most of the application logic's details have to be defined by programming scripts.

Like with other technologies as well, in Unity script files have to be written by using a programming language. This language exactly defines the formal constraints that have to be met for files to be valid. These constraints include the names and characters that can be used, and the whole structure a script has to follow. Unity provides support for three different programming languages: JavaScript, C# and Boo. They cannot be mixed within a single file, but multiple files can use different languages in the same project. Because it is the language most commonly used, you will only see **JavaScript** in these modules.

Programming in Unity works pretty much the same as programming with other tools. Every script can make use of the following concepts:

Commands or instructions are lines in the script that tell the computer to perform a certain action. In JavaScript they always have to be ended with a semicolon “;” character. For example a command can set a value of something using the “=” character. The following (example) command sets the “speed” to the value of 50:

```
speed = 50;
```

Of course it is also possible to execute multiple commands successively. They are executed one after another within split seconds by the machine. The following lines of code set the value of something that damages the character to 10. Then the “health” of the character is set to its current value minus the damage it took. Finally a command is called that updates the “healthBar” which shows the current health to the player. Be aware of the fact that the code lines shown in this module are only examples and probably won’t work in your own project one-to-one:

```
damage = 10;
health = health – damage;
healthBar.update();
```

Any programming language furthermore supports the usage of **comments**. Comments are text and sentences in the programming file that are ignored by the computer. They only have meaning to persons who read the program. They are very useful to document the code (which can become quite complex after time) and make it easier to understand. It is a very recommendable practice to add lots of comments to a program.

A comment that only goes over a single line has to be marked with double-slashes “//” at the start. A comment that goes over multiple lines has to start with “/*” and end with “*/”. Take a look at the comments in the following example:

```
// at first we initialize the highscore
var highScore = 0;
var currentScore = 30; // the current score is 30
var scoreMultiplier = 2; // multiplier is 2

/* We have all necessary parameters.
So we set the new highscore now. */
highScore = currentScore * scoreMultiplier;
```

It is also very easy to do basic **arithmetic operations** in JavaScript: You just have to use the correct character for the operations of addition, subtraction, multiplication and division. JavaScript also allows to easily add or subtract 1 from a value by using the “increment” or “decrement” operators, marked by double plus or double minus characters:

```
result1 = 5 + 10;    // result1 is 15
result2 = 5 – 10;    // result2 is -5
result3 = 5 * 10;    // result3 is 50
result4 = 5 / 10;    // result4 is 0.5

result1++;           // result1 is now 16
result1--;           // result1 is now 15 again
```

In the preceding examples a programming concept was used that is essential in almost any scripting program: **Variables**. A variable is some kind of container that can store data for later usage. A variable has a name and its value can be set with the “=” operator. There can also be lines that simply “declare” a variable (tell the computer that it exists) without setting its value. Then it is necessary to use the “var” keyword. The following lines demonstrate the usage and declaration of variables:

```
var totalScore;

scorePerEnemy = 50;
enemiesDestroyed = 22;
totalScore = scorePerEnemy * enemiesDestroyed;
```

Until now just numeric values were saved in the variables. But it is also possible to save other so-called **“datatypes”** in variables. The most important datatypes in Unity are:

- **String**: Saves text. It can consist of characters, numbers and punctuation marks. If you define text in programming code it has to be declared within “double quotes”.
- **Integer**: Stores an integer number, hence a number without any decimal places.
- **Float**: Contains a floating point number that can have decimal places.
- **Double**: The same as “Float”, but is able to store even more decimal places and can be more precise if necessary. But “Double” variables also take up more memory of the machine.
- **Boolean**: A logical datatype that can only have one of two values: “true” or “false”.

It is optional to declare the datatype for a variable. But generally it is good practice because it can prevent some problems. To declare a datatype, the “:” character has to be used. The following lines of code show the declaration and instant setting of different datatypes:

```
var sentence : String = "Hello Player!";
var integerNumber : int = 5;
var floatingPointNumber : float = 3.1415;
var preciseFloatingPointNumber : double = 3.14159265358;
var logicalValue1 : boolean = true;
var logicalValue2 : boolean = false;
```

It is furthermore possible to set the so-called **“visibility”** for variables. The visibility defines whether or not a variable can (programmatically) be accessed from other scripts or by the user through Unity’s interface. The latter issue will be a topic of the next module.

There are two different types of visibility: “Public” and “Private”. The first means “accessible from anywhere”, the second “only accessible within the script”. If you define no visibility for a variable, it automatically becomes “public”.

For beginners it is not fundamental to care about visibility. When using a variable, ask the following questions: Is it only used within this script? Do other scripts need access to the variable? Is it important to tweak the variable's initial value often? This might help to choose the right visibility. The following lines of code show how to set the visibility:

```
var highScore = 0;                  // automatically "public"
public var currentScore = 100;      // declared as "public"
private var scoreMultiplier = 2;    // declared as "private"
```

Another important concept that exists in most programming languages are **conditional statements**. These are commands that will only be executed if a logical condition (a condition that can be evaluated to be "true" or "false") is "true". Such a condition is marked with the keyword "if", the condition itself has to be placed inside parentheses. It can contain comparison operators like "<" (smaller), ">" (greater), "<=" (smaller or equals), ">=" (greater or equals) or "==" (equals) and "!=" (not equals). Be aware of the difference between "=" and "==". The first one is used to set the value of variables, while the second one checks for equality of two values.

It can also contain variables of type "boolean". The code that shall be executed when the condition is true must be inside "{…}"-parentheses. The following example sets the highscore to the current score's value if it is greater than the highscore was before:

```
if(currentScore > highScore){
   // this part is only executed if the condition is true:
   highScore = currentScore;
}
// what comes here is executed in any case afterwards
```

It is also possible to provide an alternative that will only be executed if the condition is false. This means that only either the first code or the alternative will be performed. An alternative has to be marked with the "else" keyword:

```
if(currentScore > highScore){
   message = "Congratulations! New Highscore!";
}
else{
   // this part will only be executed if the condition is false
   message = "Try again!";
}
```

There is also the opportunity to provide the “else” part with a condition. Then it is necessary to write “else if”, together with the condition. It is possible to state an arbitrary number of “else if” conditions. But be aware that they will be checked from top to bottom, and if one evaluates to “true”, all others won’t be checked anymore!

```
// only one of the following 3 instructions will be executed
if(currentScore > highScore){
   message = "Congratulations! New Highscore!";
}
else if(currentScore == highScore){
   message = "You tied the Highscore!";
}
else{
   message = "Try again!";
}
```

In general condition blocks have the following structure:

- They always start with an “if” statement including a condition.
- Then optionally follow an arbitrary number of “else if” statements with conditions.
- Finally there can be an optional “else” statement (without condition).

Of course it is also possible to use several complete condition blocks consecutively.

Conditions can furthermore be assembled with several smaller conditions. Therefore the logical operations “&&” (called “AND”) and “||” (called “OR”) can be applied. They allow to state conditions like “if condition1 AND condition2 are true”. The following example sets the “message” to “Game Over!” if the user has no more health AND no more lives left. The message is set to “Level solved!” if there are EITHER no more enemies OR the key to the exit was found:

```
if(currentHealth == 0 && currentLives == 0){
   message = "Game Over!";
}

if(numberOfEnemies == 0 || exitKeyFound == true){
   message = "Level solved!";
}
```

By using the exclamation mark “!” (called “NOT”) a condition can be negated. The following example negates the previous one. Also note the extra parentheses that are necessary to negate the whole condition and not just the first part of it:

```
if( !(numberOfEnemies == 0 || exitKeyFound == true) ){
   message = "You can't get to the next Level yet!";
}
```

Another very common structure in programming languages are **loops.** By using loops a certain part of the code can be executed several times. There are different types of loops.

The first one is the “for”-loop. It is marked with the keyword “for”. It needs two instructions and one condition as arguments (in the order: instruction 1, condition, instruction 2 – separated by semicolons). The code inside the “for”-loop is executed as long as the condition evaluates to “true”. The first instruction is executed once (and only once!) at the very beginning of the loop, and the second one is executed at the end of each single loop iteration. “for”-loops are usually used to execute some code a defined number of times. The following example code adds the value 20 to the current score for each enemy that has been destroyed. Therefore a “count” variable is introduced and set to zero at the beginning. Then it is incremented as long as it is smaller than the value of “enemiesDestroyed”:

```
var count : int;
for(count = 0; count < enemiesDestroyed; count++){
   currentScore = currentScore + 20;
}
```

Another type of loop are “while”-loops. These are less complicated, because they only need a condition in their declaration. The code inside the loop is executed as long as the condition is true. The following demonstrative example code creates new enemies as long as there are less than 50 enemies:

```
while(enemiesCount < 50){
   createNewEnemy();
}
```

A similar type of loop is the “do-while” loop. It also only needs a condition. The difference to the “while”-loop is as follows: The “while” loop checks if the condition is met, and then executes the code statements inside. But the “do-while” loop first executes the statements, and then checks the condition afterwards. So it is possible that the code in the “while”-loop is not executed at all, but the “do-while” always processes the code inside at least once!

Usage looks similar to the following example. It increases the lives of the player until the maximum is reached. But no matter how high the maximum is set, the lives are at least increased by 1:

```
do{
   playerLives++;
}while(playerLives < maxLives);
```

When working with loops it is very important to check that the conditions are designed in a way that makes it always possible for the computer to exit the loop. If this is not the case, there is an **infinite loop** which will cause the application to crash. So always make sure that the conditions defined in the loops will evaluate to “false” at some point!

The final general concept of programming that will be described in this module are **functions**. These are constructs that allow to use the same lines of code at different locations without the need to write them multiple times. They can be defined once and “called” (meaning that the code in the function will be executed) anywhere in the script. A function must be defined by using the keyword “function” like shown in the following example:

```
function createNewEnemy(){
   enemiesCount++;
   message = "Alert! New enemy detected!";
}
```

A call to the function to execute it always contains its name and parentheses:

```
createNewEnemy();
```

It is not only possible that functions contain commands to execute, they can also return values. In this case the keyword “return” has to be used. A call to such a function can be utilized to get a result for further processing:

```
function getTotalScore(){
   totalScore = scorePerEnemy * enemiesDestroyed;
   return totalScore;
}

message = "Your total score is: " + getTotalScore();
```

This makes even more sense when introducing parameters to a function. Parameters make functions flexible. They allow to adapt the call of a function to the current situation by passing the values given in the call to the code in the function. To do so, variables are declared when defining the function. These variables can be set when calling it:

```
function getTotalScore(scorePerEnemy : int, enemiesDestroyed : int){
   totalScore = scorePerEnemy * enemiesDestroyed;
   return totalScore;
}

message = "Your total score is: " + getTotalScore(50, 22);
```

In general programming is a huge topic and cannot be handled in further details here. But the concepts shown are the most fundamental ones that should be clear to every Unity developer.

3.7.2 MODULE 06.02: Programming scripts for Unity

There are some particularities a developer needs to be aware of when creating scripts for Unity.

First of all: A script itself is a normal asset in the project folder. But it won’t be executed unless it is applied to a game object (or prefab). When applying a script to an object, it becomes a **component** of this object. But the script file still can be modified, though it is applied as a component of an object. The same asset file can be set onto several different objects. But all of those components execute the code from the same asset file!

There can be parts of a code that refer to a certain object. In general these parts automatically refer to the object the script is attached to. If the script is attached to multiple game objects, than the script is executed for each object separately.

Another fundamental issue are special functions that need to be defined when working with scripts. When writing scripts with Unity, most code has to be inside of a function.

But not any arbitrary function: Unity provides different function names that represent different actions that may happen during the execution of an application. Unity automatically performs the code defined in such a function when the corresponding action happens. For example, there is a function that is called automatically when the user clicks onto an object in the scene. The developer can now define the commands that shall be executed when this action happens by defining the corresponding function. You will see this in action later. Some examples of the most important functions that are provided with Unity are:

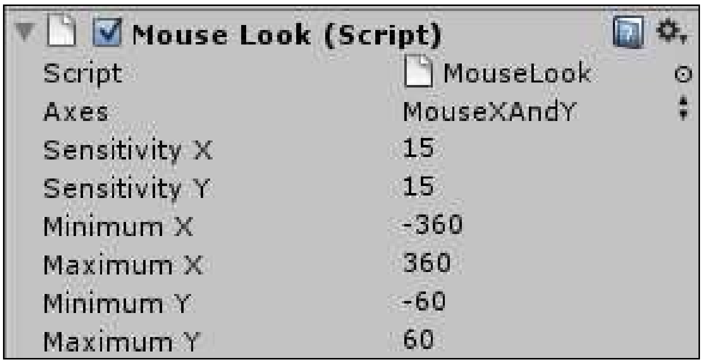
- **Update**: This is probably the most central function. It is being executed during every frame during the whole application. Place code that needs to be performed all the time into this function.
- **FixedUpdate**: While “Update” is called every frame that is rendered by the engine, “FixedUpdate” is only executed at certain frames, depending on the physics engine. Put code that manipulates object physics (see later modules) into this function.
- **Start**: “Start” is only called once at the begin of the application. It can be used for example to initialize the state of an object.

- **OnMouseDown**: This function is automatically being called when the object the script is attached to is being clicked on by the user. This could be used for instance for a top-down strategy game, where the user can give commands through clicking.
- **OnTriggerEnter & OnCollisionEnter**: These functions are being called when two different objects touch or “collide” with each other. They will be handled in more detail in module 7.

You can make use of an arbitrary number of functions in a script, there is no limit. But of course the code should always serve the purpose, so make sure that you only use functions that are really necessary! A complete list of all available built-in functions can be found at http://unity3d.com/support/documentation/ScriptReference/MonoBehaviour.html. There also the exact definition of the functions (including parentheses and parameters) can be found.
Important: Unity provides these “built-in” functions, but don’t forget that it is also possible to define your own arbitrary functions in a program!

Most code you write can be put into either a built-in function or functions you created on your own. But it is also possible to write code into the script without any function surrounding it. This code will be executed when the game object that has the script attached is loaded. But it generally can be recommend to put any initializing commands into the “Start” function.

However there is one thing that makes sense to put code outside any function: Variable declarations. If you declare a variable inside a function, it will only be accessible from within this function. This makes sense in some cases, but often you want a variable to be accessed from the whole script. Then you need to declare it outside. If it is defined as “private” it can only be modified from within the (whole) script, “public” means to make it visible to other scripts and objects as well. Declaring a variable as “public” (or with no defined visibility) has another consequence: The value of the variable can be manipulated conveniently through the inspector in the interface. This looks similar to the following figure:



☑ **Mouse Look (Script)**

Script	MouseLook
Axes	MouseXAndY
Sensitivity X	15
Sensitivity Y	15
Minimum X	-360
Maximum X	360
Minimum Y	-60
Maximum Y	60

Figure 28: Public variables can be modified with the inspector view

Here the script component called “Mouse Look” (which is a script provided with Unity’s standard assets) has the following public variables: “Axes”, “Sensitivity X”, “Sensitivity Y”, “Minimum X”, “Maximum X”, “Minimum Y” and “Maximum Y”. The values of these variables can be modified directly through the inspector view.

Be aware of the fact that this **only sets the initial state** of the variables, which might be changed during the running application. Furthermore note that though you change the values in the inspector, the initial values in the script asset itself (if they were set at all) do not change. **The values set in the inspector always override the original values defined in the script file**. This behavior is a source for lots of mistakes when working with Unity!

You can reset all values set in the inspector to the original values in the script by clicking the “cog”-icon in the upper right corner of the script component and choosing “Reset”.

You will now build a simple example script that demonstrates some concepts and allow the simple movement of an object. Create a “Capsule” game object that will serve as provisional player character and name it “Player”. Also change the layout by clicking “Window” → “Layouts” → “2 by 3”. This will allow to keep the capsule in the field of sight if it moves away.

Now create a new JavaScript file by clicking on “Assets” → “Create” → “JavaScript”. An asset called “NewBehaviourScript” will appear in the project folder. Rename this file to the name “PlayerMove”. When it is selected, the inspector shows the contents of the file. It was automatically filled with an empty “Update” function that’s ready to be used.

But you cannot edit the file in the inspector. To do so either click the button “Edit…” in the inspector view or double-click the asset. This will open the editor for modifying scripts. Now put the following lines of code into the script and save it:

```
public var speed : float = 10.0;

function Update () {
   var translationForward : float
                  = Input.GetAxis("Vertical") * speed;
   var translationSideward : float
                  = Input.GetAxis("Horizontal") * speed;

   translationForward  = translationForward  * Time.deltaTime;
   translationSideward = translationSideward * Time.deltaTime;

   transform.Translate(translationSideward, 0, translationForward);
}
```

On top a public “float” variable called “speed” is declared that shall define how fast the player moves in the application. It is initially set to 10.0, but this can be modified through the inspector.

Then comes the “Update” function: Everything inside this function will be executed during every rendered frame of the application, several times per second. Inside two other “float” variables called “translationForward” and “translationSideward” are defined. They shall contain the value the character object shall be moved in the current frame, either sideways, forward or backward.

The function “Input.GetAxis” is a function provided by Unity and tells us if the player pressed an arrow-key on the keyboard. The parameter “Vertical” checks for the “up” and “down” keys, the parameter “Horizontal” for “left” and “right”. If one of the keys was pressed, the function returns either 1 or -1 (depending on the direction, 1 is for up and right, -1 for left and down).

But you probably want the player to move at a certain speed, and not only 1 unit per frame. So the value returned by the function (which is either 1, -1 or 0 if no key was pressed at all) is multiplied with the “speed” variable declared at the beginning of the file.

The next two lines modify the “translation” variables again by multiplying them with “Time.deltaTime”. This is a function provided by Unity that allows to make a game independent from the current framerate. As you already know the computer renders a certain number of frames per second. How many frames can be rendered depends on the hardware of the machine: A high-end machine may constantly render 70 frames per second, while another one may only produce 20 frames per second. But you don’t want the character to move faster on a faster machine (because there the “Update” function is called more often per second).

This is where the “Time.deltaTime” value can be used. By multiplying the current speed value with it, it will be reduced proportionally to the amount of time it took to render the frame. This has the effect that the character will move 10 units per second, and not 10 units per frame. Multiplication with “Time.deltaTime” is often necessary when performing an action every frame including numeric values.

Finally you have the exact values to move the player object in the current frame. It can be done by using “transform”: This refers to the “transform” component of the game object the script is attached to. It offers several functions, one of them is “Translate”. “Translate” needs three numerical parameters, defining how much to move the object into the directions of x, y and z. Put the value for sideward translation as the parameter for x, and forward translation as parameter for z. Y remains to 0, which means that the player cannot move up or down.

Now that the “move” script is ready to use it needs to be applied to a game object. To do so, just drag the asset from the project folder view onto the “Player” object in the hierarchy. You can now see a new component in the inspector of the object called “Player Move (Script)”. It furthermore shows the “Speed” parameter with a current value of 10.

Center the capsule object in the scene view, and click the “Y”-handle on the scene gizmo to view it from top. Do that to make sure you can see the object actually moves when pressing the buttons (you could also adjust the “Main Camera” to see the object in the game view).

Now press the preview button (the “Play” button in the upper area) and try to move the capsule around by pressing the arrow keys on the keyboard. If everything has been done correctly, the capsule should be moving according to the buttons that were pressed.

After that change the “speed” value in the inspector from 10 to 20 and preview again. You will see that the object instantly moves faster. It is even possible to change the value during preview, but then the changes will be lost when stopping the preview.

When you changed the value of “speed” in the inspector you will notice that the first line in the script itself has not changed, it still shows a value of 10. Remember that values set in the inspector will override any settings from the file!

To reset the “speed” to the value of the file, click on the “cog”-icon of the component in the inspector and choose “Reset”. The “speed” will be 10 again.

The functions shown in the example script were just a small excerpt of the functions provided by Unity. Other important examples are:

- **transform.Rotate**: Allows to rotate an object.
- **transform.LookAt**: Rotates an object in a way to directly face another object.
- **Debug.Log**: Prints the given text message to the console. Use this function to test and evaluate your application.
- **GetComponent**: Returns a component of the defined type of the current object.
- **GameObject.Find**: Returns a game object with the given name.
- **Instantiate**: Programmatically clones a given object at a defined position with defined rotation.
- **animation.Play**: Starts the animation attached to this game object.
- **audio.Play**: Starts to play an audio source attached to this game object.

There are countless other functions provided by Unity that cannot be mentioned here as well. If you don't know which function to use or how to define a specific function, always refer to the **Scripting Reference**. The Scripting Reference is a documentation of all objects and functions available when programming with Unity. It also provides full-text search for when you don't know exactly where to find something you need. Many code examples can be found in the reference which you can adapt to solve your own tasks. The Scripting Reference can be viewed with an internet browser and is opened automatically by clicking "Help" → "Scripting Reference" in Unity's menu.

3.8 MODULE 07: Physics and interaction between objects

Goal: Learn to know how "triggers" and "collisions" can be used for interaction between objects, what are the differences, how to implement physical behavior and learn to know the concept of tags

Difficulty: Advanced

Prerequisites: Basic knowledge about game objects (module 04.02), prefabs (module 05), knowledge about programming scripts (module 06)

One of the most important things when developing 3D-applications are the interactions between different objects. There are different types of interactions, but the primary concept that is of interest here are “collisions”. A collision between two objects happens when the two different objects touch each other. To be more precise: When the “colliders” of the objects identify that the objects have contact.

“Colliders” are components in Unity that automatically check for collisions. A collider “weaves” an invisible web around the object it is attached to. When another object (with a collider) crosses this web, Unity registers a collision to which one can react by using scripts.

There are different types of collider shapes. A collider can precisely match the object it encloses in every detail (a so-called “mesh collider”), or just have a simple shape that does contain the object, but does not match every detail (for example “Sphere Colliders” or “Box Colliders”). Though a collider is invisible in the final application, a green “web” around the objects signify it in the scene view. The following figure shows three different colliders around the same object, each one matching the object differently:

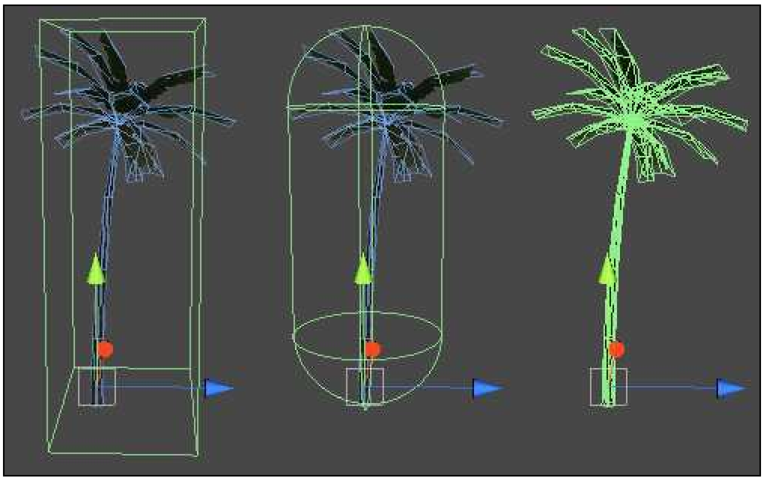


Figure 29: Different types of colliders on the same object

As you can see in the figure, only the last collider – the “mesh collider” – exactly matches the palm object. The other colliders would identify collisions even when the other object did not precisely touch the actual palm.

So why not always use “mesh colliders”? The answer is because of processing efforts. It takes up much more resources of the machine to check if any object exactly touched any of the palm leaves, than just checking if an object is within a certain distance of the palm. Furthermore in many cases it is not necessary to check collisions that precise. In a fast-paced action game the player could never see a difference in the range of some units. So always try to use simple-shaped colliders if possible.

Colliders will be dealt with later, but first another related concept in 3D-applications and Unity is introduced that can add much realism to an application: **Physical behavior**.

Unity has a built-in “physics engine”, a technology that calculates the behavior of objects in realtime under consideration of their mass, the force that acts upon them and other physical objects in proximity. For example this allows to roll an object down the hill in a realistic manner without any necessary programming.

In Unity physical behavior is not applied to game objects per default. They need to be provided with a special component that marks them as “physic objects” and allows configuration of physic parameters. This component is called “RigidBody”.

You can try out how physical behavior looks like in action with a simple example: Create a plane that serves as ground for the objects. Now create two cubes and one sphere object. Place them in some height above the ground, and also above each other. Now attach the “RigidBody” component to all three of the objects by selecting them and clicking “Component” → “Physics” → “RigidBody”. Make sure you can see the objects in the scene view, and press “Play” to preview the result: The objects should fall down to the ground and realistically collide with each other, the sphere will probably roll away. You can also play around with the “Mass” parameter in the inspector of the “RigidBody” component. Objects with higher mass will tend to push objects with lower mass aside.

You will now build a small game that makes use of physical objects and demonstrates some concepts. The purpose of the game shall be to hit some targets by shooting balls.

First create a plane that serves as ground for the objects. Put some cubes onto the plane that shall be the targets for now. Increase them in size to make it easier to hit them. Make sure they are on the plane (not below or intersecting), so they cannot fall through. Add a “RigidBody” component to all the cubes and rename their names in the hierarchy to “Target1”, “Target2”, “Target3”,… and so on. Test that the targets don’t fall through the plane by previewing the application.

Next you need to add the player to the game. If there is still a “Main Camera” object in your hierarchy: Delete it, because you will use a different camera object.

If they are not already in your project folder, import the “Character Controller” standard assets by clicking “Assets” → “Import Package” → “Character Controller”. After that choose the “First Person Controller” prefab from the “Standard Assets” / “Character Controllers” folder that was created and put it into the scene. Make sure it stands on the plane and cannot fall through. Test if it can be controlled correctly by previewing.

Then you need something to shoot for the player. Create a simple “Sphere” object and provide it with a “RigidBody” component. Because you want the player to be able to shoot lots of objects and not only one, make a prefab out of the sphere. Click “Assets” → “Create” → “Prefab” and drag the sphere object onto the grey “New Prefab”. Rename the prefab now to “ShootingBall”. Because you can use the prefab now, you don’t need the actual sphere in the scene anymore, so delete the “Sphere” object from the scene view.

Now it is time to equip the player and make him able to shoot balls in the direction he is currently looking. Create a new JavaScript file in the asset view, rename it to “PlayerShoot” and fill it with the following code:

```
var impulse = 100;
var objectToShoot : Rigidbody;

function Update () {
   if(Input.GetMouseButtonDown(0)){
         var clone : Rigidbody;
         clone =
Instantiate(objectToShoot, transform.position, transform.rotation);
         clone.AddForce(
transform.forward * impulse, ForceMode.Impulse);
   }
}
```

Note that you can insert line breaks at different positions than shown in the script above, it does not matter where to put them when using JavaScript.

The script contains several interesting parts of code: At the beginning two different (public) variables are declared. The first one is a simple number, but the second one is of type “RigidBody”. This demonstrates that it is not only possible to use numeric or text variables, but also complex objects as variables. It is called “objectToShoot” and will be set to the projectile used for shooting later.

Then follows the “Update” function: It contains code that only is being executed when “Input.GetMouseButtonDown(0)” is true, which is the case when the user clicks the left mouse button in the application. Inside another variable which also is of type “RigidBody” is declared and called “clone”. Because the player may shoot an arbitrary number of projectiles, the program needs to produce a “clone” of the original object every time he clicked the left mouse button.

This is done by using the “Instantiate” function: It expects an object to clone in the first parameter (the original “objectToShoot”) as well as values for position and rotation.

By using the name “transform” you can access the transform-component of the current object the script is attached to. “transform.position” gives the position and “transform.rotation” the rotation of the current object. So a clone projectile is created at the exact position and with the same rotation as the object the script is attached to.

Creating the object to shoot is not enough, it also has to fly away. This is done in the last line beginning with “clone.AddForce”. This is a method that can be called on a “RigidBody” object and allows to “push” a physical object. “transform.forward” gives the direction you want to push the object (in the forward direction of the current “transform” component), and by multiplying it with the public “impulse” variable, you can adjust the shooting speed in the inspector of the script by changing the value of “impulse”. The second parameter “ForceMode.Impulse” just says that the “push” is applied once in form of a single impulse, and not constantly.

You want the player to shoot the balls in the direction he is currently looking, so save the script now and attach it to the “Main Camera”, a child object of the “First Person Controller” in the hierarchy view. A dialog will warn that “this action will lose connection to the prefab”. This does not matter for now, so just click “Continue”.

You can now see the “Player Shoot (Script)” component in the inspector when selecting the “Main Camera”. There you can modify the two public variables defined in the script: The “impulse” makes it possible to configure the speed of the balls shot away.

The other one was a variable of type “RigidBody”. It is currently set to “None” and expects some object with a “RigidBody” component. Such a public object variable can simply be set in Unity by dragging it into the inspector: To do so, make sure the “Main Camera” is still selected, and drag the “ShootingBall” prefab from the asset view onto the “Object To Shoot” variable in the inspector. It is important that you **hold** the left mouse button on the “ShootingBall” prefab, otherwise it will be selected and the inspector view changes. If you have done it correctly, the “Player Shoot (Script)” component will now show the “ShootingBall” as value for the “Object To Shoot”.

After that it is time to test the application. If all steps were performed correctly, then it should be possible for the player to walk around and shoot balls into the direction he is currently looking by clicking the left mouse button. Try to hit the cubes by shooting them: They should be pushed away in a realistic way when colliding with a ball.

Now improve the little game even further: It could be a nice addition if the application would register actual hits of the targets. This could be used for example to increase the score of the player with every hit. But for now you will just give out a message to the console.

To achieve this, implement some code that reacts to collisions. Create a JavaScript file in the project folder and rename it to “ObjectCollision”. Put the following code inside and save:

```
function OnCollisionEnter(collision : Collision) {
   Debug.Log("COLLISION DETECTED !");
}
```

The function “OnCollisionEnter” is automatically called by Unity if the object the script is attached to starts a physical collision with another object. To be more precise: If the “Collider” components of the objects get in contact with each other. The parameter variable named “collision” of type “Collision” holds various types of information about the collision itself, this will be used later.

Now attach the “ObjectCollision” script onto the “ShootingBall” prefab. Because the script has no public variables, it does not need any further configuration. Preview the game and shoot the targets. You will see the “COLLISION DETECTED !” message on the bottom of the interface. It shows the last message of the console. To see all messages, click “Window” → “Console”. A window will open that shows all messages. If you hit more than one target, there should be more than one message.

Now try the following: Start the preview and shoot a ball into open space, make sure you don’t hit any of the targets, and also not the ground. You will see that the “COLLISION DETECTED !” message will be shown again. Why is this the case even when you don’t hit a target?

The answer is that the sphere is created at the position of the player character, and therefore Unity identifies a collision with the “First Person Controller” object. But you want the code only to be executed when a target cube is hit. So you need to somehow “mark” the cubes as “targets”. This is where the concept of tagging comes into play.

Tagging is a concept that is very common in various software applications. It is being used to group different items that share something in common. For example webshops or blogs use it to group articles that are concerned with the same topic.
In Unity you can use tags to mark certain objects. A tag can be any kind of name, for example “Player”, “Enemy”, “Item”, “Spawn Point”,… and so on.
You will now mark the target cubes with the tag “Target” to be able to distinguish them from other objects. To do so, select one of the cubes. On top of the inspector there is the word “Tag” displayed. Click the box with the value “Untagged” beside it: It shows some predefined tags, but the name “Target” is not among them. So choose the option “Add Tag…”.
This changes the inspector to view the “Tag Manager”. Click on the grey arrow beside the “Tags” option, which opens all existing user-defined tags. It is a list where “Element0” is the first tag, “Element1” the second one,… etc. It is currently empty and only shows “Element0”. Click into the empty field right from “Element0”, write “Target” inside and press enter.
Instantly “Element1” appears and would allow to add further tags. But for now that’s enough tags, instead select a target cube. Now click the “Tags” box again, and choose the “Target” tag that is available now. Repeat this selection for all target cubes, so they are not “Untagged” anymore.

Now modify the “ObjectCollision” script to make use of the introduced tag. Change it like shown in the following listing:

```
function OnCollisionEnter(collision : Collision) {
   if(collision.gameObject.tag.Equals("Target")){
             Debug.Log("COLLISION WITH TARGET !");
   }
}
```

The script now accesses the “collision” parameter that contains information about the collision itself. It allows to access the other game object that collided with the object the script is attached to. Any game object allows to access its tag-name. All three steps together are written like “collision.gameObject.tag”, with dots in between every access step. “Equals” is a method that allows to check if a value (for example a string) is equal to another given value.

Here the parameter “Target” is given. So the condition is only true if the tag from the other game object in the collision equals the name “Target”. If this is the case, the message “COLLISION WITH TARGET !” is printed to the console.

Now preview the application again. Shoot the balls into air – nothing should happen. Then hit a cube (that was tagged before) – the message should be shown.

Trigger

The techniques shown in the example before work pretty well when you want to identify physical collisions through script, thus collisions that inflict some kind of force upon each other. But it is very often the case that you want to register that an object touched an invisible collider without actually being affected by the collision. An example could be the case that you want to register if the player entered (or left) a certain area without notifying him to launch an event (for instance alert some enemies). If you want to detect contact between colliders without any physical force involved, Unity provides the concept of **triggers**.

So you will improve the small game even further: Because it is very easy to shoot the targets at close range, you want to identify if the player is within a certain area at which the distance is long enough.

To do so, create a cube object. Now stretch the cube in its scaling dimensions to be very large. It should cover about a third of the ground plane, on the farther side of the targets. The cube should have the dimensions of an area from which it is okay for the player to start his shooting exercises.

After that, if you preview the game you will notice that there is a cube that might be the “shooting range”, but the player cannot get inside. So change that now: Stop the preview, select the cube and check the “Is Trigger” checkbox in its “Box Collider” component. This makes the collider to act as trigger, and has the effect that the player can directly walk through the object.

Because you furthermore want the cube to be invisible, deactivate the “Mesh Renderer” component: You can do this by unchecking the checkbox beside the “Mesh Renderer” headline of the component. This means that the object is still there, but not rendered anymore by the engine.

You now need a script that reacts to entering and exiting the “shooting range”. Create a JavaScript asset called “ObjectTrigger” and put in the following code:

```
function OnTriggerEnter (other : Collider) {
   if(other.gameObject.tag.Equals("Player")){
              Debug.Log("Shooting range entered! Start shooting!");
   }
}

function OnTriggerExit (other : Collider) {
   if(other.gameObject.tag.Equals("Player")){
              Debug.Log("Left shooting range! Stop shooting!");
   }
}
```

There are two different functions used now: The first one is being executed when an object first collides with the trigger and enters the area of the collider, the second one when the object leaves the area of the collider.

Because you only want to react to the player entering or exiting the shooting range, a check for the tag “Player” is done. Of course this demands that you set the tag of the “First Person Controller” object in the scene to “Player” (which is a predefined tag and therefore does not need to be added).

Now attach the “ObjectTrigger” script to the “shooting range” cube and start the application preview: Try to walk in and out of the area. When entering, the first message should be shown, when exiting the second one. Because of the check for tags no message should appear when shooting balls directly through the “shooting range”.

This was a small overview about the most important concepts regarding collisions and triggers. There are lots of other things to know, you can get more information at http://unity3d.com/support/documentation/Components/class-BoxCollider.html and other documents in the reference manual.

3.9 MODULE 08: Cameras

Goal: Learn the basic concepts of cameras in Unity

Difficulty: Intermediate

Prerequisites: Basic knowledge about game objects (module 04.02), basic knowledge about programming scripts (module 06), tags (module 07)

“Cameras” are special objects in Unity that serve as the “eyes” of the user within the application: A user running the application will see exactly what a camera captures in its field of sight.

If you are using the “First Person Controller” or the “3rd Person Controller” provided with Unity’s standard assets, you probably do not have to care a lot about cameras. These character controllers are already provided with cameras and also scripts that automatically make those cameras controllable and follow the player object.

In case you want to create a camera yourself there are some things you need to be aware of:

- Cameras themselves are invisible objects, but they are marked with a small camera-icon in the scene view. It furthermore highlights the field of sight of the camera with white lines forming a trapezoid.
- Any application should be provided with a “Main Camera”. This is the camera that should be active when starting the application (though this is not always the case) and be the primary view for the player. In general it is a camera like any other one but named “Main Camera” and marked with the tag “MainCamera”.
- When selecting a camera in the scene view a small preview window in the bottom right area is displayed. This shows what the selected camera can “see” at the moment.
- It can be quite tricky to position a camera that shall capture a certain area from a specific point of view. This is quite difficult when trying to apply the correct rotation and position with the inspector. There is a function in Unity that can help here: Select the camera you want to position and change the point of view in the scene view to be exactly where you want the camera to look at. Now click “GameObject” → “Align With View” in the menu. Now the camera will be set at the exact position with the current rotation of the scene view.

- When you have two or more cameras in the scene, you will get the “There are X audio listeners in the scene”-message, where X is the number of cameras. The reason is that cameras do not only “see” for the player, they also “listen” to sounds. For this they use the “Audio Listener” component. But of course it does not make sense to listen from more than one location, there can only be one audio output for the user. To get rid of the warning message, just deactivate the “Audio Listener” components you don’t need (make sure one is still active) by clicking the checkbox to the left from the headline of these components.

There are some configuration options in the inspector for cameras that might also be interesting for Unity beginners:

- The **“Projection”** option allows to choose between “Perspective” and “Orthographic” projection. It defines the type of perspective that is being rendered. The first one is the standard type of three-dimensional projection with objects in distance getting smaller. “Orthographic” enables a similar perspective to what is shown when activating the scene view gizmo to enter “isometric” mode. It means that all objects remain their size relative to each other, no matter how far away they are. “Orthographic” for example can make sense when developing a top-down strategy game.
- The **“Field of View”** allows to configure the view angle of the camera in degrees. A wider field of view allows to capture a larger area, but objects seem to be farther away. A narrower vision captures a smaller region and has the effect of “zooming” onto the objects.
- A camera does not capture all objects at infinite distance or when they are too close. The **“Clipping Planes”** define at which distances the camera starts and stops rendering the environment. The “Near Clipping Plane” defines at which distance objects will be rendered, so objects closer than that won’t be displayed. The “Far Clipping Plane” defines until what distance rendering takes place. All objects outside that distance won’t be captured by the camera.

In some applications you want the possibility to allow the player to switch between different cameras. A racing game for example could let the user switch between the cockpit of the car and an outside view.

To achieve this, the following (improvable) script can be used. It works for two cameras named “Camera1” and “Camera2”, and changes the perspective when the “space” key is pressed. Attach this script to one of the cameras and make sure that only one camera is active at the beginning of the application:

```
function Update () {
   if(Input.GetKeyDown("space")){
             camera1 = GameObject.Find("Camera1");
             camera2 = GameObject.Find("Camera2");

             camera1.camera.enabled = !camera1.camera.enabled;
             camera2.camera.enabled = !camera2.camera.enabled;
   }
}
```

If the “space” key was pressed, the script first searches for the two camera objects by using the built-in “GameObject.Find” function.
It then enables the “camera” component at the camera that was disabled before and vice versa. The exclamation mark “!” is an operator that negates a boolean value. So by writing “A = !A” one can express “set to A the opposite value of what it has now”. This leads in switching the current value of A.

Further script components for cameras can be found in Unity’s menu in the section “Component” → “Camera-Control”.

3.10 MODULE 09: Animations

Goal: Learn basic concepts of animation in general and how to animate objects within Unity
Difficulty: Intermediate
Prerequisites: Basic knowledge about game objects (module 04.02)

In general animations mean the execution of predefined changes for attributes over a certain period of time. This might sound complicated at first, but the concept is quite easy to understand when giving an example: An animation could be an object at position (0, 0, 0) first, then moving to position (5, 0, 0) within the time of 30 frames. Another kind of animation could be a blue light that smoothly changes to yellow within 100 frames.

Animations need to be used quite often in 3D-applications: For opening doors, moving platforms or running enemies. It is possible to animate almost any kind of attribute. Unity provides great support in creating such animations through the interface.

But before you create your own animation it is first necessary to learn to know the basic concepts of animation in general. These are concepts that are not only applied in Unity, but in lots of other software tools that allow to define animations:

Each animation consists of several **frames**. A frame is one “picture” of the animation. When playing some frames consecutively in a short period of time the illusion of movement is created. It is important that there are only small differences between each frame. For example the first frame could be a door in closed state, the second frame shows a small gap which gets slightly larger with each frame. The last frame then shows the door in its open state.

This is where the concept of **“keyframes”** comes into play. When defining an animation, it would be very time-consuming for the developer to design every frame of the animation, thus the example door in all its states from closed to open. But smart software tools allow to shorten this process: A developer just has to define the keyframes, and the animation software automatically calculates the frames between them.

To get back to the example: The user would just define the door in its closed and open state, and the software could work out all frames between closed and open. This is done through a technique called **“interpolation”**. It works like in the following example: An object at position (0, 0, 0) in the first frame and position (10, 8, 5) in the last frame must be at position (5, 4, 2.5) after the first half of the frames passed. This position can be interpolated by the software.

Unity (and also other software tools) provide a comfortable way of analyzing the animations. The changing of the attributes over time can be captured in a two-dimensional coordinate system. The x-axis represents the current point in time of the animation, called **“time line”**. It always starts with zero. The y-axis represents the values of the different attributes, for example the transform’s z-position or the color value. When designing an animation, Unity fills this coordinate system with lines called **“animation curves”**. They represent the changing of the different attributes over time. One complete animation from start to end is called an **“animation clip”**.

You will now do a sample animation to learn to know the concept within Unity. It is also possible to import animations created with external modeling applications using the “FBX-Importer” component, but you will just use Unity here:

Create a new scene and insert a cube at position (0, 0, 0). You can also insert a light source to make things better visible if you want to. Change the window layout of the interface to “Tall” and center the cube in the scene view.

Now click on “Window” → “Animation” in the menu, which causes a new window to appear. You can fixate the floating window in the bottom area of the scene view by dragging the tap labeled with “Animation” into the lower area. Make sure you can see the animation view and the scene view both at once now.

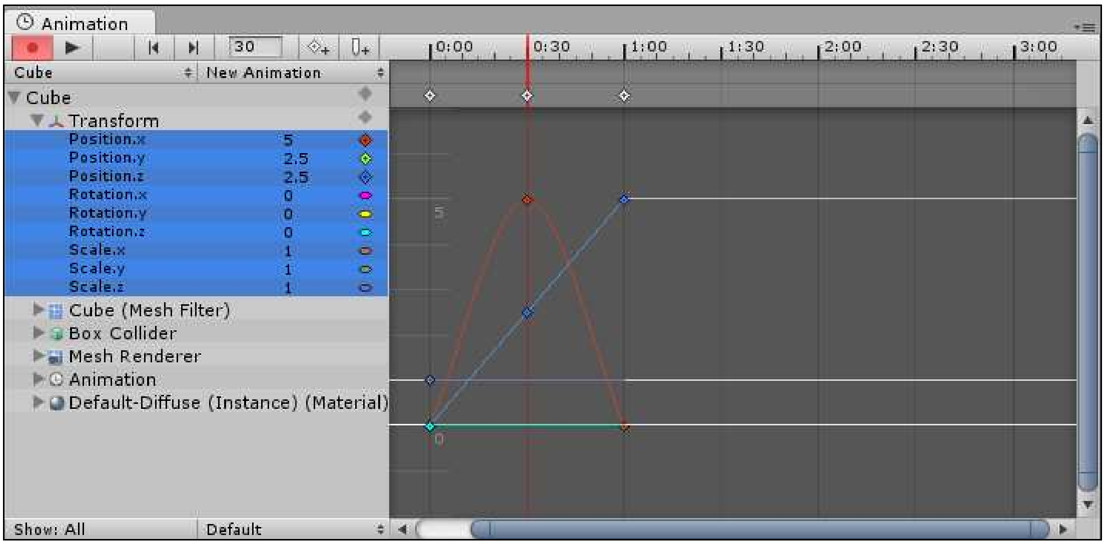
If the cube is selected, the animation view should show several attributes of the cube on the left side, for example the “transform” attributes, collider, material, … and others. These are the attributes that can be animated (which is not the default case for all properties). To start creating an animation, you have to click the red “record” button on the top left of the animation window. When clicking you will be prompted to save a file in a folder. This is because animations are normal asset files that need to be stored in your project folder. Give it an arbitrary name and click “save”. Now you are in “Animation Mode”. All changes made now affect the animation (but not the current state of scene).

The grey area on the right side of the animation view displays the animation curves (when there are any). The top part labeled with time codes like “0:00” and “0:30” is the time line. It can be used to change the current frame of the animation that is being shown. You will see a vertical red line crossing the area. This shows at which point in time of the animation you currently are. You can drag it around by clicking it in the time line (it cannot be dragged by clicking it in the grey area!), holding the mouse button and moving the cursor.

Make sure the red line is at the start of the animation, and click the “Transform” component on the left side of the animation view. This causes all “transform”-attributes like “Position.x”, and “Position.y” to be selected. Now click the “Add Keyframe” button in the top left area of the animation view. It looks like a small diamond with a plus-sign. Small “diamond”-shapes are symbols for keyframes. If you have done everything correct a small keyframe-symbol should have appeared in the upper area of the grey field.

Now drag the red line to the time code of “1:00” (which means 1 second after animation start). Change the position of the cube to (0, 5, 5). Make sure you can still see the cube in the scene view. Another keyframe-symbol should have been appeared. You have now made two keyframes, enough for a small animation. You can click the "Play" button beside the "record" button in the animation view to preview the animation created.

After that change the position of the red line to the time code of "0:30" (which means 30 frames after animation start). Set the x-position of the cube to 5 now, which creates another keyframe. Furthermore a curve has been created by Unity that looks similar to the following figure:



Cube
Transform
Position.x 5
Position.y 2.5
Position.z 2.5
Rotation.x 0
Rotation.y 0
Rotation.z 0
Scale.x 1
Scale.y 1
Scale.z 1
Cube (Mesh Filter)
Box Collider
Mesh Renderer
Animation
Default-Diffuse (Instance) (Material)

Figure 30: Animation curves

The red curve represents the change of the x-position over time. Unity automatically tries to animate smooth transitions, so a round curve without sharp edges appears. When you press the animation-preview button now, you will see that the cube does not only move diagonally upwards, but also makes a curve forward and backwards (of course depending on your point of view).

All other attributes can be changed the same way as done for the position now. It is easy to make a rotating object by changing the rotation attributes over time or make an object that increases in size by modifying its scale. But you can also animate other things than the "transform"-properties: All attributes shown on the left side of the animation view can be animated! This allows possibilities limited only by imagination.

Some more notes on animation:

- When working with the animation curves, you can use the mousewheel to zoom and press the middle-mouse button to move. But be aware of the following fact: The preview-button only plays the part of the animation that is currently viewed via the animation curves! If you want to reset the view press the “F” button while hovering the mouse cursor over the animation view.
- You can change the so-called “wrap mode” of an animation. This defines whether an animation is played once, in a loop, or forward and backward again. Select the animation file in the asset view and change the “Wrap Mode” in the inspector.
- When starting the application, animations are played automatically by default. You can change this by selecting the animated object, and uncheck the “Play Automatically” property in the inspector of the “Animation” component.
- If you want to play an animation programmatically you just have to insert the line “animation.Play();” at the position you want it to play and attach the script to the object with the animation component. If the object has more than one animation you can use a text parameter to choose the intended animation: For example “animation.Play(‘myAnimation’);” or “animation.Play(‘openDoor’);”

3.11 MODULE 10: Sounds

Goal: Learn how to integrate sounds in a Unity application

Difficulty: Intermediate

Prerequisites: Basic knowledge about game objects (module 04.02), basic knowledge about programming scripts (module 06)

A 3D-application becomes much more vivid if the user can hear sounds that dynamically fit to the current situation. Examples for sounds can be things like starting an engine, shooting a weapon, background music or character voices.

When working with sound in Unity, you need to get familiar with three basic concepts:

- **“Audio Clips”** are sound files in the asset folder. They contain the data of the sound effect (or music) to play. Unity is able to support the following file formats for sounds: *.mp3, *.wav, *.ogg and *.aif

- **“Audio Sources”** are game objects with an “AudioSource” component. They need to be provided with an “Audio Clip” to work. “Audio Sources” define the position of the sound in space, their volume, when to play it and other settings.
- **“Audio Listeners”** are components attached to a game object that shall be able to capture the sounds from the “Audio Sources”. The “Main Camera” is automatically equipped with an “Audio Listener”, which is sufficient in most cases. This way the camera does not only represent the “eyes” of the player within the application, but also his sense of hearing. Note that there can only be one active “Audio Listener” per scene.

When working with sounds in Unity, it is important to distinguish between **2D- and 3D-sounds**. The difference lies within the way they are played back to the player: While a 2D-sound always keeps the same volume no matter of the player’s position, a 3D-sound is automatically adapted to its distance to the “Audio Listener”: If the player walks away from the sound, it dies away, if he walks toward it, the volume increases.

This creates dynamic sound effects that increase the realism of an application. Both types of sounds have their use cases: While 2D-sounds are well-suited for instance for background music, 3D-sounds can be used to underlie events with a certain position, for example explosions or gunfire.

You will now see how to create “Audio Sources” in Unity. For this exercise you will need two different sound files: One with music playing over some minutes, and another one with a short sound effect that stops within seconds. Unfortunately it is not possible in this context to provide any sound files, so please search the web to find any two sounds, if you have not already any examples on your computer. See the file-formats before to know what formats are supported.

Put the files into your Unity asset folder by dragging them from your file system into the project folder view. This will cause Unity to automatically import these files.

Select the music-file in the asset view. The inspector will show several properties. The most important one is the “3D Sound” checkbox, which defines whether or not it is a three-dimensional sound. It should be checked for now. Furthermore in the bottom area there is a play-button that allows to preview the sound.

Now create a small scene that allows to experiment with these sounds a bit: Delete the “Main Camera” if existing. Then create a plane as provisional ground and put the “First Person Controller” prefab on it (if you have not already imported it: Click “Assets” → “Import Package” → “Character Controller”). The “First Person Controller” is automatically equipped with an “Audio Listener” component that makes it possible to hear sounds from any sources. But there are no sources yet that could play any sound. So you need to **create an “Audio Source”**.

To do so, create an empty game object. Rename it to “AudioSource” and equip it with the right component by clicking “Component” → “Audio” → “Audio Source”. The empty game object now gets a “Speaker”-icon in the scene view.

You furthermore need to provide the component with an “Audio Clip”. Click the small circle beside the option “Audio Clip” in the inspector of the selected game object. Now choose the music-file asset by double-clicking it.

There are several important options that can be modified in the inspector: “Mute” allows to silent the sound, “Play On Awake” defines whether the sound is started with the current scene and “Loop” repeats the sound if it has finished.

But for now leave all options unchanged. Place the empty game object on the outer regions of the plane, so the player has to walk some distance to get to it. Start the application preview: You should hear the sound of the music playing. Try to walk to the audio source and away again. The volume of the music should change with the distance of the player character. If the music sounds a bit “weird” during walking, try to change the “Doppler Level” option in the “3D Sound Settings” of the “Audio Source” component to zero. This simulates a physical effect that makes sound more realistic, but is not appropriate in all situations.

To model music as three-dimensional sound is useful in situations where the application should create the realistic impression of a speaker that is located somewhere in the environment. But in most cases ambient sound makes more sense in 2D. So you will change this now: Select the music-file in the asset view and uncheck the box at “3D Sound”. Any change here must be confirmed with the “Apply” button below the options. By clicking it, the sound will be converted into a 2D-sound.

When the process has finished preview the application again: You will see that the sound does not change anymore when moving around.

Beside background music “Audio Sources” can also be used to create sound-effects: Short sounds that underlie certain events in the application. Any object can be the source of an audio effect, even the player itself. Select the “First Person Controller” in the hierarchy view and click “Component” → “Audio” → “Audio Source” in the menu. Confirm the following dialog that warns about “losing the prefab connection” by clicking the button “Add”. This will equip the “First Person Controller” with an “Audio Source” component.

You now need to define the “Audio Clip”: Click the small circle beside the option and choose the short sound-file you have imported into the asset folder. The kind of sound does not matter for the purpose. When you preview the application now, the sound effect will be played instantly.

However you want to start the sound by script. So unselect the “Play On Awake” checkbox in the “Audio Source” component of the “First Person Controller”. When previewing the application now, the sound-effect should not be played anymore.

Next you need a script that initiates the sound on command. Create a JavaScript asset and call it “PlayerSound”. Fill it with the following lines of code and attach it to the “First Person Controller”:

```
function Update () {
   if(Input.GetKeyDown("space")){
             audio.Play();
   }
}
```

Preview the application. The sound should be played now when pressing the “space”-key. This way it is easy to play sound per scripting.

Other useful functions for sounds are “audio.Stop()”, which stops playing, and “audio.Pause()” that pauses playing. The function “audio.PlayOneShot(clipObject)” allows to play a certain object of type “AudioClip”, which could be for example a public variable configurable through the inspector.

3.12 MODULE 11: GUI objects

Goal: Learn how to use “GUI Texts”, “GUI Textures” and make a basic menu

Difficulty: Intermediate

Prerequisites: Knowledge about programming scripts (module 06)

When developing 3D-applications, it is often necessary to provide the player with information: Examples are the current score, health, speed, enemies left,… etc. For this purpose it often is not necessary to design complex 3D-objects, simple text is enough. Furthermore it is important that this text is in the players field of sight all the time, and does not get smaller when the player walks in any direction. In other words: It should not be located in 3D-space, just always on the front of the screen.

Unity provides special game objects that easily fulfill this purpose: They are called “GUI Texts”. A “GUI Text” can be created by clicking “GameObject” → “Create Other” → “GUI Text”. When creating such an object it immediately appears on screen. It is an object like any other one, but behaves a bit different: Though it has a transform component, it is not affected by changing the values for rotation and scale. Neither it is affected by the position value at the z-axis.

But the values of x- and y-position change the location of the text on screen. Be aware that the coordinates (0, 0) mark the bottom-left corner, and (1, 1) the top-right corner of the screen. These coordinates are relative to screen size. So if you want a text in the middle of the screen you just have to define the values (0.5, 0.5).

Other important configuration options of a “GUI Text” object in the inspector:

- **Text:** Defines what text shall be displayed by the object.
- **Anchor:** States the position of the text relative to the “transform” component.
- **Font:** Allows to configure the font style of the text. Fonts need to be imported into the project folder to be available. This can be done by simply dragging a *.ttf file into the asset view.
- **Font Style:** Gives the opportunity to make letters bold and/or italic.

If you want to dynamically change the text of a “GUI Text” object, you can use the command “guiText.text = ‘my new text’”. Remember to place the new text to assign under double-quotes!

Besides the possibility to show static text for the player, it is also possible to render flat images onto the screen without any location in 3D-space. They could be used to design a fancy health bar or a map of the current terrain. These game objects are called “GUI Textures”.

You can create a “GUI Texture” by clicking “GameObject” → “Create Other” → “GUI Texture” in the menu. It works quite similar to “GUI Texts”: The position can be set the same way, but note that “GUI Textures” are affected by modifying the scale values in the transform. Furthermore instead of defining a value for the “text” option, you have to specify a texture (a graphical image) with the “Texture”-option in the inspector.

Also the “Pixel Inset”-option now provides to set the “Width” and “Height” of the texture. This allows to set the dimensions from the graphic when rendered. It is useful to prevent them from changing with different screen resolutions. Without specifying these values a “GUI Texture” will scale with the current screen resolution and look very different on different screens. Make sure to set the scale values of x and y in the “transform” to zero when modifying the “Width” and “Height” of the “Pixel Inset”.

Beside the possibilities described to display information on screen, it is also often necessary to show elements the user can interact with. Probably the most common purpose, that is featured by nearly every 3D-application, is a menu that can be used to define several options, start and quit the application. Such a menu must contain elements the user can click to cause certain actions. You can do this with Unity through scripting.

To draw interactive elements onto the screen, you can make usage of the “OnGUI()” function provided by Unity. It is called automatically every frame. The function can be filled with commands that draw different GUI-elements and make it possible to react to interactions.

A very common type of GUI-elements are “buttons”. These are objects that can be clicked by the user which causes a predefined action to happen.

Create an empty game object and name it “GUI” (position does not matter). Now create a JavaScript file named “GuiMenu” and fill in the following code:

```
function OnGUI () {
   if (GUI.Button (Rect (25, 25, 100, 30), "First Button")) {
             Debug.Log("First button clicked!");
   }

   if (GUI.Button (Rect (25, 65, 100, 30), "Second Button")) {
             Debug.Log("Second button clicked!");
   }
}
```

Now attach the script file to the empty GUI object. You only need this empty object to get the script into the scene as a component, otherwise it would not be executed.

Start the preview of the application: There should be two buttons now in the upper left corner: Try to click both of them which should cause the defined text messages to appear in the console.

The interesting part of the script are the conditions of the “if”-statements. They are functions that cause a button to be drawn onto the screen, and also check if it was clicked at the same time.

These functions called “GUI.Button” take two parameters: The first one is a “rectangle” object created with the function “Rect”. It is used to define the position and dimensions of the button. The first two numbers define the x- and y-position on screen, the next two numbers the width and the height of the rectangle (and therefore the buttons). The second parameter is a text that defines the caption of the button.

Other important GUI-elements that can be created through functions are:

- **GUI.Label**: Just creates a text on screen.
- **GUI.TextField**: Draws a field that can be filled with (a single-line) text by the user.
- **GUI.TextArea**: Draws a field that can be filled with multi-line text by the user.
- **GUI.Toggle**: A checkbox that can be either checked or unchecked.
- **GUI.HorizontalSlider**: Draws a control with a knob that can be moved along a bar.

Please refer to the documentation at http://unity3d.com/support/documentation/Components/gui-Controls.html for a complete list of all GUI-elements and how to use them.

3.13 MODULE 12: Publishing the application

Goal: Learn how to publish an application made with Unity

Difficulty: Beginner

Prerequisites: None

Finally when you have finished developing your application, or you just want to test it on another computer, you need to convert it into another format. You can do a preview of the program within Unity, but another machine might not have Unity installed, and furthermore you don’t want your application to be modifiable on other computers, they should only be able to execute it. The process of converting the application into a format that allows execution on other machines without being able to modify it is called “building the application”. The result of this process is a “build”.

It is very easy to build an application with Unity. It needs some configuration to be done in a dialog after clicking “File” → “Build Settings” in the menu. The first thing you have to define is which scenes you want to include in your build. An application can consist of one or more scenes. A scene could for example represent one level of a game. By using the “Add Current” button you can add the scene you are currently editing to the list. Other scenes can be added by dragging them from the project view into the list.

This list does not only define the scenes that shall be included in the build, but also their order. On the right side of each scene an index number marks the position of the scene in the order.

If you want to change from one scene to another within the application, you have to do this by script. It can easily be done by using a function: The call “Application.LoadLevel(2)” would load the scene with index number “2” in the list of the build settings. It is also possible to give a name as parameter like “Application.LoadLevel(‘finalBoss’)”.

You can change the order of scenes by dragging them around in the build settings list. It is furthermore possible to exclude a scene from a build by unchecking the box beside it, or remove it by selecting it and pressing the “Del”-key.

The next important decision is the target-platform of the build. The final application can only be run on machines of the platform type you choose here. It offers the following choices:

- **Web Player**: Publishes applications into a format that allows to play them from a website using an internet browser. But note that the browser needs to be provided with a special plug-in.
- **PC and Mac Standalone**: Allows to choose between Windows and different Mac OS X architectures.
- **iOS & Android**: Builds the application for mobile devices with one of the two operating systems. Not possible in the free version of Unity.
- **Xbox 360, PS 3 and Wii**: Publishes the application for one of the three video game consoles. Not possible in the free version of Unity.

When you have chosen the desired platform (supported by your current version of Unity) you can click “Build”. This prompts you to choose a location for the results of the build process. You can save the files into the root of the project folder, but do not store them directly into the assets folder of the project.

The other option is to click “Build And Run”, which does not only create the build, but also executes it promptly. If you have executed an application that provides no opportunity for the player to quit yet, you need to use your operating system’s keyboard shortcut for quitting. On Windows this can be done by pressing “Alt” and “F4” together.

Unity creates different results depending on the target-platform. On Windows an executable *.exe file along with a folder with suffix “_Data” is created. You need to copy both of them onto another machine for running the application there. For Mac an “app bundle” will be produced that can be executed and also contains all necessary resources. When building for the “Web Player”, Unity creates a HTML template-file together with a data file. The application can be started by opening the HTML file in a browser.

Note that the free version of Unity always builds an application that starts with the Unity-logo. This can only be avoided through buying the “Pro”-version of Unity.

3.14 MODULE 13: Extending Unity

Goal: Get an overview of the possibilities of how to extend the features of Unity

Difficulty: Intermediate

Prerequisites: None

Although Unity is a very comprehensive software tool, there might be useful tasks and features it cannot cover. These cases can often be handled by extending the software Unity. Please note that the "extensions" in this context do not (directly) influence the applications developed with Unity, but the software Unity itself. They can add new features to the software or make some tasks easier than before.

There are many different types of extensions. They following list tries to categorize them into several groups and describe them. Please note that you might read of different types of extensions at other resources, because there are many ways how to categorize them:

- **Script libraries**: A "script library" in this context is nothing more than one (or more) complex script files that provide functions that ease certain tasks and can be used from other scripts. Usually it can be put into the asset view just like any other script file. An example is the "iTween" library that allows to do many kinds of animations with low efforts. It can be found at http://itween.pixelplacement.com/.
- **Editor scripts**: These are special types of scripts that allow to extend the standard user interface of Unity and can handle certain tasks. There are many editor scripts available on the web, and it is not a superior challenge to create one on your own. They are script files that have to be placed into a folder called "Editor" in the project folder. In most of those scripts you will find statements that start with a "@"-character. These are instructions that are interpreted directly by Unity itself and can modify or extend its interface. An example for a useful editor script are the "TransformUtilities", which allow to easily align and copy the values of different "transform" components and can be found at http://www.unifycommunity.com/wiki/index.php?title=TransformUtilities.
- **Frameworks**: Frameworks are groups of assets that can help to achieve particular tasks when developing an application. Some of those frameworks can be downloaded as files in format *.unitypackage. It is a format that packages a group of assets in a single file. Such files can be imported into your project folder by clicking "Assets" →

“Import Package” → “Custom Package…”. It is also possible that they consist of asset files packaged in a *.zip archive that needs to be unpacked into the project folder.

The different assets of a framework belong together and provide new features through novel components or other elements. An example extension is the “Terrain Toolkit” which provides several tools to ease the creation of realistic terrain. It can be found at http://unity3d.com/support/resources/unity-extensions/terrain-toolkit.

- **Plug-ins**: Plug-ins are extensions that can add or modify functionality at a very deep level of the Unity software. They allow for example to extend the editor and use new components. With plug-ins it is possible to do incredible things with Unity. But know that development of plug-ins is not trivial.

  There are two different kinds of plug-ins: Plug-ins created via the “.NET” technology and native plug-ins which are written in other languages. The latter ones can only be used with a Unity “Pro” license.

  Plug-ins provide *.dll files, which are compiled programming libraries. There are different ways of how to install plug-ins, but in general they have to be put into your project folder. An example plug-in is “Path”, which adds pathfinding functionalities to your application and can be found at http://angryant.com/path.html.

4 Development of applications with Unity in a team

Knowing the technical aspects of how to use Unity is an absolute requirement when developing interactive 3D-applications. But beside these functional issues there are other skills as well that are necessary for creating high-quality applications.

Because of their complexities such applications are often developed within a team. But developing an application within a team brings new difficulties into the process. It is not enough to produce the necessary artifacts, the work that has to be done needs to be divided among the team members. Results need to be synchronized with each other, communication has to be done in a structured way to avoid misunderstandings, some kind of planned process needs to be performed to circumvent redundant work, idle work time needs to be avoided and it is necessary to keep the time schedule.

This chapter contains approaches for improving the teamwork when developing applications with Unity. The target audience are students who are beginning to learn Unity within an university context. It is often necessary for them to work on applications in small groups, but very rarely they are experienced teams who have been developing software for a long time. Most often it is the case that these students are not only new to Unity itself, they also do not know how to implement an efficient development process that reduces efforts and increases quality of results.

These are the assumptions for the approaches shown here. Because the students are no experienced developers, the processes presented here do not only need to be efficient, they need to be simple as well. It does not make sense for students to force them to spend more time on learning and implementing a complex process (which can hardly be done within a few months) than on learning the technical basics themselves. Therefore it is necessary to provide these students with simple guidelines that can improve efficiency without too much process overhead.

4.1 Structuring and synchronizing artifacts in a team

When developing 3D-applications the team members produce different kinds of (digital) artifacts. These artifacts can for example be 3D-models, script files or whole scenes. They are stored within the project folder of Unity.

Because all members of the team work on the assets of the project folder at the same time, there is lots of potential for producing inconsistencies between the members. Therefore it is very useful to agree upon a shared way of structuring the artifacts. This makes sure that all students have the same view onto the project folder, and helps to keep it organized.

Three general ways to structure the project folder are recommended in this thesis:

- **Structuring assets by type**: This means to group all assets of the same type into a shared folder. Asset types are for example “Script”, “Material” or “Prefab”. A folder is created for each of the types necessary for your project. The following figure shows an example structure:

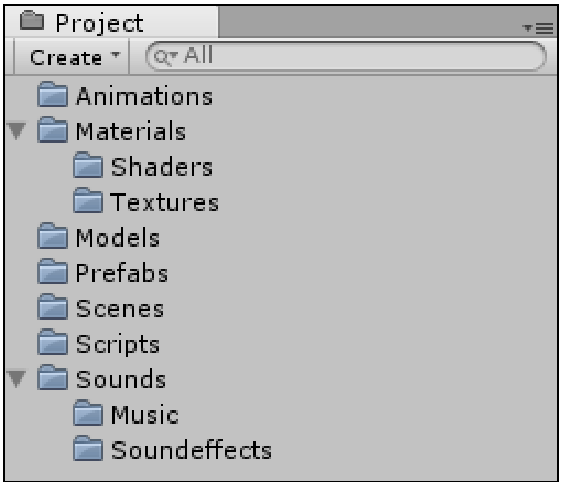


Figure 31: Structuring assets by type

This kind of structure is very easy and quick to implement. It can also be adapted to better serve the purposes of your particular project. It is very clear and unambiguous where to put newly created assets.

A drawback of this method is that it might mix different assets that do not relate to each other into the same folder. Furthermore the contents of each folder can become very huge for projects that are more ambitious. This might result in overloaded and confusing lists of assets. Therefore this method is probably best suited for small projects with a low number of assets.

- **Structuring assets by purpose**: Another way to structure the assets is to group them together by related purpose. For instance a complex enemy object can consist of several types of assets: A 3D-model, materials, several scripts that define the logic and finally a prefab to make it reusable. These assets share the same purpose: To model the “enemy” in the game. The following figure gives an example for a folder structure demonstrating this idea:

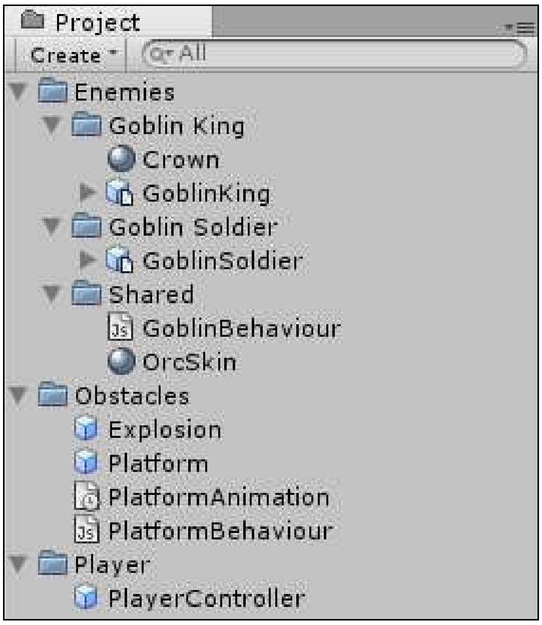
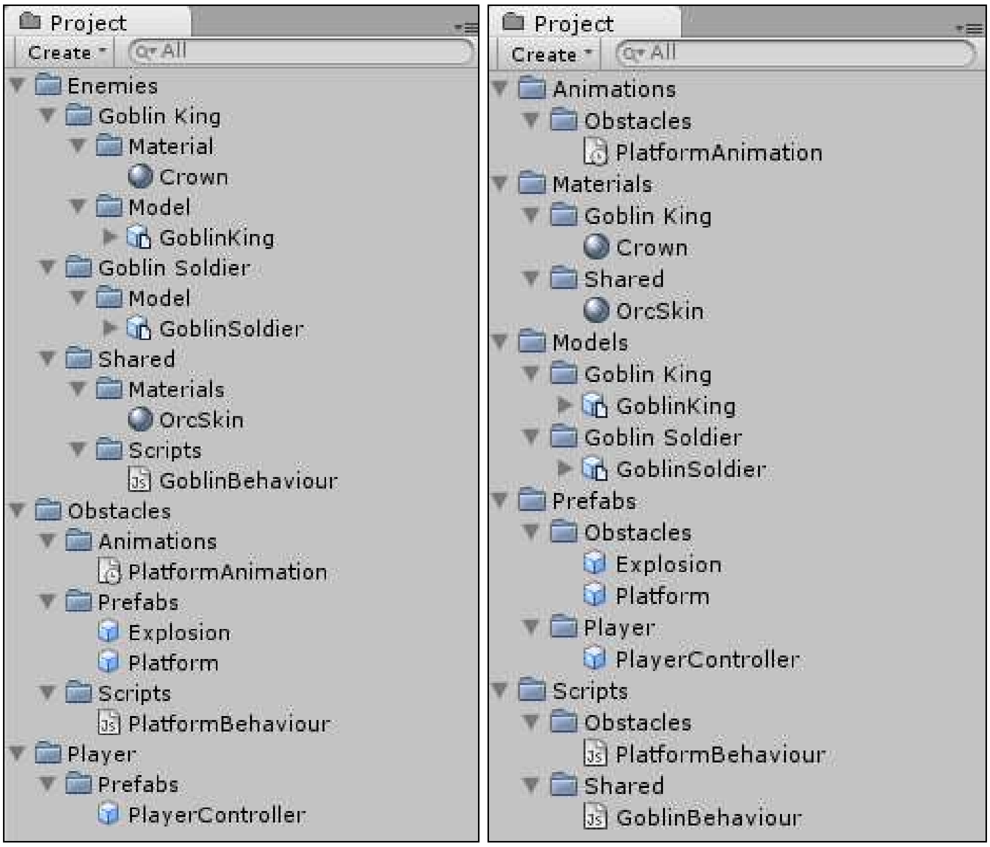


Figure 32: Structuring assets by purpose

This is a more complex structure. But it makes sure that assets that do not relate to each other are not getting mixed up. The lists of assets within the folders should not get too long, but it might be the case that there are lots of folders in the asset view. Another problem is the fact that many assets do not have a single purpose, they are needed for several different things, like for example different kinds of enemies. To solve this, “Shared” folders can be introduced that contain multi-purpose assets. This structure is well-suited for most projects, though it does not make sense to use it if there are too many files in “Shared” folders.

- **Hybrid structures**: Another possible solution is to group assets both by type and purpose. This means to have “purpose-folders” containing “type-folders” that finally contain the assets themselves. The other way round is also possible: “Type-folders” containing “purpose-folders”. The following figures give examples:



```
Enemies
    Goblin King
        Material
            Crown
        Model
            GoblinKing
    Goblin Soldier
        Model
            GoblinSoldier
    Shared
        Materials
            OrcSkin
        Scripts
            GoblinBehaviour
Obstacles
    Animations
        PlatformAnimation
    Prefabs
        Explosion
        Platform
    Scripts
        PlatformBehaviour
Player
    Prefabs
        PlayerController
```

```
Animations
    Obstacles
        PlatformAnimation
Materials
    Goblin King
        Crown
    Shared
        OrcSkin
Models
    Goblin King
        GoblinKing
    Goblin Soldier
        GoblinSoldier
Prefabs
    Obstacles
        Explosion
        Platform
    Player
        PlayerController
Scripts
    Obstacles
        PlatformBehaviour
    Shared
        GoblinBehaviour
```

Figure 33: Structuring assets by purpose and by type

The hybrid method can be very complex and result in lots of different folders. It should only be used in projects with a high number of different assets.

In general it cannot be said which method is best for development, because it depends a lot on the type of project and the number and kinds of assets used in it.

Also be aware of the fact that it is very easy with Unity to restructure the assets using the project folder view at any time. All references to assets remain intact when rearranging them. So if you do not know what structure to use, it is probably best to start with the simplest structure (group assets by type), and refactor it if it is not practicable anymore. But very important is the following: All team members should be using the same project structure!

This brings us to another issue when working in a team: **Synchronizing the artifacts**. Artifacts do not remain the same during a project, they are changed continuously. When different persons work together on the same files, possible conflicts can occur.

This is the case when the following scenario happens: Person A starts working on a file. In the meantime person B opens the same file, makes his changes and saves them. These changes could not have been adapted by A. So when A saves his changes, the modifications done by B are lost.

This can result in lots of additional efforts as well as into social conflicts between team members. Therefore it is of primary importance to avoid such conflicts. To achieve this, it is necessary to implement some kind of "lock" onto resources. A "lock" is a concept that allows a person to change resources by himself, but simultaneously block others from modifying the locked resources. Others can still "read" those resources and therefore use them, but are not able to change them. The holder of the lock then saves his changes and releases the lock. Others can now modify the resources, but it is assured that they start working on the files in their most recent state.

There are also approaches to "merge" a file in two different states if a conflict has occurred. But it can be very complicated and error-prone. In general it might be possible for files containing only text (like script files), but is hardly achievable for binary assets (like 3D-models).

Different methods exist to perform the synchronization of work results. Among them are:

- **Manual file synchronization**: This is probably the most common method of sharing work results among beginners. Team members just manually exchange the files they worked on with each other, often by using a medium like USB flash drives. There is no mechanism that resembles an automated lock. Students have to "lock" files by according the resources they work on with each other. But of course this is no guarantee to avoid conflicts. Misunderstandings, complicated relations between different files or uncooperative behavior can easily result in inconsistencies. Therefore this method can only be used for very small, simple projects with clear agreements between the different team members at project start.

- **Synchronization over a centralized file storage**: This means that each team member has access to the same location that stores the files in their most recent versions. Contrary to a process where each team member organizes the resources on their own machine, this makes sure that students share the same view onto the most recent state of the project.

There are different ways of implementing this method, from manually exchanging files with the store (like for example when using a shared FTP-server), to using services that automatically synchronize the files on the local machine with the centralized server. A locking mechanism might be available, but this is not always the case. If not, students have to arrange with each other.

This method brings some advantages through the consistent view onto the current state of the files, but is no guarantee to avoid conflicts. It furthermore might consume lots of time for the network traffic to send and receive large files from the server.

- **Transactional version control**: A sophisticated solution for development projects in general is to use a version control technology like for example “Subversion”. These technologies allow to send changes to a centralized server in a transactional way. Local copies can automatically be synchronized with the server. This ensures consistency on all machines. They usually also provide a locking mechanism that keeps other team members from modifying locked resources. Furthermore it is not only possible to view the most recent version of resources from the server, but also previous versions. This technique can be used for analyzing the evolution of assets or restore features that have been removed.

  Version control protocols are often smart enough to only send the differences of a file compared it its previous version to a server, so network traffic can be reduced to the inevitable minimum.

  Version control systems are very commonly used for collaborative development. But because of their features, they are not trivial to use. So they are probably not well-suited for small projects performed by people who are not familiar with version control.

The method to choose for synchronizing projects depends on the number and the skills of the team members, their experience in team work and the size of the project. Though it is not an advisable approach in general, a tiny project in a small team might come off best with manual synchronization if the team members agree on a process that avoids conflicts. Version control can prove its strengths in large projects, but creates an unnecessary overhead and technical problems due to inaccurate usage when used for a small assignment.

4.2 Development process

To produce a high-quality product, all efforts should be performed in a structured way. If all team members do just what they think to do right now, chaos is at hand and goals won't be met due to the lack of shared understanding of the project.
In this master thesis a development process that is well suited to teams of students who begin to learn Unity is recommended. It meets the concerns of the different phases of a project while still allowing to adapt to close deadlines. It was designed to be rather simple to make implementation for beginners very easy. The process was influenced by the incremental way of developing an application with usage of prototyping described in the work of Pearce and Ashmore. They analyzed the development of a prototype for a multiplayer game [Pearce and Ashmore, 2007].

The following figure illustrates the development process:

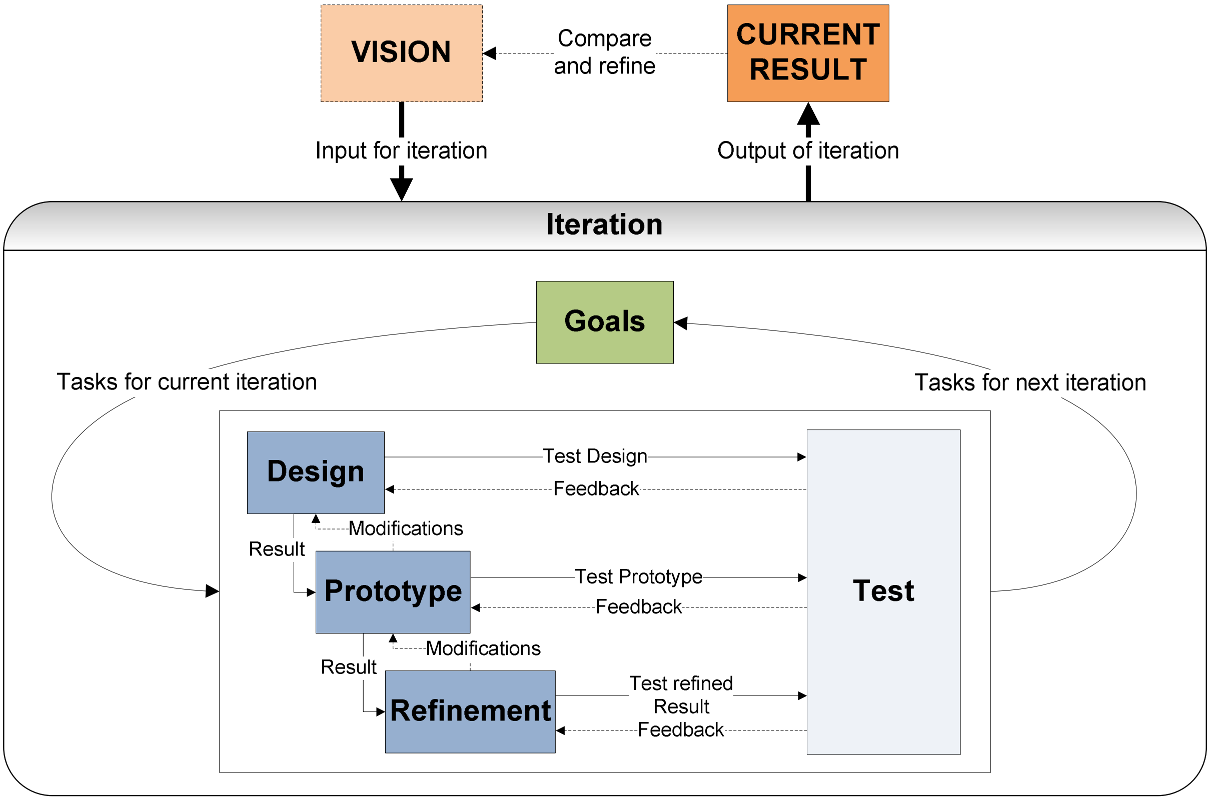


Figure 34: A development process for beginner teams

The process starts with a **vision** of the application that shall be developed. A vision is the idea for the software students must have at the beginning of the project. It is the “big picture” the students work towards to, but only consists of vague concepts that lack any details at this phase. In university context, it is also possible that the vision is already predefined for the students.

If this is not the case, a vision can be formed through communication and techniques like brainstorming. Although it is possible to refine the vision in later stages of the project, it can result in additional efforts. This is why students should spend sufficient time in developing a vision of a quality application that fulfills its purposes.

The vision is the input for the first (and following) iterations. The **work in iterative, incremental steps** is one of the most important aspects of this development process. Work is not being done in one big package, but in small incremental steps. Each step adds new features or improves existing ones of the application.

Very important is the fact, that **each iteration should produce a runnable, tested application as result**. This helps students to meet the schedule and do not overstrain themselves. It is often the case that beginners underestimate efforts when developing software. They try to implement a huge application and get overwhelmed with the requirements when deadlines come closer. A process that refines the application in small steps, each one producing a runnable result, can avoid this problem.

But of course it is important to choose the right size for an iteration. This can be done through the definition of **goals**. A goal is some objective the students plan to realize for the current iteration. Goals can be divided into several tasks to define them more precisely.

Example: A goal could be to “Implement enemy AI”, which is divided into multiple tasks like “Implement enemy pathfinding” and “Implement enemy attack patterns”.

It is furthermore important to assign all tasks to persons in a collective decision. If they are not assigned, it can be the case that persons only do the tasks they prefer and certain tasks remain untreated. There should be at least one responsible person for each task. Of course it is also possible to work together on tasks.

The actual work process can be separated into three different phases. In general the phases are handled successively, but it is also possible (and can make a lot of sense!) to interleave them. This process does not force any kind of order regarding the different work phases, but the demonstrated order tends to fit for many projects. But keep in mind that it should always be possible to refine the results from a previous phase, or have different persons working in different phases at the same time.

The results of all three phases should be tested to a varying extent. It might make little sense in early stages, but gets more and more important with progressing development stages. The different phases are:

- **Design**: The design should result in concepts that concretize the vision in more detail. While the vision itself is very vague, the design specifies the application as exact as possible at the current stage of the project. Results of design can include rules for a game, how to control the player, what the environment shall look like, background story,… etc. The design is very important, it does not only give the direction for all further work, it is also crucial for the quality of the application. It does not matter how well a game was implemented that is boring by nature.

  It is not easy but possible to test the design at an early stage. It could be done by using mock-ups that try to resemble the game or the application. Furthermore feedback from persons of the target audience might help to improve the design. But of course it is easier to test the design itself in later stages of the project, when the application can already be run. The design can and should always be refined if necessary.

- **Prototype**: During the “prototyping” phase actual development is done with Unity (or other tools used). However the goal is not to implement a finished, highly-polished product, but to realize a first prototype of the features for the current iteration. This means that not every detail is completed, only the most fundamental things. When implementing a new type of enemy for example, it does not need to be made with a highly-detailed 3D-model right away. It makes more sense to just create a simple capsule (or another shape that resembles it) and use it as provisional enemy. This allows to rapidly test the current concepts and design of the application. This way many design flaws can be avoided before any unnecessary efforts were spent.

  It furthermore helps students to estimate the efforts and further implications of their application: If it is already quite difficult to implement a racing game with simple rectangles, it shows how challenging a realistic racing game with physical behavior can be.

- **Refinement**: Refinement consists of all tasks that need to be done for a functional, presentable application (but only within the comprehension of the current iteration). This includes detailed 3D-models, working scripts and complete environments. A lot of testing should be done to ensure the quality of the application. The result should be an application that fulfills the goals defined at the beginning of the iteration in all necessary details.

After an iteration was finished, the team should be able to have an application that can be compared with the end vision of the software. The team must now decide what goals to realize for the next iteration, if there is one necessary at all. The result of the last iteration should be saved separately from the development branch. This allows to present something runnable even if the next iteration cannot be finished within schedule.

The set of goals for the next iteration should be chosen with the time schedule and the efforts the team members can invest in mind. It might also be the case that goals could not be achieved in the previous iteration, so they can be done in the next step. It makes sense to tend to choose a smaller set of goals for the iteration, because efforts are often underestimated.

Also be aware of the importance of **testing**! It can mean plenty of additional efforts, but increase quality tremendously.

5 Conclusion & Further Work

The development of interactive 3D-applications is a very complex task. It requires the investment of high efforts and expertise.

The software tool “Unity” is able to ease development of interactive 3D-applications. It combines an intuitive graphical user interface with concepts that are easy to understand and apply. It provides all necessary technologies required to implement realistic 3D-software. Therefore it is very well suited to teach development of 3D-applications to beginners.

In the context of this master thesis the requirements and problems students encounter when beginning to learn Unity were analyzed. For that reason two different university courses with students of architecture and computer science were investigated with qualitative research. This analysis showed the most important technical and didactical aspects to consider when teaching Unity to beginners.

These findings were considered for the creation of resources that help students to learn Unity. The resources contain technical instructions that cover the most important aspects when developing 3D-applications with Unity, as well as guidelines for development of applications in a team within the university context. The guidelines include the recommendation for a development process that fits well for the students.

Because the resources shall be easily accessible to the students, an evaluation was conducted on how to store them in a centralized repository. The result of this evaluation was to save the resources in a “WordPress”-blog, which ensures flexibility and low efforts for authors as well as lots of possibilities to find resources for students.

The results of this master thesis could be extended with **further work** regarding different aspects: The technical tutorials themselves cover the most important details for beginners, but skip some information that is important for students who want to improve their skills any further. Moreover the detailed usage of important plug-ins and frameworks should be described. The tutorials could also be enriched by producing demonstrative videos that show the concepts described in action. This could be a valuable resource for beginners.

Figures

References

[Goldstone, 2009]

Will Goldstone. 2009. Unity Game Development Essentials. Packt Publishing.

[Wikipedia, “Bewegte Bilder”]

Seite „Bewegte Bilder“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 5. Januar 2011, 10:16 UTC.
URL: http://de.wikipedia.org/w/index.php?title=Bewegte_Bilder&oldid=83473671
(Accessed: 19th May 2011, 16:31 UTC)

[Elliot et. al., 2002]

J. Elliot, L. Adams and A. Bruckman, No magic bullet: 3D video games in Education, in Proceedings of 5th ICLS 2002.

[Lu et. al., 2005]

Lu, T., Tai, C., Bao, L., Su, F. and Cai, S., 3D Reconstruction of Detailed Buildings from Architectual Drawings, Computer-Aided Design and Applications, Vol. 2, Nos. 1-4, 2005, pp 527-536.

[PC Games, 2010]

http://www.pcgames.de/Retrospektive-Thema-214694/Specials/Meilensteine-der-Spielgeschichte-Eine-Retrospektive-des-digitalen-Hobbys-Teil-3-744144/
(Accessed: 22nd May 2011)

[golem.de, 2004]

http://www.golem.de/0411/34765.html
(Accessed: 23rd May 2011)

[Wikipedia, “Blender”]

Seite „Blender (Software)“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 17. Mai 2011, 22:21 UTC. URL:

http://de.wikipedia.org/w/index.php?title=Blender_(Software)&oldid=88965234
(Accessed: 29th May 2011, 11:39 UTC)

[Unity]
http://unity3d.com/
(Accessed: 30th May 2011)

[Unity, “What’s New in Unity 2.5”]
http://unity3d.com/unity/whats-new/unity-2.5.html
(Accessed: 30th May 2011)

[Waldenfels]
http://www.waldenfels.at/
(Accessed: 05th June 2011)

[Klett, 2001]
Fanny Klett. 2001. A Design Framework for Interaction in 3D Real-Time Learning Environments. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies* (ICALT '01). IEEE Computer Society, Washington, DC, USA, 63-.

[Trac]
http://trac.edgewall.org/
(Accessed: 04th July 2011)

[MediaWiki]
http://www.mediawiki.org/
(Accessed: 06th July 2011)

[WordPress]
http://wordpress.org/
(Accessed: 06th July 2011)

[Kohive]
http://kohive.com/
(Accessed: 06th July 2011)

[ResourceSpace]
http://www.resourcespace.org/
(Accessed: 06th July 2011)

[Razuna]
http://www.razuna.org/
(Accessed: 06th July 2011)

[Hearn, Baker, 2003]
D. Hearn and M. P. Baker, Computer Graphics with OpenGL, 3rd ed. Prentice Hall, 2003.

[Pearce and Ashmore, 2007]
Celia Pearce and Calvin Ashmore. 2007. Principles of emergent design in online games: *Mermaids* phase 1 prototype. In *Proceedings of the 2007 ACM SIGGRAPH symposium on Video games* (Sandbox '07). ACM, New York, NY, USA, 65-71.