Εξώφυλλο Αναφοράς Πτυχιακής Εργασίας

Τεχνολογικό Εκπαιδευτικό Ίδρυμα Κρήτης

Σχολή Τεχνολογικών Εφαρμογών

Τμήμα Μηχανικών Πληροφορικής

Πτυχιακή Εργασία

Cognition in Digital Environments

Καβουσανός Γεώργιος (ΑΜ : 2744)

Επιβλέπων Καθηγητής : **Παπαδουράκης Γεώργιος**

ΗΡΑΚΛΕΙΟ

2016

Abstract of the final year project in English

This document contains the results of an attempt to explore the differences in the activity of the human brain that is exposed to a digital world through Virtual Reality(VR), as opposed to more traditional media (Computer Screen). It was a three-phased project.
Phase one consisted of creating a professional, detailed digital world for use with Virtual Reality, using Unity (a Game Engine), Blender (a 3D modeling tool), Adobe Photoshop, CrazyBump (a Normal Map creation tool) and several other programs, thoroughly mentioned in Part One of the paper.
Phase two consisted of understanding the way Emotiv's Epoc Electroencephalographer(EEG) functions, either using or implementing drivers to obtain relevant to the research data and finding volunteers to form a big enough group for the purposes of the experiment.
The final phase consisted of the actual experiment itself, the post-processing of the data, the calculation of the Summary Statistics (mean, median, mode), the creation of the graphs, the explanation of the results and the documentation of the experiment.

Περίληψη πτυχιακής στα Ελληνικά

Το παρόν έγγραφο περιέχει τα αποτελέσματα της απόπειρας να εξερευνηθούν οι διαφορές στην εγκεφαλική δραστηριότητα ενός ατόμου εκτεθειμένου σε ψηφιακό κόσμο μέσω Τεχνητής Πραγματικότητας, αντί μέσω πιο συμβατικών μεθόδων, όπως η οθόνη ενός υπολογιστή.

Το εγχείρημα είχε τρεις φάσεις.
Η πρώτη φάση περιλάμβανε τη δημιουργία ενός λεπτομερούς ψηφιακού κόσμου με χρήση διάφορων εργαλείων, όπως Unity, Blender και Adobe Photoshop.
Η δεύτερη φάση περιλάμβανε την κατανόηση του τρόπου λειτουργίας του EEG Epoc της Emotiv, η κατασκευή ή αξιοποίηση drivers για τη λήψη μέσω αυτού των απαραίτητων πληροφοριών και η εύρεση ενδιαφερομένων για τις δοκιμές.
Η τελευταία φάση περιλάμβανε το ίδιο το πείραμα, την επεξεργασία των αποτελεσμάτων, τον υπολογισμό διάφορων στατιστικών δεικτών, τη γραφική αποτύπωση αυτών και την επεξήγηση και καταγραφή των αποτελεσμάτων.

Table of Contents

Table of Pictures

List of Tables

0. Introduction

Virtual Reality(VR), also known as **immersive multimedia** or **computer-simulated reality** is an emerging technology that replicates an environment, real or imagined, and simulates a user's physical presence and environment to allow for user interaction. It, artificially, creates a sensory experience, which includes sight and hearing, and in more advanced applications touch and smell[1].
As the very name of the technology implies, audiovisual content presented through VR means tend to be a lot more immersive and vivid to the user than through more traditional devices.
And while the above are a truth easily understood and felt by any modern VR user, no modern attempts have been made to calculate and document exactly how different an experience it is.

0.1 The Objective

The work below is an attempt to prove beyond doubt that the differences between experiencing a virtual environment through the screen of a computer and a VR Headset do exist and to give an estimate as to how big these differences are.
It is not, however, the large scale research using incredibly powerful EEG equipment and thousands of testers that would be needed to obtain precise numbers and make safe, irrefutable assumptions.
It should be viewed as a first step, as proof that such a large-scale research would, in fact, yield interesting results.

0.2 Summary

The basic idea behind the experiment is simple. The user experiences a digital environment, in first-person view, first through a monitor and then through a VR Headset.
At the same time, the brain activity is being monitored by a portable EEG. Brain activity is also being monitored while the user is in a calm, neutral state.
Finally, the user fills a short survey providing information, such as age and experience with virtual environments, to help refine the data even further.
By comparing the above, one could, in theory, determine the differences between the two activities for each tester.

0.3 Motivation

Virtual reality is an emerging technology and very little research has been done on this very interesting field.
What's more, very few people can be currently considered experts on it. Experience on VR is and will be on high demand for the years to come. And, finally, this project resonates with my personal interests and skills.

0.4 Document Structure

The structure of this document directly reflects the workflow of the project.
The first chapter, the creation of the digital world that was used in conjunction with VR, is thoroughly explained in 1. The Digital Environment**.**
This is the biggest chapter, reflecting the fact that making this environment required the most time and effort.
The second chapter, 2. The EEG, describes and analyzes the hardware and software used to collect the brainwave data.
The third chapter, 3. Results, contains the documented data obtained from the EEG, the resulting inferences drawn from them and all the metrics and algorithms used to do so.

1. The Digital Environment

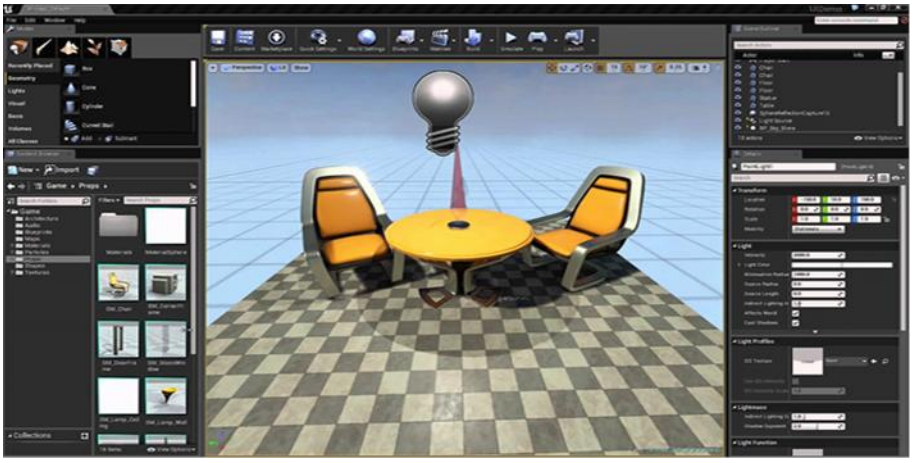
A digital environment is a simulated place made through the use of computers. For this project, two digital environments were used.
One of them was created specifically for the purposes of this experiment, to be used with a VR Headset.
The second was part of a bigger environment, made by a third-party developer, that resembled (to a degree) the custom, VR environment and filled certain criteria.

1.1 Creating a VR-compatible Environment

1.1.1 The Game Engine

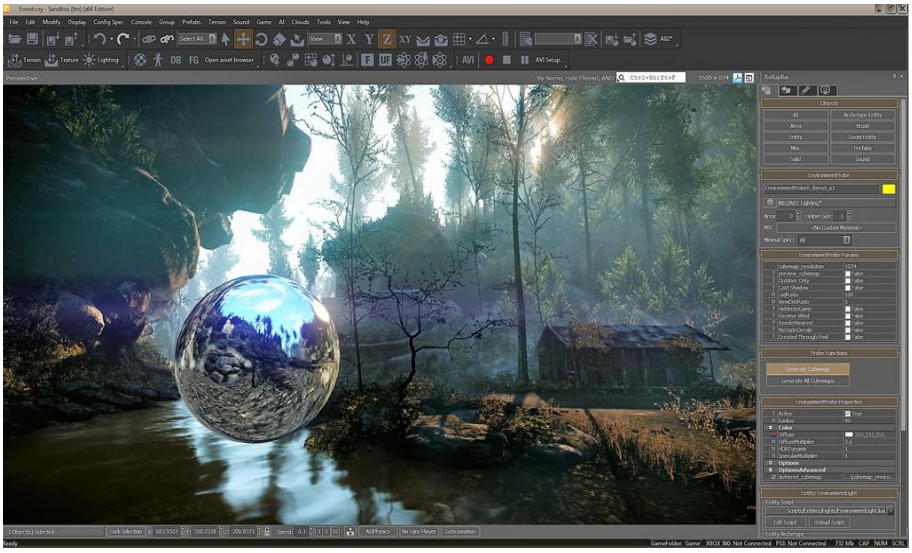
The very first, and most important, choice a developer has to make when creating a digital environment is the Game Engine he or she is going to use.
It's such an important choice because it's irreversible. A Game Engine switch halfway across the project is one of the most destructive scenarios, as it practically means starting over.The available game engines at the beginning of this project were three, Unreal Engine 4, CryEngine 3 and Unity.

Unreal Engine : A very powerful engine, created by Epic Games. Its fourth edition became available to all users for free on March 2015. It has a large community, offering a great amount of support to users old and new in the form of tutorials and ready assets, many of them free. Primary scripting language is C++.



1.1 Unreal Engine Environment

CryEngine : Another powerful engine, created by Crytek. At the start of this project, CryEngine was distributed under a monthly subscription fee, which posed a huge obstacle to developers not planning on making profit.



1.2 CryEngine 3 Environment

Unity : A relatively new engine, developed by Unity Technologies. It's being distributed for free and offers incredible support to new developers, mostly in the form of free assets and tutorials. It's arguably the least powerful of the three engines but the easiest to learn. It offers easy VR integration and a large choice of scripting languages, including C#.
Due to the above, and for many other minor reasons, Unity was, finally, chosen.

1.3 Unity Environment

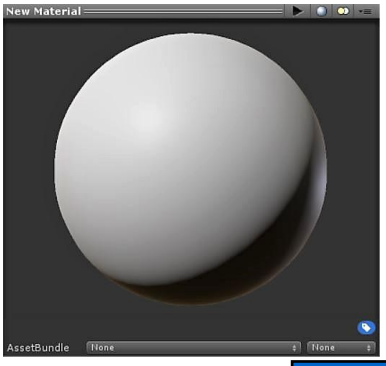
1.1.2 3D Model Creation

3D models are an essential part of every digital environment. Creating entirely new models is an art, and a very time-consuming process. For this one-man project, finding, modifying and re-using free models was a necessity.
Unity offers a very large selection of **free assets**, but even so, sometimes creating your own models or modifying existing ones to fit your needs is necessary.
To cover those needs, a **free modeling software**, Blender, was used. Made by the Blender Foundation and supported by Unity, it was the obvious choice for non-profit, academic work.

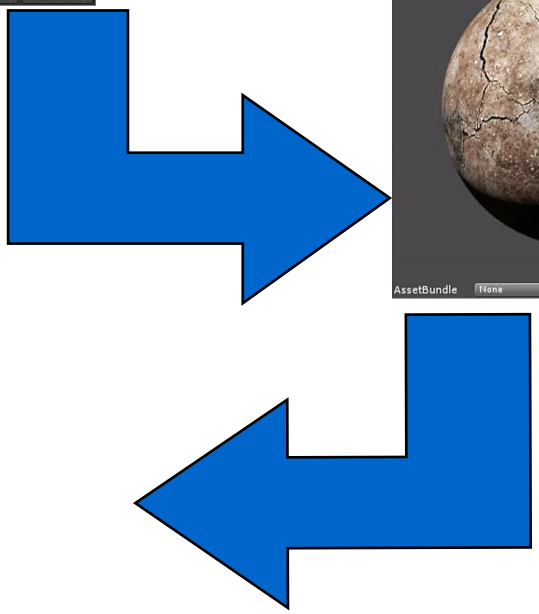
1.4 Blender Environment

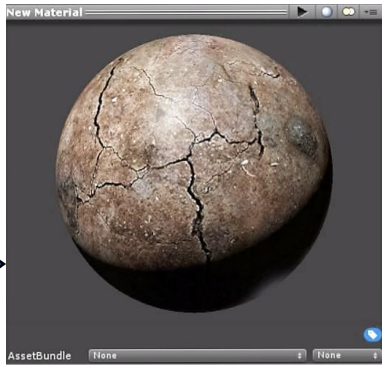
1.1.3 Normal Maps, Textures, Shaders and Materials

Materials are a huge chapter in the creation of any digital environment. They cover every model in the game, providing information on how the surface of the model should appear. To do that, they combine **textures,** which are simple bitmap images, with **shaders,** scripts that implement specific calculations and algorithms for calculating the color of the textures depending on the lighting of the environment (i.e. making water semi-transparent or golden bars shiny).
Finally, a **normal map is** a special kind of texture that adds surface detail to models.
Pictures 1.5 to 1.7 is a graphical representation of the process of creating a material for use in a game engine.



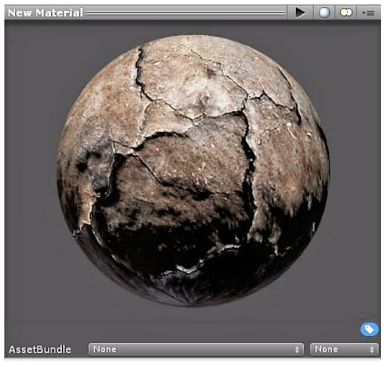
1.5 Empty, basic material





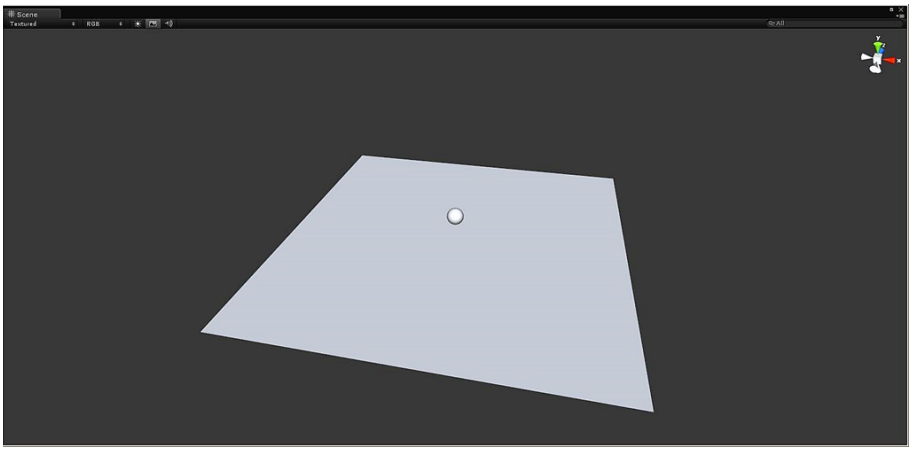
1.6 Material with texture

1.2 Terrain: Sculpting

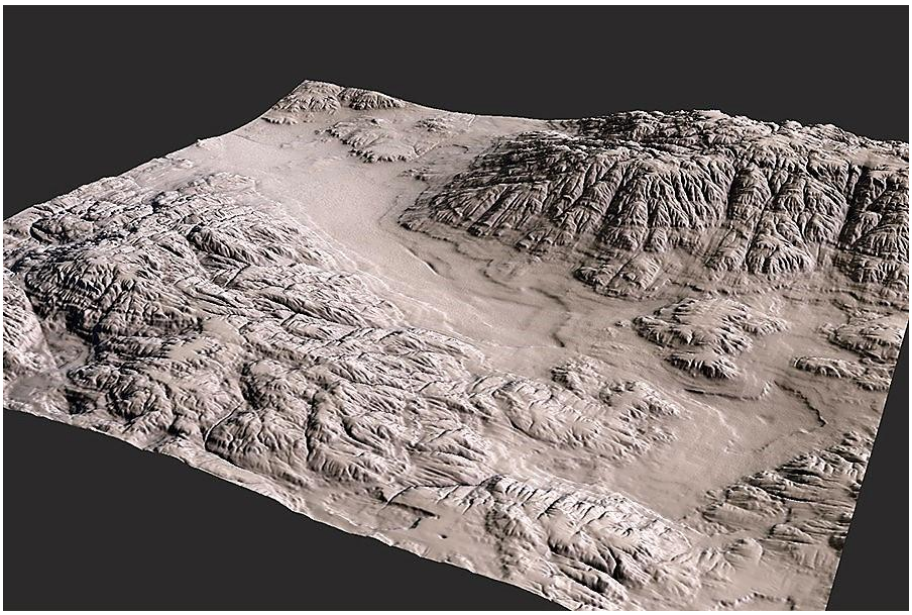


1.7 Material with texture and normal map

A **terrain** in Unity is a **large, modifiable plane** that allows for the creation of landscapes. Terrain sculpting is the process of using specific tools to turn a flat plane into a landscape, and depending on the desired quality and size, it can be a very difficult and time-consuming activity. Unity provides many types and shapes of tools for adjusting the height, smoothness and shape of the terrain.
Pictures 1.8 and 1.9 is an example project, visually explaining the concept of terrain sculpting.



1.8 Basic, empty Unity terrain



1.9 Sculpted Unity terrain

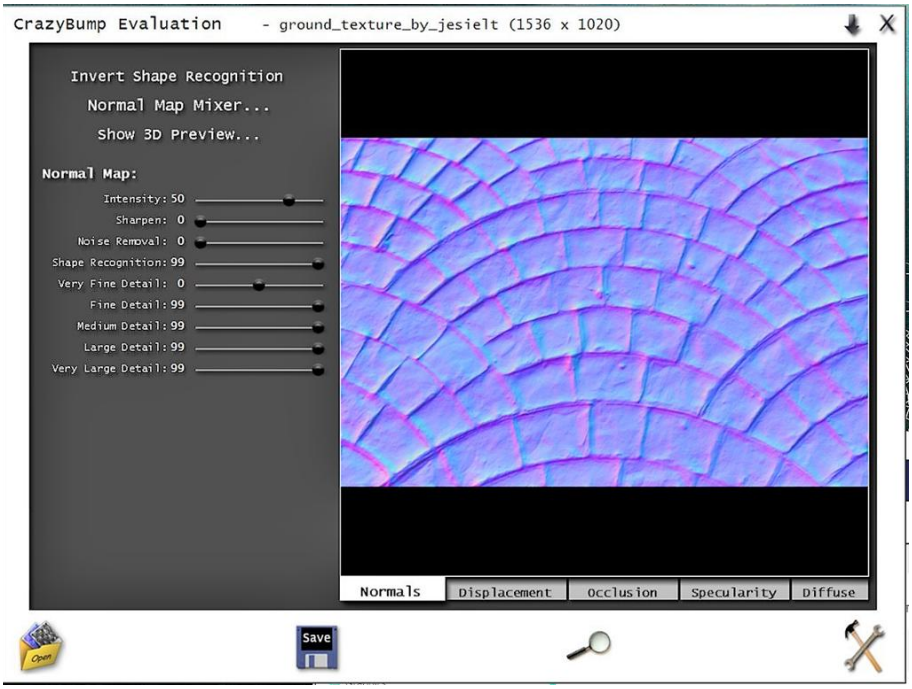
1.2.1 Terrain: Texture Painting

After the terrain has been sculpted to sufficient detail (modifications can be performed at a later stage, but it tends to be harder and more time-consuming), the next step is applying **materials**. Much like any 3D object, materials for Unity terrains are created by **combining textures and normal maps**. Shaders, though, for reasons of optimization (terrains are big, complex objects), cannot be freely chosen.
Unity provides **specific shaders** for use with its terrains.
To create a new, custom material, we must first provide the desired texture, usually in the form of **PNG** or **JPG**.
For example, wanting to paint a brick road on our terrain, we pick the following image (**Picture 1.10**).

1.10 Brick road texture

This image (also called **texture**), can, in theory, be used immediately to paint our terrain. The result might be of low quality and detail, but it is one way to make the environment lighter on hardware requirements. Applications on **mobile** might benefit from such a decision.
For a detailed, professional, terrain though, a normal map must also be created to add realism and visual fidelity.
The theory behind normal mapping is that a layer is created over a flat texture that provides information on how digital light should behave after hitting it. The map reflects light as if the texture was a detailed, 3D object and tricks the eye into believing a 2D surface is actually 3D without adding any resource-costly, polygon details.
In this project, Crazybump was used to generate such layer.

The texture is imported into Crazybump, some parameters are set (for example, detail intensity) and the normal map is automatically generated.
The resulting normal map can be observed in **Picture 1.11**.



1.11 Normal map

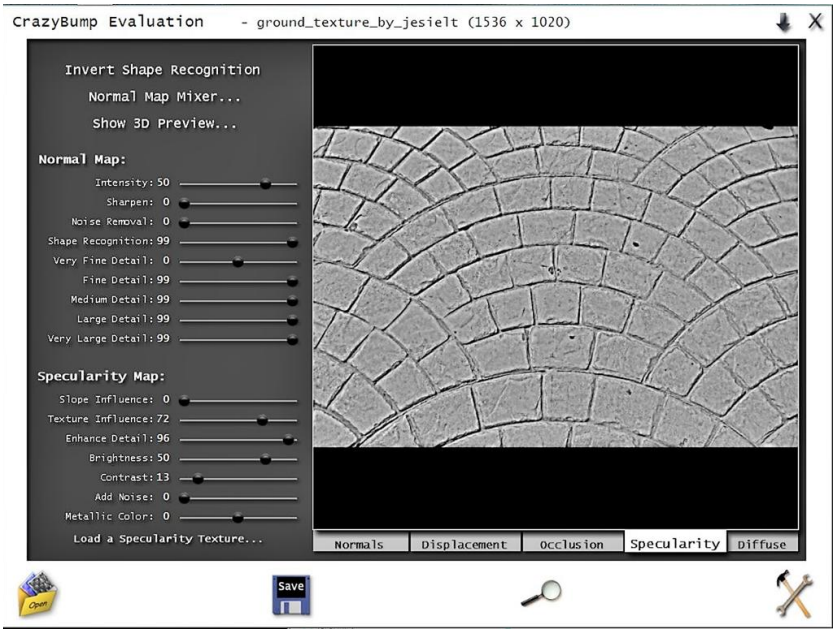
And in **Picture 1.12** the texture is displayed again, imported in the software.

1.12 Texture

1.13 Normal map over texture

Finally, when combining the above, we can see the material produced (**Picture 1.13**).

Before we apply this material, however, there is still one more layer we can generate and apply. Every real **substance** and material has some degree of **glossiness**. For example, wet marble is very glossy, while dry dirt is not.
Utilizing **Crazybumps** capabilities, we can create another layer, called **Specular**, for the purposes of simulating that effect.
In Picture **1.14** is of the **specular map** generated.



1.14 Specular map

So, by applying everything created so far, we achieve a realistic looking material (**Picture 1.15**) for use on our terrain that is relatively **computationally cheap**.

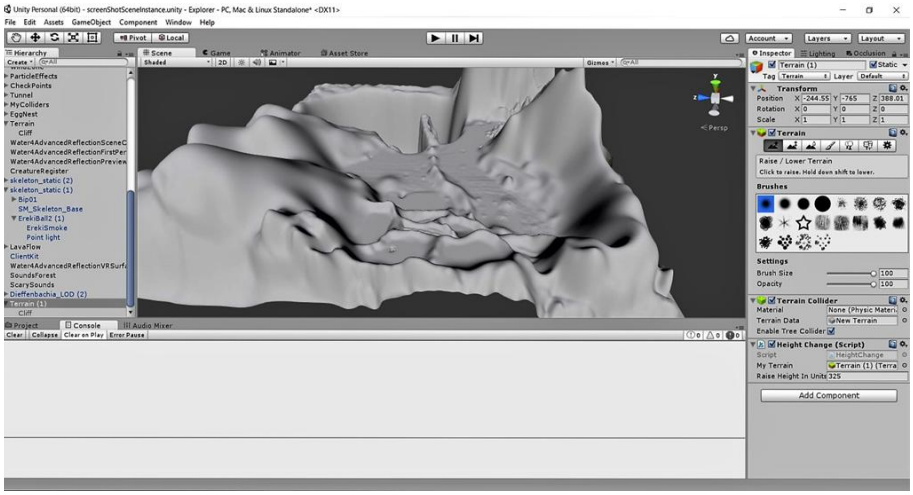
1.15 Normal and specular maps on texture

To sum everything up, **Picture 1.16** is of a comparison between a **simple** and a **normal/specular mapped** texture.

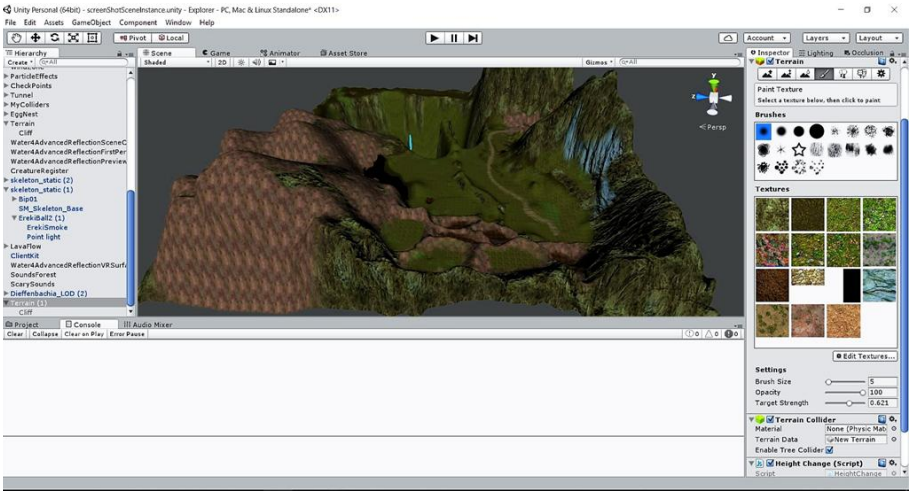
1.16 Comparison between textures

By using any or all of the above techniques, suitable materials for the terrain can be created. It also needs to be mentioned that ready-for-use materials can also be downloaded from the Unity Asset Store.
In the store, an incredible amount of **free** and **purchasable** materials, shaders and textures can be downloaded. Since time is valuable, it's always a good idea to visit the store before trying to make anything new in Unity.
Chances are it already exists and constantly "reinventing the wheel" is extremely counter-productive, both for solo developers and huge teams.

After the required materials have been made and stored, they need to be applied on the terrain. Unity provides a special **toolkit** for painting terrains, much like they do for sculpting them.
By using **brush-tools** and adjusting **parameters** such as **texture strength** and **tiling size**, the developer slowly covers the terrain and produces a **realistic**, yet void, landscape.
The following images (**Pictures 1.17 and 1.18**) depict the actual terrain created for this experiment, from sculpting to painting.



1.17 Sculpted terrain, no textures



1.18 Sculpted, painted terrain

1.2.2 Terrain: Nature and Vegetation

The next step towards a **realistic**, **mountainous landscape** is adding **vegetation** and other **natural details**.
Vegetation is, usually, consisted of two categories. **3D and 2D**.

The first includes trees, large plants, bushes and **anything** we want to be **detailed** and **standalone**.

The second is consisted of 2D assets, usually grouped together to produce a 3D feeling to the onlooker. **Grass** and **small plants** are usually put in this category due to **performance issues**. Trees and large plants generally take up space. Three or four trees can effectively cover a small area, if used correctly.

Grass, on the other hand, has to exist in **bulk** in order to be **visually appealing and realistic**. And since rendering a thousand patches of grass one by one, in 3D, would be **computationally devastating**, a technique was developed to render them grouped together, in 2D.
For this to work, every 2D-grass-image is forced to constantly face the camera (**Billboard**).
While grass is usually rendered and grouped together using the technique above, anything can be made to do so in order to save computational resources. Two fitting examples are weeds and small flowers.

For the first category, individual 3D models of the desired vegetation are needed. These can be made in any 3D graphics software, like Blender, in specialized software, like Speedtree and/or downloaded in the asset store for free or for a fee. Also, Unity **has its own built-in system** for tree-creation. Most of the flora used in this environment was downloaded from the Unity Asset Store. Then, it was either used as-is or got modified in Blender or in Unities native tree-maker.

By **re-using** pieces of models, like leaves and trunks and **combining** them together, it's easy to create an abundance of different, unique models to decorate even huge landscapes with. For this project, **at least 20 different models** were used to sufficiently fill the terrain.

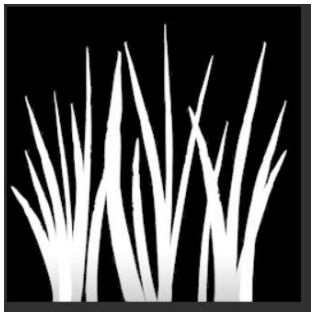
When all the needed models are found and imported, the developer faces yet another choice. He/she can **simply place** them, much like any other object, on the terrain one by one or have Unity treat them like **Tree Assets** and use **special tools** for placing them in the scene.

When they are simply placed in the scene, the developer has more control over them but Unity **does not recognize them as trees**. So, the developer must make **his/her own effects and physics**, but can also select and **affect each tree directly**.

On the other hand, if the developer decides to utilize Unities tree-placing system, they enjoy **perks**, such as **automatically generated wind movement, physics (**collision detection, for example**), randomized trees**(if a tree model fills certain prerequisites, each tree placed will differ from the rest), **better large-scale tree management**(can affect traits of all the trees at once), **mass-random tree placement and optimization**.

The same is true for grass and the rest of the 2D assets mass placed in Unity. While it is possible to place them one-by-one, it's far more efficient to **mass place** them using Unities tools. These assets, due to their nature, are more like **textures** than **models**. They can be found in the Asset Store or made in any imaging/photo editing software, with **Adobe Photoshop** being a good example. Unlike simple textures, these assets usually are a pattern in a picture, with the **alpha** being zero in all the **empty areas**.
Pictures 1.19 and 1.20 are an example of a single grass leaf patch and its alpha cutoff.

1.19 Grass texture



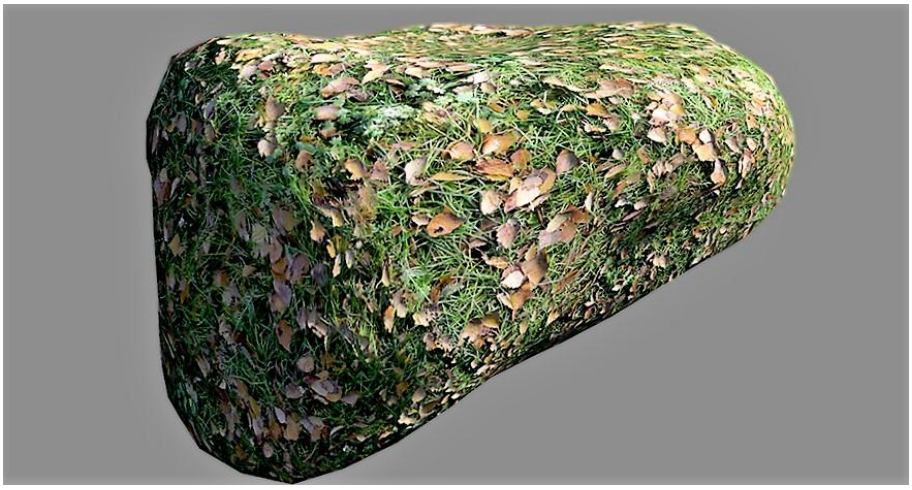
1.20 Grass texture alpha

So each piece of grass is, basically, a **2D square image** with **invisible edges** that **always faces the observer.** When one of these is rendered alone, its poor quality and strange movement are easy to notice. When, on the other hand, many of these are placed together, covering big areas, the loss in quality is hardly noticeable, as demonstrated below (**Picture 1.21 and 1.22**).

1.21 Thin grass

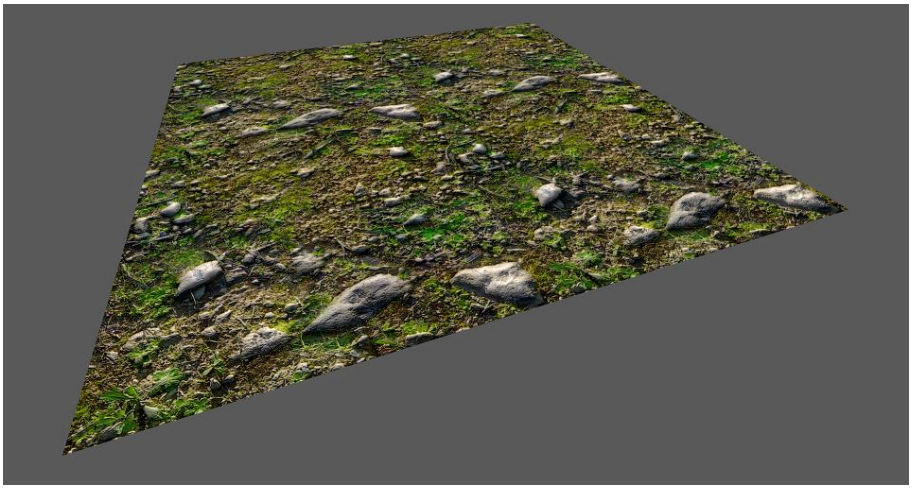
1.22 Thick grass example

Another method used for making grass **seem realistic** is painting the terrain below the grass with a **suitable**, **green texture**. It makes the grass seem **thicker** and **volumetric**. By using a proper texture, grass density can be reduced, resulting in huge **computational savings** and, thus, better **framerate**.



1.23 Under-grass texture with leaves

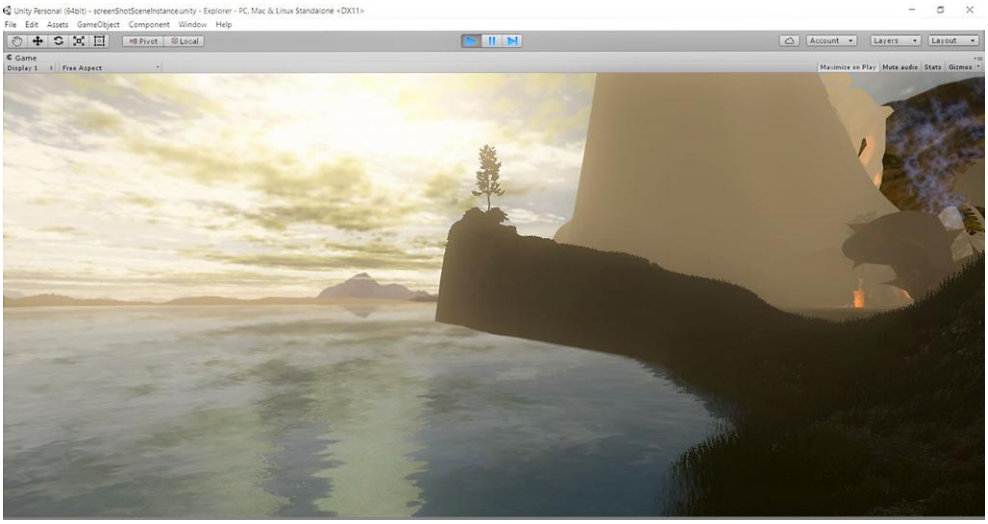
One of the many advantages of using Unities built-in tools for managing the vegetation is that the density of grass can be **adjusted** at any time using a **slider**, instead of adding-removing patches of grass.



1.24 Under-grass texture with rocks

This also means that the **end-user** can personalize the scene to a degree by choosing the desired amount of detail(grass, flowers) density his machine can adequately support.
Images 1.23 and 1.24 are two texture examples used in conjunction with grass to amplify the visual fidelity in this very project.

Finally, an element used in most digital worlds is, of course, **water**.
Unlike other elements and possible **visual effects**, water is a frequent sight in reality and one of the best ways of **naturally ending the landscape**.
Creating realistic digital water, though, is a very **complicated** activity. Water **reflects** and **refracts** light, forms **ripples**, **waves**, **foam** where it comes into contact with objects and game physics behave in a completely **different** way than on land.
Fortunately, Unity comes equipped with two free, decent **water-making shaders**. They still need a great amount of **configuration**, but it is nothing compared to the time one would need to make completely **custom** water.
In this project, the **Water4Advanced asset** was used, configured to imitate slightly fluorescent, calm lake water as shown in **Picture 1.25** below.



1.25 Water4Advanced

1.2.3 Terrain : Ornaments

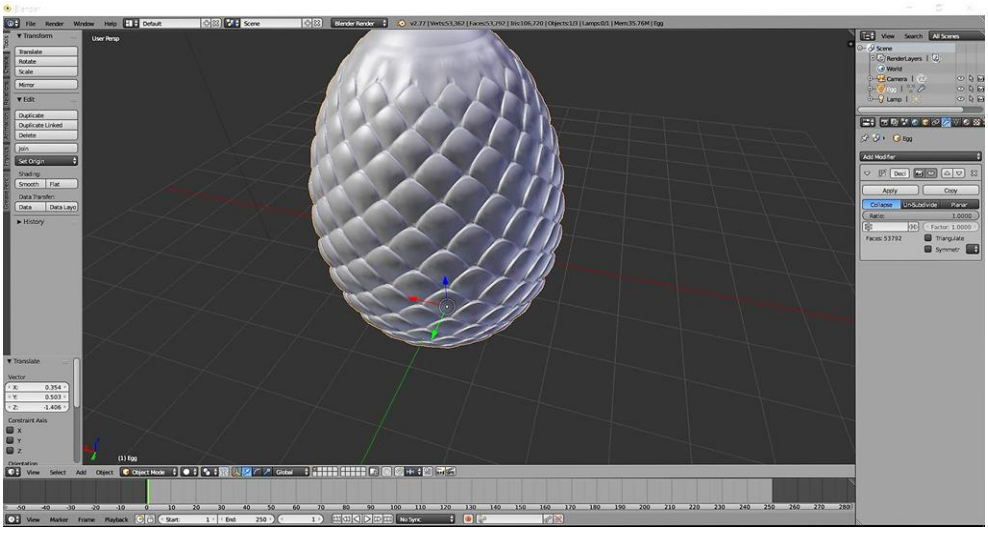
Ornaments, in a digital world, could include **rocks** of various shapes and sizes, **special effects**(fire, smoke, lava, clouds), **interact-able items** and any other visual part of the landscape that is not vegetation, water or some sort of creature.
These 3D models add a lot to the realism of the digital world and make the **virtual experience** more interesting to the user.
The Asset Store has an abundance of models, from cars and skyscrapers to skeletons and statues. Unity, though, has no built-in, optimized way of handling these models, so it's up to the user to integrate them in the scene(instance of all or part of the digital world) and make sure they are up to the project standards.
An **example** of **bad integration** is using a very high **polygon-count** model of a car in a racing game meant to be played on a mobile device. The model should be used, but its polygon-count should first be reduced using a relevant software (Blender, for example).

One of the biggest advantages of developing this project, where a general landscape is needed for use with VR, is that there are **no limitations imposed** in decoration. Any relevant 3D model, distributed for free on the Internet, could be used. And the fact that it was developed for **academic reasons** and not for profit-making, ensured that almost every asset could be **legally** used.

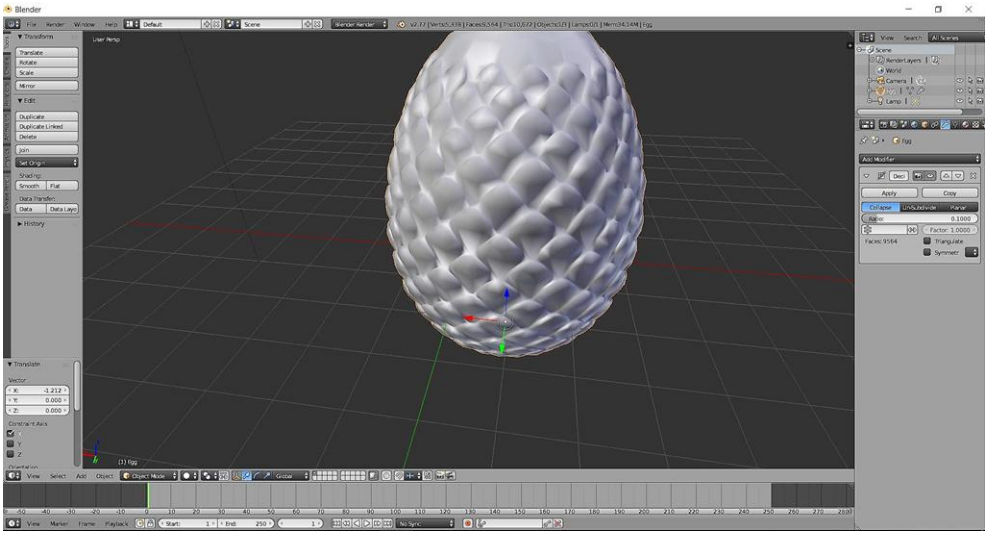
Pictures 1.26 and 1.27 describe the **typical procedure** of **downloading**, **processing** and **using** a 3D model.

It's decided that this model will have a relatively **small scale** in the scene, thus it does not have to be detailed. But it's made by an **incredibly high** amount of **polygons** (more than 50.000). It needs to be **simplified** before being used.
Using the **decimate** function of Blender, we reduce the polygon-count as much as possible while trying to keep quality high. This way we manage to **reduce polygons**, and thus computational requirements for using this model, by **90%**.

Picture 1.27 shows the **final**, **optimized** result. The differences are almost unidentifiable to the eye.

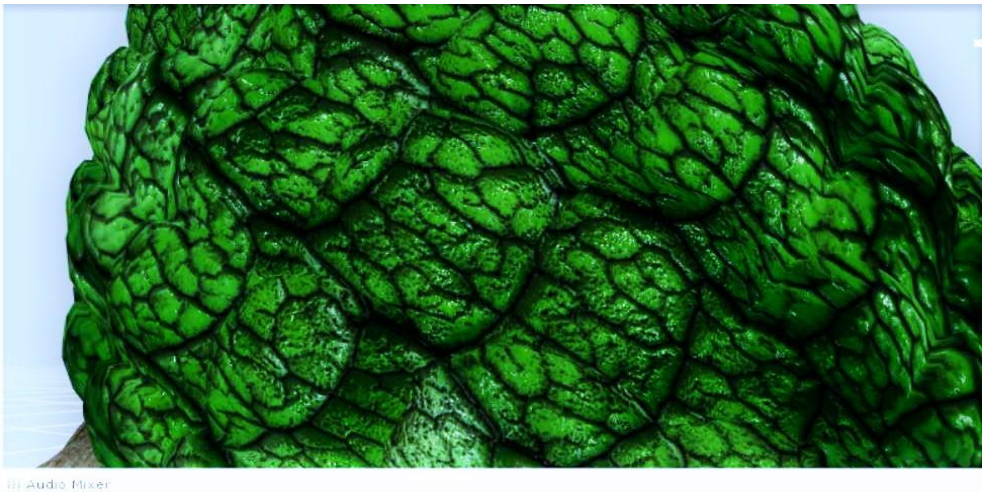


1.26 The unoptimized model



1.27 The optimized model

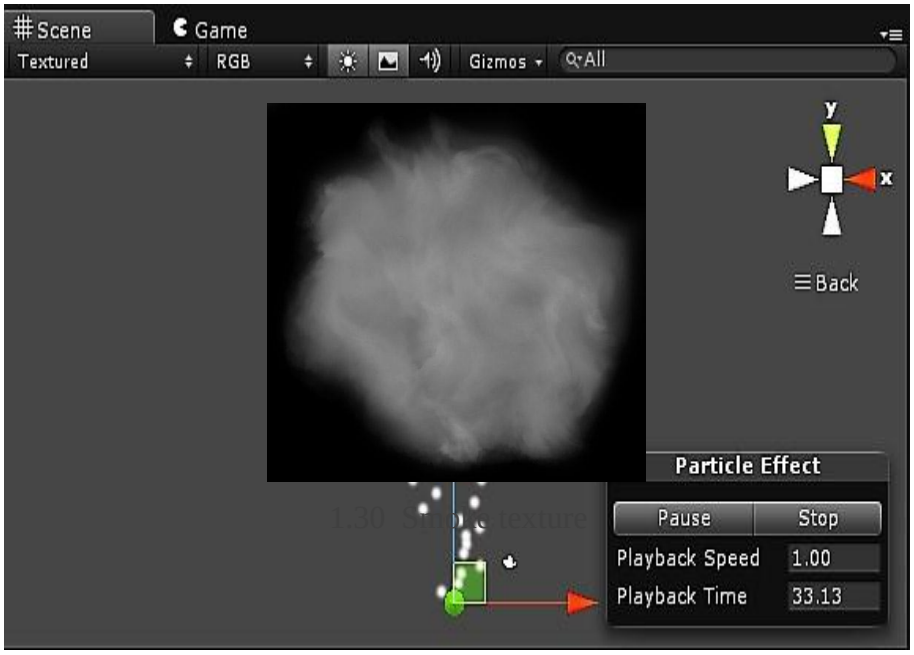
After the **optimization process** is finished, we import the model in Unity where we **create and apply a material**, much like we did for the terrain.



1.28 Textured, optimized model

The other noteworthy type of ornament widely used in digital worlds are special effects, formally named **Particle Effects** in game design.
Unity has its own, build-in system for creating and handling these effects. Its basic function is simple. There is a **source**, the **core** of the effect, that generates either **2D** or **simple 3D** models. The **size, generation rate, speed and texture** of those models, along with many other attributes, can be freely customized. What's more, every one of those models can, at any point, be made into a source of its own. Proper use of that system makes effects like **fire** easy to create and handle.
Pictures 1.29 to 1.33 show the process of making such an effect.

First, we create an empty, basic particle effect within Unity (**Picture 1.29**).

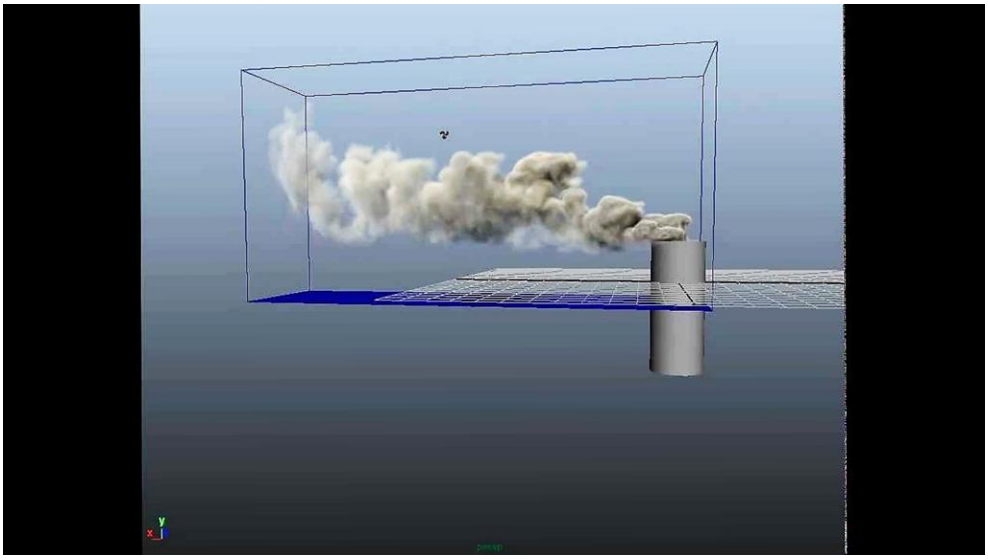


1.29 Basic, empty particle system

The white, round objects are called **particles** and, at the bottom, the invisible area that emits them is called the source. In **Picture 1.29**, the source is near the green dot. All those visible particles started there, before ascending to their current level. The basic, automatically generated **particle system** emits white(untextured) particles that are have an **initial speed** and ignore **gravity** and **collisions**. The particles are emitted in customizable **intervals** or when certain **criteria** are met. For the basic model above, emittion is set to one particle per minute.

These particles can me modified and given almost any **shape** and **size** by combinations of **geometry**, **kinetics** and **textures**. If, for example, we wanted to create **smoke**, one way would be to apply a relevant texture to the particles, as shown in **Picture 1.30** below. The **alpha values** on black areas are near **zero**, to give smoke its **transparency** effect.

By adjusting some values concerning particle physics, size, rotation and speed, we create the effect seen in **Picture 1.31**. Then, we apply the texture, adjust the effect to the specifics of the scene and get final result in **Picture 1.32**.



1.32 Smoke particle, finished and integrated

To demonstrate the **power** of the particle system creator, by adjusting some physical values (gravity strength, speed, rotation, collision..) on the system above and using a different texture, the

1.31 Smoke particle, untextured

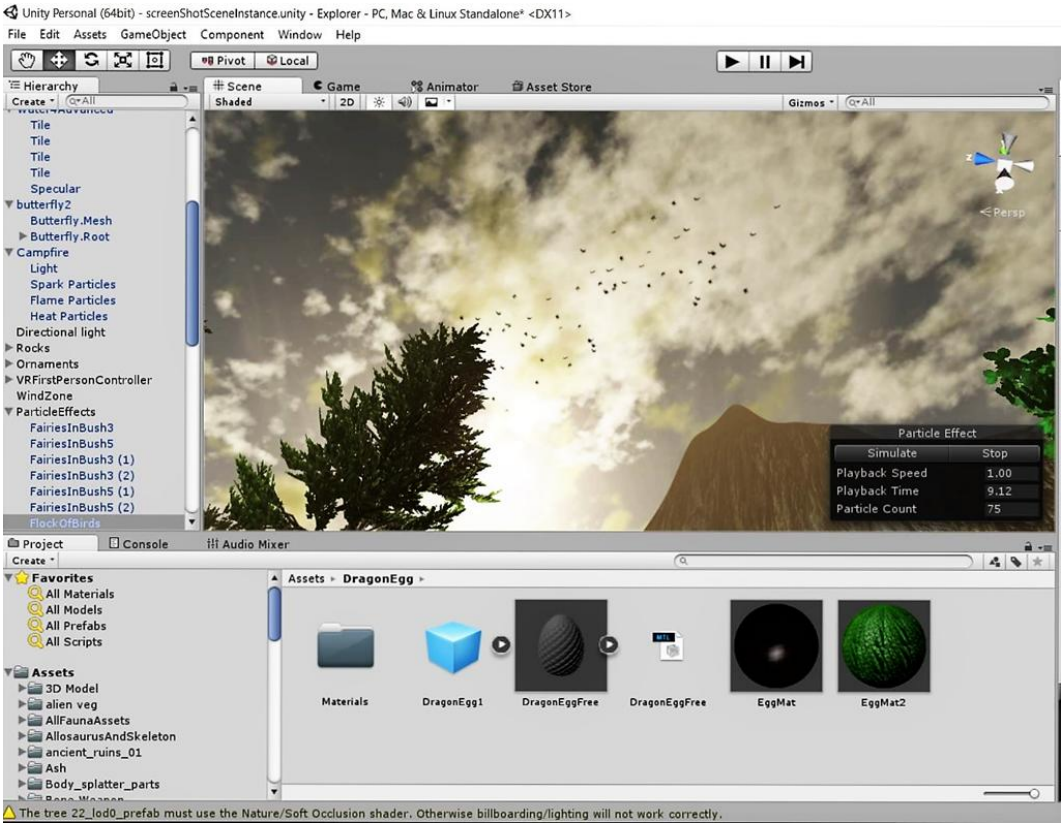
particle effect in **Picture 1.33** can, **relatively easily**, be created.

1.33 Waterfall particle system

Finally, apart from creating special effects, particle systems can also be used for a number of other things. **By creatively combining textures and components, particle systems can simulate anything.**

In this project, for example, **particle systems were used to create birds flying high across the sky**. This was achieved by constantly replacing a bird texture in various flying positions, much like the technique that was used to film cartoons in the past. Using a particle system instead of making and using actual bird models **saves** a lot of **processing power**, makes the end-product **lighter** and **smaller** and the birds easily **configurable**.

Below, **Picture 1.34** is the resulting effect in the scene.



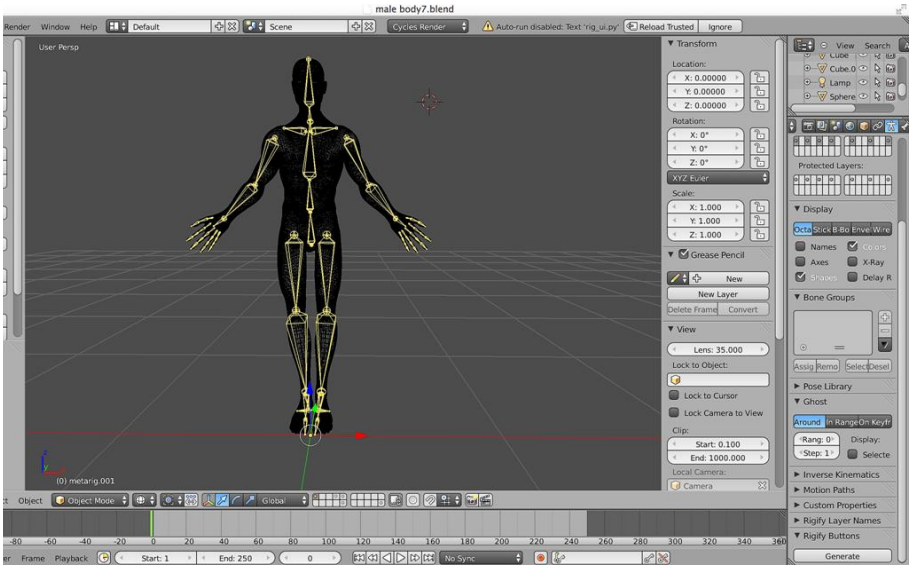
1.34 Bird flock particle system

1.2.4 Terrain: Creatures

Ornaments and flora are extremely important to any scene, but nothing brings a digital world to life more than mobile, interactive **creatures**. On the other hand, the process of integrating those creatures is a lot harder than for simple ornaments and particle effects.
Commonly, there are three major aspects to integrating a creature in the scene, **Creating a Rigged Model**, **Applying Movement and Animations** and **Pathfinding/Artificial Intelligence**.
Below, the process of integrating a creature (spider) in our digital world is explained.

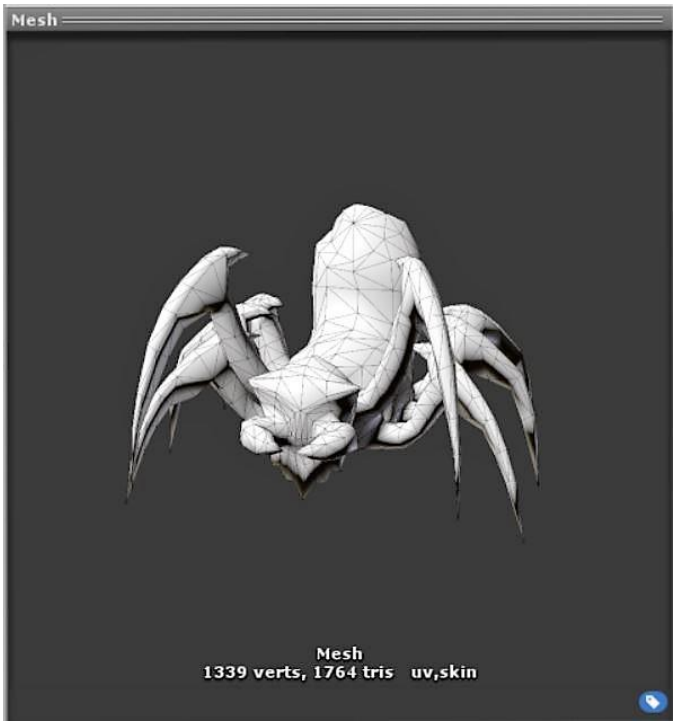
A **Rigged Model** is a special type of 3D model that contains a digital skeletal system, called an **armature** or **rig**, that is attached, mapped to it. Animation is then applied to the skeletal system, which in turn moves the model itself.
There are many reasons for doing animation this way. **First of all,** we can make minor changes in the model without the need to change every animation as well. A **second reason** is generalization. If we create an animation for a humanoid skeletal system, we can apply that animation to any model using that system. **We can animate a standard human male model, a clown model, even a fantasy human-like model like a hobbit using the same animations.**
The **third reason** is **automation**. A lot of specialized software exists for the automatic generation of skeletal systems and animations for specific skeletal systems. Such a thing would not be possible if every model was animated autonomously.

Rigged models, much like normal models, are created using specialized software (Blender, for example). Also, **normal models** can be made into **rigged models** using that same software. So, downloading suitable model and riggifying it is also an option.
Picture 1.35 is a human model rendered black to illustrate its rig.

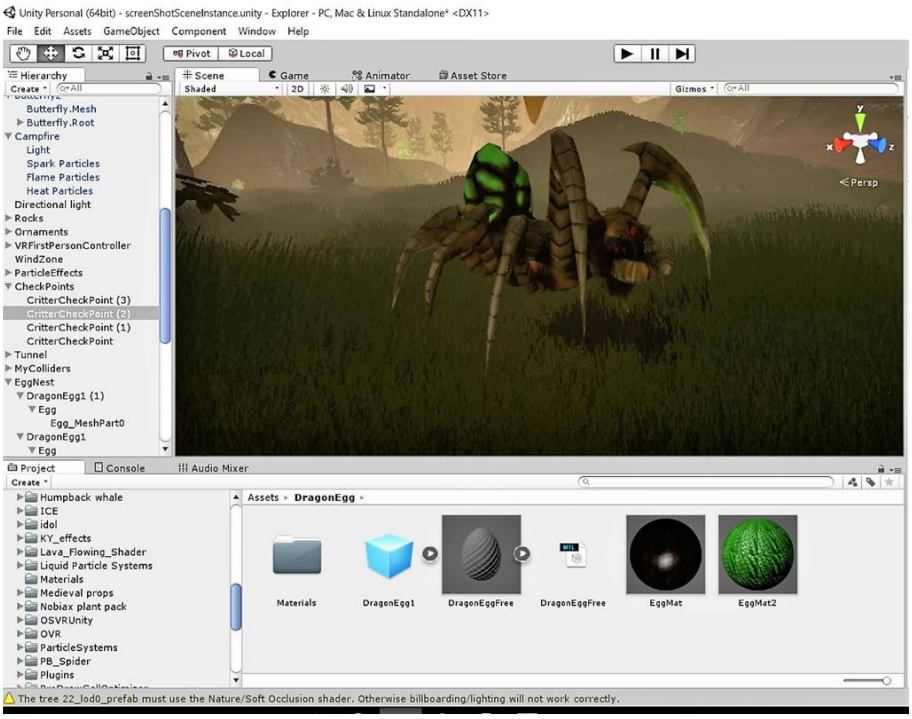


1.35 Rigged humanoid model

The spider model integrated in this digital world was downloaded from the Unity Asset Store. The model contained textures, an armature and several animations. **Picture 1.36 and 1.37** demonstrate the **basic** and the **textured**, imported spider model.



1.36 Untextured spider model

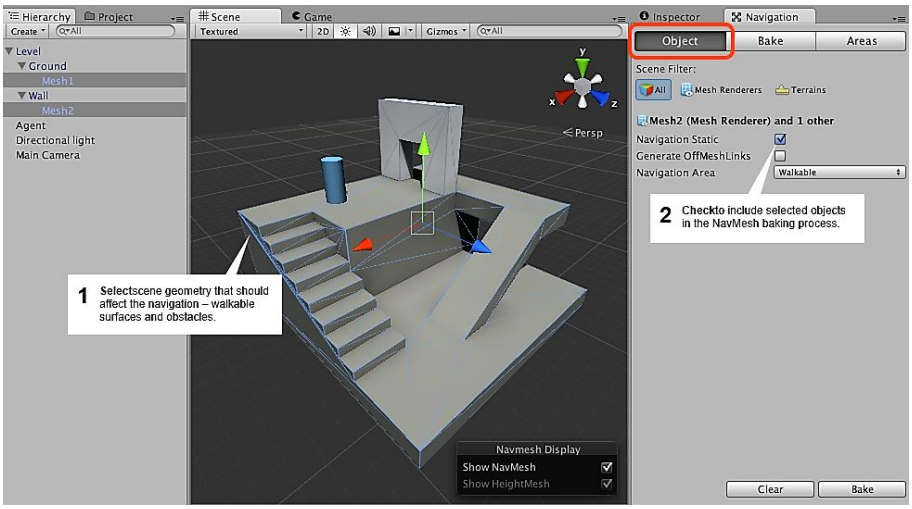


1.37 Textured, integrated spider model

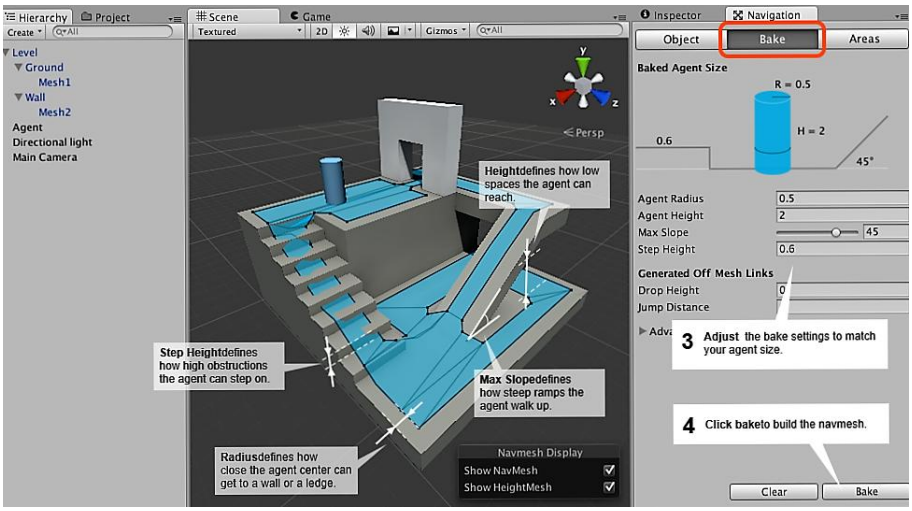
After **integrating** it, we need to enable the animations. This is usually done using Unities inbuilt animation system called **Mechanim**. This model, however, uses an older type of animation, **Legacy**. This meant that most of the animating for this model was done via **script** and was implemented within the code handling the **behavior**, or **AI** (Artificial Intelligence) of the spider.

The last thing we must do before the spider can run around the scene, interacting with objects is **Pathfinding**.

In order for a model to move around a scene safely and realistically, a map of the terrain needs to be drawn and areas where objects can and cannot reach must be specified on it. This process is called "**Baking** a Navigation Mesh, or **NavMesh**" and can be done automatically by Unity once we have the terrain and all objects (possible obstacles) set in the scene. The baking, along with the parameters needed for its correct completion, are visualized in **Pictures 1.38** and **1.39** below.



1.38 Non-navigational terrain

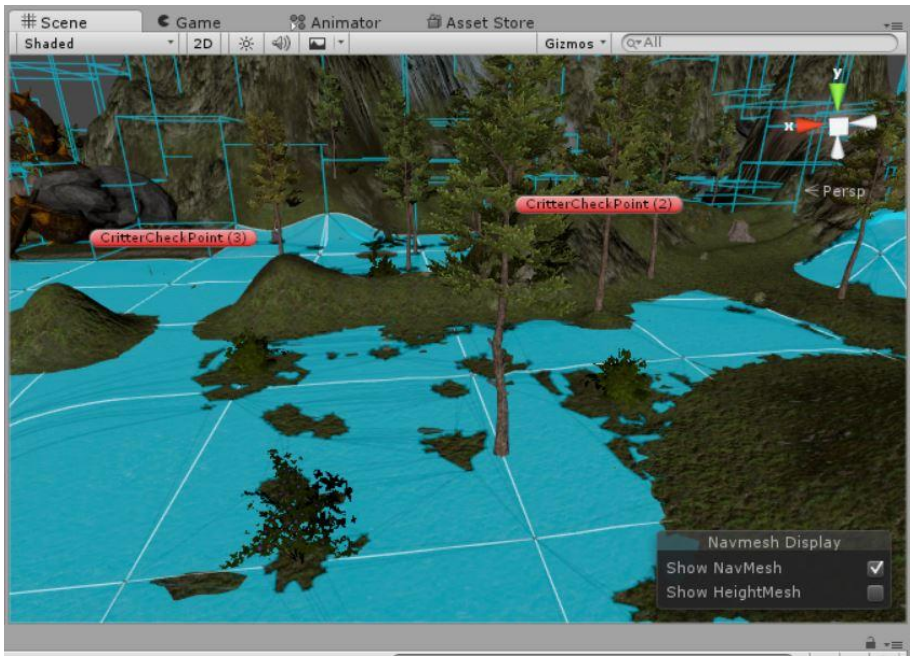


1.39 NavMesh on top of the terrain

For the bake above to be successful and functional, we also need to specify the circumference and height of our moving object. We do that by attaching a **NavMesh Agent component** to it and setting the correct parameters while baking the **NavMesh**.
The **NavMesh agent** is, basically, a cylinder containing the model. So, instead of moving the model around, we move the cylinder which has set geometry, making calculations a lot easier.
This can be visualized in **Picture 1.40**.

1.40 NavMesh Agent on spider model

To conclude navigation, **Picture 1.41** demonstrates the calculated NavMesh of our actual terrain. Blue declares the area available for navigation. Grass has been **omitted** in this picture, so the NavMesh can be properly viewed.



1.41 Terrain with baked NavMesh

After the NavMesh is baked and ready, it's very easy to **move a creature around** the scene and even create more creatures at runtime and have them navigate the scene too.
The following, simple C# script can be added on the spider to move it anywhere on the scene that the NavMesh allows.

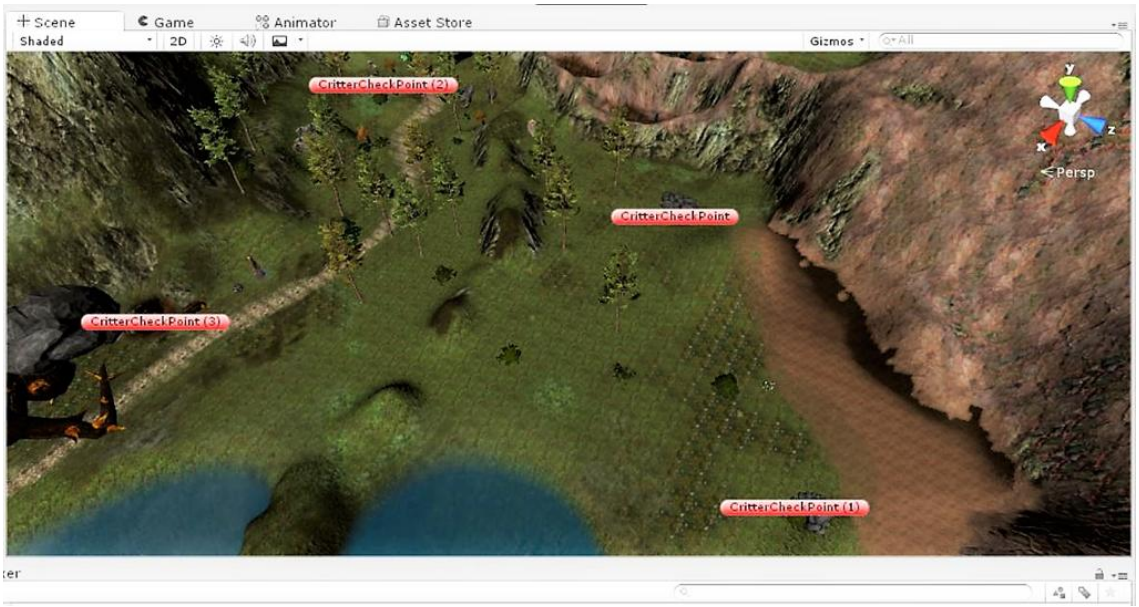
```
//MoveDestination.cs

using UnityEngine;
public class MoveDestination : MonoBehaviour
{
public Transform goal;
void Start ()
{
NavMeshAgent agent = GetComponent<NavMeshAgent>();  //get the NavMesh agent of the object
agent.destination = goal.position;  //send the agent(with the attached object) at “goal.position”
} }
```

To complete the script above, we could also enable the "walk" animation, so that the spider appears to be walking towards its destination.
The final script implementing the "**walk-to-destination**" function is:

```
//MoveDestination.cs

using UnityEngine;
public class MoveDestination : MonoBehaviour
{
public Transform goal;
void Start ()
{
NavMeshAgent agent = GetComponent<NavMeshAgent>();  //get the NavMesh agent of the object
agent.destination = goal.position;  //send the agent(with the attached object) at "goal.position"
animation.Play("walk", PlayMode.StopAll); //play the walking animation and stop all others
} }
```

Now that the **infrastructure** is ready, we can proceed to implement the **AI**. In our digital world, we wanted the spiders to roam around the forest **semi-randomly**, while **avoiding the player.** We also want them to "**die**" if the player steps on them.
To implement the above, four **Checkpoints (Picture 1.42)** were made in the scene. A pseudo-script implementing the behavior would be :

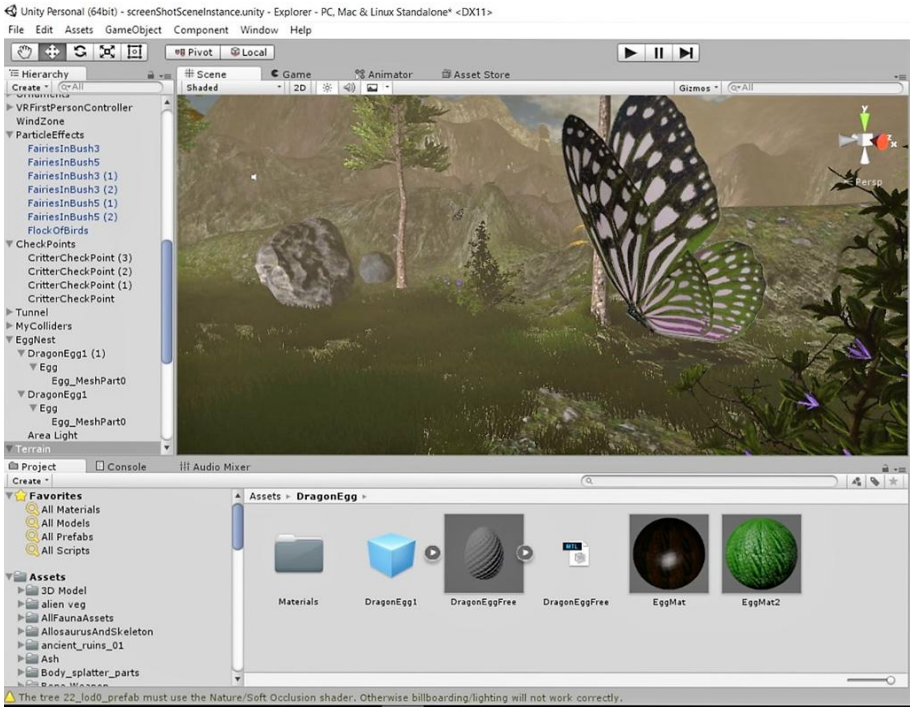
```
//pseudoScript
while (bool){ moveDestination( random(checkpoint) )
}
if( distance(player,spider)< x/10 )    //if the distance is very small,
                                                                    //stop walking, then die
 {
bool=false
dieSpider()
}
else_if(distance(player,spider)< (x) )    //if the distance is small
{                                                    //enough, run away
bool = false
runaway(player.position)
}
```



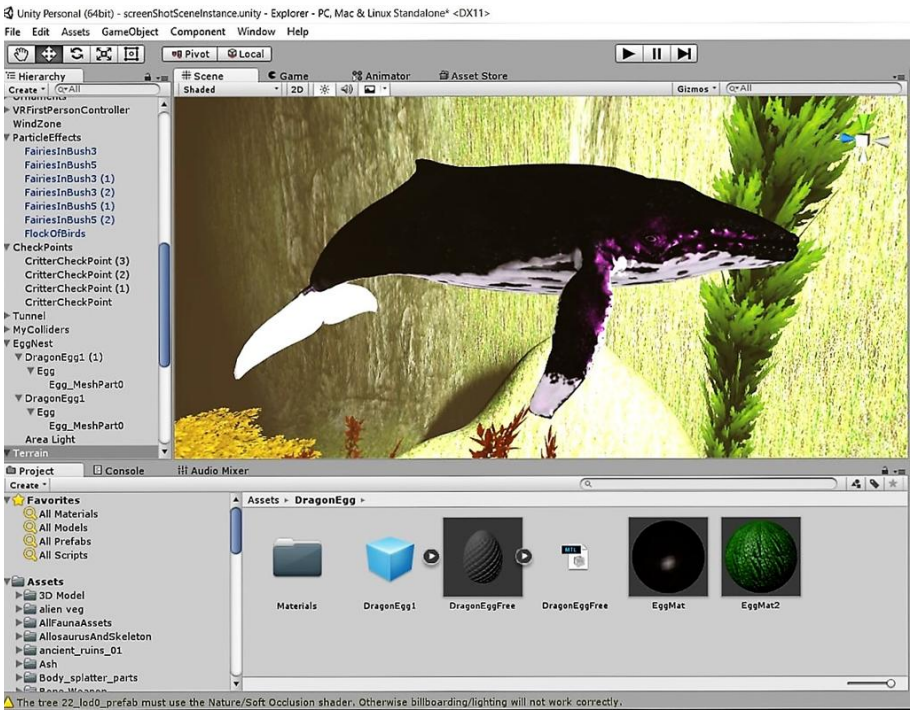
1.42 Creature Checkpoints

It's easily observed that the code used is not only **simple and short**, which means less chance for **logical errors**, it's also extremely **flexible** and **reusable**. This script can be **directly applied** on any creature with a NavMesh Agent and properly named animations and it will work perfectly. This means that **new creatures can be created and used at runtime**, that the scene can easily be **updated** with **new creatures** as they become available and that more checkpoints can be **added** or **removed**, as needed. Generally, this modular approach is extremely flexible, easy to implement and simple.

Below, **Pictures 1.43 and 1.44** show two other creatures implemented in a similar manner, a **flying** and an **aquatic** animal. For making proper NavMeshes on these creatures, **invisible**, **not-rendered terrains** were made.



1.43 Flying creature example: Butterfly



1.44 Aquatic creature example: Whale

1.2.5 Terrain: Physics

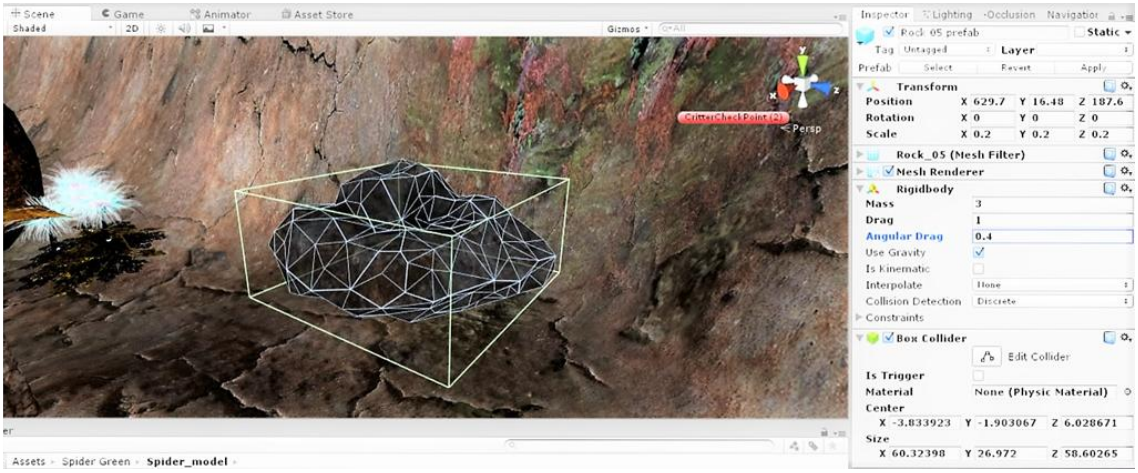
The term **Physics** in game-design is the ability to give a model(object) convincing physical behavior. An object in a game must accelerate correctly and be affected by collisions, gravity and other forces. Unity's built-in **physics engines** provide **components** that handle the physical simulation instead of the developer. With just a few parameter settings, we can create objects that behave passively in a realistic way (i.e., they will be moved by collisions and falls but will not start moving by themselves). By controlling the physics from scripts, we can give an object the dynamics of a vehicle, a machine or even a moving piece of cloth.

To utilize Unities physics system, all we have to do is add a **Rigidbody** component (component something that is contained in the Object) to it.
This will immediately cause any object to be affected by gravity. If we add a **Collider** component too, the object will start behaving realistically, colliding with the ground and with other objects having Collider components[2].

A Collider component is, essentially, a geometrical shape around an object. When this geometrical shape detect another such shape colliding with it, it notifies the Rigidbody, which, then, depending on our settings, applies physics.

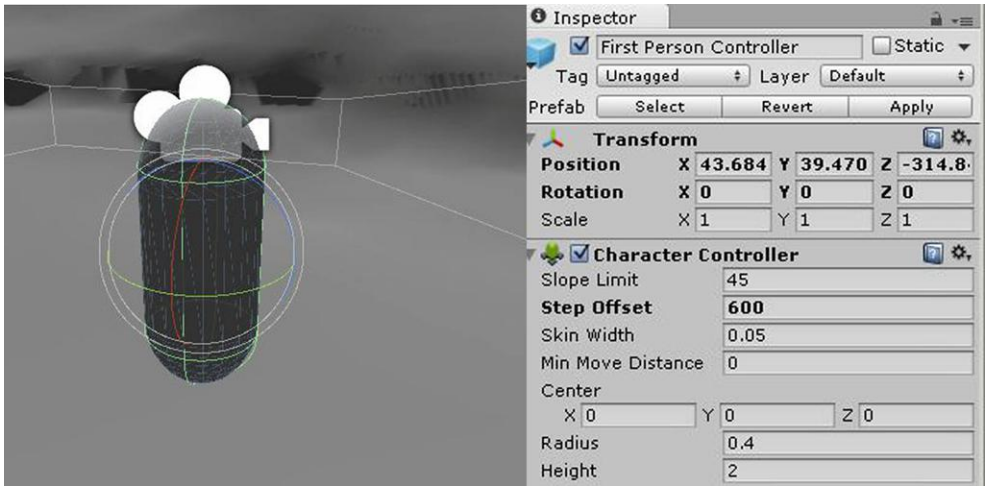
What's more, we can adjust the Rigidbody and Collider components' parameters to achieve even more realistic results, for example setting a big stones' **mass** and **drag** to a high value and making the stones' Collider a sphere that correctly envelopes it.

Picture 1.45 is such a rock object from our scene, enveloped in a correctly-sized Collider, set with a Rigidbody component.



1.45 Rock with Collider and Rigidbody Components

One last mention regarding physics in Unity are **Character Controllers**. Character Controllers (**Picture 3.1**) are special components, meant to be applied to the player-controlled character. Usually, the player-controlled character's **acceleration** and **movement** will not be physically realistic. It may be able to accelerate, brake and change direction almost instantly without being affected by momentum. Also, a character controller cannot walk through **static** colliders in a scene, and so will follow floors and be obstructed by walls. It can **push** rigidbody objects aside while moving but will not be accelerated by incoming collisions. This means that we can use the standard 3D colliders to create a scene, around which the controller will walk, but we are not limited by realistic physical behavior on the character itself[3].



3.1 First person Character Controller

1.2.6 Terrain: Sound

Sound effects add, as one easily understands, a great deal of **immersion**, **realism** and **life** to any **digital environment**. Even more so when the environment is perceived through **VR**, since the user has **limited perspective**, and sound can be used as a **point of reference** or **indicator** to the user of where he/her **attention** is needed.

In real life, sounds are emitted by objects and heard by listeners. The way a sound is perceived depends on a number of factors. A listener can tell roughly which direction a sound is coming from and may also get some sense of its distance from its loudness and quality.

A fast-moving sound source (like a falling bomb or a passing police car) will change in pitch as it moves as a result of the **Doppler Effect**.

Also, the surroundings will affect the way sound is reflected, so a voice inside a cave will have an echo but the same voice in the open air will not.

To simulate the effects of position, Unity requires sounds to originate from **Audio Sources** attached to objects. The sounds emitted are then picked up by an **Audio Listener** attached to another object, most often the player controlled character. Unity can then simulate the effects of a source's distance and position from the listener object and play them to the user accordingly.

The relative speed of the source and listener objects can also be used to simulate the Doppler Effect for added realism.

Unity can't calculate echoes purely from scene geometry but you can simulate them by adding **Audio Filters** to objects.

For example, you could apply the Echo filter to a sound that is supposed to be coming from inside a cave. The Unity **Audio Mixer** is a tool that allows the developer to mix various audio sources, apply effects to them, and perform mastering[4].

So, first of all, we add an Audio Listener to our first-person character. Then we download all the needed sounds of our scene (**MP3** and **WAV** are the most frequently used formats) and add them together with an Audio Source, to every single object we need producing sound.

We also create two empty, not-rendered objects that only contain an Audio Source for producing ambient sounds (birds singing, wind howling etc.)

Last but not least we add an audio source to the feet of the player-controlled character for producing stepping sounds. A small script detects where the player steps on, be it rocky, grassy or sandy ground and swaps the stepping sound accordingly.

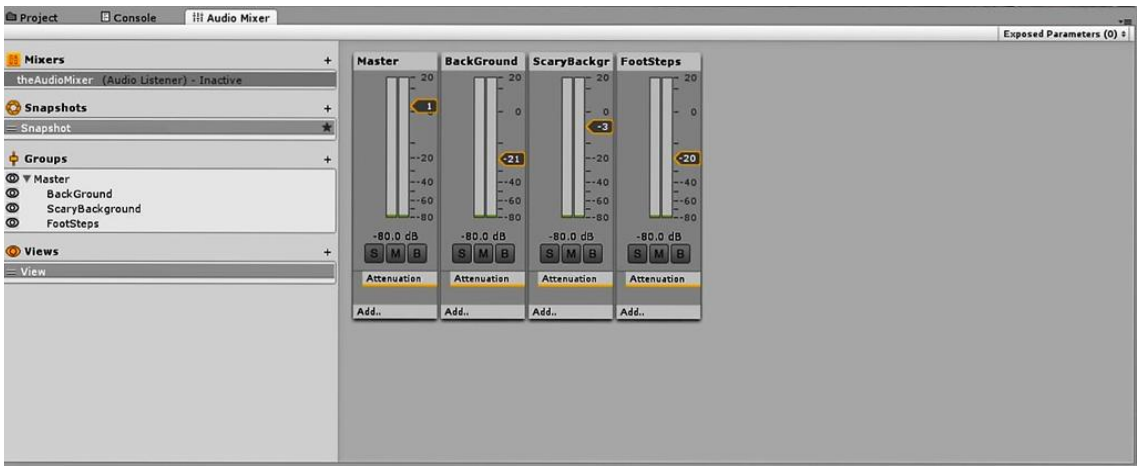
Below is part of the script, a function that produces the stepping sound.

```
//call this when a footstep sound is needed
public void Footstep(){
        for(int i = 0; i < groundTypes.Count; i++){
            for(int k = 0; k < groundTypes[i].textures.Length; k++){

if(currentTexture == groundTypes[i].textures[k]){

    footstepAudio.PlayOneShot(groundTypes[i].sounds[Random.Range(0,groundTypes[i].sounds.Length)]);
      Debug.Log(currentTexture);
```

```
Debug.Log(groundTypes[i].sounds[Random.Range(0, groundTypes[i].sounds.Length)]);

    }
                }
            }
    }
```

Another sound addition that is of interest is the one the spider emits when approached by the player. **A single line of code can make an extremely big difference in the realism of the world**. Many testers almost jumped off their seats when faced with a creature that they would have completely **overlooked** if not for the sound it made.

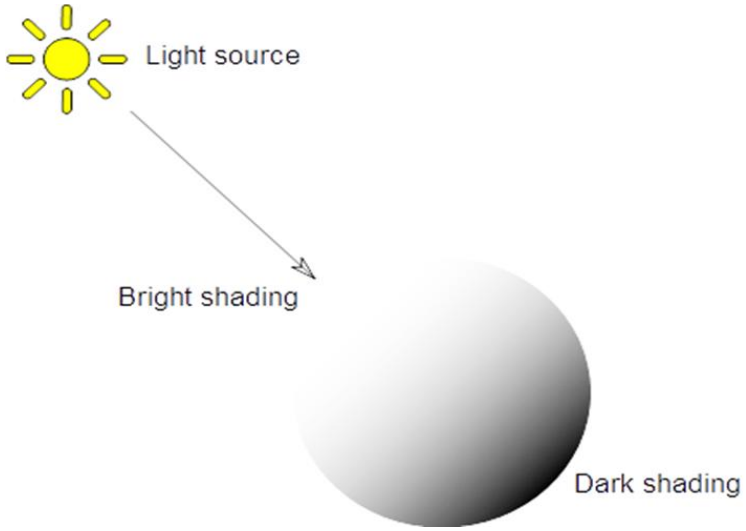
Finally, the Audio Mixer tool (**Picture 3.2**) that Unity provides is a great way to group and manage sounds. Instead of having to configure every single sound effect individually, the developer can add them in groups and handle them together. Effects meant to be ambient can, for example, be placed in an Ambient Sounds category and increase-decrease their intensity or apply filters to them as a group.



3.2 Audio Mixer

1.2.7 Terrain: Lighting

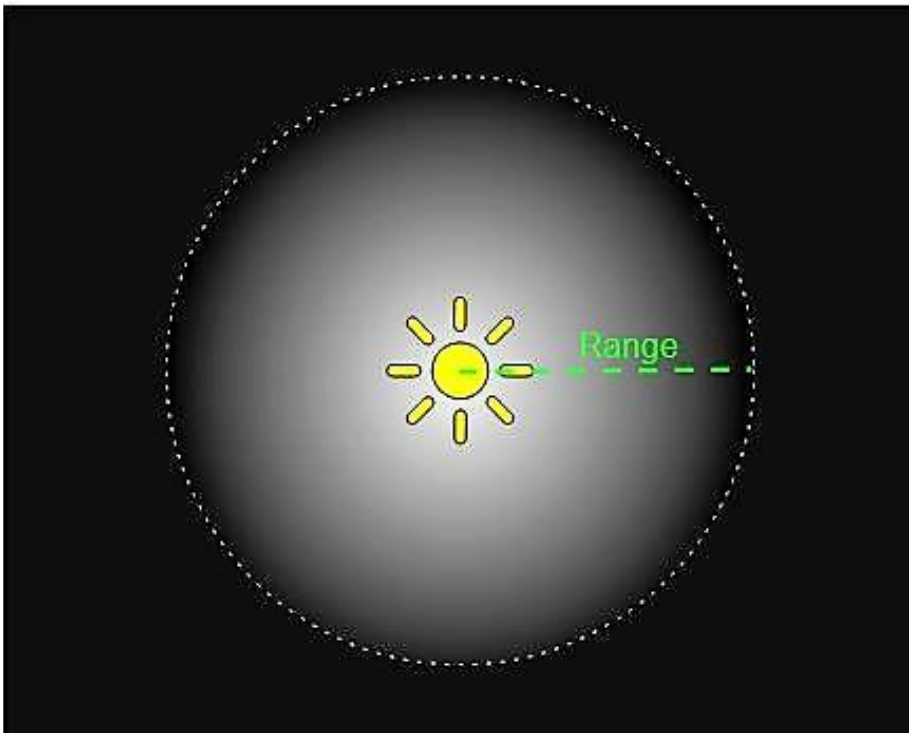
In order to calculate the shading of a 3D object, Unity needs to know the intensity, direction and color of the light that falls on it. **Picture 1.46**[5] demonstrates an example of shading.



1.46 Direct Light Shading Example

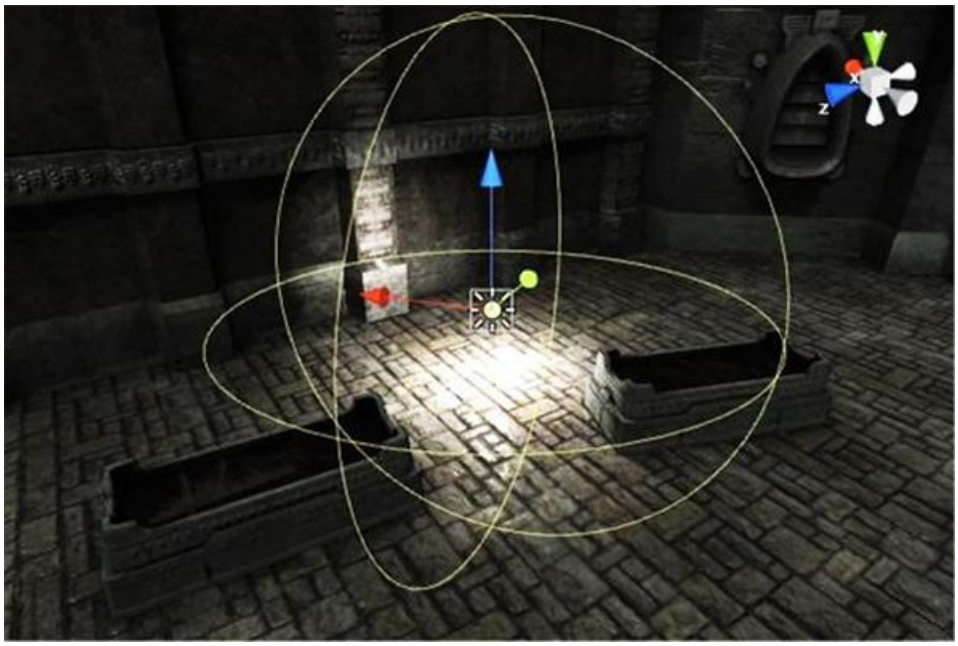
These properties are provided by **Light** objects in the scene. The base color and intensity are set identically for all lights but the direction depends on the type of light being used. Also, the light may diminish with distance from the source. The four types of lights available in Unity are described below

A **Point Light** is located at a point in space and sends light out in all directions equally. The direction of light hitting a surface is the line from the point of contact back to the center of the light object. The intensity diminishes with distance from the light, reaching zero at a specified range.
A point light is visualized in **Picture 1.47.**



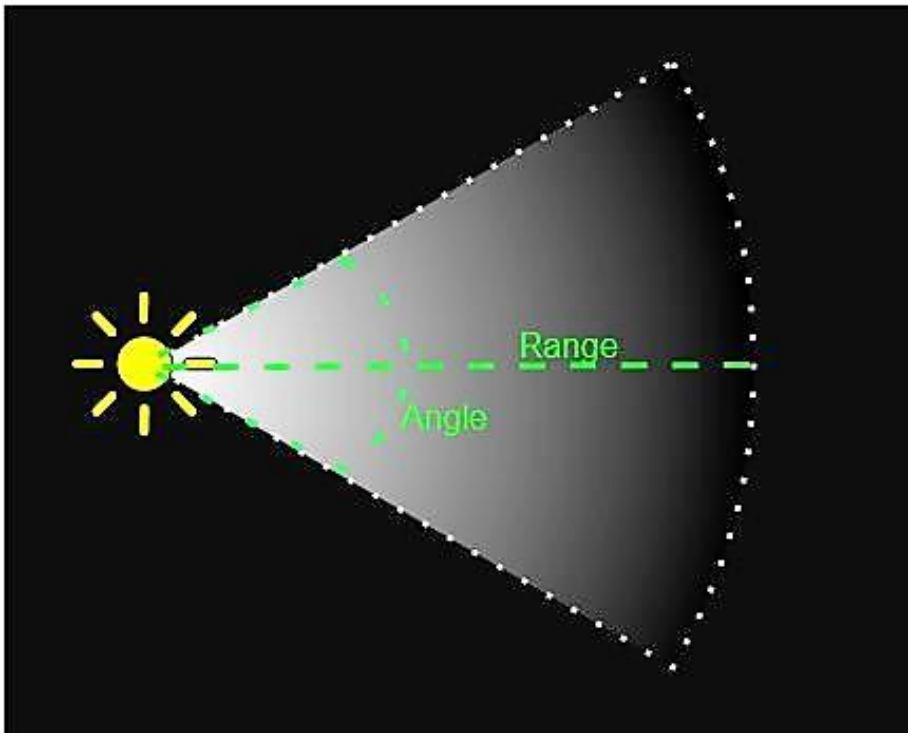
1.47 Point light

Point lights are useful for simulating lamps and other local sources of light in a scene. They can also be used to make a spark or explosion illuminate its surroundings in a convincing way. **Picture 1.48** demonstrates the effects of a point light in the scene.



1.48 Effect of Point Light in the Scene

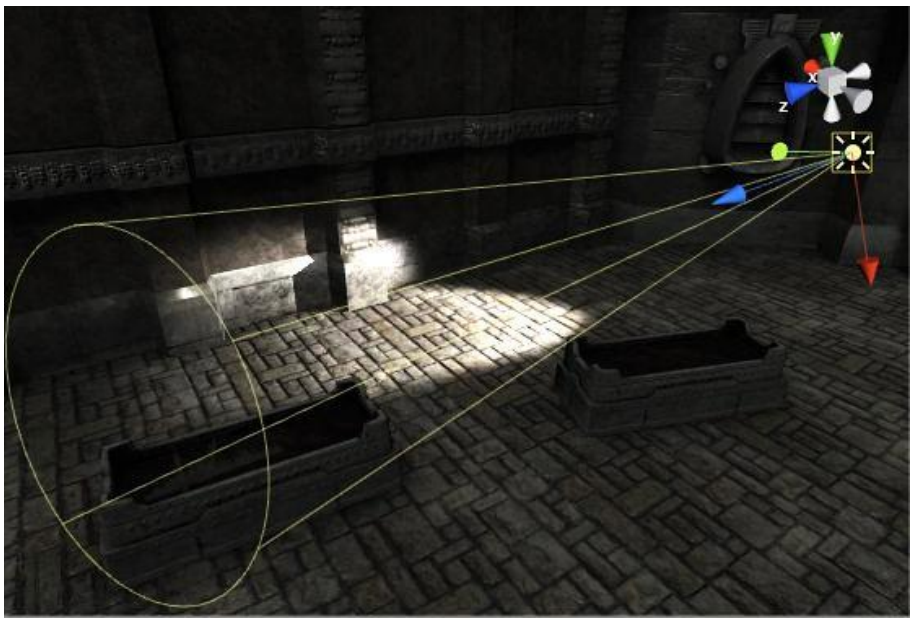
Much like a point light, a **Spot Light (Picture 1.49)** has a specified location and range over which the light falls off. However, the spot light is constrained to an angle, resulting in a cone-shaped



1.49 Spot light

region of illumination. The center of the cone points in the forward (**Z**) direction of the light object.

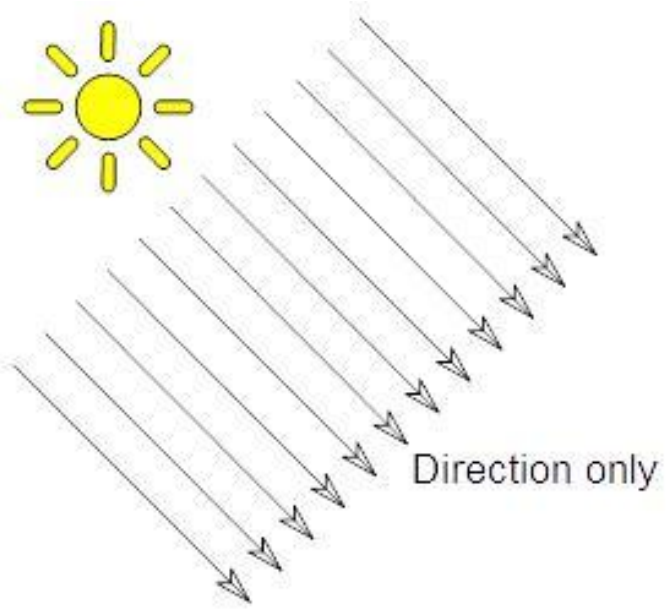
Spot lights are generally used for artificial light sources such as flashlights, car headlights and searchlights. With the direction controlled from a script or animation, a moving spot light will only



1.50 Effect of a Spot Light in the scene

illuminate a small area of the scene and create dramatic lighting effects (**Picture 1.50**).

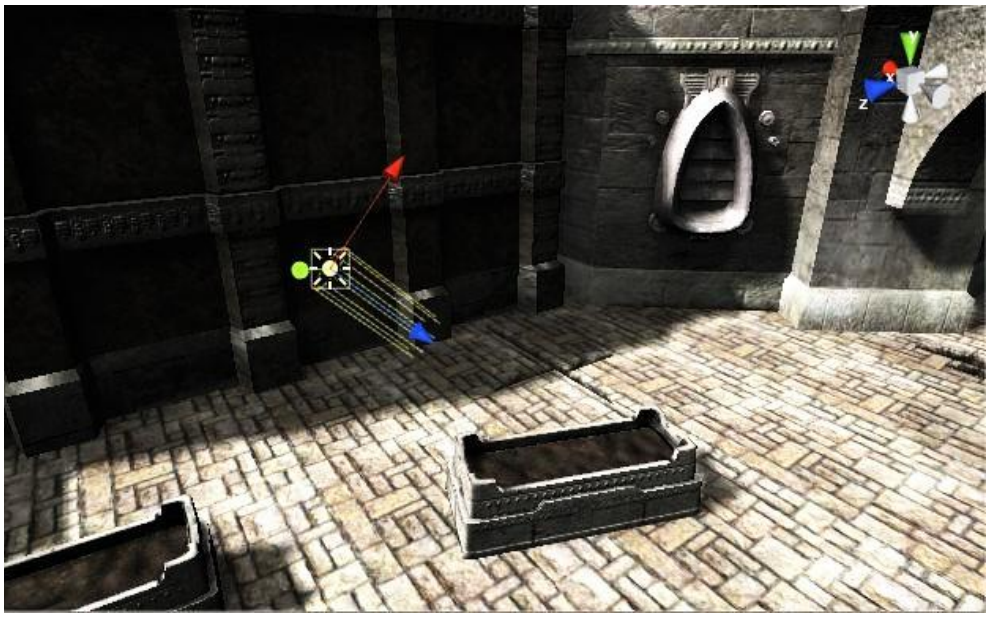
A **Directional Light (Picture 1.51)** does not have any identifiable source position and so the light object can generally be placed anywhere in the scene. All objects in the scene are illuminated as if the light is always from the same direction. The distance of the light from the target object is not



1.51 Directional light

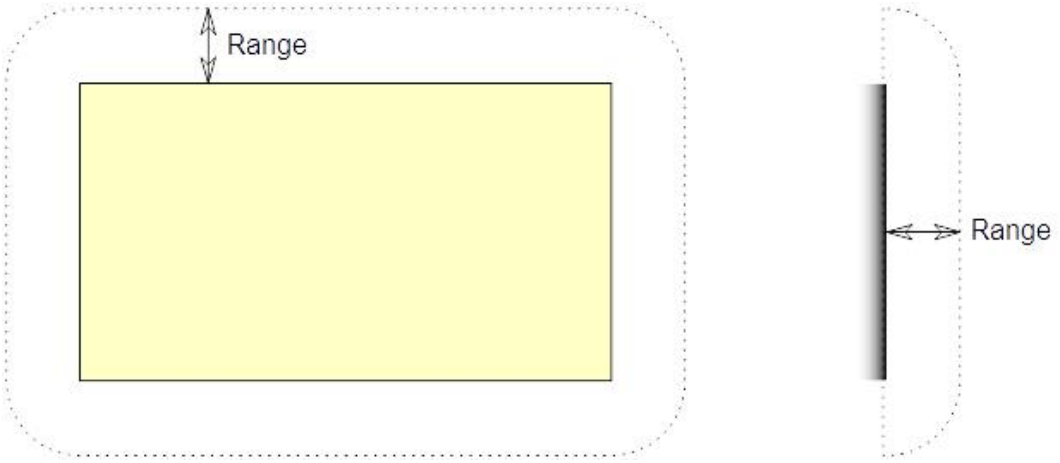
defined(infinite) and so the light does not diminish.

Directional lights represent large, distant sources that exist a position outside the range of the game world (**Picture 1.52**). In a realistic scene, they can be used to simulate the sun or moon. In an abstract game world, they can be a useful way to add convincing shading to objects without exactly specifying where the light is coming from. When checking an object in the scene view (to see how its mesh, shader and material look, for example) a directional light is often the quickest way to get an impression of how its shading will appear. For such a test, we are generally not interested in where the light is coming from but simply want to see the object looks “solid” and whether there are glitches in the model.



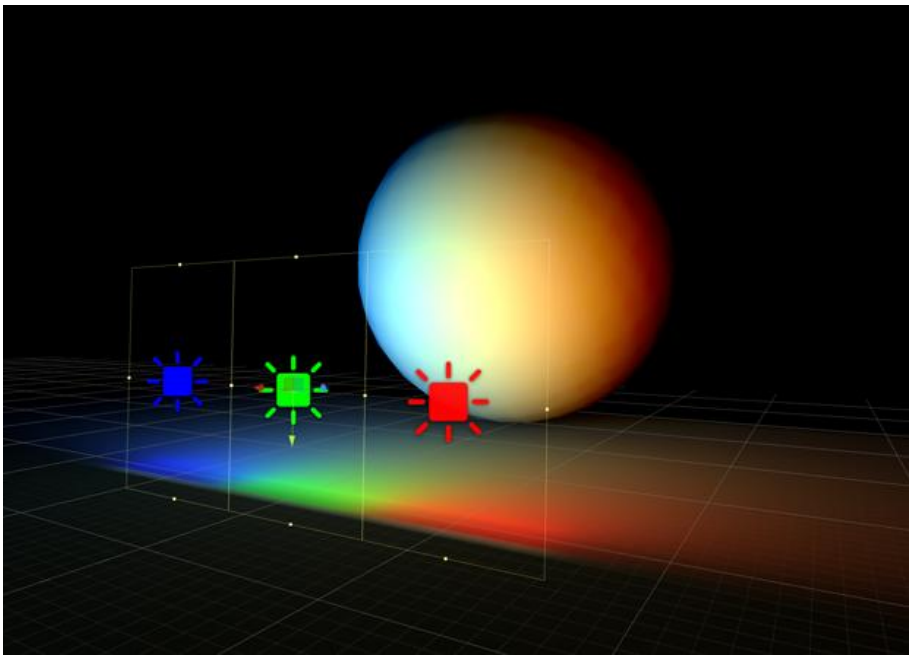
1.52 Effect of a Directional Light in the scene

An **Area Light (Picture 1.53)** is defined by a rectangle in space. Light is emitted in all directions, but only from one side of the rectangle. The light falls off over a specified range. Since the lighting calculation is quite processor-intensive, area lights are not available at runtime and can only be baked into **lightmaps**.



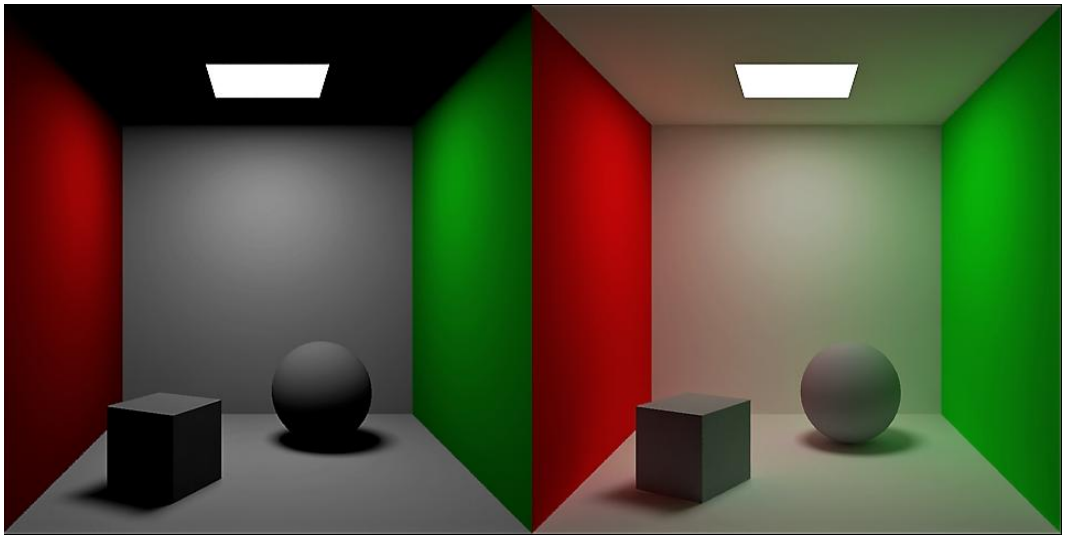
1.53 Area Light

Since an area light illuminates an object from several different directions at once, the shading tends to be more soft and subtle than the other light types (**Picture 1.54**). A proper use to it would be the creation of realistic street lights or a bank of lights close to the player. A small area light can simulate smaller sources of light (such as interior house lighting) with a more realistic effect than a point light.



1.54 Effect of an Area Light to the scene

Global Illumination (GI) is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light). Modeling indirect lighting allows for effects that make the virtual world seem more realistic and connected, since objects affect each other's appearance. One classic example is 'color bleeding' where, for example, light hitting a red wall at a specific angle will cause red color to be bounced onto the wall next to it (**Picture 1.55**). Another is when sunlight hits the floor at the opening of a cave and bounces around inside so the inner parts of the cave are illuminated too.



1.55 GI Effects: Color Bleed

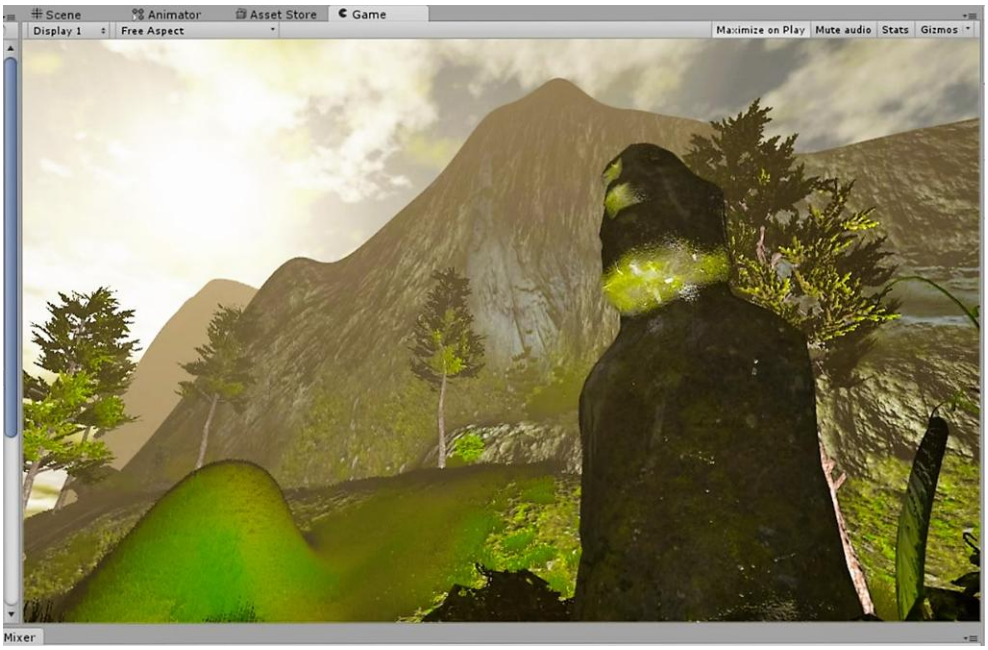
Traditionally, video games, digital worlds and other realtime graphics applications have been limited to direct lighting, while the calculations required for indirect lighting were too slow so they could only be used in non-realtime situations such as CG animated films.

A way for games to work around this limitation is to calculate indirect light only for objects and surfaces that are known ahead of time not to move around (that are **static**). That way the slow computation can be done ahead of time, and since the objects won't move, the indirect light that is pre-calculated this way will always be correct, even at runtime. Unity supports this technique, called **Baked GI** (also known as **Baked Lightmaps**), which is named after "the bake" - the process in which the indirect light is pre-calculated and stored.

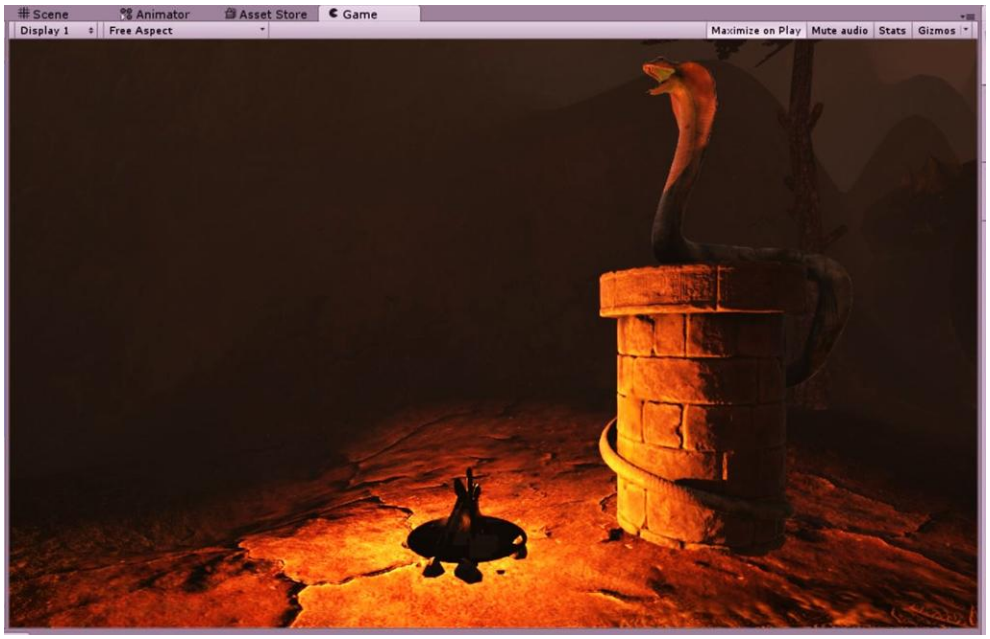
In addition to indirect light, Baked GI also takes advantage of the greater computation time available to generate more realistic soft shadows from area lights and indirect light than what can normally be achieved with realtime techniques.

One noteworthy disadvantage of this technique, however, is that it dramatically increases the size of the end-product due to the lightmaps it bakes, stores and uses.

By using all the lighting techniques above, we greatly increase the realism and visual fidelity of our digital world. **Pictures 1.55 and 1.56** below present two highlights from within that world. It is worth mentioning that, due to a bug in Unity presented at the time, these results were done using the minimum quality settings.



1.55 Highlights: Directional Light

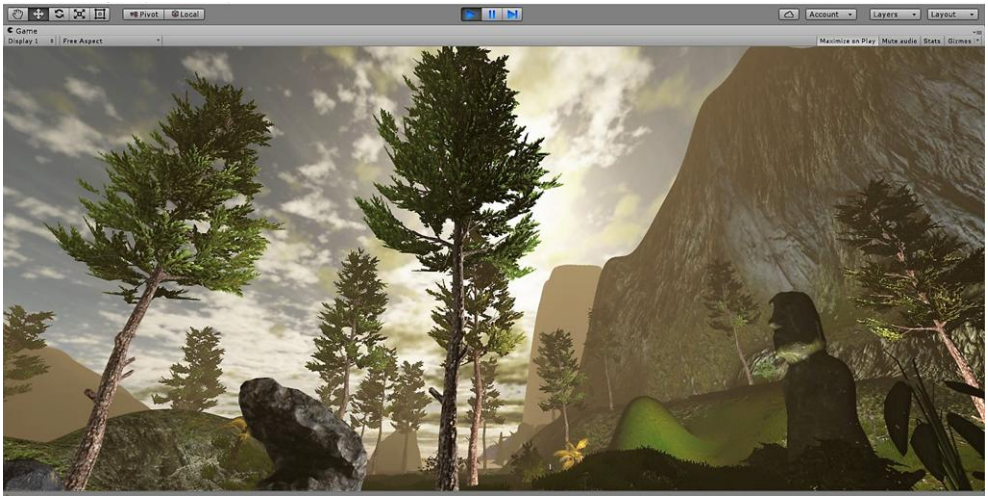


1.56 Highlights: Point Light (Colored)

1.2.8 Terrain: Image Post-processing Effects

Image post-processing effects are special effects applied on the digital world, often to simulate physical camera and film properties and their correct use can add a great deal to the look and feel of our world, especially since it's designed to be experienced in first-person view.
Important effects in this category include **Sun Shafts**, **Bloom**, **Antialiasing** and **Color Correction Curves**.

Sun Shafts, or **god ray effect,** simulates the radial light scattering that arises when a very bright light source is partly obscured. **Picture 3.4** demonstrates the scene without any effects and **Picture 1.56** god rays are included.



3.4 No Post-Processing Effects

1.57 Sun Shafts (God Ray Effect)

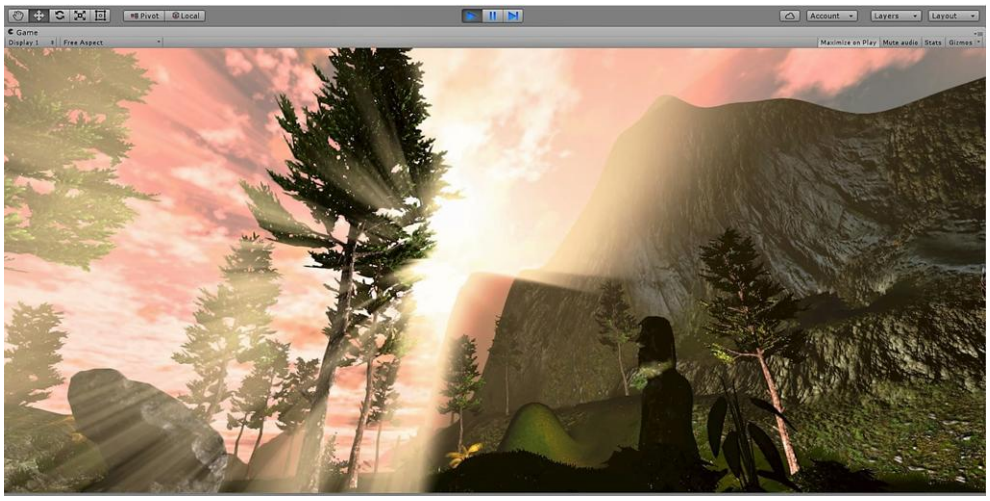
Color Correction[6] is an effect used to make **color adjustments** for each color channel. **Depth based** adjustments allow you to vary the color adjustment according to a pixel's distance from the camera. For example, objects on a landscape typically get more **desaturated** with distance due to the effect of particles in the atmosphere scattering.
Selective adjustments can also be applied, so you can swap a target color in the scene for another color of your own choosing.
Color Correction Curves is a tool visualizing and performing the above function by mapping color channel values on curves, using a Cartesian system.
Saturation is an easy way to adjust all color saturation or desaturation (until image turns black & white) which is an effect that is not achievable with curves only.

1.58 Landscape before depth-based Color Correction

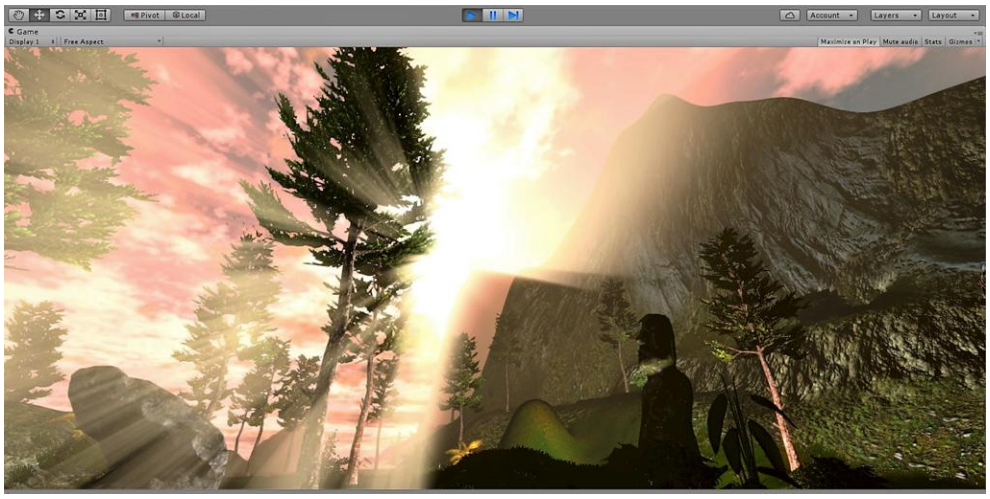
Pictures 1.58 and **1.5**9 visualize the change Color Correction can have on a scene.

1.59 Landscape after depth-based Color Correction

Blooming is the optical effect where light from a bright source (such as a glint) appears to leak into the surrounding objects. The **Bloom** image effect adds the effect above and also automatically generates **lens flares** in a highly efficient way.
Bloom is a very distinctive effect that can make a big difference to a scene and may suggest a magical or dreamlike environment especially when used in conjunction with HDR rendering. On the other hand, given proper settings, it's also possible to enhance photorealism using this effect. Glow around very bright objects is a common phenomenon observed in film and photography, where luminance values differ vastly.
Pictures 1.60 and **1.61** demonstrate the difference the **Bloom** effect can make in a scene.

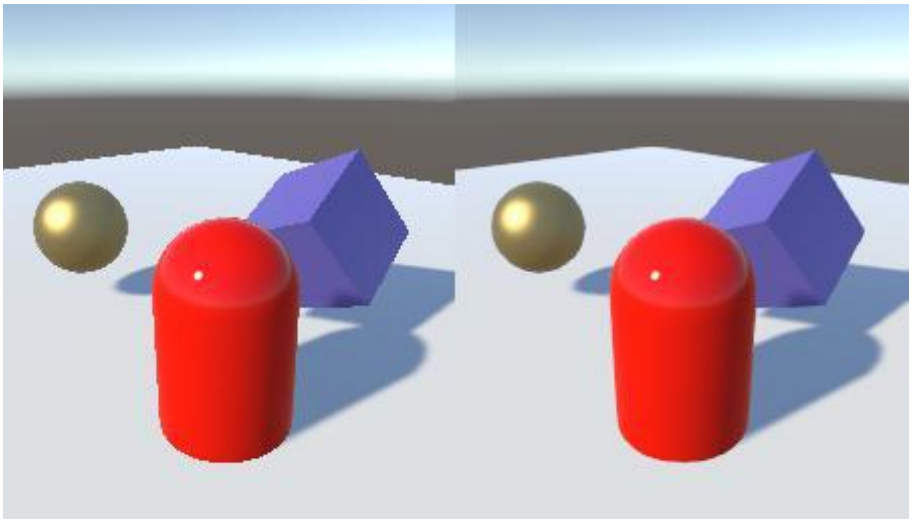


1.60 Landscape before **the application** Bloom



1.61 Landscape after the application of Bloom

Finally, the **Antialiasing**[6] effect is a post processing effect offers a set of algorithms designed to give a smoother appearance to graphics. When two areas of different color adjoin in an image, the shape of the pixels can form a very distinctive “staircase” along the boundary. This effect is known as **aliasing** and hence antialiasing refers to any measure which reduces the effect.



1.62 No antialiasing With Antialiasing (FXAA1PresetB algorithm used)

1.3 Terrain: VR Integration

Virtual reality is an artificial environment that is created with software and presented to the user in such a way that the user suspends belief and accepts it as a real environment. On a computer, virtual reality is primarily experienced through two of the five senses: **sight** and **sound**[7].
Although there were many different attempts regarding hardware for creating immersive VR, the latest and most successful is using **Headsets**, devices mounted on the head in such a way that the digital world is being presented as real. To do that, either the digital world is correctly rendered in **two different screens**, one for each eye or **two lenses correctly focus eyesight into two parts of one screen** (**Picture 1.63, Picture 1.64**).

1.63 VR Headset, Razer's OSVR Hacker Dev Kit

1.64 Game rendered in VR

For properly rendering the digital environment on **OSVR** (**Picture 1.63**), the official SDK needs to be downloaded from GitHub.
After that, there are three steps to **integration**[8]. First of all, we need to supply our f**irst-person controller** with a specific set of components (contained in the SDK). A ready-for-use, complete controller is also provided, that would mean, though, that we would need to remake all the post-processing image effects.
Those components are mainly the special **VR camera** and the script handling the extra **VR head movement** (so that the movement of the players' head translates into movement of the in-game characters' head).
Another integration step is the addition in the scene of a **ClientKit**. This object handles the connection with the hardware.
Finally, an **OSVR** server must be run and connected to the ClientKit.

1.65 VR Enabled First-person controller

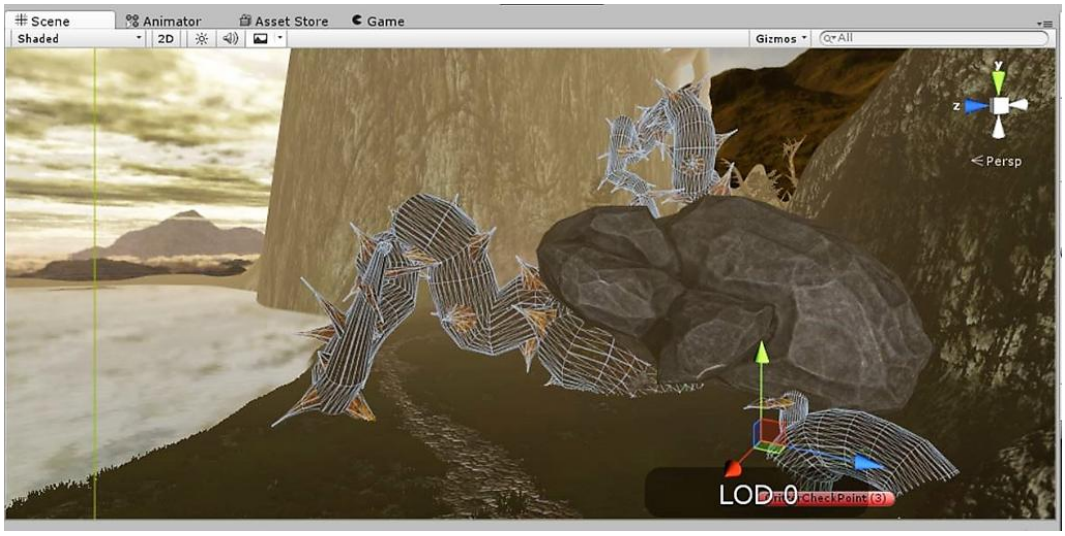
While the process itself of integrating VR in a digital world is not hard or time-consuming, the **true challenge** is making said world abide with certain **rules** or **requirements**. These rules, while not mandatory, **make VR safe** and **enjoyable** to the users.
One example of a challenge is that, usually, the VR Headsets have extremely **high-resolution screens**. Since the world is rendered on those screens, the **framerate** will **drop significantly** compared to a standard **Full-HD computer screen**. What's more, low framerate in VR has been proven to cause mild to severe motion sickness to its users[10]. That reason is why **Oculus**, a market leader in VR systems, along with most other prodigies of VR, recommend striving for a steady framerate of over **60 frames per second**.
This means that the developer should strive to achieve sufficiently high visual fidelity to succeed in immersing the user while also putting great effort in keeping the fps high. Towards that end, the digital world needs a very high and professional level of **optimization**.

1.3.1 Terrain : Optimization

Optimizing a game or digital world is a **not** an exact science. There are countless ways to increase framerate, but not one of them can **guarantee** good enough results in any situation by itself. Usually it's a combination of **many different techniques** and on every aspect of the game that produce a sufficient result.
Examples would be **reducing the polygon-count** when making/integrating models and adding **LOD** support to them, avoiding using code that overly-burdens the system, **reducing the rendering distance** for vegetation and objects, and adding a **fog effect** to cover it up and retain realism (**Picture 1.66**).
Finally, a more sophisticated, **specialized** technique for decreasing the amount of rendered objects and greatly increasing framerate is **Occlusion Culling**.

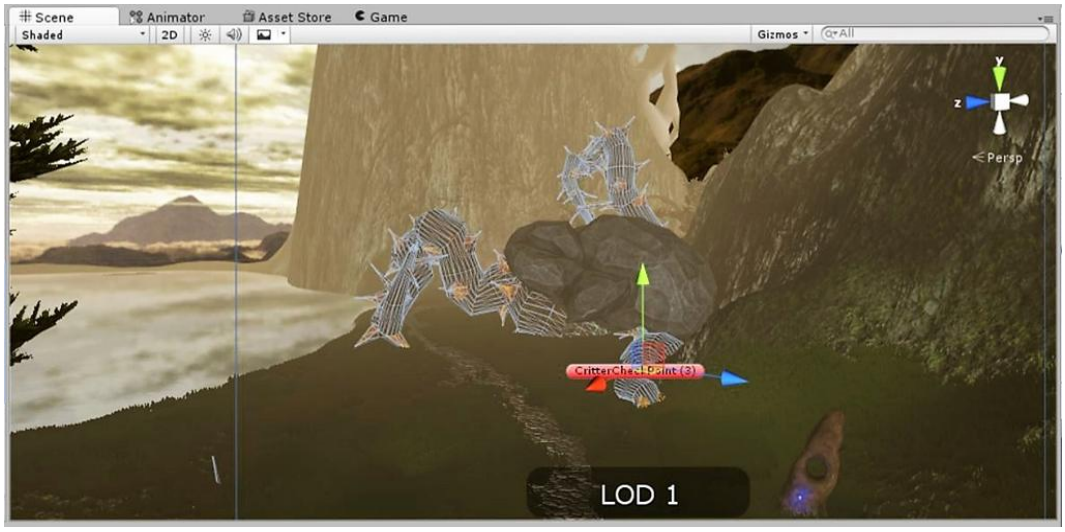
LOD, or **Level of Detail**, is a smart way of **optimizing models** while retaining **detailed environments and realism**. When an object in the scene is a long way from the camera, the amount of **detail** that can be seen on it is greatly **reduced**. However, the same number of triangles will be used to render the object, even though the detail will **not be noticed**.
LOD rendering (**Pictures 1.66 – 1.70**) allows you to **reduce** the number of triangles rendered for an object as its distance from camera **increases**. As long as your objects aren't all close to the camera at the same time, **LOD** will reduce the load on the hardware and **improve rendering performance**.

LOD 0, the fully detailed, realistic model. When the player is this close or closer to the model, he/she experiences **the fully detailed versio**n.



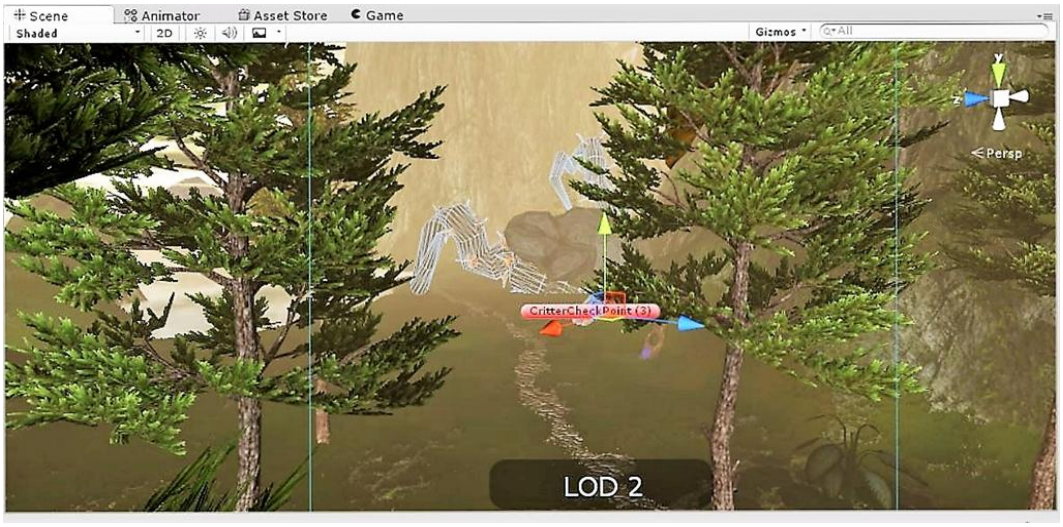
1.66 LOD 0: The fully detailed version of the model is rendered

As the player moves away, the model gets replaced by a **simplified version of itself** (**Picture 1.67**).



1.67 LOD 1: The model loses detail

As distance **increases** even further, an **even less detailed version of the model** is used. Due to the distance, though, the differences **cannot be perceived**. **Picture 1.68** remonstrates **LOD 2** from a distance and **Picture 1.69** the LOD difference from up close.



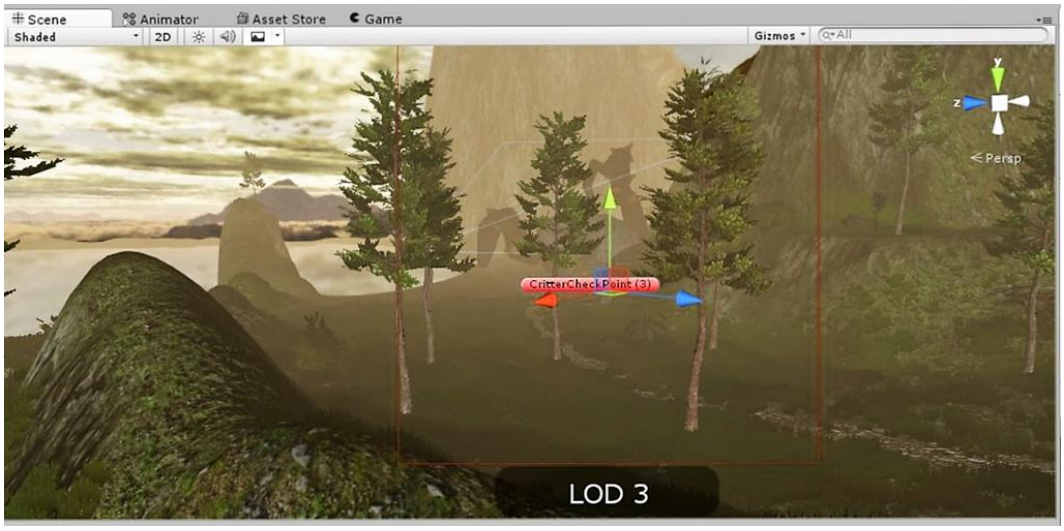
1.68 LOD 2: Model replaced by an even less detailed version

Note the huge difference in the model detail compared to how little of a difference noticed when viewed from a distance in **Pictures 1.66** and **1.68.**

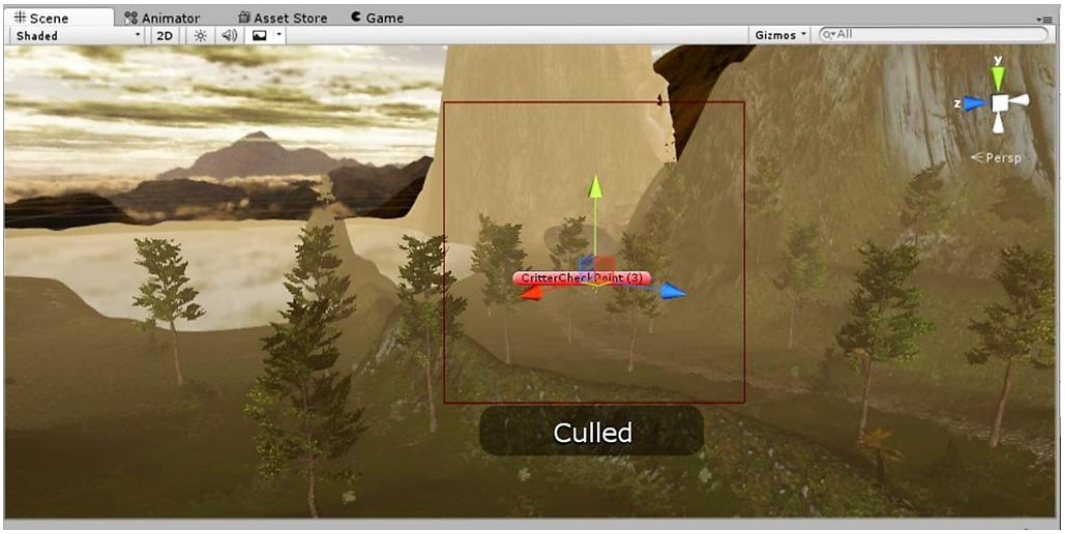
1.69 LOD 0

LOD 2

LOD 3 is the **lowest level** for this model and the distanced rendered in LOD is the **max distance** this model can be viewed from. In LOD 3, the model is replaced by a **billboard**, a 2D image. Even so, from such a great distance and using fog, it is barely noticeable.



1.70 LOD 3: The model is replaced with a billboard version of it



3.5 Culled Model

From this point and further, the model is not rendered(**culled**) at all (**Picture 3.5**).

It's easy to understand that applying LOD to every model in the scene will greatly increase performance. But it should be applied carefully, properly blending the switches in geometry and setting the distances for the LODs.

A good way of hiding the lack of detail in far-away models is **fog**. Seen above in **Pictures 1.66-1.70**, **fog** is a lighting effect that can be created in various ways. Unities lighting system has an inbuilt global **fog generator,** while in the Asset Store one can find 3D, volumetric fog systems and even some created by using Particle Systems.
The one used in this project is Unities inbuilt system, set to **linear,** meaning the fog will increase linearly, starting from the player and ending 1000 units away.
It's also set to an earthy orange color to assimilate better with the horizon (Picture 1.70).

Occlusion Culling is a feature that disables rendering of objects when they are not currently seen by the camera because they are obscured (occluded) by other objects. This does not happen automatically in 3D computer graphics since most of the time objects farthest away from the camera are drawn first and closer objects are drawn over the top of them (this is called "overdraw"). Occlusion Culling is different from **Frustum Culling**. Frustum Culling only disables the renderers for objects that are outside the camera's viewing area but does not disable anything hidden from view by overdraw. Note that when you use Occlusion Culling you will still benefit from Frustum Culling.
The occlusion culling process will go through the scene using a virtual camera to build a hierarchy of potentially visible sets of objects. This data is used at runtime by each camera to identify what is visible and what is not. Equipped with this information, Unity will ensure only visible objects get sent to be rendered. This reduces the number of draw calls and increases the performance of the game.

The data for occlusion culling is composed of cells. Each cell is a subdivision of the entire bounding volume of the scene. More specifically the cells form a binary tree. Occlusion Culling uses two trees, one for View Cells (Static Objects) and the other for Target Cells (Moving Objects).

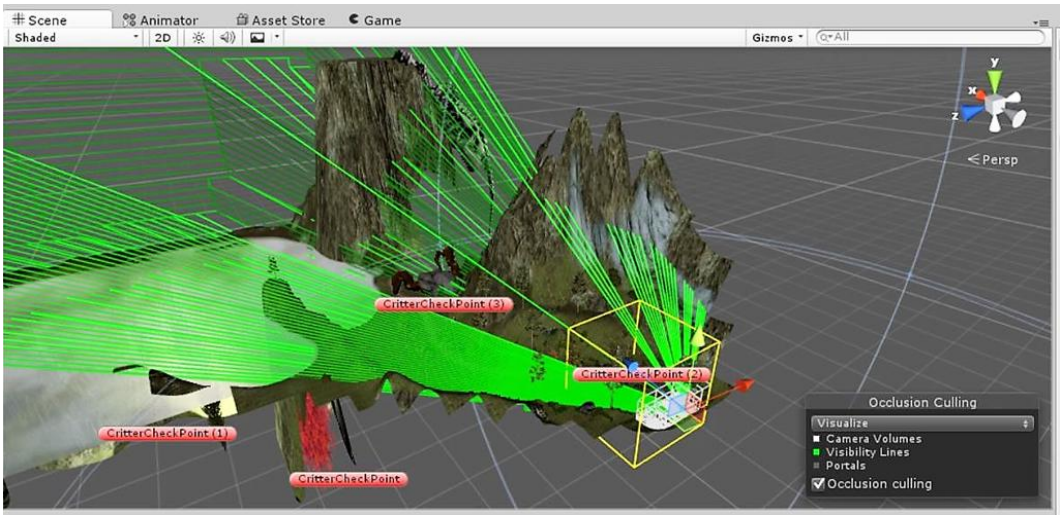
View Cells map to a list of indices that define the visible static objects which gives more accurate culling results for static objects.
It is important to keep this in mind when creating objects because a good balance is needed between the size of the objects and the size of the cells. Ideally, cells shouldn't be too small in comparison with your objects but equally there shouldn't be objects that cover many cells. Sometimes, culling can be improved by breaking large objects into smaller pieces. However, we can still merge small objects together to reduce draw calls and, as long as they all belong to the same cell, occlusion culling will not be affected.

In order to use Occlusion Culling, there is some manual setup involved. First, the level geometry must be broken into sensibly sized pieces. It is also helpful to lay out the levels into small, well defined areas that are occluded from each other by large objects such as walls, buildings, etc (this is why there is so much geometry in the scene of this project). The idea is that each individual mesh will be turned on or off based on the occlusion data. So if one object exists that contains all the objects in a room then either all or none of the entire set of objects will be culled.
This doesn't make nearly as much sense as making each object its own mesh, so each can individually be culled based on the camera's view point[6].

Any scene objects that we want to be part of the occlusion must be tagged as **Occluder Static**. The fastest way to do this is to multi-select the objects we want to be included in occlusion calculations, and mark them together.
Below (**Picture 1.71, 1.72 and 173**) are a visualization of **Occlusion** and **Frustum Culling** within the scene.

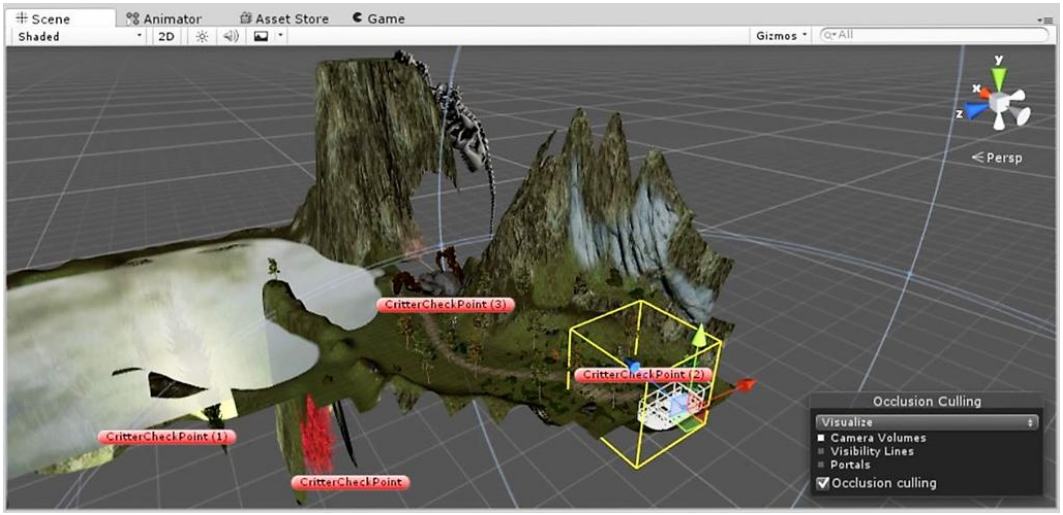
1.71 Simplified scene view, no Frustum or Occlusion Culling

Picture 1.72 is a visualization of **Occlusion** and **Frustum Culling** with the sight lines used to find obstructed objects.



1.72 Occlusion and Frustum Culling and sight lines

Picture 1.73 demonstrates Occlusion and Frustum Culling results. Notice the areas behind the mountain and in the lake that are not being rendered.



1.73 Occlusion and Frustum Culling

1.3.2 Terrain : Conclusion

This part of the project was, by far, the **hardest** and most **time consuming**. It can be quite a challenge for a single developer to understand and implement all the different parts that combine to form a complete, adequate digital experience.
Of all those parts, only the most important were mentioned in this document. Fully covering the game-design process would require thousands of pages and it would have been redundant, as many sites exist on the Internet that excel on doing just that using both pictures and video.
Below, the final game statistics and the hardware used to run it are documented.

Scene information:

Framerate: 55-**110fps**
Triangles: **3.8million**
Shadow casters: **268**
Animations: **14**

Hardware:

Intel Core i5-6600K @3.50GHz
16GB of RAM
AMD Radeon R9 390 with 8GB of dedicated memory
Razer OSVR Hacker Dev Kit v1.3(VR Headset)

2. The EEG

2.1 Understanding the Brain

The brain is a very complex system. The frontal cortex, the region where most of the conscious thoughts and decisions are made, conducts much less than a tenth of the total activity in the brain.

Planning, modeling of our surroundings, interpretation of sensory inputs up to and including our perception of reality, memory processing and storage and the basic drivers of our moods and emotions occur in many functional regions distributed around the brain, including the visual cortex at the rear, temporal cortex at the sides, parietal cortex behind the crown of the head and the limbic system deep inside the brain. The limbic system controls the basic moods and emotions, the fight/flight response and deeper long term memory encoding as well as control of basic bodily functions such as breathing and heartbeat.

Most of these deeper functions interact intimately with different parts of the cortex (the outer layer which is accessible to EEG measurements) however the interaction is quite complex and distributed. In order to map the true activity of the brain it is very important to measure signals from many different cortical structures located all around the brain surface. It is not possible to map these signals purely from the frontal and temporal regions. Determination of the user's complete mental state is very poorly approximated unless signals from the rear of the brain are also considered. With proper coverage and electrode configuration, it is possible to reconstruct a source model of all important brain regions and to see their interplay[9].

The **Epoc+ EEG** system (**Picture 2.2**) used in this experiment is a very accurate, professional device. Even compared to **Research-type EEG** equipment (costing $60,000), the data retrieved (waveforms) are similar (**Picture 2.1**).

EMOTIV currently provides **drivers** that measure 6 different emotional and sub-conscious dimensions in real time – **Excitement (Arousal), Interest (Valence), Stress (Frustration), Engagement/Boredom, Attention (Focus)** and **Meditation (Relaxation)**.

The detections above were developed based on rigorous experimental studies involving at least 20-30 volunteers for each state, where subjects were taken through experiences to elicit different levels of the desired state. They were wired up with many additional biometric measures (heart rate, respiration, blood pressure, blood volume flow, skin impedance and eye tracking), observed and recorded by a trained psychologist and also self-reported. EMOTIV Performance Metrics have been validated in many independent peer-reviewed studies.

The present experiment uses these categories (emotional states) to determine the effect of VR on the average **18 to 25-year-old human**. It, also, uses a similar test-group of 18 volunteers (20-30 were used by EMOTIV to develop the emotional states).
Furthermore, the results were separated by gender and gaming experience using a short, nameless survey each volunteer undertook.

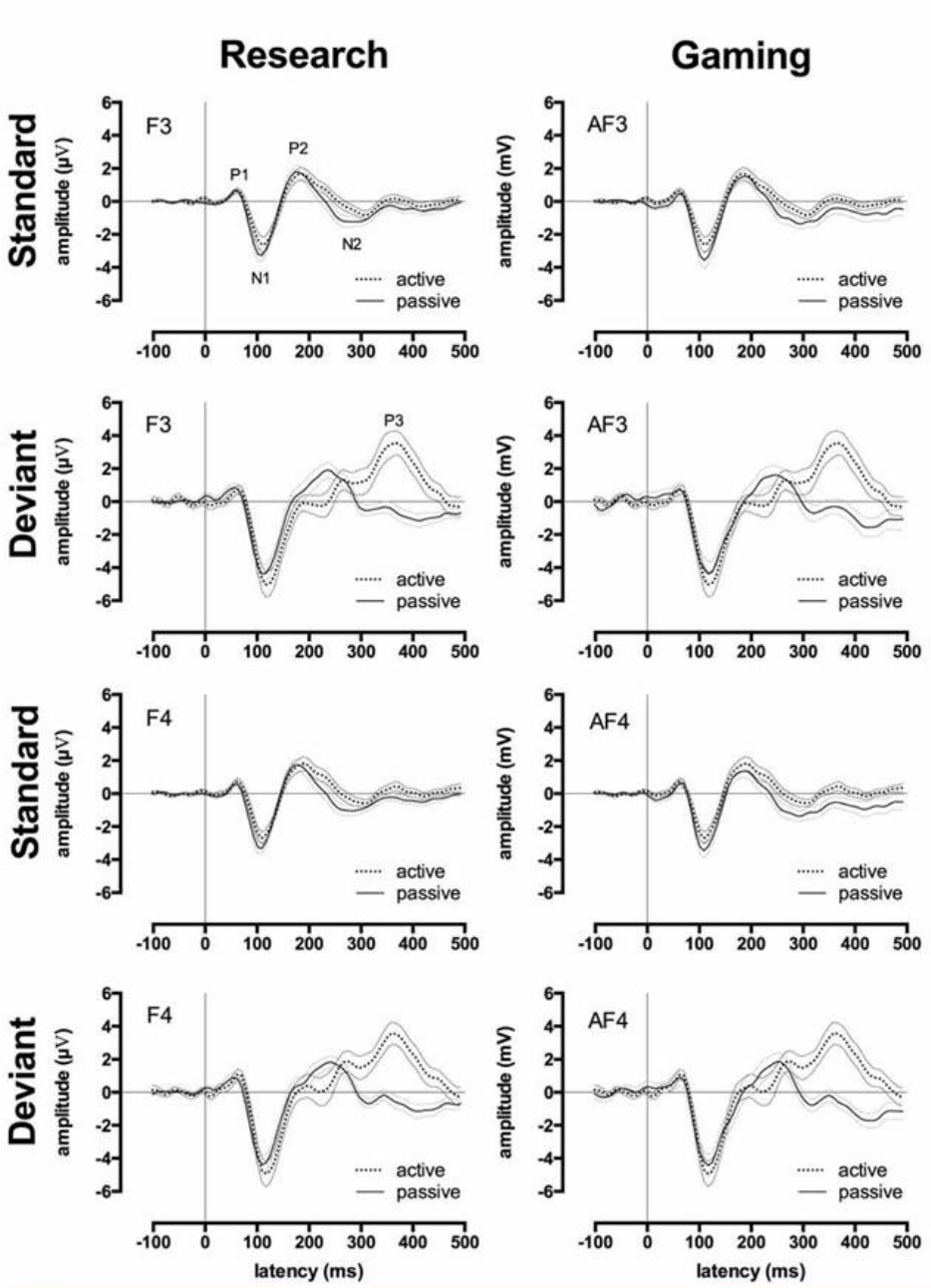


Figure 2 Research and gaming system ERP waveforms by condition, tone type, and hemisphere. Group ERP waveforms for research (left-side) and gaming (right-side) systems. All graphs display waveforms for the passive and active (counting deviant tones) listening conditions. The upper 4 graphs depict the left-hemisphere-activity (F3 and AF3) and the lower 4 graphs depict the right-hemisphere-activity (F4 and AF4). Rows 1 and 3 depict waveforms elicited by the standard tones, rows 2 and 4 depicts waveforms elicited by the deviant tones. Error waveforms (in grey) represent the standard error of the mean.

2.1 Epoc+ to Research-grade equipment comparison

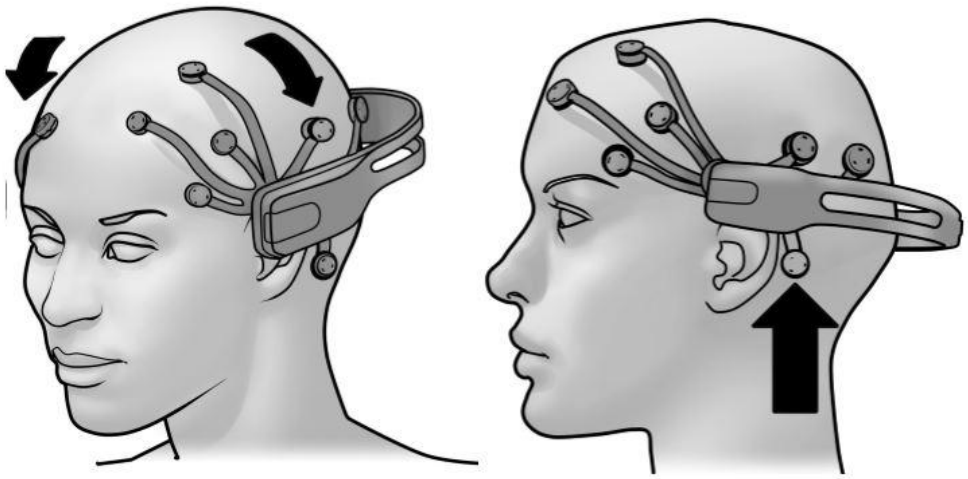
2.2 Epoc+ EEG

2.2 Testing Phase

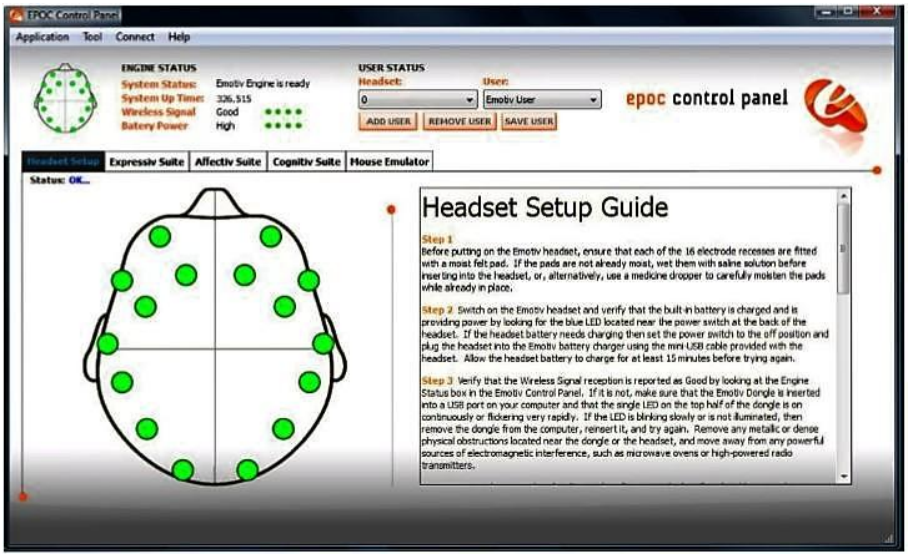
The tests were performed on **volunteers**, most of them interested in trying Virtual Reality for themselves. The subjects were 18-25 year olds of both genders, gamers and non-gamers. The testing was performed in an empty, air-conditioned room with most other stimuli apart from the people present in the room and the test itself removed. To that end, no cellphones or consumables were allowed and conversation was kept to a minimum during the actual recording of the data.

The test on every individual had **four distinct phases**. **Phase one** was consisted of **attuning** the EEG to the specifics of each individual. This was done by making a new profile for every volunteer, apply a solution to each sensor (as per the company's instructions) and having them properly wear the device, so that the sensors were on the exact spot on the head they were supposed to be (with every head and haircut being different, this was, at times, a challenge). Then, every sensor was micromanaged, to achieve maximum signal, which was presented as a green color in the setup panel. Yellow, Red and Black meant less than optimal, with Black being no signal at all (**Picture 2.3 and Picture 2.4**) [9].

When every sensor is properly setup, the system is calibrated to the specifics of that person by a series of automatic recordings as they have their eyes opened and closed.



2.3 Correct sensor placement



2.4 Green: Strong signal received by sensor

Phase two consists of a three-minute brain data recording, while the volunteer is asked to stay seated and avoid any unnecessary action. It was quickly observed that friendly, normal conversation had better results in keeping someone still for 3 minutes, so that was used with every tester.
These recording served a double purpose. First and foremost, they served as a 3-minute time window for the EEG to specialize the data to the individual. And then, they also served as a reference to understand the consistency and quality of the useful data obtained later. An example of quality control would be having a Stress value of a 100% during a simple chat, which meant the sensor was not working properly. A **report** was generated after this session and the data was saved locally and on the EMOTIVs cloud service.

Phase three, the first to produce an important set of data, consisted of 3 minutes of the volunteer playing an on-line First-Person Shooter game, mostly alone, with very few enemies occasionally encountered. They were asked to roam around, doing anything they wanted for the duration. While the game will remain nameless, in accordance with its Terms of Use, it is a typical, first person shooter game, with 3D graphics and sound, played through an average PC screen. **Picture 2.5** is another, incredibly similar game.
The reasons behind choosing it were its perspective (first person), financial model (free to play) and platform(it could be played on a browser). A report was generated and stored with the emotional-brainwave data from this experience.

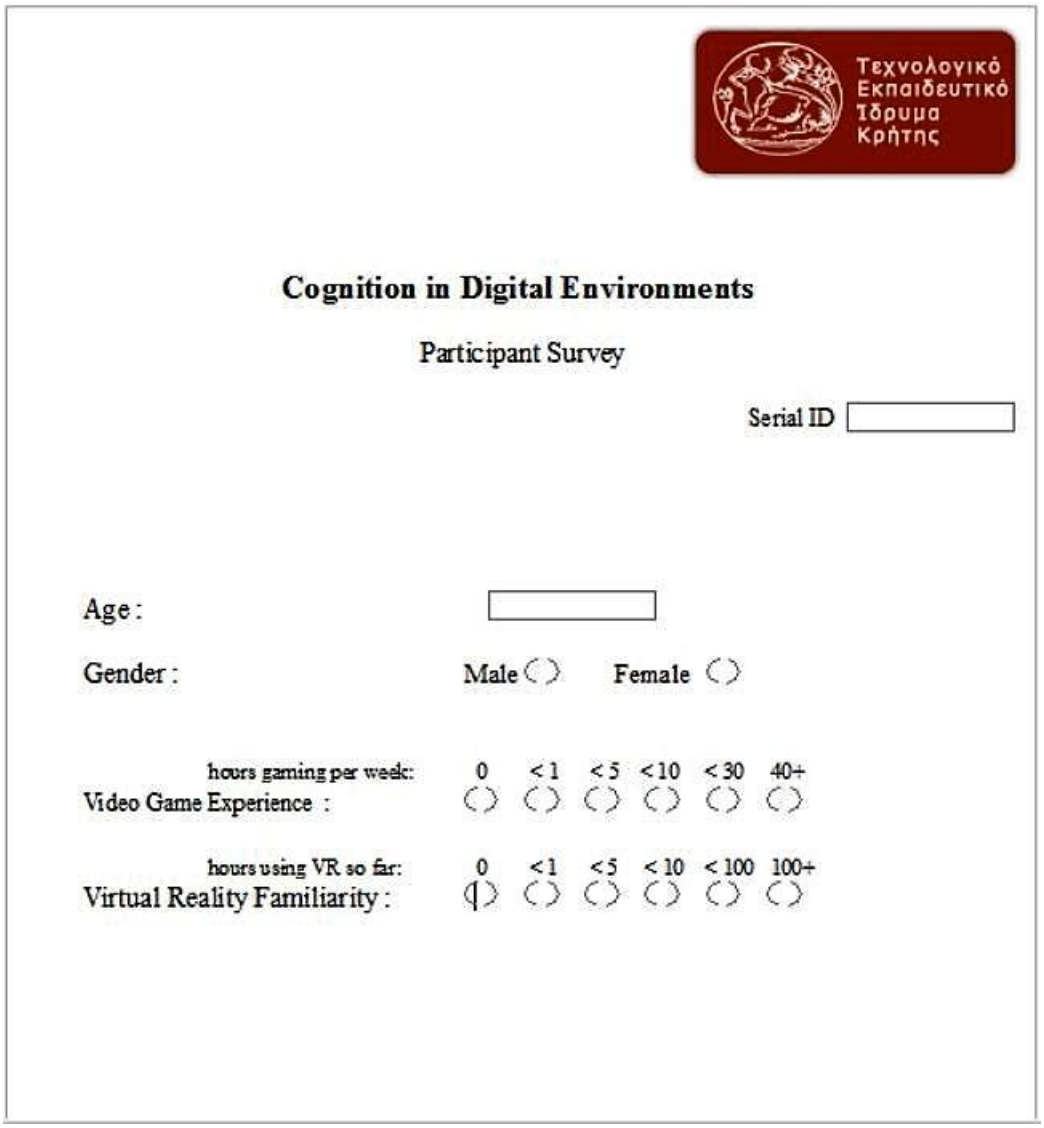
2.5 First person shooter example

The **final phase** consisted of three minutes of using the custom digital VR environment through the OSVR headset. Out of those 18 people, the first 9 had the headset correctly mounted on their head with the built-in straps(although the experience, due to already wearing the EEG, was unpleasant) and the rest kept the headset correctly placed on their head with their left hand, while navigating the scene with their right using a joystick.

The reasoning behind this was that the first group had everything correctly placed on them as specified in both the manuals of the EEG and OSVR, but felt discomfort while the other group felt comfortable but at the cost of some immersion.

In the final calculations, the results were calculated both for one group of 18 people and two groups of 9. The data generated by this phase was, of course, stored.

Then, the volunteer was given a short survey (**Picture 2.6**) to fill. This helped refine the data even further. Names were not used and the survey was connected to the data with a serial number.



Τεχνολογικό Εκπαιδευτικό Ίδρυμα Κρήτης

Cognition in Digital Environments

Participant Survey

Serial ID

Age :

Gender : Male ( ) Female ( )

	0	< 1	< 5	< 10	< 30	40+
hours gaming per week: Video Game Experience :	( )	( )	( )	( )	( )	( )

	0	< 1	< 5	< 10	< 100	100+
hours using VR so far: Virtual Reality Familiarity :	( )	( )	( )	( )	( )	( )

2.6 Survey

3. Results

3.1 Data Documentation

For every **ID** the first row is the data **(%)** recorded while having a **calm discussion**, the second while **playing the fps game** and the third while **experiencing the environment using VR**.

Table 1:

Group Numb.	ID	Engagement	Excitement	Interest	Relaxation	Stress	Focus
1	g00001	54	22	56	33	47	37
1	g00001	55	17	50	33	36	29
1	g00001	55	53	57	33	31	53
1	g00002	55	46	71	33	73	58
1	g00002	55	24	51	33	46	37
1	g00002	59	50	63	33	55	59
1	g00003	57	60	59	34	47	61
1	g00003	62	48	53	31	34	51
1	g00003	68	33	56	32	39	37
1	g00004	66	40	53	33	48	46
1	g00004	72	18	54	30	40	31
1	g00004	77	17	51	28	35	33
1	g00005	58	26	58	38	52	39
1	g00005	62	16	56	40	46	29
1	g00005	60	43	62	43	54	48
1	g00006	71	3	58	8	62	42
1	g00006	70	40	50	31	41	45
1	g00006	64	53	61	35	39	52
1	g00007	79	66	56	33	100	81
1	g00007	66	67	47	30	100	83
1	g00007	58	84	58	36	100	89
1	g00008	70	25	48	33	42	34

1	g00008	70	23	52	30	42	36
1	g00008	55	47	64	33	63	56
1	g00009	66	28	68	34	46	47
1	g00009	65	18	64	33	56	37
1	g00009	58	23	65	33	55	41
2	g00010	59	26	69	33	100	50
2	g00010	73	17	48	31	100	52
2	g00010	62	35	58	36	60	45
2	g00011	66	41	60	35	100	60
2	g00011	69	23	52	30	100	57
2	g00011	67	49	66	46	100	62
2	g00012	52	42	62	32	100	62
2	g00012	55	57	54	31	73	76
2	g00012	55	77	75	33	83	77
2	g00013	60	23	60	38	63	41
2	g00013	58	20	55	31	39	34
2	g00013	65	25	59	28	59	46
2	g00014	62	41	52	32	52	51
2	g00014	59	59	55	30	42	59
2	g00014	60	72	67	36	61	73
2	g00015	64	59	75	36	94	71
2	g00015	75	23	54	27	56	42
2	g00015	65	53	67	34	65	65
2	g00016	56	43	57	33	44	50
2	g00016	55	6	54	33	42	26
2	g00016	59	64	57	36	49	63
2	g00017	62	38	64	34	100	58
2	g00017	64	32	54	33	100	59
2	g00017	55	71	82	33	100	80
2	g00018	62	58	55	44	70	62
2	g00018	64	25	52	40	50	38
2	g00018	75	54	56	34	65	63

3.2 Summary Statistics

The easiest way to mathematically picture the differences and to present the results are Summary Statistics. For this analysis, the **mean** (the sum of a collection of numbers divided by the number of values in the collection), **median (**the value separating the higher part of a data sample from the lower part**)** and **mode(the value that appears most often in a set of data)** will be calculated. The first two for the actual representation and the last one for quality control.

First, we calculate the difference between the values of "Calm" and "VR", then "FPS" and "VR", or the 1-3 and 2-3 rows of each emotional state. We do that for all 18 IDs. Then the **mean** and the **median** will be calculated between all the corresponding values(differences) of all the IDs. We do this for every emotional state.

Then, we calculate the mode. If the mode contains the 0 value, it means that there were many readings without any difference from each other. An example would be Stress being 100 for all 3 readings of a volunteer. The logical assumption is that the reading was false, an outlier and that we need to remove the problematic reading.

3.2.1 Single 18-person group

Below are the results, without removing any outliers, of all the IDs pooled together. Numbers are the differences %. **Negative** values mean a **decrease** in the emotional state.

Table 2:

ENGAGEMENT	
Mean Calm-VR	**-0.111**
Mean FPS - VR	**-1.777**
Median Calm-VR	**1.500**
Median FPS-VR	**-1**
Mode Calm-VR	**1 , 3**
Mode FPS-VR	**0, 4, -2**
EXCITEMENT	
Mean Calm-VR	**12**
Mean FPS - VR	**20.555**
Median Calm-VR	**13**
Median FPS-VR	**22**
Mode Calm-VR	**31**

Mode FPS-VR	**26 , 13 , 5**
INTEREST	
Mean Calm-VR	**2.055**
Mean FPS - VR	**9.853**
Median Calm-VR	**0.500**
Median FPS-VR	**10.500**
Mode Calm-VR	**-3 , -1**
Mode FPS-VR	**12**
RELAXATION	
Mean Calm-VR	**1.277**
Mean FPS - VR	**2.529**
Median Calm-VR	**0**
Median FPS-VR	**3**
Mode Calm-VR	**0 , 3**
Mode FPS-VR	**0**
STRESS	
Mean Calm-VR	**-6.166**
Mean FPS - VR	**3.444**
Median Calm-VR	**-2**
Median FPS-VR	**6**
Mode Calm-VR	**0**
Mode FPS-VR	**0**
FOCUS	
Mean Calm-VR	**5.055**
Mean FPS - VR	**12.333**
Median Calm-VR	**6.500**
Median FPS-VR	**13**
Mode Calm-VR	**22**
Mode FPS-VR	**ALL**

3.2.2 Single 18-Person Group, Outliers Removed

As we can easily observe in the data above, the value 0 keeps appearing in mode. We will remove all outliers, calculate the differences again and follow the exact same procedure to re-calculate mean-median-mode for the new set of data.
Below are the results, with most changes being on Engagement, Relaxation and Stress.

Table 3:

ENGAGEMENT	
Mean Calm-VR	**0.562**
Mean FPS - VR	**-1.5**
Median Calm-VR	**1.500**
Median FPS-VR	**-2**
Mode Calm-VR	**1 , 3**
Mode FPS-VR	**-2**
EXCITEMENT	
Mean Calm-VR	**12**
Mean FPS - VR	**20.555**
Median Calm-VR	**13**
Median FPS-VR	**22**
Mode Calm-VR	**31**
Mode FPS-VR	**26 , 13 , 5**
INTEREST	
Mean Calm-VR	**2.055**
Mean FPS - VR	**9.853**
Median Calm-VR	**0.500**
Median FPS-VR	**10.500**
Mode Calm-VR	**-3 , -1**
Mode FPS-VR	**12**
RELAXATION	
Mean Calm-VR	**1.533**
Mean FPS - VR	**3**

Median Calm-VR	**1**
Median FPS-VR	**3**
Mode Calm-VR	**3**
Mode FPS-VR	**3 , 4 , 6**
STRESS	
Mean Calm-VR	**-7.583**
Mean FPS - VR	**6.230**
Median Calm-VR	**-6.500**
Median FPS-VR	**8**
Mode Calm-VR	**9**
Mode FPS-VR	**-5 , 9**
FOCUS	
Mean Calm-VR	**5.055**
Mean FPS - VR	**12.333**
Median Calm-VR	**6.500**
Median FPS-VR	**13**
Mode Calm-VR	**22**
Mode FPS-VR	**ALL**

3.2.3 Two 9-Person groups, Outliers Removed

Due to the difference in the way the **VR headset** was **worn** and the **discomfort**, the group can be split into two **pools of data**.
Table 4 contains the results. There is an additional metric, the total **MIN** and **MAX** values calculated per-group and per-emotional state.

Table 4:

GROUP 1	**-**
ENGAGEMENT	
Mean Calm-VR	**-3.25**
Mean FPS - VR	**-2**
Median Calm-VR	**-3**
Median FPS-VR	**-4**
Mode Calm-VR	**11**
Mode FPS-VR	**ALL**
Min/Max Calm-VR	**-21/11**
Min/Max FPS-VR	**-8/6**
EXCITEMENT	
Mean Calm-VR	**9.666**
Mean FPS - VR	**14.666**
Median Calm-VR	**17**
Median FPS-VR	**17**
Mode Calm-VR	**ALL**
Mode FPS-VR	**ALL**
Min/Max Calm-VR	**-27/50**
Min/Max FPS-VR	**-15/36**
INTEREST	
Mean Calm-VR	**2.111**
Mean FPS - VR	**6.555**
Median Calm-VR	**1**
Median FPS-VR	**7**
Mode Calm-VR	**-3**

Mode FPS-VR	**11 , 12**
Min/Max Calm-VR	**-8/12**
Min/Max FPS-VR	**-1/12**
RELAXATION	
Mean Calm-VR	**4.500**
Mean FPS - VR	**2.500**
Median Calm-VR	**1**
Median FPS-VR	**3**
Mode Calm-VR	**ALL**
Mode FPS-VR	**3**
Min/Max Calm-VR	**-5/27**
Min/Max FPS-VR	**-2/6**
STRESS	
Mean Calm-VR	**-9.571**
Mean FPS - VR	**0.142**
Median Calm-VR	**-13**
Median FPS-VR	**1**
Mode Calm-VR	**ALL**
Mode FPS-VR	**-5**
Min/Max Calm-VR	**-23/9**
Min/Max FPS-VR	**-12/9**
FOCUS	
Mean Calm-VR	**2.888**
Mean FPS - VR	**10.111**
Median Calm-VR	**8**
Median FPS-VR	**7**
Mode Calm-VR	**ALL**
Mode FPS-VR	**ALL**
Min/Max Calm-VR	**-23/22**
Min/Max FPS-VR	**-13/24**

Table 5:

GROUP 2	-
ENGAGEMENT	
Mean Calm-VR	2.222
Mean FPS - VR	-1
Median Calm-VR	3
Median FPS-VR	0
Mode Calm-VR	3
Mode FPS-VR	ALL
Min/Max Calm-VR	-7/13
Min/Max FPS-VR	-11/11
EXCITEMENT	
Mean Calm-VR	14.333
Mean FPS - VR	26.444
Median Calm-VR	9
Median FPS-VR	26
Mode Calm-VR	ALL
Mode FPS-VR	ALL
Min/Max Calm-VR	-6/35
Min/Max FPS-VR	5/58
INTEREST	
Mean Calm-VR	3
Mean FPS - VR	12.111
Median Calm-VR	0
Median FPS-VR	12
Mode Calm-VR	-1
Mode FPS-VR	4
Min/Max Calm-VR	-12/18
Min/Max FPS-VR	3/28
RELAXATION	
Mean Calm-VR	0
Mean FPS - VR	3
Median Calm-VR	0
Median FPS-VR	3
Mode Calm-VR	11 , -10

Mode FPS-VR	**ALL**
Min/Max Calm-VR	**-10/11**
Min/Max FPS-VR	**-6/16**
STRESS	
Mean Calm-VR	**-4.800**
Mean FPS - VR	**13.333**
Median Calm-VR	**-4**
Median FPS-VR	**12.500**
Mode Calm-VR	**ALL**
Mode FPS-VR	**ALL**
Min/Max Calm-VR	**-29/9**
Min/Max FPS-VR	**7/20**
FOCUS	
Mean Calm-VR	**7.666**
Mean FPS - VR	**14.555**
Median Calm-VR	**5**
Median FPS-VR	**14**
Mode Calm-VR	**22**
Mode FPS-VR	**ALL**
Min/Max Calm-VR	**-6/22**
Min/Max FPS-VR	**-7/37**

3.3 Interpreting the Results

There are large differences in the data between the full and no-outlier group. There are even bigger differences between the single and the double data pool approaches.
In this paper, while all the data discovered were documented for clarity and future use, only the no-outlier, two-group approach will be fully explained, although the differences to each approach will be mentioned.

Using the data from the surveys, we can define the **“average person”** as **a 22 year old male**

with a gaming experience of 21.8 hours per week.

Furthermore, of all the different metric-comparisons, the results of the female testers against the averages of the groups they belonged to presented the most interest, so they were included. Since a similar male-to-average comparison would be redundant (due to the average leaning decisively to the male side) it was omitted.

Engagement

Engagement[11] is defined as emotional involvement or commitment. Results indicate small overall changes in engagement between the different phases.

Engagement results**:**

-	-	ALL	No Outliers	Group 1	Group 2
Calm – VR	**mean**	**-0.111**	**0.562**	**-3.250**	**2.222**
FPS – VR	**mean**	**-1.777**	**-1.500**	**-2.000**	**-1.000**
Calm – VR	**median**	**1.500**	**1.500**	**-3.000**	**3.000**
FPS – VR	**median**	**-1.000**	**-2.000**	**-4.000**	**0.000**

Female deviation from the average **mean (negative values are lower than average)**:

-	Group 1	Group 2
Calm – VR	**-11.25**	**5.778**
FPS – VR	**-5.500**	**6.500**

There are **minor differences** in Engagement from phase to phase, with the mean values getting **closer to the medians** after we remove the outliers meaning our results get more robust. The above results are interpreted as Engagement slightly decreasing when feeling slight discomfort (Group 1),

but, overall, remaining the same when using VR as opposed to more traditional means.
It's noteworthy, though, that **female testers demonstrated more than twice the engagement increase of the average, which dropped 2-3 times lower than the average when in slight discomfort.**

Excitement

Excitement is a feeling of eager enthusiasm and interest. This is the emotional state with the most fascinating, definite difference in VR. Female deviation from the average is large for this emotional state too.

Excitement results**:**

-	-	ALL	No Outliers	Group 1	Group 2
Calm – VR	**mean**	**12.000**	**12.000**	**9.666**	**14.333**
FPS – VR	**mean**	**20.555**	**20.555**	**14.666**	**26.444**
Calm – VR	**median**	**13.000**	**13.000**	**17.000**	**9.000**
FPS – VR	**median**	**22.000**	**22.000**	**17.000**	**26.000**

Female deviation from the average **mean:**

-	Group 1	Group 2
Calm – VR	**-7.833**	**5.834**
FPS – VR	**-15.444**	**9.834**

The testers exhibited very big differences in this emotional state when experiencing a world through VR as opposed to more traditional means. A **similar**(26.5 to 26%) **mean and median** demonstrate that no big outliers exist in the data of Group 2, so that result is **the most robust.**
An increase of 5-12% can be noticed between the group feeling slight discomfort(Group 1) and the other, comfortable group (Group 2).
Furthermore, **with the minimum increase of Group 2 at 5% and maximum at an incredible 58%, this was the emotional state influenced most by VR.**

Female deviation was also high regarding Excitement, with an increase of 5.8-9.8% when comfortable and decrease of 7.8 to 15.4% when in slight discomfort.

Interest

Interest is a quality that attracts your attention and makes you want to learn more about something or to be involved in something. This was the most robust set of data, with no outliers detected and mean values being, generally, close to the median.

Interest results**:**

-	-	ALL	No Outliers	Group 1	Group 2
Calm – VR	**mean**	**2.055**	**2.055**	**2.111**	**3.000**
FPS – VR	**mean**	**9.853**	**9.853**	**6.555**	**12.111**

Calm – VR	**median**	**0.500**	**0.500**	**1.000**	**0.000**
FPS – VR	**median**	**10.500**	**10.500**	**7.000**	**12.000**

Female deviation from the average **mean:**

-	**Group 1**	**Group 2**
Calm – VR	**-2.611**	**3.000**
FPS – VR	**-0.515**	**2.389**

The same motif appears in this emotional state as well. Group 2 exhibits a substantial increase when using VR, especially when compared to playing a game through classic means., while group 2 demonstrates a much smaller increase.
Gender does not affect Interest as much, with minor (2-3%) differences exhibited.

Relaxation

Relaxation is something that stops someone from being nervous or worried. VR increased relaxation for all genders in a minor way. This state had the most outliers, possibly due to the relevant sensors being easier to move out of place.

Relaxation results**:**

-	**-**	**ALL**	**No Outliers**	**Group 1**	**Group 2**
Calm – VR	**mean**	**1.277**	**1.533**	**4.500**	**0.000**
FPS – VR	**mean**	**2.529**	**3.000**	**2.500**	**3.000**
Calm – VR	**median**	**0.000**	**1.000**	**1.000**	**0.000**
FPS – VR	**median**	**3.000**	**3.000**	**3.000**	**3.000**

Female deviation from the average **mean:**

-	**Group 1**	**Group 2**
Calm – VR	**-3.500**	**-2.000**
FPS – VR	**0.500**	**5.500**

Small differences for this emotional state in combination with many outliers and a relatively large difference between mean and median render these results uninteresting.

Stress

Stress is a state of mental tension and worry. There are big differences here from group to group and among readings. This was probably caused by either the lenses of the VR not being properly adjusted to each individual (something they could only do and notice themselves) which caused nausea or a conductivity-sensor problem in the EEG that sometimes caused stress to spike at 100%, an impossibility.

Stress results**:**

-	-	ALL	No Outliers	Group 1	Group 2
Calm – VR	**mean**	**-6.166**	**-7.583**	**-9.571**	**-4.800**
FPS – VR	**mean**	**3.441**	**6.230**	**0.742**	**13.333**
Calm – VR	**median**	**-2.000**	**-6.500**	**-13.000**	**-4.000**
FPS – VR	**median**	**6.000**	**8.000**	**1.000**	**12.500**

Female deviation from the average **mean:**

-	Group 1	Group 2
Calm – VR	**-14.071**	**-2.300**
FPS – VR	**0.358**	**-0.833**

Focus

Focus is defined as a center of activity, attraction, or attention. Focus had the second biggest increases of all the emotional states and was largely indifferent to genders.

Focus results**:**

-	-	ALL	No Outliers	Group 1	Group 2
Calm – VR	**mean**	**5.055**	**5.055**	**2.888**	**7.666**
FPS – VR	**mean**	**12.333**	**12.333**	**10.111**	**14.555**
Calm – VR	**median**	**6.500**	**6.500**	**8.000**	**5.000**
FPS – VR	**median**	**13.000**	**13.000**	**7.000**	**14.000**

Female deviation from the average **mean:**

-	Group 1	Group 2
Calm – VR	**-1.888**	**0.334**
FPS – VR	**-5.111**	**-1.555**

The above can be interpreted as a tendency of humans to focus more on a digital environment when using VR. The fact that eyesight is devoted to the digital world when experienced through VR may be the explanation to this. Gender plays a minor, insignificant role on Focus.

4. References

[1]: Wikipedia
[2]: Unity Physics
[3]: Unity Character Controller
[4]: Unity Sound
[5]: Unity Lighting
[6]: Unity Post-Processing Effects
[7]: Techtarget
[8]: OSVR
[9]: Emotiv EPOC+ Manual
[10]: Wikipedia: VR Motion Sickness
[11]: Merrian-Webster