Miro Vesterinen

3D Game Environment in Unreal Engine 4

Thesis
Kajaani University of Applied Sciences
Bachelor of Business Administration
Business Information Technology programme
2014

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittelyn koulutusohjelma
Tekijä(t) Miro Vesterinen	
Työn nimi 3D Game Environment in Unreal Engine 4	
Vaihtoehtoiset ammattiopinnot Peligrafiikka	Toimeksiantaja
Aika 9.12.2014	Sivumäärä ja liitteet 43

Tämä opinnäytetyö käsittelee korkealaatuisen 3D-sisällön tuottamiseen vaadittua teknistä osaamista Unreal Engine 4 -pelimoottorissa. Opinnäytetyöllä ei ollut tilaajaa, vaan se tehtiin täysin tekijän oman oppimisen tukemiseksi.

Unreal Engine on pelimoottori, jolla voi tuottaa pelejä monille eri alustoille. Uusien pelinkehitystyökalujen ja -tapojen kehittyessä on 3D-graafikolle elintärkeää opiskella ja mukautua tässä jatkuvasti muuttuvassa alassa. Tämän opinnäytetyön teoreettinen painopiste on 3D-sisällön tuottamisessa sekä erinäisten ohjelmistojen ja työtapojen esittelyssä. Teoriaosuus on kirjoitettu 3D-graafikkoa ajatellen.

Tämä opinnäytetyö sisältää myös esittelyn 3D-peliympäristöstä. Peliympäristön sisältö on luotu käyttäen Autodesk 3ds Max 2015 mallinnusohjelmistoa ja teksturointi tapahtui pääsääntöisesti Adobe Photoshopissa.

Peliympäristön luominen alustavasta suunnittelusta lopulliseen tuotokseen vaatii aikaa ja kattavaa teknistä osaamista sekä tarvittavien ohjelmistojen tuntemista. Hyvin koostettu tuotantolinja helpottaa työntekoa ja säästää aikaa, pelinkehityksen kalleinta resurssia.

Kieli	Englanti
Asiasanat	Unreal Engine, 3D-ympäristö, 3D-grafiikka
Säilytyspaikka	☒ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Miro Vesterinen	
Title 3D Game Environment in Unreal Engine 4	
Optional Professional Studies Game Graphics	Commissioned by
Date 9.12.2014	Total Number of Pages and Appendices 43

This thesis deals with the technical knowledge required from a 3D-artist to produce high-quality content using Unreal Engine 4 and was made only to support the writer's own learning and know-how of the subject.

Unreal Engine is a toolkit for developers from AAA to indie that can be used to produce games on multiple platforms. And as new development workflows and software are released, it is vital for artists to study and adapt in this constantly changing industry. The focus of the thesis is on the theoretical knowledge of content creation, as well as on introducing different software and workflows that can be used. The theory part will not feature any complex mathematics and is written for 3D-artists.

This thesis also includes a breakdown of a practical 3D environment project, a futuristic environment that was developed during a time frame of two months by me and Tuukka Mäkinen. The content for the environment was created using Autodesk 3ds Max and Adobe Photoshop.

Creating a 3D-game environment, from initial concept to final product, requires both time and technical knowledge. A proper planning combined with comprehensive technical knowledge can shrink both production time and load.

Language of Thesis	English
Keywords	Unreal Engine, 3D environment, 3D graphics
Deposited at	☒ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

TABLE OF CONTENT

SYMBOLS

Projection mesh	Copy of the low-poly mesh used in normal map baking. Used to determine the distance rays are cast from during baking
Resolution	Number of pixels in each dimension of a digital image.
RGB bitmap	Image file that contain red, green and blue color channels
Smoothing split	Used to split vertex normal. Often used to add definition to a low-poly mesh
Shader	A program that determines how an object is drawn with given parameters
Texel	One pixel in a texture map
Mipmapping	Optimization method that determines the texture resolution based on the distance from camera
Texture Streaming	Priority-based system for loading and unloading texture maps

1 INTRODUCTION

This thesis is to discuss the technical knowledge and theory required from a 3D-artist to create content for Unreal Engine 4, as well as to introduce some of the workflows and software that are used in the games industry. Game development technologies are constantly changing and it is vital for a 3D-artist to stay updated and to learn new workflows. The thesis will tackle the subject from a point of view of a 3D-environment artist, yet most of content can be used for general content creation. This thesis will not review the actual workflows of creating 3D-meshes, but discusses the subject at a theoretical level. This thesis will also briefly discuss a practical case project, a futuristic office environment, developed within a time frame of two months by me and Tuukka Mäkinen. The focus of this thesis will be mainly on pc and console game development as all of the features presented in this document are not available on mobile.

During the time of writing this thesis the latest version of Unreal Engine 4 is 4.5.1. Some of the content within this thesis may not be accurate in other versions of Unreal Engine.

2 3D-MODELING

The field of 3D-graphics is rapidly expanding and currently 3D-modeling is used by a wide range of industries from architectural visualization to healthcare. This thesis will focus on the terms and concepts of 3D-modeling that are used within the games industry. 3D-modeling is a process of developing a mathematical representation of any three-dimensional surface of an object. A 3D-space is described in terms of a 3D-cartesian coordinates, three axis lines representing height, width, and depth that intersect at the origin. The product of a polygonal 3D-modeling is a mesh, a collection of vertices, edges and polygons that define the shape of the object. (Digital-Tutors, 2014.)

- Vertex - vertices are the smallest components of a 3D-mesh, a point in 3D-space that uses XYZ coordinates to determine its location.
- Edge - as two vertices are connected by a line an edge is formed
- Polygon - a closed set of edges that form a visible plane, called a polygon. Polygons are often categorized as tris, quads or ngons depending on the number of sides it has. (Digital-Tutors, 2014.)

As these components are connected together they create a flow around the 3D-mesh, this flow is called topology. In most applications polygons are one-sided and because of this every 3D-surface has a normal, a vector that determines the direction the polygon is facing. These 3D-meshes can be authored in a 3D-modeling package, such as Autodesk's 3ds Max or Foundry's Modo. These packages provide a wide variety of tools for creating and editing 3D-meshes, and in most cases animating and rendering as well. The process of generating a 2D-image from a 3D-mesh is called rendering. For games to be interactive, the rendering needs to happen multiple times per second during run-time, this is called real-time rendering, and the rate at which the images are displayed is measured in frames per second. For the interaction in a game to happen seamlessly the rate should be at least 30 frames per seconds. (Polycount, 2014b.)

2.1 Low-poly and high-poly modeling

When modeling for games meshes are often referred as low-poly and high-poly, these terms serve both a technical, as well as a descriptive term, even though there is no actual numerical amount of polygons that determine when a mesh is low- or high-poly.

Low-poly meshes are designed to be used in a real-time application, such as a game engine. Generally these meshes have a relatively low amount of polygons and feature just the main shapes of the object, but as hardware has advanced and the computing power has increased, the amount of detail in low-poly meshes has increased as well. For example, modern character meshes are constructed out of tens of thousands of polygons compared to just a few thousand polygons seen at the beginning of the millennia. A game environment can total up to a millions of polygons, but this does not mean that polygons should be used carelessly as efficiency is still needed for optimal results. (Crytek, 2014a.) In comparison just a single high-poly mesh can be constructed of millions of polygons and feature extreme amount of detail, this is why high-poly meshes are more commonly used in animated films or in-game cutscenes that are rendered offline. Yet high-poly meshes have their purpose in game development as well, as they are often created to have certain data extracted from them. The visual difference of low-poly and high-poly model can clearly be seen in Figure 1. (Crytek, 2014b.)

b

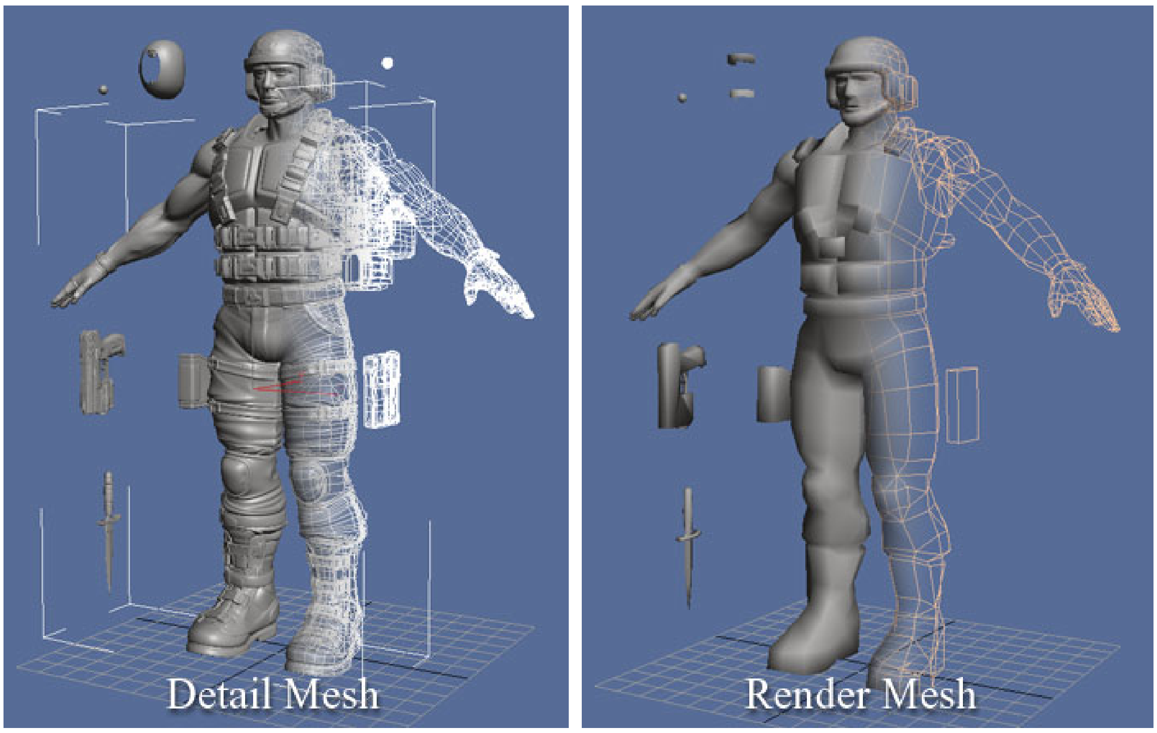

Figure 1. A high-poly mesh and a low-poly mesh. (Epic, 2012a.)

2.1.1 Digital sculpting

3D-meshes can also be digitally sculpted in a specialized sculpting software. Sculpting is a newer workflow that allows the 3D-mesh to be created with tools that resemble tradition sculpting. By using a real-world sculpting techniques to edit a virtual clay, digital sculpting tools give the ability to create highly detailed 3D-meshes. Generally digital sculpting has been used to create organic high-poly meshes, such as characters, but as the tools have matured hard-surface objects can also be created. (Digital-Tutors, 2014.)

Zbrush by Pixologic, seen in Figure 2, is a digital sculpting and painting software that provides tools for creating realistic organic and hard-surface models and supports up to a millions of polygons. Zbrush provides a workflow that is less technical as it resembles traditional sculpting and this is why it has gained a strong foothold in games industry. (Pixologic, 2014.)

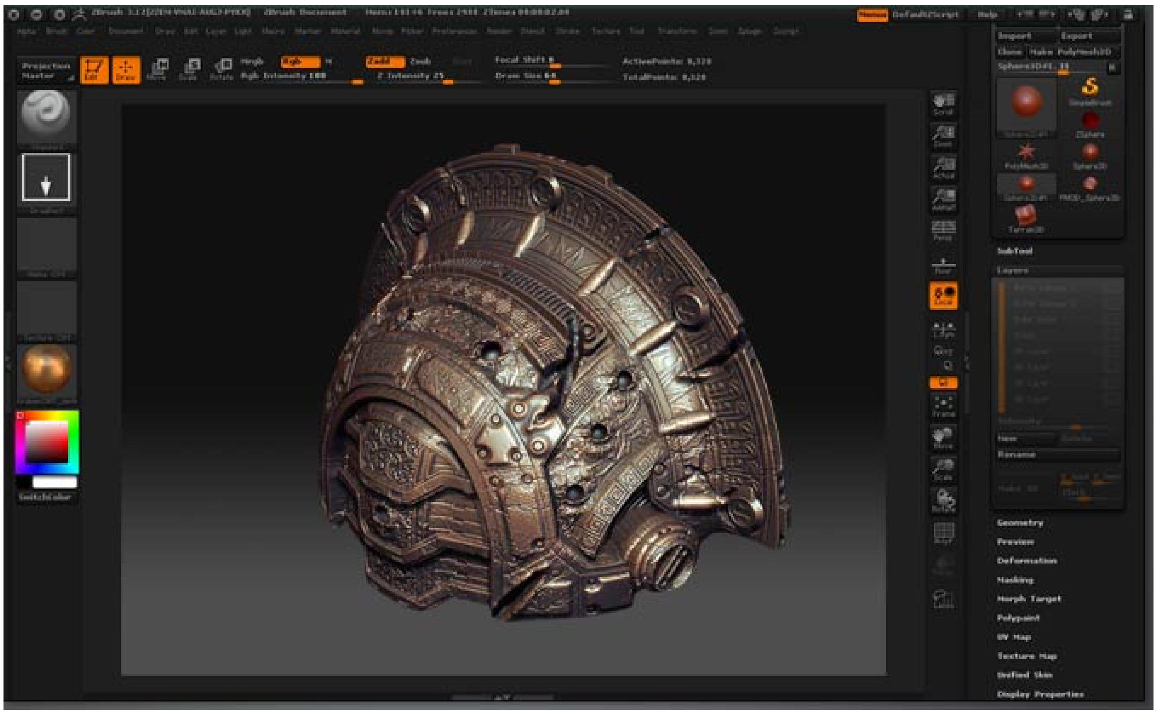

Figure 2. Zbrush digital sculpting software (Pixologic, 2014.)

2.1.2 Special cases

Generally 3D-packages provide tools for modeling, animation, rendering and so on, but some 3D-modeling software are far better suited, or created just for, a certain type of mesh creation. In some cases 3D-models can even be scanned from real-world objects to provide accurate data. Below a quick introduction to a few widely used software examples that provide a toolset for a very specific mesh creation.

IDV's SpeedTree is a toolkit used to create 3D animated plants and trees for games. SpeedTree has been used in the games industry since 2002 and has been featured in multiple commercially successful games from studios, such as Bungie and Volition. Speedtree provides integrated real-time vegetation creation tools for Unreal Engine 4, and is available for versions 4.3 and later. (IDV, 2014.)

Marvelous Designer, by CLO, is a software used to create 3D-clothing. With unique technology based on art of sewing and patternmaking, Marvelous designer produces more natural results than digital sculpting. Marvelous Designer provides seamless data transfer with various modeling packages, and thus multiple top game studios, such as Ubisoft and Konami, have already integrated Marvelous designer as part of their pipeline. (CLO Virtual Fashion, 2014.)

World Machine is a node-based landscape generator that combines procedural terrain creation with nature simulations to produce realistic looking terrains. World Machine makes it possible to create terrain detail that would be nearly impossible to achieve by hand, seen in Figure 3. Providing both height maps and masks for different materials. (World Machine Software, 2014.)

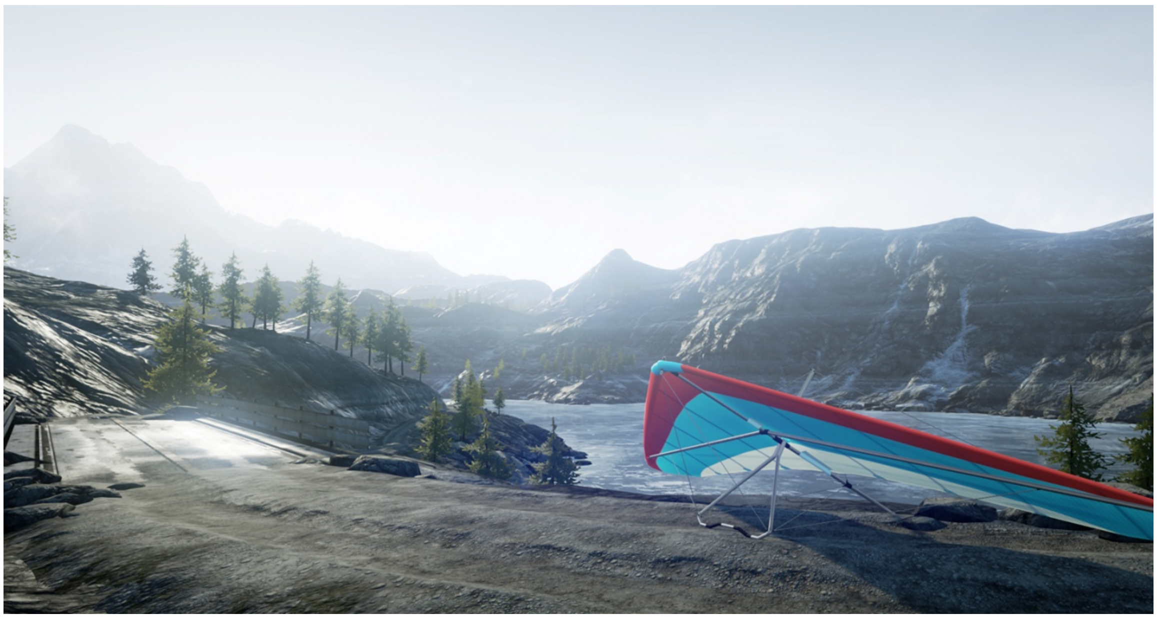

Figure 3. Landscape in Unreal Engine 4 created with World Machine. (Epic, 2014ä.)

3 NORMAL MAPS

A Normal map is a texture map that defines the normal vector of a point on the surface of a 3D-mesh and the process of applying a normal map to a mesh is called normal mapping. This widely used technique is used to create an illusion of a high-resolution detail or better curvature. Normal maps have been widely used in real-time rendering for more than a decade because of their good quality to processing requirements ratio. (Polycount, 2014.)

The red, green, and blue channels of the normal map are used to control the direction of each pixel on the low-poly mesh. The direction of a normal vector defines how each shading point reflects light when rendered. The silhouette of the mesh will not be affected by normal mapping and thus this technique will not work from every viewing angle. Normal mapping is especially effective for real-time rendering applications such as games, because it requires less geometric data to be loaded in to the memory. Normal mapping does however require additional storage for the texture maps. (Autodesk, n.d.)

Normal Maps come in two basic types, tangent-space or object-space, Figure 4. Normal maps can be converted between spaces by using tools designed for it. (Polycount, 2014.)

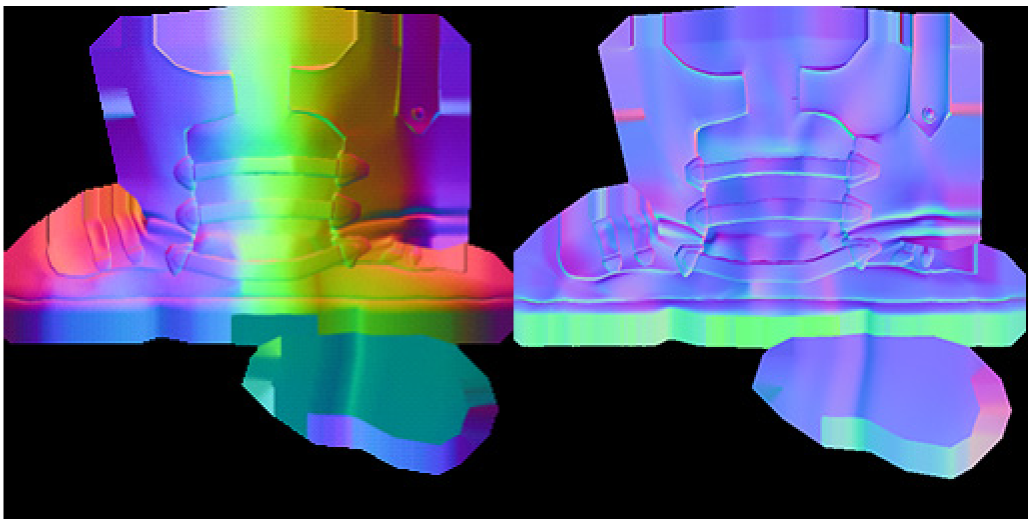
Figure 4. Object-space and tangent-space normal maps. (Crytek, 2013b.)

3.1 Tangent-space

In tangent-space normal maps the information is stored relative to the underlying surface. This gives the tangent-space map its dominant blue color as the blue channel, or Z-axis, is always positive, pointing directly away from the surface. Using Tangent-space normal maps has a lot of benefits and this is why they are more commonly used in games. (Crytek, 2013b.)

- Tangent-space normal maps can be reused on other geometry. The normal map will automatically align to the surface regardless of rotation, mirroring, scale or deformation.

- The geometry independence allows efficient reuse, for example by tiling normal maps on a larger surface to save memory.

- Tangent-space maps are easy to modify to add small detail such as wrinkles or scratches to an already existing normal map.

- Tangent-space maps also give quite good compression, as only two of the channels are used to define the direction. They also provide an ability to reconstruct the blue channel in the shader, so that only two components need to be stored, making it possible to use two-channel textures to store normal maps. (Polycount, 2014.)

3.2 Object-space

In object-space normal maps the information is relative to the object orientation. The map gets its rainbow colors as the normals are stored on a -1 to 1 unit sphere. Even though object-space normal maps have their own advantages the drawbacks make them less commonly used in games. (Crytek, 2013b.)

- Object-space normal maps generally give better quality compared to tangent-space normal maps, as it completely ignores the smoothing of the low-poly vertex normals.

- Object-space maps cannot be easily reused, so different meshes generally require their own normal maps and mirroring requires a shader that supports it.
- The maps don't compress very well, as all of the three channels are used to determine the normal direction.
- Object-space maps cannot be animated either, unless the shader supports deformation. (Polycount, 2014.)

3.3 Creating Normal Maps

Normal maps can be created by either baking information from a high-poly model or by generating a normal map from a 2D-image source. There are multiple workflows and software that can be used, but this thesis will only cover the basic concepts of normal map creation. (Autodesk, n.d.)

3.3.1 Baking

The process of collecting a normal vector data from a high-poly mesh to a texture map is called normal map baking. The software used for baking, a baker, projects a certain distance out from the low-poly mesh and then sends rays inwards towards the high-poly mesh. The projection distance can either be a numerical value or represented as a projection mesh. When a ray comes in contact with the high-poly mesh it stores the mesh's surface normal and saves it in the normal map. The concept of using geometric details of a high-poly mesh was first introduced 1996 in "Fitting Smooth Surfaces to Dense Polygon Meshes" by Krishnamurthy and Levoy. (Autodesk, n.d.)

When baking tangent-space normal maps it is often better not to use smoothing splits because they cause vertex normals to be split. Using split vertex normals results in baking artifacts as the split normals store a different part of the high-poly mesh. However split vertex normals can be used at the UV borders of the mesh to reduce extreme gradients in the normal map. Introducing a smoothing split at an UV border is also efficient as the vertices have already been split once. Having less gradients in the normal map can give better quality when the normal map is compressed. Smoothing splits are not an issue with object-space normal maps as they completely ignore the vertex normals. (Polycount, 2014.)

Most of current day 3D-modeling software are capable of baking normal maps, but there are also software designed just for baking. When choosing a baker it is important to make sure that it uses an averaged projection mesh instead of using explicit mesh normals for baking. This is to avoid projection gaps at smoothing splits during the bake, seen in Figure 5. In 3ds Max 2015 this can be set up by enabling a projection cage in the Render to Texture settings. Another common error that can occur during baking a normal map is having wavy lines when capturing cylindrical shapes. Often times this is caused by the difference between the low-poly and the high-poly mesh and can be easily solved by adding more polygons to the low-poly mesh. (Polycount, 2014.)

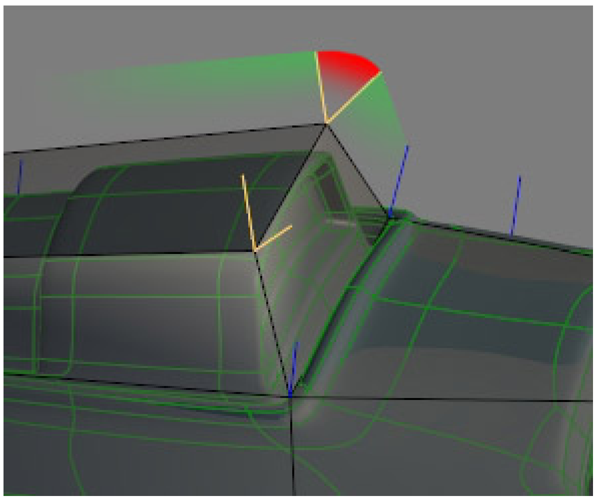

Figure 5. Gap in projection caused by split vertex normals (Tech-artists, 2014b.)

Xnormal, by Santiago Orgaz & co, is a software designed for baking, but it also includes tools for converting height maps to normal maps and so on. xNormal also includes a complete and easy-to-use C++ software development kit to write plug-ins or to create custom mesh importers or exporters for example. xNormal also supports custom tangent-spaces, but by default uses Mikk-Tspace. (Santiago, 2014.)

3.3.2 Generating from 2d source

Normal maps can also be created from a 2D-image source. Generally this type of image conversion tools assume the input as a height map, where black is low and white is high. This height map can be created by hand in generally any image editing software. Photoshop has a plugin by Nvidia that allows to convert an image to a normal map. There are also software designed just for doing this, such as CrazyBump or Knald. As a normal map generated from a 2D-source cannot quite reach the quality level of a baked normal map, this workflow is often used to create smaller detail that is then overlaid to an already existing normal map. This can be done in Photoshop by using the overlay blending mode and clamping the blue channel of the layered normal map to 127. This workflow does not generate the most accurate results, seen in Figure 6. For better quality it is best to use a software that is capable of combining normal maps, such as CrazyBump. After editing normal maps by hand it is generally best, but not always required, to re-normalize the normal map to make sure the resulting vector lengths equal one (1.0). (Polycount, 2014.)

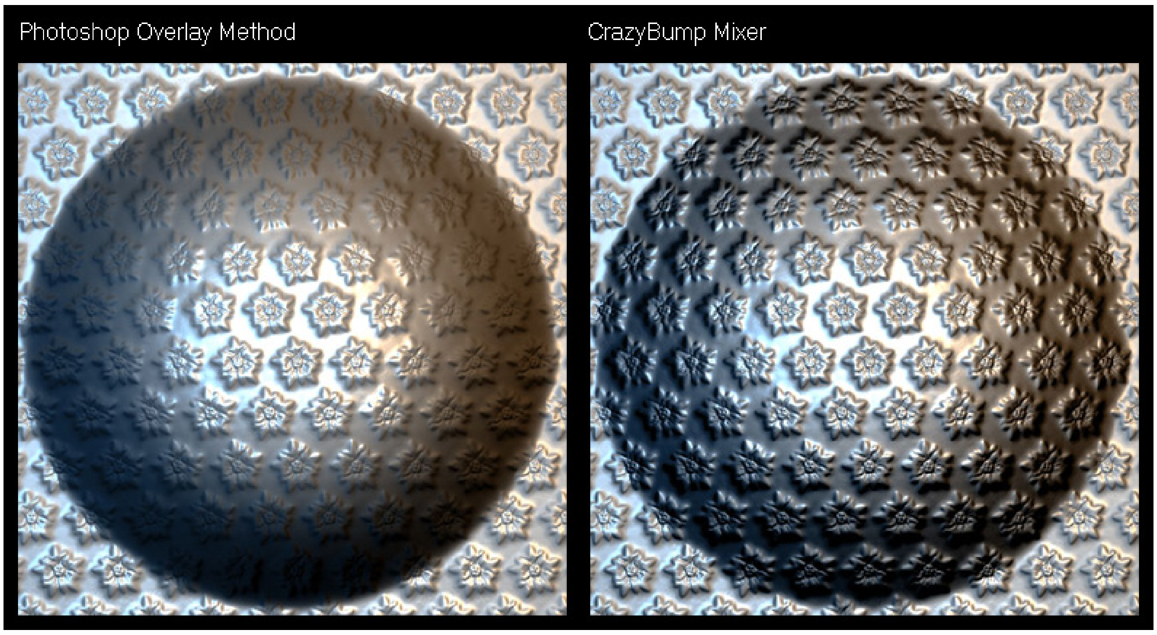

Figure 6. Texture overlay in Photoshop and CrazyBump. (Clark, 2009.)

3.3.3 Tangent-spaces

When using tangent-space normal maps it is important to make sure that the application used to bake the normal map and the application used to render the normal map agree how the normal maps are computed, especially how the tangent basis is calculated. In other words, it is important to make sure that both of them talk the same language. Tangent basis is a vertex data that provides directionality across the surface. Tangent plane basis vectors, vertex's normal, tangent, and bitangent are calculated for each vertex in the mesh to create an axis that is used to transform the incoming light from world-space to tangent-space. This is done to have the normal mapped mesh lit properly in the rendering application. There are no standards for how tangent-space normal maps should be calculated and most software have their own implementations. Tanget-space mismatch errors cause visible seams and shading errors, seen in Figure 7, when the normal map is rendered. Handplane is a tangent-space calculator, a tool designed to solve tangent-space mismatches. By combining an object-space map with the low-poly mesh, it can create an engine specific tangent-space map. Currently Unreal Engine 4 output is not supported by handplane. (Polycount, 2014.)

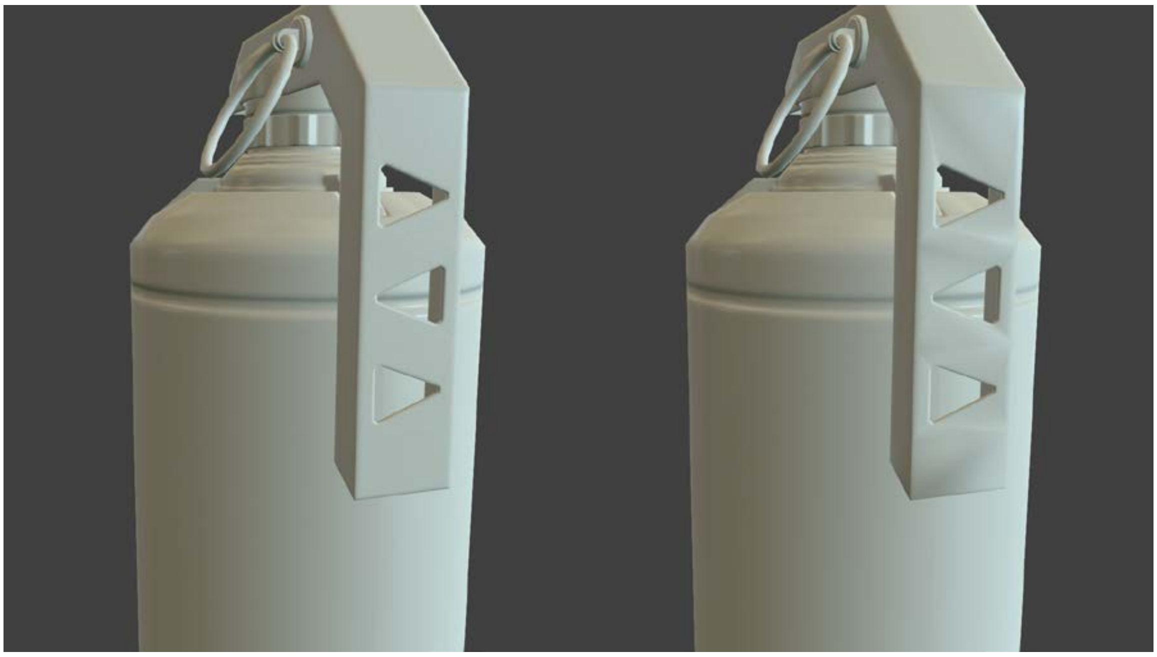

Figure 7. Normal mapped mesh rendered in synced and non-synced renderers.

Unreal Engine 4's tangent-space is almost synced to Mikk-Tspace. Because the support is not full there can still occur some errors when it comes to calculating tangents. Currently this can be gone around by importing custom tangents and normals from a 3D-modeling software to be used by xNormal and Unreal Engine 4. Shaders are written to use a particular direction for the X and Y axes in a normal map. This feature is called Handedness, which means that the green and red channel of the normal map need to be setup correctly for the shader. Unreal Engine 4 uses +X, -Y, +Z, and gives an option for flipping the green channel, Y-axis, in the Texture options. (Polycount, 2014c.)

4 PHYSICALLY BASED RENDERING

Physically based rendering, PBR, is a concept of using realistic shading and lighting models with measured surface parameters to accurately represent real-world materials. PBR has quickly gained popularity in game development and with increased computing power PBR has become the new standard in real-time rendering. Using physically based, energy-conserving shading models, makes it easier to achieve realistic materials that give a consistent look in a variety of lighting conditions. It is important to understand that PBR is a concept and that the actual implementations between systems may vary. The biggest benefits of a PBR system is that it gives artists less parameters that are more intuitive to work with. Another benefit is consistency, having measured base values for materials removes some of the guess work, making content more consistent between artists. (Russel, 2014.)

4.1 Theory

The physical phenomena of shading is related to the interaction of light and material**.** The interaction depends on the physical characteristic of the light as well as the parameters of the matter. For example a rough plastic surface reflects light very differently than a smooth chrome surface. With PBR most materials can be created using just one type of a shader. (Russel, 2014.)

The core essences of a PBR system are the bi-directional reflectance distribution functions that describe how and how much of the light is reflected when light makes contact with a certain material, and how light reflects off a surface when viewed under various viewing positions. This document will not assess the actual functions, but will discuss some of the physical properties that describe how the incoming light interacts with a material surface. (Karis, 2014.)

4.1.1 Diffuse reflection

Diffuse reflection is the reflection of light from a surface in which an incident ray is reflected at many angles, Figure 8. When light hits a surface some of the rays will enter the object, and within the object the light will either be scattered or absorbed by the material. As some of the scattered light resurfaces from the object it becomes visible again. Different wavelengths of the light are absorbed differently giving the object its visible color, for example if an object absorbs most light but scatters red, it will appear red. Albedo, the amount of diffuse reflection from an objects surface, can be calculated and is measured on a scale from 0.0 for no reflection of a perfectly black surface to 1.0 for perfect reflection of a white surface. Generally these extremes don't appear in real world. For example charcoal, one of the darkest substances on earth, has an albedo value of around 0.04 and fresh snow, the whitest substance, has a value of around 0.90. (Russel, 2014.)

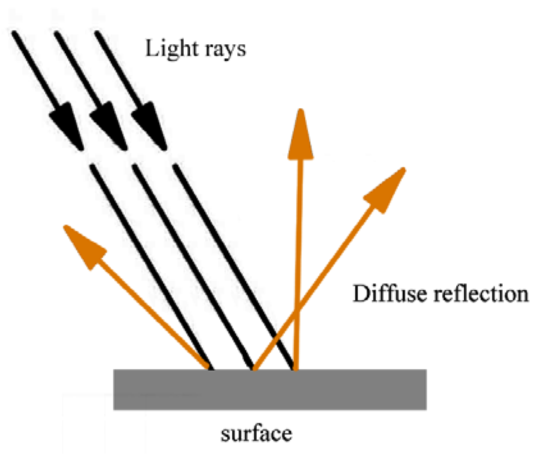

Figure 8. Simplified illustration of a diffuse reflection.

4.1.2 Specular Reflection

Specular reflection is the reflection, in which the light from a single incoming direction is reflected into a single outgoing direction, Figure 9. This type of reflection happens immediately when the light hits the surface. Generally materials can be divided into a two major categories based on their specular reflectivity; conductors and insulators. Every material in the real-world has some amount of specular reflection and the reflectivity often remains fairly constant for each given material type. The amount of specular reflection can be defined by the materials Index of refraction (IOR). The IOR scale determines how fast light travels through a material in relation to a vacuum. (Russel, 2014.)

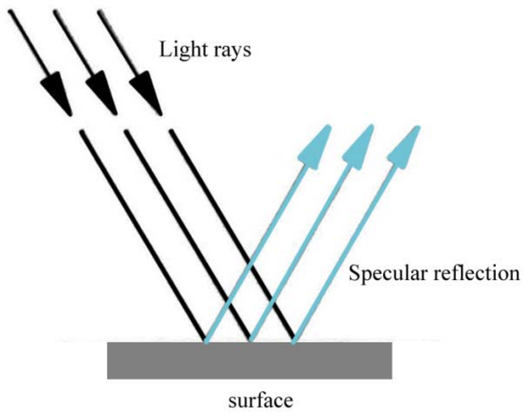

Figure 9.Simplified illustration of a specular reflection.

Electrically conductive materials, most notably metals, tend to have a high amount of specular reflectivity, up to 60 - 90%. As most of the light does not reach the interior of the material the surface appears shiny. Conductors also usually absorb rather than scatter light and because of this conductors generally do not show any diffuse reflection. Certain metals, such as gold and copper, also have a tinted reflection when lit by a white light, which is a trait mainly featured by conductors. Insulators on the other hand generally feature a low amount of specular reflectivity, from 2 - 5%. It is also common for the reflection to appear white when lit by a white light. (Russel, 2014.)

4.1.3 Energy Conservation

An important aspect of diffuse and specular reflection is that they are mutually exclusive, seen in Figure 10, as the light needs to first enter to object to be diffuse reflected. This concept is known as energy conservation, which means that the light reflected from a surface is never any brighter than that which illuminated it in the first place. The law of conservation of energy makes it harder for artists to create materials that are not physically plausible, as it prevent the surface from emitting more light than it received. (Russel, 2014.)

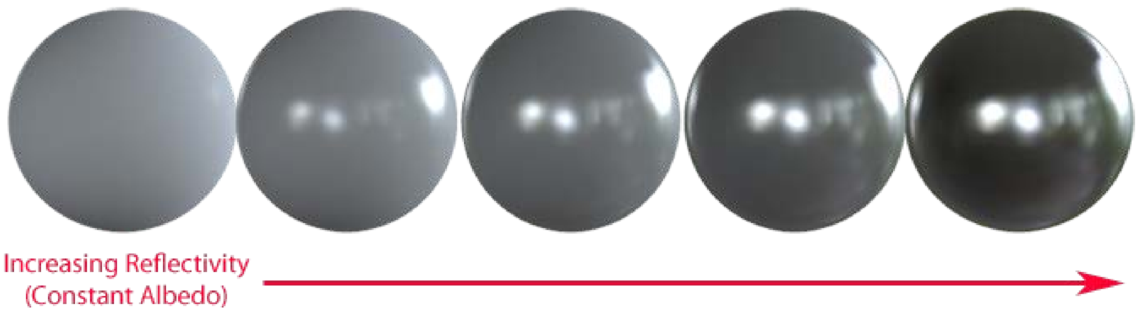

Figure 10. Energy conservation. (Russel, 2014.)

4.1.4 Microsurface

Microsurface defines the roughness of a surface. These small imperfections featured in real-world surfaces can be too small for the eye to see. Despite of this, these miniscule dents and grooves affect the reflection of light, causing the incoming light to diverge when reflected from a rougher surface, Figure 11. This small detail plays an important role when defining a material as the real-world is full of a wide variety of microsurface features. As the light rays reflected from the surface spread the reflection appears more blurry and the surface displays a wider specular highlight. And because of the law of energy conservation, the wider reflections also appear dimmer than sharper highlights on a smooth surface. (Russel, 2014.)

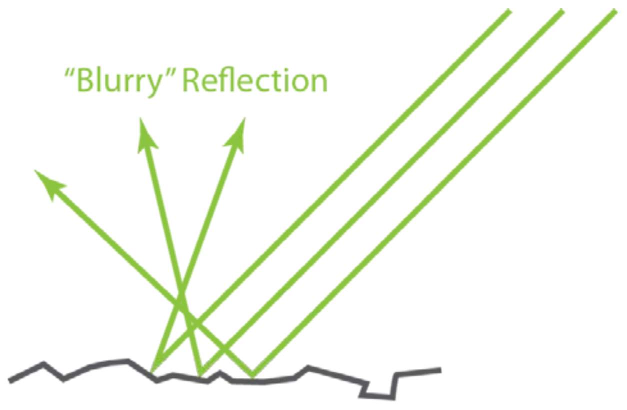

Figure 11. Illustration of specular reflection from a rough surface (Russel, 2014.)

4.1.5 Fresnel

Fresnel is the percentage of light that a surface reflects at grazing angles. This is because the light that hits a surface at a greater angle will be much more likely to be reflected than that which hits the surface dead-on, meaning that for all smooth materials the reflection becomes total at an extreme angle. Again, this is an effect featured by all materials in the real-world, even though this effect becomes less evident with more rough surfaces. (Russel, 2014.)

4.2 Comparison to traditional systems

What makes a physically-based rendering system different from a traditional model is the more detailed behavior of light and matter. As shading capabilities have increased can some of the old approximations now be discarded, and with them some of the old workflows of creating content. (Wilson, 2014.)

One of the biggest differences is the texture map inputs for creating materials. Most of the new inputs still share similarities to the inputs of a traditional model, but some are completely new and some of the traditional inputs are now just called different to better describe what the input actually does. The use of measured reference tables is also extremely important when working with a PBR system, as the shaders rely on physical values in order to produce photorealistic results. A major difference compared to a traditional texturing workflow is that the diffuse texture map should not include any lighting information or ambient occlusion. All lighting effects are now either done by the engine or have their own input slot in the shader. The diffuse texture should only include the measured color and albedo of the material wanted. Specular maps are generally authored only when needed, as most insulators can be represented with reflectivity of 0.04. The values used in a specular map should be a physical value that is a constant for a single material type. The importance of a microsurface texture map has also increased massively from traditional models as it is the texture map where all the small imperfections are stored. (Wilson, 2014.)

4.3 Physically Based Rendering in Unreal Engine

Unreal Engine 4 has adopted a new physically based material, shading and lighting model, in which the materials are defines by three different properties. The material model used in Unreal Engine 4 is a simplified version of Disney's model for real-time rendering that features more intuitive parameters compared to traditional ad-hoc models. All of these inputs take in values between 0.0 and 1.0. (Karis, 2013)

4.3.1 Base Color

The Base color input defines the diffuse reflectivity of the material, the intensity and color of the light that bounces off in all directions from the surface. The amount should be in the range of 50-243, as seen in Figure 12. The base color parameter is used in the diffuse interreflection calculations of a global illumination lighting effect. In case of conductors the base color also determines the amount of specular reflectivity and generally this is between the range of 186-255. (Epic 2014a.)

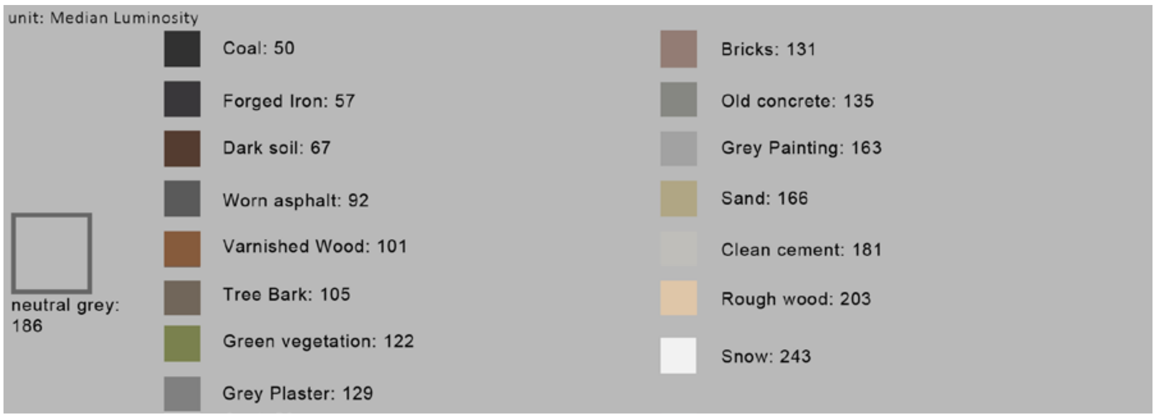

Figure 12. sRGB Albedo values. (Dontnod Entertainment, 2014)

4.3.2 Metallic

Unreal Engine 4 features a new texture map type, called a metallic map, to control the specular reflectivity of a material. This shading model also takes Fresnel effect into account. Based on the idea that in real-world conductors and insulators are shaded very differently, the metallic input works as a mask to define in which category a pixel on the surface belongs to. The metallic input replaces the specular input of traditional models. (Epic 2014a.)

Insulators have a metallic value of 0.0 and conductors have a metallic value of 1.0. For pure insulator and for pure conductor this value will be 0.0 or 1.0, not anything in between. When creating surfaces such as rusty or oxidized metals, values between 0.0 and 1.0 can be used. The reflectivity value used by insulators is 4% and the reflectivity for conductors is defined in the base color input. (Epic 2014a.)

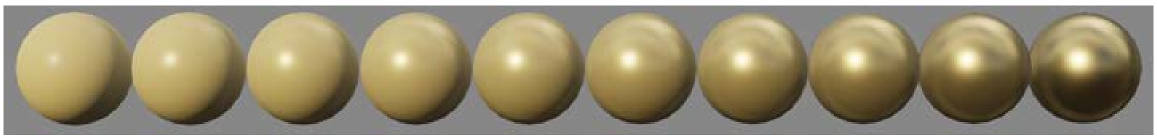

Figure 13. A chart representing metallic values from 0.0 to 1.0. (Epic 2014a.)

4.3.3 Roughness

The roughness input defines how rough the material is. Roughness of 0.0 is a mirror reflection and a roughness of 1.0 is a completely matte surface, seen in Figure 14. Roughness has become one of the most important texture map as the actual reflectivity value of a given material does not vary widely in the real-world. Roughness maps are used to adjust both the apparent brightness of a reflection and the size and sharpness of a reflection. (Epic 2014a.)

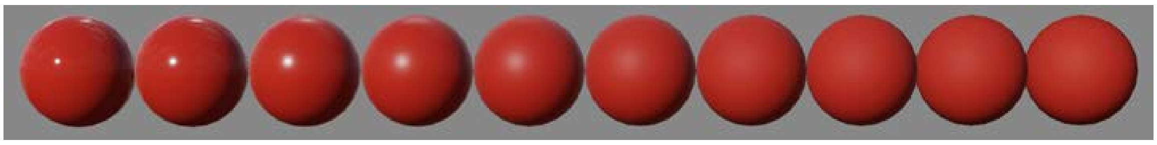

Figure 14. A chart representing roughness values from 0.0 to 1.0. (Epic, 2014a.)

5 TEXTURING

Texture mapping is a method of applying surface parameters to a 3D-mesh. The common method is to create a 2D-bitmap called a texture map, Figure 15, which is then projected around a 3D-mesh based on the UV-coordinates stored in the vertices of the mesh. This texture map provides a pixel based data, a component of a material, to the shader. A single material may take multiple texture maps that all have a different purpose. (Epic, 2014b.). A UV-space is a 2D-coordinate space in which a point on the object's surface represents a point in the coordinate-system. For the texture maps to view correctly on the 3D-mesh, the polygons need to be laid properly to the UV-space, this process is called UV-mapping. (Tech-Artists, 2008.)

The resolution of a texture map is generally in power of two, for example 128x128 or 512x512. The texture map does not need to be a square as long as the width and height are in power of two, for example a texture map resolution could be 1024x256. This comes from making the texture maps optimized for the graphics hardware. Currently Unreal Engine 4 supports texture resolutions from 1x1 to 4096x4096 and non power of two texture maps can also be imported, but they will not be streamed nor mipmapped properly. (Epic, 2014i)

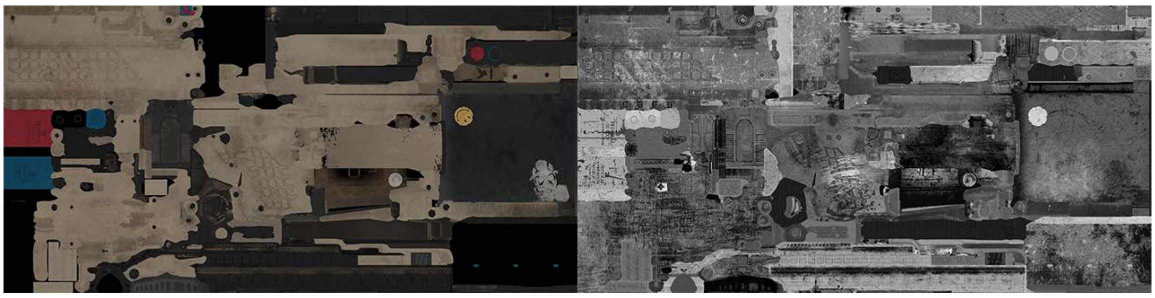

Figure 15. Texture maps for diffuse reflection and microsurface. (Vromman, 2014)

5.1 Linear and Gamma space

When creating texture maps for a physically based shaders it is important to understand that Photoshop works in a gamma color space. A medium gray value in gamma is 127, but 0.5 raised by the inverse of gamma 2.2 which equals to 186. This is because the monitor is applying a gamma curve, seen in Figure 16, to the output. This important factor comes into play as most rendering equations expect linear inputs. (Crytek, 2013.) Calculating in linear space gives a multiple benefits compared to calculating in gamma space, these benefits are for example more natural light falloff and intensity response. (Gritz & d'Eon, 2008.) For proper texture creation should the Photoshop color management be set up as RGB to sRGB and Gray to Gray Gamma 2.2. This is because the default Gray setting Dot Grain 20% causes color transformation in the alpha channel. (Crytek, 2013.) Unreal Engine 4 gives an option to determine whether the texture map should be gamma corrected or not by flagging the textures as sRGB. Generally sRGB should only be used for color textures, such as the base color texture, but ultimately it can be used for every other texture type but normal. It is also important to calibrate the monitor for proper texture authoring if possible. (Epic, 2014q.)

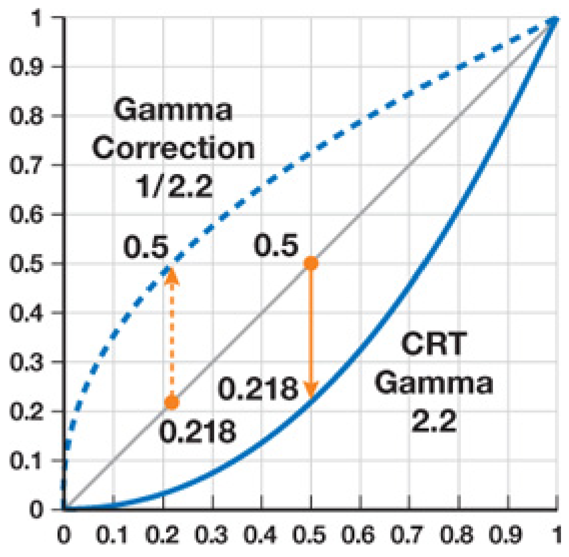

Figure 16. Gamma curve. (Gritz & d'Eon, 2008.)

5.2 Texturing workflows

Creating texture maps for a PBR systems, such as Unreal Engine 4, is very similar to texturing for traditional models. There are many ways of creating texture content for PBR systems and it is important to remember that it is the end result that matters. With physically based shading the importance of material reference sheets and base material libraries has grown significantly, but gathering this data for every material needed can be time consuming and expensive. This is why it is important for artists to also have the ability to eyeball values when necessary and to use reference sheet as a guideline to create as physically accurate content as possible. Texture map data can also be scanned from real-world surfaces. Quixel offers a service called Megascans that provides scanned texture data for content creation, and the workflow of using scanned material data has already been used in games such as Metal Gear Solid 5 and the Order: 1886. This document will not focus on how to acquire this data, but on a couple common workflows and software that can be used for authoring texture maps. (Wilson, 2014)

Photoshop, or any image editor, can be used as a texturing tool, but as hardware computing power has increased, so has the detail level of a high-quality asset. And with the introduction of specific tools designed just for texturing can Photoshop's native toolset for texturing feel rather basic, yet it is still widely used in the industry and is good for creating stylized hand painted textures as seen in games such as League of Legends and World of Warcraft. (Adobe, 2014)

Quixel Suite is a texture creation toolkit add-on for Photoshop. The Suite provides tools for all kinds of texturing needs, and a part of the Quixel Megascans material library is also included with the Suite. Quixel has quickly grown in popularity and is used by studios such as Blizzard, Crytek and Infinity Ward. (Quixel, n.d.)

Substance Painter and substance Designer, by Allegorithmic, a 3D painting software and a node-based texture compositing tool that allow easier texturing with a wide library of tools. Substance painter, seen in Figure 17, won the Best New Application prize at 3D world's CG awards 2014. The Allegorithmic tools are used by companies such as 343 studios and 2K. Allegorithmic's pipelines are fully supported by Unreal Engine 4. (Allegorithmic, 2013)

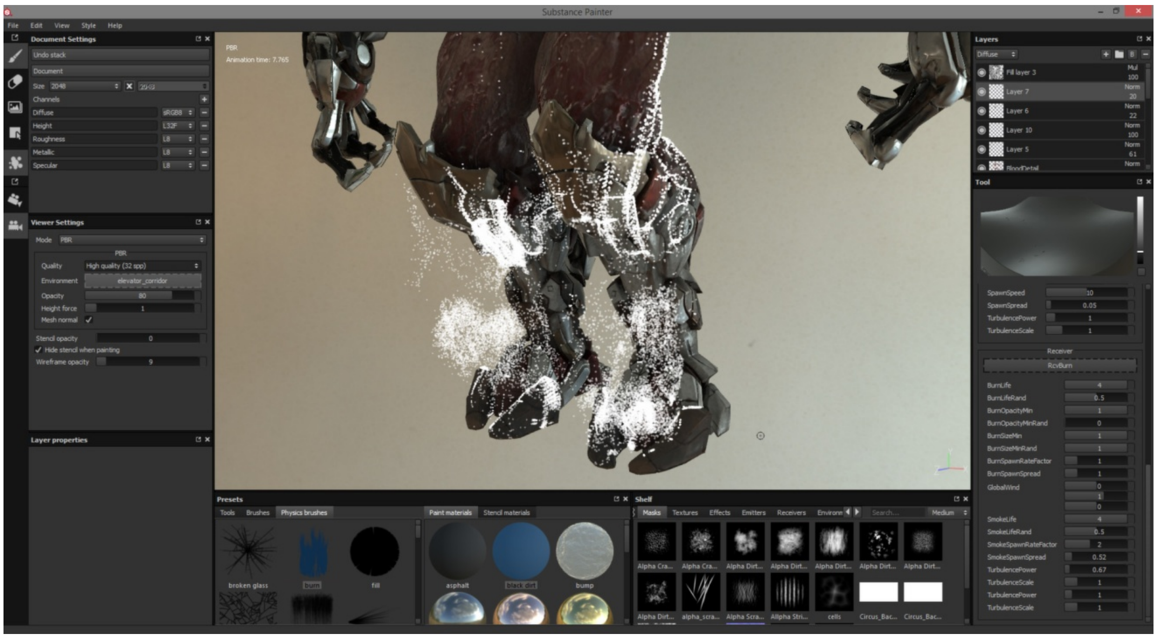

Figure 17. Substance Painter. (Allegorithmic, 2013b.)

5.3 Texel-Density

Texel-density describes how many texels cover a game unit. A constant texel-density between meshes helps to maintain visual consistency and texture memory usage. (Haneman, 2013.) The texture size for a mesh should be related to the amount of space that the mesh will use on screen, bigger meshes require bigger texture maps. It is also common for meshes that are always on screen, like the player character, to have a higher texel ratio. (Crytek, 2014.) There are tools, like TexTools, that make it easy to ensure that the texel-density is even between meshes. Low texel-density on a surface can also be efficiently hidden by using tiling detail maps in combination with the base textures. Unreal Engine 4 provides a view mode to see lightmap texel-density across the game environment. (Epic, 2014j.)

6 UNREAL ENGINE

Unreal Engine 4, Figure 18, is a complete suite of game development tools and technologies developed by Epic Games, released March 2014, for building games on a wide range of platforms. Currently Unreal Engine technology powers hundreds of games from AAA to indie. (Epic. 2014c.)

Unreal Engine is a game engine, a software framework designed for the creation of video games. The core components that a game engine can provide include a rendering engine, a physics engine, sound, animation systems, Artificial intelligence, and so on. The benefits of using a game engine is that it contains all the core components needed to start a development. In many cases the game engine also provides multiple visual development tools in addition to the reusable components. (Ward, 2008.)

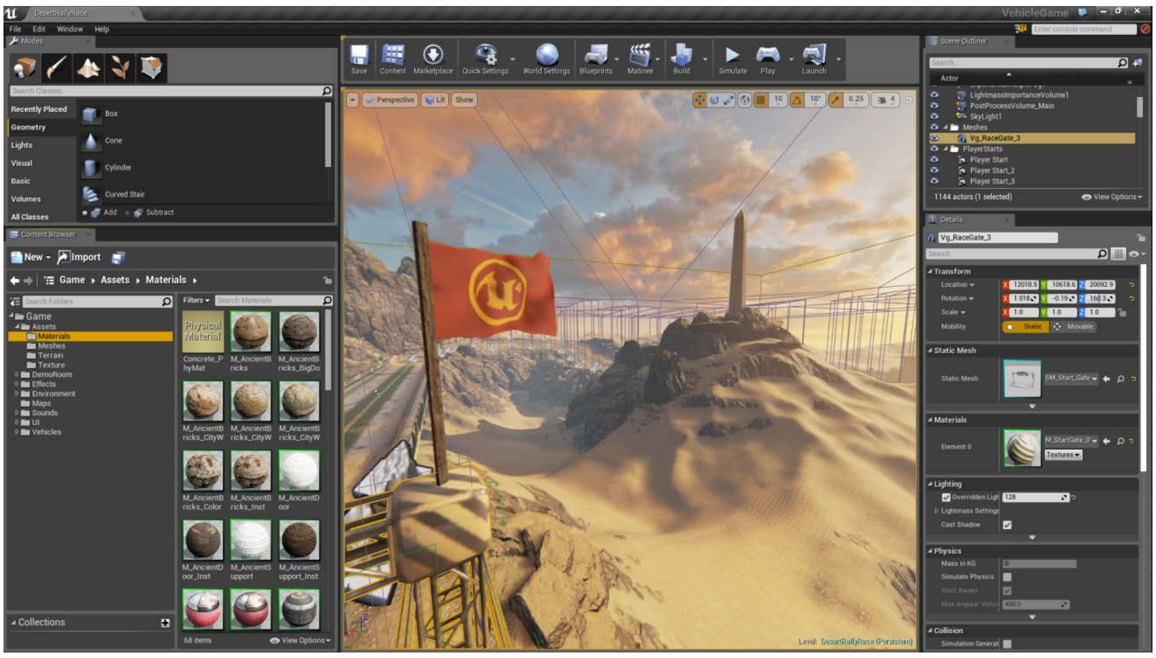

Figure 18. Unreal Engine 4.

Unreal Engine 4's DirectX 11 rendering system includes deferred shading, global illumination, as well as GPU particles among other features (Epic, 2014c). The greatest advantage of deferred shading is the decoupling of scene geometry from lighting. Just one geometry pass is required and lights are calculated only for those pixels that they affects. This gives the ability to render multiple dynamic lights in a scene without a significant performance-hit. Traditional Unreal Engine 3 lighting is called forward shading, because the dynamic lighting calculations are done while rendering the meshes of the scene. Deferred shading comes with a certain restrictions, for example custom lighting models are not supported by Unreal Engine 4. (Epic, 2012b.)

6.1 Physically Based

Unreal Engine 4 has moved to a PBR system. This means that the lighting and shading used in Unreal Engine 4 resembles the physical interaction of light and matter seen in real-world. Also the units used in Unreal Engine are common physical measurements, and the material parameters can be translated from real-world. For example 1 default unit, called unreal unit in Unreal Engine 4, translates to 1 centimeter in real-world. (Epic, 2014d.)

Lights in Unreal Engine 4 also use algorithms and units that closely follow real-world examples. For example the light intensity is provided in brightness unit of lumens and the light intensity fall-off follows the physically accurate inverse square law to achieve more natural results. (Karis, 2014.) And because every surface in real-world gives at least some level of a specular reflection as well as scene reflection, so do the materials in Unreal Engine 4. The material shading also follows the law of conservation of energy. (Epic, 2014d.) From version 4.5 Unreal Engine 4 also supports ray traced distance field shadows to produce soft area shadows with sharp contacts, this can be enabled by generating required mesh distance fields. (Epic, 2014k.)

6.2 Material Editor

The Material editor, seen in Figure 19, in Unreal Engine 4 is used to create different materials by using texture map inputs or math functions to determine how light interacts with the surface. The materials are constructed by nodes, called material expression, that are designed to perform certain tasks or act as an input. Essentially the material editor is a form of a visual scripting as every node contains a bit of HLSL code and a more programming-savvy users can also create their own custom material expression nodes. The material editor reads values from each channel on a range from 0.0 to 1.0. The material editor also provides multiple different settings for shading models, tessellation, and translucency to name a few. Materials in Unreal Engine 4 can be simple with just a few expressions or an extremely complicated web of nodes. (Epic, 2014e)

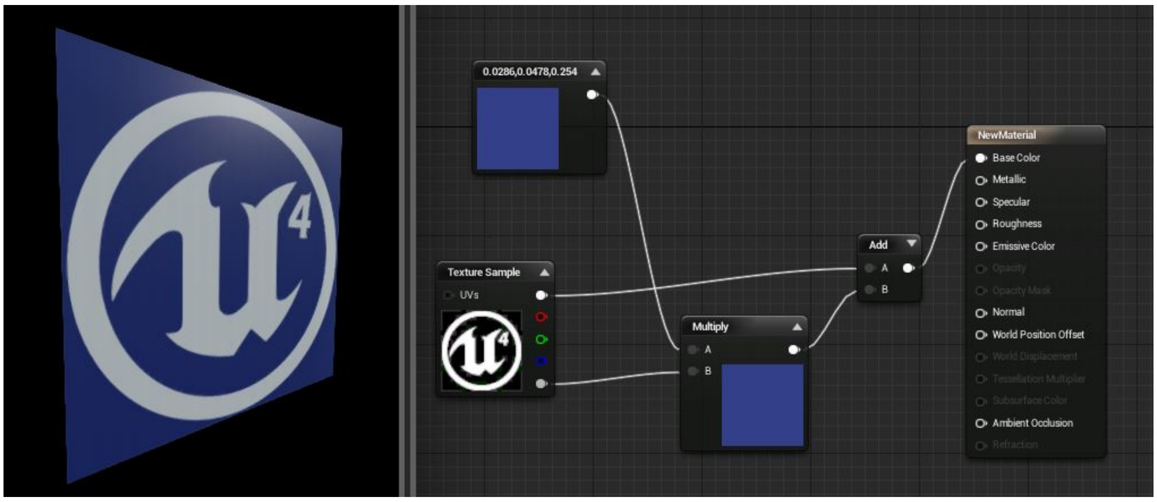

Figure 19. Unreal Engine 4 material editor. (Epic, 2014a.)

Material Instancing in Unreal Engine 4 is used to change the appearance of a material without recompiling the material. Some instanced materials can also be changed during run-time, thus giving more artistic flexibility. These material instances require that certain properties are made editable, this step is called parameterizing the material. There are two type of instanced materials available, constant and dynamic. Material instance constants are calculated once, prior run-time, and cannot change during gameplay, but provide performance advantage. One workflow is to create a master material that represents the base aspects of a given generic material, for example car paint, and then to create material instance constants to represent the variations of the car paint. Material instance dynamics are calculated during runtime and can be scripted to change during gameplay. (Epic, 2014l.)

Unreal Engine 4 also features layered material which essentially works as materials within materials. Layered materials allow to create a single material that has a series of sub-materials that can be placed across the mesh using masks. Layered materials work as an extension material functions, a self-contained node network that can be reused at any number of materials. The benefits of material layering are that it allows to break a complicated material into a smaller, more manageable components. The reusability of material functions also allows to set up a library of basic material types that define a variety of basic real-world surface types. Even though layered materials are useful when handling multi-Material setups, they can also be performance intensive as all of the layers are rendered simultaneously. The system must test to see which of the layers are blended and then reject any of them that are not in use. This is why it can be more convenient to increase the material element count of an object when no per-pixel control is required. (Epic, 2014m.)

6.3 Lightmaps and Lightmass

Lightmass creates lightmaps with complex light interactions like area shadowing and diffuse interreflection. Lightmass precomputes portions of the lighting of lights that are set as stationary or static in the Unreal editor and saves them in a lightmap. Lightmaps provide an efficient way of achieving a high-quality global illumination effect and ambient occlusion**.** (Epic, 2014f.)

Global illumination is a group of algorithms that take into account not only the light which comes directly from a light source, but also the surface reflected rays of the same light. The brightness and color of the bounced light depend on the light source as well as the material that the light reflects from. This technique creates a much more realistic and detailed lighting in the scene, providing more natural lighting results. Lightmass also places samples in a 3D-grid, seen in Figure 20, that store the indirect lighting from all direction. These samples are then used to calculate the indirect lighting for moving or dynamic objects, such as characters. Light propagation volumes is a technique that allows to approximately achieve global illumination in real-time. This technique is still in development by Epic Games, but can already be enabled from the ConsoleVariables.ini file of the Engine. (Epic, 2014f.)

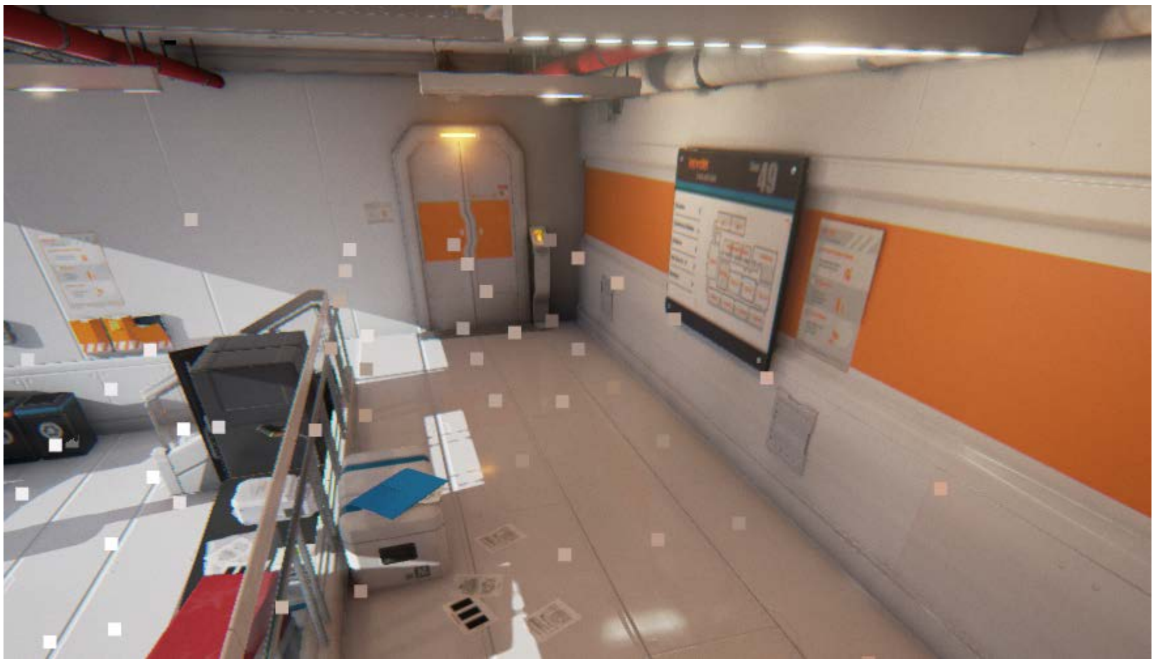

Figure 20. Indirect lighting samples.

Lightmass ambient occlusion calculates indirect shadowing for ambient lighting, applying it to both direct and indirect lighting (Epic, 2014f.) Ambient occlusion can also be enabled as a post-processing effect, called screen space ambient occlusion, but this does not affect the direct lighting and is generally best used in addition to the baked ambient occlusion effect. (Epic, 2014o.) Unreal Engine 4 also supports distance field ambient occlusion that generates ambient occlusion from signed distance field volumes precomputed around each object, allowing dynamic changes in the scene. (Epic, 2014p.)

Lightmass Lightmaps require their own UV channel and by default Unreal Engine 4 uses the channel 0 for lightmaps, but this can be changed from the lightmap coordinate index in the mesh editor. Good lightmap UVs need to be uniquely mapped so every part of the mesh has its own space to receive shadow and lighting information. All of the faces also need to be within a single UV space. Each UV shell also needs to have enough padding around it to prevent bleeding, generally 2-4 pixel padding is enough. It is also recommended to snap UV borders to a pixel grid if possible to prevent seams in the lightmaps. (Epic, 2014n.)

6.4 Collision

Collisions are a part of the physics engine that determines how two meshes react when colliding**.** In real-time graphics collisions are generally calculated by using a simplified collision meshes to ensure a simple collisions, seen in Figure 21. This is mainly because it is faster to calculate collision for a simple mesh with just few polygons. In Unreal Engine 4 the collision mesh can either be created within the mesh editor by using the basic tools provided, or be imported with the actual in-game asset from an external software using a UBX_, USP_ or UCX_ prefix, depending on the type of custom collision shape needed. Unreal Engine gives an option to choose from multiple models, called collision responses, that determine how Unreal Engine handles collisions between different object types. In a simple collision, objects can either block or ignore each other, but they can also send a signal for a part of code to be triggered. The mesh editor also allows for physical materials to be set for an objects, allowing the user to fine tune certain physical attributes, such as friction, that happens during the collision. **(**Epic, 2014g.)

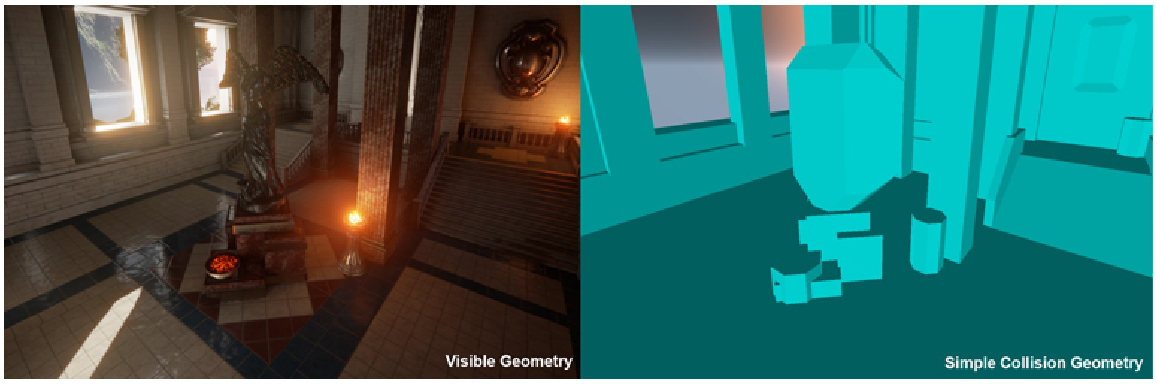

Figure 21. Visible geometry and collision geometry of a scene. (Epic, 2014.)

6.5 Blueprint system

The blueprint system is a visual scripting system in Unreal Engine 4. Like the materials of the material editor, blueprints are based on a concept of using nodes to create various game-play features. Scripts are instructions that can be executed by a computer software. The blueprint system makes it easier for artists to create certain effects, such as controlling the intensity of a dynamic light within the scene during run-time or to create procedural content. Blueprints come in multiple different types, each used for different purpose, for example level blueprints are a level-wide global event, whereas blueprint utilities can only perform editor actions or extend the editor's functionality. (Epic, 2014h)

7 THESIS PROJECT FUTURISTIC LAB

The thesis project was created within a time frame of two months, from design to final product, by Tuukka Mäkinen and myself. The project's main goals was to get familiar with the user interface and asset pipeline of Unreal Engine 4, as well as to brush up high- and low-poly modeling skills. The project's original concept was to create an indoor scene with a futuristic theme that is playable on PC. Initially the idea was to have a team of four artists, but in the end we had to go with just two.

Unreal Engine 4 had been out for just a couple of months before the project started, so some of the documents were still outdated and part of the features were no fully operational, for example we had to enable decals from the .ini files of the engine. This caused some issues during development as the workflows in the documents produced unexpected results and due to time constraints we did not have the time to test all possible workflows.

7.1 Preparation

Of the seven weeks we spent approximately a week on concept and design as well as getting used to the engine. This included creating a simple concept picture of the layout, gathering reference images from real-life and from other video games and established a naming convention and folder structure for the assets and textures. Most of the actual assets were conceptualized on the go, so there are next to no 2D-art done for the preparations.

The design was to create a rather realistic futuristic office or lab space with clean surfaces and a strong, but sparse, use of color. After the initial concepts were done, we started to block out the environment within the Unreal Engine using the Geometry Brushes, to quickly prototype what we wanted. We decided to use a first person perspective player camera, so we had to make sure that the scale of the room and objects felt correct. In the end we had to scale some of the objects slightly bigger in the engine to make sure that scale of the objects worked well together. After we got a somewhat working structure for the environment done, we started to set up the actual asset creation pipeline. This included setting up 3ds

Max unit and export settings to match what Unreal Engine expected. The fact that there were only two of us proved to be quite beneficial during this preparation stage as we were able to make decisions fast and we both quickly understood what we wanted to create.

The next step was to create an asset list, where we listed all the assets we were about to create based on their priority. We already had some experience on estimating the time required to create the content, so we were able to create a rather precise list that lasted until the end of the project with just few additions. Luckily the .FBX pipeline from 3ds Max to Unreal Engine 4 turned out to be a rather simple process, so we got content into the engine quite quickly. Looking back at the project now, we could have used an export script to increase the production speed and scene management. Because of the time constraint we ended up doing quite a few non-optimal choices for the asset and texture pipeline that we would not have done for an actual game project. For example, we could have compressed our texture map more efficiently by combining multiple grayscale maps into different channels of a single texture map and so on, but without a proper setup this could have taken more time than just importing them separately. Essentially we sacrificed efficiency for development speed now and then.

By the time the normal mapping documents of Unreal Engine 4 were outdated and did not give expected results but created shading errors, tri splits and so on, we assumed that the normal maps were not synced to Mikk-Tspace and by the time documentation for other workflows was still few and far between. And as we did not have the time to test multiple normal baking workflows we made a decision to bake normal maps in 3ds Max and to introduce smoothing splits to control problematic shading areas. In the end this produced acceptable quality and saved a little time as we did not need to move between different software. Overall, this still cost more time and did not produce as good quality that a completely synced pipeline would.

For the materials we chose to use a few master materials that were then instanced to create multiple variations. These include simple tiling material for large surfaces such as the floor and walls, a basic material for props, an emissive material and so on. We set the texel-density for the scene to be 512 pixels for each meter in the game environment. So a 400x400uu wall had a 2048x2048 texture map. We did not use any tools to verify the texel-densities for more complex meshes, so most of the time we just went with what looked good. For texture reference sheets we mainly used the values provided by Epic Games as well as a Unreal Engine 4 material chart by Dontnod studios. These provided enough data for us as we did not have a wide variety of different materials.

Lightmass and lightmapping were familiar to us as concepts because of previous experience with Unity game engine, but the lightmaps of Unreal Engine behave quite a bit differently. We set up a scale reference to determine an estimation of needed lightmap resolution, bigger objects, such as the tables, use a lightmap resolution of 256, whereas smaller ones use sizes of 64 or 32. Almost all of the lights in the scene are either static or stationary, so they are used by the lightmass.

7.2 Tools

The main tool used for asset creation was 3ds Max as we did modeling, UV mapping, lightmapping and in the end normal map baking in it as well. Texturing was done in Photoshop with the help of the Quixel suite. Quixel Suite was a new software that neither of us had used before, so the workflow took some time to get used to. As we had limited time to learn new software we did not use the Suite to its full extent and mainly used NDO, which is the normal map generation tool of the package, to produce smaller detail that we overlaid on our normal maps.

3ds Max was already familiar to us, so the actual content creation was rather straightforward. The unit setup was set so that one unit in 3ds Max matched to a one cm, this is the default scale used by Unreal Engine. One of the goals of the project was to practice our high-poly modeling, so we made high-poly meshes for almost every asset seen in the scene, Figure 22. During the development we had access to Mudbox, a sculpting software by Autodesk, but because the environment was mostly constructed of man-made hard-surface objects we did not feel the need to use digital sculpting. We followed the .FBX pipeline provided by Epic to export our meshes from 3dsMax to Unreal. The .FBX document included the information for the correct .FBX version to use, where the pivot point is located and whether textures are imported with the model. By the time Unreal Engine 4's import pipeline used .FBX version 2013.

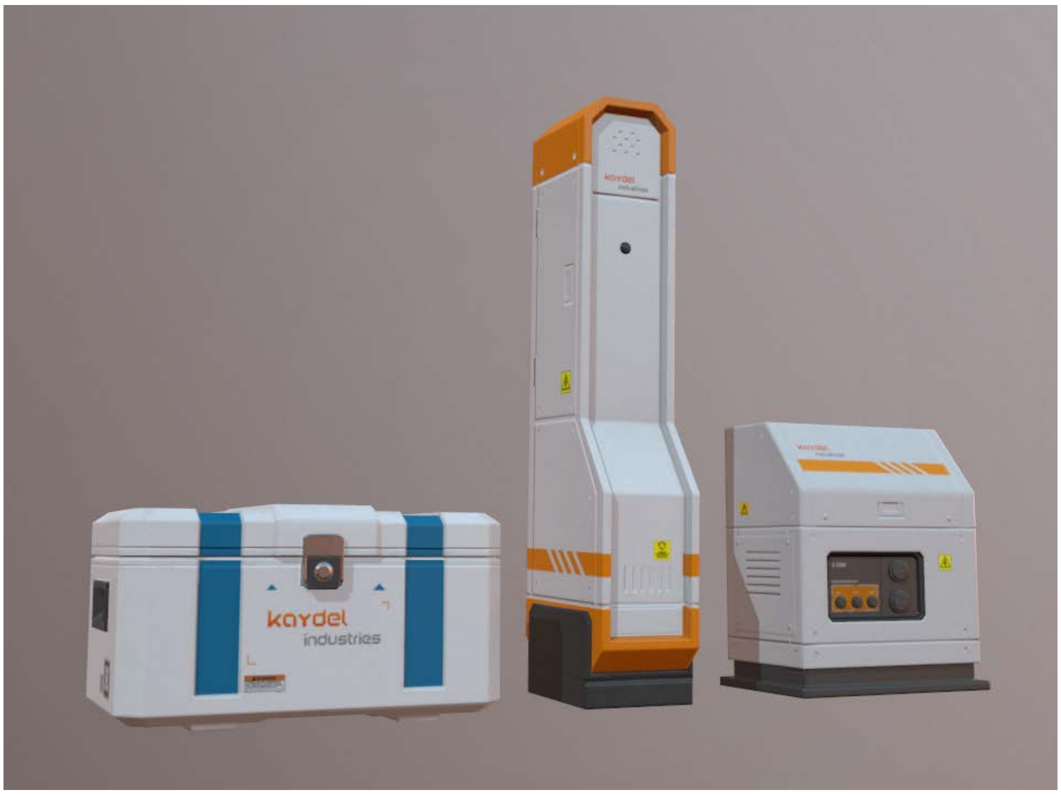

Figure 22. Some of the assets rendered in Marmoset Toolbag 2.

Again we possibly could have sped up the development by using Quixel suite or Substance designer to their full potential instead of relying to more a traditional Photoshop texturing workflow, but this would have required prior knowledge of these software.

7.3 Outcome

Overall the project was a success as all of the goals set in the original plan were met. The outcome matches the original concept we had and we were able to produce all of the assets in time. Even though some of the assets could have needed a bit more time the result is acceptable as a whole. In the end there are 74 individual assets and around 190 texture maps in the scene. All of the assets sit rather well together and the scene colors and materials work well. There are some minor issues with the shadows that we could have possibly fixed with having uncompressed or just bigger resolution lightmaps. The fact that we only had two artists instead of four is somewhat visible, as we did not quite have the time to author some of the texture maps, especially the roughness maps, to the level we would have wanted. The scene also lacks some sort of a focal point and this is mainly due to the fact that the original concept was too hastily done. A view from the side of the room can be seen in the Figure 23 below. If the time frame would have been longer we would have been able to create a fairly larger environment as most of the walls and other building block assets are created in a modular fashion. The original asset lists did not have any blueprint systems except the actual controls for the first person camera, but still we snuck in few extra to the final product. The clock on the wall moves in real-time and the light shafts from the window are also animated using blueprint systems.

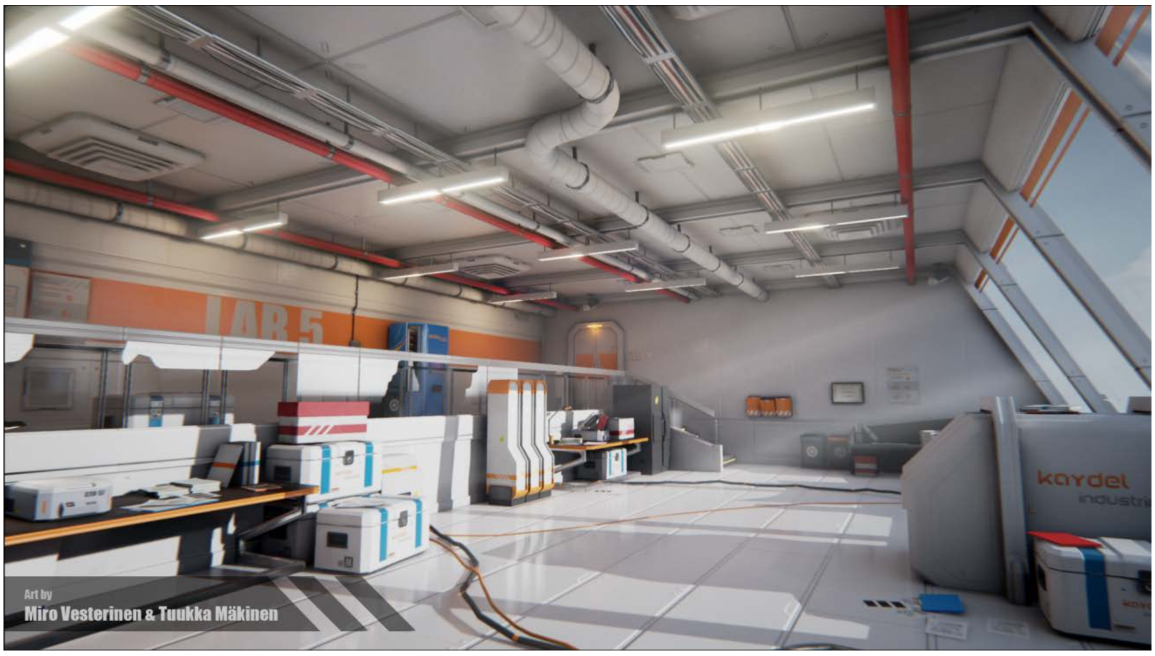

Figure 23. Futuristic office environment.

As we used an early version of the Unreal Engine we did not have access to skylight global illumination or some of the newer shadow features and updates that would have given the overall lighting a bit more natural look and increased the quality of the baked lightmaps. Also the emissive materials did not work as a light source yet. We also produced all of the lightmap UVs by hand as Unreal Engine 4 did not yet have a proper automatic lightmap UV generation. This workflow produced good results, but was rather time consuming. In a nutshell the development could have been faster and easier, but the project taught us a lot as we learned from our mistakes.

8 CONCLUSION

3D-graphics, a vast subject that combines both technical knowledge and artistic vision, is currently used at almost every major game release and game studios invest huge amounts of time and money to strive for photorealistic rendering results. Some of these new technologies have become new standards in real-time rendering and artists need to shape their workflows for content creation to match these new requirements.

Creating a 3D-game environment, from initial concept to final product, requires both time and technical knowledge. 3D-artists need to make sure that their workflows and software used are suitable to achieve results that are both correct and efficient. A proper planning combined with comprehensive technical knowledge can shrink both production time and load, making development more efficient. As new software and technologies emerge in a steady stream, artists need to be able to adapt and possibly re-learn outdated working methods.

REFERENCES

Adobe. 2014. 3D texture editing. http://helpx.adobe.com/photoshop/using/3d-texture-editing.html (read 21.11.2014).

Allegorithmic. 2013a. Substance Designer. http://www.allegorithmic.com/products/substance-designer (read 21.11.2014).

Allegorithmic. 2013b. Substance Painter. http://www.allegorithmic.com/products/substance-painter (read 21.11.2014).

Autodesk. n.d. The Generation and Display of Normal Maps in 3ds Max. http://area.autodesk.com/userdata/fckdata/239955/The%20Generation%20and%20Display%20of%20Normal%20Maps%20in%203ds%20Max.pdf (read 21.11.2014).

Clark, Ryan. 2009. Overlaying normals in UT3 material editor? http://www.polycount.com/forum/showpost.php?p=1011754&postcount=5(read 21.11.2014)

CLO Virtual Fashion. 2014. Marvelous Designer. http://www.marvelousdesigner.com/ (read 21.11.2014).

Crytek. 2014a, Getting Started Modeling. http://docs.cryengine.com/display/SDKDOC2/Getting+Started+Modeling (read 21.11.2014).

Crytek. 2014b. Creating Armor Assets. http://docs.cryengine.com/display/SDKDOC2/Creating+Armor+Assets (read 21.11.2014).

Crytek. 2013a, Gamma-Correct Rendering (sRGB). http://docs.cryengine.com/pages/viewpage.action?pageId=1605651 (read 21.11.2014).

Crytek. 2013b, Tangent Space Normal Mapping. http://docs.cryengine.com/display/SDKDOC4/Tangent+Space+Normal+Mapping (read 21.11.2014).

Digital-Tutors. 2014. Key 3D modeling Terms Beginners need to Master. http://go.digitaltutors.com/essential-modeling-term (read 21.11.2014)

Dontnod Entertainment. 2014. Physically based rendering chart for Unreal Engine 4. http://seblagarde.wordpress.com/2014/04/14/dontnod-physically-based-rendering-chart-for-unreal-engine-4/ (read 21.11.2014).

Epic. 2014a. Physically Based Materials. https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/PhysicallyBased/index.html (read 21.11.2014).

Epic. 2014b. Textures. https://docs.unrealengine.com/latest/INT/Engine/Content/Types/Textures/index.html (read 21.11.2014).

Epic. 2014c. Unreal Engine 4 Features. https://www.unrealengine.com/products/unreal-engine-4 (read 21.11.2014).

Epic. 2014d. Physically based shading in UE4. https://www.unrealengine.com/blog/physically-based-shading-in-ue4 (read 21.11.2014).

Epic. 2014e. Essential Material Concepts. https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/IntroductionToMaterials/index.html (read 21.11.2014).

Epic. 2014f. Lightmass Global Illumination. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/index.html (read 21.11.2014).

Epic. 2014g. Collision Responses. https://docs.unrealengine.com/latest/INT/Engine/Physics/Collision/index.html (read 21.11.2014).

Epic. 2014h. Blueprint overview. https://docs.unrealengine.com/latest/INT/Engine/Blueprints/Overview/index.html (read 21.11.2014).

Epic. 2014i. Texture Support and Settings. https://docs.unrealengine.com/latest/INT/Engine/Content/Types/Textures/SupportAndSettings/index.html (read 21.11.2014).

Epic. 2014j. Texturing. https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/Functions/Reference/Texturing/index.html (read 21.11.2014).

Epic. 2014k. Ray Trayced Distance Field Soft Shadows. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/RayTracedDistanceFieldShadowing/index.html (read 21.11.2014).

Epic. 2014l. Instanced Materials. https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/MaterialInstances/index.html (read 21.11.2014).

Epic. 2014m. Layered Materials. https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/LayeredMaterials/index.html (read 21.11.2014).

Epic. 2014n. Unwrapping UVs for Lightmaps. https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/LightmapUnwrapping/index.html (read 21.11.2014).

Epic. 2014o. Ambient Occlusion. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/AmbientOcclusion/index.html (read 21.11.2014).

Epic. 2014p. Distance Field Ambient Occlusion. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/DistanceFieldAmbientOcclusion/index.html (read 21.11.2014).

Epic. 2014q. Texture Properties. https://docs.unrealengine.com/latest/INT/Engine/Content/Types/Textures/Properties/index.html (read 21.11.2014).

Epic. 2012a. Importin Skeletal Mesh Tutorial. http://udn.epicgames.com/Three/ImportingSkeletalMeshTutorial.html (read 21.11.2014).

Epic. 2012b. Deferred Shading in DirectX 11. http://udn.epicgames.com/Three/DeferredShadingDX11.html (read 21.11.2014).

Haneman, Lee. 2013. Game Development - The Texture Guide. http://www.unorthodoxentertainment.com/game-development-texture-guide/ (read 21.11.2014).

Gritz, Larry. d'Eon Eugene. 2008. The importance of being Linear. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html (read 21.11.2014).

IDV. 2014. SpeedTree. http://www.speedtree.com/ (read 21.11.2014).

Karis, Brian. 2013. Real Shading in Unreal Engine 4. https://de45xmedrsdbp.cloudfront.net/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf (read 21.11.2014).

Pixologic. 2014. Zbrush. http://pixologic.com/zbrush/features/overview/ (read 21.11.2014).

Polycount. 2014a. Normal map. http://wiki.polycount.com/wiki/Normal_map (read 21.11.2014).

Polycount. 2014b. Rendering. http://wiki.polycount.com/wiki/Category:Rendering (read 21.11.2014).

Polycount. 2014c. UE4 FBX Importer. http://www.polycount.com/forum/showthread.php?t=141659 (read 21.11.2014).

Quixel. n.d. Quixel Suite. http://quixel.se/ (read 21.11.2014).

Russel, Jeff. 2014. Basic Theory of Physically-based Rendering. http://www.marmoset.co/toolbag/learn/pbr-theory (read 21.11.2014).

Santiago, Orgaz. 2014. Xnormal. http://www.xnormal.net/1.aspx (read 21.11.2014).

Tech-Artists. 2008. Texture coordinate. http://tech-artists.org/wiki/Texture_coordinate (read 21.11.2014).

Vromman Joeri. 2014. Kel-Tec KSG. http://www.joerivromman.com/KSG.html (read 21.11.2014)

Ward, Jeff. 2008. What is a Game Engine? http://www.gamecareerguide.com/features/529/what_is_a_game_.php (read 21.11.2014)

Wilson, Joe. 2014. Tutorial: Physically Based Rendering, And You Can Too! http://www.marmoset.co/toolbag/learn/pbr-practice (read 21.11.2014).

World Machine Software. 2014. World Machine. http://www.world-machine.com/ (read 21.11.2014)