Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

# Creating Realistic Environment using Photogrammetry

Bachelor Thesis

Daniel Hanák

Field of study: Computer Games and Graphics

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

May 2019

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hanák** Jméno: **Daniel** Osobní číslo: **467182**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Tvorba realistického prostředí pomocí fotogrametrie**

Název bakalářské práce anglicky:

**Creating Realistic Environments using Photogrammetry**

Pokyny pro vypracování:

Zmapujte metody tvorby realistických prostředí pro počítačové hry a dostupné související softwarové nástroje. Na základě provedené analýzy navrhněte postup pro vytvoření realistického modelu přírodního prostředí, konkrétně prostředí lesa. Zaměřte se na metody využívající fotogrametrie pro získání detailního geometrického modelu scény a jejích částí. Pomocí navržené metodiky vytvořte nejméně pět různých modelů stromů, vegetace a podloží, které poskytnou základ pro následnou kompozici celé scény. Modely zpracujte do formy, která umožní jejich zobrazování v reálném čase s využitím normálových map a textur definujících parametry PBR shaderů. Kde je to možné použijte pro získání výsledných modelů open-source nástroje. Popište vytvořený řetězec včetně všech použitých nástrojů a analyzujte jeho silné a slabé stránky. Ze získaných modelů vytvořte komponovanou scénu o rozloze nejméně čtyř čtverečních kilometrů. Důkladně otestujte zobrazování vytvořené scény z hlediska vizuální kvality a rychlosti zobrazování v herním enginu Unreal.

Seznam doporučené literatury:

[1] Snavely, Noah, Steven M. Seitz, and Richard Szeliski. 'Modeling the world from internet photo collections.' International journal of computer vision 80.2 (2008): 189-210.
[2] Lagarde, S., and C. D. Rousiers. 'Moving frostbite to physically based rendering.' SIGGRAPH 2014 Conference, Vancouver. 2014.
[3] McDermott, W. M. 'The Comprehensive PBR Guide by Allegorithmic, vol. 1.”
https://www.allegorithmic.com/system/files/software/download/build/PBR_Guide_Vol.1.pdf. 2015.
[4] J. L. Schönberger and J. Frahm, 'Structure-from-Motion Revisited,' 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 4104-4113.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

| | | |
|---|---|---|
| doc. Ing. Jiří Bittner, Ph.D.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Ing. Pavel Ripka, CSc.<br>podpis děkana(ky) |

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

| | |
|---|---|
| . | |
| Datum převzetí zadání | Podpis studenta |

## Acknowledgements

I would like to express my gratitude to my supervisor Jiří Bittner, associate professor in the Department of Computer Graphics and Interaction, for his useful advice, comments, and remarks. Furthermore, I would like to thank my parents for all the support, emotional and financial, they provided me with.

## Declaration

I hereby declare I have written this thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

Prague, May 20, 2019

## Abstract

When creating a realistic environment for games, we want to achieve maximum credibility of our scene. Simultaneously we want to render everything in real-time. Creating realistic vegetation, trees, and surfaces may be a rather challenging and time-consuming problem. This thesis gives an overview of the 3D computer modeling workflow with the integration of photogrammetry.

We have implemented a workflow of creating a realistic environment with photogrammetry. For testing purposes, we have created a coniferous forest scene, and we have presented results of benchmark evaluated in Unreal Engine.

**Keywords:** Game environment, Photogrammetry, Unreal Engine, Realistic forest, Computer graphics

**Supervisor:**
doc. Ing. Jiří Bittner, Ph.D.
DCGI, FEE, CTU in Prague
Praha 2, Karlovo náměstí 13

## Abstrakt

Při tvorbě realistických prostředí do her chceme dosáhnout maximální věrohodnosti naší scény a zároveň chceme, aby se vše vykreslovalo v reálném čase. Tvorba realistických porostů, stromů a povrchů může být poměrně složitý a časově náročný problém. Tato práce dává čtenáři přehled o 3D počítačovém modelování s využitím fotogrammetrie.

Implementovali jsme postup vytvoření realistického prostředí pomocí fotogrammetrie. Pro testování jsme vytvořili scénu jehličnatého lesa a prezentujeme vý-sledky zátěžových testů vyhodnocené v Unreal Engine.

**Klíčová slova:** Herní prostředí, Fotogrammetrie, Unreal Engine, Realistický les, Počítačová grafika

**Překlad názvu:** Tvorba realistického prostředí pomocí fotogrammetrie

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

Computer graphics in games is constantly improving. Nowadays AAA[1] games already have large realistic environments that are close to reality. Creating a realistic environment for games with maximum credibility and simultaneously rendering in real-time is a difficult task. A common approach to make a three-dimensional (3D) model for a game is to create a low poly mesh and a high poly mesh in modeling software, mesh parameterization, generating texture maps from the high poly mesh (also called texture baking) and creating textures for the low poly mesh. Creating the low poly mesh usually starts with a simple primitive, then complexity is added until the desired form is reached. Modeling polygonal objects is generally done by technical, environment, and character 3D artists. This standard computer modeling workflow and tools for games are expanding and rapidly changing.

Modern approach is algorithmic creation of game content with limited or indirect user input (procedural generation method) [29]. The main advantage of procedural generation methods is the removal of the need to have a human designer or artist creating specific game content. Game content in our definition is everything, what is contained within a game: landscapes, levels, textures, vegetation, quests, music, weapons, characters, etc [29]. If we need to create a huge number of objects that are similar, but also slightly different from each other, we should not make them manually. This task can be time-consuming, on the other hand, just copy-pasting objects will not deal with it. Having too many identical objects may be noticeable for the player. The procedural generation method is a suitable approach for this task. When creating a realistic environment for games, procedurally generated content such as landscapes, treetops, music or procedural placement of objects can be very helpful. For example, sandbox video game Minecraft from 2011 generates the whole world procedurally; every

[1]A triple-A video game (AAA) is generally a title developed by a large studio that offers more content and a realistic graphical style.

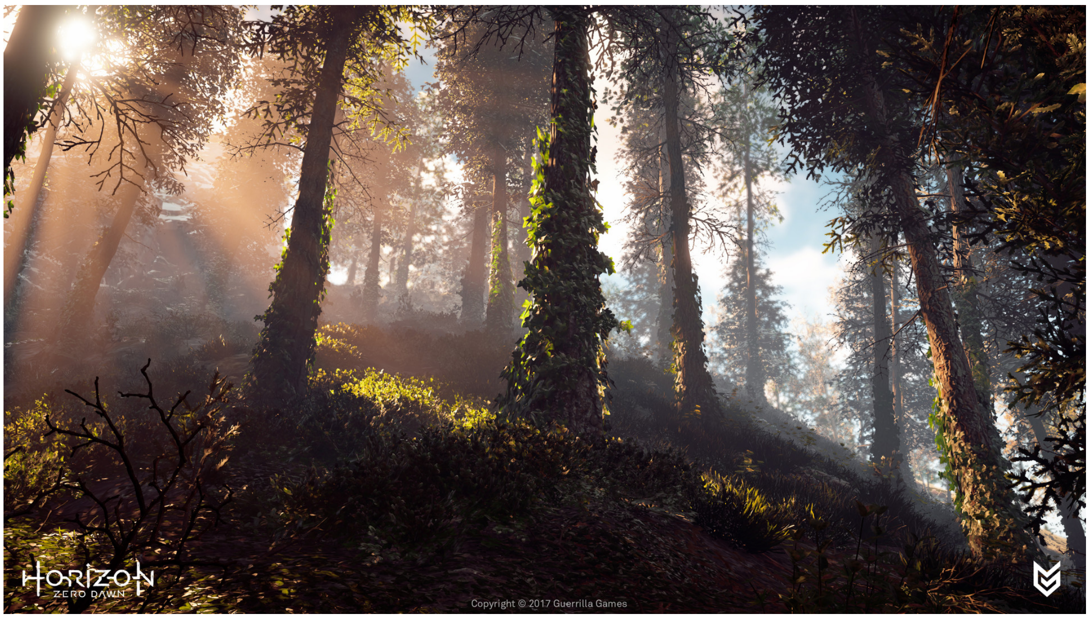


**Figure 1.1:** Horizon: Zero Dawn has a massive open world with realistic vegetation which has been created by hand and procedurally [26]. (Image courtesy of Sony Interactive Entertainment Europe. Game developed by Guerrilla.)

generated world has different environments composed of different biomes. Horizon: Zero Dawn is another game that uses a procedural placement system that dynamically assembles the game world with vegetation, sounds, effects, wildlife, and game-play elements [31]. A shot from the game with a subalpine forest is shown in Figure 1.1.

During the past few years, some game studios have used photogrammetry method for creating realistic materials and objects. Ubisoft Entertainment has used photogrammetry for recreating Montana [15] for Far Cry 5, BioWare has utilized this method for creating advanced textures such as molten lava for their game Anthem [11], EA Digital Illusions CE AB (DICE) has first used this method on Battlefield, and they fully embraced the technology and workflow for Star Wars: Battlefront [7]. Photogrammetry has started to gain popularity within the game industry and continuously push the boundaries of creating a realistic environment for games.

In this thesis, we will cover computer graphics for games, workflow for creating 3D game assets[2], basics of real-time rendering pipeline, and physically based rendering. Then we will analyze the photogrammetry method, integrate this method into the 3D asset creating workflow, and use it for creating game ready assets. Furthermore, we will go through the implementation of the game environment in Unreal Engine. Finally, we will deploy a simple game and test it on a set of different hardware.

[2]Assets in the video game industry are elements (2D and 3D objects, textures, scripts, audio, etc.) from which a video game can be built.

# Chapter 2

# Computer Graphics for Games

## 2.1 Game Asset Creation

The exact process of game asset creation varies from game to game. The whole process is specialized in stages. Every stage involves people with different skill sets working on each asset at various times. This section covers the 3D game asset creation process and the fundamentals of computer graphics for games.

### 2.1.1 Concept Art

Everything that will populate our game, such as characters, environments or props should be drawn on paper. Concept art is used to convey our ideas and settle the visual style of our game content. The reason is to make sure that all the game assets that we bring to life belong to the same stylization. It is developed through several iterations with multiple variants to pick the best design we want.

### 2.1.2 Geometry Modeling

The standard 3D game asset creation process starts by modeling the object geometry according to concept art. The fundamental building block of object geometry is a polygon. The inside area of a polygon is called a face, the sides which bound the face are called edges and points where two edges meet are vertices [10]. A group of polygons that forms a model is called a polygon mesh. Typically, the polygon mesh is created in two versions. The low poly mesh will be used in the game engine for the representation of the object's geometry and the high poly mesh which will be used for baking details into a parameterized low poly mesh. Let us divide object geometry creation into three groups: standard polygon modeling (also called box modeling), digital sculpting and procedural modeling.

Standard polygon modeling is performed in a general 3D modeling software such as Autodesk 3D studio Max, Maya, or Foundry Modo. Artists can scale the object, translate vertices, edges, and add more faces by cutting into geometry. Vertices can be welded together, broken apart, and aligned. Edge functions include chamfer and bridge. Face functions include extrude, bevel, bridge, inset, outline, etc [2]. More complex functions such as Boolean operations allow intersection, difference, and union of geometry; we can also edit geometry with modifiers which transform our geometry.

Creating creatures or characters with small details is difficult with box modeling technique. Digital sculpting is preferred for achieving photorealistic results, mainly used for high poly organic modeling and hard surface modeling. Industry standard software for digital sculpting is Pixelogic Zbrush. Digital sculpting works like sculpting with real clay; artists use brushes and tools that push, smooth, pinch and pull. The 3D asset is typically created in a multi-layer process. The first layer defines a basic shape and silhouette of our geometry. More detail is added after subdividing the geometry in higher layers. Digital sculpting uses a lot of resources, tools, and plugins which help to create complex structures. Everything you can imagine can be digitally sculpted from scratch.

Node-based approach for creating geometry is called procedural modeling. Nodes represent specific functions with inputs and outputs. Wired node network defines a recipe for creating tweakable and unique results. Fully procedural software for game development, motion graphics, film, and virtual reality is SideFX Houdini which is used for procedural modeling, terrain generation, animation, rigging, physical simulations, hair and cloth generation, etc. For example, Ubisoft Entertainment used Houdini for automatization of the world generation in Far Cry 5 [9, 20] . Another example is Manhattan from Marvel's Spider-Man game which was also created with procedural techniques [27]. More specialized software like SpeedTree is used as middleware for generating procedural vegetation.

### 2.1.3 Geometry Parameterization

Process of projecting a 3D geometry into a 2D plane is called UV parameterization. UV coordinates are generated for each vertex in the low poly mesh. We can match the UV coordinates to points on a texture map; these points are called texture coordinates [10]. Parameterization starts with a seam definition, edges selected as seams will split polygons apart. Polygons on the 2D plane are unfolded, normalized and arranged, so they fill the available texture space as densely as possible. A checkered texture is used to visually track any distortion which can cause problems with textures later. UV parameterization can be done in a general 3D software or a specialized software such as Headus UV Layout or RizomUV Virtual Spaces.

### 2.1.4 Texturing

In the game industry, we usually use the term texture to refer to a 2D static image, applied to a mesh [1]. Textures add a variety of different types of information and detail to our model without adding to the polygon count.

One of the most noticeable attributes of any surface is its color. Diffuse reflected color represents a diffuse map (also called an albedo map) in RGB space [21]. An example of the diffuse map applied to a mesh is shown in Figure 2.1. Projected details of a high poly mesh to a low poly mesh represent a normal map in RGB space. Normal maps are commonly used in video games to simulate surface details. Each channel of the normal map corresponds to the $x$, $y$ and $z$ components of the surface normal [21].

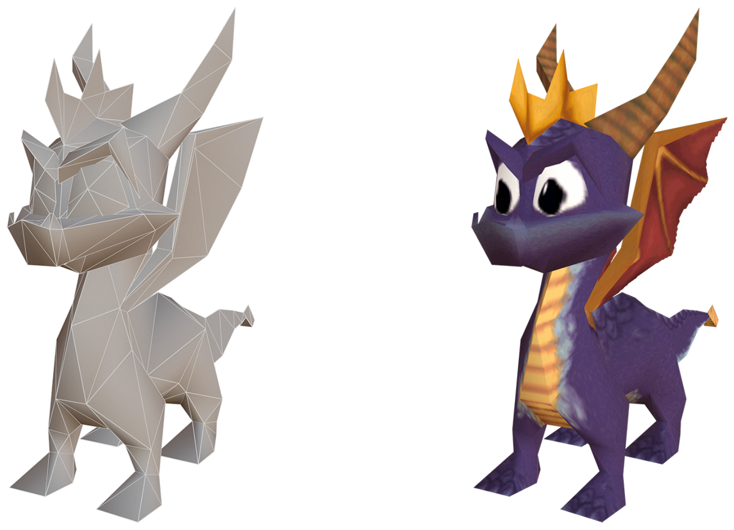


**Figure 2.1:** The main character from Spyro the Dragon, game released in 1998. On the left is mesh geometry with a wireframe of about 400 triangles. On the right, a mesh with textures. (Image courtesy of Activision Publishing, Inc. Game developed by Insomniac Games, Inc.)

Let us assume that we have battle-scarred metal armor with a layer of old paint and dust that will not be reflective and areas where this layer has been worn away to reveal reflective metal. The reflectance values are defined in a specular map [21]. It controls what parts of the surface are shiny or not based on the grayscale values of the map. Specular maps are commonly created from the diffuse maps [1]. The surface inconsistency that causes light diffusion is stored in a glossiness map [21]. It is a grayscale map, where white color represents a smooth surface, and black color represents a rough surface. The glossiness map defines the size of the specular highlight, and the specular map controls the intensity of the highlight on the surface. These two maps represent old specular/glossiness texturing workflow; we cover new physically based rendering (PBR) workflow in section 2.3.

If we want to change the height of the surface along the normal, we can use a height map which is used to displace the actual geometry. It is a grayscale map, where white color represents positive displace, and black color represents negative displace along the normal. Exactly grey color (value of 128) will not affect our geometry.

Not every piece of geometry is opaque. Opacity maps (also called transparency or alpha maps) are grayscale images used to change the transparency of textures. We can use this map to make part of our model completely invisible or partially transparent. Opacity map can be used on simple planes for creating leaves for plant models, fences, windows, or for in-game effects such as explosions, smoke, and fire. The trade-off is that transparency is commonly used on surfaces with low polygon count, but it requires more processing power [1].

The last texture we are going to discuss is an ambient occlusion map. It defines how much of the ambient environment lighting is accessible to a surface point [21]. This map is generally used to create a more natural and realistic look.

Textures are commonly created by texture artists. They can be hand-painted or created from photographs in a general 2D software such as Adobe Photoshop. Nowadays AAA games use procedurally generated textures; the industry-standard software for creating procedural textures is Allegorithmic Substance Designer. Artists can create materials with full control and infinite variations. Another approach for texturing 3D assets is texturing directly in 3D; this can be done in software such as Allegorithmic Substance Painter.

### 2.1.5 Rigging

Before animating our object, we are going to create controls, constraints, and connections for our model to simplify animation. This process is called rigging. The most common type of deformers used to rig and manipulate geometry are joints. Geometry can be connected to joints via parenting, constraining, and skinning. The model with assigned joints can be deformed in many ways depending on our parameters [5]. Parenting (also called hierarchy) is a grouping of objects into parent and child nodes. Any transformation applied to the parent node will affect the child node, but the child node is still controlled independently. Joints are created singularly or connected in hierarchical chains; connections between two joints are commonly represented as bones. After creating a skeleton, the mesh is attached to it. This process is called skinning [5]. For each vertex of each joint on the mesh, there is a weight parameter which defines, how the mesh is goint to be affected. Rigging is a complex and challenging task that needs custom tools to automate and simplify the whole rigging process.

### 2.1.6 Animation

The animation process is where our assets come to life. When we are in a game, we are usually moving around the world, and we interact with things or other players. All movements and interactions have specific animations. Animations are commonly created in cycles which are used in combinations. For example, a breathing cycle will be used with idle, walking, and climbing animation. Many animations are driven by the real movement of animals and people. Making realistic animations by recording the movement of objects or people is called motion capture. The capture subject has marks that are tracked by calibrated cameras. Any performance done by the subject is recorded and processed into actual animations that can be used in games. The animations are done; we need to define how we want to apply them in the world. Nowadays, games use animation systems driven by artificial intelligence (AI); for example, developers from id Software created an advanced animation system with full-body animations driven by AI for their game DOOM [8]. A screenshot from the game is shown in Figure 2.2.

**Figure 2.2:** A shot from first-person shooter game DOOM. (Image courtesy of Bethesda Softworks® LLC. Game developed by id Software, Inc.)

## 2.2 Real-time Rendering Pipeline

Let us assume that we have created our 3D environment for our game. How do game engines render our scene to a 2D screen? A conceptual model which describes the sequence of operations that is required to render a 3D scene to a 2D screen is called graphics rendering pipeline. The shapes and locations of our objects from the scene in the rendered 2D image are determined by their geometry and placement of a camera. The visual representation of the objects is affected by light sources, materials, textures, and shading equations. Engine core of the rendering pipeline which is used in real-time computer graphics applications commonly uses the following stages: application, geometry processing, rasterization, and pixel processing. [3].

### 2.2.1 Geometry Processing

The application stage defines which primitives (points, lines, and triangles) should be rendered and sends them into the geometry processing stage. We can divide the geometry processing stage into vertex shading, projection, clipping, and screen mapping substage [3]. In the vertex shading substage the main task is to compute the position for a vertex and define its ouput data such as normal and texture coordinates. The projection substage transforms a view frustum into a unit cube. Commonly used projections are orthographic and perspective. Optional vertex processing tasks that can be executed are tessellation, geometry shading, and stream output [3]. Tessellation converts sets of vertices into bigger sets of vertices that are used to create new sets of triangles. Geometry shader converts vertices into complex structures: triangles, squares made of triangles, etc. The last optional task stream output puts vertices to an array for further processing instead of sending them to the rest of the pipeline.

The clipping substage defines which primitives should be sent to the rest of the pipeline. Primitives that are entirely outside of the view frustum are not rendered and are discarded. Primitives that are partially inside of the view frustum are clipped. A vertex that is outside of the view frustum is replaced by a new one that is in the intersection between the corresponding edge and the view frustum [3]. Primitives that are fully inside of the view frustum are kept. Primitives passed from the clipping substage into the screen mapping substage are still in 3D. Mapping substage transforms each primitive coordinates into screen coordinates.

### 2.2.2 Rasterization

The task for the rasterization stage is to find all pixels on the screen that are inside primitives (triangles) which are being rendered. We can split the rasterization into two substages: triangle setup and triangle traversal [3]. One way of determining if the triangle overlaps the pixel is a single point sampling method. All pixels have a single sample assigned in the middle of them, and if the sample point is inside the triangle, the corresponding pixel is assigned to the triangle. In the triangle setup substage, there are computed data for interpolation of the various shading data, produced by the geometry stage. They are passed to the triangle traversal substage, where are generated fragments for each pixel assigned to the triangle [3].

### 2.2.3 Pixel Processing

Any desired computation defined in a pixel shader program[1] can be done here. Per-pixel shading computations use the interpolated shading data as input. The output of this stage is color value for each fragment defined in the rasterization stage [3]. One crucial task here is defining colors based on input textures. The computed color value is passed to a color buffer. When all colors are computed, they are displayed on the screen.

## 2.3 Physically Based Rendering

Physically based materials are widespread and present in all modern game engines such as Unreal Engine or Unity. These materials use physically based rendering (PBR) which is a rendering and shading technique that provides an accurate representation of how light interacts with surfaces [21]. However, what does physically based actually mean? It derives from light-matter interaction, follows basic physical rules, e.g., energy conservation, has representation with metrics in the real world and separates lighting and material. Our goal with this technique is to simulate photorealism.

### 2.3.1 Diffuse and Specular Reflection

The fundamental building block of PBR is how light interacts with objects. When a light ray hits a surface, it can be reflected off the surface, absorbed by the material and scattered internally. Absorbed light changes into another form of energy which

[1]Pixel shaders are programmable blocks which compute colors for each fragment.

causes intensity decreasing. Scattered light does not change its intensity, but it changes the direction based on the material. Specularly reflected light also does not change its intensity, but it travels in a different direction based on the surface irregularity. Diffuse reflection happens when the light ray is refracted, scattered multiple times, and reflected out of the object [21]. This process is illustrated in Figure 2.3.

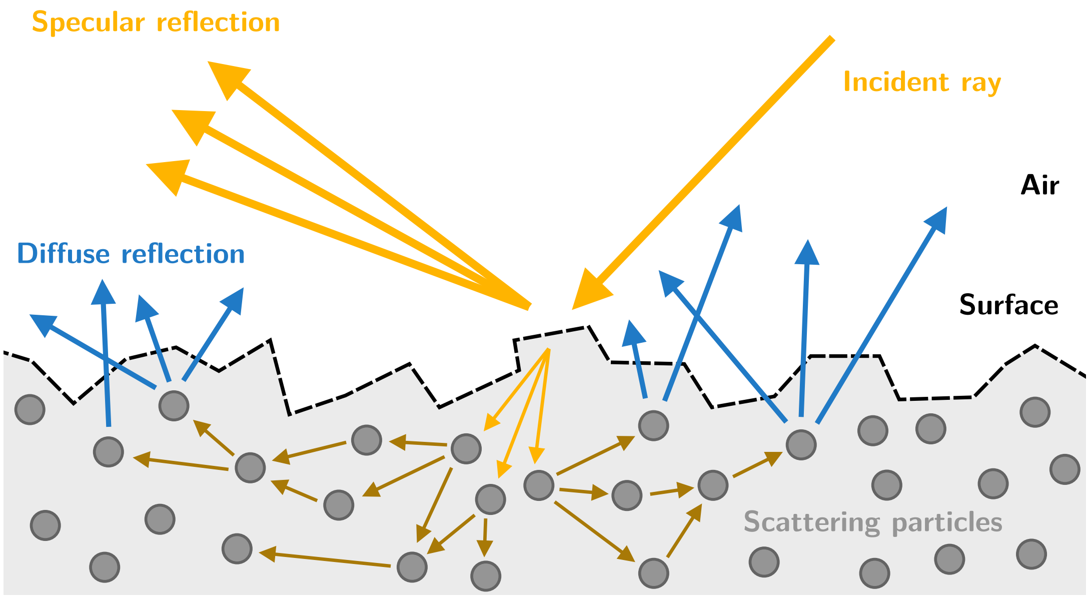


**Figure 2.3:** The refracted light travels through the material and scatters from particles inside the material. Some refracted light is scattered out of the surface in various directions.

The index of refraction (IOR) describes the change in the direction a light ray is traveling when it passes through one material to another. IOR is measured from real-world data. Surface color is determined by the scattering and absorption which is different for different wavelengths of light. If a surface scatters a red wavelength and absorbs the rest, it will appear red.

### 2.3.2 Bidirectional Reflectance Distribution Function

Reflectance properties of a surface are described in the bidirectional reflectance distribution function (BRDF). It meets energy conservation which states the total amount of light reflected and scattered back by a surface is less than the total amount of light received [21]. Another coefficient of BRDF is a Fresnel reflection factor which states the amount of light reflected from a surface depends on the viewing angle at which it is perceived. Energy conversation and Fresnel effect are aspects of physics that are both handled by the PBR shader.

The scattering factor of the reflected light depends on the structure of the microsurface described by surface roughness. Common scattering factors depend on additional surface layers such as dirt, dust, and physical damage. The last factor is the definition of a metallic surface. Metals are good conductors of heat and electricity; they contain many free electrons that partially reflect the light and absorb all the refracted light. Non-metals are poor conductors of electricity. They reflect a smaller amount of light than metals, so they reflect mostly diffuse color [21].

### 2.3.3 Physically Based Materials

We will go through the two most common PBR texturing workflows, which are metal/roughness and specular/glossiness, for physically based material definition. The PBR shader uses standard maps for both of them; these maps are normal map, ambient occlusion map, and height map.

The metal/roughness workflow is determined through a set of textures, maps specific to this workflow are base color, metallic, and roughness. The base color texture is a map in RGB space that contains diffuse reflected color for dielectrics (reflected wavelengths) and reflectance values for metals. If a surface is metallic or not is described in the metallic texture. It acts like a standard mask that tells the shader how to interpret data found in the base color texture. The metallic texture is a grayscale map, where white colors represent raw metal and define base color texture as metal reflectance; black colors represent dielectric and define base color texture as reflected. The base color texture values defined as metallic should be obtained from real-world measured values [21]. The surface irregularities that cause light diffusion are defined in the roughness texture. It is a grayscale map, where white colors represent the rough surface and create more significant and dimmer-looking highlights; black colors represent the smooth surface and create a brighter reflected light.

The specular/glossiness workflow is also determined through a set of textures but uses different maps. Specific textures for this workflow are diffuse, specular, and glossiness. The diffuse map contains only diffuse reflected color; it does not have any reflectance values. The surface areas with raw metal have a black color defined in diffuse texture, as metal does not have a diffuse color. The specular map represents the reflectance values for metallic surfaces and the Fresnel reflectance at zero degrees (F0) for dielectric surfaces. The F0 value should be obtained from real-world measured data [21]. The surface irregularities that cause light diffusion are defined in the glossiness texture. It is the inverse of the roughness texture from the metal/roughness workflow, so it is also a grayscale map, where white colors represent the smooth surface and create a brighter reflected light, black colors represent a rough surface and create larger and dimmer-looking highlights [21].

# Chapter 3

# Photogrammetry for Games

Photogrammetry is the science of extracting reliable measurements from images, in our case, for generating textures and geometry. This field of study connects many other fields and disciplines such as computer vision, mathematics, optics, and projective geometry. We focus on 3D meshes generated from a workflow utilizing a technique known as stereophotogrammetry which compares multiple images taken from different positions for estimating object coordinates in 3D space [15].

**Figure 3.1:** Far Cry 5 is an action-adventure first-person shooter game, set in an open world environment that is created with the photogrammetry method. (Image courtesy of Ubisoft Entertainment.)

Generating 3D assets with photogrammetry has existed in many forms within other fields and disciplines. Within the entertainment and game industry, the technology is relatively new, but a few games already have assets created with photogrammetry. An art team from Ubisoft Entertainment took a series of trips to Montana, during which the developers got to know Montana's ecosystem, wildlife, and people, and took thousands of photos. They were able to re-create the whole environment with realistic soils, trees, vegetation, and buildings [24]. A shot from a fictional country in the United States from Far Cry 5 is shown in Figure 3.1. Another example is Star Wars: Battlefront; developers from DICE visited real film locations such as Iceland and California's national redwood forests [7]. They captured complete asset libraries for all Star Wars planets in outstanding quality and automated the whole process of creating assets with the photogrammetry method. An example of their result is shown in Figure 3.2. This method can be even used for creating more complex objects such as cars and trucks. SCS Software company sent a small technical team to the Scania Demo Centre in Sweden, to acquire detailed reference material for Scania's new S and R trucks for their game Euro Truck Simulator 2 [28].

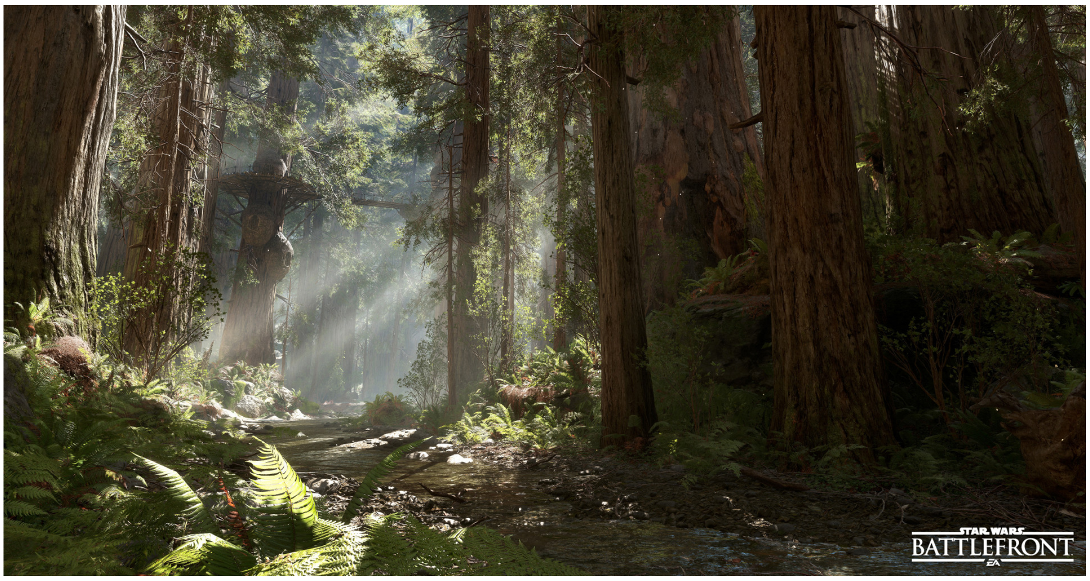


**Figure 3.2:** A shot from Star Wars: Battlefront. Endor environment was created from photographs taken in California's national redwood forests. (Image courtesy of Electronic Arts Inc. Game developed by EA Digital Illusions CE AB.)

One of the most exciting tasks in game development is creating game assets and game environments. Every 3D game asset can be digitally sculpted or created procedurally, so why should we use the photogrammetry method? This method brings a realistic visual appearance based on real-world data and reduces creating time [6]. The standard process of creating game assets with photogrammetry is divided into the following stages: taking a series of photographs, aligning photographs in photogrammetry software, generating a high-resolution mesh with color information, mesh simplification, mesh retopology, mesh parameterization, and generating (baking) textures. This process seems to be simple. We are going to find out, it is not.

## 3.1 Capturing Photographs

For taking photographs, we need a camera. Technically, any camera can be used, such as a digital single-lens reflex camera (DSLR), a smartphone or a drone, but the quality of our asset depends on the taken photographs. We need a camera that allows us to use manual control, a camera histogram, and shoots in RAW[1]. The lenses for our camera should have focal lengths from 24 mm to 70 mm; we avoid ultra-wide angle and fish-eye lenses. A polarisation filter that removes unwanted reflections by selecting which light rays enter the lens can be very handy. We can use tripods and monopods for stabilizing the camera; they can be beneficial with limited lighting conditions. Our last gear for photography pre-processing and calibration is a color checker. We will use it for setting correct white balance and photography calibration to get the most accurate colors.

### 3.1.1 Research and Development

The first step before organizing a photogrammetry adventure is to find references for game assets that we want to create. Not every asset can be created with this method; it is necessary to be familiar with the photogrammetry algorithm to get the best possible results. The algorithm hunts for parallax shifts between features within multiple photographs [15]. Recreating assets need to have different features, objects without structure or objects with repetitive patches can cause the photogrammetry algorithm to fail when trying to match the feature sets.

Capturing surface should be consistent, we are unable to create transparent, reflective and moving objects such as wet surfaces, glass, metallic surfaces, and vegetation in the wind. We will take photographs from different angles, but reflective surfaces appear differently from different view angles; this can cause similar confusion for the photogrammetry algorithm. It works very well with rough and static objects.

When is the best time to go on the photogrammetry adventure? The best lighting conditions for our photographing are during overcast days with constant illumination. Correct PBR diffuse texture should contain only colors without any shadows. We do not want any strong shadows on our object; we use artificial light (probably day/night cycle) in our game and shadows are generated artificially. It is possible to remove shadows from the texture, but it is a time-consuming and challenging task which can decrease overall texture quality.

[1]RAW image format contains captured image data as recorded by the camera sensor [12].

### 3.1.2 Capturing on Location

Let us assume that we have chosen our object for reconstruction. We photograph our object from all sides and different angles, so make sure that every part of it is accessible. First, we remove all undesired elements on the photographed object. All these elements that we do not remove will be present in the final game asset. If we want to capture surface ground for tileable[2] material, we should remove significant patterns. They will be visible after repeating the material in our game. Then use markers to define the reconstruction region, based on the texel density of our material. Texels are pixels in the image texture; we use this term to separate them from the pixels on the screen. If we want to achieve $2048 \times 2048$ textures with texel ratio of 1000 texels per one meter in the game, we have to photograph an area of $4\,\mathrm{m}^2$. It is crucial to keep the same scale ratio between all assets, otherwise over-detailed assets or assets with poor quality may be created. A prepared tree stump is shown in Figure 3.3.

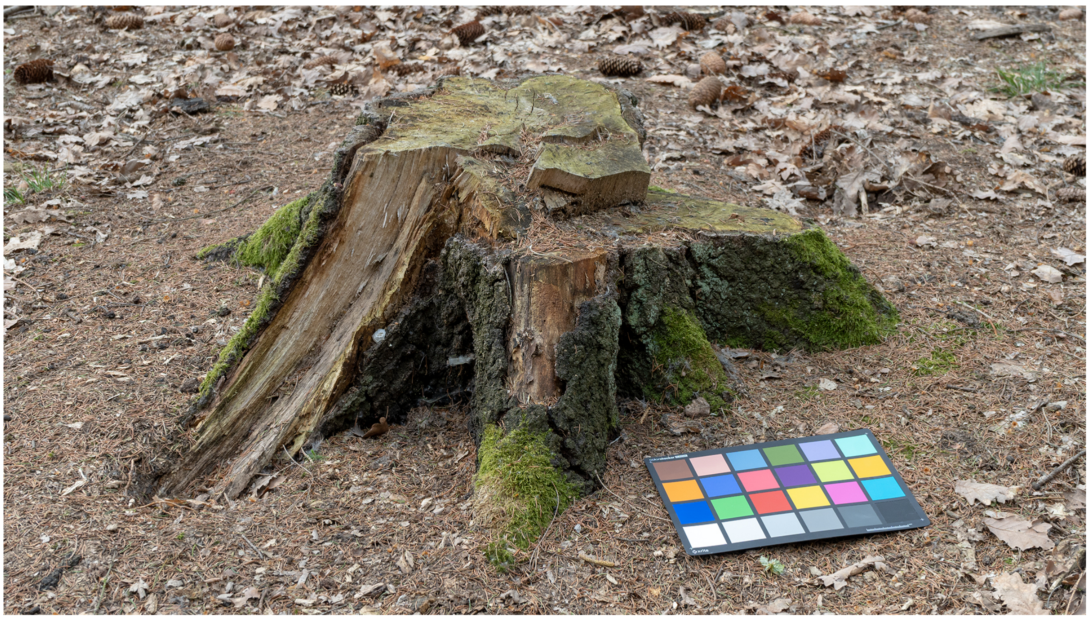


**Figure 3.3:** Prepared tree stump for reconstruction with the color checker.

The most important thing is a need to take photographs that are sharp, correctly exposed and focused [23]. We switch our camera to manual mode; it is recommended to have good practice with it. We will go through some common parameters that we need to set correctly. The amount of light reaching the digital camera's sensor is defined by the amount of time the light is allowed to fall (exposure time) and the size of the hole through which the light travels inside the lens (aperture) [30]. The overall lighting can be changed with the camera's sensor sensitivity (ISO). These three parameters drive the quality of our game asset.

[2]Tileable texture (also called seamless) is an image that can be repeated without a visible seam.

ISO gives us the ability to take photographs with poor lighting conditions. Unfortunately, high sensor sensitivity settings produce grain in the photograph which usually takes the appearance of chroma noise that is visible as random colored spots across the photograph [30]. It is not recommended to set ISO higher than 1000, because it affects photogrammetry software during reconstruction and causes noise which is visible on generated geometry. Size of the lens aperture affects depth of field, which is the distance between the nearest and the furthest objects that are in sharp focus. It is recommended to set aperture higher than f/8, but it depends on the depth of photographed surface, shooting angle, and weather conditions. Exposure time depends on lighting conditions. If we have the tripod or monopod, we can set higher exposure time. If we have to stabilize our camera with hands, it is recommended to use as small exposure time as possible. These parameters need to be set before every capture and should not be changed during the entire capture.

We need to photograph every part of our object. First, we capture one loop around the object with three height levels in ten-degree increments, where the whole object fits in the image (approximately 108 images). Next, we capture the close-range images with details of the object. It is recommended to shot images with at least 60% overlap [17]; it is crucial because low overlap causes photogrammetry software to fail image alignment.

### 3.1.3 Images Processing

Captured images need to be calibrated and exported. It can be done in image-processing software such as DxO PhotoLab or Adobe Lightroom. First, we create a preset based on the color checker for correcting image colors. Next, we fix the exposure if needed and lighten the shadowy parts and darken the lighted areas. We should minimize all the shadows captured on our object. After updating our preset, we apply it to all captured images and export them into the specific format supported by a chosen photogrammetry software.

## 3.2 Generating Geometry

Generating high-resolution meshes is done in a photogrammetry software such as Meshroom, Colmap, and RealityCapture. It requires a powerful computer with hardware shown in Table 3.1. We register all exported images into the memory. The software starts with extracting distinctive groups of pixels that are invariant to changing camera viewpoints. The algorithm used for feature detection is called Scale-invariant feature transform (SIFT) [19]. The algorithm first extracts a set of

| System Requirements | Minimum | Recommended |
|---|---|---|
| **Processor** | 64-bit processor with SSE 4.2 (Streaming SIMD Extensions) | Intel i7-9700K or AMD RYZEN Threadripper 1920X |
| **Memory** | 8 GB DDR3-1600 RAM | 64 GB DDR4-2133 RAM |
| **Graphics** | NVIDIA graphics card with CUDA 2.0 and 1 GB VRAM | NVIDIA GeForce GTX 1080 or TITAN X |

**Table 3.1:** Photogrammetry software system requirements

images and compares individual features irrespective of rotation, translation, and scale. Features are matched together, and individual camera positions, directions, and up vectors are estimated.

After the camera alignment, we align our scan preview to the ground and set the reconstruction region for our asset. Then we run meshing which is the most time-consuming task. For example, the reconstructed tree stump shown in Figure 3.4 took eight hours to generate with Intel i7-4700HQ and NVIDIA GeForce GTX 780M.

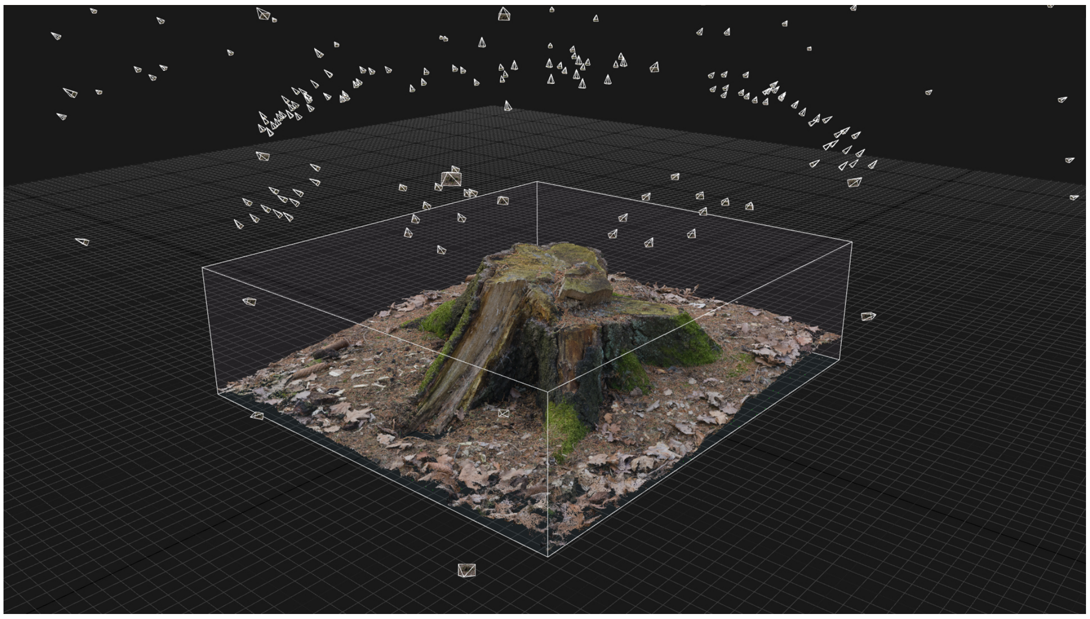


**Figure 3.4:** Generated and textured tree stump with camera alignment and reconstruction region. All photographs were authored with full-frame camera Sony $\alpha$7 III; the geometry was reconstructed from 260 images in RealityCapture.

We create a simplified version of reconstructed geometry because many external geometry tools cannot handle full resolution generated meshes which consist of millions of polygons. The base color can be exported in two ways. We can save color into vertices or generate mesh parameterization for our simplified mesh and generate the color texture with fixed texel density. We also keep full resolution mesh for normal maps, height maps, and ambient occlusion. Finally, we export all the generated meshes and proceed to optimization.

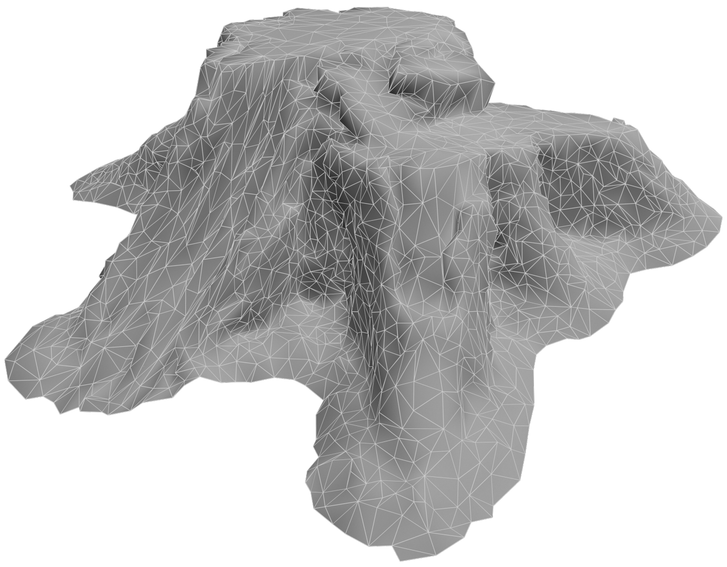


**Figure 3.5:** Retopologized tree stump with a wireframe topology of about 6000 triangles.

## 3.3 Mesh Optimization

Now we have a high poly mesh and a simplified mesh with color information, but we need an efficient low poly asset. We need to decimate (reduce the polygon count) the geometry, fix geometry errors, and retopology our asset. Mesh decimation and initial cleaning can be done in MeshLab which gives accurate results. Widespread problems in the decimated mesh are holes, T-vertices (edges in T formation), and non-manifold mesh. The manifold mesh is without any topological inconsistencies, such as having three or more polygons sharing an edge, disconnected vertices, and disconnected edges [3]. Then we bring the mesh into a general 3D modeling software for retopology. The type of retopology we perform depends on the asset; whether it is going to be static or animated. Before updating the mesh topology, we delete all parts of the mesh that we do not want on the final game asset. We can draw a new topology on the decimated mesh or update the topology of the current mesh. This task is crucial; all the topology errors that we do not remove are going to occur during the mesh parameterization. See Figure 3.5 for an example of the finished topology of the tree stump.

## 3.4 Texture Creation

To create textures for the low poly mesh, we have to create a mesh parameterization. Then we can define all maps needed for PBR shader.

### 3.4.1 UV Parameterization

UV parameterization can either be generated automatically or created manually. Automatic methods are useful for simple primitives, but it is not recommended to use automatic methods for complex meshes, they tend to waste vast amounts of texture space. We open UV Texture Editor in our preferred 3D modeling software and start with seam definition. Next, we unwrap our mesh based on seams into a 2D plane. We can use translation, rotation, and scale to make as efficient use of the UV space available as possible. We also want to maintain proportional UV density throughout the object. While doing unwrapping, a checkered texture can be used to track any created distortion (see Figure 3.6).

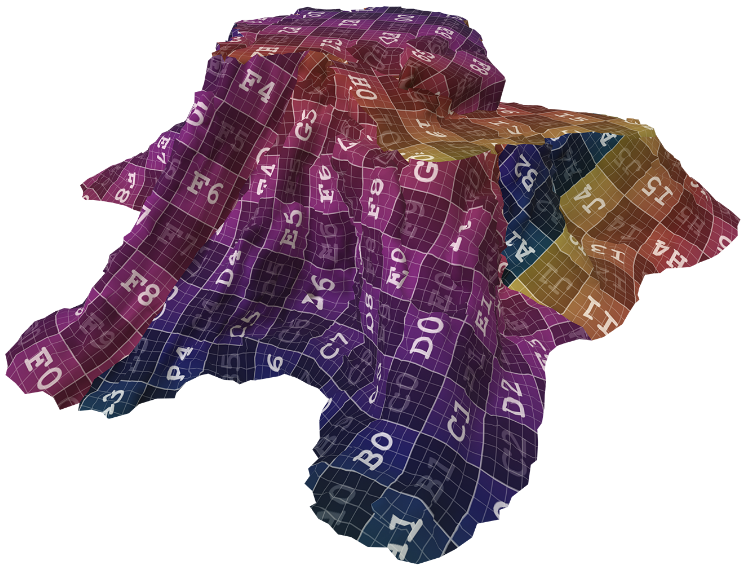


**Figure 3.6:** Checkered texture can be used to track any created distortion.

### 3.4.2 Texture Baking

Process of projecting details from one model to another (commonly from high poly mesh to low poly mesh) is called texture baking. For each texel of parameterized texture space are interpolated tangent basis, surface position, and texture coordinate. Baker casts a ray from the interpolated boundary towards the high poly mesh. When it intersects high poly geometry, it determines surface detail (tangent normal, height, and base color) which is stored into the texture using texture coordinates [22]. Ambient occlusion is commonly computed via Monte Carlo methods. Rays are cast from uniformly distributed positions over the hemisphere in random directions [3].

Texture baking is an automated process which is implemented in all standard 3D packages. We bake tangent space normal map, height map, ambient occlusion, and base color from the high poly model. Authoring accurate roughness and metallic textures is hard, but we can use a physically based material chart as a starting reference [32]. The chart shows us linear microsurface, to get the roughness texture, we need to invert the microsurface map from the chart. We can convert base color texture to grayscale and use it as a mask for defining roughness values to different parts of the game asset. We avoid tweaking materials values to look more impressive in a specific lighting environment. They should be consistent and look realistic no matter how we light them. See Figure 3.7 for the finished tree stump with applied textures.

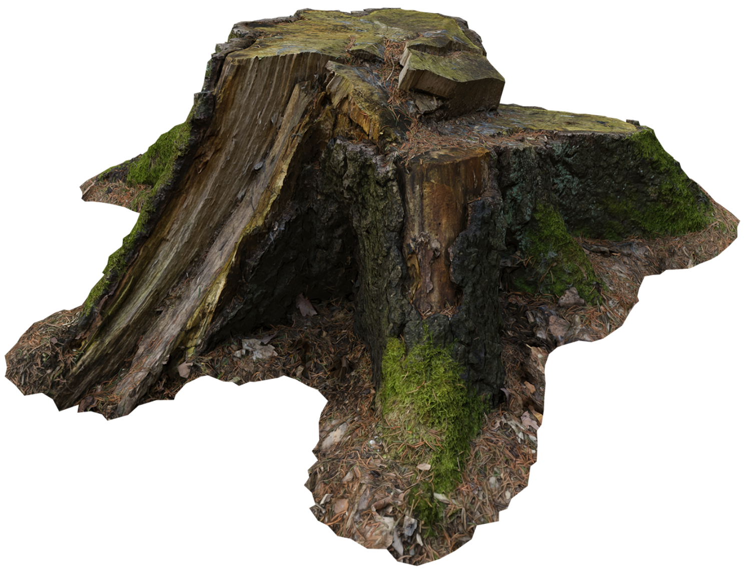


**Figure 3.7:** Game ready tree stump with physically based textures.

# Chapter 4

# Environment Creation in Unreal Engine

Software for developing unbelievable 3D game worlds—game engines like Unreal Engine 4 developed by the Epic Games company is a complete suite of development tools made for anyone working with real-time technology. Unreal Engine was first used in first-person shooter Unreal, which was released in 1998. This version of the engine had many robust systems, such as rendering, artificial intelligence, collision detection, and scripting [25], the current version of the engine, Unreal Engine 4.22, is an extremely powerful tool containing everything we need for building a realistic game environment.

**Figure 4.1:** Stylized environment from the most popular online video game Fortnite, game created in Unreal Engine. (Image courtesy of Epic Games, Inc.)

When we first open Unreal Engine Launcher, we can open already created projects or create a new one based on available templates. Each template provides specific assets needed for the chosen game type. They are available in two versions: C++ version, and a Blueprint (visual script) version with the same behaviour. Blueprint Visual Scripting is a flexible and extremely powerful node-based interface to create gameplay elements without writing a single line of code. It is used to describe object-oriented classes in the engine [14].

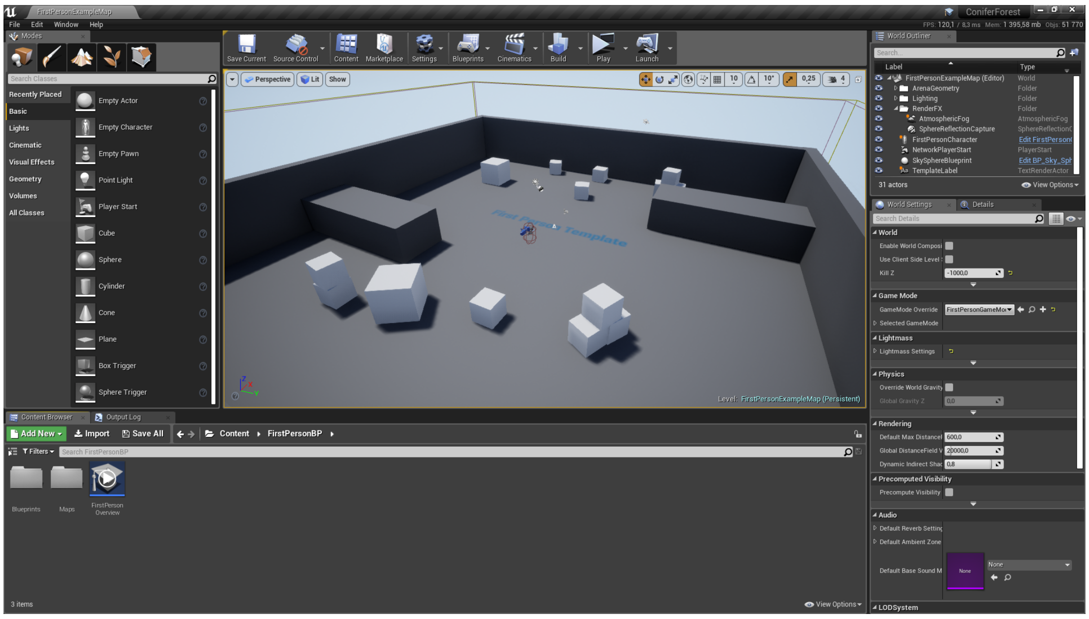
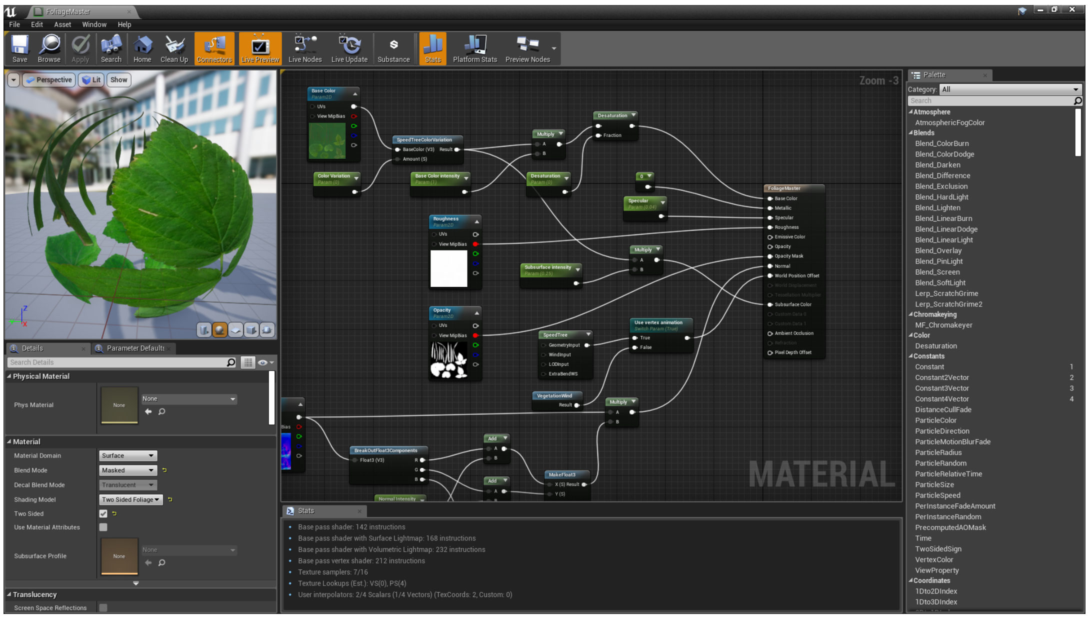


Figure 4.2: The Unreal Editor provides the core level creation functionality. It is where levels are created, viewed, and modified.

There are many templates available; for example, a first-person shooter template gives us a player controller which is viewed from the first-person perspective. The player character is equipped with a gun that fires a simple sphere projectile. Another example is a vehicle template, which features simple physics-driven vehicle. Vehicle controller has two cameras with implemented head-up display, first is behind the vehicle (also called chase view), second is inside the vehicle (also called cockpit view). The head-up display informs the player about current gear and speed. After creating a new project based on the available template, we can see the Unreal Engine interface (see Figure 4.2).

## 4.1 Creating a Landscape

Unreal Engine has the ability to build huge maps (a single level can be as large as The Elder Scrolls V: Skyrim). First, we create a new level and start with landscape creation. Creating a new landscape from scratch is done through the Landscape tool. We have two different possibilities for creating a landscape for our environment. The first method of landscape creation is importing a custom height map which is a grayscale map that uses the scale from black to white to form the height of the landscape [25]. This method requires the use of an external program such as World Machine. The second method is creating a flat landscape based on input parameters. Landscapes are made of multiple square components which can optionally be divided into subsections. They are used for landscape level of detail (LOD) calculations [14]. When we generate our landscape geometry, we can use sculpting tools to modify landscape height. After updating the overall landscape shape, we add some erosion effects to the landscape to give it a weathered look.



Details

Physical Material
- Phys Material: None

Material
- Material Domain: Surface
- Blend Mode: Masked
- Decal Blend Mode: Translucent
- Shading Model: Two Sided Foliage
- Two Sided: ✓
- Use Material Attributes: ☐
- Subsurface Profile: None

Translucency
- Screen Space Reflections: ☐

Stats

- Base pass shader: 142 instructions
- Base pass shader with Surface Lightmap: 168 instructions
- Base pass shader with Volumetric Lightmap: 232 instructions
- Base pass vertex shader: 212 instructions
- Texture samplers: 7/16
- Texture Lookups (Est.): VS(0), PS(4)
- User interpolators: 2/4 Scalars (1/4 Vectors) (TexCoords: 2, Custom: 0)

**Figure 4.3:** The Material Editor is a node-based graph interface that enables us to create shaders that can be applied to our geometry.

## 4.2 Creating Landscape Materials

Materials in Unreal Engine are constructed not through a code but via a network of visual scripting nodes (See Figure 4.3). Each node contains a snippet of DirectX's High-Level Shader Language (HLSL), created to execute a specific task [14]. The landscape contains layers with different materials that can be blended. We create material functions for each layer of the landscape material. Material functions are small snippets of graphs that can be reused across multiple materials. They are simplifying the creation process by providing instant access to regularly used material networks. Each material function has texture samples for albedo, tangent space normal, roughness, ambient occlusion, and height. Texture samples have UV (two-vector) input and tiling (scalar) input; we set tiling based on landscape size and UV input to the absolute world position[1]. Material functions outputs go into LandscapeLayerBlend node which creates material painting layers. Each landscape layer has a grayscale map which defines where is the material function applied. We can also blend two materials based on their height map. Let us assume that we have dirt material with dynamic height map and calm water with constant height. We can blend these two materials based on their height textures; we get dirt with puddles. Figure 4.4 shows a physically based dirt material with adjusted textures based on a mask created with height blending.

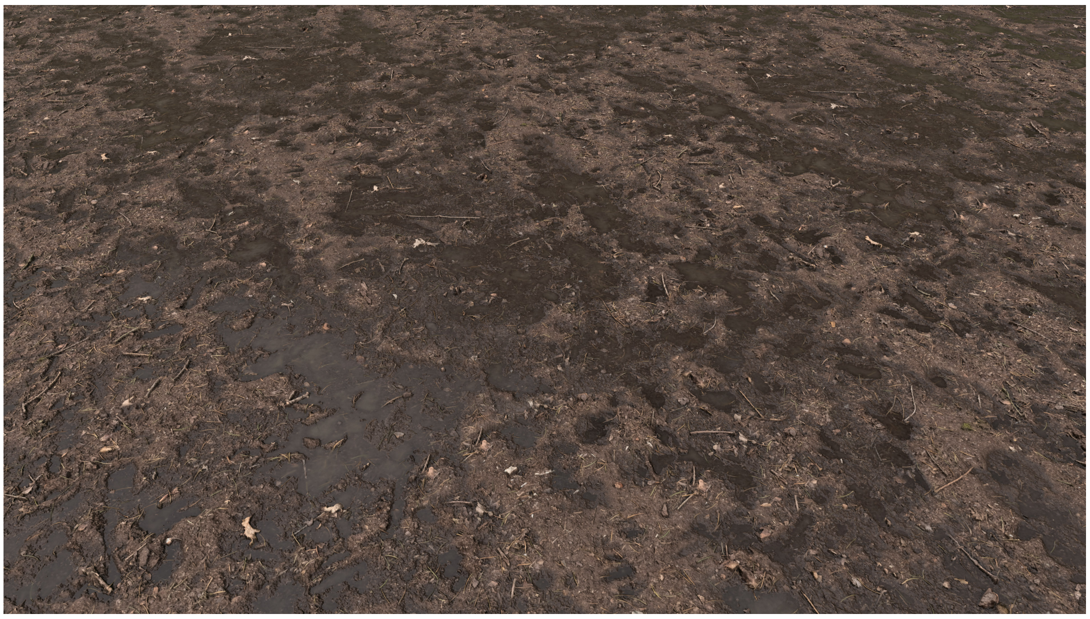


**Figure 4.4:** Dirt material with puddles created with photogrammetry method and height blending.

Landscape material can modify the landscape geometry based on assigned height maps. We create a material function that tessellates and displaces the actual geometry of the landscape. Tessellation is a runtime process that splits triangles into smaller triangles to increase the surface detail [3]. We create a tessellation function based on the actual position of the camera. The function calculates vector length of substracted camera position and absolute world position. Then it divides calculated length with a scalar parameter. The function output is normalized and goes to Tessellation node in landscape material. Displacement function takes masks from landscape layers and multiplies them with world space vertex normal. The landscape is tessellated and displaced based on landscape height maps and player position.

[1]Absolute world position node outputs the position of the current pixel in world space.

## 4.3 Generating Vegetation

Creating and texturing vegetation such as grass, ferns, and trees can be challenging, but once we use photogrammetry and procedural techniques for populating our game, a game environment can be relatively easy to create. In order to create an environment that looks believable, we need to understand what should be in a particular environment and focus on shape and color.

### 4.3.1 Setting Up Collisions

By default, there is no collision set on our assets. Without collisions, the player would be able to move through geometry or fall into the void. Collisions are determined by entities that are invisible to the player; these objects include sphere, triangles, axis-aligned bounding box (AABB), oriented bounding box (OBB), and discrete oriented polytope (k-DOP) [3]. Unreal Engine sets object type and series of responses to every object that can collide. We can easily set what collides with what and what happens to involved objects when a collision event occurs. We can add collision entities in Mesh Editor; created entities can be moved, rotated, and scaled into the desired position.

### 4.3.2 Creating Level of Detail

When an asset is far off in the game and displayed using a few pixels, we do not need to draw it with full resolution. We can swap it for a lower version which increases the efficiency of rendering. We can build several versions of the asset, each with progressively smaller polygon count, and these asset versions can be swapped by the engine [2]. Another possibility is generating other versions in the engine. Unreal Engine uses Simplygon which provides high-quality mesh reduction. Individual mesh versions are swapped based on the covered screen size.

Since we are creating an open world game with massive forest, the LODs of our assets are crucial. Small plants, rocks, and sticks will disappear when the screen coverage is deficient, but trees will not. We can create tree billboards and apply them as the last displayed LOD. Created LODs with colored meshes are shown in Figure 4.5.

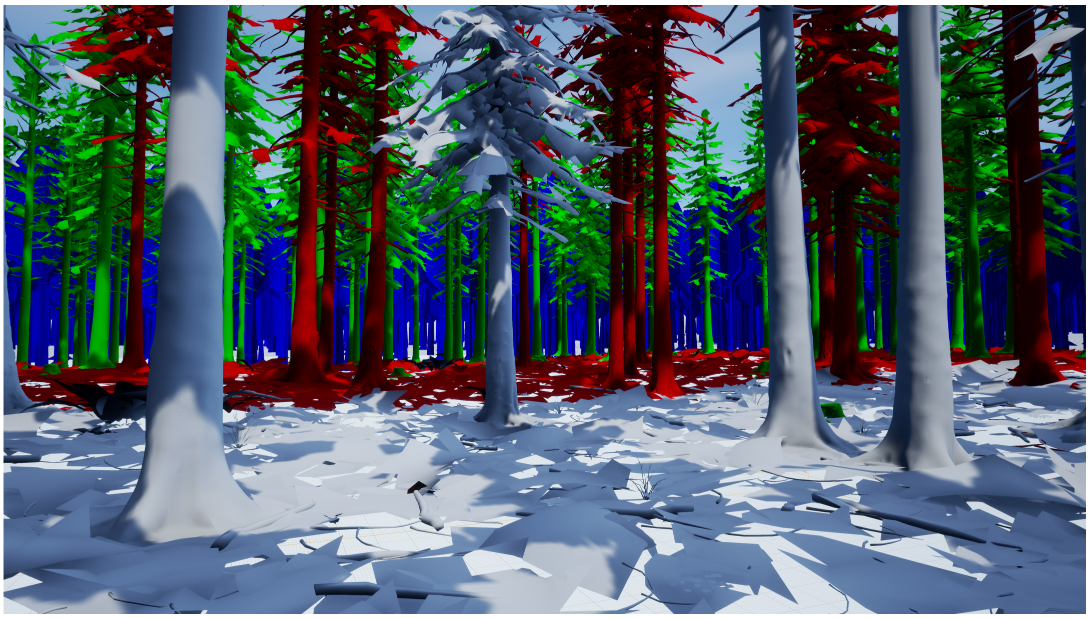


**Figure 4.5:** Level of detail coloration view displays the current LOD index of a mesh. Assets with white color are rendered with the maximum resolution; blue assets use billboard geometry.

### 4.3.3 Procedural Foliage Spawner

Unreal Engine allows us to place created assets inside the level using a procedural placement system instead of placing everything manually. The Procedural Foliage Spawner is used for simulating enormous forests that are filled with many different kinds of assets such as grass, plants, bushes, and trees. It simulates how a forest grows over the years using steps to define individual years. Each step casts new virtual seeds into the level; these seeds act as spawn location for new foliage actors [14].

We can also spawn our assets on the landscape based on a specific layer mask. We create a material function that generates specific assets on the landscape. Let us assume that we have grass material and dirt material applied to the landscape. We want cloverleaf meshes and branches on dirt material, so we create foliage type for all assets that we want to be generated on dirt material and connect them to dirt layer. For each asset, we set spawn density, random scale, and placement settings. Branches and plants are aligned to landscape normal vector; trees are spawned only with random yaw rotation.

These procedural methods allow us to populate our world based on specific input parameters; we can generate an extensive game environment with relative ease.

## 4.4 Profiling and Optimization

Game optimization is an essential part of game development. We will go through basic profiling and optimization that we should always keep in mind when creating real-time game assets.

### 4.4.1 Draw Calls

Request that invokes the graphics application programming interface (API) such as Direct3D, OpenGL, and Vulkan to draw a group of primitives is called draw call [3]. These requests are CPU bound; the render thread needs to process each object and its data such as material, lighting setup, collision, and culling. We can reduce draw calls by combining materials accepting more complex pixel shaders (creating atlases with textures for different assets), reducing view distance, combining mesh elements into larger assets, and disabling shadow casting [14].

### 4.4.2 Shader Complexity

Shader complexity and overdraws are issues that cause GPU slogging. Overdraw occurs when a pixel must be drawn multiple times; this happens when objects are drawn on top of others and contributes significantly to fill rate issues [3]. We can reduce shader complexity by minimizing the geometry area for overdraw. Mesh geometry should exactly match the outline of the texture opacity. For example, state-of-the-art grass planes in games significantly increase shader complexity; individual blades are packed and mapped into a plane; these planes are stacked together and create 3D grass clusters. Advantage of this method is that the vertex count of the actual geometry is quite low.

The better approach for creating grass and foliage takes advantage of shader complexity. We can create planes for individual grass blades and minimize geometry area for overdraw. These planes are packed into different grass clusters for better mesh instancing. This method for creating grass increases draw calls; overall performance is significantly better. On the other hand, this method is time-consuming and can not be used on every type of vegetation [4].

### 4.4.3 MIP Mapping

Creating multiple sizes of a texture displayed at various distances is called MIP mapping [1]. Generation of MIP maps happens during importing of the texture; Unreal Engine creates a MIP map chain which consists of multiple levels of the same texture. We can author textures at pretty much whatever resolution we like with photogrammetry, but we should keep in mind that we may not always use full resolution. We can use texture streaming profiler to see what level of MIP maps we are using for a given texture and remove unused MIP levels. Figure 4.6 shows beaked hazelnut leaves which were mapped to $2048 \times 2048$ texture atlas. Textures were gathered in $4096 \times 4096$ resolution, but there is no need to use them, they require four times more memory and storage space.

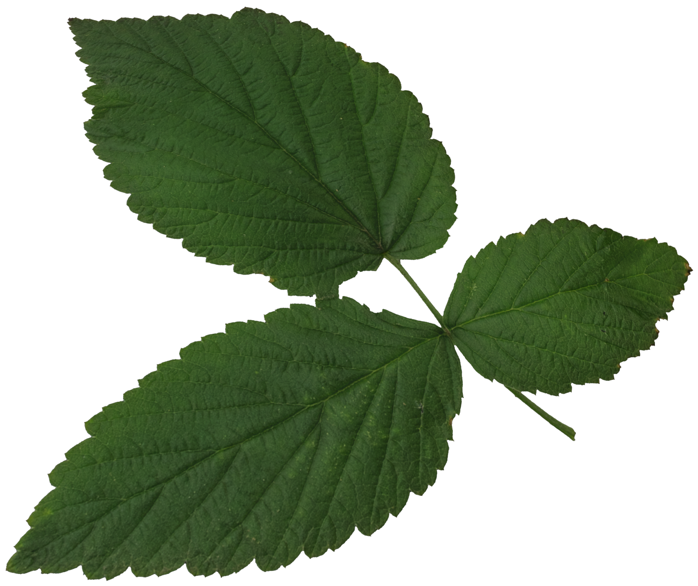


**Figure 4.6:** Beaked hazelnut leaves with downscaled $2048 \times 2048$ textures mapped on a geometry with 22 triangles.

# Chapter 5

# Results

Modern games need to run on a wide range of hardware with an acceptable frame rate. Depending on the platform and game, 30, 60, or even more frames per second may be the target. In this chapter, we summarize the photogrammetry pipeline for creating game-ready assets. Furthermore, we present measured performance with an automated cinematic level sequence on different hardware, and we compare the measured frame rate with Far Cry 5 benchmark. Finally, we discuss the visual quality of the assets created with photogrammetry.

## 5.1 Photogrammetry and Statistics

The photogrammetry method is not a silver bullet — it is only one part of achieving the final results in the game environment. It brings realistic visual appearance based on real-world data and significantly reduces asset creating time. With the photogrammetry method, anyone can become a high precision 3D scanner with a basic camera.

This method requires a significant amount of data captured during overcast days without any strong shadows. The capturing surface needs to be consistent; we are unable to reconstruct transparent, reflective, and moving objects. The capture process has taken approximately ten days; actual times needed for generating meshes in photogrammetry software vary from asset to asset, but they range from eight hours to fifteen hours.

The photogrammetry method is suitable for complex objects. Nowadays, game studios use this method mainly for creating people, natural environments, and vehicles. It is a creative process which is incredibly repetitive; it can and should be automated.

The whole pipeline consists of the following steps:

- Taking series of photographs
- Images processing
- Aligning data in photogrammetry software
- Generating a high-resolution mesh
- Mesh simplification
- Geometry retopology and parameterization
- Textures baking

We have used Autodesk 3D studio Max, Allegorithmic Substance Painter, and Adobe Photoshop for the standard modeling and texturing techniques; mesh optimization was done in MeshLab. We have also tested Meshroom, Colmap, and RealityCapture photogrammetry software; each provides different features, but all of them have given accurate results. Pine trunks and spruce trunks were reconstructed from approximately 350 images per object. Forest grounds such as needles, grass, and dirt were reconstructed from approximately 200 images per object. We have also authored approximately 250 images for each tree stump.

## 5.2 Performance

When we move a joystick or push a button, the central processing unit (CPU) processes input from the player and updates the game logic. The graphics renderer takes the current state of the game established by CPU and uses it to render an image which is displayed on the screen. We will measure this process on different hardware with automated cinematic level sequence to see the actual difference.

We have deployed the game with conifer forest running on DirextX. The game implements player controller with the first-person mode and the third-person mode. It has a basic user interface that allows the player to change graphics scalability. The world includes $4\,\mathrm{km}^2$ of playable area, background mountains, volumetric clouds, and fully dynamic lighting. The performance was evaluated on a set of different hardware shown in Table 5.1. For each test, we set graphics scalability to EPIC based on Epic Games scalability reference to use the best quality available [14]. Figure 5.1 shows the Full HD comparison of the frame times measured with the automated cinematic level sequence.

| Computer | System information |
|---|---|
| **TITAN Xp** | Windows Server 2012 R2; Intel Xeon W-2125; 64 GB DDR4-2666 RAM; NVIDIA TITAN Xp |
| **RTX 2080** | Windows 10; Intel Core i7-870; 16 GB DDR3-1333 RAM; NVIDIA GeForce RTX 2080 |
| **GTX 1080 Ti** | Windows 10; Intel Core i7-950; 12 GB DDR3-1066 RAM; NVIDIA GeForce GTX 1080 Ti |
| **GTX 1070** | Windows 10; Intel Core i7-4790; 16 GB DDR3-1333 RAM NVIDIA GeForce GTX 1070 |
| **GTX 770** | Windows 10; Intel Core i7-4770K; 20 GB DDR3-1333 RAM NVIDIA GeForce GTX 770 |

**Table 5.1:** Tested hardware.

Measured frame times for each computer are expected due to graphics cards performance. The game thread times were approximately 2 ms, so the game was GPU bound. The best frame times (approximately 7.2 ms) were measured on TITAN Xp powered by NVIDIA Pascal architecture. RTX 2080 powered by NVIDIA Turing architecture measured average time 8.2 ms. Since we did not implement real-time raytracing which is supported in the latest Unreal Engine version, GTX 1080 Ti had a similar average frame time 8.7 ms. Note that both computers had DRR3 memory with lower data rate and input/output bus clock than standard DDR4 memory.

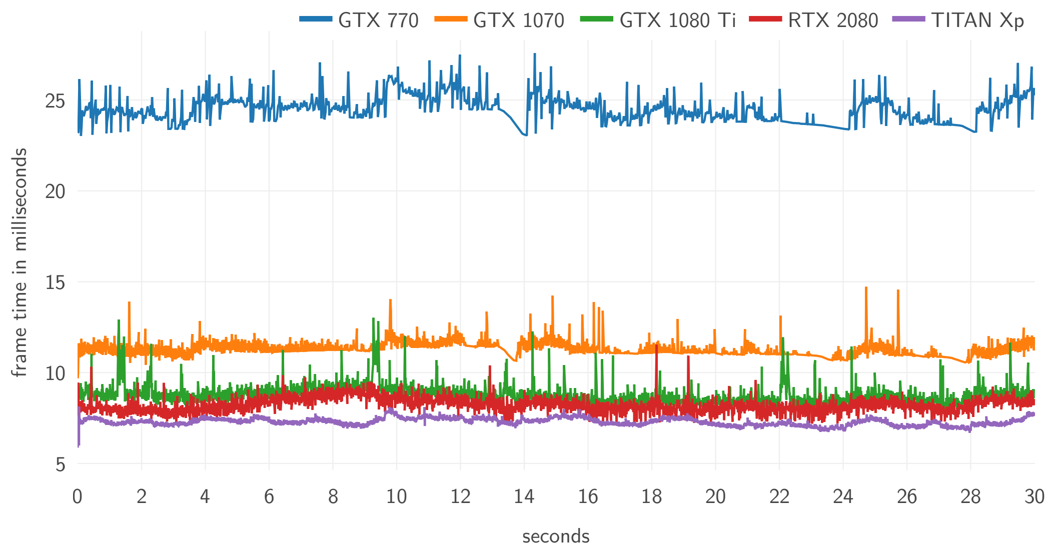


**Figure 5.1:** Frame time measured with the level sequence on Full HD (1920 × 1080).

Important measurement brought GTX 1070; this video card and GTX 1060 are currently used by most players on Steam video game distribution platform [13], so that most computers frame time will be approximately 11.3 ms. The highest average frame time 24.4 ms was measured on older NVIDIA video card GTX 770 powered by Kepler architecture. Figure 5.2 shows the measured average frame times per second (FPS).

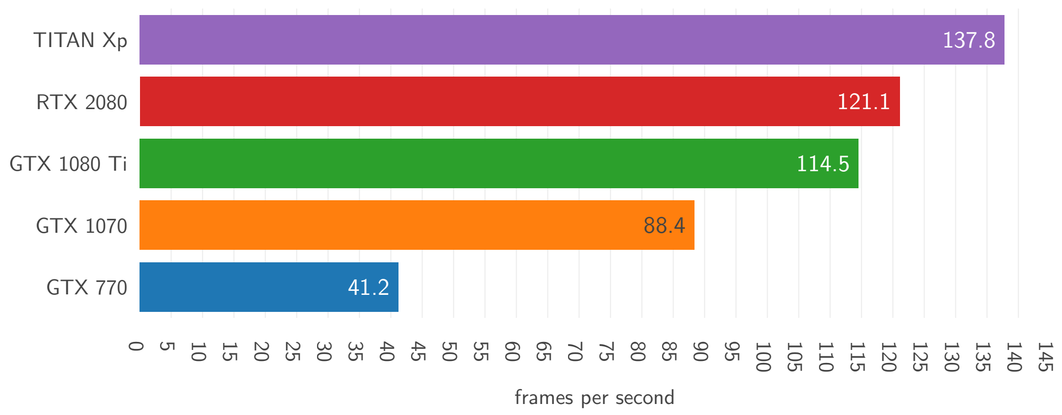


**Figure 5.2:** Average frame times per second measured on Full HD (1920 × 1080).

Nowadays monitor with 1920 × 1080 pixel resolution is still the standard for video games, but graphics cards aim to 4K monitors with a 3840 × 2160 pixel resolution. The game consoles are also capable of 4K resolution, so we have decided to measure deployed game with the same level sequence and scalability with 4K resolution (see Figure 5.3).

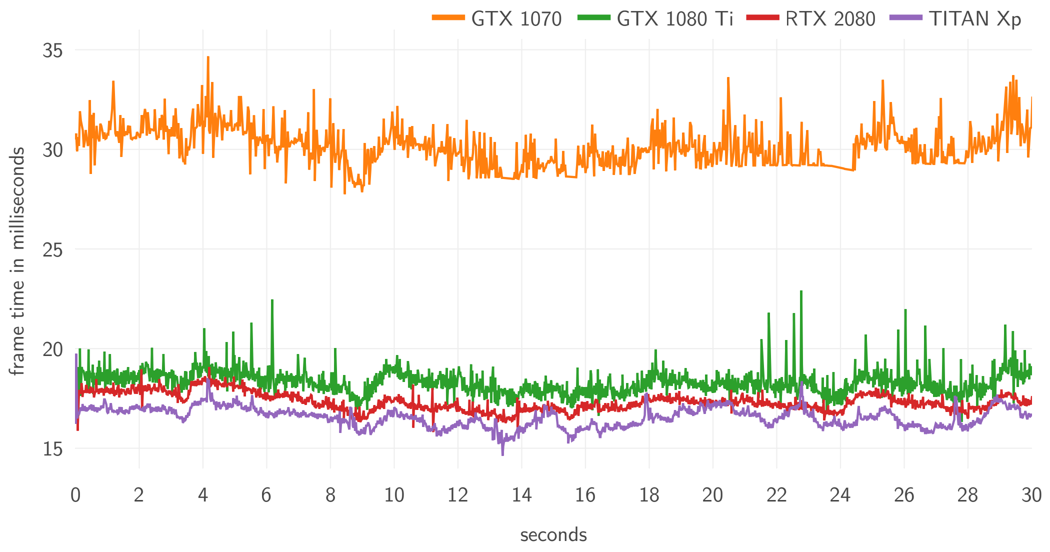


**Figure 5.3:** Frame time measured with the level sequence on 4K UHD (3840 × 2160).

The GTX 1070 average frame time was 30 ms which is a significant drop. Other computers average frame times were approximately the same, GTX 1080 Ti measured 18.2 ms, RTX 2080 succeeded with 17.3 ms, and finally, TITAN Xp conquered with 16.5 ms. Frames per second measured on 4K resolution are shown in Figure 5.4.

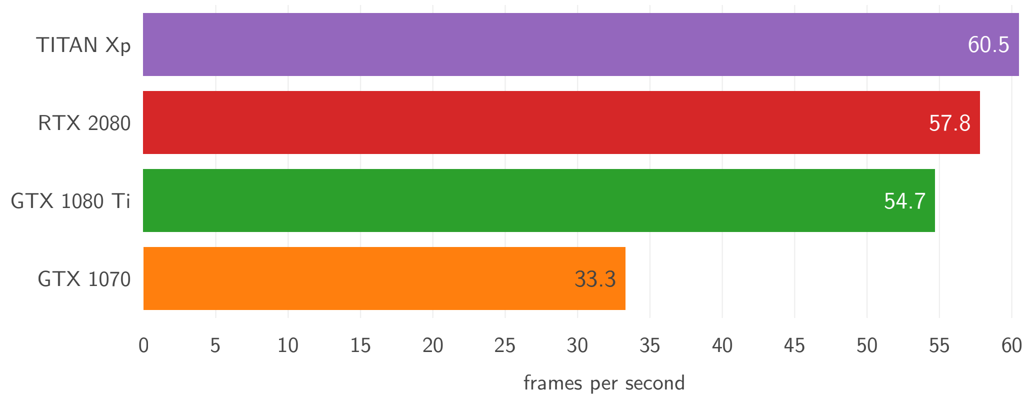


**Figure 5.4:** Average frame times per second measured on 4K UHD (3840 × 2160).

We have experimented with GTX 1070 computer to have frame rate reference with AAA game. We have chosen Far Cry 5 which includes performance sequence similar to ours. We have set the graphics quality to ultra which defines texture filtering, graphical details of shadows and water, graphical complexity of the world geometry and vegetation, and quality of the volumetric fog. Ultra quality also enables Temporal anti-aliasing[1] (TAA) and motion blur. Note that we have also deployed the game with TAA. Far Cry 5 with ultra settings requires 2.7 GB of VRAM; we with EPIC settings require only 1.6 GB of VRAM. The measured average FPS on Full-HD monitor was 76 frames; the deployed game with conifer forest ran with 88 FPS, so we have some space for adding explosions, water, characters, and vehicles.

## 5.3 Visual Quality

The main reason why we used photogrammetry method is the realistic visual appearance based on real-world data. The deployed conifer forest consists of essential ground materials such as grass, mud, dirt, and needles. The landscape use displacement based on material height textures and camera position. We have created rocks, branches, needles, and bark clusters with many variations to build realistic distinctive forest floor. The forest includes various vegetation assets such as leaf clovers, grass, ground ivy, beaked hazelnuts, and ferns. They are controlled with procedural mesh spawner based on specified landscape layer masks. The forest also consists of tree stumps, spruce

[1]Anti-aliasing programs such as Fast approximate anti-aliasing (FXAA) or Temporal anti-aliasing (TAA) are pixel shader programs that reduce contrast on pixel and sub-pixel aliasing [18, 16].

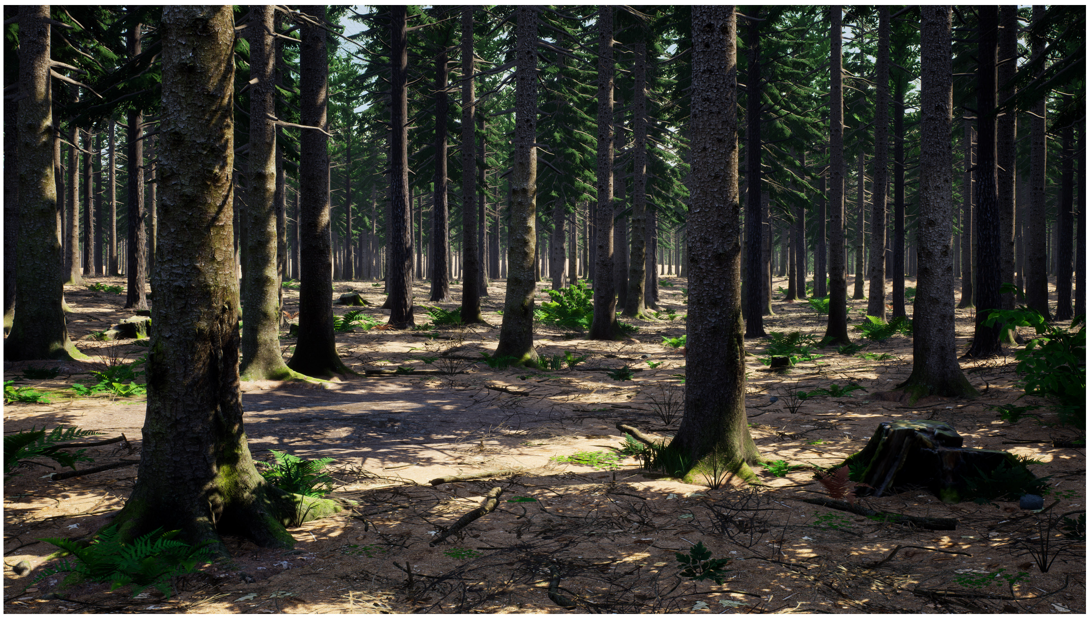


**Figure 5.5:** A screenshot from the conifer forest game.

trees, and pine trees (one tree instance has approximately 20 000 triangles). Tree bases were created with photogrammetry method; treetops were generated procedurally in SpeedTree. Trees are controlled with the procedural spawner which simulates forest grown. All small assets such as foliage, rocks, and branches are packed to 2048 × 2048 textures, only tree trunks are rendered with 4096 × 4096 textures. Figures 5.5 and 5.6 show the visual quality of the whole conifer forest and vegetation.

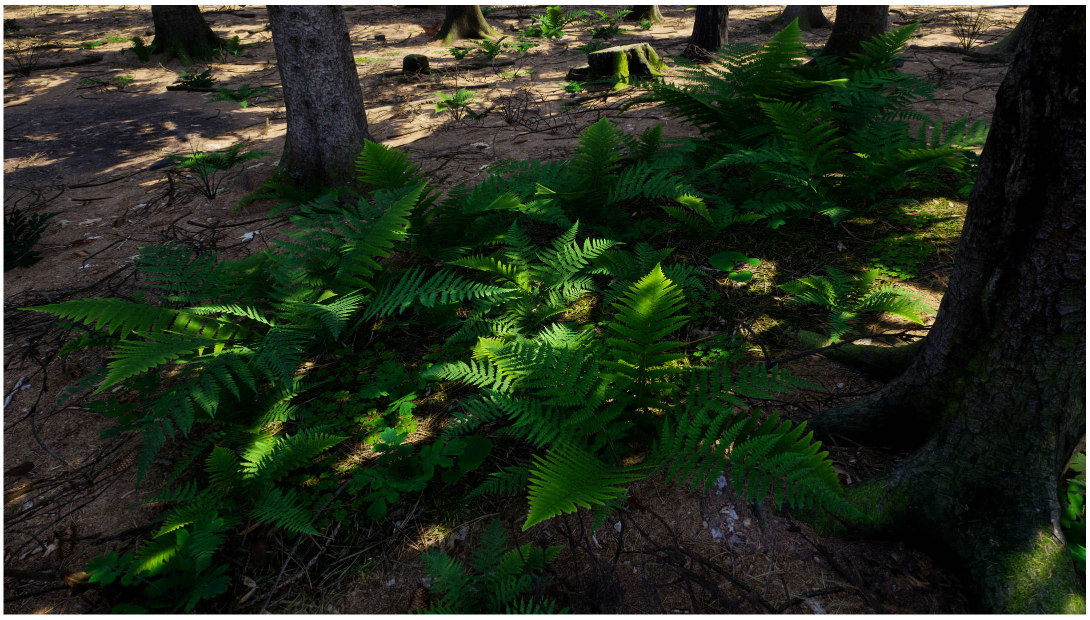


**Figure 5.6:** Forest vegetation close up.

# Chapter 6

# Conclusions

We have described a photogrammetry pipeline for creating game-ready assets. It is based on comparing multiple images taken from different positions for estimating object coordinates in 3D space. We have discussed the research and development before the actual capture and shown how to photograph the asset for reconstruction. Then we have reconstructed objects from conifer forest and optimized them for real-time rendering. Assets were efficiently parameterized into 2D space with proportional UV density. After that, we have projected details from the reconstructed geometry into the low poly version and authored textures for the physically based rendering.

Created assets were tested in a game environment built in Unreal Engine. We have described the development process and techniques that were used for making the conifer forest. First, we have created extensive landscape geometry with displaced materials based on their height map. Then we have designed LODs for the assets and added collision entities to block any movement through the geometry. The massive forest was generated procedurally with a designed spawner that allowed us to simulate growing forest over the years. Finally, we have optimized the created game environment and deployed the game for performance testing.

We have measured average frame time 7.2 ms on TITAN Xp; on RTX 2080 the level sequence has run with average frame time 8.2 ms; the similar result 8.7 ms was measured on GTX 1080 Ti. The most popular NVIDIA video card GTX 1070 measured average frame time 11.3 ms, and the highest frame time 24.4 ms was achieved on older NVIDIA video card GTX 770.

Future research should focus on asset blending with the landscape. Landscape textures should be interpolated to most reconstructed assets; they could also have landscape radiosity projected on to them. We would also like to automate the repetitive process of the photogrammetry method.

# References

[1] L. Ahearn. *3D Game Textures: Create Professional Game Art Using Photoshop.* CRC Press, 4th edition, 2016.

[2] L. Ahearn. *3D Game Environments: Create Professional 3D Game Worlds.* CRC Press, 2nd edition, 2017.

[3] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, S. Hillaire, and M. Iwanicki. *Real-Time Rendering, Fourth Edition.* CRC Press, 4th edition, 2018.

[4] N. Arenz. Preparing Realistic Grass in Unreal Engine 4. `https://80.lv/articles/preparing-realistic-grass-in-ue4/`.

[5] E. Assaf. *Rigging for Games: A Primer for Technical Artists Using Maya and Python.* CRC Press, 1st edition, 2015.

[6] L. M. Bishop, M. Jančošek, and C. Cowan. Photogrammetry for Games: Art, Technology and Pipeline Integration for Amazing Worlds. Game Developers Conference (GDC), 2017. `https://developer.nvidia.com/gdc17`.

[7] K. Brown and A. Hamilton. Photogrammetry and Star Wars Battlefront. Game Developers Conference (GDC), 2016. `www.gdcvault.com/play/1023272/Photogrammetry-and-Star-Wars-Battlefront`.

[8] J. Campbell. Bringing Hell to Life: AI and Full Body Animation in DOOM. Game Developers Conference (GDC), 2017. `www.gdcvault.com/play/1024186/Bringing-Hell-to-Life-AI`.

[9] E. Carrier. Procedural World Generation of Far Cry 5. Game Developers Conference (GDC), 2018. `www.gdcvault.com/play/1025557/Procedural-World-Generation-of-Far`.

[10] A. Chopine. *3D Art Essentials: The Fundamentals of 3D Modeling, Texturing, and Animation*. Focal Press, 1st edition, 2011.

[11] B. Cloward. Shading the World of Anthem. Game Developers Conference (GDC), 2019. www.gdcvault.com/play/1026274/Shading-the-World-of-Anthem.

[12] C. Coe and C. Weston. *Creative DSLR Photography: The ultimate creative workflow guide (Digital Workflow)*. Focal Press, 1st edition, 2009.

[13] V. Corporation. Steam Hardware and Software Survey. https://store.steampowered.com/hwsurvey/videocard/.

[14] Epic Games. Unreal Engine 4 Documentation. https://docs.unrealengine.com/en-us/.

[15] S. Foster and D. Halbstein. *Integrating 3D Modeling, Photogrammetry and Design (SpringerBriefs in Computer Science)*. Springer, 1st edition, 2014.

[16] J. Korein and N. Badler. Temporal Anti-Aliasing in Computer Generated Animation. *SIGGRAPH Computer Graphics*, 1983.

[17] V. Kuzmin. Full Photogrammetry Guide for 3D Artists. https://80.lv/articles/full-photogrammetry-guide-for-3d-artists.

[18] T. Lottes. FXAA. *ACM SIGGRAPH Filtering Approaches for Real-Time Anti-Aliasing course*, 2011.

[19] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the International Conference on Computer Vision*. IEEE Computer Society, 1999.

[20] S. McAuley. The Challenges of Rendering an Open World in Far Cry 5. *SIGGRAPH Advances in Real-Time Rendering in 3D Graphics and Games course*, 2018.

[21] W. McDermott. *The PBR Guide: A Handbook for Physically Based Rendering*. Allegorithmic, 3rd edition, 2018.

[22] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 1st edition, 2007.

[23] RealityCapture Support. Taking Pictures for Photogrammetry. https://support.capturingreality.com/hc/en-us/articles/115001528211-Taking-pictures-for-photogrammetry.

[24] M. Reparaz. Far Cry 5 – How Hope County was built into a believable slice of Montana. https://news.ubisoft.com/en-us/article/315853/far-cry-5-hope-county-built-believable-slice-montana.

[25] A. Sanders. *An Introduction to Unreal Engine 4*. CRC Press, 1st edition, 2017.

[26] G. Sanders. Between Tech and Art: The Vegetation of Horizon: Zero Dawn. Game Developers Conference (GDC), 2018. `www.gdcvault.com/play/1025530/Between-Tech-and-Art-The`.

[27] D. Santiago. Procedurally Crafting Manhattan for Marvel's Spider-Man. Game Developers Conference (GDC), 2019. `www.gdcvault.com/play/1026415/Procedurally-Crafting-Manhattan-for-Marvel`.

[28] SCS Software. 3D Scanning - The Next Generation Scania. `https://blog.scssoft.com/2017/02/3d-scanning-next-generation-scania.html`.

[29] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games (Computational Synthesis and Creative Systems)*. Springer, 1st edition, 2016.

[30] J. Tarrant. *Understanding Digital Cameras: Getting the Best Image from Capture to Output*. Focal Press, 1st edition, 2007.

[31] J. van Muijden. GPU-Based Run-Time Procedural Placement in Horizon: Zero Dawn. Game Developers Conference (GDC), 2017. `www.gdcvault.com/play/1024700/GPU-Based-Run-Time-Procedural`.

[32] J. Wilson. Physically-Based Material Values. `https://marmoset.co/posts/physically-based-rendering-and-you-can-too/`.

# Appendix A

# DVD Contents

The enclosed DVD contains the text of this thesis and the deployed game with packed conifer forest. It requires 1 GB of free file space and DirectX runtime; EPIC graphics settings require graphics card with at least 1.6 GB of VRAM.