Concept study for the use of a render engine for computer games in a flight simulation

Bachelor's thesis

for the degree

Bachelor of Science

of the course Information Technology

at the Baden-Wuerttemberg Cooperative State University Mannheim

by

Michael Ratzel

Submission on 14.09.2020

Processing Period:	22.06.2020 - 13.09.2020
Student id, course:	6614999, TINF17ITIN
Department:	FT-FDS
Apprenticing company:	German Aerospace Center (DLR)
Company's supervisor:	Torsten Gerlach
University's reviewer:	Prof. Dr. Heinz Jürgen Müller

Declaration

I hereby assure you that I have written my bachelor's thesis on the

Subject

Concept study for the use of a render engine for computer games in a flight simulation

independently and that I have not used any other sources and aids than those indicated.

I also assure you that the electronic version submitted is the same as the printed version.*

* if both versions are required.

Braunschweig, 14.09.2020

Abstract

The Air Vehicle Simulator (AVES) is operated by the German Aerospace Center and provides a highly flexible platform with up to four different cockpit modules for research and development purposes. During the flight simulation, its visuals are created by the AVESViewer software based on the open-source rendering framework OpenSceneGraph (OSG). Because the further development of OSG was terminated in 2019, a replacement is required in the midterm. This work presents the Unreal Engine 4 (UE4) as a possible candidate by conducting a concept study to develop the new UnrealViewer application coupled to the existing AVES environment. At first general requirements for a visual system of flight simulation are specified, and the current AVESViewer is analyzed how those are fulfilled. For most of them, UE4 techniques could be presented that achieve the same purpose as the AVESViewer ones. The two exceptions are investigated in-depth.

The first is that the model import process aiming to reuse the terrain model's height and image data and the FLT aircraft models was developed, as UE4 can not natively import it. Both model types can only provide a compromise between optimal performance, visual quality, and usability, and although the full potential was not exploited, the general possibility to import the models was shown.

The other aspect of the concept study, the controllability via the AVES Visual Simulation Animation Protocol, was successfully shown in combination with a concept for configuring everything via XML and a low input lag.

Kurzzusammenfassung

Der Air Vehicle Simulator (AVES) wird vom Deutschen Zentrum für Luft- und Raumfahrt betrieben und bietet eine hochflexible Plattform mit bis zu vier verschiedenen Cockpitmodulen für Forschungs- und Entwicklungszwecke. Die Visualisierung während der Flugsimulation erfolgt mit der Software AVESViewer, die auf dem Open-Source-Rendering-Framework OpenSceneGraph (OSG) basiert. Da die Weiterentwicklung von OSG im Jahr 2019 eingestellt wurde, ist mittelfristig ein Ersatz erforderlich. In dieser Arbeit wird die Unreal Engine 4 (UE4) als möglicher Kandidat vorgestellt, indem eine Konzeptstudie zur Entwicklung der neuen, an die bestehende AVES-Umgebung gekoppelten UnrealViewer-Anwendung durchgeführt wird. Zunächst werden allgemeine Anforderungen an ein visuelles System der Flugsimulation spezifiziert, und der aktuelle AVESViewer wird analysiert, wie diese erfüllt werden. Für die meisten von ihnen konnten UE4-Techniken vorgestellt werden, die den gleichen Zweck wie die AVESViewer-Techniken erfüllen. Die beiden Ausnahmen werden eingehend untersucht.

Die erste ist, dass der Modellimportprozess mit dem Ziel der Wiederverwendung der Höhen- und Bilddaten des Geländemodells und der FLT-Flugzeugmodelle entwickelt wurde, da UE4 diese nicht nativ importieren kann. Beide Modelltypen können nur einen Kompromiss zwischen optimaler Leistung, visueller Qualität und Benutzerfreundlichkeit darstellen, und obwohl das volle Potenzial nicht ausgeschöpft wurde, zeigte sich die generelle Möglichkeit, die Modelle zu importieren.

Der andere Aspekt der Konzeptstudie, die Steuerbarkeit über das AVES Visual Simulation Animation Protocol, wurde erfolgreich in Kombination mit einem Konzept zur Konfiguration über XML und einer geringen Eingabeverzögerung gezeigt.

Contents

List of Figures

List of Tables

Listings

Abbreviations

AVES	Air Vehicle Simulator
CPU	Central Processing Unit
DAE	Digital Asset Exchange
DDS	DirectDraw Surface
DLR	German Aerospace Center
DOF	Degrees Of Freedom
FBX	Filmbox
FLT	OpenFlight
FoV	Field Of View
FPS	Frames Per Second
GUI	Graphical User Interface
IC	Interface Computer
IG	Image Generator
IVE	OpenSceneGraph Binary File
LOD	Level Of Detail
MPCDI	Multiple Projector Common Data Interchange
O2U	OpenFlight To Unreal
OpenGL	Open Graphics Library
OSG	OpenSceneGraph
RAM	Random-access Memory

TGA Targa Image File

UDP User Datagram Protocol

UE4 Unreal Engine 4

UHT UnrealHeaderTool

UTM Universal Transverse Mercator

VSAP Visual Simulation Animation Protocol

XML Extensible Markup Language

1. Introduction

1.1. Air Vehicle Simulator

The German Aerospace Center (DLR) is the research center of the federal republic of Germany, conducting research and development work in the fields of aeronautics, space, energy, transport, digitization, and security. These projects are integrated into national and international cooperative ventures and are leading in their fields. One of DLR's institutes, the Institute of Flight Systems (FT) at the Braunschweig site, is active in flight mechanics, measurement, and systems technology of flying systems. For research purposes, large-scale facilities such as the Air Vehicle Simulator (AVES) are operated. The AVES is a flight simulator of the highest reality [10]. Since 2013 it is available for use by other DLR institutes and departments, as well as external companies and research groups. Three different modules are currently (2020) available, one cockpit from a Eurocopter EC135 helicopter and one from an Airbus A320 passenger aircraft, as well as a cabin module with 16 seats. A fourth module, the cockpit of a Falcon 2000LX airplane, is currently in development. As an example of the AVES simulation architecture, the A320 one can be seen in figure 1.1. The simulation's backbone is a central Ethernet connection between the single parts managed by the Interface Computer (IC) [10]. Various types of subsystems are integrated into the architecture, like the visual simulation. This part, in particular, generates the images, which are used in the projection system.
Two of those are currently available, and both provide 15 projectors to create an extensive viewing range of 240° horizontally and 90° vertically, but only one is mounted on six degrees of freedom (DOF) motion system, while the other can only be used in simulations with a fixed base. Combined with the interchangeable cockpit

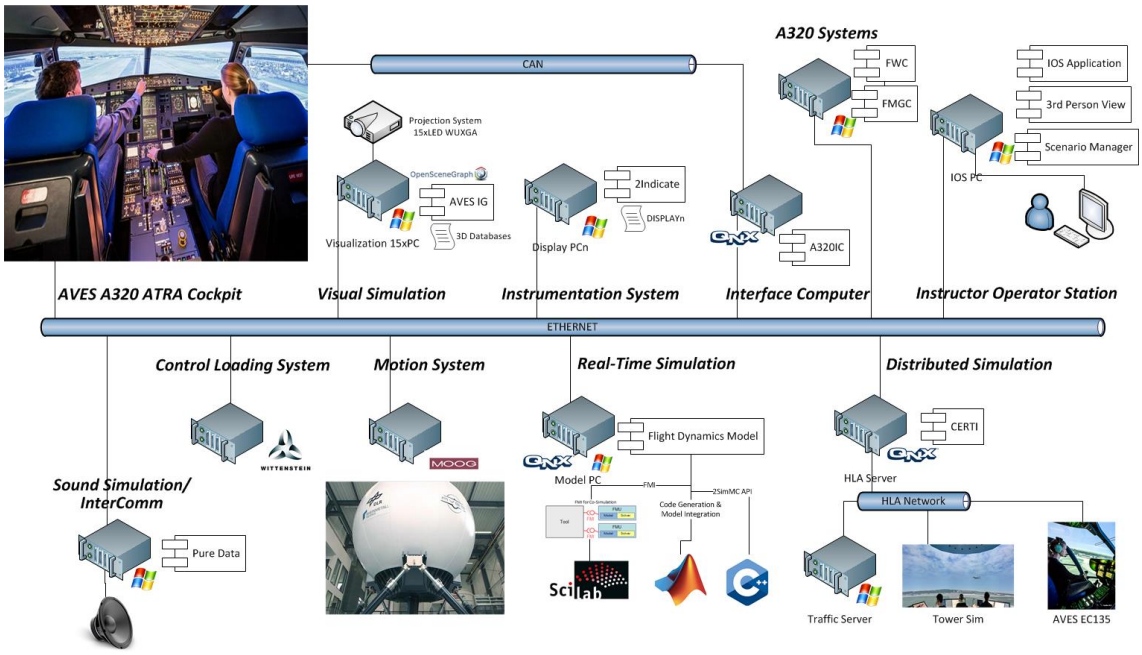


Figure 1.1.: A320 Simulation Architecture

modules, a highly flexible simulation infrastructure is available for research and development purposes [10].

1.2. AVESViewer

To utilize the simulator properly and create an immersive experience, high-quality graphics are required in real-time, which the DLR developed render application AVESViewer can produce. It uses OpenSceneGraph (OSG) as a base and is controlled by the Visual Simulation Animation Protocol (VSAP).
OSG is a scene graph application that structures the rendered scene into a graph and performs multiple traversals to render the scene in a performant manner. It acts as a middleware between the low-level Open Graphics Library (OpenGL) and the AVESViewer.
VSAP is a DLR developed protocol that handles the communication between the IC

and the AVESViewer and is based on a User Datagram Protocol (UDP) connection. The AVESViewer renders large virtual worlds, like a whole map of Germany, with 60frames per second (FPS) in real-time. The Trian3DBuilder creates these worlds with data from multiple satellite and aerial images and measurements. Therefore a high degree of realism and accuracy can be achieved. Additionally, the render application allows simulating traffic by rendering multiple different aircraft models, which are also controlled via VSAP.

1.3. Aim of this Work

In 2019 it was announced that OSG is no longer being developed and will continue only in a maintenance phase [44]. Therefore a replacement for the AVESViewer needs to be found in the midterm, and this work carries out a concept study to determine if the widely-used Unreal Engine 4 is applicable in replacing OSG concerning two main aspects: The reusability of the existing aircraft models and terrain data, and the controllability via a User Datagram Protocol (UDP) connection, especially with the VSAP.

Therefore general requirements for a visual simulation in the area of flight simulation are specified in chapter 2. Afterward, in chapter 3, the current AVESViewer system is analyzed to identify how these requirements are fulfilled, and chapter 4 describes the possibilities of achieving this with the Unreal Engine. Based on these understandings, the criteria for the concept study are designed in chapter 5. The new import process is described in chapter 6 and the implementation of the VSAP-controllable UnrealViewer application in chapter 7.
The results are presented, and the implementation is verified if the criteria specified in the concept study design are met, and future challenges for productive usage are discussed.

2. Visual Simulation Requirements

2.1. Performance Requirements

2.1.1. Frame Rate

The most prominent performance requirement is the frame refresh rate, generally speaking, the higher the FPS are, the better and smoother the user experience is, but everything above 50 FPS is sufficient to avoid negative effects, like flickering [3, Chapter 8][35]. Even higher frame rates do not provide a much better result, especially in flight simulations with civil aircraft, where the camera rotations and movements are usually relatively slow.

2.1.2. Distributed Simulation Architecture

Exemplary for a professional flight simulation architecture , the AVES A320 one was presented in section 1.1. It uses the approach of distributing the single parts of a simulator onto multiple machines, as a single one can not handle the high number of highly complex subsystems due to performance reasons [3, Chapter 8]. Therefore some communication protocols between the systems must be implemented.

2.1.3. Input lag

The last performance-related requirement is the input lag, which describes the time between the pilot's input and visual cues. The delay is usually composed of many single sources. For example, the network connection between the single parts of the simulator may cost several tens of milliseconds, but also the render application may take up to one complete frame until the inputs are processed. Generally speaking, the input lag shall be as low as possible, and the more demanding flying an aircraft is, the more crucial a low input lag is [7].

2.2. Visual Requiremmcpdients

2.2.1. Terrain model

Realistic visuals can only be achieved if a realistic virtual world is present. As the terrain model is the main component of the world, it must meet high-quality requirements while still providing an acceptable performance [35]. The model is often of the size of countries, continents, or the whole world and requires a high resolution if the simulated aircraft is close. This represents a challenge for the visualization due to file sizes of multiple tens of gigabytes and a challenge for the creation. Designing a model of this size by hand with the required grade of detail is very time-consuming and expensive. Therefore the terrain shall be created automatically based on real-world data, like aerial and satellite images and elevation models.

2.2.2. 3D Assets

3D Assets can be placed inside the world to improve it further. Those can be static assets like trees, rocks or houses, or dynamic ones like traffic. The static models break the terrain model's relatively flat surface and create a more realistic skyline, especially noticeable for flights at low altitudes. They also provide a size reference

for the pilot, improving their feeling for distances [3, Chapter 8][35]. On the other hand, dynamic assets mainly increase the number of possible scenarios, which can be simulated. A landing approach to a busy airport or an aerial refueling scenario is not possible without movable aircraft.
Nevertheless, the 3D Assets must be of sufficient quality. This requires them to consist of a correct geometry as well as high-quality textures. In combination with a good lighting and shadowing model, this can be very beneficial for perceiving the position and size of objects [48].

2.2.3. Effects

Another aspect of an excellent visual simulation is using realistic effects. Weather, clouds, water, or light effects are essential topics [3, Chapter 8]. A simulated thunderstorm can not create the same immersion if neither clouds are visible nor rain is falling, but it does not only look wrong, it also decreases the quality of a pilots performance in the simulator, as those visual cues are missing and with them, the perceived information about the virtual world [36].

2.3. Multichannel Projection

A multichannel projection system is required due to the needed high resolution for the visual system, which is necessary because of the required field of view (FoV) and pixel density. A wide horizontal FoV improves the pilot's peripheral vision, which "is the primary source of information for body orientation, direction of travel, changes in direction of travel, approximate speed, and objects entering view from the side" [35]. A large vertical FoV is primarily indispensable in the context of rotorcraft simulations [10]. If the pixel density is too low, and therefore the pixel size is too big, they can be perceived by the pilots as a single pixel and not as a whole picture, limiting the pilot's capabilities in the simulator and reducing their performance [13].

This resolution and FoV is currently only achievable with multiple projectors.
A multichannel dome projection is a common approach to align those projectors. However, for this setup, additional problems arise: the dome itself introduces distortion to the images, the edges of the projectors must be blended to avoid a different brightness on the overlapping areas, and the projectors must be aligned with each other to create seamless transitions without visible steps between them.

3. AVES Visual System Environment

3.1. Tools and Programs

The AVES Visual System consists of multiple tools and programs to prepare and generate three-dimensional models and display them on the projection system:

AVESViewer Rendering application based on OSG

OSG Open source 3D graphics toolkit

Trian3DBuilder (TrianGraphics GmbH) 3D terrain generation software

osgconv OSG based program to convert models between different file formats

3ds Max (Autodesk Inc) 3D modeling and rendering software

The primary tool is OSG, which is used as a middleware between the AVESViewer render application and the low-level graphics application programming interface (API) OpenGL. Access to the source code of the open-source software is a vital aspect of OSG. It creates high expandability and flexibility for the AVESViewer because every aspect can be altered or expanded, which helps to match the versatile use cases in the context of research and development [10]. As an example, the viewer did not support to render many light sources and was expanded to use a modern light rendering approach, which enabled the AVESViewer to show thousands of dynamic light sources that increase the realism of night scenarios [31].
With the Trian3DBuilder, 3ds Max, and osgconv the diverse model import processes of the AVESViewer are created. They are used to create aircraft and terrain models, with data from multiple sources, and convert them into the OpenSceneGraph Binary File (IVE) format, to be rendered efficiently with the OSG based AVESViewer.

3.2. Terrain Model

The Trian3DBuilder handles the creation of the virtual worlds for the AVES simulation. A brief overview of the process is shown in figure 3.1.

Figure 3.1.: Current Creation Process of the Virtual Worlds

The builder can create the models in various coordinate systems. For example, a geocentric model preserves the earth's curvature. It represents a segment of the earth's ellipsoid directly, while a Universal Transverse Mercator (UTM) model represents the earth's surface projected onto a flat plane. The AVESViewer currently uses models based on those two coordinate systems, geocentric and the UTM.
In the selected coordinate system, a rectangular grid is defined, and the size of the finished model is defined by how many tiles the grid has and how big each tile is. Figure 3.2 shows a grid in the UTM coordinate systems with three by two tiles and a tile size of 4 km by 4 km.
To create the models, the Trian3dBuilder reads color and height information from input files and connects them based on their position in the coordinate system. The final model's geometry is built from the height information and the texture of the color information. The density of the model's vertices and the texture resolution are essential criteria for the model's quality.
The Trian3DBuilder can refine the terrain model by placing additional 3D Assets into the virtual world. The extra airport module of the Trian3DBuilder is capable of inserting airport-related models into the terrain model, e. g. a runway can be baked directly into the ground and replace the previously used aerial or satellite image at that point. In addition, automatically generated light sources can be created for the simulator [31].
After the terrain model designing is finished, it can be exported into the IVE format. This is the native OSG binary format for computer models and can be used directly by OSG and, therefore, by the AVESViewer. No other conversion tools or scripts

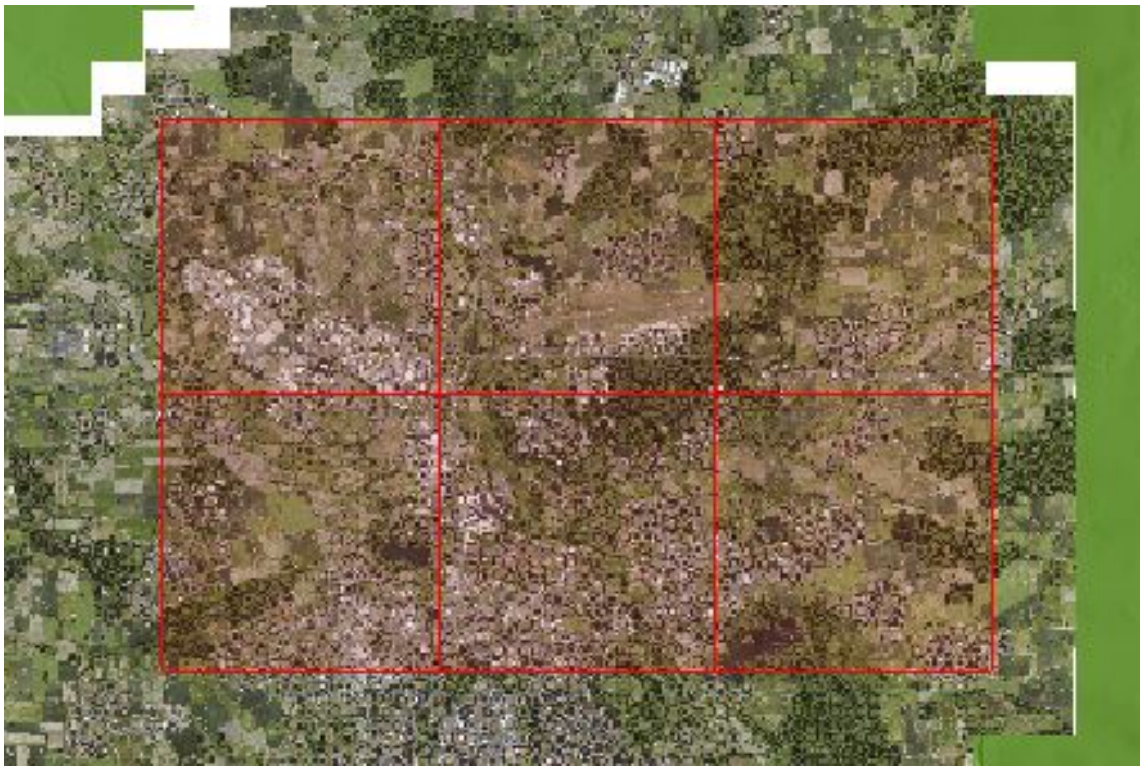


Figure 3.2.: UTM grid in the Trian3DBuilder

are necessary. The export process supports the OSG technique of PagedLOD, and exporting light sources [31].

The PagedLOD system is one of the multiple OSG techniques used in the AVES environment to increase the terrain models' performance. It combines levels of detail (LODs) with dynamic data loading.
LODs reduce the large virtual world's performance impact, especially when combined with the substantial viewing distances of multiple tens of kilometers. For each tile of the world, multiple models are created by the Trian3dBuilder, each with a different grade of detail. Lower details enable faster rendering times, and especially at far distances, where an object occupies only a few pixels on the screen, fine details are not distinguishable. Therefore a level of the model with a low grade of detail can be used [3, Chapter 8][17].
The dynamic data loading helps manage large terrain models, which can easily exceed file sizes of multiple gigabytes, and therefore loading them entirely into random-

access memory (RAM) is not possible due to size limitations of the used hardware. The OSG PagedLOD technique loads only the viewable LOD and all lower detail levels of the terrain model so that only the close tiles have their high detail and memory-intensive levels in RAM, helping to reduce the amount of data down to a manageable size [4].

Additionally, different areas of the terrain need different grades of details. The areas close to an airport require the highest detail LODs, while the areas where the aircraft is in cruise height at high altitudes are only barely visible and need close to no details. The PagedLOD system helps mitigate problems in terms of RAM usage, but the disk space wasted by unused high detail LODs is considerable, as well as the central processing unit (CPU) overhead, which is created by managing a large amount of PagedLODs.
As the LOD settings are changed for every tile of the grid simultaneously, one single grid forces the user to compromise between the detailed and vague areas. This is solved by the Train3DBuilder multigrid feature, which allows defining multiple grids and exports them into the same terrain model. For example, the areas close to the airport can be cut out of the primary grid and replaced with a detail grid.

3.3. 3D Assets

The DLR has a large collection of aircraft models from a large variety of sources. Thus, the aircraft models' import or creation process differs more than the one for the terrain. The models are present in numerous file formats, e. g. OpenFlight (FLT), digital asset exchange (DAE), Filmbox (FBX), or Microsoft specific file formats and can be imported, or they are directly created via the 3ds Max modeling software. This work focuses on importing models in the FLT format, as this is a standard file format in the simulation industry and more than 100 aircraft models are available in this format for usage in the AVESViewer.
The models are loaded into 3ds Max via the included FlightStudio utility. Afterward, they are exported in the COLLADA format DAE and converted with the osgconv

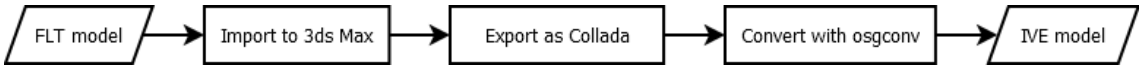


Figure 3.3.: Current OpenFlight model preparation

tool into the IVE format. Without any conversion during the loading, the OSG native format can be loaded faster in the AVESViewer. As the FLT import process uses 3ds Max as a tool, native 3ds Max projects are handled in a very similar way and do not differ by a lot, which is also true for FBX files, as they are imported into 3ds Max as well and after that, follow the same steps as the FLT files.
Similarly to the terrain models, the aircraft models also use LODs, to ensure adequate performance. Still, no dynamic data loading with PagedLOD is used because the file size of an aircraft model is only around a few megabytes, and they do not use a lot of RAM space, so the CPU overhead of managing dynamic loading can be avoided. To animate the single parts of the aircraft, like the ailerons or gears, the model's structure must follow a specific naming convention, the AVES Spec. On simulation start, the AVESViewer parses the model and collects the corresponding nodes. Consequently, the preparation process for the aircraft models needs a modification, as shown in figure 3.4. An additional step is introduced, which parses the Extensible Markup Language (XML) based DAE files and renames the model structure to match the convention.

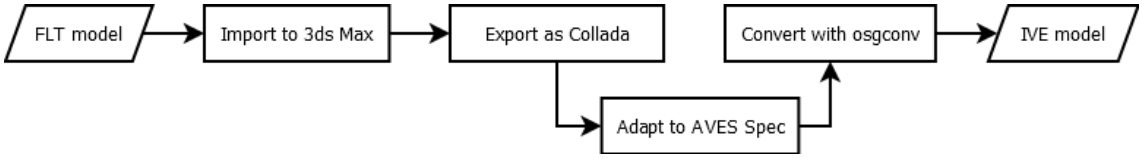


Figure 3.4.: Current OpenFlight Model Preparation AVES Spec conform

3.4. Environmental and Visual Effects

The environmental and visual effects shown in the AVESViewer are partially bought from a third party company and partially made in house. One of the DLR developed is the light and shadow model, which was only possible due to the open-source of

OSG [31][32].

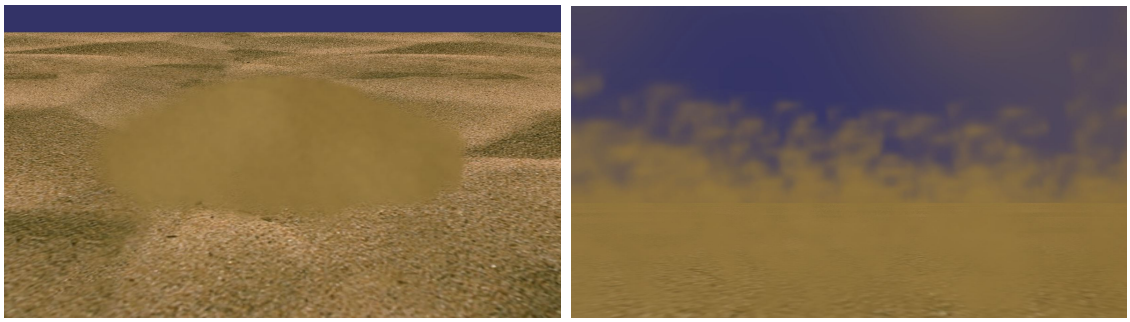


Figure 3.5.: Brown Out Effect

An effect, where a helicopter pilot's view is obstructed by surface material raised by the downwash of a helicopter close to the ground, the so-called brown out effect was also implemented by the AVES team [14] (see figure 3.5).
On the other hand, the weather, cloud, and ocean simulation are shown in figure 3.6a. They are powered by the SilverLining Sky & 3D Clouds and the Triton 3D Ocean & Water SDK, which are both published by Sundog Software LLC and provide an interface for the usage with OSG.
However, as the area of effects is one of the fastest developing in computer graphics and many new techniques are published every year [27][28], the rendering application needs to be continuously improved to stay up to date.

(a) SilverLining Sky & 3D Clouds **(b)** Triton 3D Ocean & Water SDK

Figure 3.6.: Environmental Effects (Source: `https://sundog-soft.com/evaluate/gallery`)

3.5. Visual Simulation Animation Protocol

For the task of distributing the simulation, many custom protocols were developed in the AVES environment. One of them is the Visual Simulation Animation Protocol (VSAP), which can control multiple instances of the AVESViewer on different machines via a UDP connection. It is a protocol with a fixed-size header and a dynamic payload, containing an arbitrary amount of VSAP objects. A total amount of 19 different object types are currently supported, but some are only available for legacy reasons.
In the following, this variable data structure is explained with an example message containing a VSAP `MODEL_GEO` object. This represents an input data stream for an animated 3D model, with positions in the geodetic coordinate system.
The first structure, which will be transmitted, is the header of the message, which consists of different meta information and the number of VSAP objects. Those objects represent the payload of the data and are directly following after the header in the data stream.

Listing 3.1: VSAP Header Definition

```
1 // Header for visual simulation animation protocol
2 typedef struct
3 {
4     float vers;     // Protocol version
5     int size;       // DEPRECATED
6     int reply;
7     int objCnt;     // Number of objects
8 } VSAP_HDR;
```

As the object length differs per type, each object starts with a separate object header. This structure contains the type information and the index, which is only relevant for some objects. The `MODEL_GEO` object is one of those because it is the data to animate this one specific 3d model assigned to the index.
One additional feature used for `MODEL_GEO` objects is using setup objects. Those are optional and can be appended to the `MODEL_GEO` object. How many of those are present is also specified in the object header, and parsing them follows a similar approach as with the main objects. They have their type information at the same

position, which is used to parse the setup object and determine the length. The next object begins directly after the current one so that the position of the next type information is known, and the parser can iterate over the objects.

Listing 3.2: VSAP Object Header Definition

```
1  // Defines the data stream for the general object header
2  typedef struct
3  {
4     int iType;         // Object typ
5     int iLen;          // DEPRECATED
6     int iIndex;        // Model or intersection index or NULL otherwise
7     int iSetup;        // Number of setup objects
8     double dSimTime;   // Simulation time of this object
9  } VSAP_OBJ_HDR;
```

The `MODEL_GEO` itself consists, without the setup objects, only of its main payload, a `MODEL_GEO_DATA` structure, in addition to the object header. The payload contains the information required to set the position and rotation of the indexed 3D model, and the next VSAP object starts directly after the last byte parsed for this `MODEL_GEO` object and its setup data, until the last object specified by the `objCnt`, in the message header, is parsed.

Listing 3.3: VSAP Object `MODEL_GEO` Definition

```
1  // Defines the geodetic data stream for an animated 3D object
2  typedef struct
3  {
4     double lat;    // Latitude coordinate [deg]
5     double lon;    // Longitude coordinate [deg]
6     float height;  // Height value [m]
7     float h;     // Heading angle [rad]
8     float p;     // Pitch angle [rad]
9     float r;     // Roll angle [rad]
10 } VSAP_OBJ_MODEL_DATA_GEO;
11
12 //! Defines the geodetic data stream for an animated model
13 typedef struct
14 {
15    VSAP_OBJ_HDR hdr;          // Header with basic information
16    VSAP_OBJ_MODEL_DATA_GEO data; // Model data
17 } VSAP_OBJ_MODEL_GEO;
```

4. Unreal Engine 4

4.1. General

UE4 is developed by Epic Games and is currently one of the most spread graphics engines and is used to develop many graphics applications [5][41]. It combines high visual fidelity with a high-performance rendering system for the final application as well as a powerful editor for content creators and developers [38]. The engine and the editor are written in C++, and the source code is entirely open-source, which enables the user to change everything to their needs. Using the software comes without a royalty, except for commercial off-the-shelf products with gross revenue exceeding one million USD. After accepting these terms and conditions, access to the full source code is granted via GitHub [11], but the engine can also be installed with build binaries and debug symbols via the Epic Games Launcher to avoid building it from source.

Due to the extensive usage in the game industry and community, many online resources like forums or tutorials can be found to increase the development speed and lower the entry barrier in the engine. In addition, stream recordings are available for many engine features, where the internal staff presents and explains one particular feature of the engine over for 1 hour, and answers questions from the community to explain how the feature is intended to use.

The provided editor helps manage, arrange, and import the assets and control the gameplay. The latter can be done with two different methods, writing C++

code or using the Blueprints Visual Scripting system, which is a node-based interface to the engine. The main workflow idea is to create custom Blueprints as C++ classes with an external text program or integrated development environment (IDE) and provide them to the editor users so that they can combine them into the bigger picture. For example, an actor's capabilities to be controlled via an external protocol, like VSAP, should be implemented in C++ and exposed to the Blueprint system. Afterward, an Instance of this Actor is created, and the aircraft model is assigned to it in the editor. This separates the low-level task of creating those high-performance low-level parts in C++, from combining them and creating the whole experience in Blueprints, which enables persons without programming skills and knowledge of the underlying implementation to develop aspects of the game.

4.2. Build system

The custom UE4 build system provides a system to separate the source code in different modules. It is a process based on a C# file per module in which the dependencies and build rules for the module are declared.
Apart from the modules, the build system also uses the UnrealHeaderTool (UHT) to create the necessary code for the reflection system so that C++ classes can be used in Blueprints. The UHT utilizes the `UENUM`, `UCLASS`, `UINTERFACE`, `USTRUCT`, `UFUNCTION`, and `UPROPERTY` macros. Each macro is used for a different part of the C++ code and indicates the UHT to generate code for this particular part, which handles multiple purposes. For example, the `UFUNCTION` macro can be used with the `BlueprintCallable` argument, which causes the reflection system to generate the necessary code, so that this function can be called in Blueprints [22].

4.3. C++ Object Handling

The Object Handling system of UE4 revolves around the `UOjbect`, every object of a class that inherits directly or indirectly from `UObject` underlies special rules. A garbage collection system manages those objects, and as soon as no reference points to the object, it will be freed in the next garbage collection cycle. A raw C pointer is not sufficient to count as a reference and prevent the object from being garbage collected. The pointer must be either declared with the `UPROPERTY` macro or stored in a UE4 container, like `TArray`. References declared in this way are automatically set to null on initialization, and if it is a reference to an `AActor` or `UActorComponent`, it will be updated to null on the deletion of the referenced object [46]. The actor class is another of the main classes in UE4 and extends an `UObject`, with the possibility to place it in a virtual world.
UE4 provides a naming scheme for the class names, to differentiate them based on the first letter of their names [6]:

U Inheriting from UObject

A Inheriting from AActor

T Template classes

I For interfaces

E For Enums

F For everything else

4.4. Asset Types

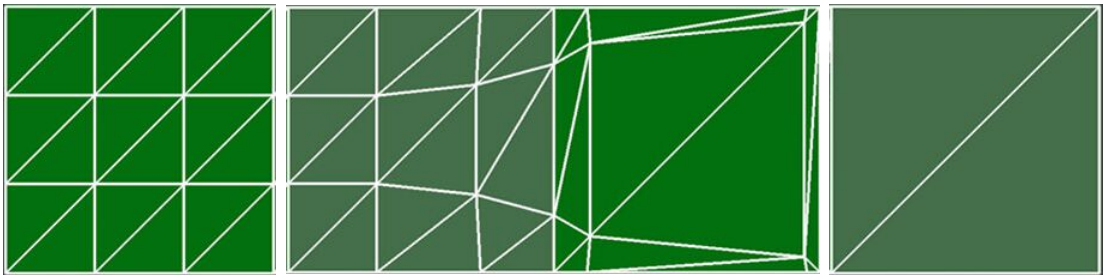
The engine provides two different systems to represent external models, Static and Skeletal Meshes. The former is an efficient structure that uses caching to lower the model's performance impact, but the assets ca not be modified to create animations [39]. On the other hand, the Skeletal Meshes consist of "a set of polygons composed to make up the surface of the Skeletal Mesh, and a hierarchical set of interconnected bones which can be used to animate the vertices of the polygons" [37], but they perform worse in comparison with static meshes, and they can not be created or edited from within the UE4Editor [37]. Therefore an external editing application is necessary.

Importing assets into UE4 can be done with FBX files or with the Datasmith plugin. The FBX pipeline handles more complex meshes and provides more features than the Datasmith plugin. For example, Skeletal Meshes are not possible via the Datasmith plugin and can only be imported as an FBX file [2]. On the other hand, Datasmith provides a larger variety of supported file formats, including the uDatasmith format [9], and can import multiple assets combined in a Datasmith scene while maintaining the single assets' hierarchy and relational positions [2].
Multiple companies of the area of flight simulation are currently working on providing a Datasmith import pipeline for their products. The OpenFlight To Unreal (O2U) plugin developed by Presagis introduced in 2019 aims to streamline importing FLT models into UE4 by converting them to a Datasmith scene [15][25][42]. Moreover, TrianGraphics extended their Trian3DBuilder with an optional Plugin to directly transfer the models to UE4 via the Datasmith [44].

4.5. Landscapes

The UE4s landscape technique is specialized for creating vast terrains and performs better than static meshes due to a lower memory footprint of roughly one-sixth [19]. This is possible because the floor of the world consists of a quadratic grid so that only the height information must be stored and the x, y, and UV information can be easily calculated on the fly by knowing which point of the grid is currently considered.
The up to 16 bit height information per grid point is imported via an image file, a so-called heightmap. The images are usually a single channel Portable Network Graphics (PNG) or RAW image file with 8 or 16 bits of precision, and internally the heightmaps are handled like a texture and can benefit from all the regarding optimizations of UE4, like texture streaming for dynamic loading or mipmaps to create a lower detail representation.
The landscape system's additional features are smoothed transitions between two mipmap levels with a morphing effect, shown in figure 4.1, and procedural foliage. The latter can cover large areas of the world with very high detail vegetation with

nearly no memory usage. The system does not store every single tree and its position. It only stores the area where a forest shall be and places the trees dynamically [26].



(a) Fully LOD 1 **(b)** Morphing from LOD 1 to LOD 2 **(c)** Fully LOD 2

Figure 4.1.: Continuous Geo-MipMap LOD (Source: [19])

4.6. Dynamic data loading

4.6.1. Texture streaming

The dynamic data loading processes of UE4 are called streaming, and there are many different types for multiple use cases. One example is the already mentioned texture streaming, which calculates for each visible texture the best mipmap level and loads it asynchronously into memory. At this stage, the loading processes are dynamically prioritized and adapted so that the most critical textures are loaded first. The importance can be set by the designer or automatically calculated based on the screen size of the texture [40].

4.6.2. Level streaming

Another streaming system is level streaming. A virtual world can be separated into different sublevels. These can be loaded and unloaded dynamically based on user-defined rules. Therefore only the parts of the world close to the player need to be in memory while the rest is streamed in as soon as the player comes closer.

4.6.3. World Composition

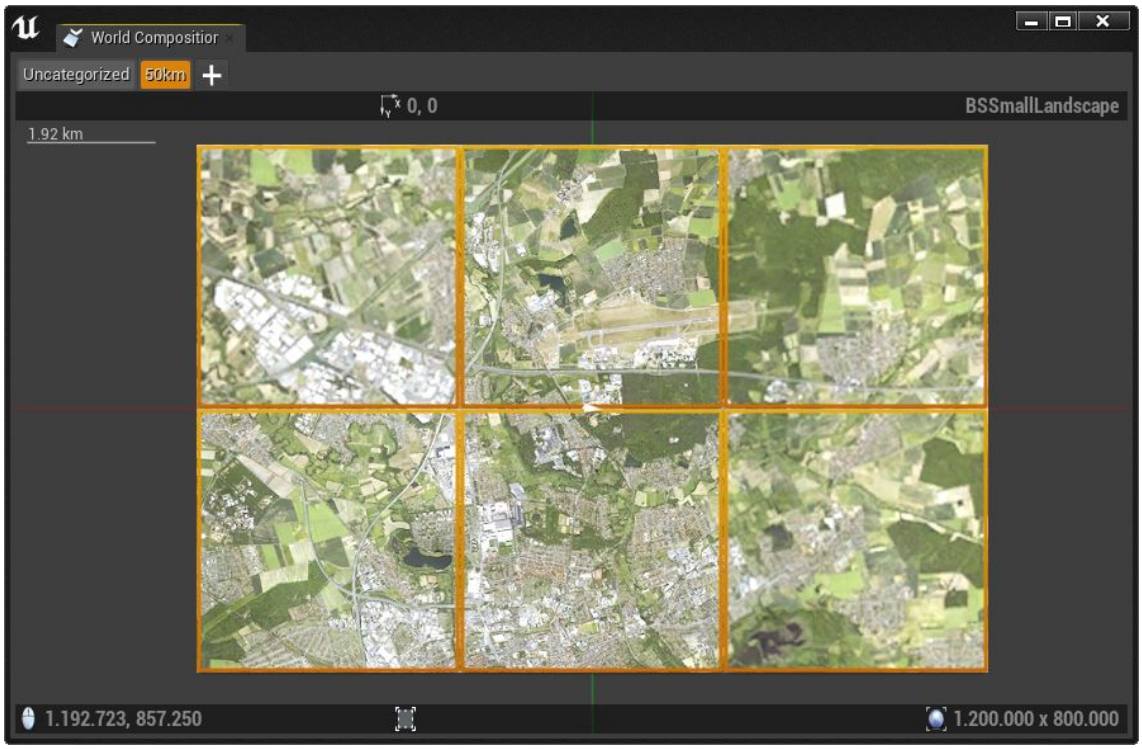


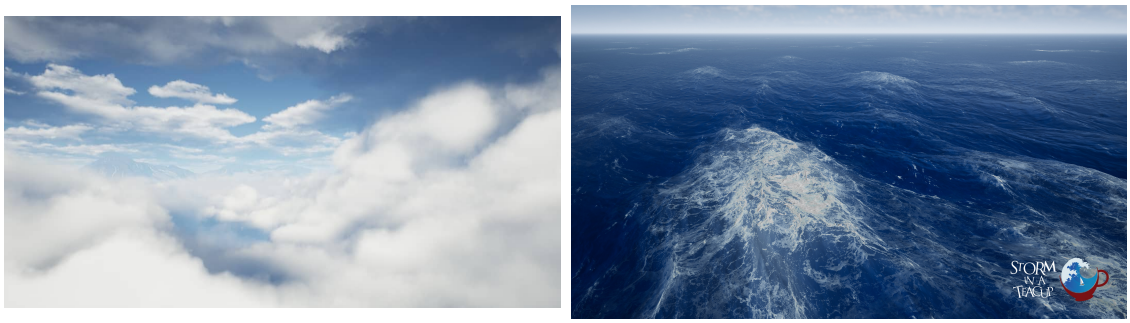
Figure 4.2.: World Composition Minimap View

A specialization of the level streaming technique is the World Composition. It combines the dynamic data loading with automatic sublevel detection and World Origin Shifting.
World composition is activated in the root level, which will be the persistent level and unaffected of the streaming. Every other level placed in the same folder or a subfolder is then detected and added as a sublevel. They are shown in a Minimap view, where they can be rearranged and assigned to streaming layers. The layers are shown in the top row. For example, in figure 4.2, the custom `50km` layer was added, and it is currently selected so that all assigned levels are shown. The layers group the sublevels, and each has its own streaming distance. In the example, the levels assigned to the `50km` layer are streamed in at a distance of 50 km from the player.
Additionally, a LOD system is supported by the level detection. Each sublevel can have up to 4 LOD streaming levels. Those LOD levels follow the naming scheme

`[sublevelname]_LOD#` and are combined automatically. The distance at which the LOD is streamed in is in relation to the base sublevel. For example, if the layer of one sublevel has a streaming distance of 50 km and the first LOD has an offset of 20 km, the LOD is streamed in at 70 km. The LODs can be created by an external application or an auto generator, reducing the contained meshes and landscapes based on reduction settings. [49].
The other main feature of World Composition is the World Origin Shifting. It provides a technique to keep the origin of the world always close to the Viewer. This is necessary because the UE4 position calculations are entirely based on single-precision floats, and therefore, a numerical error is introduced quickly for objects which are too far away from the origin. When the player approaches the border of the area with sufficient precision, the system will shift every actor with the same offset vector closer to the origin, so that the relative positions are preserved [49]

4.7. Environmental and Visual Effects

For the Unreal Engine are already many effects available, either directly provided with UE4 or via third party plugins, and the effect database is increasing with nearly every release of the engine. For example, in 4.24, a new physically based atmospheric sky model was introduced, and with the next 4.26 release, UE4 is expanded by water effects to visualize rivers, lakes, and oceans [45]. The lighting effects are also continuously improved, with 4.25, the ray tracing system is flagged as ready for real-time productions, and the previously existing lighting and shading models are already production-proven with good visual results [18][38].
Furthermore, even if the default implementation is not sufficient enough, the engine can be extended with an own implementation following one of the many available tutorials or with a third party plugin, like the TrueSky plugin from Simul Software Ltd, which renders volumetric clouds, weather effects, or the Realistic Ocean Simulator by "Storm in a Teacup S.R.L." as shown in figure 4.3a[30].
Due to this active development and continuous improvement in the area of effects



(a) TrueSky weather simulation (Source: Instagram `@truesky_simul`)

(b) Realistic Ocean Simulator (Source: `http://www.stcware.com/rd/`)

Figure 4.3.: Environmental Effects

and the already large amount of available and high-quality effects, the requirements of a visual simulation regarding effects can be fulfilled sufficiently.

4.8. nDisplay

UE4 provides a system for rendering with multiple Image Generators (IGs) simultaneously, nDisplay. It uses a custom Transmission Control Protocol (TCP) connection to communicate and synchronize between the different IGs. A synchronized state is achieved, by synchronizing information and events, like the actor's location or the frame start, and combining them with deterministic behavior. For example, if the player opens a door in the game, only the event when this animation starts must be synchronized as long as the action results in a deterministic behavior of the scene so that each IG will generate precisely the same images.
In section 2.3, other issues regarding a multichannel projection are mentioned, like dome distortion, projector alignment, or edge blending. These can also be solved by the nDisplay system, which is already production-proven in dome arrangements [8][33]. However, the task of creating a correct configuration file can be very time consuming [29][21], although the VESA Multiple Projector Common Data Interchange (MPCDI) standard is supported and a third-party applications from the Scalable Display Technologies company [34].

5. Concept Study Design

5.1. General

Within the scope of this concept study, the UnrealViewer will be developed. It is an executable program that shall be coupled with the AVES simulation environment. It answers the questions to be examined: Is it possible to import the existing model data into the application, and is it possible to control an instance of the UnrealViewer via the Visual Simulation Animation Protocol?

5.2. Import Process

Two new import processes need to be developed, one for the aircraft model and one for the terrain data. The following four criteria are used to determine if the new processes are considered as successful:

- Use existing data (FLT models, and elevation and image data)
- A simple process with as few tools, scripts, or programs as possible
- Create high-performance models, especially utilize LOD and streaming techniques
- Enable animations for the aircraft

The Trian3DBuilder can create two file formats, which the UE4Editor can read, FBX and Datasmith, but the FBX exporter only supports the Unity engine and is therefore not usable for this work. On the other hand, the Datasmith exporter was firstly announced in 2019 and is very new [44] and might not be completely mature. This must be tested, especially if the exporter creates usable results, and if additional preparations of the models are necessary before they can be displayed with sufficient performance. Especially the use of the UE4 World Composition system needs a more in-depth investigation, as it is irreplaceable for large scale maps, due to the World Origin Shifting.
Two different options are available for importing FLT models, using the O2U plugin by Presagis or via 3ds Max. Both are promising approaches with O2U as the much simpler tool and 3ds Max as the already used tool, but due to the limitation that UE4 can not import skeletal meshes via Datasmith, it is not possible to import animated aircraft via O2U. Other widespread 3d modeling tools like Cinema4D or Blender do not provide an importer for FLT models and are therefore not usable [16][23]. 3ds Max is the only option remaining, which supports not only the in-depth investigated FLT models, but also many more formats.

5.3. Controllability

The UnrealViewer aims to create not only one finished scenario with one fixed terrain and only a predefined set of aircraft but to make the scenario highly configurable and dynamic. The loaded terrain, as well as the aircraft, shall be configurable via an XML file or directly controllable via the VSAP. The quality of the implementation is determined with the following four criteria:

- Configure the scenario with an XML file (terrain and aircraft models)
- Setup the VSAP connection with an XML file
- Control the aircraft position and rotation via VSAP
- Low control input latency

6. UnrealViewer Import Process

6.1. Terrain Data

The Trian3DBuilder offers two different model types of how the terrain is exported, either as a static mesh or a landscape. In this section, the better-suited model type is selected with the following criteria and afterward evaluated according to the requirements presented in section 5.2:

- Multigrid support
- Model coordinate system
- World Composition

6.1.1. Multigrid support

A Trian3DBuilder feature commonly used in the AVES environment, multigrids, is currently unusable in combination with the Datasmith exporter, regardless of whether it is exported as a static mesh or as a landscape. No official documentation of TrianGraphics does confirm this, but the software crashes regardless of the tested configurations, which create correct multigrid terrains with other exporters like OpenSceneGraphs IVE.

6.1.2. Model coordinate system

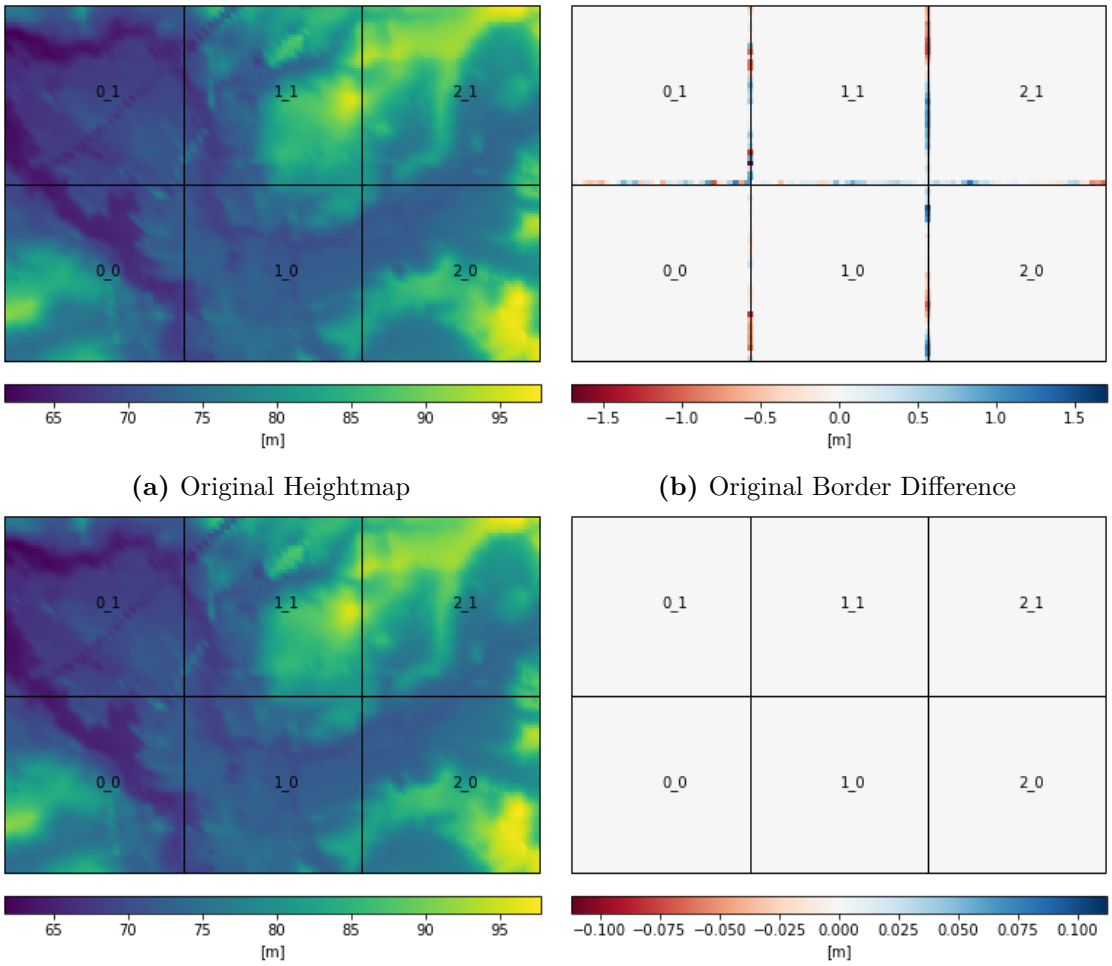
The AVESViewer currently supports two different coordinate systems for the terrain models, UTM and geocentric (See section 3.2). The main difference between the models is that the earth's surface is projected onto a flat plane for a UTM model, while the geocentric model has a curved surface.
This curvature is a problem for a UE4 landscape system, as the system is based on a flat and regular grid, and only the heights for each point are stored. A similar approach, where only the distance to the earth's center is stored in the heightmap, might be possible in theory, but it must be added manually to the engine. On the other hand, a UTM model can be easily represented by a landscape due to the flat plane's projection.
The situation for static meshes is similar: UTM works and geocentric does not, but the reason is different. A static mesh can represent a curved surface, but for a geocentric model, the reference point and, therefore, the world origin is at the center of the earth. This results in positions for the static meshes, where the system of the single-precision position of UE4 already creates significant errors, which results in unacceptable gaps between the single tiles, as shown in figure 6.1.

Figure 6.1.: Gap in geocentric terrain

6.1.3. World Composition

The Trian3DBuilder generates for each tile of the terrain a single uDatasmith file and one combined asset file for reoccurring assets, like trees, materials, or textures. The default process for importing in the UE4Editor is importing the asset file in the root content directory and using a special import widget provided by TrianGraphics for the single levels. Each tiled uDatasmith file is imported directly into a sublevel of the currently opened level, without creating correct sublevel files on the file system. As the World Composition changes how sublevels are detected and the new one is based on an automatic approach looking in specific directories, the level structure created by the TrianGraphics widget is not suitable for World Composition. Therefore a custom widget was created, which is based on the TrianGraphics one, but it takes care of automatically importing the assets file as well as importing each tile explicitly into a separate sublevel file in the correct folder structure (See section 6.1.4). Another task solved by the import widget is creating the persistent base level for the World Composition. This level is created as a copy of a default map, where the World Composition system is already activated.
As a result of the changed structure, the asset file's paths in the uDatasmith tile file are broken and need adaption. For static meshes, the meshes' materials are defined within the level, and the path to the textures needs adaption, while the materials of the landscape files are defined in the assets file, and therefore the path from the model to the material has to be corrected. Two python scripts were developed for this task. Python was selected because of its widespread usage in the area of model pipeline automation. Both UE4 and 3ds Max support python scripting for manipulating files.
With this `FixStaticTerrain.py` script, a static mesh can be successfully imported with World Composition, while the landscape terrain pipeline must solve another challenge. Exporting tiled terrains as landscapes are officially not supported by the Trian3DBuilder [43, p. 270]. UE4 expects the adjacent landscape tiles to share their border grid points, so the same height must be stored in both heightmaps to create a smooth transition without sudden jumps or wholes in the terrain [20]. This requirement is not fulfilled by the Trian3DBuilder, as shown in figure 6.2, the

`FixLandscapTerrain.py` script is therefore extended to also correct this misalignment.



(a) Original Heightmap

(b) Original Border Difference

(c) Adapted Heightmap

(d) Adapted Border Difference

Figure 6.2.: Adaption of a tiled heightmap created by Trian3DBuilder

Figure 6.2a shows the combined heightmap of an example terrain, with three by two tiles. The example uses a very low-resolution heightmap of 64 by 64 pixels per tile to increase the plot's visibility, but the real used terrain with a resolution of roughly 500 by 500 shows the same effect. The differences at the borders are shown in figure 6.2b and are roughly between −1.7 m and 1.5 m.
The python script parses all heightmap files and averages adjacent tile's border pixels to the same value. Additionally, all height values are adapted so that they are

transformed in the same manner. That is necessary because UE4 transforms the 16 bit height information to the specific use case, e. g. in a very flat environment with a very fine structure, the 16 bit information must contain a very high precision, while a landscape with high mountains and deep valleys requires less precision but a broad range of values. With a different scaling and translation for each heightmap, the borders will not match exactly due to numerical imprecision. Equation (6.1) shows how the `uint16` z_{in} from the heightmap is scaled by the factor s and translated by the offset t to the `float` z_{out}, representing the UE4 height in cm.

$$z_{out} = 512\text{cm} \cdot s \cdot \left(\frac{z_{in} - 2^{15}}{2^{16}} \right) + t \tag{6.1}$$

With the default values $s = 1$ and $t = 0\,\text{cm}$ the heightmap represents a range from -256cm to +256cm, with a difference of 78.125 µm between adjacent height values. The corrected heightmaps in Figure 6.2c are scaled by $s \approx 7.04$ and translated by $t \approx 7972\,\text{cm}$ to create the exact matching borders shown in figure 6.2d and a difference between two height values of circa 0.55 mm.

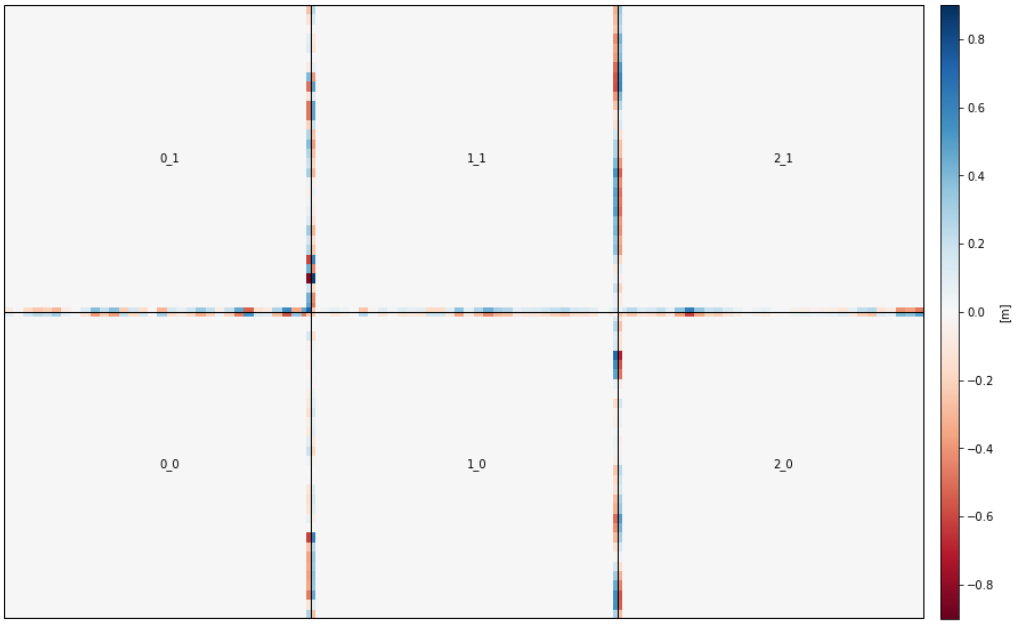
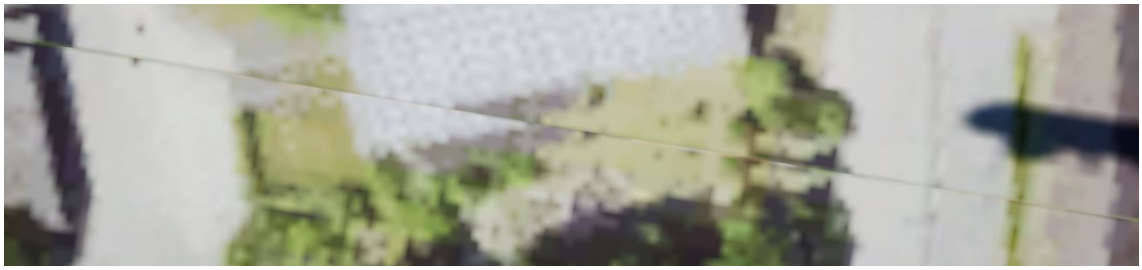


Figure 6.3.: Error introduced by adapting the Heightmaps

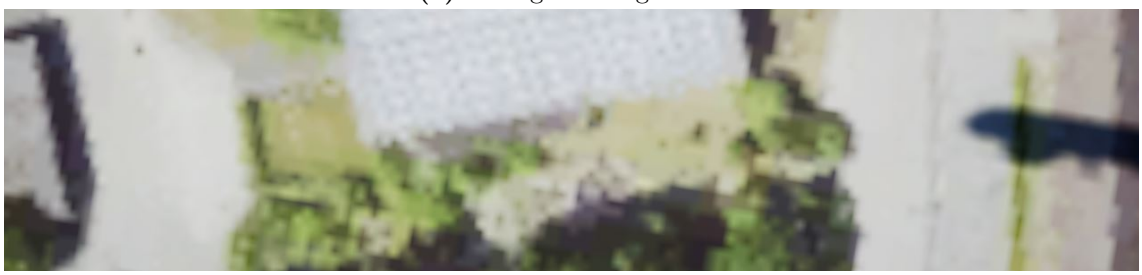
Averaging the border heights and changing the transformation introduces an error to

the terrain's height, which is shown in figure 6.3. In this specific example, the highest absolute error is 0.844367 m, which is acceptable compared to the error introduced by the Trian3DBuilder's terrain creation process. When used for the AVESViewer, the software is usually configured to reduce the terrain model's complexity by removing vertices from the model. Regarding the LOD, this process is usually allowed to introduce an error between multiple meters and centimeters. Therefore, the error introduced by the averaging is in the acceptable range.

Another problem of the landscape models with tiled landscapes is a misalignment of the UV coordinates. Textures created by the Trian3DBuilder can not be fully used, as they have overlapping parts, which is shown in figure 6.4a. A UE4 landscape with 64 by 64 pixels creates UV-coordinates in the range $[0; 63]$. To span a whole image over the landscape, the UV-coordinates must be scaled by 63 and offset by 0 to create the necessary texture look-up coordinates in the range $[0; 1]$.



(a) Wrong UV Alignment



(b) Corrected UV Alignment

Figure 6.4.: Landscape UV-channel problem

The Trian3DBuilder textures have a 4-pixel wide border with duplicate pixels so that the two outmost pixels of the texture must be ignored for a clean border between

adjacent landscapes. This is also done by the `FixLandscapeTerrain.py` script, which calculates the correct scaling factor *scale* and offset values $pan_{U/V}$ for the materials UV channel with equations (6.2) and (6.3). The variables res_H and res_T represent the resolutions of the heightmap and the texture.

$$scale = (res_H - 1) + \frac{4 \times res_H}{res_T} \tag{6.2}$$

$$pan_{U/V} = \frac{2}{res_T} \tag{6.3}$$

6.1.4. Final Import Process

The newly developed four-step process for importing a terrain database into the UE4Editor is shown in figure 6.5. It starts with configuring the terrain model with the Trian3DBuilder and exporting it as a uDatasmith file. After it was corrected using one of the two python scripts, it can be imported into the correct directory structure by using the provided custom import widget. After that, the tiles' level streaming settings need to be adapted.

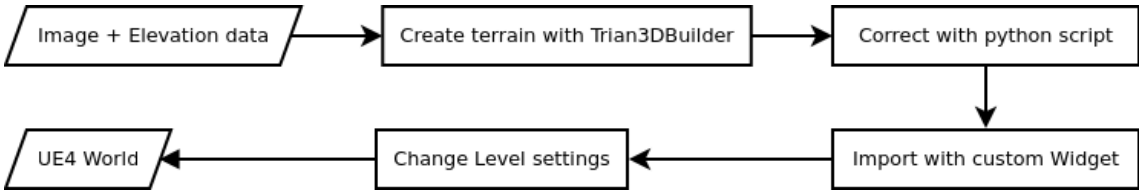


Figure 6.5.: New Terrain Import Process

The detailed settings to choose are explained with an example. Hereby is the Trian3DBuilder project file `Test01` called and stored in `D:/Testing/`. The model is configured with a two by three tile grid, and the primary difference to the current IVE Trian3DBuilder configuration is apart from changing the file format and avoiding multigrids, to create only one LOD. Any other LOD extends the generation process duration without affecting the result. The next step is to run one of the python scripts, which both follow the same syntax:

```
FixLandscapeTerrain.py <InputFolder> <BaseFileName>
```

```
FixStaticTerrain.py <InputFolder> <BaseFileName>
```

The scripts' input folder is where the uDatasmith files are, and the base file name equals the Trian3DBuilder project name, for this example:

InputFolder `D:/Testing/Test01/UnrealDatasmith/terrain/`

BaseFileName `Test01`

The python scripts' outputs are adapted uDatasmith files stored in the same directory, and filenames are prefixed with `FIXED_` to differentiate them from the originals. After that, the files can be imported to the UE4Editor, with the custom widget. It is shown in figure 6.6 with the inputs set according to the example. The `Use Fixed Files` checkbox controls if the corrected files with the `FIXED_` prefix are imported, or the uncorrected original ones. The world name specifies how the root level of the newly created terrain is called, and where it is placed in the content directory of the UE4Editor. This position is where the UnrealViewer application will search for a map with the world name specified via XML.

```
/Game/Content/Terrains/<WorldName>/<WorldName>.umap
```

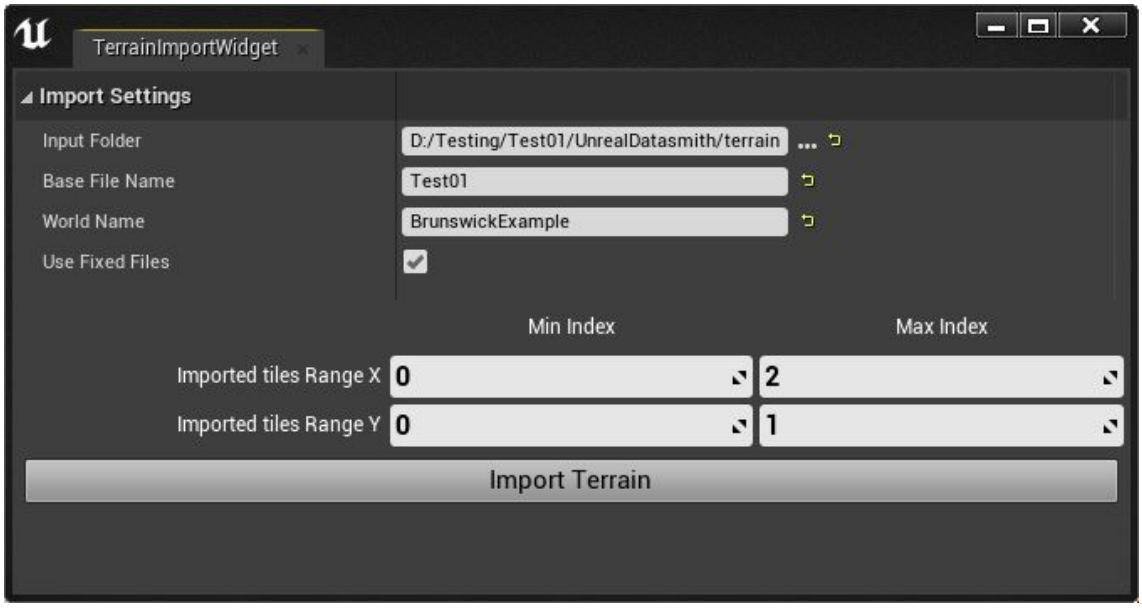


Figure 6.6.: Custom Import Terrain Widget

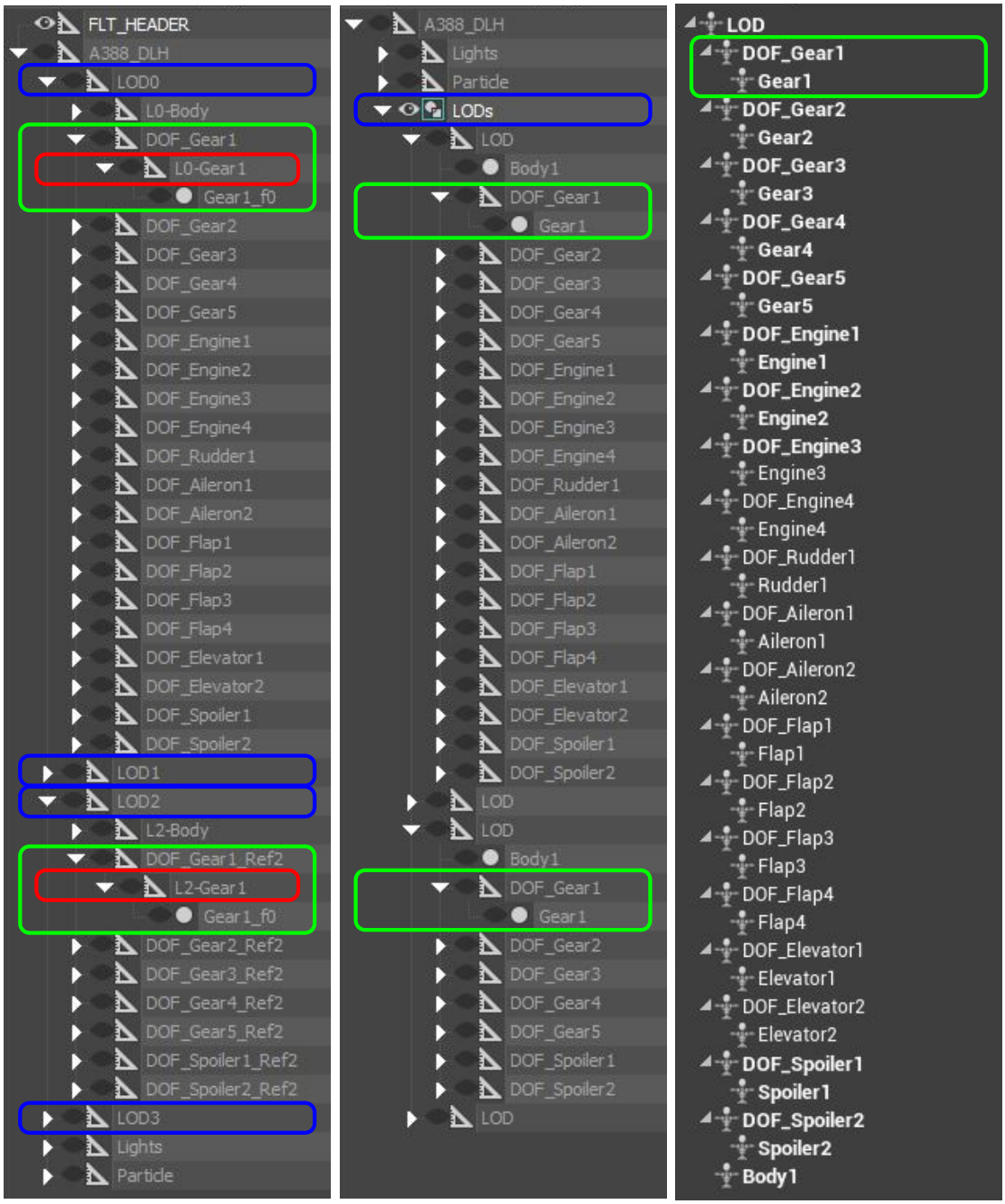
The final step is to assign the levels to a streaming layer and create additional LODs at will, with the World Composition utilities. The basic lighting settings, which are predefined in the default world, might also need specialization on a per-world basis.

6.2. Aircraft Models

As skeletal meshes are required for animated models, the FBX pipeline must be used together with the 3ds Max software. A relatively up to date version of 3ds Max is necessary, as the FBX import pipeline of UE4 is based on the 2018 version of FBX, but this software can both import the FLT models and export them in FBX with native functionalities. However, the FLT file format does not contain the concept of skeletal meshes [24] and therefore are the current FLT models constructed without a skeleton. Importing each aircraft model into 3ds Max, creating a skeleton for it, and exporting it as an FBX file is a very time-consuming work to do manually.
However, the UE4Editor can import models without a skeleton as a skeletal mesh by automatically creating the bone architecture. It traverses the scene graph and creates a new bone for each detected node, and if the model is exported with LODs, the UE4Editor builds its skeleton with the highest detail LOD and tries to match the other LODs to this. A successful result is, therefore, heavily dependent on exactly matching node structures in each LOD.

6.2.1. Node Tree Uniformization

This structure is not given with the currently used FLT models. Figure 6.7 shows the problem with the node trees of an Airbus A380-800 model in different stages of the import process. The green markings at the top of the images highlight the nodes regarding gear one of the aircraft in the highest-detail LOD, while the green ones at the bottom show the nodes for the same part in a lower-detail LOD. The structure of the nodes is correct, while the naming does not match. A python script using the 3ds Max python interface `pymxs` was developed to correct the mismatch. The



(a) Original FLT **(b)** Adapted **(c)** Imported in UE4

Figure 6.7.: Node Tree Comparison of the different Import Stages

script uses the regular expression library `re` of python to group the nodes with the following expression:

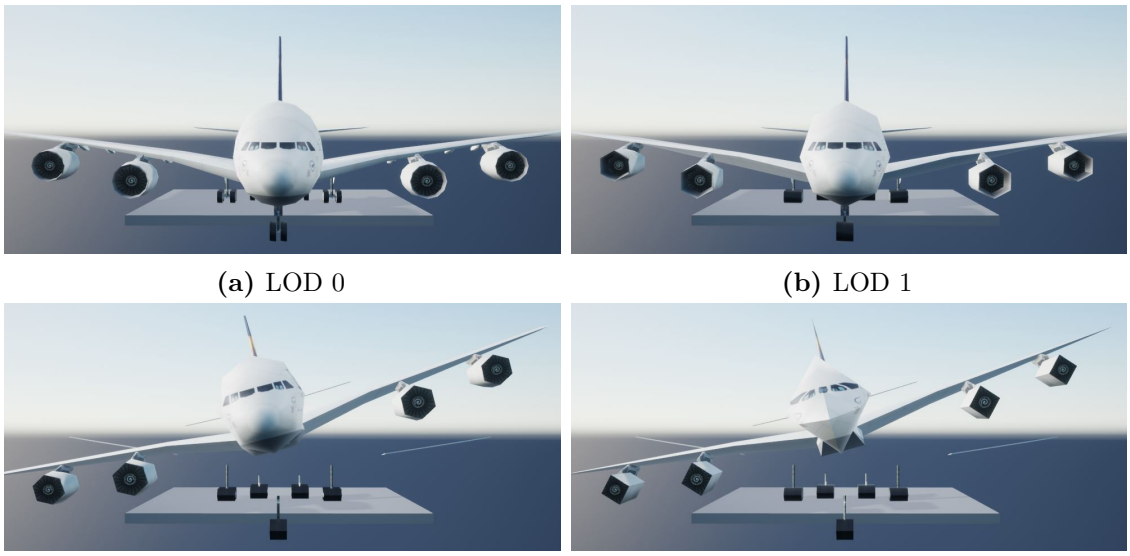
Listing 6.1: Node Detection Regular Expressions

```
1  import re
2
3  LOD = re.compile(r"^LOD[0-3]?$")
4  DOF = re.compile(r"^DOF_")
5  REMOVABLE = re.compile(r"^L[0-3]-")
6  BODY = re.compile(r"[bB]ody_?[1-4]")
7  UNDEFINED = re.compile(r".*")
```

The script's first step is to remove the nodes of the `REMOVEABLE` group from the node tree and assign the children of the removed nodes to their parents to preserve those. Because of this, the DOF node of the gear has in the adapted node tree only one child and not two like in the original The two red marked nodes were deleted.
In the slimmed-down node tree, the `LOD` nodes are searched. In the example, the four blue marked nodes in figure 6.7 are detected. The blue highlighted group node in figure 6.7b is created, and the respective children assigned to group them.
For unifying the node names, two suffixes are removed, `"_Ref[1-3]$"` and `"_f0(_-[1-4])?$"`, to reduce the differences, but not completely eliminate them. Therefore the highest-level `LOD` node is selected as the reference and, each other LOD's node tree is traversed an additional time, looking for names in the lower level LODs that are not present in the highest-level one. If a mismatch is detected, the parent node must have been equal, because otherwise, the parent must have mismatched first. Because of that, the script creates a name list based on the reference node tree. All names of the matching parents' children fill up the list and are the possible candidates. The closest matching name is selected from the list and assigned as the new name for the mismatching node. When the traversal has finished, it is safe to say that each node in LOD has the same name, parent, and node path in the reference LOD, but not that each parent has the same amount of children.
After unifying the names, the LODs are created with the "Level of Detail" utility of 3ds Max. As the utility is not directly accessible with the `pymxs` package, the graphical user interface (GUI) scripting capabilities of `pymxs` need to be used to

simulate the click on the "Create New Set" button of the LOD utility. Creating new LODs is necessary, although they already have one in the FLT model because that information is not automatically interchanged.
The last steps of the script are to correct two more straightforward issues. The FLT models are designed so, that each part of the aircraft has a different instance of the same material assigned, which causes the UE4Editor's importer to create multiple copies of the same texture file and wasting disk, as well as RAM space. Replacing duplicate instances by a single one avoids this behavior. The other issue is the incompatibility of UE4 with DirectDraw Surface (DDS) texture files. The FLT models in the AVES environment use this file format, but UE4 can not use it properly. The script changes the file endings of the texture files referenced in the models from `".dds"` to `".tga"`, but creating the Targa Image File (TGA) textures needs to be done with an external program.

One problem which the script can not solve is a bug in the UE4Editor FBX importer. For LODs that do not have the full node tree from the reference LOD, the importer might fail to connect the geometry to the correct node automatically. Figure 6.7c shows the imported node tree of LOD 2, the one with green markings at the bottom of figure 6.7b. This model does not contain, for example, the `DOF_Engine3` node, as the rotation of the aircraft engine shall not be animated after a certain distance. Nevertheless, the bone `DOF_Engine3` is present in the UE4 representation, indicated by the bold name. During the import process, the connection between the single bones is correctly detected and the regarding bones created. Thus, the five bones at the bottom of the tree are created and written in bold. However, the parts of the geometry are assigned from top to bottom without gaps so that until the `Gear3` node, everything aligns correctly, but after that, the geometry of spoiler one is assigned to the `DOF_Engine1` node and so on until the airplane body is matched wit `DOF_Engine3`. As soon as the aircraft switches into LOD 2 and engine three is spinning, the body starts to spin, as shown in figure 6.8. This is not the desired effect. Therefore LODs with reduced animation capabilities are currently not possible with skeletal meshes and need to be avoided.



(a) LOD 0 **(b)** LOD 1

(c) LOD 2 **(d)** LOD 3

Figure 6.8.: Different LODs with rotated `DOF_Engine3`

6.2.2. Final Import Process

The complete import process for an FLT model is a six-step process, as shown in figure 6.9. It starts with converting the DDS files to TGA ones and continues with importing the FLT file into 3ds Max by utilizing the native importer. Afterward, the mismatching node trees are uniformized with the python script, and the texture paths, as well as duplicate materials, are corrected. After importing the file as FBX and importing it with the skeletal mesh pipeline, a skeletal mesh asset is available in the UE4Editor. The lower-detail LODs of the assets might still contain mismatched bones, which will be problematic as soon as the model shall be animated.

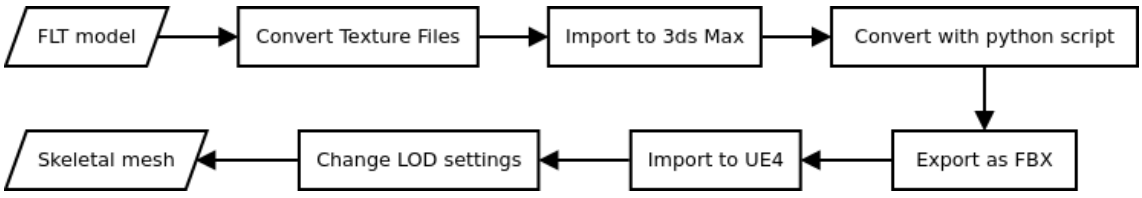


Figure 6.9.: New Aircraft Import Process

The details of the process are explained following an Airbus A380-800 example model, with the following relevant paths:

```
D:/3DAssets/A388_DLH.flt
```

```
D:/3DAssets/A388_DLH_Big.dds
```

```
D:/3DAssets/A388_DLH_Big.tga
```

The first path is the FLT model file, and the other two are the texture files before and after the conversion from DDS to TGA. The XnView was used for the conversion, but the used program is arbitrary.
Afterward, the FLT file is imported into 3ds Max, and the node tree looks like figure 6.7a. Then the python script can be run to create the node tree shown in figure 6.7b. The script should be run via the "MAXScript" utility in the "Utilities" panel of the 3ds Max GUI to ensure it runs properly, as the required GUI scripting needs to access the opened "Utilities" panel.
The fourth step is to export the FBX file with embed media activated to ensure that the textures are exported and export in the FBX 2018 version, as this is the officially supported version of UE4.
The so created FBX file can then be imported with the skeletal mesh pipeline of UE4. At this moment, the important settings are the "Import Mesh LODs" set to true, "Normal Import Method" to "Import Normal" as the normals exported from 3ds Max provide a better visual result, "Convert Scene Unit" true to automatically scale the model and a rotation of around the z-axis of 90°. The latter is necessary, as the FLT model is imported with their nose heading to the left. If the model shall be configurable via the XML file, it must be stored with the following path:

```
/Game/Content/Aircraft/<ModelName>/<ModelName>.uasset
```

The imported model consists at this stage of every LOD, even the broken ones, but those can be removed entirely within the UE4Editor Skeletal Mesh Editor, or replaced with automatically generated ones, that will build a reduced model. However, the auto-generated ones only provide a reduced visual quality compared to the hand-crafted FLT LODs.

7. UnrealViewer Control

7.1. General

Like the engine itself, the UnrealViewer is separated into modules, the VSAPConnection, the CoordConversion, and the main module, the UnrealViewer. Figure 7.1 gives an overview of the different classes in the modules, while the detailed class diagrams can be found in appendix A. The application's design revolves around a custom startup behavior, encapsulated in the UnrealViewer model and described in-depth in section 7.4. A newly implemented game instance class is utilized to distribute the specific workloads to other classes and modules during the startup. The VSAPConnection encapsulates creating the UDP socket, receiving and parsing VSAP messages, and sending the replies, while the CoordConversion module converts locations and rotations from the geocentric to the UE4 space.

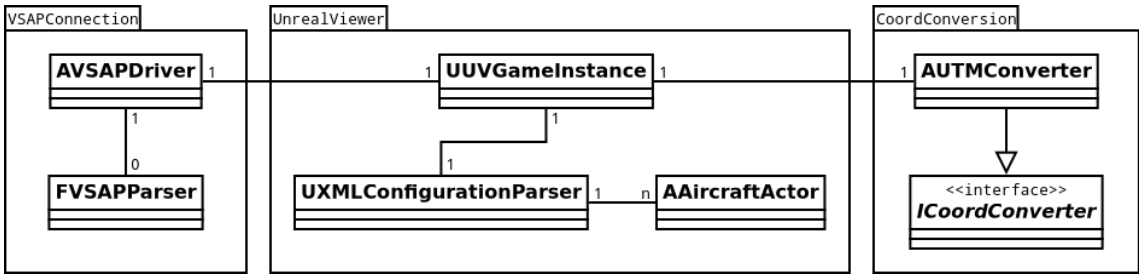


Figure 7.1.: Unreal Viewer Classes Overview

This application aims to create an entirely via XML configurable system, which contradicts the usual UE4 approach of creating the basic functionalities in C++ and reflecting them into the editor, where a game designer combines them into the final experience. That is because the XML file does the job of setting up the

scenery. Therefore the Blueprint system is used relatively rarely for the gameplay implementation of the UnrealViewer.

7.2. VSAPConnection Module

The VSAPConnection module provides the `AVSAPDriver` and `FVSAPParser` classes for communicating with VSAP messages (see figure 7.1). `FVSAPParser` is a module-private class used only by the driver to separate the message's continuous data into the single VSAP objects. The parser determines the type and the length of the incoming VSAP object by parsing the header object and determining the type and the variable length. The parser can process any type of VSAP message, so that the driver can receive any VSAP message, without necessarily processing it further, and continue with the next one.
The driver himself handles setting up the UDP socket, receiving the data, and forwarding the detected messages to the respective objects. In the `setup` method, the `FUdpSocketBuilder` factory provided by UE4 is used to create the socket with the respective IP address and port. With this socket, a `FUdpSocketSender` is created, which handles sending arbitrary data asynchronously, while the `FUdpSocketReceiver` class is not used. The receiver works similarly to the sender also asynchronously, but it does not use an interrupt controlled approach and polls the socket in a given time interval. An unfortunate combination of the frame rate and the receiving cycle can lead to problems, where an additional input lag is introduced, or the first part of the frame is processed with older information than the other one. To mitigate this, the `AVSAPDriver` inherits from `AActor` to enable ticking for the driver, so that the engine calls its `Tick` function once per frame, which then handles the data receiving. This makes sure that right at the beginning of a frame, the most up to date messages are received, and after that, the whole frame is processed with one consistent set of information.
For distributing the messages to the respective objects, the UE4 delegate system is used. Arbitrary functions can be bound to a delegate, as long as they match the correct function signature. As soon as the delegate is executed, the bound functions

are called to carry out their logic. For each message type, an individual delegate is available, and if an object is interested in receiving, for example, `MODEL_GEO` messages, it binds its receive method to the delegate, which is then executed with the respective data when the next `MODEL_GEO` is received. As some of the VSAP messages use indexes to address one specific object and others do not, two different delegate types are provided by the `VSAPDriver`. A multicast delegate is used for none indexed communication, where more than one function can be bound to it. On the other hand, for indexed communication, a single cast delegate is stored for each bound index in a `TMap` container.

7.3. CoordConversion Module

The CoordConversion Module takes care of converting input coordinates in the geocentric space, with latitude, longitude, and height, into the internal UE4 space, with x, y, and z. Depending on the projection used to create the loaded terrain model, different intermediate conversion steps are necessary. Therefore the `ICoordConverter` class is provided to create a standard interface for using any converter type.
During this concept study, only one implementation of the Interface was created because the only relevant coordinate system is UTM, as the new terrain import process does not support the geocentric projection, and other projections are not used in the AVES environment. Therefore the `AUTMConverter` converts the latitude, longitude, and height position from the geocentric space to UTM space and afterward to UE4 space. The first conversion is done using the same underlying math implementation as in the current AVESViewer to convert the latitude and longitude into UTM northing and easting values and vice versa. This implementation is encapsulated in the `GEO2UTMCONV` namespace and consists of 2 functions `geo2utm` and `utm2geo`.
The second step is described in figure 7.2. The black outer circle shows the northing and easting position and the aircraft heading angle, while the red inner circle shows the corresponding x and y position and the yaw angle. The different signs for the northing- and the y-axis result are caused by right-handed coordinate system in

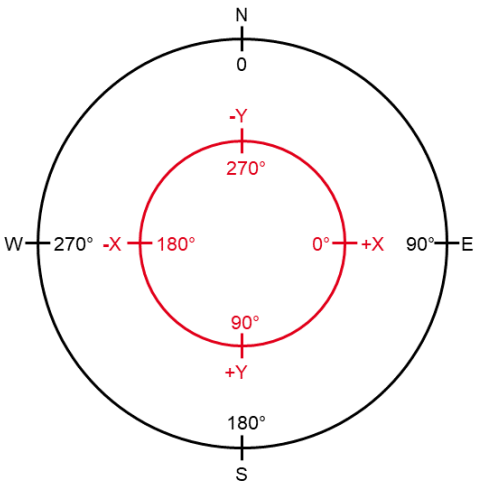


Figure 7.2.: Conversion from UTM (black) to UE4 (red) space

UTM and a left-handed in UE4. The difference of 90° between the heading and the yaw angle is due to an arbitrary choice of UE4 that east is at 0 degree. The elevation directly corresponds to the z-axis, and at last, the position is scaled by 100 because the UE4 space is in centimeter and the UTM space in meter.

To work correctly, the `AUTMConverter` must be configured with his `setup` method. It sets the UTM zone where the converter is currently operating in and the terrain origin. The position is used in the system against the precision loss of the converter. As the Trian3DBuilder exports UTM terrain models with the terrain origin point in the center of the model and not at 0°N and 0°E, this results in a shifting vector, which has to be applied during the conversion. The terrain origin provided with the `setup` method is therefore transformed into UTM space and then added as an offset to every other conversion, see listing 7.1 for the implementation source code.
Nevertheless, as UE4 uses only single-precision floating-point calculations, this system is not enough, because an aircraft can quickly leave the 20 km by 20 km area, where UE4 considers positions as sufficiently precise. The World Origin Shifting system is therefore also required to keep the precision up. That is why `AUTMConverter` inherits from `AActor`, the wrapper itself can be spawned into a world, and the `ApplyWorldOffset` method is called as soon as the world's origin is shifted. The

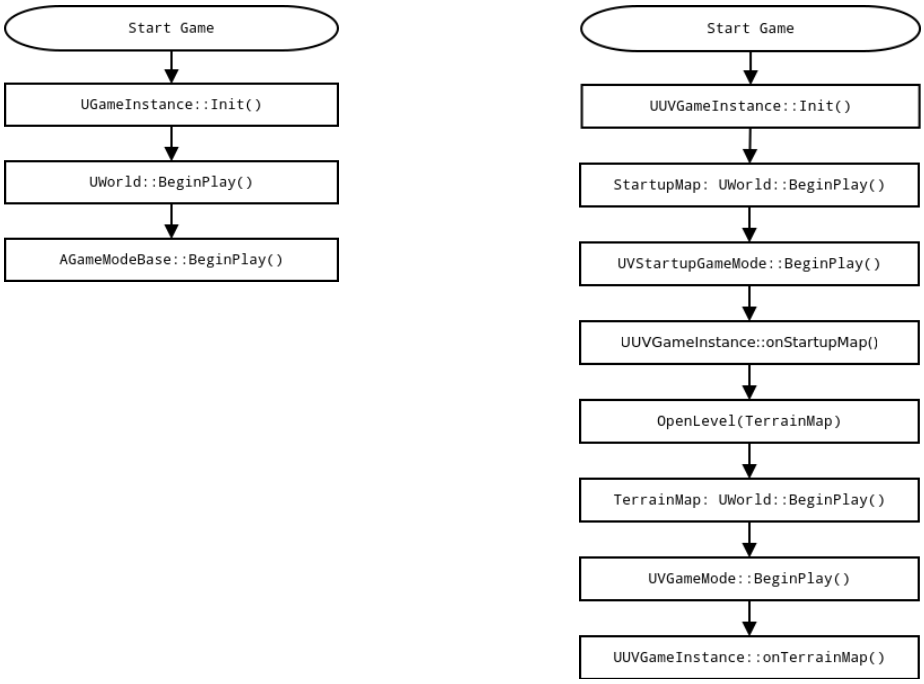
wrapper accumulates this shifting with double precision into another offset vector for the position. The process for converting a location from geocentric to UE4 is entirely in double precision, and only the output is in single precision.

Listing 7.1: Geocentric to UE4 Conversion on UTM Map

```
1  // Convert lat/long to northing/easting
2  const double dLatRad = dLat * GEO2UTMCONV::DEG2RAD;
3  const double dLongRad = dLong * GEO2UTMCONV::DEG2RAD;
4  double dNorthing, dEasting;
5  GEO2UTMCONV::geo2utm(m_iUTMZone, dLatRad, dLongRad, &dNorthing, &dEasting);
6
7  // Convert meters to UE4 centimeters and shift with terrain center
8  const double shiftedEasting = dEasting * 100.0 - m_dShiftEasting;
9  const double shiftedNorthing = dNorthing * 100.0 - m_dShiftNorthing;
10 const double shiftedHeight = dHeight * 100.0;
11
12 // Convert shifted northing/easting/height to x/y/z and shift by the world
       origin shifting offset
13 xyz.X = (float)(shiftedEasting + m_dWorldOriginShiftX);
14 xyz.Y = (float)(-shiftedNorthing + m_dWorldOriginShiftY);
15 xyz.Z = (float)(shiftedHeight + m_dWorldOriginShiftZ);
```

7.4. UnrealViewer Module

The default startup behavior of UE4 is changed to start the application as specified in the XML configuration file, as shown in figure 7.3. Usually, when the game is started, the game instance is initialized, a predefined map is loaded with an associated game mode before the game starts. The game instance is an object which exists exactly once for the whole duration of the game and can be accessed from nearly every other UObject, while the game mode declares a set of rules applying to the game, which defines what is currently allowed and the current aim for the players.
The adapted startup behavior is constructed around the new `UUVGameInstance` object, which creates and setups the `AVSAPDriver` and the `AGEO2UTMWrapper` by utilizing the `UXMLConfigurationParser`. The startup is a two-step process and splits into the `onStartupMap` and the `onTerrainMap` methods of the game instance. The `onStartupMap` method is called by the `UVStartupGameMode` Blueprint, as soon



(a) Native Unreal Engine 4 [12] **(b)** UnrealViewer

Figure 7.3.: Game Flow during Startup

as its `BeginPlay` is executed (see figure 7.4). The Blueprint is assigned to the startup map to be executed as soon as the map is loaded. This startup map is an empty map with the single purpose to call `onStartupMap`. The startup map is set to be the default map of the UnrealViewer, in the project settings, and behaves like a completely black main menu screen, which is controlled via XML files. When `onStartupMap` is called, it creates the XML parser, loads the configuration file specified in the command line argument with the `UXMLConfigurationParsers` method `loadConfig`, and parses them for the first part of information with `onStartupMap`, see table 7.1. The parser looks at this moment for additional files specified to parse, the IP address and port for the UDP socket, the terrain map's name, which should be loaded, and the terrain origin position. After that, the engine is instructed by the game instance to open the specified terrain map, and the remaining information is stored so that it can be used in the `onTerrainMap` method.

This method is called by the `UVGameMode` Blueprint, which is the default game mode and automatically assigned to every map unless otherwise stated. Therefore

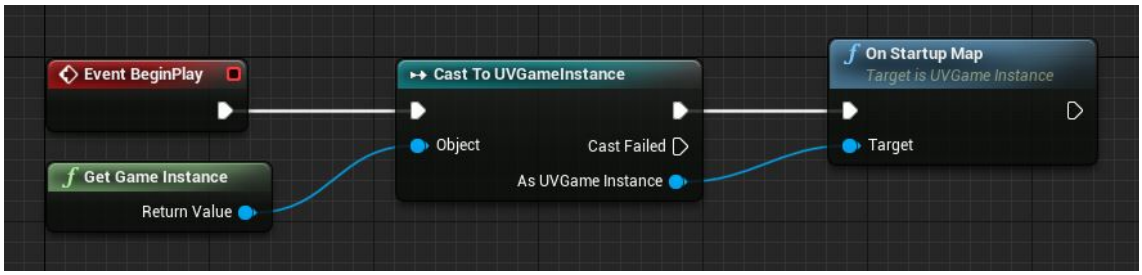


Figure 7.4.: `UVStartupGameMode`

XML symbol	Function
`import`	Use the specified file as input for the parser
`terrain`	
`world`	World name
`zone`	UTM zone of the terrain
`origin`	Terrain origin
`connect`	Specifying the VSAP connection
`ip`	Receiving IP address
`port`	Receiving port

Table 7.1.: OnStartupMap: Parsed XML Information

on loading the terrain map, the `UVGameMode` starts and behaves very similarly to the `UVStartupGameMode`, except that it calls the `onTerrainMap` method of the game instance. It utilizes the previously achieved information to spawn and setup the `AVSAPDriver` and `AGEO2UTMWrapper` actors, as both are required to run in the correct world context, one needs to tick each frame, and the other uses the World Origin Shifting system.

After that, the remaining parts of the XML configuration are parsed, with the `onTerrainMap` function of the parser (see table 7.2). Each aircraft definition found will result in spawning one actor object of one of the two classes, an `AStaticMeshActor` or an `AAircraftActor`. The deciding aspect is if the aircraft shall be controllable via VSAP or is a static decoration for the terrain, and the deciding XML configuration is if the index argument of the aircraft node is present and not set to -1. After spawning

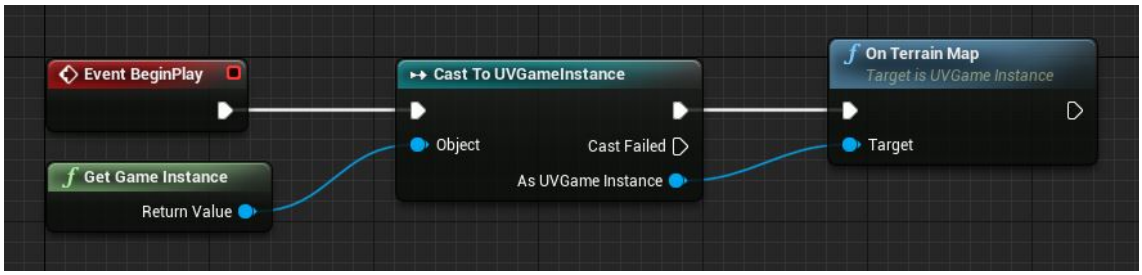


Figure 7.5.: `UVGameMode`

XML symbol	Function
`scenery`	Containing an arbitrary amount of aircraft child nodes
`aircraft`	Specifying an aircraft to be placed in the world
`name`	Name of the spawned actor
`index`	VSAP index
`main`	If the main camera shall be attached to this aircraft
`model`	Model name of the asset
`rot`	Initial rotation
`pos`	Initial position in geocentric coordinates

Table 7.2.: OnTerrainMap: Parsed XML Information

the actor, a reference for each is added to a `TArray` in the parser, to prevent them from being garbage collected.

As mentioned, the `AAircraftActor` class's task is to receive VSAP messages and control with them the 3D model. An object of the class consists of three components, the default root component, which sets the location and rotation in the world, a camera component with an offset transformation to the root component, and a `UPoseableMeshComponent`, which controls the Skeleton of the attached Skeletal Mesh by accessing the single bones by name.

As soon as the actor's VSAP index is set, it registers its `receiveModelGeo` method at the indexed `VSAPDriver` delegate. Another important setup method is `setMain`. If this is set to true, the aircraft's model is set invisible, while still casting shadows. That is important for the main aircraft to avoid partially visible parts of the own aircraft because the main camera is placed inside the object at the cockpit position.

8. Conclusion

8.1. Requirements

This work determined multiple requirements for three categories: performance, visuals, and multichannel. After analyzing how these are currently solved in the AVES environment, new approaches were developed for UE4. The majority of the requirements were already satisfied by production-proven build-in solutions or third-party plugins, as shown in table 8.1, but two significant features had to be analyzed in this work: importing existing data and controllability.

Requirement	Solution with UE4
Performance	
High frame rate	Modern high-performance system is build in
Distributed	VSAP protocol
Low input lag	Receive at frame start in UnrealViewer
Visual	
Terrain model	New import process for models created by the Trian3DBuilder
Static assets	Can be placed with the Trian3DBuilder
Dynamic assets	New import process for FLT models and configuration via XML
Shading model	The build-in model is one of the best available
Lighting effects	Large number of build-in effects
Weather effects	Promising third-party plugin TrueSky
Water effects	Promising third-party plugin Realistic Ocean Simulator
Multichannel	Completely handled by nDisplay system

Table 8.1.: Requirements Overview and UE4 Solutions

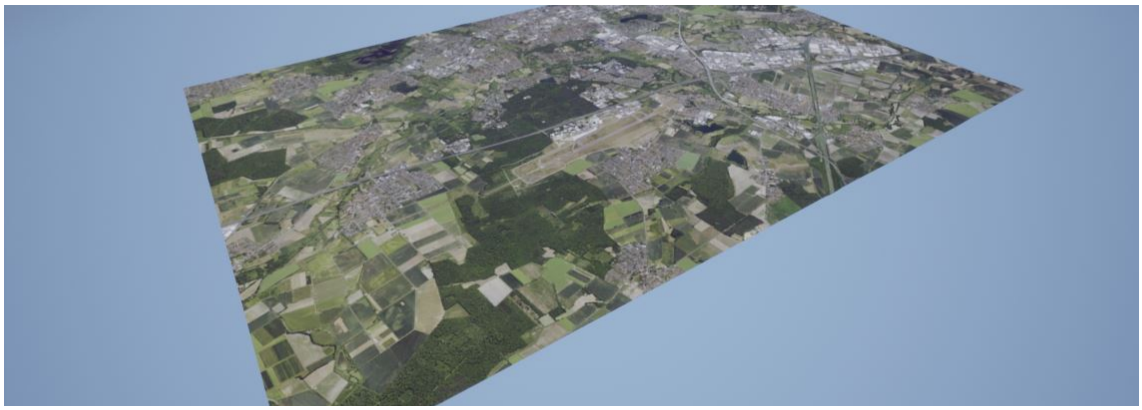


Figure 8.1.: Terrain Model Imported with new Process

8.2. Terrain Import Process

A new terrain import process was developed, which can be considered as mainly successful, regarding the criteria presented in the concept study design. It consists of two newly implemented python scripts and one new import widget for the UE4Editor. The existing elevation and image data used for the AVESViewer can be reused by utilizing the same tool to create the terrain databases, the Trian3DBuilder. The process contains four simple steps, as shown in section 6.1.4, and the developed tools hide the complexities of adapting the exported Datasmith files and importing them in the correct folder structure to be used with the UnrealViewer. The process supports both exporter model types offered by the Trian3DBuilder: static meshes and landscapes. Both model types are compared in table 8.2, and the landscape model performs in every aspect as good or better than the static mesh export because it is UE4's dedicated system for large terrain models, with the only exception that a small error introduced, while the tilled export is corrected. A successfully imported landscape terrain model is shown in figure 8.1, which was rendered on the test systems with more than 300 FPS on an Intel Core i7-3820 from 2012, and an NVIDIA GeForce GTX 1080, at a resolution of 1920 by 1200.

Nevertheless, even the landscape model is not the best possible model to create, due to the lack of a working multigrid exporter from the Trian3DBuilder. This forces the

model to be created with one tiling and detail setting for the whole virtual world, which creates an unwanted compromise between the desired high-details in areas where the aircraft will be close, like airports and the remaining areas, and the areas where a lower detail model would reduce file sizes and CPU overhead.

Criteria	Static	Landscape
Multigrid export	Trian3DBuilder crashes	Trian3DBuilder crashes
Procedural foliage	Unsupported	Supported
LOD morphing	Unsupported	Supported
Tiled export	Correct	Can be corrected
UTM model	Possible	Possible
Geocentric model	With precision loss	Impossible
World composition	Possible	Possible

Table 8.2.: Trian3DBuilder Datasmith Model Types Comparison

8.3. Aircraft Import Process

Regarding the criteria presented in the concept study design, the newly developed aircraft import process can be considered as partially successful. It consists of six simple steps utilizing an image conversion tool, 3ds Max, and a python script, which hide the tedious work of unifying the LOD's node trees, as shown in section 6.2.2. By using this process the existing FLT models, used for the AVESViewer, can be reused, as shown with the successfully imported aircraft in figure 8.2. However, the result is only a compromise: either all four handcrafted LODs can be used for the best visual experience and better performance, or the models can be animated. It is not possible to combine both features due to a flaw in the UE4Editor's FBX import pipeline for skeletal meshes, that mismatches sparsely filled LOD bones to the geometry

Nevertheless, the import process is based on 3ds Max, which supports FLT files, and additionally, multiple other file formats used in the AVES environment. Due to this flexibility, the process can be adapted to different types of file formats or structured

Figure 8.2.: Airbus A380-800 Aircraft Model Imported with new Process

aircraft, and as soon as the flaw in the FBX importer of UE4 is eradicated, all file formats can be imported without a compromise.

8.4. Controllability

With the UnrealViewer, an application was developed to successfully showcase the concept of a fully configurable and VSAP controllable rendering application in the AVES environment.
The custom startup behavior constructs a highly flexible environment, altering both main aspects of the scenery, the terrain model, and the aircraft models with an XML configuration file. The possible configurations consist of importing extra configuration files, setting the VSAP connection IP Address and port, selecting one of the maps containing an imported terrain and configuring the UTM projection, as well as spawning and positioning multiple aircraft.
The UnrealViewer's VSAPConnection can communicate via the VSAP while maintaining a low input lag and simultaneously ensuring the whole frame is rendered with the same data. This is achieved by a custom receiving solution, which leverages the ticking function and receives at the beginning of each frame. Although only the aircraft actor receives messages to position and rotate itself, the module can process every possible VSAP message without breaking the application. This was tested by using the existing VSAP output of the simulation environment.

8.5. Future Work

To utilize the full potential of the 3D assets, both import processes need further adaption, the terrain model does not support multigrids, and the FLT models have partially broken LODs. The former can only be done by TrianGraphics themselves, while the latter can be solved in multiple ways. Either the flaw in the UE4 skeletal mesh pipeline is removed, or the meshes must be assigned manually with a skeleton. Nevertheless, both problems only limit the potential of the models and do not contradict using them.

On the other hand, the UE4 can decently provide productive usage in the AVES environment, a proper setup, containing general settings, effects, and the nDisplay system, must be created.
Most of this process is mainly integration or configuration work because, for nearly all of the problems, solutions exist, but the vast number of available options complicates it, especially in the AVES environment. The render application is not one product that will be finished at a given point in time. The application is continuously extended to visualize the new demands of the research and development projects.
For streamlining one part, the setup of the nDisplay system, UE4 supports the VESA MPCDI standard, and the Scalable Display Technologies company provides a third-party solution.[21][34][47].

The last aspect mentioned is the announcement of the new Unreal Engine 5 for late 2021, which claims to revolutionize the rendering by removing polygon budgets and introducing a fully dynamic illumination solution, which significantly simplifies the game design [1]. Forward compatibility for current UE4 was also announced, but before the time is invested in setting up a complex UE4 project and migrate it later on to version 5, it might be worth waiting until more information is available.

Bibliography

[1] *A first look at Unreal Engine 5.* 2020. URL: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5 (visited on 06/30/2020).

[2] *About the Datasmith Import Process: Contains details about specific issues in the way Datasmith imports scenes into Unreal, and next steps you can follow to work with the imported Assets in Unreal.* URL: https://docs.unrealengine.com/en-US/Engine/Content/Importing/Datasmith/Overview/ImportProcess/index.html (visited on 08/10/2020).

[3] David Allerton. *Principles of Flight Simulation.* 1st ed. Aerospace Series. s.l.: Wiley-Blackwell, 2009. DOI: 10.1002/9780470685662.

[4] AlphaPixel. *LOD (Level of Detail) in OpenSceneGraph.* 2015. URL: https://alphapixel.com/wp-content/uploads/2015/04/LOD-Level-of-detail-in-OpenSceneGraph-OSG.pdf (visited on 07/21/2020).

[5] *Awesome games coming in 2020 and beyond.* 2019. URL: https://www.unrealengine.com/en-US/blog/awesome-games-coming-in-2020-and-beyond (visited on 08/03/2020).

[6] *Coding Standard: Standards and conventions used by Epic Games in the Unreal Engine 4 codebase.* URL: https://docs.unrealengine.com/en-US/Programming/Development/CodingStandard/index.html (visited on 08/27/2020).

[7] D. Francis Crane. "Flight Simulator Visual-Display Delay Compensation". In: *Proceedings of the 13th Conference on Winter Simulation - Volume 1.* WSC '81. IEEE Press, 1981, pp. 59–67.

[8] Sevan Dalkian. *nDisplay Technology: Limitless scaling of real-time content.* Ed. by Epic Games. 2019.

[9] *Datasmith-Supported Platforms: Details what Datasmith features work on which different platforms.* URL: https://docs.unrealengine.com/en-US/Engine/Content/Importing/Datasmith/Platforms/index.html (visited on 08/10/2020).

[10] Holger Duda et al. "Design of the DLR AVES Research Flight Simulator". In: *AIAA Modeling and Simulation Technologies (MST) Conference*. Reston, Virginia: American Institute of Aeronautics and Astronautics, 2013. DOI: 10.2514/6.2013-4737.

[11] *Frequently Asked Questions - Unreal Engine*. URL: https://www.unrealengine.com/en-US/faq (visited on 07/16/2020).

[12] *Game Flow Overview: The process of starting the engine and launching a game or play-in-editor session.* URL: https://docs.unrealengine.com/en-US/Gameplay/Framework/GameFlow/index.html (visited on 06/09/2020).

[13] James P. Gaska et al. "Pixel Size Requirements for Eye-Limited Flight Simulation". In: (2010).

[14] T. Gerlach. "Visualisation of the brownout phenomenon, integration and test on a helicopter flight simulator". In: *The Aeronautical Journal* 115.1163 (2011), pp. 57–63. DOI: 10.1017/S0001924000005364.

[15] *Import OpenFlight Models and Terrains into Unreal Engine*. 2020. URL: https://www.presagis.com/en/press-center/detail/import-openflight-models-and-terrains-into-unreal-engine/ (visited on 06/25/2020).

[16] *Import-Export*. URL: https://docs.blender.org/manual/en/latest/addons/import_export/index.html#addons-io (visited on 08/10/2020).

[17] Peigang Jiao and Lei Wang. "Research on Virtual Visualization Technology of Flight Simulation". In: *IOP Conference Series: Materials Science and Engineering* 452 (2018), p. 042206. DOI: 10.1088/1757-899X/452/4/042206.

[18] Brian Karis and Epic Games. "Real shading in unreal engine 4". In: *Proc. Physically Based Shading Theory Practice* 4 (2013), p. 3.

[19] *Landscape Outdoor Terrain: Landscape system for creating terrain for large, open, outdoor environments.* URL: https://docs.unrealengine.com/en-US/Engine/Landscape/index.html (visited on 08/29/2020).

[20] *Landscape Technical Guide: Technical Settings for Landscape.* URL: https://docs.unrealengine.com/en-US/Engine/Landscape/TechnicalGuide/index.html (visited on 08/08/2020).

[21] *nDisplay Configuration File Reference: A reference companion for all the settings available in the nDisplay configuration file.* URL: https://docs.unrealengine.com/en-US/Engine/Rendering/nDisplay/Configuration/index.html (visited on 09/11/2020).

[22] Michael Noland. *Unreal Property System (Reflection)*. 2014. URL: https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection (visited on 09/11/2020).

[23] *Plug-Ins & File Exchange*. URL: https://www.maxon.net/de/produkte/cinema-4d/workflow-integration/plug-ins-file-exchange/ (visited on 08/10/2020).

[24] Presagis. *OpenFlight Scene Description Database Specification: Version 16.7*. 2018. URL: https://portal.presagis.com/support/solutions/articles/19000096896-openflight-16-7.

[25] *Presagis and Epic Games Join Forces to Provide New Capabilities to the Simulation Industry*. 2018. URL: https://www.presagis.com/en/press-center/detail/presagis-and-epic-games-join-forces-to-provide-new-capabilities-to-the-simulation-industry/ (visited on 06/25/2020).

[26] *Procedural Foliage Tool Quick Start: How to set up and use the Procedural Foliage tool*. URL: https://docs.unrealengine.com/en-US/Engine/OpenWorldTools/ProceduralFoliage/QuickStart/index.html (visited on 09/03/2020).

[27] *Programs & Events - SIGGRAPH*. 2019. URL: https://s2019.siggraph.org/conference/programs-events/ (visited on 08/03/2020).

[28] *Programs & Events - SIGGRAPH*. 2020. URL: https://s2020.siggraph.org/conference/program-events/ (visited on 08/03/2020).

[29] *Projection Policies in nDisplay: Reference for policies supported in Unreal Engine for multiple screen displays*. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/nDisplay/ProjectionPolicies/index.html (visited on 09/11/2020).

[30] *R&D - Storm In a Teacup*. URL: http://www.stcware.com/rd/ (visited on 09/11/2020).

[31] Michael Ratzel. *Erweiterung des AVES Viewer zur performanten Visualisierung von Lichtquellen*. 2019.

[32] Michael Ratzel. *Expansion of the AVESViewer with realistic Light and Shadow effects*. 2020.

[33] *Rendering to Multiple Displays with nDisplay: nDisplay renders your Unreal Engine scene on multiple synchronized display devices*. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/nDisplay/index.html (visited on 08/04/2020).

[34] *Scalable Display Technologies |Automatic warp & blend for multi-projector displays*. URL: https://www.scalabledisplay.com/ (visited on 09/11/2020).

[35] Schachter. "Computer Image Generation for Flight Simulation". In: *IEEE Computer Graphics and Applications* 1.4 (1981), pp. 29–68. DOI: 10.1109/MCG.1981.1674014.

[36] Michael J. Sieverding, Clarence W. Stephens, and Amos E. Kent. "The Emerging DoD Requirement for More Realistic Weather in Flight Simulation". In: (2010).

[37] *Skeletal Meshes: Meshes bound to a hierarchical skeleton of bones which can be animated for the purpose of deforming the mesh.* URL: https://docs.unrealengine.com/en-US/Engine/Content/Types/SkeletalMeshes/index.html (visited on 07/16/2020).

[38] Antonín Šmíd. "Comparison of unity and unreal engine". PhD thesis. 2017.

[39] *Static Meshes: Static geometry which can be cached in video memory and rendered by the graphics card.* URL: https://docs.unrealengine.com/en-US/Engine/Content/Types/StaticMeshes/index.html (visited on 07/22/2020).

[40] *Texture Streaming: System for loading and unloading textures into and out of memory during play.* URL: https://docs.unrealengine.com/en-US/Engine/Content/Types/Textures/Streaming/Overview/index.html (visited on 08/04/2020).

[41] TIGA. *Game engines used by video game developers UK 2014 | Statista.* 2014. URL: https://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/ (visited on 07/16/2020).

[42] Amanda Towner. *Presagis Brings Physics-Based Simulation to Gaming.* 2019. URL: https://www.halldale.com/articles/16557-presagis-brings-physics-based-simulation-to-gaming (visited on 06/25/2020).

[43] TrianGraphics. *Trian3DBuilder Manual v7.1.* 2020.

[44] *TrianGraphics at 2019 I/ITSEC.* 2019. URL: https://exhibits.iitsec.org/2019/Public/eBooth.aspx?IndexInList=0&FromPage=Exhibitors.aspx&ParentBoothID=&ListByBooth=true&BoothID=175674 (visited on 06/25/2020).

[45] *UE4 Roadmap | Trello.* 2020. URL: https://trello.com/b/TTAVI7Ny/ue4-roadmap (visited on 08/04/2020).

[46] *Unreal Object Handling: Overview of the features of the UObject system.* URL: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Objects/Optimizations/index.html (visited on 09/08/2020).

[47] VESA - Interface Standards for The Display Industry. *Free Standards - VESA - Interface Standards for The Display Industry.* URL: https://vesa.org/vesa-standards/ (visited on 09/11/2020).

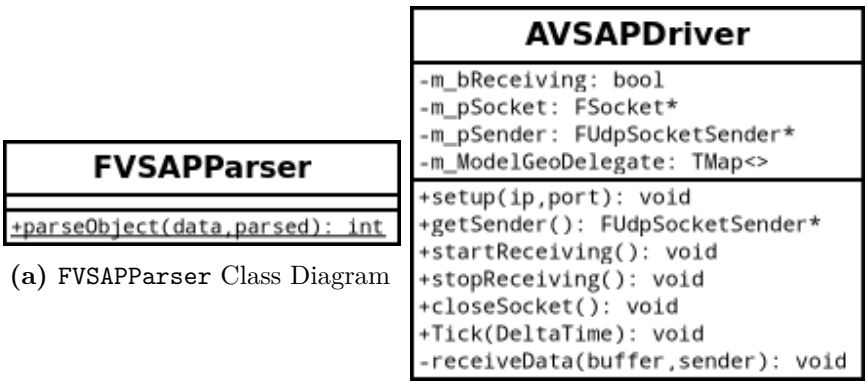
[48] Leonard Wanger. "The effect of shadow quality on the perception of spatial relationships in computer generated imagery". In: *Proceedings of the 1992 symposium on Interactive 3D graphics - SI3D '92*. Ed. by Marc Levoy, Edwin E. Catmull, and David Zeltzer. New York, New York, USA: ACM Press, 1992, pp. 39–42. DOI: 10.1145/147156.147161.

[49] *World Composition User Guide: System for managing large worlds including origin shifting technology.* URL: https://docs.unrealengine.com/en-US/Engine/LevelStreaming/WorldBrowser/index.html (visited on 08/08/2020).

Appendix

Appendix A.

UnrealViewer Class Diagrams



(a) `FVSAPParser` Class Diagram

(b) `AVSAPDriver` Class Diagram

Figure A.1.: VSAPConnection Class Diagrams

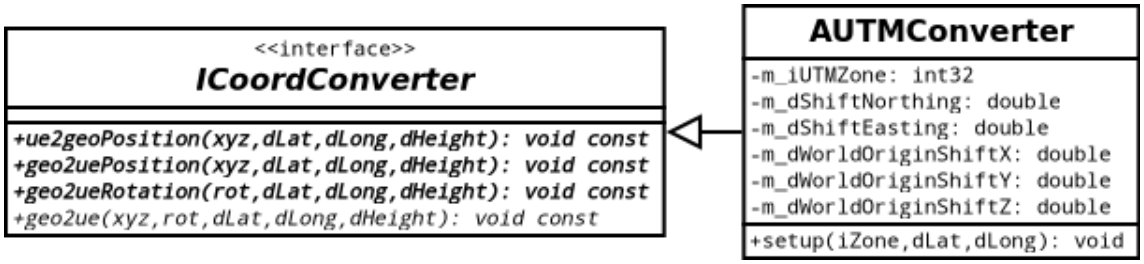
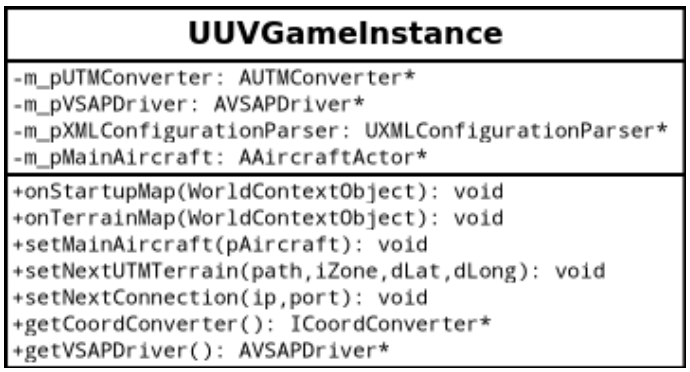
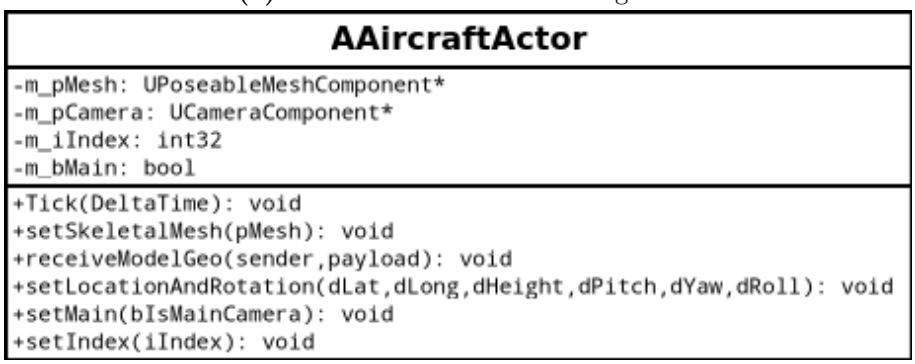


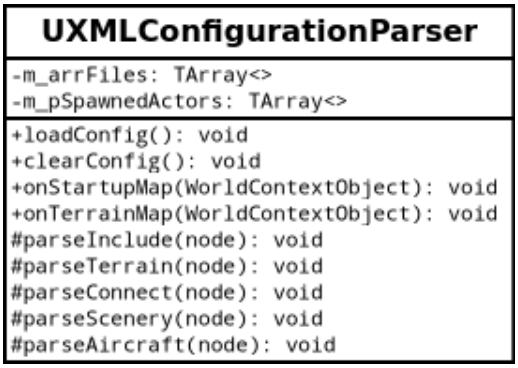
Figure A.2.: CoordConversion Class Diagrams



(a) `UUVGameInstance` Class Diagram



(b) `AAircraftActor` Class Diagram



(c) `UXMLConfigurationParser` Class Diagram

Figure A.3.: UnreaViewer Class Diagrams