DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Games Engineering

Procedurally Generated and Digitally Recreated Environments Designed for Interactive Content in Augmented and Virtual Reality

Jan-Philipp Fahlbusch

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Games Engineering

Procedurally Generated and Digitally Recreated Environments Designed for Interactive Content in Augmented and Virtual Reality

Prozedural generierte und digital nachgebildete Umgebungen konzipiert für interaktive Inhalte für den Einsatz in der erweiterten und virtuellen Realität

Author:	Jan-Philipp Fahlbusch
Supervisor:	Prof. Gudrun Klinker
Advisor:	Sven Liedtke
Submission Date:	15.05.2020

I confirm that this master's thesis in Informatics: Games Engineering is my own work and I have documented all sources and material used.

Munich, 15.05.2020 Jan-Philipp Fahlbusch

Acknowledgements

First of all, I would like to thank my supervisor Prof. Gudrun Klinker for overseeing my thesis and making this work possible.

I would like to express my gratitude to Sven, my advisor, for guiding me through this work. I thank you for helping me develop the scope and topic of this thesis and giving me valuable, helpful, and important feedback on the subject-matter of this work. I particularly thank you for pushing me to take that extra step in developing my thesis. Without your weekly input on my implementation, this work would not be in the presented quality.

I also thank Sabrina, Maxi, Thomas, Dietmar, Annette and Alex for taking the time to proofread this work and finding all those small mistakes I overlooked.

I deeply thank Sabrina for her encouragements and patience to help me complete this thesis. Your support helped me overcome even the most difficult obstacles.

Finally, I am deeply grateful for my parents, Annette and Dietmar. Only through your extensive support throughout my education and in all circumstances have you made this thesis possible.

Abstract

Creating understandable and easy to use interactive content is the goal of any software, wanting its user base to intuitively work within its environment. It is very important to test many different interaction concepts early on during development. This is especially true for interactive objects in new technologies, such as augmented and virtual reality, as they often require new approaches for a streamlined user experience. For the aforementioned rapid prototyping process, we want to create a solution that can be used in any current game engine, supporting the creation of augmented or virtual reality content. Furthermore, this thesis provides a proof of concept, showing that the automatic placement of content and environment objects is possible in real-time.

The future of any head-mounted display in both augmented and virtual reality lies in wireless setups, making the ability to analyse the surrounding of the user in real-time an imminent requirement of any application deployed on such a device. By only processing the point cloud of a room, our approach is able to procedurally generate and digitally recreate environments, based on the user's actual surroundings. Through the detection of distinctive room features, this implementation is able to provide areas, such as walls or tables, to place interactive content on automatically and independently of any user interactions, except the required prior scanning of their surroundings.

In this thesis, we will combine both the augmented and virtual reality settings into a single solution, despite the slightly different requirements, generating a simplified and render optimized environment over the existing room mesh, creating bounds with at least fifteen times less vertices and triangles compared to the original. The bounds are calculated in less than 50 milliseconds on recommended virtual reality hardware, allowing our solution to function as a real-time replacement of the original mesh.

Moreover, this work groups the created spatial mappings into categories for different types of interactive content, allowing for automatic placement in suitable locations. With a processing time of less than 200 milliseconds to analyse a standard surrounding, this enables applications to distribute content optimally around the user in a very short time frame. Creating such an environment with different areas to test interactive instances helps simplify the testing and comparison process of new interaction concepts in augmented and virtual reality, speeding up the development cycle of new and user friendly technologies. Additionally, this can be used in finished applications, procedurally placing new environments around the headset wearer based on their surroundings, with special areas for predefined interaction objects placed in easily reachable locations.

Contents

1. Introduction

This work will provide solutions for specific problems, which arise in the process of development for augmented reality (AR) and virtual reality (VR) devices. There are two main issues that are addressed in this thesis. Firstly, we are helping users interact with virtual worlds by suggesting interactive content areas based on the surroundings. Secondly, users are enabled to navigate complex environments in the real-world, either partially (AR) or fully (VR) occluded by a worn headset, by providing them with a simplified recreation of the room. The first chapter will give the motivation, why the stated topics need to be solved and presents recent solutions that were already developed in this area recently. We conclude with the aim of this thesis.

1.1. Motivation

In recent years, AR and VR headset devices have become increasingly more popular for the private user at home [IDC, 2020; Petrock, 2019]. Currently, most VR headsets, especially those with high graphical capabilities, are still attached by a cable to the personal computer (PC). Nevertheless, a few devices already use cableless headsets and even more are announced to be released in the near future [Cherdo, 2020; Greenwald, 2020]. In contrast, AR devices have already very prominent examples for cableless headsets, such as the Microsoft HoloLens and Magic Leap [Magic Leap, 2020; Microsoft, 2020a]. With both types of head-mounted displays (HMDs) getting rid of their restrictive cords, the movement freedom of the user increases drastically. Examples of the above-mentioned headsets can be seen in Figure 1.1.

With the obtained freedom, the user can now freely navigate through large and complex spaces. It becomes vital to create understandable and easy to use interactive content, so users can intuitively interact with their virtual surrounding in an overlay, which is the case for AR, or in a complete digital representation, such as in VR [Evans, 2018; Purwar, 2019]. Firstly, an automatic understanding of the users surrounding becomes important to relieve the user of the task to find a suitable spot for any interactive content object in their room. Devices such as the HoloLens already have built-in technologies to automatically create a spatial map of its surrounding [Microsoft, 2018c], while VR users have to revert back to third-party hardware solutions, such as the ZED Mini [Stereolabs, 2020a]. However, both solutions can only visually represent and reproduce the surrounding of the HMD user.

In the future it will become especially important for VR devices to represent the natural boundaries of the area the user is currently physically occupying. With cordless

Figure 1.1.: Some examples of AR and VR headsets available in 2019. The examples, which are shown, include the Microsoft HoloLens (bottom left), HTC Vive (top right), Magic Leap 1 (middle), Oculus Quest (bottom right), and the Google Cardboard (top centre). Image taken from [Fidel, 2019].

devices, the user can freely walk around any complex environment, without actually seeing their surroundings. One way to visually limit the newly gained freedom for users would be to automatically adapt the virtual environment to feature objects in the exact same position to their real-world counterparts. This, in return, requires a simplified boundary of the room, enabling faster collision and limit checks against the geometry, mainly because most represented features in the scanned spatial map are too insignificant to be digitally recreated and place objects around the virtual environment.

The detection and placement of suitable areas for interactive content are not only important to simplify the interaction with the environment for the user, but also for the developer. To create the perfect interactive experience, developers have to rapidly prototype possible solutions. However, always manipulating their application to show current samples and possible comparisons can be tiresome [Evans, 2018; Purwar, 2019]. It would be much easier if the interactive contents were to be placed automatically in the scene based on a few classification parameters.

For a plugin to incorporate the above mentioned features, one more aspect is of importance: the creation process. Many current applications for AR and VR are produced in game engines, such as the Unity and Unreal Engine [Kim, 2019]. For maximum compatibility of a plugin, this would require an uncomplicated solution that can be easily implemented in any environment on any device. To support many representations of the physical world around the user, the processing of the virtual environment should be based on a point cloud, which is the most widely used geometry and most other representation methods incorporate it in one way or another. It is thus an easy to achieve format through conversion steps.

1.2. Overview of Previous Work

After describing the problem, we first take a look at already existing solutions, which attempt to fully or partially solve the stated issues. As multiple and different topics are part of this master thesis, we will turn to related works in each respective field. We will analyse previous solutions for spatial understanding, space categorization and room reconstruction.

1.2.1. Spatial Understanding

In clarifying the term spatial understanding, the terminology for spatial mapping needs to be determined, as it provides the foundation and starting point of spatially understanding the environment.

Spatial mapping provides the ability to visually represent a real-world space in a three-dimensional (3D) map. An example of a spatially mapped environment can be seen in Figure 1.2, where a couch is represented in the virtual space through a texturized mesh, consisting of connected vertices with edges, forming triangular polygonal faces. This type of mesh representation is a polygon mesh, or more specifically in this example a triangle mesh. Especially for AR applications, these 3D representations of the environment are vital to support the placement of digital objects on real surfaces and allow them to correctly occlude or be occluded by the real-world [Microsoft, 2018c; Stereolabs, 2020c].

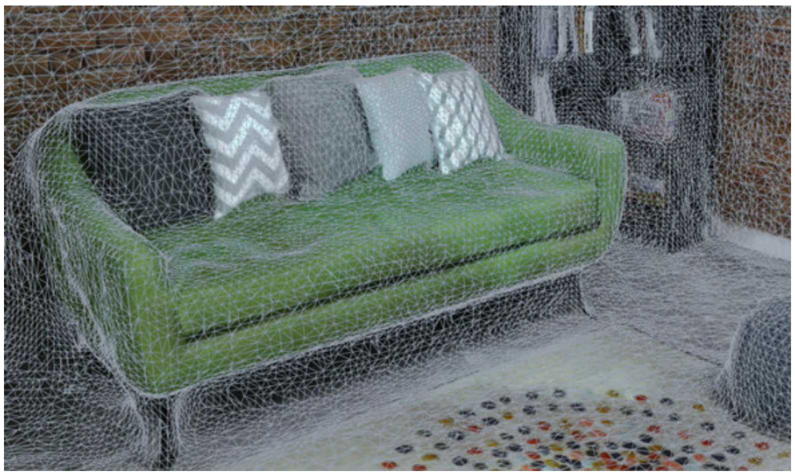


Figure 1.2.: An example of spatial mapping, where this couch is represented through a texturized mesh, consisting of many connected vertices through edges, forming triangular polygonal faces. This form of mesh is called a triangle mesh. The mesh was generated with the help of a ZED Mini. Image taken from [Stereolabs, 2020c].

The physical surroundings can now be correctly visualized in the virtual environment, but comprehending what is being represented in the current geometry, which is the definition of spatial understanding, is not given in the spatial map information. This might not be as apparent, because the couch, bookshelf, wall, and floor in Figure 1.2 are all clearly visible and identifiable through a person viewing the image. However, the computer does not have the same spatial abilities as a human, unless we teach it through algorithms to actually understand and classify the digital mesh. There have already been many works dealing with this issue, so we will only highlight those closest to our problem.

In the field of AR, there are already two working solutions to spatially understand the environment developed exclusively for the Microsoft HoloLens. The first one was developed by Asobo Studios[1] for the applications Young Conker and Fragments [Microsoft, 2018b]. This project was then later integrated into the Microsoft Mixed Reality Toolkit (MRTK), which is intended as a starting point to accelerate the development of applications for the Windows Mixed Reality (WMR) platform by providing a collection of scripts and components [Microsoft, 2019a]. MRTK supports HMD especially developed for the Windows architecture and includes the HoloLens, HoloLens 2, and the immersive WMR headsets. The MRTK is currently actively developed and extended for Unity to provide a cross-platform mixed reality (MR) application development environment [Microsoft, 2020c].

This integrated solution is optimized only for the first generation HoloLens and is wrapped into a Universal Windows Platform (UWP) dynamic-link library (DLL). This library was developed especially for the two applications from Asobo Studios and allows developers to quickly identify floors, ceilings, and walls, including the ability to determine the most desirable physical location for holographic objects. This tool is usable within Unity through the contained prefab within the HoloToolkit [Microsoft, 2018b]. This toolkit does fulfil various requirements we have for our set of problems, but has a few fundamental issues and flaws. For one, it is only optimized for the use with the HoloLens and Unity, but it can be used with other platforms and hardware but this would require fundamental changes to the provided source code. Additionally, it needs to reanalyse the spatial mapping for each quarry and does not provide an option to save the spatial scene understanding. It provides an additional small internal game engine to analyse the room, causing this DLL to be a very space-consuming solution for the room analysis. And lastly, it cannot provide a complete list of suitable spaces for walls, floors and on other objects with only one query.

With the HoloLens 2, Microsoft went ahead and implemented a new spatial understanding software development kit (SDK). As seen in Figure 1.3, this SDK can give a complete scene understanding analysis through the spatial mapping input mesh. They achieve this by combining the power of existing MR runtimes, such as spatial mapping, and a new artificial intelligence (AI) driven runtime. Through this combination, they can recreate the 3D environment digitally with correctly labelled tags and assign plane regions for object placement back to the user [Microsoft, 2019b]. However, the scene understanding application programming interface (API) can only run on the

[1]https://www.asobostudio.com/

new HoloLens 2, as it uses an AI coprocessor implemented in the custom HoloLens multiprocessor central processing unit (CPU) called the Holographic Processing Unit (HPU). This custom HPU allows them to natively and flexibly implement Deep Neural Networks (DNNs), which in turn can now run the spatial scene understanding with the help of DNNs [Microsoft, 2017; Microsoft, 2019c]. Despite providing exactly the solution we require for solving our stated problems, this SDK has one major flaw for our use case, which we already mentioned: it can only run on the HoloLens 2. This would limit our solution to be only deployable on the HMD from Microsoft and not have the hardware and software flexibility we actually want for our solution.

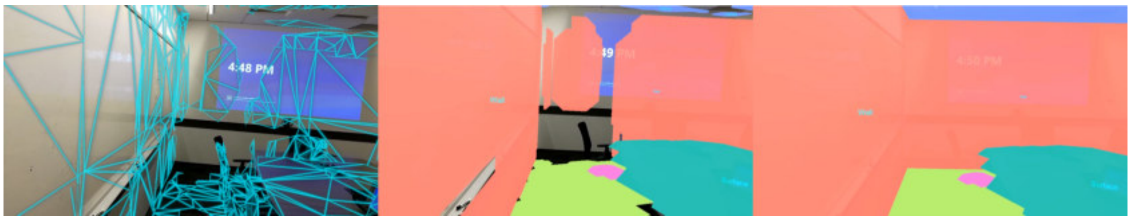


Figure 1.3.: Highlighting the spatial scene understanding on the HoloLens 2. On the left, the spatial mapping of the room can be seen, in the middle the categorization of the spatial mapping mesh, and on the right the completed spatial understanding of the room without any holes through enabled inference. Image taken from [Microsoft, 2019b].

As currently already used solutions do not meet all of our needs, we will look at related scene understanding works next. Since the analysation of the scene will take a point cloud as an input format, we will only look at relevant proposals to our issues in this area of room understanding.

A relevant research paper from Anagnostopoulos et al. successfully creates Building Information Models (BIMs) by analysing indoor point clouds. They detect floors, walls, and ceilings in point clouds under the assumption of Manhattan-World (MW) buildings, which are defined by horizontal floors and ceilings and walls that are always parallel to the X-Z or Y-Z plane (with Z being the vertical axis). Using these MW rules, they are able to propose an algorithm that can analyse the Point Cloud Data (PCD) and create a BIM. The ceilings are detected through a simple projection of the PCD onto two planes (X-Z and Y-Z) and analysing the resulting octree for the maximum point densities. The walls are detected by analysing the vertical points for an alignment to the X-Z or Y-Z planes [Anagnostopoulos, 2016]. There are many great ideas in this paper to detect floors, walls, and ceilings of interior rooms, but the limitations on the position of the walls towards the vertical axis planes are an important aspect to this work. We want to solve this constraint and enable to detect walls in rooms, which are not aligned to the X-Z and Y-Z planes, as homes not always tend to be built after the MW building rules. Many similar approaches have been developed, such as [Adan, 2011; Becker, 2015; Hong, 2015; Jung, 2014; Ochmann, 2016; Sanchez, 2012; Thomson, 2015], but all have similar shortcomings for our problem.

In recent years, many approaches solved the object recognition task in 3D models with the help of Neural Networks (NNs). These can be mainly split between approaches trying to detect individual objects, such as in [Liu, 2017; Maturana, 2015; Qi, 2016; Qi, 2019] or proposals that understand whole scenes, such as [Dai, 2018; Huang, 2018; Jiang, 2018; Tchapmi, 2017; Wang, 2018]. Both types of approaches have in common that they train deep Convolutional Neural Networks (CNNs) on 3D models or PCD. The trained network models can then predict the classes of objects and scenes in new geometry meshes and rooms. Most proposals, such as [Dai, 2018; Liu, 2017; Maturana, 2015; Qi, 2016; Qi, 2019; Tchapmi, 2017] also use some form of voxelized representations for the analysis of their objects and point clouds.

In Figure 1.4, a sample of these approaches can be seen, more specifically the approach from Dai et al. They actually go a little further and first complete a partial scan of the environment and then predict the semantics, representing them in the voxelized representation, which makes it a lot easier to analyse the structures, since the model is aligned to a standardised grid. These CNN approaches tend to perform rather well, but have one major drawback. They need to be trained beforehand and afterwards have to process the PCD through the trained network. This can, especially on devices with limiting hardware such as the HoloLens, consume most of the processing power, leaving little room for the actual application to run. Even optimizing for the specialized HPUs of the HoloLens 2 is not an option, as other devices we want to support, such as the first-generation HoloLens and the Magic Leap, do not possess such a specialized CPU. Due to the described issues, we decided against a NN solution.

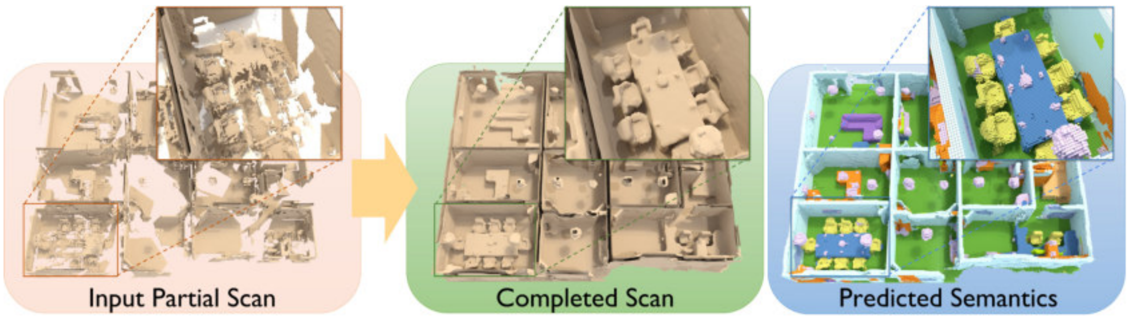


Figure 1.4.: Predicting the room semantics as proposed by Dai et al. First, they take the partial input scan and complete it through processing it in a specialised 3D CNN in different voxel resolutions. By doing this several times, they get a more precise prediction for the final room resolution, which is shown in the voxelized representation on the right. Image taken from [Dai, 2018].

Considering that all of the previously mentioned solutions have some promising approaches, but do not meet all of our requirements, we will implement a combination of the above. That enables us to provide a real-time room understanding approach, which can run on various hardware and supports any software without reinventing the wheel. We will go into more details on the adopted parts in the implementation found in Chapter 3.

1.2.2. Space Categorization

The next part of this thesis will investigate a suitable categorization for our valid placement areas for interactive content. The related work in this topic is very limited, as there are not many existing solutions for AR and VR devices. The only two APIs found, which incorporated a categorization feature for placing objects in a virtual environment, are available for the HoloLens and HoloLens 2. Whereas Section 1.2.1 introduces the spatial understanding functionalities of [Microsoft, 2018b] and [Microsoft, 2019b], we will now go into detail on the categorization process.

Microsoft's solution for the original HoloLens can categorise the following surface types: invalid, other, floor, floor-like, platform, ceiling, wall external, and wall-like [Microsoft, 2018b]. The new scene understanding API from Microsoft for the HoloLens 2 can label the scene into the following surface types: background, wall, floor, ceiling, platform, world, and unknown [Microsoft, 2019c]. For surface types, this is a relatively complete categorisation, but they do not answer the question of how we can interact with each content placed in a respective area. For this, interesting categories would be the following: can the user stand in front of the areas to interact with the content, is it only accessible over a distance, or is it in a hard to reach spot and most suitable for only informative objects without interaction.

As these existing solutions are incomplete for our use case, we will look into this further within Chapter 3, where we use some of the labels above, replace some by more meaningful classification, and introduce new categories.

1.2.3. Room Generation

The last topic of this thesis will be the generation of new environments on top of the real-world boundaries of the room. This topic has not been the focus of many recent works, so we will mainly concentrate on the ability to place 3D models in correct positions based on the matching structure of the input PCD or the detection of the model's shape to generate a replacement model.

There are many approaches to recognize objects in 3D environments, mostly based on DNNs, such as [Dai, 2018; Hou, 2019a; Hou, 2019b; Wu, 2019]. These methods fixate on completing scans and then detecting objects within. This is helpful for identifying certain objects in a room such as chairs, lamps or other furniture items. The approaches from [Avetisyan, 2019a; Avetisyan, 2019b; Dai, 2019; You, 2018] take this one step further and place computer-aided design (CAD) models above the detected object at the correct orientation and position. Nevertheless, both approaches do not suit our problem, as we want to detect the bounding area of our room and then place different models in such a way that the boundaries are kept.

Previous work, which are focused towards that direction are approaches trying to model cities from 3D PCD, such as in [Ennafii, 2018; Lafarge, 2012; Özdemir, 2018].

These solutions try to represent the shapes of buildings through geometric 3D-primitives extracted through the clearly defined edges of buildings. In Figure 1.5 taken from Özdemir et al.'s approach, we can see how this concept may look. On the left, we have the original point cloud and on the right the reconstructed simplified model. These concepts do not meet our requirements to the fullest again, since we want to reconstruct the inside of a room with simplified bounding shapes. Nevertheless, it has some interesting ideas for detecting the simplified shapes of a point cloud. Unfortunately, these approaches use very costly algorithms to create those reconstructions, as a real-time application was not the goal of these ideas.

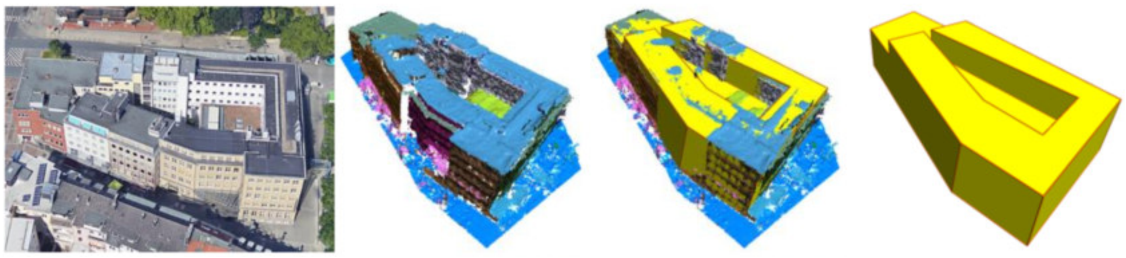


Figure 1.5.: Example of the reconstructed building from a point cloud. On the left, we have an image of the original building, followed by the PCD. On the right, the final and simplified reconstructed shape of the building is shown, which is preceded by the merge between the point cloud and the reconstructed model, showcasing the captured and missed details. Image taken from [Özdemir, 2018].

As both categories to approach the room recreation do not meet our requirements to the fullest, we will again implement our own solution, which is explained in more detail in Chapter 3.

Many different approaches already exist to spatially understand a room, with some already appearing on consumer-ready hard and software solutions. Nevertheless, all results fall short on some level compared to the actual issues we are trying to solve. Some explanations have a high impact on performance and available resources (DNNs), while others are only applicable to specific hardware such as the HoloLens ([Microsoft, 2018b; Microsoft, 2019b]). But all results are not exhaustive to create procedurally generated and digitally recreated environments specially designed for interactive content. Through the implementation presented in this thesis, such a solution is achieved.

1.3. Aim of Thesis

This thesis will investigate the possibility to automatically reproduce real-world boundaries in the virtual world optimised for interactive content in real-time and thus provide the option to dynamically adjust the virtual environment of the user. Moreover, we will look into providing a universal solution for any individual room scan. The origin of the room scan can be any imaginable generation method for 3D structures, such as geometries sourced from spatial maps of AR devices, recreating the room from two cameras located on VR headsets and with the help of reconstruction algorithms such as Structure from Motion (SfM). Additionally, we will provide an independent solution, which can be used on any platform in conjunction with various hardware. The implemented solution will achieve this in three steps.

First, we will analyse the point cloud of the scanned room for suitable locations for interactive content, mainly on walls, floors, ceilings, and furniture. The implementation then precedes in classifying each valid area, providing anchors for the recreation of the room in the virtual environment. As we want to create a software-independent solution, this work will take best practices for creating a plug-in for multiple environments, such as game engines, into account. Afterwards, this thesis will test the solution on recommended VR hardware configurations. As such, the results of this thesis will provide important insights into understanding a room created from a point cloud in real-time, correctly classifying suitable areas for interactive content and recreating a virtual environment based on the boundaries defined by the real room.

2. General Methods and Definitions

With this chapter, the thesis provides some fundamental understandings necessary to follow the developed approach in Chapter 3. We will give an overview of game engines, mesh representations, and room scanning methods, to highlight why certain approaches were used and where the limits of our solution lie.

2.1. Game Engines

The best way to recreate a three-dimensional model in a synthetic environment is a game engine. Game engines are normally easy to access software frameworks that are designed for the creation and development of video games. A core feature of a game engine is to provide a platform for not just one game, but rather an environment used for many different games [Hudlicka, 2009].

In the early days of game development, games usually were written as a single entity, designed from the bottom up to make optimal use of the hardware. As the performance of the hardware grew, so did the scope of the games. Designing each game from the ground up was getting less economical and resulted in the development of game engines. These combined frequently used features, such as a rendering engine for two or three-dimensional graphics, a physics engine, sound, scripting or animations. This reduced the cost and time to develop a game drastically, allowing to build more complex games with larger scopes [Lowood, 2014; Paul, 2012].

Recent trends in the usage of game engines, due to the maturing of the technology and more user-friendly environments, are not limited to just video games anymore, but are also used in serious games or other applications using 3D models [Paul, 2012]. Frostbite, Lumberyard, REDengine, Unity and Unreal Engine are just a few of the modern game engines that were the base of some well-known triple-A game (AAA game) titles [Amazon, 2020; CD Projekt, 2020; EA, 2020; Epic Games, 2020c; Unity, 2020g].

The Unreal Engine and Unity will be highlighted in more detail, as we chose to develop a direct support of our plugin for both engines. With this decision, we cannot only support the two biggest engines regarding AR and VR development [Kim, 2019], but additionally cover a large portion of possible implementation variants for our plugin, as both engines use different languages and philosophies for embedding third party libraries. Since our plugin is written in C++, the API can be directly included in the Unreal Engine by linking to the external library and in Unity we just add the DLL file into the project folder [Epic Games, 2018; Unity, 2019]. Since Unity uses a different scripting language, it tests the compatibility of a C++ created plugin within a C# environment, while the Unreal Engine natively supports C++.

2.1.1. Unreal Engine

The Unreal Engine is developed by Epic Games and was first released in 1998 with the first-person shooter (FPS) *Unreal* and was developed by the founder of Epic Games, Tim Sweeney [Horvath, 2012]. Unreal Engine 2 followed in 2002, debuting with the game *America's Army*, an FPS developed by the U.S. Army as part of its recruitment strategy [McLeroy, 2008]. In 2006, the third instalment of the Unreal Engine was published and *Gears of War*, an FPS for the Xbox 360, was one of the first games powered by it [Reed, 2004]. The latest version, Unreal Engine 4, was released in 2014 and has been receiving a regular stream of updates since [Sweeney, 2014].

Earlier versions of the Unreal Engine were very focused on the FPS genre, while later instalments tried to shift the spotlight to a variety of game development areas. This led to the support of over 20 different platforms with the Unreal Engine 4, ranging from PC, consoles and mobile support to VR (such as HTC Vive and Oculus Rift) and AR (such as Magic Leap and HoloLens 2). This makes Unreal a complete suite of development tools for real-time technology. The engine comes with full access to the C++ source code and the C++ API can be easily extended through new classes and features. With all these features, it makes this engine a preferred environment for Triple-A games and their studios, enabling developers to take full advantage of AR and VR technologies [Epic Games, 2020b].

For implementing and testing our plugin, we used the Unreal Engine 4.23, the latest stable release of the framework available at the start of this thesis in early November 2019 [Epic Games, 2020d].

2.1.2. Unity

Unity was officially launched in 2005, seven years after the Unreal Engine's first appearance. It was aiming to make game development possible regardless of know-how and budget and was originally released for Mac OS X and later added support for Microsoft Windows [Axon, 2016]. The original Unity game engine was followed by Unity 2 in 2007, Unity 3 in 2010, Unity 4 in 2012 and Unity 5 in 2015 [Cohen, 2007; Girard, 2010; Robertson, 2015; Tach, 2012]. In late 2016, Unity Technologies announced a change from the sequence-based versioning numbering system towards the year of release, together with a new release cycle and Long Term Support (LTS) releases. This followed with the Unity versions 2017.x, 2018.x and 2019.x for each respective year [Batchelor, 2016].

Unity always had a focus on allowing a wide variety of developers to create their games, independently of genre or preferred perspective of two-dimensional (2D) or 3D scenes [Unity, 2020h]. With this approach, they made sure that developers can create their content for a large diverse line-up of different hardware, supporting over 20 platforms in their current stable release. This ranges, similarly to the Unreal Engine, from mobile, across consoles and PC, all the way to AR and VR, supporting some devices that are ignored by the Unreal Engine, such as the original HoloLens [Unity, 2020a]. Unity comes with a wide range of extensions, which allow it to be used in many different industries and scenarios [Unity, 2020c].

For implementing and testing our plugin, we used Unity 2019.3, which was released at the end of January 2020 in a stable version [Unity, 2020f]. Additionally, we used Unity as our primary platform for testing the libraries functionality by utilizing a plugin. This allowed us to replace a loaded DLL in Unity, enabling us to faster cycle through testing and tweaking our implemented solution [Smith, 2019]. Despite the primary development taking place with Unity, the final implementation for both platforms does not differ.

2.2. Surface Representations

During this thesis, we will talk about a few different surface representations. First off, this work will give a small overview of surface representations for 3D objects in computer graphics and then dive deeper into the methods found within this thesis.

The simplest method, to represent raw data in 3D space is a point cloud, where there is a set of 3D point coordinates. A more structured representations approach is the volumetric representation, where the object is represented through an ordered grid. The most popular approach for computer games to digitally store a 3D model is the polygonal mesh, where the model is defined by many polygonal faces. Instead of polygonal meshes, parametric surfaces are an option, where a set of points defines a control grid, creating a surface through the help of Bézier curves. There are other options for representing the surfaces, but Bézier curves form the most popular choice. Other methods, which are not as important for this thesis, are implicit surfaces (contours through some scalar field in 3D), explicit surfaces (represented through some functional form and mostly used for height maps, such as for terrains) and constructive solid geometries (creates complex models through boolean operations combining simpler models) [Bourke, 1997; Niessner, 2018a; Niessner, 2018b]. In Figure 2.1, we present these seven forms of surface representations visually.

2.2.1. Polygonal Meshes

The information on polygonal meshes is based on Bærentzen et al.'s book "Polygonal Meshes" and will mainly focus on triangle meshes, a popular subgroup of the polygonal mesh. The popularity of the triangle meshes is based on the following reasons: graphics cards can render a large number of triangles in real-time, graphics processing units (GPUs) are optimized to handle these triangular meshes, and since the size of triangle meshes is ever increasing, they can fall back on many algorithms for manipulating triangle meshes [Bærentzen, 2012].

Nevertheless, many other types of polygons are used and are sometimes more useful,

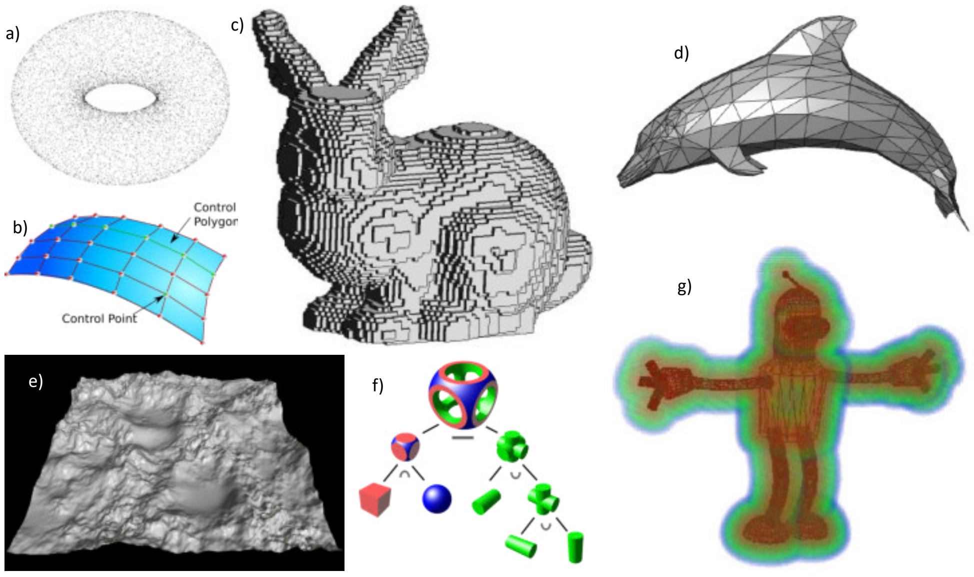


Figure 2.1.: A collection of surface representations: point cloud (a), parametric surface (b), volumetric representation (c), polygonal mesh (d), explicit surface (e), constructive solid geometry (f), and implicit surface (g). Images taken from [Niessner, 2018a; Niessner, 2018b; Wikimedia, 2019a; Wikimedia, 2019b; Wikimedia, 2020a].

compared to the triangle representation. Quadrilaterals or quads are a popular approach for modelling tools, as they tend to often align better with the shapes of an object than triangles would. These are primarily used for voxelized representation, which are converted to the polygonal form. For other types of applications, different polygons with even more edges are preferred. As previously mentioned, there are many different forms of surface representations in the virtual world, but the polygonal mesh representation is widely used for geometry processing and all modern game engines implement some form of polygonal meshes to represent their 3D objects [Bærentzen, 2012]. The Unity and Unreal Engine both use triangle meshes in their representation, making the triangle mesh representation the main focus of this chapter, as this depiction is the best performing representation method on graphics hardware [Epic Games, 2020a; Unity, 2020d].

Figure 2.2 exemplifies the composition of a triangle mesh. It is a set of faces $\mathcal{F}$, edges $\mathcal{E}$ and vertices $\mathcal{V}$. Each vertex $\mathbf{v}_i$ corresponds to a point in 3D space $\mathbf{v}_i = (x, y, z)$, where x, y, and z correspond to the position of the vertex on each axis of the coordinate system, and two vertices $\mathbf{v}_i$ and $\mathbf{v}_j$ define an edge $\mathbf{e}_i$ as follows: $\mathbf{e}_i = \mathbf{v}_j - \mathbf{v}_i$. In turn, a face $\mathbf{f}_i$, or triangle, of the object is defined by three edges ($\mathbf{f}_i = \{\mathbf{e}_i, \mathbf{e}_j, \mathbf{e}_k\}$) or the three vertices of the edges ($\mathbf{f}_i = \{\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k\}$). The numbering of vertices, edges and faces always starts at zero and goes up to one short of the total count of elements. Through a list of faces, edges and vertices, any 3D model can be represented [Bærentzen, 2012].

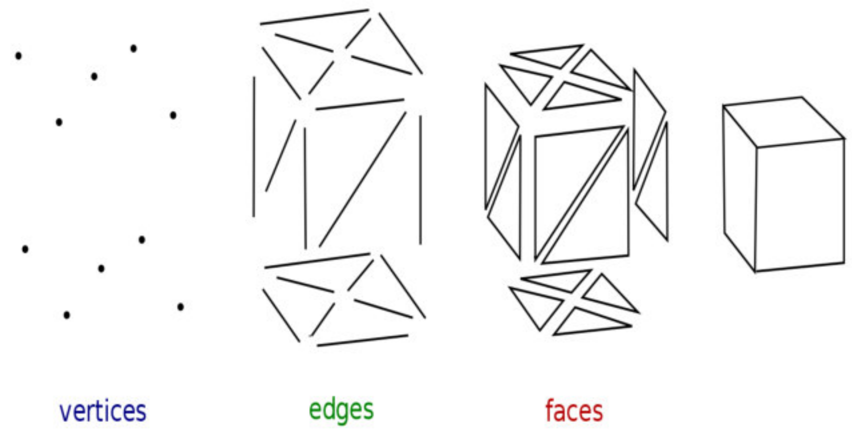


Figure 2.2.: Parts of a polygonal mesh representation, more specifically a triangle mesh. The representations consists of three main elements (from left to right): vertices, edges and faces. The surface of an object consists of many triangular faces, where each face is defined by three edges and each edge is defined by two vertices. Image taken and slightly adjusted from [Wikimedia, 2020b].

2.2.2. Point Clouds

The point cloud is a far more simplistic representation method than the previous described polygonal meshes. Nevertheless, it is a very important raw 3D object representation, as it is mainly used by laser scanners technology as an output format. A point cloud always consists of an array of vertices $\mathbf{v}_i = (x, y, z)$, with the axis coordinates defining its location in 3D space. Optional attributes can contain the colour ($\mathbf{c} = (r, g, b\{, a\})$, with r, g, b, and a being the values for red, green, blue and optionally alpha) either in the red, green, and blue (RGB) or red, green, blue, and alpha (RGBA) format and the vertex normal ($\mathbf{n} = (x, y, z)$, with x, y, and z as the normal vector direction towards each coordinate axis) [Niessner, 2018a; Rouse, 2016].

Figure 2.3 shows how a point cloud can be represented. In this image, all vertices drawn at their 3D coordinate location can be seen and the colour highlights the normal vector of each point, by matching the x, y, and z coordinates directly to the RGB format. A few downsides appear through this representation, such as no definition of surfaces and the missing possibilities of inter- or extrapolation. Nevertheless, this representation method is very powerful for storing and displaying the information of huge data clouds, as no complex relation between the single vertices exist and graphic cards do not have to compute which parts of a triangle lie in which pixel, as it only has to check each individual pixel [Niessner, 2018a; Rouse, 2016].

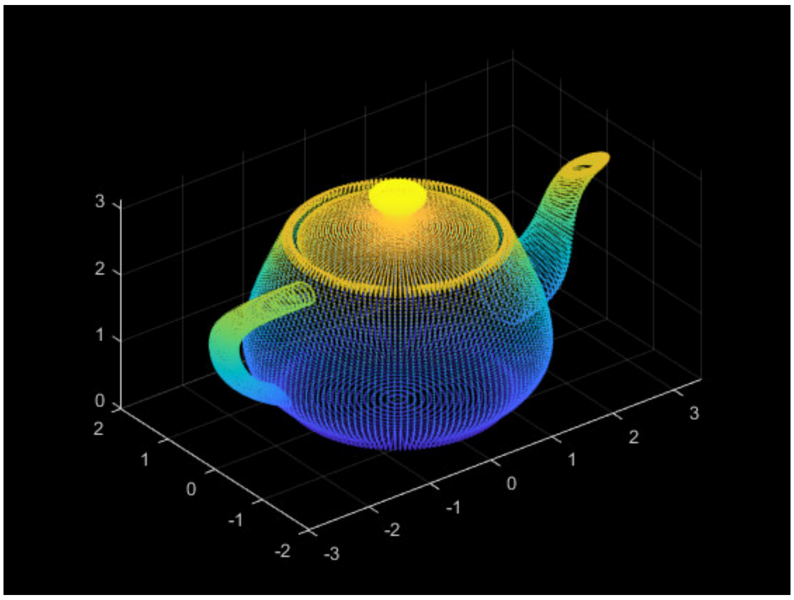


Figure 2.3.: Point cloud representation of a teapot. The model is represented by individual points, which contain colour data for each vertex based on the normal direction, where the RGB values directly correspond to the x, y, and z coordinates of the normal vector. The position of each vertex is defined by the 3D coordinates of the point. Image taken from [MathWorks, 2020a].

2.2.3. Volumetric Representations

A volumetric representation portraits a 3D object in an orderly grid, where a voxel represents a value within this grid. The word voxel comes from the combination of pixel and volume, as a voxel is nothing else, but a pixel projected into a 3D environment. It is defined by the location of the point as $\mathbf{v}_i = (x, y, z)$ and any additional data corresponding to that point. The simplest form of a volumetric representation is the spatial-occupancy enumeration, where each cell only decides if it is active or not. Through this, an object can be represented through an array of active cells. Nevertheless, more information can be saved for each voxel, such as colour, special attributes, flow rate, density or pressure. Due to these characteristics, voxel representations are often used for medical and scientific data, as they can represent complex patterns in a simplistic and ordered structure [Foley, 1996; Hansen, 2011; Oomes, 1997].

It is fairly easy to switch to the voxelized representation, as shown in Figure 2.4. There, the triangle mesh of the object is converted to a volumetric representation, where every space occupied by a mesh face is an active voxel. Depending on the resolution (dimension of a single voxel), more or less details of the original model can be captured in the voxelized representation. [Foley, 1996; Hansen, 2011; Oomes, 1997].

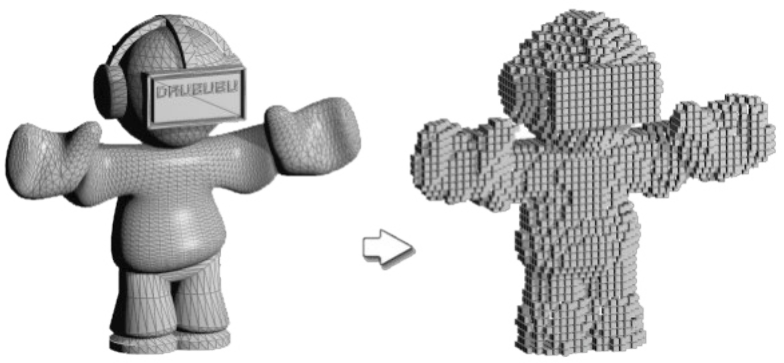


Figure 2.4.: Comparison of the same 3D model, once as a polygonal mesh (left) and once in a volumetric representation (right). For every space in the ordered grid that is occupied by the mesh, a voxel is activated and displayed. Depending on the resolution, a different level of detail can be captured. Image taken from [Pickton, 2012].

2.3. Room Scanning

To better understand, which types of room representations are most likely encountered, the next section presents the most common scanning and reconstruction techniques: Laser Scanning, SfM, Simultaneous Localization and Mapping (SLAM), and red, green, blue, and depth (RGB-D) reconstruction.

2.3.1. Laser Scanning

Laser scanners, also known as light detection and ranging (LiDAR), are used in a variety of scenarios to capture detailed representations of an object, mainly for room, building, and landscape scans. To capture the surrounding, these scanners emit laser light pulses across a wide field of view (FOV) to detect detailed spatial location and shape of objects. Through a high data collection rate, the scanners can capture complex structures in a short period of time, compared to traditional methods, such as recreating a room from images. The resulting format is a point cloud that can be combined with image recordings of the real, creating a more detailed virtual representation. Additionally, LiDARs sometimes use Global Positioning System (GPS) information to accurately place the scanned area, or combine the scans of multiple scanners. Recently, the laser scanning technology is often used in games, to capture realistic and detailed environments to be represented within the digital worlds [Delta, 2020; LSE, 2020; NOAA, 2020; Wasser, 2020].

Figure 2.5 shows an example of a room scan through the LiDAR technology. This can capture many details of the room, but also has areas with almost no points, as the scanner was unable to reach those places. These holes in the structure have to be

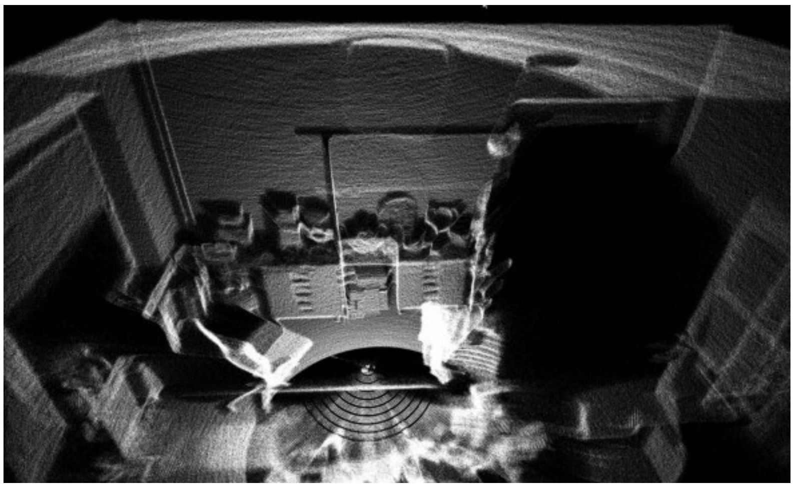


Figure 2.5.: A LiDAR scan of a room. This point cloud with almost five million points represents many details of the room, but some areas might not be reached with the scan and will create black holes within the point cloud. Image taken from [Hackster, 2017].

accounted for in our solution, because often enough, room sections are occluded through obstacles in between the scanner and objects. Since this technique is often used with BIMs (as seen in the image), to capture the inside of buildings, PCDs needs to be a valid input format for our algorithm, to allow a wide compatibility for different use cases and capture approaches for AR and VR room environments.

2.3.2. Structure from Motion

To recreate an object from a collection of unordered images, SfM is a prevalent strategy since it has the goal to recover both 3D points and camera positions. Due to this, it is used in many applications such as 3D scanning and AR. In this section, we will discuss the incremental SfM algorithm, as it gives a good overview of the process and is widely adapted. This sequential processing pipeline commonly starts with feature extractions from each image and matching them over the set of images. With those paired key points, the matches can be solved for the transformation matrix of the camera and the 3D location of the key points. With bundle adjustment, the reprojection error can be minimized to give a more accurate point cloud for the object of interest. With this technique, a sparse point cloud of an object is obtained. A denser point cloud of an object can be created with the Multi-View Stereo (MVS) algorithm [Hoiem, 2017b; MathWorks, 2020b; Niessner, 2018c; Schonberger, 2016].

This can be used to recreate entire parts of a city, as seen in Figure 2.6. Despite being a slow approach to capture the details of an object, it is an easy method to recreate rooms from a set of images and thus can be used to capture the surrounding of an AR or VR play space. It does, however, come with the same downsides as the laser scanners. Point

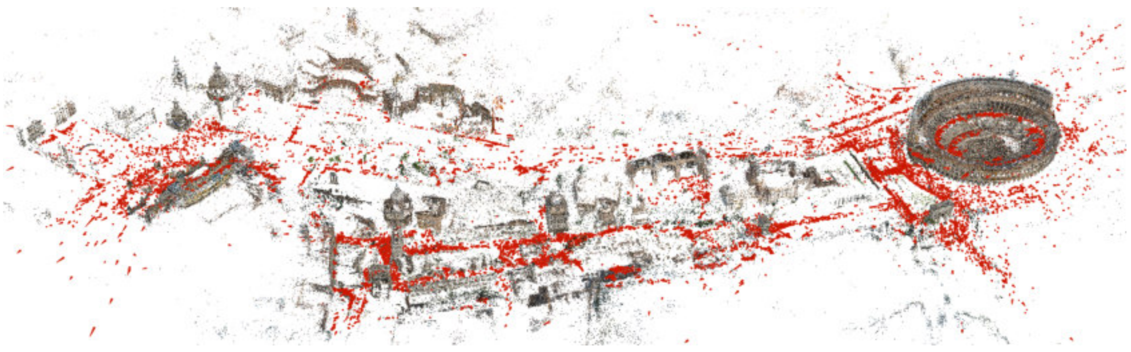


Figure 2.6.: A sparse point cloud of parts of Rome, created with SfM. This technique is able to recreate buildings from a collection of images based on found features across the set of samples. Image taken from [Schonberger, 2016].

clouds created with this method can have holes in their structure, caused by occluding objects.

2.3.3. Simultaneous Localization and Mapping

Another method for creating scans of environments is SLAM. SLAM is mainly used for robots, as it creates a map of the environment, while estimating the pose of the image within the localization. For this, the point cloud is built up incrementally through RGB images or a video stream, which results in a smaller computational budget compared to SfM. Due to this, there is often only a frame-to-frame tracking, omitting the global optimization. Nevertheless, the functionality is very similar to SfM. First, we extract the key frames and pair features across a local key frame window. Afterwards, the bundle adjustment method is applied to obtain a sparse point cloud and the pose estimations of the key frames. To obtain a dense representation of the sparse point cloud, there are again existing algorithms [Burgard, 2012; Niessner, 2018c; Stachniss, 2016].

The herby created point cloud looks similar to the previous models, as can be seen in Figure 2.7. Again, this method provides the same problems for hard to reach areas creating holes in the point cloud.

Seeing that most raw data from all these methods result in a point cloud, it becomes clear that this data structure must be supported by our algorithm. A downside of point clouds is, as previously mentioned, that we do not have any information about the relation of the points to each other. This will become an important step our algorithm has to overcome.

2.3.4. RGB-D Reconstruction

Another often-used technique to reconstruct environments or objects is depth data. To obtain depth information, there exist three main techniques: stereo cameras, structured light and Time of Flight (ToF). With all three methods, the depth data is captured and

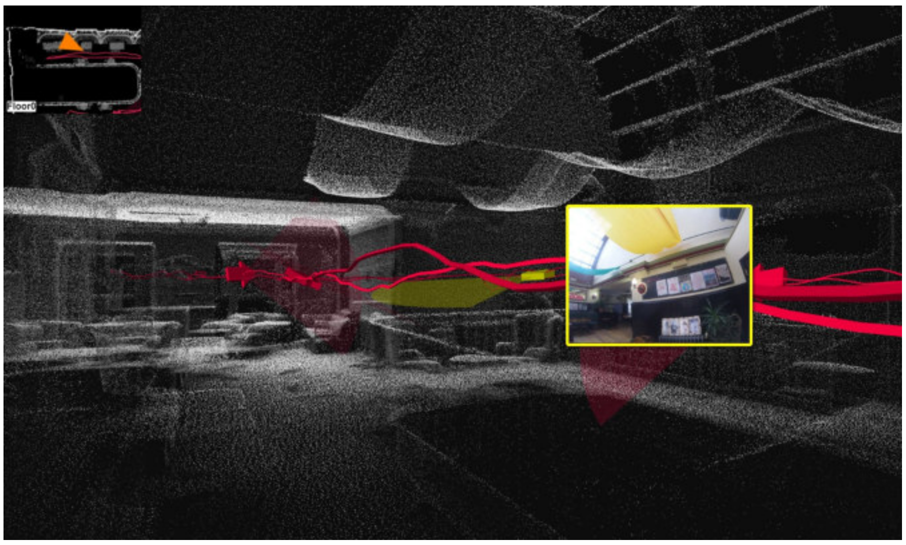


Figure 2.7.: A room scan taken with the SLAM method. The red line shows the movement of the camera through the room, while the yellow framed image shows a sample a keyframe from the video feed. Through SLAM, both the path of the camera and a reconstruction of the environment can be created. Image taken and slightly adjusted from [LES, 2020].

then converted to a point cloud. Alternatively, it can be used for spatial mapping to create a mesh geometry of the environment. The first approach, stereo cameras, mimics the depth sense of humans by placing two cameras at a certain distance apart from each other and then calculating the depth of each pixel by using a stereo matching algorithm. The structured light method takes this approach one step further and is used for example in the Microsoft Kinect. Here the depth information is captured by projecting known features into the 3D scene (e.g. points or lines) and then match those with stereo matching algorithms. The Kinect 2, appearing with the release of the Xbox One in 2013, replaced this technique with the ToF camera, the third method. Here, the sensor measures the round trip time of an artificial light signal provided by the scanner. The light pulse, unlike those from the similar LiDAR system, captures the whole scene with a single impulse, where LiDAR captures the environment point-by-point [Hoiem, 2017a; Niessner, 2018c].

Two examples for depth environment captures and often used for AR and VR devices are the ZED Mini and the HoloLens. The first device works independently and can be attached to an HMD, capturing the environment through two RGB cameras. The resulting depth image can be seen in Figure 2.8, complete with the resulting 3D point cloud [Stereolabs, 2020b]. The HoloLens on the other side uses a more sophisticated ToF approach with a total of eight data streams. These consist of four grey-scale cameras, used for head tracking and map building. Additionally, there are two versions of the depth camera data, one high frequency (30 frames per second (FPS)) near-depth sensing stream used for hand tracking and one low frequency (1-5 FPS) far-depth sensing data, mainly used for spatial mapping. Lastly, the HoloLens uses an infrared radiation (IR)-reflectivity stream (again one short and one long depth-sensing version) to compute the depth, remaining largely unaffected by ambient lighting. The resulting spatial mapping mesh created from all those data streams can be seen in Figure 2.9 [Microsoft, 2018a].

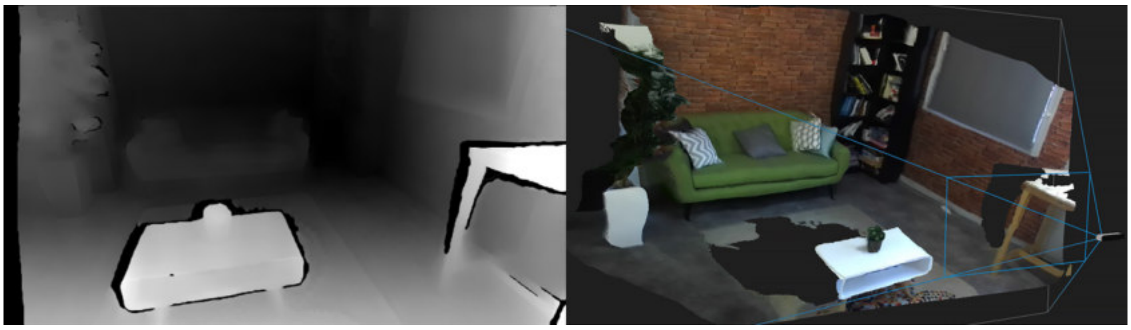


Figure 2.8.: Depth image generated with the stereo cameras from the ZED Mini. The left shows the recorded depth image, while the right showcases the resulting 3D point cloud. Areas that are obscured through other objects occlude entities and surfaces behind them, generating incomplete structures. Images taken from [Stereolabs, 2020b].

2.4. Plugins

As we are creating a plugin that should be compatible with many different software solutions, we will look at distinctive methods to implement widely useable plugins. There are two main approaches to create a plugin, which can be included by a program: a DLL or a static library.

2.4.1. Dynamic Link Library

DLLs, also known as shared libraries in UNIX-based operating systems, are used to share code and resources and shrink the size of applications. They exist as separate files outside of the executable code base and can be modified without recompiling the application. Moreover, the program only needs to make one copy of the library's files at compile-time and multiple running applications can still use the same library data. Nevertheless, DLLs come with some downsides, since they are more prone to breaking a program. If for example, the library becomes corrupt, the executable file usually no longer works. Additionally, a function call to a DLL is slower compared to static libraries, as it is called from outside of the executable [Kuredjian, 2017; Microsoft, 2019d].

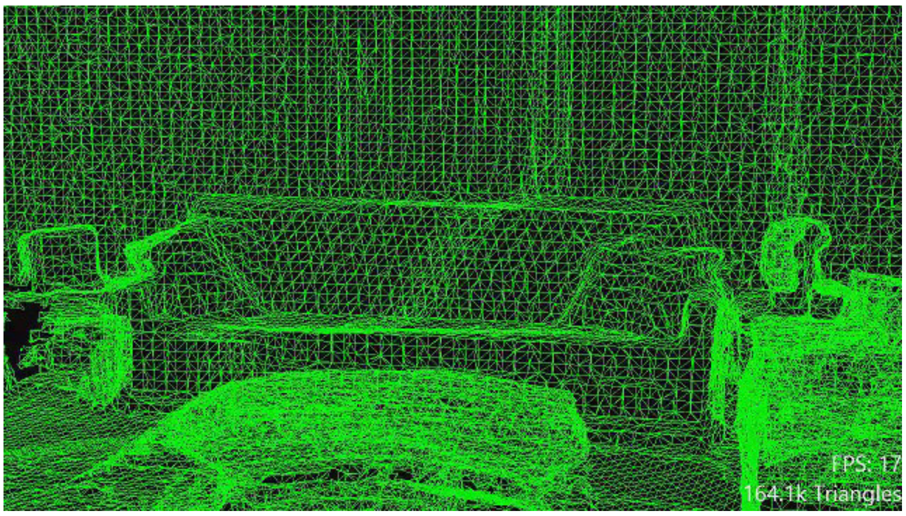


Figure 2.9.: Devices such as the HoloLens, can generate spatial map meshes from depth streams, captured through on device hardware technology. The HoloLens uses data from eight different data streams (four grey-scale cameras, two versions of depth camera data (high and low frequency), and two versions of IR-reflectivity streams (high and low frequency)). Image taken from [Microsoft, 2018c].

2.4.2. Static Library

Static libraries, as the name already suggests, are static implementations, since they are locked into a program at compile-time. This is the major downside of this library type, as a modification on the plugin needs a re-compilation. Moreover, every file of the program must have its own copy of the library's files at compile-time. The upsides of a static library are that it cannot be corrupted outside your application, as it lives inside the executable file. Additionally, the execution speed of static libraries is much quicker, compared to DLLs, due to the binary object code being included in the executable [Kuredjian, 2017; Microsoft, 2019e].

Usually, the decision of which library to use is based on the role of the plugin. However, we want to support a wide variety of programs, especially game engines such as Unity and Unreal Engine. Especially the Unity game engine comes with strict limitations for the inclusion of plugins. Only DLLs are allowed and can be implemented to be used within the Unity environment. For the Unreal Engine on the other hand, both implementation methods would be possible. Due to this restriction, we will implement our plugin as a DLL [Epic Games, 2018; Unity, 2019].

2.5. Requirements for AR and VR Applications

Gregory's book *Game Engine Architecture* states basic game engine requirements for VR devices, which need to be reached or overcome for a smooth user experience. These are stereoscopic rendering, very high frame rate and navigation issues. Stereoscopic rendering means that each frame needs to be rendered twice from a slightly different virtual camera, doubling the number of graphic primitives that need to be rendered. The high frame rate requirement is based on studies, showing that a VR application running below 90 FPS is likely to cause disorientation, nausea and other negative effects for the user. Gregory states that navigation might cause nausea, such as travelling by flying. This is the reason that most games opt for the point-and-click teleportation solution. All these requirements concern only the graphical side of the experience [Gregory, 2019].

On AR devices, we do not have such strict requirements towards a high FPS rate, as the user still sees his surrounding in real-time, and only virtual elements are drawn over the real environment. Nevertheless, a frame rate that is too low will still cause some discomfort, due to lagging virtual objects. Moreover, AR devices usually have a limited amount of hardware, making it vital, that any application running in parallel to the actual application has a low performance impact on the device [Gregory, 2019].

Because of these strict requirements on the side of AR and VR, we have to make sure, that our plugin does not cause any issues with the FPS or blocking the main thread of the application in any case. This would cause a negative user experience, which would negate the usefulness of our solution.

3. Room Detection and Recreation

In this chapter we will discuss the implementation of our solution as a DLL in C++. Before going through the actual implementation, the preprocessing of the data will be presented. Afterwards, this work goes through the three main points of our solution in detail, consisting of the room detection, space classification and room recreation. At the end, the usage of the created plugin is explained with examples for Unity and the Unreal Engine.

3.1. Pre-Processing

For passing data between the game engine and our plugin, we have to account for some language specific problems, especially for converting the data from Unity to our library and vice versa. Since Unity uses C# as a scripting language, we need an option to convert an array of information from C++ to C#, which is supported by `IntPtr`, a special type used to refer to data between languages that do and do not support pointers [Microsoft, 2020b]. This structure is implemented in the presented solution for the inclusion of the plugin within C# and thus can only use data arrays of basic value types, such as integer or floating-point values, limiting the types of information usable as in and output for the library.

As we decided to support the most basic form of room representations within our plugin, the library only takes an array of all vertex positions $\mathcal{V}_{\text{Input}}$ from the room as input. Each vertex $\mathbf{v}_i$ corresponds to a point in 3D space, defining $\mathbf{v}_i = (x, y, z)$ as the vertex with a position in the 3D coordinate system through corresponding values on the x, y, and z axis. The total number of vertices in the mesh is established through $n = |\mathcal{V}|$. With the described input limitations, the vertex positions can be passed as a flattened floating-point array with the following format:

$$\mathcal{V}_{\text{Input}} = \left\{ \mathbf{v}_{1_x}, \mathbf{v}_{1_y}, \mathbf{v}_{1_z}, \mathbf{v}_{2_x}, \mathbf{v}_{2_y}, \mathbf{v}_{2_z}, \ldots, \mathbf{v}_{n_x}, \mathbf{v}_{n_y}, \mathbf{v}_{n_z} \right\} . \tag{3.1}$$

Since software for creating or processing 3D models does not have a uniform coordinate system, we need to define a system that is used internally by our solution. With a large portion of the plugin tests conducted in Unity, the adoption of Unity's left-handed, Y-up coordinate system was the obvious choice. The directions are defined as follows, considering the default viewing orientation established through the forward direction: Z-axis goes from backward (negative) to forward (positive), X-axis goes from left (negative) to right (positive), and the Y-axis goes from down (negative) to up (positive). A visual representation of the used coordinate system can be seen in Figure 3.1.

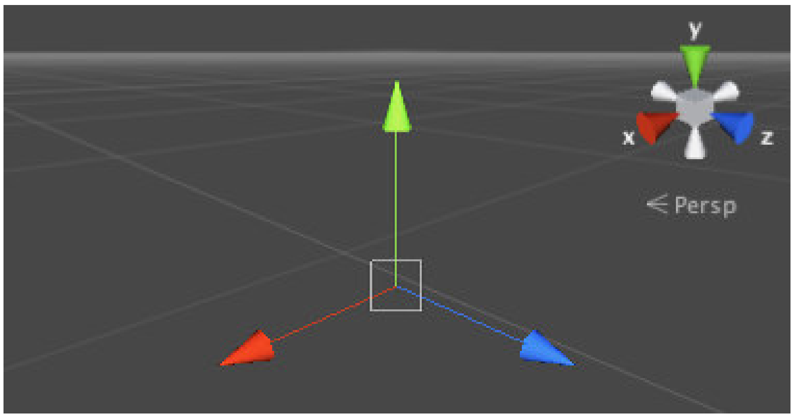


Figure 3.1.: The coordinate system used in our plugin. We use the same left-handed, Y-up coordinate system as used within Unity. The axes are represented through the following colours: X = red, Y = green, and Z = blue. The directions are defined as follows: Z-axis goes from backward (negative) to forward (positive), X-axis goes from left (negative) to right (positive), and the Y-axis goes from down (negative) to up (positive). Image taken from [Unity, 2010].

In turn, this has the consequence that diverging coordinate systems have to be converted, before the plugin can use the transferred data. In the Unreal Engine fo example, the used left-handed, Z-up coordinate system produces the following assignment for the internally used axes: $x_{\text{Plugin}} = y_{\text{Unreal}}$, $y_{\text{Plugin}} = z_{\text{Unreal}}$, $z_{\text{Plugin}} = x_{\text{Unreal}}$.

Since we use Unity's unit scale system of one unit length equals one metre (m), the internally used system of the Unreal Engine, which is set to one unit length equal to one centimetre (cm), needs a further conversion step. This gives us the following conversion forms from a point in the Unreal Engine $\mathbf{p}_{\text{Unreal}}$ to a point in our plugin $\mathbf{p}_{\text{Plugin}}$ (Equation 3.2) and vice versa (Equation 3.3).

$$\begin{aligned} \mathbf{p}_{\text{Plugin}_x} &= \frac{\mathbf{p}_{\text{Unreal}_y}}{100} \\ \mathbf{p}_{\text{Plugin}_y} &= \frac{\mathbf{p}_{\text{Unreal}_z}}{100} \\ \mathbf{p}_{\text{Plugin}_z} &= \frac{\mathbf{p}_{\text{Unreal}_x}}{100} \end{aligned} \tag{3.2}$$

$$\begin{aligned} \mathbf{p}_{\text{Unreal}_x} &= \mathbf{p}_{\text{Plugin}_z} \times 100 \\ \mathbf{p}_{\text{Unreal}_y} &= \mathbf{p}_{\text{Plugin}_x} \times 100 \\ \mathbf{p}_{\text{Unreal}_z} &= \mathbf{p}_{\text{Plugin}_y} \times 100 \end{aligned} \tag{3.3}$$

If the targeted software uses another coordinate system and unit scale length, this has to be adjusted accordingly. Nevertheless, our solution does not provide multiple functions for different coordinate system. It only implements a single function that uses the internal system and scale. This could, however, be added in future work to this plugin.

The next step is to bring the input point cloud representation to the internally used voxel representation from our plugin. The dimensions of a single voxel are $0.1 \times 0.1 \times 0.1$ m^3 (Length $\times$ Width $\times$ Height). Since voxels are always cubed for our grid, we define the voxel size $S_{\text{Voxel}} = 0.1$ and use it for each dimension of the voxel. For this, we first converted the flattened input array back to a 3D vector representation. Then we defined the dimensions of our voxel grid for the length L_{Grid}, width W_{Grid}, and height H_{Grid}:

$$\begin{aligned} L_{\text{Grid}} &= \left\lfloor \frac{\mathbf{v}_{max_x} - \mathbf{v}_{min_x}}{S_{\text{Voxel}}} \right\rfloor + 1 \\ W_{\text{Grid}} &= \left\lfloor \frac{\mathbf{v}_{max_z} - \mathbf{v}_{min_z}}{S_{\text{Voxel}}} \right\rfloor + 1 \\ H_{\text{Grid}} &= \left\lfloor \frac{\mathbf{v}_{max_y} - \mathbf{v}_{min_y}}{S_{\text{Voxel}}} \right\rfloor + 1 \; . \end{aligned} \tag{3.4}$$

Here and hereinafter, $\lfloor a \rfloor$ is the floor function, which rounds to the closest integer less than or equal to a. The values for $\mathbf{v}_{max}$ and $\mathbf{v}_{min}$ for each axis are obtained through finding the largest and smallest coordinates present in all input vertex positions for each axis respectively. The following equations present this process exemplary for the X-axis:

$$\begin{aligned} \mathbf{v}_{max_x} &= \max\left(\left\{\mathbf{v}_{1_x}, \ldots, \mathbf{v}_{n_x}\right\}\right) \\ \mathbf{v}_{min_x} &= \min\left(\left\{\mathbf{v}_{1_x}, \ldots, \mathbf{v}_{n_x}\right\}\right) \; . \end{aligned} \tag{3.5}$$

The Equation 3.4, which gets the dimensions of the voxel grid, is based on the total space between the smallest and largest coordinates (numerator) and fitting it to the size of a single voxel cell (denominator). Through taking the floor of the fraction and adding one at the end, we ensure that the dimension is always large enough to accommodate all coordinate values in between the minimum and maximum.

After obtaining the maximum dimensions of the internal voxel representation grid, voxels can be activated if a point from the input point cloud lies within a voxel cell. We determine the indices ($\mathcal{I}_l$ for the length, $\mathcal{I}_w$ for the width, and $\mathcal{I}_h$ for the height) of the voxel for a vertex $\mathbf{v}_i$ through the following formulas:

$$\begin{aligned} \mathcal{I}_l &= \left\lfloor \frac{\mathbf{v}_{i_x} - \mathbf{v}_{min_x}}{S_{\text{Voxel}}} \right\rfloor \\ \mathcal{I}_w &= \left\lfloor \frac{\mathbf{v}_{i_z} - \mathbf{v}_{min_z}}{S_{\text{Voxel}}} \right\rfloor \\ \mathcal{I}_h &= \left\lfloor \frac{\mathbf{v}_{i_y} - \mathbf{v}_{min_y}}{S_{\text{Voxel}}} \right\rfloor \; . \end{aligned} \tag{3.6}$$

With the formulas in Equation 3.6, we can go through all input vertex positions and activate all voxel cells defined through $\mathcal{I}_l$, $\mathcal{I}_w$, and $\mathcal{I}_h$ in our voxel grid, which is defined in Equation 3.7. With this step an ordered grid representation of the input point cloud of the room is obtained. Depending on the defined voxel dimensions, few (small value) or many (large value) details will get lost in this conversion step. We choose ten cm for our voxel size, since it strikes a good balance between relevant details kept and the impact on performance through our implemented algorithms. Larger voxel dimensions will result in faster execution times (as the number of values in the ordered grid decreases), but will return less accurate results, since details of the room get lost. With this, we obtain a voxelized representation as a 3D matrix **V**, with the previous defined dimensions.

$$\mathbf{V} = (v_{xyz}) \in \mathbb{B}^{L_{\text{Grid}} \times H_{\text{Grid}} \times W_{\text{Grid}}}$$
$$(v_{xyz}) = \begin{cases} 1, & \text{if } x = \mathcal{I}_l \land y = \mathcal{I}_h \land z = \mathcal{I}_w \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

Through the above definition, we activate a voxel v_{xyz} by setting it to one if the index of each dimension matches the x, y and z position of the voxel. The values for x, y and z lie in the interval from zero to their respective dimension maximum that is inclusive for zero and exclusive for the maximum. For the following work, it is important to note that for any indices mentioned the count starts at zero and not at one. Additionally, we define $\mathbb{B} \in \{0, 1\}$ with $0 :=$ false and $1 :=$ true, allowing calculation shortcuts in later sections. This will from here on be the definition of $\mathbb{B}$.

In Figure 3.2, a sample room[1] scanned by a HoloLens is converted into a voxelized representation. Figure 3.2a and 3.2b show this process for the outside of the room and Figure 3.2c and 3.2d for the inside. It is clearly visible that certain details get lost in this conversion, but for an approximate estimation of the room, our chosen voxelized representation has a good enough quality, as demonstrated in the evaluation chapter (Chapter 4).

3.2. Room Feature Detections

The first step of our algorithm is to detect basic features of the room, building the foundation for the definition of suitable areas for the interactive content (Section 3.3). The following subsection might seem randomly ordered at first glance, but the ordering is defined through the dependence of each step on previously detected features. At first, we can define the floor and ceiling levels of a room and afterwards the location of walls is detected. With both of those in place, the following algorithms are able to detect furniture locations and valid tiles for the floor and ceiling. Since all three parts require no data from each other, we are running them in parallel to optimize the execution time on slow hardware and for large rooms (Chapter 3.7). For defining suitable areas on walls in later methods, the normals for the detected walls spots are needed.

[1]The sample rooms used to test the algorithms are showcased in more detail in Chapter 4.

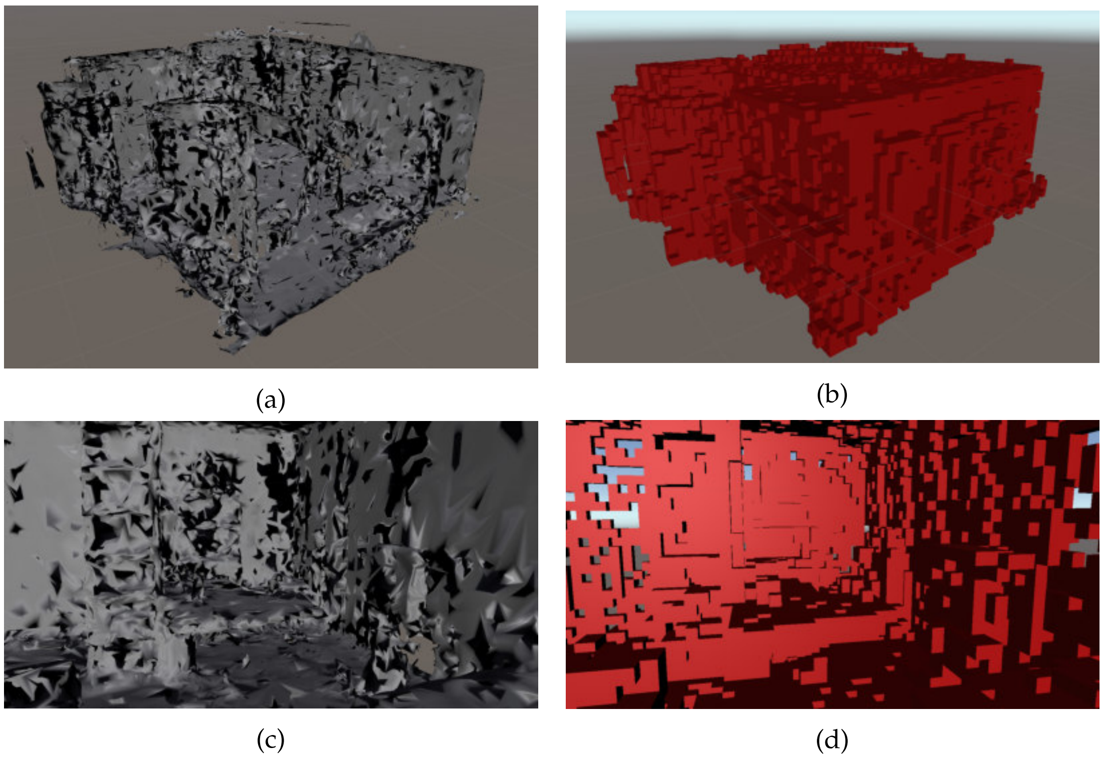


Figure 3.2.: These screenshots showcase a room as a mesh representation (a and c) and from the same perspective as a voxel representation (b and d). The top row depicts the room from the outside and the bottom row from the inside. As the triangles of the room are only drawn from one side, Image (a) misses many geometry aspects visually, since these can only be seen from the opposite direction. In this comparison, the details that are lost and preserved through the conversion can be detected. Images are screenshots taken within the Unity Editor.

3.2.1. Floor and Ceiling Level Detection

The floor and ceiling levels are the backbone of the following algorithms, as they allow to optimize the extraction of features by limiting the search area to the space between these two planes. The first step taken is defining the index levels, on which the floor ($\mathcal{I}_{\text{Floor}}$) and ceiling($\mathcal{I}_{\text{Ceiling}}$) are located at. For this, we assume that in a point cloud of a room, the floor and ceiling are always parallel to the X-Z-plane of the coordinate system and that through this parallelism a floor and ceiling level can be detected through a large amount of active voxels located on a plane parallel to the X-Z-plane.

To detect the two planes with the highest amount of active voxels in our internal voxelized representation, we count the amount of active voxels on each Y-level $|\mathbf{V}_y|$, which is defined by the sum of all active voxels for each Y-Level:

$$|\mathbf{V}_y| = \sum_{x=0}^{L_{\text{Grid}}} \sum_{z=0}^{W_{\text{Grid}}} v_{xyz} \ . \tag{3.8}$$

The occurrence of the highest two voxel counts gives us the floor and ceiling level and is specified by the following equations, which save the highest ($\mathcal{I}_1$) and second highest ($\mathcal{I}_2$) index:

$$\begin{aligned} \mathcal{I}_1 &= y \mid \max\left(\{|\mathbf{V}_y| \mid \forall y \in \mathbb{N}_0 \land y \leq H_{\text{Grid}}\}\right) \\ \mathcal{I}_2 &= y \mid \max\left(\{|\mathbf{V}_y| \mid \forall y \in \mathbb{N}_0 \land y \leq H_{\text{Grid}} \land y \neq \mathcal{I}_1\}\right) \ . \end{aligned} \tag{3.9}$$

Knowing the two highest voxel counts for any plane parallel to the X-Z-plane, the floor ($\mathcal{I}_{\text{Floor}}$) and ceiling ($\mathcal{I}_{\text{Ceiling}}$) levels can be determined, since the ceiling level has to be above the floor:

$$\begin{aligned} \mathcal{I}_{\text{Floor}} &= \min\left(\{\mathcal{I}_1, \mathcal{I}_2\}\right) \\ \mathcal{I}_{\text{Ceiling}} &= \max\left(\{\mathcal{I}_1, \mathcal{I}_2\}\right) \ . \end{aligned} \tag{3.10}$$

Afterwards, we assume that the floor and ceiling level are at least 1.5 m apart. This step is done to safely calculate valid detection areas for the floor and ceiling. Justification for the aforementioned step comes through the fact that a room with less vertical space between the floor and ceiling will not be a suitable area for an HMD to be used.

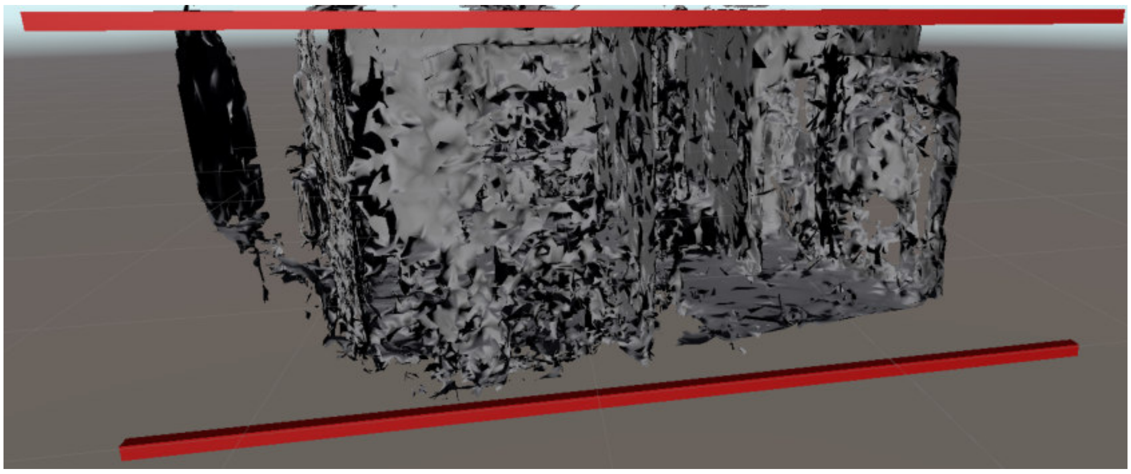


Figure 3.3.: The two red bars show the floor (bottom) and ceiling (top) level for this room. It aligns correctly with the floor and ceiling of the room mesh. Image is a screenshot taken within the Unity Editor.

In Figure 3.3, the results of the above described algorithm is shown on the sample room. The algorithm correctly detects the floor and ceiling level of the room and marks them with the red bars.

3.2.2. Wall Detection

With the floor and ceiling levels detected, the search for walls can be started with an optimized algorithm, as only the space between these two levels has to be taken into account. To simplify the feature detection process in this step, we assume that the height of a ceiling above the floor must be at least 2.3 m, as required by German law for living quarters. In reality, the height will average at least 2.5 m [UGC, 2020]. In the algorithm, a wall is defined as at least 50 percent of the total ceiling height. Since holes in the room scan have to be assumed and furniture are partially covering wall areas, half the rooms height (1.15 m) can be safely defined as a wall. We reckon that factors such as higher ceilings or inaccuracies of the scan (holes, displaced points of the point cloud) increase the minimum wall threshold by at least 30 percent, resulting in walls needing at least 1.5 m of plane surface to be recognized. Using this threshold, even when the perceived part is not a wall (such as a cabinet), the object will be too high to allow for comfortable usage of interactive content placed on top of it. Thus, we might get incorrectly labelled walls, which are nevertheless used as walls, since the detected surface allows for the placement of interactive content on a vertical plane area.

To detect walls, every location on the X-Z-plane is checked on how many activated voxels it contains across the Y-dimension of the internal model. The information is stored in the matrix $\mathbf{A}_{\text{Wall}}$:

$$\begin{aligned}
\mathbf{A}_{\text{Wall}} &= (a_{xz}) \in \mathbb{B}^{L_{\text{Grid}} \times W_{\text{Grid}}} \\
(a_{xz}) &= \begin{cases} 1, & \text{if } |\mathbf{V}_{xz}| > \left(\mathcal{I}_{\text{Ceiling}} - \mathcal{I}_{\text{Floor}}\right) \times 0.5 \\ 0, & \text{otherwise} \end{cases} \\
|\mathbf{V}_{xz}| &= \sum_{y=\mathcal{I}_{\text{Floor}}}^{\mathcal{I}_{\text{Ceiling}}} v_{xyz} \ .
\end{aligned} \tag{3.11}$$

In parallel, this algorithm extracts all positions on the X-Z-plane containing an active voxel in the Y-direction. This allows us to later define if an extracted wall space points into the room or faces the outside and is thus not suitable for placing content. For the achievement of the aforementioned step, each location on the X-Z-plane is viewed and the detection of any active voxel on the Y-dimension is recorded. If an active voxel was found, the result is saved in matrix $\mathbf{B}_{\text{Room}}$. To fill any holes, each false entry in this matrix is cycled again and checked for at least three active neighbouring spaces. If this is the case, this position is also activated. The matrix $\mathbf{B}_{\text{Room}}$ is defined as follows:

$$\begin{aligned}
\mathbf{B}_{\text{Room}} &= (b_{xz}) \in \mathbb{B}^{L_{\text{Grid}} \times W_{\text{Grid}}} \\
(b_{xz}) &= \begin{cases} 1, & \text{if } \exists v_{xyz} = 1 \mid \forall y \in \mathbb{N}_0 \wedge \mathcal{I}_{\text{Floor}} \leq y \leq \mathcal{I}_{\text{Ceiling}} \\ 1, & \text{if } b_{xz} = 0 \wedge b_{(x+1)z} + b_{(x-1)z} + b_{x(z+1)} + b_{x(z-1)} \geq 3 \\ 0, & \text{otherwise} \end{cases} \ .
\end{aligned} \tag{3.12}$$

In practise, Figure 3.4 proves that the above algorithm provides good results for

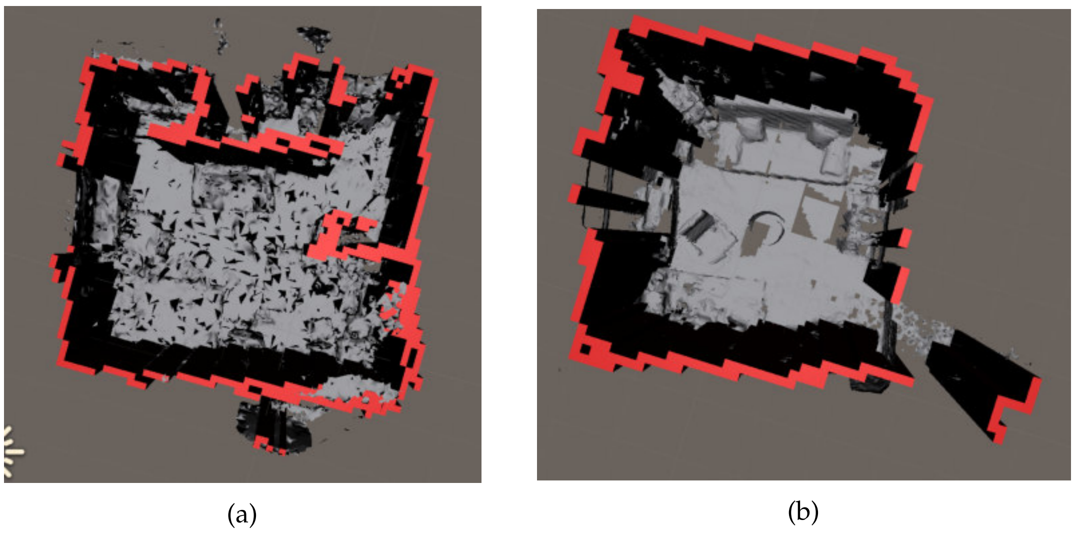


Figure 3.4.: The results from our wall detection algorithm. It can detect more than 90 percent of the walls very well, while also omitting windows from the results (left side on both rooms). Holes in the wall, due to incomplete scanning data, will cause this algorithm to fail (top right-handed corner (a) and right side (b)). Some redundant scanning data of the outside of the room (independent wall detections at the bottom (a) and bottom left-handed corner (b)) can lead to false wall detections. Images are screenshots taken within the Unity Editor.

detecting walls in rooms. However, the proposed method had problems in both rooms to detect windows (large open space on the left) and larger holes in the wall, where the scan data was insufficient (3.4a top right-hand corner and 3.4a right side). Especially for AR applications the windows, correctly classified not as a wall, are actually a welcome behaviour, as placing holograms over a bright light source from the outside will hinder a good visibility of the hologram. As the algorithm works well on more than 90 percent of the walls in the room, we can say that incorrectly labelled walls are mostly caused by faults in the provided scanning data and less in the proposed algorithm.

3.2.3. Furniture Location Detection

For the detection of furniture features in the voxelized room representation, we assume that a detected furniture item is only suitable for putting interactive content on top if it, if it has a maximum height of 1.5 m. Above this height, it will become difficult to place content for easy access and usability for the user. Since we only want to detect the top surfaces of any furniture, our proposed method starts to detect active voxels at a Z-level of 1.5 m. It stores the level of the first encounter for every position on the X-Y-plane in matrix $\mathbf{C}_{\text{Furniture}}$. Any already detected wall spaces are exempt from this detection, as they are already classified as wall and cannot be also furniture. Additionally excluded are floor spaces, since they will be differently classified in the algorithm described in the next chapter. To be a suitable tile for an interaction area placed onto a furniture surface, we defined that at least one m of clear space needs to be above this spot. This way, interactive content with a maximum height of one m or lower can be placed. This results in the following definition of matrix $\mathbf{C}_{\text{Furniture}}$:

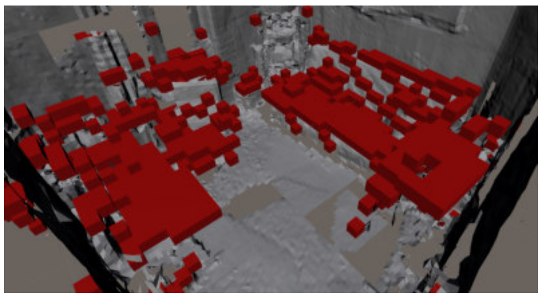
$$\begin{aligned}
\mathbf{C}_{\text{Furniture}} &= (c_{xz}) \in \mathbb{Z}^{L_{\text{Grid}} \times W_{\text{Grid}}} \\
(c_{xz}) &= \max_{\mathcal{I}_{\text{Floor}} < y \leq \mathcal{I}_{y\text{End}}} f_{\text{Level}}(y) \\
f_{\text{Level}}(y) &= \begin{cases} y+1, & \text{if } a_{xz} = 0 \wedge \sum\limits_{i=y+1}^{f_{\text{Limit}}(y)} v_{xiz} = 0 \wedge v_{xyz} = 1 \wedge f_{\text{Limit}}(y) < \mathcal{I}_{\text{Ceiling}} \\ -1, & \text{otherwise} \end{cases} \\
f_{\text{Limit}}(y) &= \max\left(y + \left\lfloor \frac{1}{S_{\text{Voxel}}} \right\rfloor, \mathcal{I}_{y\text{End}} \right) \\
\mathcal{I}_{y\text{End}} &= \mathcal{I}_{\text{Floor}} + \left\lfloor \frac{1.5}{S_{\text{Voxel}}} \right\rfloor .
\end{aligned} \tag{3.13}$$

The above definition for the matrix $\mathbf{C}_{\text{Furniture}}$ guarantees that only the active voxels with the highest Y-level between the floor and $\mathcal{I}_{y\text{End}}$ are stored. This is ensured by checking the sum of all voxels in the one m zone above the current position or up to the 1.5 m threshold, whichever is higher. No voxels within this limit is allowed to be active (equal to one) for a valid furniture tile. To confirm if an element is a wall tile, the in Equation 3.11 defined matrix is checked for the corresponding element on the X-Z-plane ($a_{xz} = 0$). For $f_{\text{Level}}(y)$ the value $y + 1$ is returned for a valid level, as the clipping of interactive content into the geometry of the room should be kept at a minimum. By placing the valid tile one space above the last active voxel, the chances of clipping are minimized.

To test the feasibility of the in Equation 3.13 defined matrix, its result was visualized in Figure 3.5 on two sample rooms. Here, the red cubes mark the detected level of the furniture. Since only the flat surface parallel to the X-Z-plane on top of the furniture are of interest, there is no need to detect and define the vertical sides of the objects. Tables, beds, sofas and other furniture objects get detected by our implemented approach and flat surfaces are clearly marked.

3.2.4. Valid Floor and Ceiling Tile Detection

For both the floor and ceiling tile detection, very similar algorithms are used, with the main difference being the direction the method checks for empty areas above and below the valid tile respectively. Valid floor tiles are tested for the following three factors: do they have at least 1.5 m of free area above the surface, is the location of the tile within the room, and is it not already part of a wall. The ceilings are checked for the


(a)

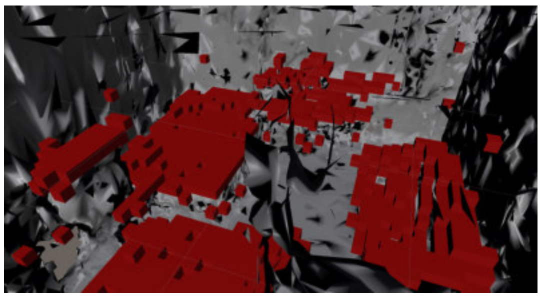

(b)

Figure 3.5.: The red cubes show the results from the in Equation 3.13 defined functions. With this approach, clearly defined outlines of tables, beds, sofas, and other furniture objects are obtained. Since only the flat surfaces parallel to the X-Z-plane are of interest, the sides of the furniture are not detected and defined. Images are screenshots taken within the Unity Editor.

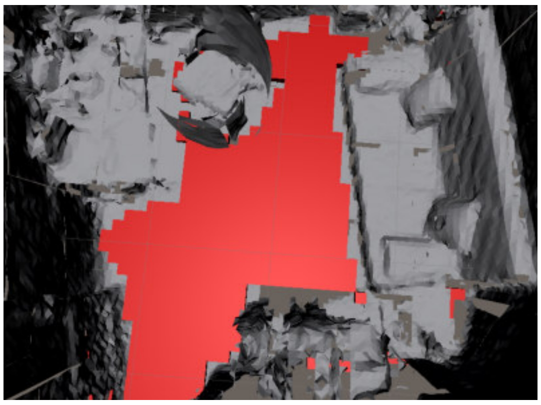
same parameters with the only difference being the free area check. Here an area of up to one m below the ceiling tile is investigated if it is not occupied by a voxel. These informations will be saved in the matrices $\mathbf{D}_{\text{Floor}}$ and $\mathbf{E}_{\text{Ceiling}}$ and are defined as follows:

$$\mathbf{D}_{\text{Floor}} = (d_{xz}) \in \mathbb{B}^{L_{\text{Grid}} \times W_{\text{Grid}}}$$
$$(d_{xz}) = \begin{cases} 1, & \text{if } a_{xz} = 0 \land b_{xz} = 1 \land \sum\limits_{y=\mathcal{I}_{\text{Floor}}+1}^{\mathcal{I}_{y\text{End}}} v_{xyz} = 0 \\ 0, & \text{otherwise} \end{cases}, \tag{3.14}$$

$$\mathbf{E}_{\text{Ceiling}} = (e_{xz}) \in \mathbb{B}^{L_{\text{Grid}} \times W_{\text{Grid}}}$$
$$(e_{xz}) = \begin{cases} 1, & \text{if } a_{xz} = 0 \land b_{xz} = 1 \land \sum\limits_{y=\mathcal{I}_{y\text{Start}}}^{\mathcal{I}_{\text{Ceiling}}-1} v_{xyz} = 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.15}$$
$$\mathcal{I}_{y\text{Start}} = \mathcal{I}_{\text{Ceiling}} - \left\lfloor \frac{1}{S_{\text{Voxel}}} \right\rfloor .$$

The check for walls is again done with the in Equation 3.11 defined matrix ($a_{xz} = 0$). Additionally, the valid tiles need to overlap with the in Equation 3.12 established room ($b_{xz} = 1$). The maximum distance from the floor is already defined in $\mathcal{I}_{y\text{End}}$ from Equation 3.13. We only need to add the maximum distance from the ceiling with $\mathcal{I}_{y\text{Start}}$. Moreover, the summation process is started and ended one Y-level early respectively, since the actual voxel on the floor and ceiling level is of no interest to the established algorithm. This method easily fill holes that might appear through incomplete scanning data.

In practice, we get very good results from the algorithm described in Equation 3.14 and 3.15, as can be seen in Figure 3.6. For both sample rooms, the approach marks


(a)

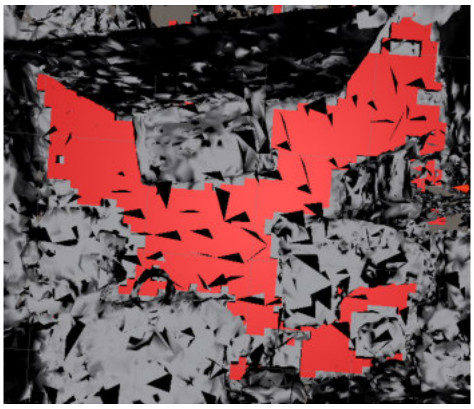

(b)

Figure 3.6.: The results from the in Equation 3.14 and 3.15 defined algorithm. These images only showcase the results for the floor tiles, by placing a red cube on each valid tile. Only tiles on the floor level are marked, which have at least 1.5 m of free space above them, are not occupied by a wall, and are within the room. Images are screenshots taken within the Unity Editor.

all suitable floor tiles correctly and highlights the floor as one large area, with a few outliers.

3.2.5. Wall Normal Detection

In the last feature detection step, the normals of the walls are defined. This information is needed to correctly orient a suitable wall area away from the vertical surface. In contrast, this is not needed for floor, furniture and ceiling areas, since they are parallel to the X-Z-plane and have always the same orientation. Through the voxelized representation of a room, each voxel can point into up to four different directions. These four directions are defined with the following normalized vectors:

$$\begin{aligned} \text{Left} &:= (-1,0,0) \\ \text{Right} &:= (1,0,0) \\ \text{Forward} &:= (0,0,1) \\ \text{Backward} &:= (0,0,-1)\ . \end{aligned} \tag{3.16}$$

Enumeration flags are used, since an efficient assignment of these normals to the matrix $\mathbf{F}_{\text{Normals}}$, which will hold the wall normals for each voxel location, is favourable. Additionally, this allows the bit field treatment,, which combines defined enumerator values through setting certain bits active. For example, if "Left" is defined with a one and for simplification purposes it is assumed that the enumerator is defined by a four bit unsigned integer, this one would look like 0001 in the bit representation[2]. If

[2]Binary is a base-2 number system, that calculates for example the unsigned integer through adding the

"Right" is defined as 2 (0010), the direction combination of "Left" and "Right" for a voxel would be defined as 3 by combining both directions with a bitwise or operator (0011) and converting the bit representation back to the integer format. We now define our directions to integers corresponding to the base-2 number column:

$$\begin{aligned} \text{None} &:= 0\ (0000) \\ \text{Left} &:= 1\ (0001) \\ \text{Right} &:= 2\ (0010) \\ \text{Forward} &:= 4\ (0100) \\ \text{Backward} &:= 8\ (1000)\ . \end{aligned} \tag{3.17}$$

With this the definition of all possible elements can be created, which is established in $S_{\text{Direction}}$. This gives a maximum of 16 different normal combinations, as each direction can either be active (1) or inactive (0) and four bits are needed at most to represent all four labels ($2 \times 2 \times 2 \times 2 = 16$). We define $S_{\text{Direction}}$ as follows to represent the aforementioned:

$$S_{\text{Direction}} = \{s \in \mathbb{N}_0 \mid s < 16\}\ . \tag{3.18}$$

Each s in Equation 3.18 defines one combination of directions. The direction for a valid wall voxel is calculated next by starting to consider all active voxels in matrix $\mathbf{A}_{\text{Wall}}$ individually. For each suitable wall voxel, every Y-level between the floor and the ceiling is checked for one m ($\mathcal{I}_{\text{Bounds}}$) into the four directions (the normal orientation is defined through the vectors in Equation 3.16), if they do not contain an active voxel, are not a wall, are within the room, are not at or below the level of a furniture object, and are within the bounds of the voxelized representation. The voxel that is one step towards the defined normal direction from the original wall tile is marked with the orientation, if all of these conditions are met. This minimizes the clipping of suitable wall areas, similar to previous algorithms. Moreover, this shift by one voxel verifies that no wall tiles intersect valid vertical areas for interactive content, allowing us to define the matrix $\mathbf{F}_{\text{Normals}}$, where the orientations are saved for each voxel.

$$\begin{aligned} \mathbf{F}_{\text{Normals}} &= (f_{xyz}) \in S_{\text{Direction}}^{L_{\text{Grid}} \times H_{\text{Grid}} \times W_{\text{Grid}}} \\ \mathcal{I}_{\text{Bounds}} &= \left\lfloor \frac{1}{S_{\text{Voxel}}} \right\rfloor \end{aligned} \tag{3.19}$$

Algorithm 1 describes the process of filling the 3D matrix $\mathbf{F}_{\text{Normals}}$ with the valid normal directions for adjacent walls. The "OR" operator used at the end of the method describes the bitwise or operator [Mohabia, 2018] and is used for adding a direction to a voxel, which will keep any previously assigned directions. Moreover, assigning an

resulting number columns (similar to the traditional base-10 system). For example, our 0001 can be represented by $2^3 \times 0 + 2^2 \times 0 + 2^1 \times 0 + 2^0 \times 1 = 8 \times 0 + 4 \times 0 + 2 \times 0 + 1 \times 1 = 1$. For a more detailed explanation of the binary system, please visit [Wienand, 2019].

Algorithm 1: Defining the wall normal orientation for each voxel

```
Result: $\mathbf{F}_{\text{Normals}}$
foreach $f_{xyz} \in \mathbf{F}_{\text{Normals}}$ do
    $f_{xyz} = 0$;
end
foreach $v_{xyz} \in \mathbf{V} \wedge \mathcal{I}_{\text{Floor}} \leq y \leq \mathcal{I}_{\text{Ceiling}}$ do
    if $a_{xz} = 1$ then
        foreach $i \in \mathbb{N}_0 \wedge x - \mathcal{I}_{\text{Bounds}} \leq i < x$ do
            if $a_{iz} = 1 \vee v_{iyz} = 1 \vee b_{iz} = 0 \vee c_{iz} \geq y$ then
                Remember that the "Left" direction is not valid;
            end
        end
        foreach $i \in \mathbb{N}_0 \wedge x < i \leq x + \mathcal{I}_{\text{Bounds}}$ do
            if $i \geq L_{\text{Grid}} \vee a_{iz} = 1 \vee v_{iyz} = 1 \vee b_{iz} = 0 \vee c_{iz} \geq y$ then
                Remember that the "Right" direction is not valid;
            end
        end
        foreach $i \in \mathbb{N}_0 \wedge z - \mathcal{I}_{\text{Bounds}} \leq i < z$ do
            if $a_{xi} = 1 \vee v_{xyi} = 1 \vee b_{xi} = 0 \vee c_{xi} \geq y$ then
                Remember that the "Backward" direction is not valid;
            end
        end
        foreach $i \in \mathbb{N}_0 \wedge z < i \leq z + \mathcal{I}_{\text{Bounds}}$ do
            if $i \geq W_{\text{Grid}} \vee a_{xi} = 1 \vee v_{xyi} = 1 \vee b_{xi} = 0 \vee c_{xi} \geq y$ then
                Remember that the "Forward" direction is not valid;
            end
        end
        if "Left" direction is valid then
            $f_{x-1,y,z} = f_{x-1,y,z}$ OR Left;
        end
        if "Right" direction is valid then
            $f_{x+1,y,z} = f_{x+1,y,z}$ OR Right;
        end
        if "Backward" direction is valid then
            $f_{x,y,z-1} = f_{x,y,z-1}$ OR Backward;
        end
        if "Forward" direction is valid then
            $f_{x,y,z+1} = f_{x,y,z+1}$ OR Forward;
        end
    end
end
```

orientation again will not invalidate a direction, since the bitwise or operator outcome will always result in one, if at least one bit of the number column is activated.

3.3. Defining Suitable Areas for Interactive Content

With all required room features detected, we can now start defining the suitable areas for interactive content for walls, floors, ceilings, and flat furniture surfaces. No algorithm described below generates new informations needed by another method, allowing them to run in parallel. The results, however, are then saved in a single thread and in a predefined order afterwards, making the reduction step of the results in a later section easier. Areas are created in predefined sizes, reducing the performance impact of the proposed approaches and allowing a simpler reduction algorithm for smaller surfaces fitting into larger ones.

3.3.1. Suitable Wall Areas

First in order are suitable wall areas for interactive content. The following three rectangle dimensions (width $\times$ height) are chosen for the predefined areas: 2×1 m^2 (Large), 1×0.5 m^2 (Medium), and 0.5×0.3 m^2 (Small). The internally used voxelized representation makes the distance between two voxels not uniform, requiring a range of ± 0.1 for the width to capture voxels with not the exact predefined distance apart. In practice, the height h is calculated from the actual distance d between two voxels and can vary slightly from the predefined rectangle dimensions.

$$\begin{aligned} h &= \left\lfloor \frac{d}{2} \right\rfloor \\ d &= \sqrt{(x_2 - x_1)^2 + (z_2 - z_1)^2} \end{aligned} \tag{3.20}$$

The height h is half the actual distance between voxel v_{x_1,y_1,z_1} and v_{x_2,y_2,z_2}. Since the suitable wall location's normal is always parallel to the orientation of the X-Z-plane, there is only a need to calculate the distance of the two voxel based on the x and z coordinates. The y coordinates will always be the same for the distance calculation and can be ignored with an outcome of 0 for the distance equation. Voxel v_{x_1,y_1,z_1} will be selected for all possible x and z values and the range of the y coordinates is limited from $\mathcal{I}_{\text{Floor}} + \lfloor \frac{w_{\min}}{2} \rfloor$ towards $\mathcal{I}_{\text{Ceiling}}$, where w is the previously defined width and $w_{\min}$ and $w_{\max}$ are the lower and upper bounds of the distance range ($w \pm 0.1$). We start at a higher distance above the floor level, leaving room for the height of the rectangle, which is always added from the current level downwards. Voxel v_{x_2,y_2,z_2} will be chosen based on the coordinates of v_{x_1,y_1,z_1}, as not all possible voxels have to be reviewed, only those within the reach of the maximum distance between two corners. The limits for the coordinates are defined as follows:

$$\begin{aligned} x_1 \le x_2 &< \min(L_{\text{Grid}}, \lfloor x_1 + w_{\max} + 1 \rfloor) \\ f_{\text{Lower}}(z_2) \le z_2 &< \min(W_{\text{Grid}}, \lfloor x_1 + w_{\max} + 1 \rfloor) \\ f_{\text{Lower}}(z_2) &= \begin{cases} z_2 + 1, & \text{if } x_1 = x_2 \\ \lfloor z_2 - w_{\max} \rfloor, & \text{if } \lfloor z_2 - w_{\max} \rfloor > 0 \land x_1 \neq x_2 \\ 0, & \text{otherwise} \end{cases} . \end{aligned} \tag{3.21}$$

The y coordinates are not present, since $y_1 = y_2$. The upper bounds of both limits have an added one, so the maximum distance the two voxels could be apart is included in the range. The maximum range may never surpass the axis limit of the voxel grid. For z_2, this complex calculation was chosen for the lower bounds, in order to increase performance, allowing fewer cycles through a for loop if $x_1 = x_2$. Any other values, which are excluded from these ranges were either already checked (such as $x_2 < x_1$), are out of bounds, or are out of range from the maximum distance.

When the first voxel v_{x_1,y_1,z_1} is selected, we check if this voxel has a defined normal direction, and if yes, check for all voxels v_{x_2,y_2,z_2}, if they also have a valid normal direction. If the voxel v_{x_2,y_2,z_2} has only normals pointing in opposite directions to the orientations of the first voxel, voxel two is ignored. Otherwise, the distance between these two voxel is analysed to match a range of the three predefined rectangle widths. In case this is also fulfilled, a rectangle can be constructed and checked for being a suitable spot for interactive content.

For the rectangle creation some tricks from linear algebra were used, by defining the line segment between the two voxels. The linear equation for this segment is created by taking the aforementioned two suitable voxels, and if $x_1 \neq x_2$, the following equation is valid:

$$\begin{aligned} h(x) &= m(x - x_1) + z_1 \\ m &= \frac{z_2 - z_1}{x_2 - x_1} . \end{aligned} \tag{3.22}$$

For $x_1 = x_2$, this function is undefined and we create our rectangle in a different fashion explained further below. The two edge voxels of the line segment are already determined, which are represented in Figure 3.7 through red voxels on the bottom left (v_{x_1,y_1,z_1}) and on the top right (v_{x_2,y_2,z_2}). With Equation 3.22, we can now create the black line connecting the two edge voxels and determine if every voxel touching the line (green voxels in Figure 3.7) are suitable or not.

As in Figure 3.7 shown, our line segment goes from centre to centre. We now calculate for each voxel's within the line segment, the range of Z-coordinates containing suitable wall area candidates which are later checked for their validity (presence of normal). The values for x_{Lower} and x_{Upper} are calculated first, which is the range for every single

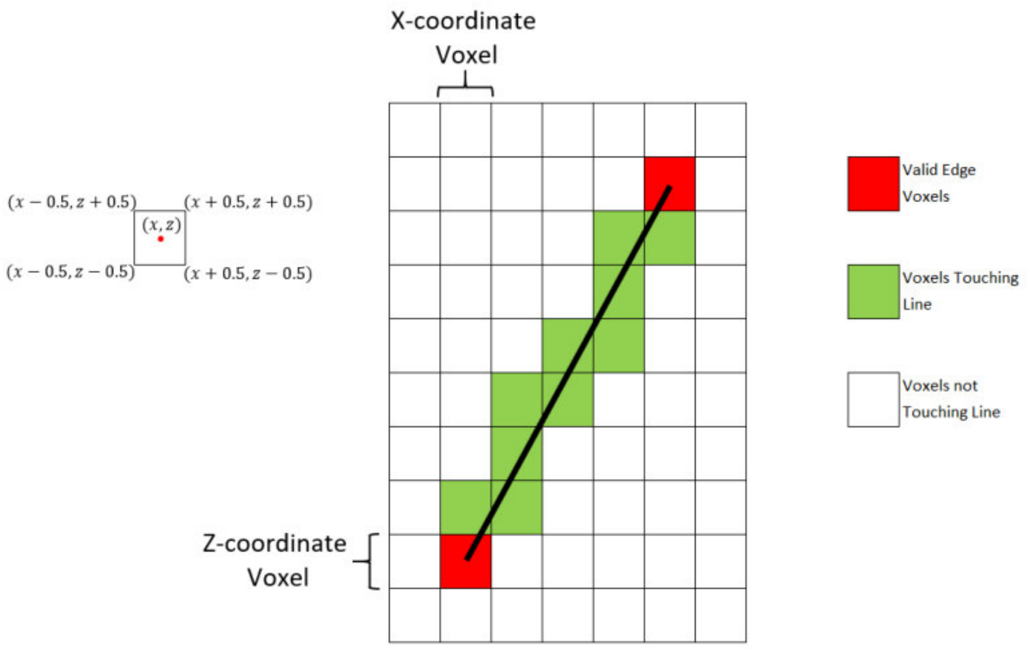


Figure 3.7.: An example top down view of the X-Z-plane, where two valid voxels have been located, which are within the range of the previously defined distance. The red voxels in the bottom left (v_{x_1,y_1,z_1}) and top right (v_{x_2,y_2,z_2}) corner represent these edge voxels. The black line connecting them is the in Equation 3.22 defined line segment and the green voxels are any voxels touching the line. White voxels are of no interest, as they are not part of our rectangle. The line segment represents the upper boarder of the rectangle. The centre of each square represents the (x, z) coordinate of the voxel and the corners are always 0.5 units away from the centre (see clarification on the left for the coordinates and values for each corner and centre of the voxels). Image was created with the help of Microsoft Excel.

X-coordinate within the line segment:

$$
\begin{gathered}
x_{\text{Lower}} = \begin{cases} x, & \text{if } x = x_1 \\ x - 0.5, & \text{otherwise} \end{cases} \\
x_{\text{Upper}} = \begin{cases} x, & \text{if } x = x_2 \\ x + 0.5, & \text{otherwise} \end{cases} \\
x \in \mathbb{N}_0 \wedge x_1 \leq x \leq x_2 \, .
\end{gathered}
\tag{3.23}
$$

The lower bounds z_{Lower} is the result of $h(x_{\text{Lower}})$, allowing to check the line segment's intersection with the voxel boundary on the left side. To convert $h(x_{\text{Lower}})$ to an integer value for the lowest Z-coordinate, the result is rounded depending on the slope. If the slope is positive and the fractional digit is equal or greater than 0.5, z_{Lower} is rounded up to the next integer, since the voxel below will not be passed by the line segment. If the slope is negative, all decimal points lower or equal to five get rounded down, since in this case the voxel below is hit by the line. Otherwise, the start is defined at the voxel index from $\lfloor h(x_{\text{Lower}}) \rfloor$. If the slope is undefined, z_1 is established for the lower bounds. The upper bounds z_{Upper} are calculated very similarly. The main difference is the interest in the outcome of $h(x_{\text{Upper}})$ instead, which checks where the line segment leaves the voxel boundary. Other small changes are that the slopes are exchanged for the check and that if the slope is undefined, the upper bound is defined at the centre of the end voxel at z_2. As outcome, all green voxels seen in Figure 3.7 are obtained. This gives us the following definitions for z_{Lower} and z_{Upper}:

$$z_{\text{Lower}} = \begin{cases} \lceil h(x_{\text{Lower}}) \rceil, & \text{if } m \geq 0 \land h(x_{\text{Lower}}) - \lfloor h(x_{\text{Lower}}) \rfloor \geq 0.5 \\ \lceil h(x_{\text{Lower}}) \rceil, & \text{if } m < 0 \land h(x_{\text{Lower}}) - \lfloor h(x_{\text{Lower}}) \rfloor > 0.5 \\ z_1, & \text{if } m \text{ undefined} \\ \lfloor h(x_{\text{Lower}}) \rfloor, & \text{otherwise} \end{cases}$$
$$z_{\text{Upper}} = \begin{cases} \left\lceil h(x_{\text{Upper}}) \right\rceil, & \text{if } m < 0 \land h(x_{\text{Upper}}) - \left\lfloor h(x_{\text{Upper}}) \right\rfloor \geq 0.5 \\ \left\lceil h(x_{\text{Upper}}) \right\rceil, & \text{if } m \geq 0 \land h(x_{\text{Upper}}) - \left\lfloor h(x_{\text{Upper}}) \right\rfloor > 0.5 \\ z_2, & \text{if } m \text{ undefined} \\ \left\lfloor h(x_{\text{Upper}}) \right\rfloor, & \text{otherwise} \end{cases} . \tag{3.24}$$

Subsequently, the algorithm views all extracted Z-coordinates by going from z_{Lower} to z_{Upper} for each X-coordinate in the range from x_1 to x_2. As soon as one checked tile was marked as a wall before ($a_{xz} = 1$), the found rectangle is no suitable wall spot and is discarded. All valid tiles are checked if a normal is defined and then the total amount of all such tiles is counted. We repeat this for each Y-coordinate between y_1 and $y_1 - h$, where h is obtained from Equation 3.20. If a suitable area is still possible after checking all valid tiles for the rectangle, the four corner points of the rectangle are saved if the suitable voxel count c_{suitable} is above a certain threshold. For our cases, we found the best value with a threshold of 80 percent:

$$c_{\text{suitable}} \geq 0.8 \times (\lfloor d \rfloor + 1) \times (h + 1) \ . \tag{3.25}$$

If not 80 percent of tiles in the rectangle are valid voxels, meaning possessing a normal vector, the area is not deemed suitable. The rectangle area is estimated by adding one to the distance d and height h, giving an approximation of columns and rows present in the rectangular surface, as overlapping voxels are not captured. The added one is necessary, as the distance and height both do not include the first row and columns of voxels. Through this, the following four corner points are obtained for the rectangles, by going through each corner, starting at voxel v_{x_1,y_1,z_1}, in a clockwise order:

$$\begin{aligned}
\mathbf{p}_{\text{Corner1}} &= (x_1 \quad, y_1 \qquad , z_1) \\
\mathbf{p}_{\text{Corner2}} &= (x_2 \quad, y_1 \qquad , z_2) \\
\mathbf{p}_{\text{Corner3}} &= (x_2 \quad, y_1 - h \quad , z_2) \\
\mathbf{p}_{\text{Corner4}} &= (x_1 \quad, y_1 - h \quad , z_1) \; .
\end{aligned} \tag{3.26}$$

These corners will then be reduced in further steps after all suitable spots were detected. For each of our three boundary sizes (Large, Medium, and Small), we save all valid rectangles in a respective vector $\mathcal{V}_{\text{WallLarge}}$, $\mathcal{V}_{\text{WallMedium}}$ and $\mathcal{V}_{\text{WallSmall}}$, where always four consecutive elements, starting at element one, belong to one rectangle (e.g. see Equation 3.27).

$$\begin{aligned}
\mathcal{V}_{\text{WallLarge}} = \{ &\mathbf{p}1_{\text{Corner1}}, \mathbf{p}1_{\text{Corner2}}, \mathbf{p}1_{\text{Corner3}}, \mathbf{p}1_{\text{Corner4}}, \\
&\mathbf{p}2_{\text{Corner1}}, \mathbf{p}2_{\text{Corner2}}, \mathbf{p}2_{\text{Corner3}}, \mathbf{p}2_{\text{Corner4}}, \dots \}
\end{aligned} \tag{3.27}$$

Algorithm 2: Detect all suitable wall areas for a defined boundary size

```
Input: $w_{\text{min}}, w_{\text{max}}$, Boundary Size
Output: $\mathcal{V}_{\text{Wall<BoundarySize>}}$
foreach $v_{x_1,y_1,z_1} \mid f_{x_1,y_1,z_1} \neq 0$ do
    foreach $v_{x_2,y_2,z_2} \mid d \in [w_{\text{min}}, w_{\text{max}}] \land f_{x_2,y_2,z_2}$ valid normal do
        foreach $x, y, z \mid x, y, z \in \mathbb{N}_0 \land y \in [y_1 - h, y_1] \land$
        $y \in [\mathcal{I}_{\text{Floor}}, \mathcal{I}_{\text{Ceiling}}] \land x \in [x_1, x_2] \land z \in [z_{\text{Lower}}, z_{\text{Upper}}]$ do
            if $a_{xz} = 1$ then
                Remember that the rectangle is not suitable;
            else if $f_{xyz} \neq 0$ then
                $c_{\text{suitable}} = c_{\text{suitable}} + 1$;
        end
        if Rectangle valid $\land c_{\text{suitable}} \geq 0.8 \times (d + 1) \times (h + 1)$ then
            $\mathcal{V}_{\text{Wall<BoundarySize>}}.\text{Append}(\mathbf{p}_{\text{Corner1}}, \mathbf{p}_{\text{Corner2}}, \mathbf{p}_{\text{Corner3}}, \mathbf{p}_{\text{Corner4}})$;
        end
    end
end
```

In Algorithm 2, the above described method is represented in a more compact form. A valid normal for f_{x_2,y_2,z_2} is, as previously already mentioned, if f_{x_2,y_2,z_2} contains a normal from f_{x_1,y_1,z_1} or one, which is an adjacent normal to one in f_{x_1,y_1,z_1}. The algorithm does provide a few options, where the most inner for loop can be terminated early, in case of an invalid rectangle. The bounds for v_{x_1,y_1,z_1} and v_{x_2,y_2,z_2} are defined through the Functions 3.21 in the beginning of this section. At the end, the four corner elements are appended to the vector $\mathcal{V}_{\text{Wall<BoundarySize>}}$, which is the output of the method. The boundary size is a flag, which will be explained in Section 3.5 and is used to represent the size classification of the rectangle.

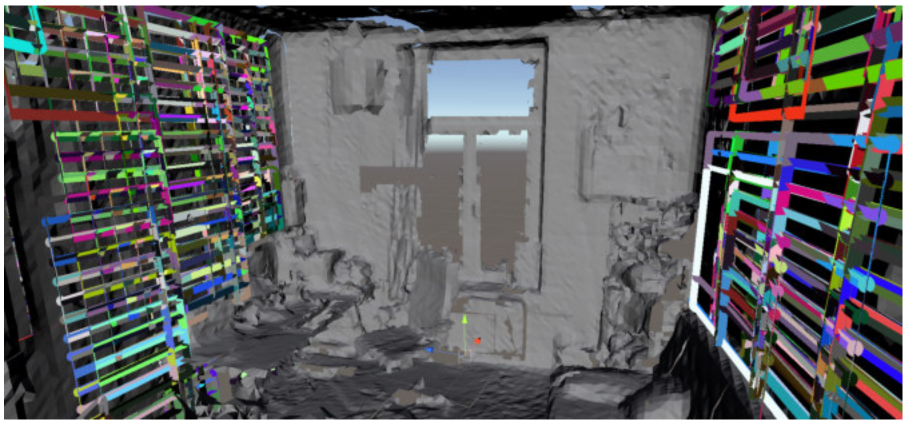


Figure 3.8.: The marked rectangles show a few samples of valid wall areas for interactive content on the left and right walls of the room. Different sizes of rectangles can be recognized, but it becomes clear that many areas are overlapping or are only minimally different. This is the reason, why we decided to reduce the amount of valid areas, which are returned from our algorithm (see Section 3.4). Image is a screenshot taken within the Unity Editor.

After running the above algorithm, the outcome can look like presented in Figure 3.8, where highlighted rectangles are shown for the left and right walls. Different sizes for the rectangles can be clearly seen, especially on the right side, but this makes it apparent for the need to reduce the amount of valid areas returned for the room. Although the above sample only shows the results for the left and right walls, it contains already many hundreds of recognized areas, which are often overlapping other valid tiles. In Section 3.4, we are talking about the reduction of the detected areas.

3.3.2. Suitable Floor Areas

For horizontal surface areas parallel to the X-Z-plane similar algorithm are going to be used, varying slightly for floors, ceilings and furniture. All reusable methods will be explained in this section on floors, but will be reused and referenced in the following two sections.

Similar to the wall rectangles, our flat surfaces come in three sizes: 2×2 m^2 (Large), 1×1 m^2 (Medium), and 0.5×0.5 m^2 (Small). Since placed objects on a floor tend to be more quadratic, compared to interactive windows placed on walls, a quadratic dimension is chosen for these areas. The range of the width w for the predefined areas are again established with w_{min} and w_{max} for the lower and upper bounds ($w \pm 0.1$).

In general, all voxels located on X-Z-planes have to be cycled at a certain Y-level, which is for the floor at $y_1 = \mathcal{I}_{\text{Floor}} + 1$. For each voxel v_{x_1,y_1,z_1}, we now have to check if it is a floor tile ($d_{x_1,z_1} = 1$). For the second voxel v_{x_2,y_2,z_2}, the validity of the floor tile is analysed, for $d \in [w_{\min}, w_{\max}]$ and the previously in Equation 3.21 defined X and Z-coordinates. Since we are creating squares parallel to the X-Z-plane, a little different approach is needed compared to the used method for creating the rectangles placed on the walls.

To create squares, four line segments are needed, containing two different slopes (for squares the opposite sites are always parallel). The first slope and linear equation can be calculated the same to Function 3.22, since the first line segment is located between voxel v_{x_1,y_1,z_1} and v_{x_2,y_2,z_2} likewise. The other parallel line segment is established by the remaining two corners of the square and are obtained the following way:

$$\begin{aligned} &v_{x_3,y_3,z_3} \mid x_3 = x_2 - (z_2 - z_1) \wedge y_3 = y_1 \wedge z_3 = z_2 + x_2 - x_1 \\ &v_{x_4,y_4,z_4} \mid x_4 = x_1 - (z_2 - z_1) \wedge y_4 = y_1 \wedge z_4 = z_1 + x_2 - x_1 \ . \end{aligned} \tag{3.28}$$

The coordinates of voxel v_{x_3,y_3,z_3} and v_{x_4,y_4,z_4} are obtained through the perpendicular slope of the original two voxels. By looking at Figure 3.9, where the bottom two red voxels represent v_{x_1,y_1,z_1} and v_{x_2,y_2,z_2}, the process to obtain voxel v_{x_3,y_3,z_3} (red voxel top right) can be reproduced. The X-coordinate is established by subtracting the numerator of the slope fraction (Equation 3.22) from the x_2 coordinate, which in our example has the outcome $x_3 = x_2 - 1$. For the Z-coordinates, the denominator of the slope fraction has to be added, as the z_3-coordinates lie above z_2, pointing always towards a positive direction. For our example, this gives us $z_3 = z_2 + 6$. As we are operating on the X-Z-plane, the y_3 coordinate stays on the same level as the previous Y-coordinates. For the last voxel v_{x_4,y_4,z_4}, we do the same operations with one discrepancy. The starting voxel changes from v_{x_2,y_2,z_2} to v_{x_1,y_1,z_1}, as v_{x_4,y_4,z_4} is located above the first voxel.

For the line segments $\overline{v_{x_1,y_1,z_1} v_{x_4,y_4,z_4}}$ and $\overline{v_{x_2,y_2,z_2} v_{x_3,y_3,z_3}}$, the perpendicular slope $m_\perp$ needs to be calculated, where the linear equation $h_\perp(x)$ is undefined for $z_1 = z_2$. The perpendicular deviation $m_\perp$ to any given linear slope is obtained by switching the numerator and denominator of the gradient in function $h(x)$ defined in 3.22 and multiplying the fraction by negative one:

$$\begin{aligned} h_\perp(x) &= m_\perp(x - x_1) + z_1 \\ m_\perp &= -1 \times \frac{x_2 - x_1}{z_2 - z_1} \ . \end{aligned} \tag{3.29}$$

As there is a need to cycle through all possible X-Z-combinations, which are included inside the square, we are also defining X_{Start}, X_{End}, Z_{Start}, and Z_{End} for the maximum bounds of our square. They help manage the total number of voxels required for analysation on validity, since these bounds always enclose the suitable square seen in

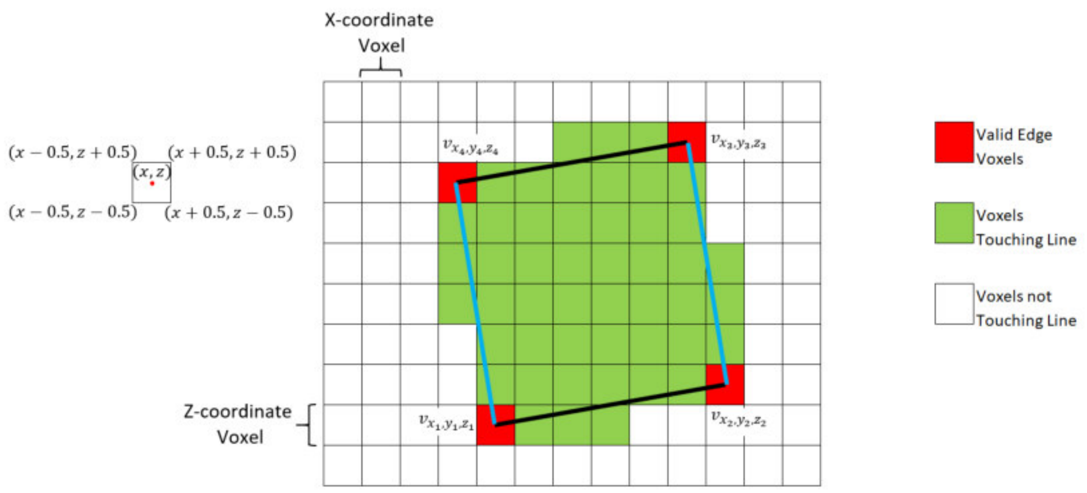


Figure 3.9.: An exemplary top-down view of the X-Z-plane for a valid horizontal square, where the four red voxels are the valid corner points v_{x_1,y_1,z_1}, v_{x_2,y_2,z_2}, v_{x_3,y_3,z_3}, and v_{x_4,y_4,z_4}. The four line segments connecting the four corners are the boundaries of the suitable square and all voxels, which are located at least partly within those boundaries, are checked for validity (green voxels). The black lines are described by the linear equation $h(x)$ from Equation 3.22, while the blue lines are described by $h_\perp(x)$ from Equation 3.29. White voxels are of no interest to this square, as they lie outside the suitable surface. The centre of each square represents the (x, z) coordinate of the voxel and the corners are always 0.5 units away from the centre (see clarification on the left for the coordinates and values for each corner and centre of the voxels). Image was created with the help of Microsoft Excel.

Figure 3.9. These are obtained through the following methods:

$$\begin{aligned}
X_{\text{Start}} &= \begin{cases} x_1, & \text{if } m < 0 \\ x_4, & \text{otherwise} \end{cases} \\
X_{\text{End}} &= \begin{cases} x_3, & \text{if } m < 0 \\ x_2, & \text{otherwise} \end{cases} \\
Z_{\text{Start}} &= \begin{cases} z_2, & \text{if } m < 0 \\ z_1, & \text{otherwise} \end{cases} \\
Z_{\text{End}} &= \begin{cases} z_4, & \text{if } m < 0 \\ z_3, & \text{otherwise} \end{cases} .
\end{aligned} \tag{3.30}$$

All these values in Equation 3.30 are easily obtained by imagining which voxels are located at each X and Z-coordinate's maximum and minimum, which depends on the slope. For the example in Figure 3.9 with the positive slope, it can be clearly seen that the smallest X-value for the square is x_4, while the largest is x_2. Same goes for the Z-values. If the slope would now be negative, the bottom left voxel becomes the smallest X-value, while the top right voxel becomes the biggest.

Of course, we also have to check our two new corner points for validity by examining if they are within the bounds of the voxel representation and if they are valid floor tiles. If this is the case, all voxels within the bound square are cycled and inspected if it is located within the bounds of the valid area square. This process is similar to the introduced method in Equation 3.24 with a slight adjustment. Since each individual voxel is examined to be within the bounds of the square, only checking if the current Z-coordinate is within the range of the square for the current X-coordinate is necessary. First off, x_{Lower} and x_{Upper} are calculate as in Equation 3.23, with the following adjustment to the range of x:

$$x \in \mathbb{N}_0 \land X_{\text{Start}} \leq x \leq X_{\text{End}} \ . \tag{3.31}$$

The Z-bounds for our four line segments are determined next. For keeping the following formulas short and reader friendly, we replace our four corner voxels with the variables $v_1 = v_{x_1,y_1,z_1}$, $v_2 = v_{x_2,y_2,z_2}$, $v_3 = v_{x_3,y_3,z_3}$, and $v_4 = v_{x_4,y_4,z_4}$. Additionally, two new linear functions $h_4(x)$ and $h_{\perp_2}(x)$ are introduced. These are similar to Equations 3.22 and 3.29, but replace the predefined point on the line with voxel v_4 and v_2 respectively. This results in the following bounds for the Z-coordinates for each line segment:

$$\begin{aligned}
z_{\overline{v_1v_2}\text{Lower}} &= \begin{cases} x_1, & \text{if } m \text{ undefined} \\ h(x_{\text{Lower}}), & \text{otherwise} \end{cases} \\
z_{\overline{v_1v_2}\text{Upper}} &= \begin{cases} x_1, & \text{if } m \text{ undefined} \\ h(x_{\text{Upper}}), & \text{otherwise} \end{cases} \\
z_{\overline{v_3v_4}\text{Lower}} &= \begin{cases} x_4, & \text{if } m \text{ undefined} \\ h_4(x_{\text{Lower}}), & \text{otherwise} \end{cases} \\
z_{\overline{v_3v_4}\text{Upper}} &= \begin{cases} x_4, & \text{if } m \text{ undefined} \\ h_4(x_{\text{Upper}}), & \text{otherwise} \end{cases} \\
h_4(x) &= m(x - x_4) + z_4 \ .
\end{aligned} \tag{3.32}$$

$$\begin{aligned}
z_{\overline{v_1v_4}\text{Lower}} &= \begin{cases} x_1, & \text{if } m_\perp \text{ undefined} \\ h_\perp(x_{\text{Lower}}), & \text{otherwise} \end{cases} \\
z_{\overline{v_1v_4}\text{Upper}} &= \begin{cases} x_1, & \text{if } m_\perp \text{ undefined} \\ h_\perp(x_{\text{Upper}}), & \text{otherwise} \end{cases} \\
z_{\overline{v_2v_3}\text{Lower}} &= \begin{cases} x_2, & \text{if } m_\perp \text{ undefined} \\ h_{\perp_2}(x_{\text{Lower}}), & \text{otherwise} \end{cases} \\
z_{\overline{v_2v_3}\text{Upper}} &= \begin{cases} x_2, & \text{if } m_\perp \text{ undefined} \\ h_{\perp_2}(x_{\text{Upper}}), & \text{otherwise} \end{cases} \\
h_{\perp_2}(x) &= m_\perp(x - x_2) + z_2 \ .
\end{aligned} \tag{3.33}$$

This delivers all bounds for the defined Z-coordinates. There is also a reason, why x values are assigned to the Z-bounds if a slope is undefined, which will become more apparent when these special cases are handled. Equations 3.36 and 3.37 explain in detail, what checks are done for these undefined slopes and why the X-coordinate is needed. The first two cases are without any undefined slopes. The first one checks the bounds for a negative slope m:

$$\begin{aligned}
&(z_{\overline{v_1v_2}\text{Lower}} < z + 0.5 \lor z_{\overline{v_1v_2}\text{Upper}} < z + 0.5) \land \\
&(z_{\overline{v_3v_4}\text{Lower}} > z - 0.5 \lor z_{\overline{v_3v_4}\text{Upper}} > z - 0.5) \land \\
&(z_{\overline{v_1v_4}\text{Lower}} > z - 0.5 \lor z_{\overline{v_1v_4}\text{Upper}} > z - 0.5) \land \\
&(z_{\overline{v_2v_3}\text{Lower}} < z + 0.5 \lor z_{\overline{v_2v_3}\text{Upper}} < z + 0.5) \ .
\end{aligned} \tag{3.34}$$

If Equation 3.34 is true, then the voxel defined through x and z is within the square. The checks in this equation basically examine, if the voxel boundaries are above $\overline{v_1v_2}$ and below $\overline{v_3v_4}$ (bottom and top black line in Figure 3.9). The perpendicular segments (blue lines in Figure 3.9) are also analysed. Here, $\overline{v_1v_4}$ needs to be above (in terms of Z-coordinate) the voxel, while the segment $\overline{v_2v_3}$ is located below. The next case determines for a positive slope m, if the point is within the square:

$$\begin{aligned}
&(z_{\overline{v_1v_2}\text{Lower}} < z + 0.5 \lor z_{\overline{v_1v_2}\text{Upper}} < z + 0.5) \land \\
&(z_{\overline{v_3v_4}\text{Lower}} > z - 0.5 \lor z_{\overline{v_3v_4}\text{Upper}} > z - 0.5) \land \\
&(z_{\overline{v_1v_4}\text{Lower}} < z + 0.5 \lor z_{\overline{v_1v_4}\text{Upper}} < z + 0.5) \land \\
&(z_{\overline{v_2v_3}\text{Lower}} > z - 0.5 \lor z_{\overline{v_2v_3}\text{Upper}} > z - 0.5) \ .
\end{aligned} \tag{3.35}$$

In Equation 3.35 only the relation of the perpendicular segments change. Through the positive slope, segment $\overline{v_1v_4}$ is now the lower boundary, while $\overline{v_2v_3}$ needs to be above (in terms of Z-coordinate) the valid voxel. The first special case appears when the slope $m = 0$. This results in undefined slopes for the perpendicular segments. For this case, we determine if the voxel resides within the bounds of the square through the following method:

$$\begin{aligned}(z_{\overline{v_1v_2}\text{Lower}} < z + 0.5 \vee z_{\overline{v_1v_2}\text{Upper}} < z + 0.5)\wedge\\(z_{\overline{v_3v_4}\text{Lower}} > z - 0.5 \vee z_{\overline{v_3v_4}\text{Upper}} > z - 0.5)\wedge\\(z_{\overline{v_1v_4}\text{Lower}} < x + 0.5 \vee z_{\overline{v_1v_4}\text{Upper}} < x + 0.5)\wedge\\(z_{\overline{v_2v_3}\text{Lower}} > x - 0.5 \vee z_{\overline{v_2v_3}\text{Upper}} > x - 0.5)\ .\end{aligned} \tag{3.36}$$

As $m_\perp$ is undefined, we can no longer check if it is above or below the Z-coordinate of segments $\overline{v_1v_4}$ and $\overline{v_1v_4}$. However, as seen in Equations 3.32 and 3.33, x_1 is assigned to the left blue line and x_2 to the right. With Equation 3.36, we can now evaluate the current X-coordinate of the voxel against those values, establishing if it is on the left side of the right boundary and vice versa. Through this smart assignment of an X-value, it can now compare it against the X-coordinate instead of a Z-value in the event of an undefined slope $m_\perp$. We are using the same technique for our fourth and last case, which is defined as follows:

$$\begin{aligned}(z_{\overline{v_1v_2}\text{Lower}} > x - 0.5 \vee z_{\overline{v_1v_2}\text{Upper}} > x - 0.5)\wedge\\(z_{\overline{v_3v_4}\text{Lower}} < x + 0.5 \vee z_{\overline{v_3v_4}\text{Upper}} < x + 0.5)\wedge\\(z_{\overline{v_1v_4}\text{Lower}} < z + 0.5 \vee z_{\overline{v_1v_4}\text{Upper}} < z + 0.5)\wedge\\(z_{\overline{v_2v_3}\text{Lower}} > z - 0.5 \vee z_{\overline{v_2v_3}\text{Upper}} > z - 0.5)\ .\end{aligned} \tag{3.37}$$

This case covers the special event of an undefined slope m and results in horizontal black lines for the square. While segments $\overline{v_1v_4}$ and $\overline{v_2v_3}$ experience the same case as in Equation 3.35, segments $\overline{v_1v_2}$ and $\overline{v_3v_4}$, are now evaluated to their location to the right and left of the current voxel respectively. This is achieved through the assignment of x_1 and x_4 in Equations 3.32 and 3.33.

If the current voxel is within the created square, we establish if it is a wall tile or a suitable floor tile. If it is a wall tile, the square is deemed as unsuitable and is discarded. For the whole square, the amount of suitable floor tiles is counted in c_{suitable}. Similar to Equation 3.25, the square is saved into the vectors $\mathcal{V}_{\text{FloorLarge}}$, $\mathcal{V}_{\text{FloorMedium}}$ and $\mathcal{V}_{\text{FloorSmall}}$ respectively if it is above the threshold of 95 percent.

$$c_{\text{suitable}} \geq 0.95 \times (d+1)^2. \tag{3.38}$$

Through testing, we found that around 95 percent of the square needed to be valid for the best results. As the above vectors are defined in the same fashion as the wall vectors in Equation 3.27, only the four corners of the square have to be defined anew:

$$\begin{aligned}\mathbf{p}_{\text{Corner1}} &= (x_1, y_1, z_1)\\\mathbf{p}_{\text{Corner2}} &= (x_2, y_1, z_2)\\\mathbf{p}_{\text{Corner3}} &= (x_3, y_1, z_3)\\\mathbf{p}_{\text{Corner4}} &= (x_4, y_1, z_4)\ .\end{aligned} \tag{3.39}$$

Algorithm 3: Detect all suitable floor areas for a defined boundary size

```
Input: $w_{\text{min}}, w_{\text{max}}$, Boundary Size
Output: $\mathcal{V}_{\text{Floor<BoundarySize>}}$
foreach $v_{x_1,y_1,z_1} \mid d_{x_1,z_1} = 1 \wedge y_1 = \mathcal{I}_{\text{Floor}} + 1$ do
    foreach $v_{x_2,y_1,z_2} \mid d \in [w_{\text{min}}, w_{\text{max}}] \wedge d_{x_2,z_2} = 1$ do
        foreach
          $x, z \mid x, z \in \mathbb{N}_0 \wedge x \in [X_{\text{Start}}, X_{\text{End}}] \wedge z \in [Z_{\text{Start}}, Z_{\text{End}}] \wedge d_{x_3,z_3} = d_{x_4,z_4} = 1$
         do
            if $v_{x,y_1,z}$ within square then
                if $a_{xz} = 1$ then
                    Remember that the square is not suitable;
                else if $d_{xz} \neq 0$ then
                    $c_{\text{suitable}} = c_{\text{suitable}} + 1$;
            end
        end
        if Square valid $\wedge$ $c_{\text{suitable}} \geq 0.95 \times (d+1)^2$ then
            $\mathcal{V}_{\text{Floor<BoundarySize>}}$.Append($\mathbf{p}_{\text{Corner1}}, \mathbf{p}_{\text{Corner2}}, \mathbf{p}_{\text{Corner3}}, \mathbf{p}_{\text{Corner4}}$);
        end
    end
end
```

With Algorithm 3, we describe the high level functionality of our above described process to define suitable floor squares. This algorithm does share some similarities with Algorithm 2 for calculating the suitable wall rectangles. This fact can be used to share functionalities in the implementation and reuse code passages.

In Figure 3.10, the suitable floor areas are added to the previous obtained outcomes for valid wall rectangles. These defined squares cover all areas on the floor, which are large enough to accommodate a square. However, our algorithm does generate a lot of redundant areas, which are sometimes exactly the same, only with a different orientation. Other areas have large overlapping chunks. This issue will be addressed in Section 3.4, where we reduce our valid results and only return unique squares with minimal overlap.

3.3.3. Suitable Ceiling Areas

The definition of suitable areas for the ceiling largely overlap with the previous section on suitable squares for floors. In fact, all equations, values and definitions used for the algorithm to define the squares for the floor can be reused for ceilings almost unchanged. The only details that need adjusting are the definitions of y_1 and d_{xz}. For ceilings, y_1 needs to be one level below the ceiling level, resulting in $y_1 = \mathcal{I}_{\text{Ceiling}} - 1$. Since we are now checking for suitable ceiling tiles instead of floor tiles, all occurrences of d_{xz} are changed to e_{xz}, determining if those suitable tiles are located on the ceiling.

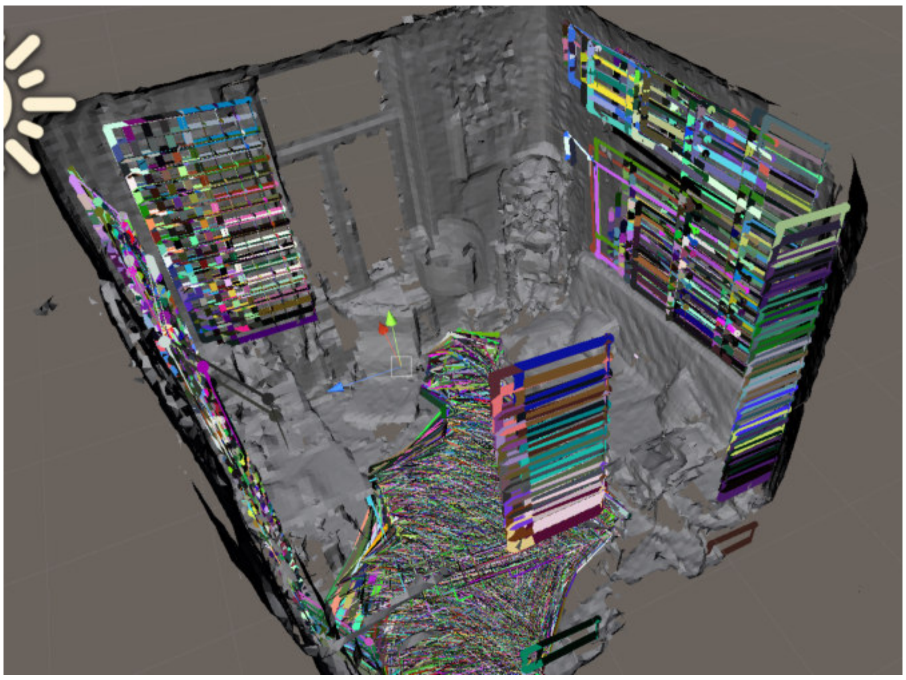


Figure 3.10.: The same sample room from Figure 3.8, now also with the results from Algorithm 3. The squares on the floor cover all suitable areas of the floor in different orientations. This will later be reduced in Section 3.4. Image is a screenshot taken within the Unity Editor.

The above adaptation of the algorithm for suitable floor areas is actually possible, because the ceiling is nothing else then an upside-down floor. Since internally a flag is set, to notify the algorithm to detect ceilings instead of floors, this also allows the assumption of other things about the detected areas, which will be discussed in detail in a later section. Since the three dimensions exist likewise for ceiling squares, the following three output vectors are used instead of the three floor arrays $\mathcal{V}_{\text{FloorLarge}}$, $\mathcal{V}_{\text{FloorMedium}}$ and $\mathcal{V}_{\text{FloorSmall}}$ respectively: $\mathcal{V}_{\text{CeilingLarge}}$, $\mathcal{V}_{\text{CeilingMedium}}$ and $\mathcal{V}_{\text{CeilingSmall}}$.

3.3.4. Suitable Furniture Areas

Compared to the previous section, the algorithm from Section 3.3.2 has to be adjusted on a few more levels, mainly due to suitable furniture areas not only appearing on one Y-level, but rather on all levels between $\mathcal{I}_{\text{Floor}}$ exclusive and $\mathcal{I}_{y\text{End}}$ inclusive. $\mathcal{I}_{y\text{End}}$ is obtained from Equation 3.13 and y_1 is now defined as follows:

$$y_1 \in (\mathcal{I}_{\text{Floor}}, \mathcal{I}_{y\text{End}}] \land y \in \mathbb{N}_0. \tag{3.40}$$

Additionally, we examine if the current voxel or any other voxel within a created square is valid in a more complex way, to ensure better and larger result areas. Since furniture surfaces are often cluttered with objects, these surfaces may be uneven and could produce many solitary cubes in our voxelized representation, which would make flat and even surfaces a rare occurrence on top of furniture. With this in mind, we actually determine if every point (besides the first corner, where we just check for $c_{x_1,z_1} = y_1$) is within the square, if it is at the current y_1 level or one above ($c_{xz} = y_1 \vee c_{xz} = y_1 + 1$). Equation 3.13 provides the definition for c_{xz}.

We also have to redefine the four corner points for our algorithm, since the occurrence of the highest Y-level should be reflected in the final position, reducing overlapping of the square with room structures.

$$\begin{aligned}
\mathbf{p}_{\text{Corner1}} &= (x_1, y_{\text{Actual}}, z_1) \\
\mathbf{p}_{\text{Corner2}} &= (x_2, y_{\text{Actual}}, z_2) \\
\mathbf{p}_{\text{Corner3}} &= (x_3, y_{\text{Actual}}, z_3) \\
\mathbf{p}_{\text{Corner4}} &= (x_4, y_{\text{Actual}}, z_4) \\
y_{\text{Actual}} &= \begin{cases} y_1 + 1, & \text{if } \exists c_{xz} \mid c_{xz} = y_1 + 1 \\ y_1, & \text{otherwise} \end{cases} .
\end{aligned} \tag{3.41}$$

These points are now saved within their respective vectors, which are $\mathcal{V}_{\text{FurnitureLarge}}$, $\mathcal{V}_{\text{FurnitureMedium}}$ and $\mathcal{V}_{\text{FurnitureSmall}}$. In Figure 3.11, the outcome from the furniture algorithm is added to the previously detected floor and wall areas. It reliably detects flat surfaces on sofas (left), beds (right), or tables (middle front and back). Moreover, the same redundancy seen with the floor outcomes is found and will be reduce in Section 3.4.

3.4. Result Reduction

With the suitable area detection done, we now have 12 vectors with results for four different surface features and three unique sizes. The next step will be the reduction of the results, where any outcomes are omitted in the first phase, which have the same centre point. This requires the calculation of the centre point, which is obtained by taking the mean of each coordinate:

$$\begin{aligned}
\mathbf{p}_{\text{Centre}} &= (x_{\text{Centre}}, y_{\text{Centre}}, z_{\text{Centre}}) \\
&= \left(\frac{x_1 + x_2 + x_3 + x_4}{4}, \frac{y_1 + y_2 + y_3 + y_4}{4}, \frac{z_1 + z_2 + z_3 + z_4}{4} \right) .
\end{aligned} \tag{3.42}$$

As Equation 3.42 can deliver non integer results, a conversion back to the voxelized representation is needed. Depending on the fraction part of the outcome, only certain voxels get marked as centres. The following equation demonstrates the calculation for the X-coordinate of $\mathbf{p}_{\text{Centre}}$, but can be used for all three axes by using y_{Centre} and z_{Centre}

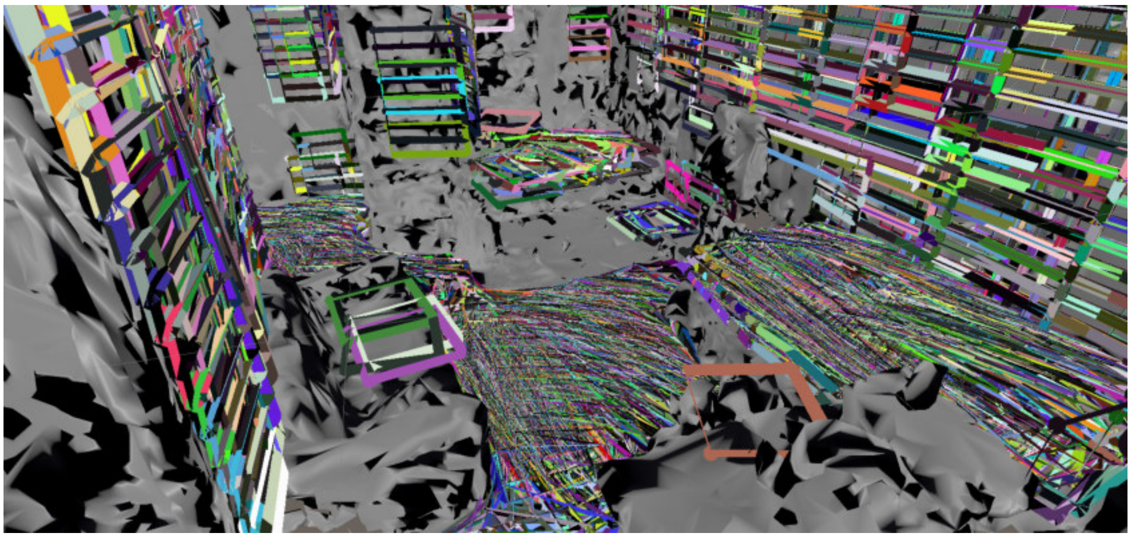


Figure 3.11.: A different sample room, showing the results for wall, floor, and now newly added furniture areas. The squares on the furniture cover all suitable areas, such as the sofa on the left, the bed on the right, or the table in the middle, in multiple orientations. This will later be reduced in Section 3.4. Image is a screenshot taken within the Unity Editor.

instead of x_{Centre} respectively:

$$x_{\text{Centre}} = \begin{cases} \{\lfloor x_{\text{Centre}} \rfloor, \lfloor x_{\text{Centre}} \rfloor + 1\}, & \text{if } x_{\text{Centre}} - \lfloor x_{\text{Centre}} \rfloor = 0.5 \\ \{\lfloor x_{\text{Centre}} \rfloor + 1\}, & \text{if } x_{\text{Centre}} - \lfloor x_{\text{Centre}} \rfloor > 0.5 \\ \{\lfloor x_{\text{Centre}} \rfloor\}, & \text{otherwise} \end{cases} . \tag{3.43}$$

For all marked centre points $\mathbf{p}_{\text{Centre}}$, it is established if any voxels are already marked as centres for another suitable area. If this is not the case, we mark those centres and add the four corners $\mathbf{p}_{\text{Corner1}}$, $\mathbf{p}_{\text{Corner2}}$, $\mathbf{p}_{\text{Corner3}}$, and $\mathbf{p}_{\text{Corner4}}$ to a single vector of suitable areas $\mathcal{V}_{\text{Wall}}$, $\mathcal{V}_{\text{Floor}}$, $\mathcal{V}_{\text{Ceiling}}$, and $\mathcal{V}_{\text{Furniture}}$ for each respective feature. For these vectors, the addition order of suitable areas is important, since it allows for an optimized next reduction phase. First all wall areas are added, then the floors, ceilings and at last the furniture to their respective vector. For each collection of areas, the large areas are validated and included, getting priority and being not omitted, because a smaller area has already claimed that centre point. The medium areas come next and are then followed by the small areas:

$$\begin{aligned} \mathcal{V}_{\text{Wall}} &\subseteq \{\mathcal{V}_{\text{WallLarge}}, \mathcal{V}_{\text{WallMedium}}, \mathcal{V}_{\text{WallSmall}}\} \\ \mathcal{V}_{\text{Floor}} &\subseteq \{\mathcal{V}_{\text{FloorLarge}}, \mathcal{V}_{\text{FloorMedium}}, \mathcal{V}_{\text{FloorSmall}}\} \\ \mathcal{V}_{\text{Ceiling}} &\subseteq \{\mathcal{V}_{\text{CeilingLarge}}, \mathcal{V}_{\text{CeilingMedium}}, \mathcal{V}_{\text{CeilingSmall}}\} \\ \mathcal{V}_{\text{Furniture}} &\subseteq \{\mathcal{V}_{\text{FurnitureLarge}}, \mathcal{V}_{\text{FurnitureMedium}}, \mathcal{V}_{\text{FurnitureSmall}}\} \end{aligned} \tag{3.44}$$

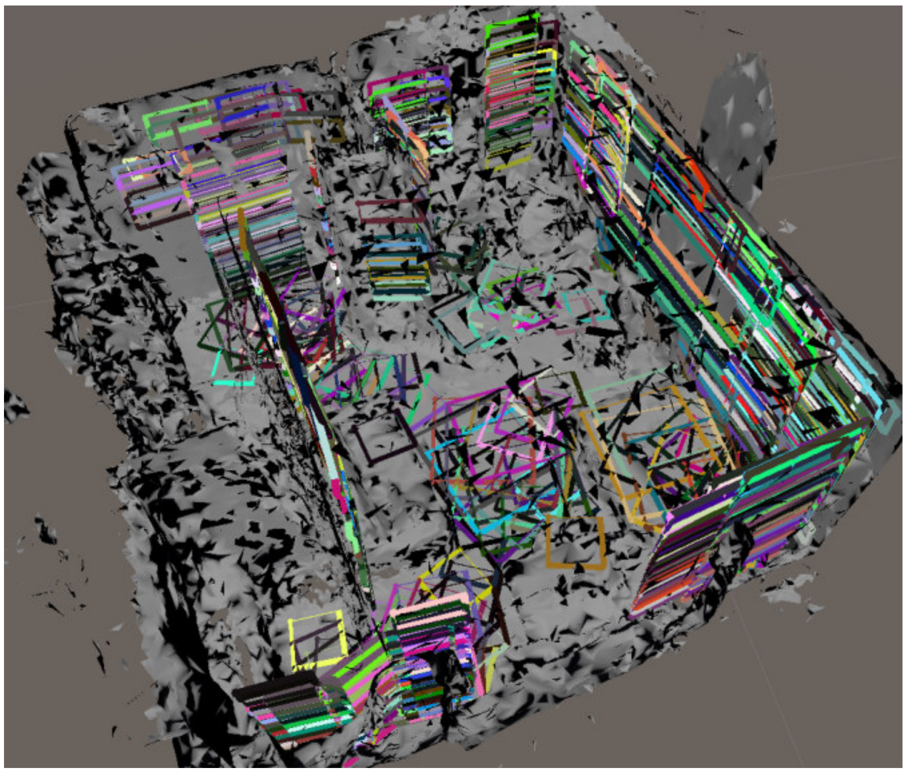


Figure 3.12.: The above described reduction results in the shown suitable areas remaining. Compared to the not reduced results in Figure 3.11, we could omit large numbers of suitable areas, especially for horizontal surfaces. However, there are still squares and rectangles remaining, which overlap with their neighbours. Image is a screenshot taken within the Unity Editor.

In Figure 3.12,the results of our first reduction step are showcased. While it does reduce large numbers of suitable areas, especially for horizontal surfaces, (compared to Figure 3.11), still many areas remain, which are overlapping with their neighbours. To reduce those rectangles and squares, we propose the algorithm described below.

The next step can be done asynchronously for each feature, deleting all elements, which are overlapping with other detected areas. Since each feature has its own area it affects, these do not interfere with each other. There are again major differences, between vertical and horizontal areas, which are split over the following two sections. The final result will be saved in our results vector $\mathcal{V}_{\text{Results}}$. After all overlapping elements have been deleted, the number of entries n that were written into the results vector are remembered. The order we add the suitable areas for each feature is the same as above.

3.4.1. Reduce Wall Recognitions

To reduce the amount of wall recognitions, all elements in vector $\mathcal{V}_{\text{Wall}}$ are cycled and saved, if they do not occupy another suitable area in vector $\mathcal{V}_{\text{WallFinal}}$ by not overlapping with already cleared elements. For walls, it is not as easy to check if surfaces are intersecting each other, or are close in front of another surface, compared to horizontal surfaces where all areas are located on the same plane.

The simplest feature to check for overlapping, is the Y-coordinate of the suitable wall areas. For each subsequent four corners in $\mathcal{V}_{\text{Wall}}$, we check if their Y-coordinate is smaller than all Y-coordinates of the first corner of each area in $\mathcal{V}_{\text{WallFinal}}$ and larger or equal to the third corner of each area in $\mathcal{V}_{\text{WallFinal}}$:

$$\begin{aligned} \mathbf{p} &= (x_{\mathbf{p}}, y_{\mathbf{p}}, z_{\mathbf{p}}) \land \mathbf{p} \in \{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4\} \subseteq \mathcal{V}_{\text{Wall}} \\ \{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4\} &:= \text{four corners of suitable area} \\ \mathbf{q} &= (x_{\mathbf{q}}, y_{\mathbf{q}}, z_{\mathbf{q}}) \land \mathbf{q} \in \{\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{q}_4\} \subseteq \mathcal{V}_{\text{WallFinal}} \\ \{\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{q}_4\} &:= \text{four corners of suitable area} \\ y_{\mathbf{q}_3} \le y_{\mathbf{p}} &< y_{\mathbf{q}_1} \mid \forall y_{\mathbf{p}} \in \mathbf{p} \land \forall y_{\mathbf{q}_1}, y_{\mathbf{q}_3} \in \mathcal{V}_{\text{WallFinal}} \ . \end{aligned} \tag{3.45}$$

For the overlap check on the X-coordinate, an offset factor of two is added to the area, as overlapping walls do not always align perfectly. This factor reduced overlaying walls with great accuracy, while also not excluding walls without overlaps four our tested sample rooms. For X-coordinates, the following comparison is proposed, since rectangles are always created in such a fashion, that x_1 is smaller or equal to x_2:

$$x_{\mathbf{q}_1} - 2 < x_{\mathbf{p}} < x_{\mathbf{q}_2} + 2 \mid \forall x_{\mathbf{p}} \land \forall x_{\mathbf{q}_1}, x_{\mathbf{q}_2} \in \mathcal{V}_{\text{WallFinal}} \ . \tag{3.46}$$

Due to the aforementioned, it is proven that $x_{\mathbf{p}}$ always needs to be larger than $x_{\mathbf{q}_1}$ and smaller than $x_{\mathbf{q}_2}$. Lastly, we have to check for the Z-coordinate. For walls the slope of the line segment has to be taken into account, as it will reveal, which Z-coordinate is larger:

$$\begin{cases} z_{\mathbf{q}_1} - 2 < z_{\mathbf{p}} < z_{\mathbf{q}_2} + 2, & \text{if } z_{\mathbf{q}_1} < z_{\mathbf{q}_2} \mid \forall z_{\mathbf{p}} \land \forall z_{\mathbf{q}_1}, z_{\mathbf{q}_2} \in \mathcal{V}_{\text{WallFinal}} \\ z_{\mathbf{q}_2} - 2 < z_{\mathbf{p}} < z_{\mathbf{q}_1} + 2, & \text{otherwise} \mid \forall z_{\mathbf{p}} \land \forall z_{\mathbf{q}_1}, z_{\mathbf{q}_2} \in \mathcal{V}_{\text{WallFinal}} \ . \end{cases} \tag{3.47}$$

If the three Equations 3.45, 3.46, and 3.47 are true, we have a valid surface and $\mathcal{V}_{\text{WallFinal}}$ will be extended by the corners $\mathbf{p}$. The next area in $\mathcal{V}_{\text{Wall}}$ is then checked against all rectangles the previous corners have been analysed against, plus the newly added rectangle, if it was valid. Through this, rectangles have to be compared against already valid areas, making sure that large areas are more likely to be valid, since they are processed earlier. When we compared all surfaces within $\mathcal{V}_{\text{Wall}}$, all elements in $\mathcal{V}_{\text{WallFinal}}$ are appended to $\mathcal{V}_{\text{Results}}$ and the number of entries for $\mathcal{V}_{\text{Results}}$ is remembered:

$$n_{\text{Wall}} = |\mathcal{V}_{\text{Results}}| \ \mid \mathcal{V}_{\text{Results}} = \{\mathcal{V}_{\text{WallFinal}}\} \ . \tag{3.48}$$

3.4.2. Reduce Horizontal Surface Recognitions

Next up are the reductions of horizontal surfaces. This method is exactly the same for all three features with horizontal surfaces (floors, ceilings, furniture). For furniture, one small addition can be found, which will be highlighted at the section it is used in. Otherwise, this work will explain this algorithm with the help of the floor features and note any differences for each other feature throughout this section.

This method creates a new vector $\mathcal{V}_{\text{FloorFinal}}$ for the final results and cycles through all squares located in $\mathcal{V}_{\text{Floor}}$. The points $\mathbf{p}$ and $\mathbf{q}$ are defined exactly the same as in Equation 3.45, with the variation of replacing $\mathcal{V}_{\text{Wall}}$ and $\mathcal{V}_{\text{WallFinal}}$ with $\mathcal{V}_{\text{Floor}}$ and $\mathcal{V}_{\text{FloorFinal}}$ respectively. For the ceiling and furniture, the vectors with the respective feature in their names get used instead of "Floor".

Before starting to check if a square is within the bounds of another one, one special case has to be handled first for the furniture areas. Since these can be on different height levels, we check if the Y-coordinate is within two voxels above or below the Y-level of an already valid square:

$$y_{\mathbf{q}_1} - 2 < y_{\mathbf{p}_1} < y_{\mathbf{q}_1} + 2 \mid \forall x_{\mathbf{p}} \land \forall y_{\mathbf{q}_1} \in \mathcal{V}_{\text{FurnitureFinal}} \ . \tag{3.49}$$

Next, the algorithm has to establish if a corner point of a square from $\mathcal{V}_{\text{Floor}}$ is within an area of $\mathcal{V}_{\text{FloorFinal}}$. Previously, the method made a complicated comparison for this to check if any part of a voxel is within a square (Equations 3.34 through 3.37). However, this time we only want to check if the centre of a corner voxel is within a square, which can be achieved through easier methods. The algorithm does this by comparing the length of the vector from corner A of the square to the point M to the projected dimensions of the vectors $\overrightarrow{\text{AB}}$ and $\overrightarrow{\text{AD}}$ respectively. For a visual example of this method, please check Figure 3.13. Now we only need to determine that the projected lengths are not smaller than zero (vector points into the opposite direction) or longer than the vectors $\overrightarrow{\text{AB}}$ or $\overrightarrow{\text{AD}}$ [Azad, 2019]. This gives the following comparison statement for our method:

$$(0 < \overrightarrow{\text{AM}} \cdot \overrightarrow{\text{AB}} < \overrightarrow{\text{AB}} \cdot \overrightarrow{\text{AB}}) \land (0 < \overrightarrow{\text{AM}} \cdot \overrightarrow{\text{AD}} < \overrightarrow{\text{AD}} \cdot \overrightarrow{\text{AD}}) \ . \tag{3.50}$$

In Equation 3.50, $\overrightarrow{\text{AB}}$ defines a vector and $\overrightarrow{\text{AM}} \cdot \overrightarrow{\text{AB}}$ describes the scalar product of both vectors, which are defined as follows:

$$\begin{aligned} \overrightarrow{\text{AB}} &= (x_\text{B} - x_\text{A}, y_\text{B} - y_\text{A}, z_\text{B} - z_\text{A}) \\ \overrightarrow{\text{A}} \cdot \overrightarrow{\text{B}} &= x_\text{A} \times x_\text{B} + y_\text{A} \times y_\text{B} + z_\text{A} \times z_\text{B}. \end{aligned} \tag{3.51}$$

With Equation 3.50, the algorithm can now establish if a corner point is within a valid rectangle. We determine this for each corner point of a square in $\mathcal{V}_{\text{Floor}}$ and also for the midpoint between two corner points. Through these two checks, the method can validate if large portions of a square were overlapping with a valid area. We define the

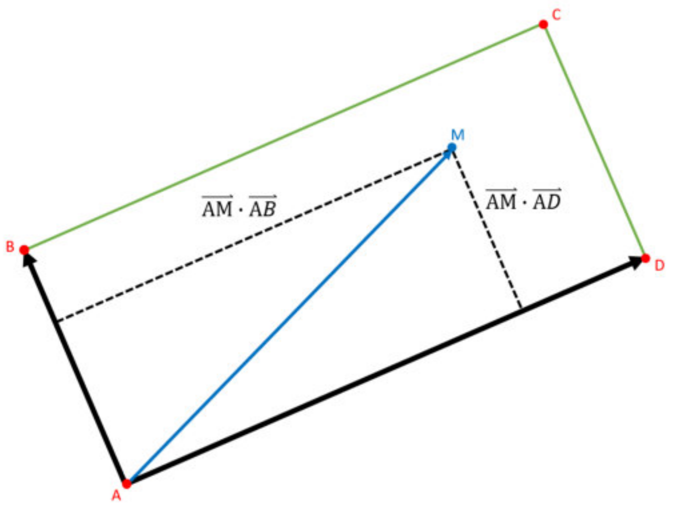

C
M
$\overrightarrow{AM} \cdot \overrightarrow{AB}$
$\overrightarrow{AM} \cdot \overrightarrow{AD}$
B
D
A

Figure 3.13.: A sample rectangle ABCD and a point M. Point M lies within this rectangle, if and only if $(0 < \overrightarrow{AM} \cdot \overrightarrow{AB} < \overrightarrow{AB} \cdot \overrightarrow{AB}) \wedge (0 < \overrightarrow{AM} \cdot \overrightarrow{AD} < \overrightarrow{AD} \cdot \overrightarrow{AD})$. As the quantity $\overrightarrow{AM} \cdot \overrightarrow{AB}$ expresses the length of the projection $\overrightarrow{AM}$ in the direction of $\overrightarrow{AB}$, we only need to check, if $\overrightarrow{AM} \cdot \overrightarrow{AB}$ is greater than 0 and smaller than $\overrightarrow{AB} \cdot \overrightarrow{AB}$. If this is fulfilled, we know for point M to be within the range of vector $\overrightarrow{AB}$. When this is also true for vector $\overrightarrow{AD}$, we can claim with certainty that M lies within the rectangle. Image was created with the help of Microsoft PowerPoint.

points A, B, C, and M as follows:

$$
\begin{aligned}
A &= \mathbf{q}_1 \mid \forall \mathbf{q} \in \mathcal{V}_{\text{FloorFinal}} \\
B &= \mathbf{q}_2 \mid \forall \mathbf{q} \in \mathcal{V}_{\text{FloorFinal}} \\
D &= \mathbf{q}_4 \mid \forall \mathbf{q} \in \mathcal{V}_{\text{FloorFinal}} \\
M &= \begin{cases} \mathbf{p}, & \text{if check for corner} \mid \forall \mathbf{p} \\ \left(\frac{x_{\mathbf{p}_L} + x_{\mathbf{p}_R}}{2}, \frac{y_{\mathbf{p}_L} + y_{\mathbf{p}_R}}{2}, \frac{z_{\mathbf{p}_L} + z_{\mathbf{p}_R}}{2}\right), & \text{if check for midpoint} \mid \mathbf{p}_L, \mathbf{p}_R \text{ are left and} \\ & \text{ right point of line segment of square} \in \mathcal{V}_{\text{Floor}} \end{cases} .
\end{aligned}
\tag{3.52}
$$

A square is valid, if no points M of that square are within an already suitable rectangle. If it is valid, we add the four corners to our vector $\mathcal{V}_{\text{FloorFinal}}$. As soon as all surfaces within $\mathcal{V}_{\text{Floor}}$ are compared and $\mathcal{V}_{\text{WallFinal}}$ was already added to $\mathcal{V}_{\text{Results}}$, this work adds the floor outcome to $\mathcal{V}_{\text{Results}}$ and remembers the number of entries afterwards. We repeat this for our other two features and get the following results:

$$\begin{aligned} n_{\text{Floor}} &= |\mathcal{V}_{\text{Results}}| \mid \mathcal{V}_{\text{Results}} = \{\mathcal{V}_{\text{WallFinal}}, \mathcal{V}_{\text{FloorFinal}}\} \\ n_{\text{Ceiling}} &= |\mathcal{V}_{\text{Results}}| \mid \mathcal{V}_{\text{Results}} = \{\mathcal{V}_{\text{WallFinal}}, \mathcal{V}_{\text{FloorFinal}}, \mathcal{V}_{\text{CeilingFinal}}\} \\ n_{\text{Furniture}} &= |\mathcal{V}_{\text{Results}}| \mid \mathcal{V}_{\text{Results}} = \{\mathcal{V}_{\text{WallFinal}}, \mathcal{V}_{\text{FloorFinal}}, \mathcal{V}_{\text{CeilingFinal}}, \mathcal{V}_{\text{FurnitureFinal}}\} . \end{aligned} \tag{3.53}$$

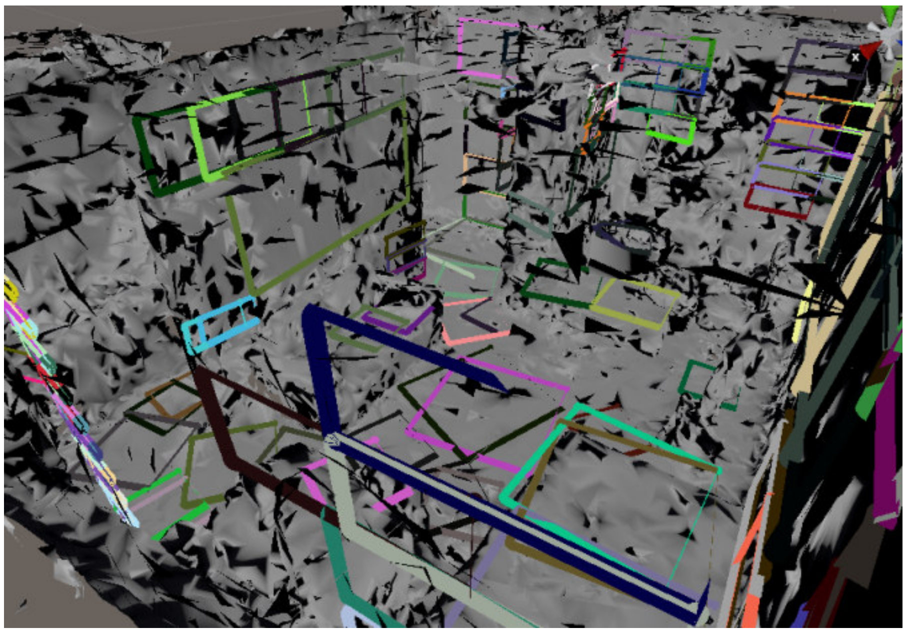


Figure 3.14.: This shows the further reduced results of our suitable surfaces after only checking the corner points of the squares. As there are still many overlapping squares for our flat surfaces, it becomes apparent, why we checked for the midpoints. The same goes for the extracted wall spaces and the integration of the bottom Y-coordinate into the checks (Equation 3.45). The final results are seen in Figure 3.15. Image is a screenshot taken within the Unity Editor.

Through the process of reduction, the algorithm took two extra steps, which might not be obvious. For the wall recognition, the proposed method defined $y_{\mathbf{q}_3} \leq y_{\mathbf{p}}$ in Equation 3.45, while in all other comparisons, we only used the less than or greater than operators. With this small adjustment, the comparison could get rid of overlapping wall recognitions, which can be still seen in Figure 3.14. When checking the bottom Y-level to be less than or equal to a valid surface bottom Y-level, areas are omitted, which are located on the same height and do not have a corner within a valid rectangle. Of course the algorithm could check for the mid points between the Y-levels, but this would need more calculations. Through this method, we only reduce areas, which are directly over a valid surface and not all areas which are just touching the boarders.

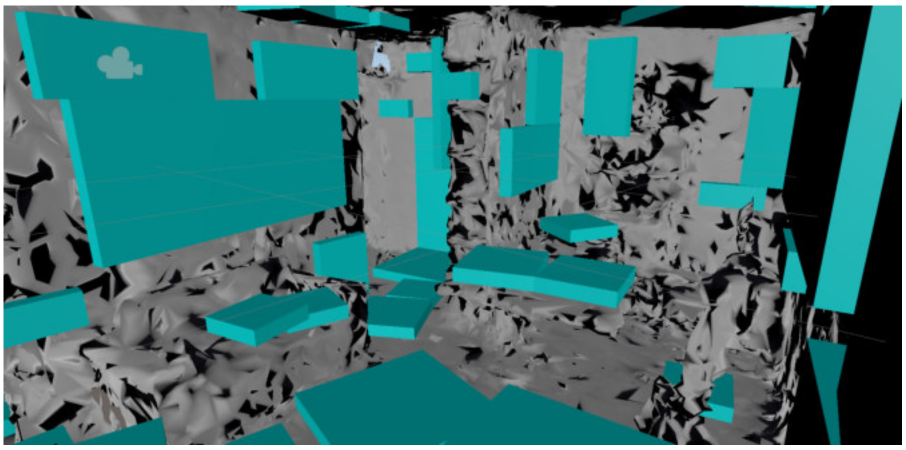


Figure 3.15.: The final outcome of our algorithm detecting and marking valid surfaces for interactive content. Our methods reduce all overlapping areas on walls and leave only minor overlaps for horizontal surfaces. However, these areas can be neglected as seen in this figure. All cyan coloured squares and rectangles are valid surfaces. For this room our algorithm detects 115 suitable areas. Image is a screenshot taken within the Unity Editor.

The same holds for the squares, always establishing if it is within a valid square and not on the boarders. If only taking the corner points of new squares into account, we would get the results seen in Figure 3.14. To additionally omit these overlapping squares, the midpoint is checked likewise. Of course, this still leaves a few shapes still overlapping, but only for minor areas, as seen in the final results displayed in Figure 3.15. Our algorithm can detect suitable areas over the whole room, independent of the surface feature, such as walls and ceilings, proven by the screenshot in Figure 3.15. However, we do have some unlabelled holes, which would be suitable for interactive content, but were not detected by the algorithm. An analysis of the limits, will be done in the next chapter.

3.5. Space Classification

With the detected areas all reduced to a non overlapping amount, we can now go ahead and classify our areas. The type classification is oriented after existing solutions from [Microsoft, 2018b] and [Microsoft, 2019c]. We chose to use four features for the type classification, giving us walls, floors, ceilings and furniture. These are the most important types to mark for the most suitable locations of interactive content. Besides the type, the algorithm classifies the size (big, medium, and small) and the level of interaction (close, distance, and no interaction). The levels of interaction gives a placement algorithm for the areas an understanding that a floor is located directly next to it and the user can directly interact with the content (close), the floor is further away and the user can only interact over a distance (distance), or the content cannot be easily interacted with (no interaction), which can be used for only visual representations as an example.

For each detected area, we are returning the following information from our plugin: centre point, surface normal, area height, area width, rotation, volume height and category. We will discuss each outcome information and how it was obtained in detail below.

The centre point was already calculated in Equation 3.42. The only thing that is left, before assigning the final result, is to convert our voxel format back to the original coordinate system with the following formulas and already defined variables from Equation 3.5:

$$\begin{aligned}\mathbf{v}_{\text{Original}} = \Bigg(& x_{\mathbf{p}_{\text{Voxel}}} \times S_{\text{Voxel}} + \mathbf{v}_{min_x} + \frac{S_{\text{Voxel}}}{2}, \\ & y_{\mathbf{p}_{\text{Voxel}}} \times S_{\text{Voxel}} + \mathbf{v}_{min_y} + \frac{S_{\text{Voxel}}}{2}, \\ & z_{\mathbf{p}_{\text{Voxel}}} \times S_{\text{Voxel}} + \mathbf{v}_{min_z} + \frac{S_{\text{Voxel}}}{2} \Bigg) .\end{aligned} \tag{3.54}$$

Through this conversion, the output gets the final centre point $\mathbf{v}_{\text{Centre}}$ in the metric of the original coordinate system. For the normal direction, the method has to do some more calculations, depending on the type of feature:

$$\begin{aligned}\mathbf{v}_{\text{Normal}} &= \begin{cases} g_{\text{Normal}}(), & \text{if surface is of type wall} \\ (0,-1,0), & \text{if surface is of type ceiling} \\ (0,1,0), & \text{otherwise} \end{cases} \\ g_{\text{Normal}}() &= \overrightarrow{\mathbf{p}_{\text{Corner}_2}\mathbf{p}_{\text{Corner}_3}} \times \overrightarrow{\mathbf{p}_{\text{Corner}_2}\mathbf{p}_{\text{Corner}_1}} \\ \overrightarrow{\text{A}} \times \overrightarrow{\text{B}} &= \left(y_{\text{A}} \times z_{\text{B}} - z_{\text{A}} \times y_{\text{B}}, z_{\text{A}} \times x_{\text{B}} - x_{\text{A}} \times z_{\text{B}}, x_{\text{A}} \times y_{\text{B}} - y_{\text{A}} \times x_{\text{B}}\right) .\end{aligned} \tag{3.55}$$

For walls, the algorithm has to calculate the normals with the cross product of the vector $\overrightarrow{\mathbf{p}_{\text{Corner}_2}\mathbf{p}_{\text{Corner}_3}}$ going from the second to the third corner of the rectangle and the vector $\overrightarrow{\mathbf{p}_{\text{Corner}_2}\mathbf{p}_{\text{Corner}_1}}$ going from the second to the first corner of the surface, as is stated by [Khronos, 2013]. The corner points are converted to the original coordinate system prior to this with the help of Equation 3.54 and the creation of the vectors is defined in Equation 3.51. For ceilings, we know that the surface is always parallel to the X-Z-plane and pointing downwards, which results in $(0,-1,0)$ and for the floor and furniture, we have the same vector, only pointing in the opposite direction. For the walls, the method has to do one more check, before the normal calculation is final. Currently the normal points towards the direction, where the side with the smaller X-coordinates is on the left, when looking from the top onto the rectangle. However, this is not always the correct direction for the normal, as the actual wall could be on either side of the rectangle. To check if the normal points into the correct direction, the algorithm checks all four directions of the first corner of the surface (we know through the rectangle creation process that a wall piece has to be present in at least one direction). For each detected wall piece, the algorithm establishes if the normal is pointing in the correct direction by evaluating the scalar product of the normal vector and the vector from the corner to the wall point. It verifies if the scalar product of the normal vector and the vector from the corner to the wall point is greater than zero. We are using the property of the scalar product that a positive result means both vectors point into a similar direction, 0 is a 90 degrees angle between both vectors, and a negative number indicates vectors pointing in opposite directions [Azad, 2019]. For our case, as soon as the scalar product is larger than 0, we know that the wall and normal orientation are both in the same direction, which indicates that we must switch the normal direction by multiplying the vector by negative one. Since the algorithm has some edge cases, where corner one of the rectangle was in a corner of the room, which also sometimes triggered the switch, the method is executed for any detected wall voxel next to the corner. This work could not produce a case, where this technique did not prevent the normal from pointing into the correct direction.

For the area width and height, this implementation measured the distance between the centre and right line segment and centre and upper line segment respectively, using the midpoint equation found in Equation 3.52. The midpoint is calculated between corner one and two for the upper bounds and the midpoint between corner two and three for the right bounds. The distances are then obtained with an adaption of the distance function found in Equation 3.20, which also includes the Y-coordinate, resulting in:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}\ . \tag{3.56}$$

The rotation around the Y-axis is saved in the rotation variable, measuring how much the square needs to turn so that the standard left line segment aligns with the actual left line segment. For walls this does not need to be done, as for them the standard left lines are always perpendicular to the X-Z-plane and are not experiencing any rotation around the normal. The standard left line segment is the directional vector $\overrightarrow{v_{\text{Standard}}} = (-1, 0, 0)$. In the next step, the algorithm detects the angle it needs to rotate this vector to match up with the vector $\overrightarrow{\mathbf{v}_{\text{Centre}}\mathbf{v}_{\text{MidLeft}}}$, where $\mathbf{v}_{\text{MidLeft}}$ is the midpoint between the first and fourth corner in the original coordinate system. The method can then get the angle between these two vectors, by using the formula for the scalar product and solve for θ on the X-Z-plane [Azad, 2019]:

$$\theta = \arccos\left(\frac{\overrightarrow{\mathrm{A}} \cdot \overrightarrow{\mathrm{B}}}{\parallel \overrightarrow{\mathrm{A}} \parallel \times \parallel \overrightarrow{\mathrm{B}} \parallel}\right)\ . \tag{3.57}$$

For these equations, the method only needs the X and Z-coordinate and thus uses the 2D formula for the scalar product. $\parallel \overrightarrow{\mathrm{A}} \parallel$ describes the length of the vector $\overrightarrow{\mathrm{A}}$ and arccos is the inverse of the cosine. As a result, the algorithm returns the angle θ in degrees, for which the standard vector has to be rotated to match the actual left side of the square.

As ceilings are upside down to the other horizontal spaces, the algorithm has to use the negative rotations there. This is the case, since when the square is created in the game engine, the scale and translation is applied first, before applying the rotation. The rotation is additionally first matched to the normal and then afterwards the rotation around the Y-axis is applied. Since the square representation within our plugin has the actual normal pointing upwards (for the creation process), the rotation has to be applied in the opposite direction, because the square in the game engine is flipped upside down (normal points in the opposite direction).

The volume height is defined for each feature. We take the non-occluded volume in front of the suitable are that was defined and checked for each feature. Through this the volume obtains the following dimensions: one m for the walls, 1.5 m for the floors, one m for the ceilings, and one m for the furniture. The volume height is always measured along the normal of the suitable surface.

The last step, is to add the categorization. For this we implemented an enumeration flag, which could only contain one element for each main category. The main categories are size, interaction, and type and the enumerator was mapped to the following integers:

$$\begin{aligned}
\text{NONE} &:= 0\\
\text{SIZE\_BIG} &:= 1\\
\text{SIZE\_MEDIUM} &:= 2\\
\text{SIZE\_SMALL} &:= 4\\
\text{INTERACT\_CLOSE} &:= 8\\
\text{INTERACT\_DISTANCE} &:= 16\\
\text{INTERACT\_NO} &:= 32\\
\text{TYPE\_WALL} &:= 64\\
\text{TYPE\_FLOOR} &:= 128\\
\text{TYPE\_FURNITURE} &:= 256\\
\text{TYPE\_CEILING} &:= 512
\end{aligned} \tag{3.58}$$

To make sure and check if always only one flag is set for each major category, the method counts the active bits for each range of bits assigned to one category. For the size and the type, we can easily obtain the correct category information by remembering the size of the set rectangle dimension and the feature that was detected in the areas. The interaction categorization is a little more complex, as the algorithm has to find the nearest floor area to the interactive content area, which is not obscured by any objects. This can be done in parallel for each feature type, but has a few differences for each of them. We will discuss this for each aspect in detail. To detect the nearest floor and wall areas occluding the suitable surface, we use $\mathcal{V}_{\text{Wall}}$ and $\mathcal{V}_{\text{Floor}}$ before we reduced them to only non overlapping surfaces. For the areas to be compared, the process only uses the results after the reduction.

3.5.1. Interaction Category Wall

As walls tend to go from floor to ceiling, the process to detect walls between floor and wall surfaces can be compressed to the X-Z-plane. If the wall surface centre is above 1.5 m, the algorithm deems it to be too far away for direct interaction and only checks if it is reachable from a distance. The method cycles through all elements in $\mathcal{V}_{\text{WallFinal}}$ and compares it with each floor element in $\mathcal{V}_{\text{Floor}}$, if a suitable floor area can be found, which is closer than 20 cm to the wall (if the wall rectangle is below 1.5 m, otherwise we only need a suitable floor square). Such a space needs to be located at most 60 degrees off from the wall's normal orientation. For this, the process creates a vector from the wall centre to the floor centre and calculates the angle to the normal with Equation 3.57. If this is fulfilled, it checks if a wall element is between the wall and the floor, comparing it with all walls in $\mathcal{V}_{\text{Wall}}$. The algorithm establishes if the distance between both centres is smaller than 20 cm or if the upper line segments of both walls intersect, making sure that the suitable area is not invalidated by a rectangle attached to the same wall. For the distance, this implementation subtracts the area width of both rectangles from the total distance, determining if the centres lie within the range of the other surface.

The intersection of the two upper line segments is established based on the formulas provided by [Geeks, 2020]. All the algorithm needs to validate the intersection are the two end points of each line segment (p_1, q_1) and (p_2, q_2). An ordered triplet of these four points can have three different orientations: clockwise, counter-clockwise, and collinear. For a visual representation, see Figure 3.16.

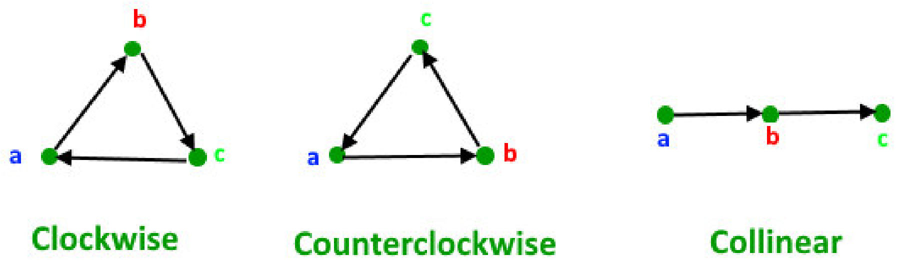


Figure 3.16.: The three possible orientations for a triplet of points (a, b, c) on a 2D plane. The left shows the clockwise order, the middle the counter-clockwise, and the right the collinear combination. Image taken from [Geeks, 2020].

The orientation helps the process to determine if the two line segments intersect, if and only if (p_1, q_1, p_2) and (p_1, q_1, q_2) have a different orientation and (p_2, q_2, p_1) and (p_2, q_2, q_1) have a different orientation. There is a special case if all four triplets are collinear. If this is the case, the method checks if the X and Y projections of both line segments intersect.

The orientation can be obtained by comparing the slope of segment (a, b) and (b, c), because if the slope of $\overrightarrow{ab}$ is larger than $\overrightarrow{bc}$, then the orientation is clockwise. Thus, the orientation depends on the sign of the following expression:

$$(z_b - z_a) \times (x_c - x_b) - (z_c - z_b) \times (x_b - x_a) \; . \tag{3.59}$$

If the result to Equation 3.59 is zero, then it is collinear, if it is larger than zero, the orientation is clockwise and counter-clockwise otherwise. For the collinear special case, the algorithm establishes for any collinear triplet, if the point from the other line segment lies on the segment of the first line. For this, we check if the X and Z coordinate of the other point lies between those of the first line segment. We only have to determine if the coordinates are within range, as we already know that the point from the other segment lies on the line through it being collinear.

If the wall piece does not lie on the wall surface, for which the method wants to determine the interaction level or for which the upper line segments do not intersect, the process can determine if the line of sight is blocked by the wall. We do this by creating a line segment between the centre of the floor and the centre of the wall and establish if the upper line segment of the blocking wall intersects this line of sight (by using the formulas described above). If this is not the case, we know that the floor tile is valid.

For all valid floor tiles for a wall rectangle, the process can now determine if the distance is less than 20 cm from the edge of the square, by calculating the distance of both centre points on the X-Z-plane and remove the area height of the square from this distance, establishing if the floor area is close enough to allow direct interaction.

With the above information, we can now easily categorise the interaction level of our wall area. If no suitable floor was found within the above restrictions, we set the interaction level to *INTERACT_NO*. If a floor tile within 20 cm was found *INTERACT_CLOSE* is set and *INTERACT_DISTANCE* otherwise.

3.5.2. Interaction Category Horizontal Surface

The floor interaction level algorithm needs some adjustments, compared to the wall interaction level algorithm described in the previous chapter. An unsuitable angle between two floor tiles does not exist, causing us to omit the angle check of 60 degrees. Instead, the algorithm only compares the distance of each entry centre in $\mathcal{V}_{\text{FloorFinal}}$ to a floor element core in $\mathcal{V}_{\text{Floor}}$. If the distance is larger than zero after subtracting both area heights, it determines if the direct line of sight is intersected by an upper line segment of a wall rectangle. We use the intersection algorithm described in the previous section for this purpose.

As soon as a minimum distance of less than 20 cm is reached or no suitable floor square was found for a valid floor tile, the algorithm is terminated and the interaction level is assigned to the category flag. If no suitable square is found, *INTERACT_NO* is assigned, while if a floor tile within 20 cm distance is identified, the valid space gets the flag *INTERACT_CLOSE* and *INTERACT_DISTANCE* is set otherwise.

The ceiling categorization has the same foundation as the floor interaction level algorithm, where the process checks the distance on the X-Z-plane of all elements in

$\mathcal{V}_{\text{CeilingFinal}}$ to the floor tiles. However, as ceiling elements are located above 1.5 m (assumption that ceilings are at least that much above the ground in previous section for detecting ceiling and floor levels), we cannot have a close interaction flag. For this the algorithm can then terminate as soon as a valid floor tile was found, which is not blocked by wall piece for the direct line of sight. This leaves us with the flags *INTERACT_NO* if no such floor tile was found or *INTERACT_DISTANCE* otherwise.

The furniture categorization uses the same algorithm as applied to the floor by going through all elements in $\mathcal{V}_{\text{FurnitureFinal}}$ and subtract the area height from both areas from the distance between the two centres. This leaves us with labelling the flag to either *INTERACT_NO*, *INTERACT_CLOSE*, and *INTERACT_DISTANCE* dependent on no suitable floor tile, floor tile closer than 20 cm, and otherwise respectively.

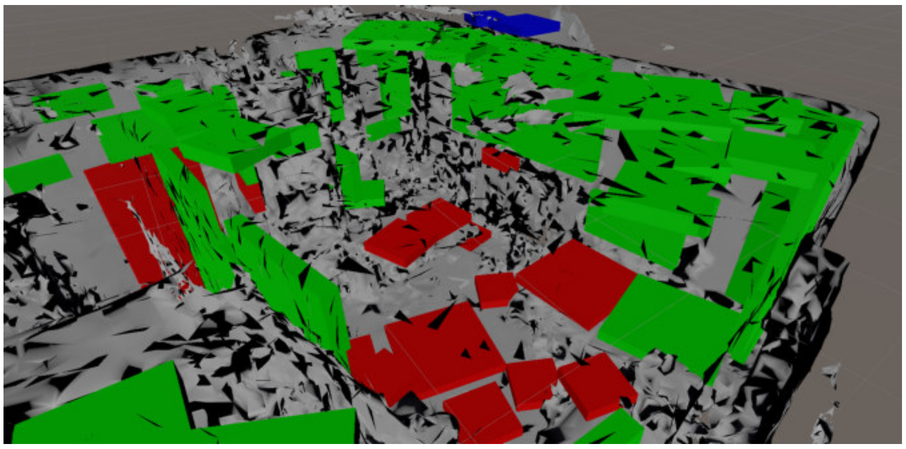


Figure 3.17.: A visual representation of the interaction level of each valid surface. Red indicates close interaction, while green represents the interaction possibility over a distance. Blue stands for no interaction possible. The last category only appears for areas, which cannot be reached by any floor tiles, which in our sample are two areas at the top. These ceiling areas are outside the room and are cut of by walls from any floor tile. Image is a screenshot taken within the Unity Editor.

The final interaction level categorization result can be seen in Figure 3.17. Any areas close to the floor are correctly labelled as *INTERACT_CLOSE* with the red colour. Green colours indicate the flags *INTERACT_DISTANCE*, which should be the flag for almost all other tiles. Only a few special cases should be of the colour blue (as is also the case in the image), since an area with no interaction possibility is not a good area for interactive content. In our sample, we only have two ceiling tiles outside of the room, which are cut off by walls from any floor tiles (top most areas in Figure 3.17). This, however, does prove the correct functionality of our algorithm rather nicely.

3.6. Room Recreation

The goal of this thesis is in part to allow creators to design procedurally generated levels with a different environment based on the boundaries of the room. For this, we decided to create a very simplified mesh of the room, which contains most of the relevant structures. With this simplified mesh, the strain on the game engine is reduced for checking physical collisions or ray-casts against the room boundaries, which are needed to obtain data about the shape and bounds of obstacles.

For this algorithm, the beginning uses the same methods as the room understanding algorithm. The method takes a vector $\mathcal{V}_{\text{Input}}$, defined in Equation 3.1, as input and converts the coordinates to the internal voxelized representation, giving us matrix $\mathbf{V}$ defined in Equation 3.7. Nevertheless, in the feature detection, which is next step, the first differences start appearing, as our proposed method does not need to detect all the features needed in the room understanding approach. The algorithm detects the floor and ceiling levels using the algorithm described in Section 3.2.1 and the wall detection algorithm described in Section 3.2.2, as well as the matrix $\mathbf{B}_{\text{Room}}$ for the boundary detection method.

With these information, we now want to create a simplified voxel representation of the room, filling empty areas, which only represent unreachable areas for HMD users. Areas counting towards this categorization are areas below desks, which can only be reached while crouching, holes in the mesh, or empty spaces between for example two walls, which cannot be reached at all. To achieve this in the most simplistic way, we use an adapted variant of our furniture level algorithm described in Section 3.2.3.

The algorithm goes through the room in two different directions, once from 1.5 m above the floor to the floor level and once from 1.5 m above the floor to the ceiling level. When going though each Y level of the room, the method remembers, similarly to the furniture algorithm, the level, at which the first voxel was encountered for each Y-column. The outcome of this are the two matrices $\mathbf{G}_{\text{Bounds}}$ and $\mathbf{H}_{\text{Bounds}}$, which are filled for the first and second pass respectively:

$$\begin{aligned}
\mathbf{G}_{\text{Bounds}} &= (g_{xz}) \in \mathbb{Z}^{L_{\text{Grid}} \times W_{\text{Grid}}} \\
(g_{xz}) &= \max_{\mathcal{I}_{\text{Floor}} < y \leq \mathcal{I}_{y\text{End}}} f_{\text{LevelLower}}(y) \\
f_{\text{LevelLower}}(y) &= \begin{cases} y, & \text{if } v_{xyz} = 1 \\ \mathcal{I}_{\text{Floor}}, & \text{if } b_{xz} = 1 \wedge y = \mathcal{I}_{\text{Floor}} + 1 \wedge v_{xyz} = 0 \\ -1, & \text{otherwise} \end{cases}
\end{aligned} \tag{3.60}$$

$$\begin{aligned}
\mathbf{H}_{\text{Bounds}} &= (h_{xz}) \in \mathbb{Z}^{L_{\text{Grid}} \times W_{\text{Grid}}} \\
(h_{xz}) &= \min_{\mathcal{I}_{y\text{End}} \leq y < \mathcal{I}_{\text{Ceiling}}} f_{\text{LevelUpper}}(y) \\
f_{\text{LevelUpper}}(y) &= \begin{cases} y, & \text{if } v_{xyz} = 1 \\ \mathcal{I}_{\text{Ceiling}}, & \text{if } b_{xz} = 1 \wedge y = \mathcal{I}_{\text{Ceiling}} - 1 \wedge v_{xyz} = 0 \\ \mathcal{I}_{\text{Ceiling}} + 1, & \text{otherwise} \end{cases} .
\end{aligned} \tag{3.61}$$

$\mathcal{I}_{y\text{End}}$ is defined in Equation 3.13. The algorithm sets all voxels on floor and ceiling level to be part of the room bounds, as long as this coordinate pair is within the room. For ceilings, the process establishes the value for non suitable bounds on the Y-column to $\mathcal{I}_{\text{Ceiling}} + 1$, as it searches for the minimum value and will only obtain this, if none of the other options are fulfilled. We now go ahead and create a final bounds matrix $\mathbf{I}_{\text{Bounds}}$, which activates all voxels between the floor level and the value saved in $\mathbf{G}_{\text{Bounds}}$, as long as the value is not negative one and all voxels between the ceiling level and the value saved in $\mathbf{H}_{\text{Bounds}}$, as long as it is not equal to $\mathcal{I}_{\text{Ceiling}} + 1$. This creates the described boundaries, which do not contain any holes or empty and unreachable areas:

$$\begin{aligned}
\mathbf{I}_{\text{Bounds}} &= (i_{xyz}) \in \mathbb{B}^{L_{\text{Grid}} \times H_{\text{Grid}} \times W_{\text{Grid}}} \\
(i_{xyz}) &= \begin{cases} 1, & \text{if } \mathcal{I}_{\text{Floor}} \leq y \leq g_{xz} \vee \mathcal{I}_{\text{Ceiling}} \geq y \geq h_{xz} \\ 0, & \text{otherwise} \end{cases} .
\end{aligned} \tag{3.62}$$

The last step to complete the voxelized representation of the room boundary, any holes are filled, which might still remain and are easily detectable. For this, the method checks each voxel and its four surrounding voxels for the X-Z, X-Y and Y-Z-plane respectively. If the voxel is not active and on any of the three planes at least three voxels are active, this voxel is additionally activated.

We now have to convert our voxelized representation to a simplified mesh. Since our implementation is using a voxel representation, existing algorithms can be utilized to convert the representation to an almost optimal mesh, maintaining the right balance between the speed of the algorithm and the amount of vertices and faces contained in the resulting mesh. The algorithm described by [Lysenko, 2012] is adapted in our implementation and is called greedy meshing.

The basic idea is to join adjacent cubes together into larger regions, which reduces the total size of the voxelized representation. For this, the algorithm only needs to look at the voxelized representation from three different directions along the X, Y, and Z axis, reducing a 3D problem down to 2D. This returns a collection of adjacent side slices along each axis, leaving only the meshing for each of these 2D slices for each direction. An optimal representation with the fewest rectangles is quite hard to find and performance intensive. So instead the method uses lexicographic ordering, which has at most eight times the number of rectangles, compared to the optimal mesh [Lysenko, 2012]. For the lexicographic ordering, we define at which voxel we start, and the order of voxels checked by the algorithm next to create the new rectangle. We define the following total order for the three axes $(\text{FirstCoordinateAxis}, \text{SecondCoordinateAxis}) = (a, b)$: first X then Y for the X-Y-plane, first Y then Z for the Y-Z-plane, and first Z then X for the X-Z-plane. The third axis, along which the process checks is defined as c.

This creates the 2D mask $\mathbf{J}_{\text{Mask}}$ with the following formula for each level of axis c, starting at negative one and ending at $\max(c)$:

$$\begin{aligned} \mathbf{J}_{\text{Mask}} &= (j_{ab}) \in \mathbb{B}^{\max(a) \times \max(b)} \\ (j_{ab}) &= \begin{cases} 1, & \text{if } i_{abc} \neq i_{a,b,c+1} \\ 0, & \text{otherwise} \end{cases} . \end{aligned} \tag{3.63}$$

For accessing the bounds matrix i_{xyz}, each axis a, b, c needs to be matched to its correct counterpart defined previously with the lexicographic ordering. If an access on i_{xyz} is invalid, because the axis coordinate is outside the bounds of the matrix, the result will just default to zero and is used instead of Equation 3.63. For each mask, the algorithm then creates a set of rectangles, starting at axis a and checking each element in the mask if it is active. A soon as the process encounters an active element, it checks for each consecutive mask entry if it is active and increase the width dimension of the rectangle by one. As soon as an inactive voxel in the mask is encountered, the width w is remembered and the analysation to define the height of the rectangle starts. As our implementation already checked the first height level, we now need to compare along axis b, if any active elements for all active voxels along the same values for axis a can be found. We do this for each consecutive level along axis b, until not all active voxels can be matched. The final height h is remembered and then the rectangle is added to the list of rectangles $\mathcal{V}_{\text{Quads}}$, by adding the four corners:

$$\begin{aligned} \mathbf{p}_{\text{Corner1}} &= (a \quad\quad , b \quad\quad , c) \\ \mathbf{p}_{\text{Corner2}} &= (a + w \quad , b \quad\quad , c) \\ \mathbf{p}_{\text{Corner3}} &= (a + w \quad , b + h \quad , c) \\ \mathbf{p}_{\text{Corner4}} &= (a \quad\quad , b + h \quad , c). \end{aligned} \tag{3.64}$$

For Equation 3.64, the algorithm has to match the axes a, b, c again to the correct axes x, y, z. Afterwards, it sets all mask elements within the rectangle to false, as they have been covered and continues on the first element on the a-axis after the rectangle. Repeat this process for all mask slices in all three axis directions.

In Figure 3.18, we show on a smaller sample mask the process of creating a greedy mesh. In the above process, we have omitted one step, which is necessary to get a correctly represented mesh of the room. As the algorithm currently checks along the three axes and creates a greedy mesh based on those masks, it only builds rectangles with a normal orientation pointing towards the axis direction from which it went through each level. With one small adjustment, we can correct this and return the rectangles facing the correct direction (towards the outside of the mesh volume). The process can define the side on which a voxel is blocked by another voxel through checking if

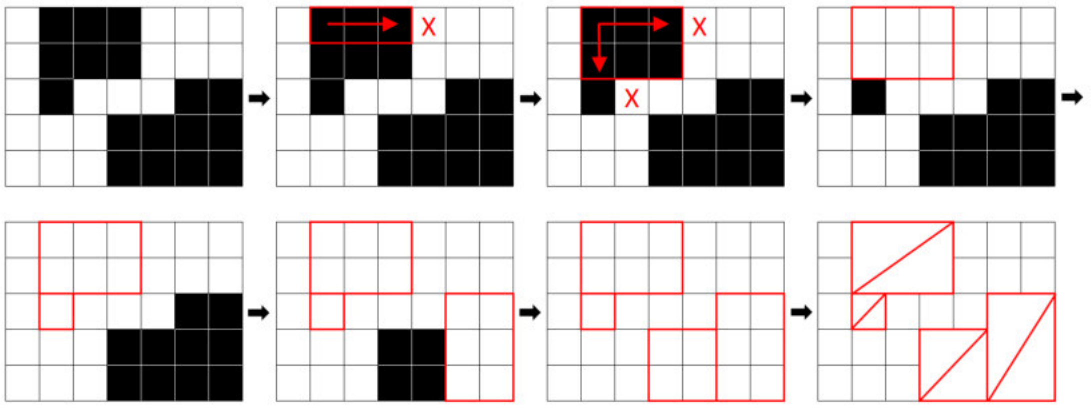


Figure 3.18.: The process of creating a mesh through greedy meshing from the top left to the bottom right. We start with the mask for the current slice, where black squares represent activated voxels. We then go through each voxel and when the algorithm hits the first active element, we start creating the rectangle width until we hit the first non active spot (red cross). Now the process starts on the next row and checks if all previous active elements are also active here, which is the case. On the next row, our algorithm terminates, as the second voxel is not active. This creates a rectangle of width 3 and height 2. We repeat this process until all active elements in the mask have been taken care of and we are left with four rectangles in our example. The method then always connects corner one, two, and four, as well as two three and four to a triangulated face. With this, each rectangle is turned into two triangles. Image was created with the help of Microsoft Excel.

$i_{abc} = 1 \wedge i_{a,b,c+1} = 0$. If this statement is false, we know that the algorithm has to switch the orientation of the rectangle. For creating the rectangle, it has to be additionally established for each active voxel in the mask, if they are facing towards the same side. For an orientation switched rectangle, we save the four corners in the following adjusted way:

$$\begin{aligned} \mathbf{p}_{\text{Corner1}} &= (a &&, b &&, c) \\ \mathbf{p}_{\text{Corner2}} &= (a &&, b+h &&, c) \\ \mathbf{p}_{\text{Corner3}} &= (a+w &&, b+h &&, c) \\ \mathbf{p}_{\text{Corner4}} &= (a+w &&, b &&, c)\ . \end{aligned} \tag{3.65}$$

This changes the orientation, as we are creating the triangulated mesh representation by turning one rectangle into two triangles with the vertices $(\mathbf{p}_{\text{Corner1}}, \mathbf{p}_{\text{Corner2}}, \mathbf{p}_{\text{Corner4}})$ and $(\mathbf{p}_{\text{Corner2}}, \mathbf{p}_{\text{Corner3}}, \mathbf{p}_{\text{Corner4}})$. We adapted our vertex order for triangles to Unity's internal system, which uses a clockwise order [Unity, 2020b]. Through the switch of corner two and four, we flip the vertex order of the triangle and thus the render orientation.

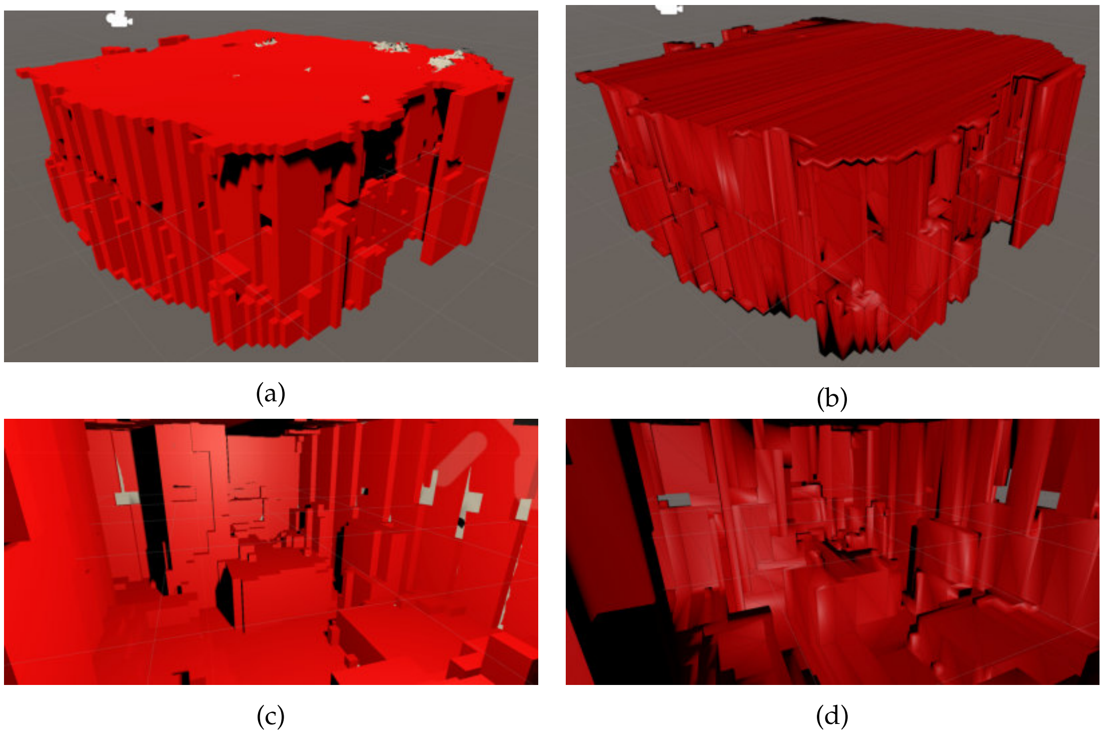


Figure 3.19.: The boundaries of the room. Figure (a) and (c) show the outside and inside directly from the voxelized representation, by rendering a cube for each active voxel. Compared to the input voxel representation seen in Figure 3.2, the room bounds representation only contains a few holes and has very clean sides with only little noise. Figure (b) and (d) are the same bounding volume, only as a mesh after the greedy meshing algorithm. The structure of the voxels were kept from the voxel representation. The inaccurate behaviour for shading the room mesh is due to not normal and UV mapping the vertices. Images are screenshots taken within the Unity Editor.

To obtain the final mesh representation of the room seen in Figure 3.19 and 3.20, we have to shift the whole mesh by half a voxel dimension towards each negative axis. This is due to using the voxel centres for our greedy mesh algorithm and defining a width and height of one when we encountered one active voxel. This resulted in a rectangle stretching from the centre of the current voxel to the centres of the next voxels. By shifting the whole mesh by half a voxel, the voxel centre is returned to the middle of the rectangle and the sides of the rectangle match again with sides of the voxel.

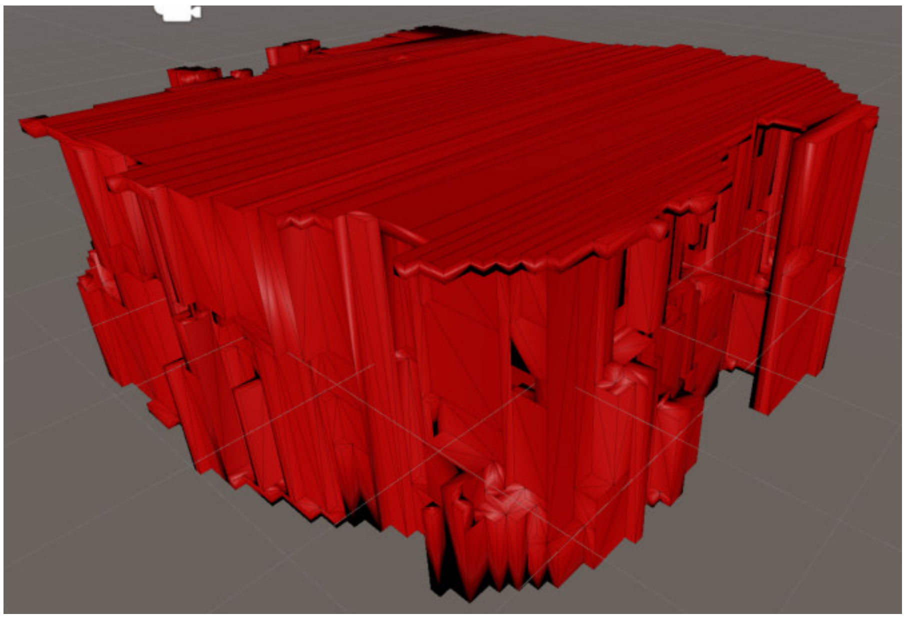


Figure 3.20.: This is a close-up look at Figure 3.19b. The triangulation of the mesh can actually be seen in this screenshot, as we render the shaded geometry and the wireframe. This mesh representation might not be the most optimal solution in relation to vertex and triangle count, but reduces the numbers considerably compared to the original room mesh. The specific reductions are shown in Chapter 4. Image is a screenshot taken within the Unity Editor.

3.7. Optimization

We introduced a few steps and methods to optimize the plugin and keep the performance load from the main execution loop in the game engine. To keep the main loop in the application from freezing and keep it independent from our plugin's calculations, the entire plugin runs in an extra thread. Additionally, our implementations try to parallelize as much of the code as possible. Throughout the above sections, we mentioned, which parts can be executed in parallel. For those sections, the algorithm used an asynchronous execution (this has less overhead than threads in C++) and synchronized the thread afterwards again, when the results were written to a common variable.

The room understanding algorithm has four sections, which can be easily run independently from each other, consisting firstly of the feature detection, where not all but the floor, ceiling, and furniture feature detections can run in parallel. Next come the suitable areas for interactive content definition, where each size for each feature gets its own thread for a total of 12 threads. Up next is the reduction of the recognition, and lastly the setting of the interaction flag. All of these are quite computing extensive task and speed up the execution time immensely, especially for large rooms and weaker hardware with at least four CPU cores for optimal utilization of our threads.

For our room bounds algorithm, we do not use any asynchronous executions, since the current execution time is already very fast and the threading would create more overhead, compared to the time that could be saved through the parallel processing. Here, only the main plugin thread executes in a separate thread to the main game loop (same as for the room understanding).

As one might have already noticed in the descriptions of all the algorithms, we do not always cycle through our whole voxel representation, which would increase the execution time dramatically. We try to optimize each algorithm towards its minimum number of cycles. After determining the level of the floor and the ceiling, we seldomly analyse the Y-dimension outside these two levels. The algorithm additionally terminates cycles as soon as a final answer is found and never cycles through more elements than necessary. Especially for large rooms, we can speed up the suitable area detection process by only searching for corresponding corner points in the maximum distance range to the first corner. Often the process gains speed in the execution time through assumptions and focus on usable results, which omits many special cases with computing intensive calculations. These special cases are mentioned again in the future work with analysis of their need and suggestions on the implementation.

3.8. Plugin Usage

Below we will describe how our plugin is implemented for Unity and Unreal Engine, starting by discussing the general usage for any framework. In Section 3.1, we already portrayed the input format, which is a flattened floating-point array with all vertex positions of the room mesh. The unit length and scale are adapted to fit to Unity's internally used system, which is a left-handed, Y-up coordinate system with a unit length equal to one m. The room understanding algorithm is initialized by the function

```
int StartRoomUnderstanding(const float* VertexPositions,
                           const int ArraySize);
```

and takes the flattened vertex array and the number of elements within as input. The integer returned by the function, indicates if it was successful in initializing the room understanding algorithm. A zero indicates success, while one is returned when there was no input array or the size was smaller than one. A two is returned when the size is not divisible by three and a four if there is currently another room understanding thread still running.

As we do not want to block the main loop of the calling software, we are executing our plugin core calculations in a separate thread. To obtain the results, the method

```
float* RetrieveRoomUnderstanding(int& OutErrorCode,
                                 size_t& OutArraySize);
```

needs to be called. Since there is no indication when our algorithm is finished, we allow this function to be called whenever it is suitable for the calling software, returning negative one for the error code if the algorithm is not yet finished and zero if the results are ready and can be retrieved. A one is returned if the thread finished, but due to an internal error, no vertices were returned. Both the error code and array size are passed by reference and will hold a result, where the array size is the number of elements of the returned array and is a multiple of eleven. The returned array holds the relevant space classification information for each valid surface area as a flattened floating-point array. Each valid surface holds eleven elements in the array and are distributed as follows:

1. X-coordinate of the centre point of the valid surface
2. Y-coordinate of the centre point of the valid surface
3. Z-coordinate of the centre point of the valid surface
4. X-coordinate of the surface normal of the valid surface
5. Y-coordinate of the surface normal of the valid surface
6. Z-coordinate of the surface normal of the valid surface
7. Area height measured from the centre to the upper bounds of the area
8. Area width measured from the centre to the right bounds of the area
9. Rotation around the centre point with the normal vector as the rotation axis
10. Volume height indicating the maximum height of the volume away from the surface plane
11. Category as an enumerator flag converted to a floating-point value

The number returned for the category needs to be converted back to the enumerator flag defined in Equation 3.58. This is easily done by converting the floating-point value to an integer and then converting it to the enumerator. The returned coordinates and units are indifferent to the input format and have to be converted to the target platform format if necessary.

We moved the memory management of the output array to the software calling our plugin, so the results do not get deleted prematurely. This has the consequence, that the memory has to be deleted afterwards by calling the method

```
int CleanUpRoomUnderstanding();
```

that deletes any remaining elements and frees the memory. If this is not called, a memory leak will be inevitable.

The usage of the bounds algorithm works similarly to the room understanding and can be used independently of it (running in parallel to it is possible). It is initialized with

```
int StartRoomBounds(const float* VertexPositions,
                    const int ArraySize);
```

and takes the same input and returns the same error codes as the initialization function of the room understanding algorithm. As this algorithm is executed in a separate thread, the user of the plugin needs to call

```
float* RetrieveRoomBounds(int& OutErrorCode,
                          size_t & OutVertexSize,
                          size_t & OutTriangleSize);
```

from time to time and check if the results are ready. The returned error code is equal to the one from the room understanding retrieval function. Here, we have the size of the returned vertices and the triangles. The size of the total returned array is the added size of the vertex and triangle portions. Again, we are returning a flattened array, where the first number of elements, equal to the returned vertex size, are the coordinates of the vertices of the returned mesh. A group of three entries always returns the X, Y, and Z-coordinates of the vertex. There are no multiple entries for the same vertex, as this was already optimized in our algorithm. The list of triangles comes after the vertex entries and contains the number of elements defined through the returned triangle size. A triplet describes three vertices forming a triangle and are ordered in a clockwise rotation (which is the standard for Unity). The entries in each triplet are an index reference to the corresponding vertex.

Again, we left the memory management to the software calling our plugin, which is why the call of

```
int CleanUpRoomBounds();
```

is required after all data was retrieved from our plugin.

All functions described above are defined in `RoomCreatorLibrary.h` and are located in the namespace `RoomCreator`. Our plugin uses no other dependency than the C++ Standard Library, to minimize compatibility issues and allow a flawless deployment over a range of platforms and hardware devices.

3.8.1. Usage within Unity

It is rather easy to implement and use a DLL in a project. The DLL file needs to be added to the *Assets* folder and can be accessed by the following code snippet [Unity, 2020e]:

```
[DllImport("RoomCreatorLibrary", EntryPoint = "StartRoomUnderstanding")]
extern static public int StartRoomUnderstanding(float[] vertexPositions,
                                                int arraySize);

[DllImport("RoomCreatorLibrary", EntryPoint = "RetrieveRoomUnderstanding")]
extern static public IntPtr RetrieveRoomUnderstanding(ref int errorCode,
                                                      ref int arraySize);

[DllImport("RoomCreatorLibrary", EntryPoint = "CleanUpRoomUnderstanding")]
```

```
extern static public int CleanUpRoomUnderstanding();

[DllImport("RoomCreatorLibrary", EntryPoint = "StartRoomBounds")]
extern static public int StartRoomBounds(float[] vertexPositions,
                                         int arraySize);

[DllImport("RoomCreatorLibrary", EntryPoint = "RetrieveRoomBounds")]
extern static public IntPtr RetrieveRoomBounds(ref int errorCode,
                                               ref int vertexSize,
                                               ref int triangleSize);

[DllImport("RoomCreatorLibrary", EntryPoint = "CleanUpRoomBounds")]
extern static public int CleanUpRoomBounds();
```

This code allows all six functions of our plugin to be called within the unity environment. The `ref` keyword allows to pass a reference of that variable to our plugin and an `IntPtr` can handle a pointer from C++ in C# [Microsoft, 2020b]. As we know the size of the array, we can easily copy the data in the `IntPtr` towards a floating-point array in Unity. Optionally, a class to store all the relevant details for each detected surface can be created.

3.8.2. Usage within Unreal Engine

To include our plugin within the Unreal engine, a few more steps are necessary, but once included, the usage is simpler, compared to Unity. First, we need to save the DLL file to `<Project Root Folder>\Binaries\Win64`. Next, we need to include the `.lib` file of the library in the folder `<Project Root Folder>\ThirdParty\RoomCreatorLibrary\Libraries` and the header files of our plugin in the folder `<Project Root Folder>\ThirdParty\RoomCreatorLibrary\Includes`. Now the library files need to be included in the build process of the engine, which can be accessed in `<Project Name>.Build.cs`, which should be located at `<Project Root Folder>\Source\<Project Name>`. In the function

```
public <Project Name>(ReadOnlyTargetRules Target) : base(Target)
{
       //[...]

       LoadRoomCreator(Target);
}
```

add `LoadRoomCreator(Target);` at the end and add the following code in the same file:

```
private string ModulePath
{
       get { return ModuleDirectory; }
}

private string ThirdPartyPath
{
        get { return Path.GetFullPath(Path.Combine(ModulePath,
                "../../ThirdParty/")); }
}

public bool LoadRoomCreator(ReadOnlyTargetRules Target)
{
        bool isLibrarySupported = false;
        string LibraryName = "RoomCreatorLibrary";
        if ((Target.Platform == UnrealTargetPlatform.Win64))
        {
                isLibrarySupported = true;
                string LibrariesPath = Path.Combine(ThirdPartyPath,
                        LibraryName, "Libraries");
                PublicAdditionalLibraries.Add(Path.Combine(LibrariesPath,
                        LibraryName + ".lib"));
        }
        if (isLibrarySupported)
        {
                PublicIncludePaths.Add(Path.Combine(ThirdPartyPath,
                        LibraryName, "Includes"));
        }
        Definitions.Add(string.Format("WITH_ROOM_CREATOR_BINDING={0}",
                isLibrarySupported ? 1 : 0));
        return isLibrarySupported;
}
```

Currently, this only adds the plugin for the `Win64` platform, but this can be adjusted to any supported platform from the plugin. With these steps performed, the functions of the plugin can be accessed by including the header file of our six functions. As our plugin is written in C++, this takes care of everything and we can use the functions natively within the code of the Unreal Engine project.

We only have to adjust the input and output format to the from the plugin internally used formats, since Unreal uses as left handed, Z-up coordinate system with a unit scale of one cm. For the coordinates of the vertices, we convert them based on Equation 3.2 to be used within our plugin and Equation 3.3 for the conversion back to the Unreal Engine. For the rendering of triangles, Unreal utilises a counter-clockwise ordering for the vertices of a triangle. To obtain a correctly rendered mesh, we switch corners two and three for the triangles, which turns the order around.

4. Evaluation and Discussion

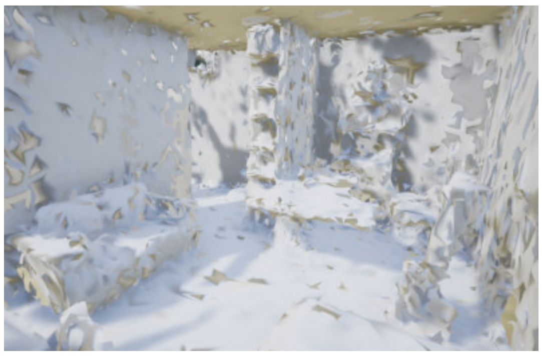
For the evaluation of our plugin, we cannot take the general route to use some existing tests and compare our solution to others, since there are no solutions, which are comparable in scope and compatibility. The existing solutions work only on the HoloLens and HoloLens 2 and return results in different formats, where either a list of all suitable locations is not accessible, or the computation is hidden away on special HPUs, making a comparable timing difficult, especially as both approaches have separate steps for analysation and computation of outcomes [Microsoft, 2018b; Microsoft, 2019b].

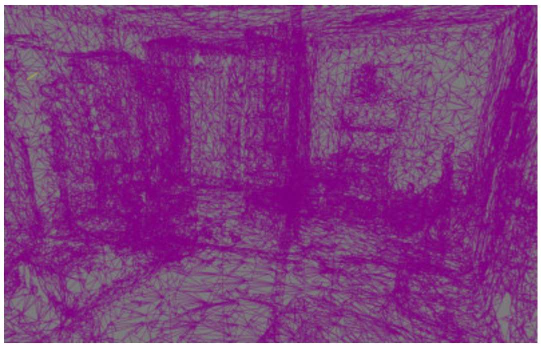
While a comparison to existing methods might not be possible for an evaluation, we do have the options of measuring the execution time for different test environments and evaluating the provided results of our algorithms based on visual observations of detected areas and noticed shortcomings.

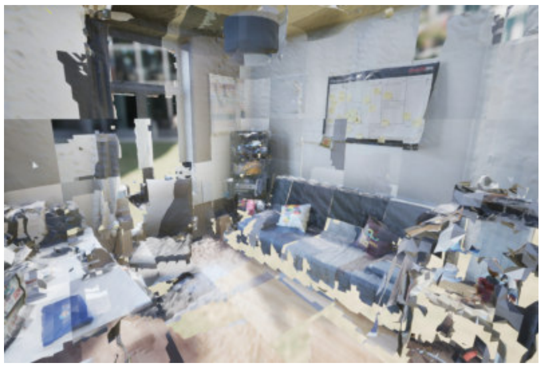
4.1. Tested Environments

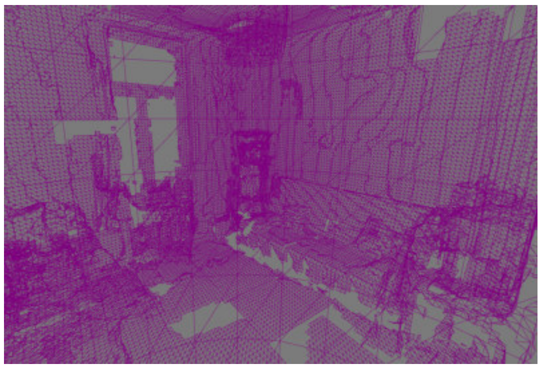
To evaluate our algorithm in different scenarios, we selected five distinct room scans for testing and evaluation. All scans are provided by Sketchfab under the Creative Commons license [CC, 2020]. Room one is a small studio scanned by a HoloLens and can be seen in Figure 4.1a [Sketchfab, 2017b]. The second room scan is of a small office scanned with an Evryway Scanner and is depicted in Figure 4.1c. The third environment and last one shown in Figure 4.1 is a HoloLens scan of a complete house from the inside. It contains two floors and multiple rooms (4.1e). Rooms four (Figure 4.2a) and five (Figure 4.2c) are both scanned with Agisoft Photoscan and the meshes were cleaned afterwards through 3D modelling software. The rooms depict a living room and a loft.

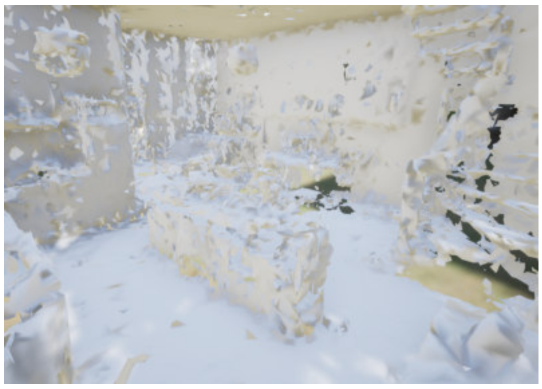
These five scans showed the first limitations of our current algorithm, as two scans are incompatible with our approach. While rooms one through three (Figure 4.1) could produce usable results, rooms four and five (Figure 4.2) could not be analysed by our proposed methods. The reason for this incompatibility can be discovered by looking at the wireframe representation of each room (the right column in both figures, where the triangle edges are represented through purple lines). While the first three rooms have a very even distribution of faces across the mesh geometry, except for a few holes especially on the floor, ceiling, and walls, this cannot be said about the fourth room. Here, the post-processing of the scan might have optimized it for rendering the geometry, but robs our algorithm of important features, which it can no longer detect. The high distribution density on furniture would be enough for our algorithm, however, the sparing amount of faces present on walls, floor, and ceiling are making the detection


(a)


(b)


(c)


(d)


(e)

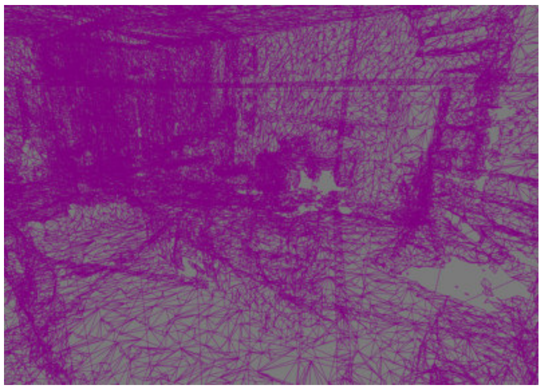

(f)

Figure 4.1.: The three room scans, which were compatible with our algorithm approach. The first room (a) is a HoloLens scan of a small studio. The vertices and triangles (see b) are evenly distributed across the room. Room two is a small office scanned with the Evryway Scanner (c). This scan has an even distribution, but does contain larger holes within the scan, especially for walls, floor, and ceiling (d). The third room (e) is a complete house scanned from the inside with a HoloLens. This scan has an even distribution of the mesh geometry, containing holes, two floors, and multiple rooms. Images are screenshots taken within the Unreal Engine Editor. Room scans provided by [Sketchfab, 2017a; Sketchfab, 2017b; Sketchfab, 2017d].

of those three features impossible only by the vertices. Through this, we are unable to detect the correct floor and ceiling level, which breaks the feature detection algorithm, since it no longer finds the correct levels. Room five on the other hand, has an even distribution of faces across the mesh geometry, but has another issue. The pitched roof area does not comply with our assumption, that floors and ceilings are both parallel to the X-Z-plane. Due to this, our approach cannot detect the correct ceiling level and again busts our algorithm for the same reasons as for room four.

This results in one more requirement for our room scans. They need vertices evenly distributed across the whole geometry of the room scan, otherwise it might break our implementation. Nevertheless, this should rarely be an issue, as our suggested approaches for scanning a room are either the HoloLens or the ZED mini, which both provide scans with a high distribution density across the entire mesh.

For the three suitable rooms, we looked at the dimensions, analysing if there is any correlation between room size and execution times and quality of the results. As Table 4.1 shows, all three rooms have a ceiling height of at least 2.5 m, which was one assumption for our rooms dimension. For room three, we need to divide the height by two, as this geometry depicts two floors. The rooms one and two have a rather quadratic foundation shape, while room three tends more to a rectangular boundary. The volumes of the observed room bounds are similar for rooms one and two. Despite the similar volumes, both rooms do perform differently in our algorithm, which we will discuss further in the coming sections.

		Room 1	**Room 2**	**Room 3**
Dimension	X (m)	6.52	6.49	13.70
	Y (m)	3.30	4.83	7.00
	Z (m)	6.72	5.16	10.33
	Volume (m^3)	144.48	161.89	990.32
Vertices	Original	78,430	332,088	232,097
Triangles	Original	131,178	110,696	385,207

Table 4.1.: The dimensions of our two suitable rooms. They all come with a ceiling height of at least 2.5 m and room one and two have a mostly quadratic foundation shape. Our room three, with its two floors, has a more rectangular shape and sufficient room height with around 3.5 m. The number of vertices and triangles shows that room two has a very high amount of vertices, which is a result of a non-optimized mesh, having for each triangle a separate triplet of vertices, which is not the case in the other two geometries.

We included the observed vertex and triangle count of each mesh, which are in part important to measure the quality of our simplified room boundaries produced by our approach, but also show how optimized the original mesh is. Room two, for example, is less optimized than rooms one and three, as it has a separate triplet of vertices for

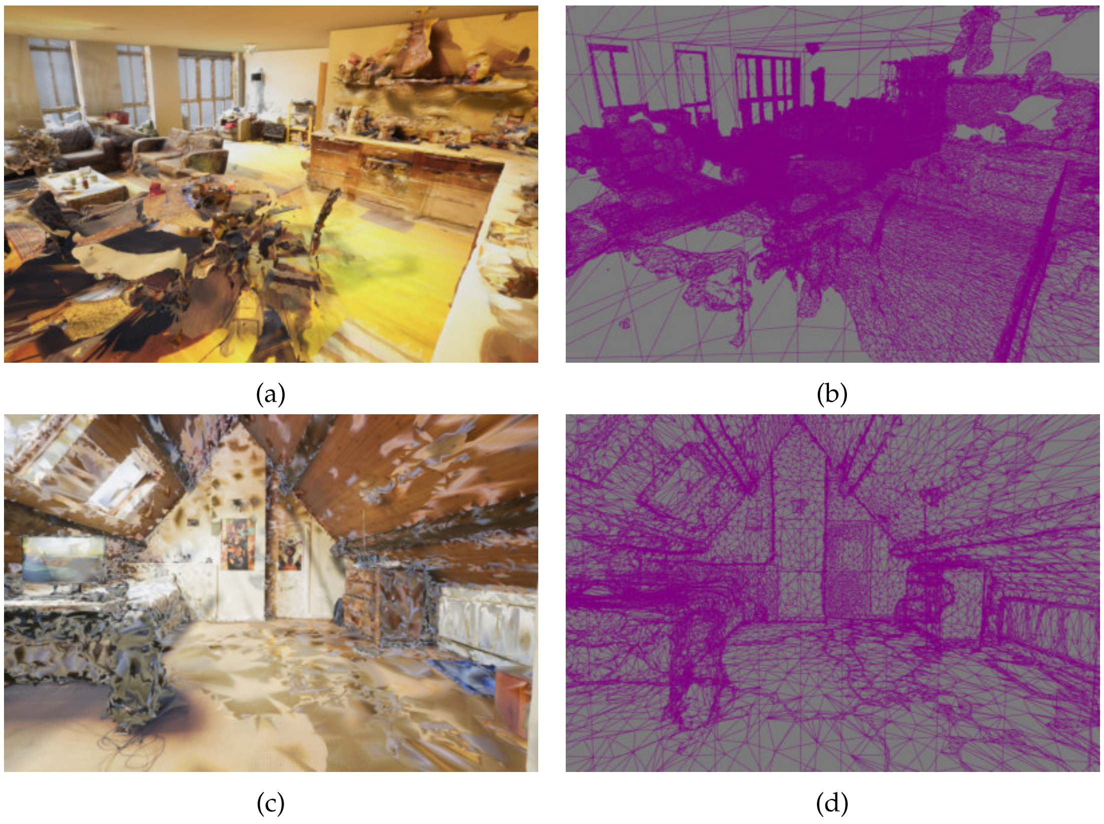


Figure 4.2.: Our approach could not produce any usable results for these two rooms. Room four (a) is a capture of a living room and was generated with Agisoft Photoscan. Afterwards it was cleaned up in Maya, which results in an uneven distribution of triangles and vertices, where detailed structures have a high density and walls, ceilings, and floors have a low density (see b). Through this, our algorithm could not correctly detect the ceiling and floor and thus could not operate correctly. Room five (c) is also scanned with Agisoft Photoscan and represents a loft. It was cleaned up, but still maintained a more or less evenly distribution of faces across the mesh (d). The reason for not being compatible with our algorithm lies in a pitched roof area, making our algorithm unable to detect the ceiling. Images are screenshots taken within the Unreal Engine Editor. Room scans provided by [Sketchfab, 2017c; Sketchfab, 2017e].

each triangle entry, where normally all shared vertices reference a single vertex entry. The triangle size is smaller for room two, since it has a comparable number to room one, but offers far less complexity in comparison. Room one has a few more niches and structures along the walls and furniture and the size difference can be observed by looking at Figures 4.1b and 4.1d

4.2. Tested Hardware

One main target for our algorithm is the usage with VR devices and their appropriate PC hardware, as well as applicable AR headsets. As it is easier to monitor tests on PC hardware, we decided to test our algorithm on recommended hardware for VR ready PCs. The recommended specifications for most VR headsets are an Intel i5-4590 CPU, NVIDIA GTX 1060 GPU, 8 gigabyte (GB) random access memory (RAM) and a Windows 10 operating system (OS) [Lang, 2019].

Our used hardware is comparable to the recommended specifications with a slightly more powerful CPU, which was verified using [Benchmark, 2020a; Benchmark, 2020b]. We used a PC with an Intel i7-5820K CPU, NVIDIA GTX 980 GPU, 16 GB RAM and a Windows 10 OS. As our plugin runs exclusively on the CPU, this becomes the crucial part of the tested hardware.

With the recent reveal of the new console generation specifications, featuring new eight-core CPUs, the currently four-core norm for gaming PCs (and recommended for VR) will adjust to those of the new consoles [Chacos, 2020]. Through this, we can assume that for the future, our used six-core CPU gives a better performance assumption of our plugin. Our CPU has a clock speed of around 3.3 gigahertz (GHz), which increases overall six-cores to up to 4.0 GHz if there is a medium to high CPU load. This is possible through a water-cooled CPU system.

4.3. Performance Analysis

The performance is measured individually for both main methods of our algorithm, the room recognition and the room reconstruction. For both methods, we will look at the execution time measured from the call of the initialization function until the results are obtained by the game engine. All times will be measured for Unity and the Unreal Engine, executed in the editor and as a built version for each framework. For the execution times, it will be important how many frames will pass in an application, considering that 90 FPS is deemed as the lowest refresh rate for VR devices to counteract disorientation, nausea, and other negative effects. Rounding down, this translates to rendering each frame in roughly 11 millisecond (ms), which will be our benchmark for an optimal execution time.

For the second part of the analysis, we will look at the visual results for each room and discuss possible shortcomings and limitations of our current approach. For this, we will compare if any differences are existing between the Unreal Engine and Unity or if our algorithms are deterministic.

4.3.1. Evaluation of Room Recognition

First off, we will look at the execution times for the room recognition algorithm. We tested our method in the editor and build versions of each engine and recorded the execution time for ten different runs. The results of these tests for each room can be found in Tables A.1, A.3, and A.5. We determined the average of our ten runs, by calculating the mathematical mean, which adds all results and then divides it by the total number of measurements. The average, as well as the range of all measurements, can be found in Graphic 4.3.

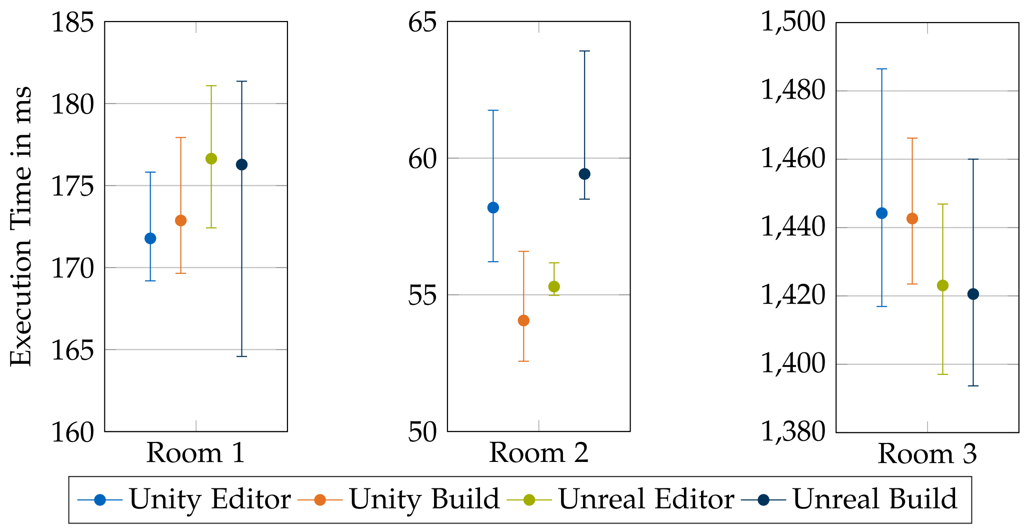


Figure 4.3.: The graphical representation of the average execution time of the room understanding algorithm for each room and tested version. The range of measured results is presented through the interval, while the average is the dot in between. Each room had very different execution times, which were mostly consistent over all tested versions.

One of the first things we noticed, was the vastly different execution times for each room. In contrast, the execution times stayed mostly consistent across the tested versions for each room. The time differences between each version can be explained by the distinctive CPU load. While the Unity and Unreal editor versions have a large overhead due to editor related operations, the build versions could operate on a very low load. This resulted in different clock speeds for the CPU. We observed constant clock speeds of 4.0 GHz for both editor versions, while the build targets only clocked speeds between 3.0 GHz and 3.7 GHz.

Despite being the second largest room, room two scored the best execution times between 52 and 64 ms, resulting in around 5-6 passed frames until the results could be processed by the target software. This is around three times faster, compared to the time it took the outcomes for room one to be ready (around 15-18 passed frames). We believe this has to do with the number of suitable areas detected and the complexity of the room geometry. Room two has a very simple structure, with four walls, one office desk, a sofa and only a few other furniture objects. In comparison, room one has two desks, a bed, a sofa, a storage rack and two niches. Each recognized feature in our algorithm has then to be checked if it contains a valid space, which results in a larger number of surfaces that have to be considered for exclusion, increasing the execution time of our algorithms exponentially. Nevertheless, an ideal room for any AR or VR experience will be in between room one and two, consisting of an almost square playing area and a set of furniture located at the walls of the room. Thus it is safe to say that in an average real-world environment, we will see an execution time between 55 and 180 ms, which in our eyes is fast enough for a real-time usage of our plugin, since our library does not need to be called every frame (not even the HoloLens updates the room geometry this fast [Tuliper, 2019]). The third room takes by far the longest, needing around 1.4 seconds (around 125-135 frames). Despite only analysing the first floor, the increase in computation cost is immense, as we are detecting a large amount of features and suitable areas throughout the geometry. The total number of detected features for each room are listed in Table 4.2. Nevertheless, for the size of the scan, waiting less than 1.5 seconds is still not a long time, especially as the main thread is not blocked.

	Room 1	**Room 2**	**Room 3**
Areas Detected	116	75	227

Table 4.2.: The number of detected areas for each room. While room one was actually the smallest room, our algorithm could detect 116 suitable areas on walls, floor, ceiling, and furniture. Room two had through its simpler geometry only 75 areas, while room three, due to its huge size, has 227 areas.

The different execution times between each tested version does suggest that a fixed and fast clock speed of the CPU is needed for consistent and fast processing times. Especially on slower hardware devices, such as the HoloLens, this could increase the times significantly. Nevertheless, as we are executing our plugin not in the main thread, this inconsistency (which ranges between less than one frame to a few frames in the large room), could be caused by the implementation of our plugin. We are checking each frame update if the plugin can provide the results already. If our plugin finishes only a few moments after the update call checked, this could delay the reception of the results by several ms (depending on the current FPS), as another frame has to be rendered before the update function is called again.

The left column in both Figures 4.4 and 4.6 shows the room understanding results from our plugin in Unity and Unreal Engine respectively. The most important observation is that both results are exactly the same, proving the deterministic nature of our

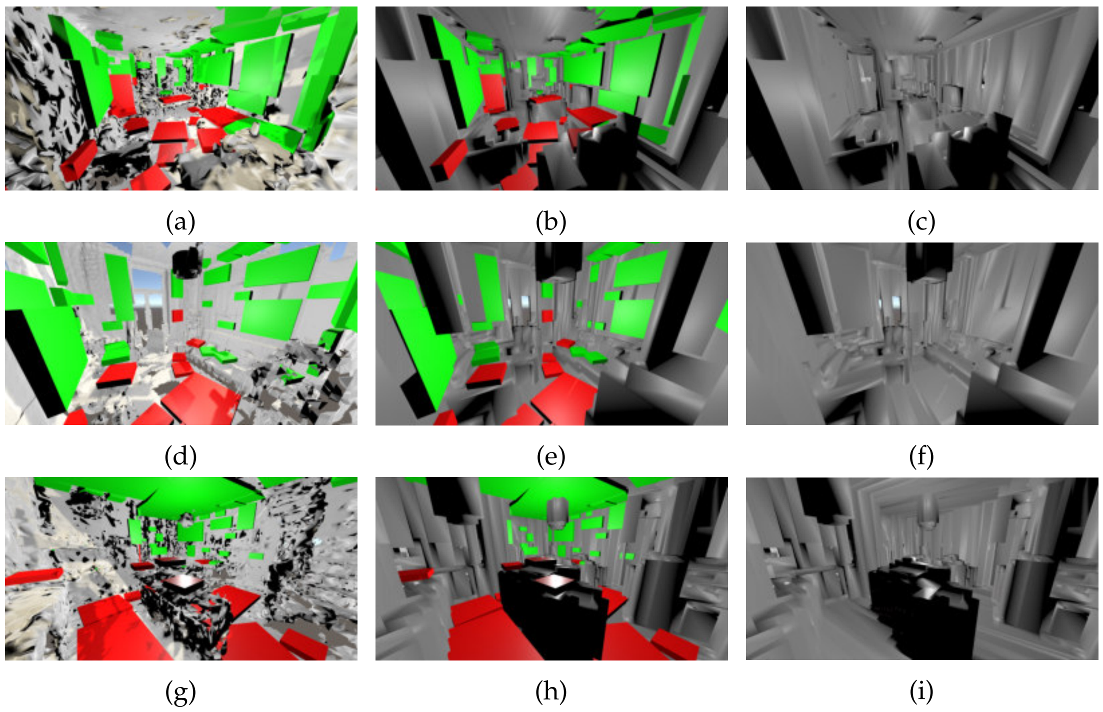


Figure 4.4.: The result of our algorithm implemented in Unity. The left column shows the room understanding results, where a red area indicates close interaction and green an interaction over a distance. The middle column depicts both algorithms results, while the right only renders the calculated simplified bounding volume. Only small areas of the detected surfaces overlap with the bounds and all important details are maintained. Images are screenshots taken within the Unity Editor.

implementation. In those images, we highlight the interaction type of each area, by colour coding those for close interaction in red and those for interaction over a distance in green. Most areas are categorised correctly. A few areas could have been labelled for close interaction, but instead received the *INTERACT_DISTANCE* label. While these categorizations are not wrong, they occur especially to areas located on a wall, where parts (especially the centre) are behind furniture and furniture areas, which are not located at the exact edge to a valid floor tile. Examples of those areas are in Figure 4.4a the green area on the bed and in Figure 4.4d the wall directly on the left and the spots on the sofa.

Another issue are the missed suitable locations on walls. These occur through holes or uneven surfaces in the scans and the wall feature detection algorithm did not accumulate enough vertical voxels for a wall recognition. A solution might be to use the created bounds voxel representation as a base for the feature detections. However, this will add another operation process resulting in longer execution times. The benefit of the resulting feature detection has to be evaluated in comparison to the increased execution time. We discovered that in room one and room two, the ceiling was not totally parallel to the X-Z-plane of our representation. The middle parts were a little lower compared to the boarders near the walls. This resulted in the detection of obstacles for ceiling areas in the middle and not suggesting the ceiling centres as suitable areas for interactive content. These shortcomings can both be either solved by factoring those edge cases into our algorithm or by making better scans a requirement for the algorithm. Both of the previous mentioned inaccuracy scenarios would disappear if the scans were constructed with less holes and flatter surfaces for walls and ceilings.

The large room covers a special case, which was not considered in our algorithm. It covers two floors and thus has four horizontal levels, with a high voxel count in our voxelized representation. By chance, this geometry achieved the highest voxel count for the floor and ceiling of the lower floor and thus allows our algorithm to analyse it correctly. However, for different scenarios and if this is an often occurring case, counter-measures to ensure the recognition of the correct floor and ceiling levels need to be taken into account within our proposed algorithm.

4.3.2. Evaluation of Room Reconstruction

For testing the execution time of the room recreation algorithm, the same technique is used for testing the room understanding. The recorded times across the three rooms and four versions can be found in Tables A.2, A.4, and A.6, with a visual representation of the average and range of those tables depicted in Graphic 4.5. As this algorithm needs a lot less operations, we can see a lot quicker execution times. The second room achieved times between 18 and 30 ms, which are 2-3 passed frames. Even the more complex room one recorded times between 27 and 37 ms, which is still under four passed frames. This would allow this algorithm to be executed between 10 and 40 times per second, depending on the room and CPU clock speed, making it faster than the mesh update rate from the HoloLens and would support the bounds calculation of the room in only tens of ms after a new mesh was generated. Even for the large room, we can generate a simplified bounding volume in only 165 to 195 ms, which will give a result in only 15-18 rendered frames. For the times across the four different versions, the same ranges as in Figure 4.3 can be observed, where only one barely missed delivery of the results can add a few more ms to the execution time. For room one and two, we always retrieved the results after the same frame times (31 and 22 ms respectively). This indicates, that a constant FPS adds to a more stable execution time.

For the room reconstruction algorithm, it is not only important how fast the algorithm can be executed, but also how many triangles and vertices are present in the final results, compared to the original count. For the HoloLens scanned rooms one and three, this reduced the count down to a twentieth of the original geometry. For the second room, even higher numbers were achieved. These can all be viewed in Table 4.3.

This drastic reduction of the vertex and triangle count shows that the created room bounds are highly optimized for collision meshes, compared to those provided by the original scans. At the cost of waiting only a few hundredths of a second, we deliver a

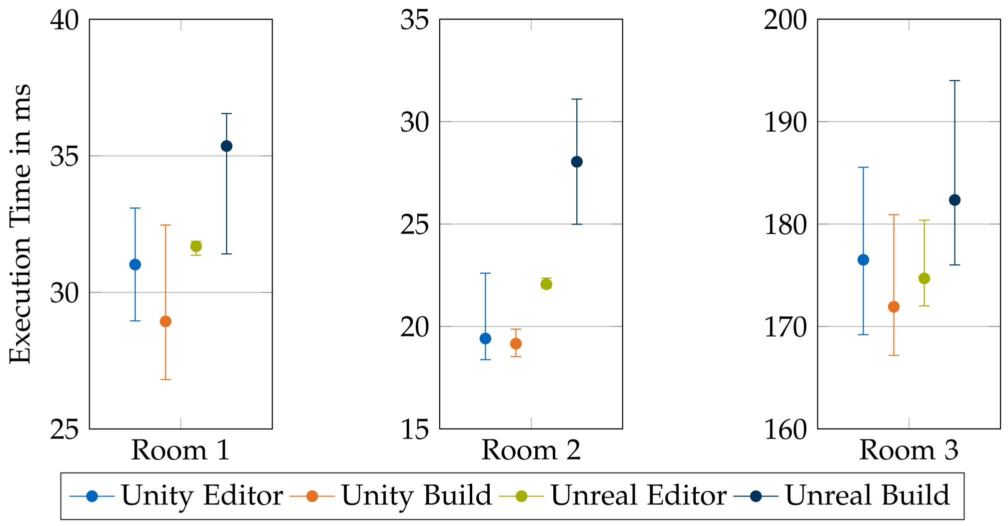


Figure 4.5.: The graphical representation of the average execution time for the room reconstruction algorithm measured in each room and tested version. The range of measured results is presented through the interval, while the average is the dot in between. Each room had very different execution times, which were mostly consistent over all tested versions.

		Room 1	**Room 2**	**Room 3**
Vertices	Original	78,430	332,088	232,097
	Reduced	4,770	2,524	11,664
Triangles	Original	131,178	110,696	385,207
	Reduced	6,812	3,608	16,702

Table 4.3.: Our room reconstruction algorithm is not only measured on its execution time, but also on the vertex and triangle count of the returned mesh. This count mainly depends on the complexity and size of the room. For the two rooms scanned by the HoloLens, we could reduce the triangle and vertex count to around a twentieth of the original count, while for room two, the vertex count was reduced down to less than a 130th of the original count.

boundary mesh which cuts the cost for collision checks drastically.

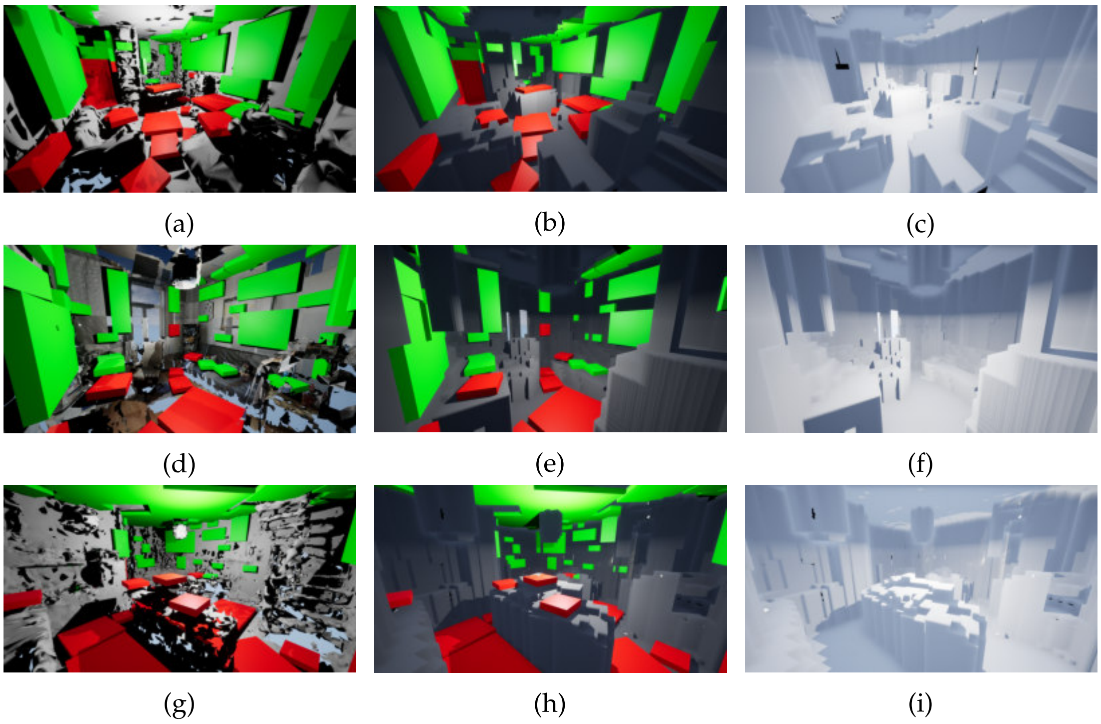


Figure 4.6.: The result of our algorithm implemented in the Unreal Engine. The left column shows the room understanding results, where a red area indicates close interaction and green an interaction over a distance. The middle column depicts both algorithms results, while the right only renders the calculated simplified bounding volume. Only small areas of the detected surfaces overlap with the bounds and all important details are maintained. Images are screenshots taken within the Unreal Engine Editor.

The visual results of the room recreation algorithm can be observed in the right column of Figure 4.4 and 4.6 for Unity and the Unreal Engine respectively. In the middle column, we combine both results of the room understanding and bounds algorithm. For the results, the original mesh geometry of the room was left active, so areas, where the bounds do not encompass the original room, can be spotted. Since these rare occurrences are mostly small areas, these are hard to detect. In Figure 4.6c and 4.6i, black spots can be observed in a few cases, especially for room one, where a larger hole is visible. These spots are all covered through edges of the original geometry, which connect different vertices with each other. As our algorithm has only these vertices to work with, it is practically impossible for it to surround these areas. As these faults are few in numbers and only occur in corners or holes within the walls, ceiling, and floor, these can be safely ignored. None of these are in easy to reach areas or large enough to be accidentally passed by a player unaware of his real surrounding.

Looking at the results, we can say that all important features are enclosed within our recreated room bounds and when looking at the combined results of both our algorithms, only few areas overlap with our valid surfaces. This is due to the fact, that we allow suitable spaces for interactive content to be detected, even if not a hundred percent of the results within the area are valid. We omit results, which intersect with walls or other defining features, which only results in slight overlaps. This fact has to be considered when placing interactive content, as there might still be a danger of a user accidentally touching real objects when interacting with a content placed slightly into the real environment.

Another point we can note is that despite some large holes existing in room one and two, our algorithm successfully filled them and created an almost continuous environment around the room. The largest issues we encountered are large holes within the walls, such as is the case with room two, where the back wall has a large open area, which probably comes from a window. Comparing this spot between Images 4.4d and 4.4f, we see that large portions could be closed and only two tiny holes remain. For floors and ceilings, we could close all holes successfully, if there was an existing floor or ceiling area beneath or above a hole in the ceiling or floor respectively. In room one, we could observe such a case, where this was not given, and noticed a small hole in both the floor and ceiling. This area was, however, in a niche, which was badly scanned and had a lot of noise. Moreover, it was also hard to be reached from the main room. Through better scan qualities, these errors should be completely omitted.

5. Conclusion

The following chapter concludes this thesis, by giving a summary of the achievements of this work. Afterwards, an outlook is given on future work to improve the performance and detected results of our approach, as well as further suggestions for testing the validity of our results.

5.1. Summary

The main contributions of this thesis are twofold. First, we provided an algorithm to detect and categorise the features of a room and translate them to suitable areas for interactive content. Our approach is, compared to existing solutions, hardware and software independent and can be adopted through its ensured compatibility throughout many different platforms, such as AR and VR headsets. Moreover, it can be used in other software and hardware solutions, which allow the usage of DLLs written in C++. This thesis proves that our suggested implementation can handle data of expected complexity and average room sizes in real-time, taking less than 200 ms to compute the valid areas on recommended VR PC hardware. With the targeted frame rate of 90 FPS for VR headsets, this translates to less than 20 passed frames until the results are available to be displayed for the user. Not blocking the main thread of the game engine, means that the application can continue to run smoothly while the results are being calculated. Comparing the maximum times recorded to the time it takes on average to blink an eye, which is between 100 and 400 ms, depending on the individual [Soma, 2018], our algorithm literally is finished in the blink of an eye. This is achieved by implementing a four-part algorithm, going in order from feature detection, suitable area definition, outcome reduction, to the categorization of the results.

Secondly, this thesis provides a solution for recreating room bounds custom-tailored to the needs of an AR and VR environment. This room recreation implementation does not need to recreate every exact detail present in the room, but rather the important boundaries keeping the user from running into real-world objects. Additionally it gives developers an idea of where objects should be placed to mirror the environment of the real room for recreating a new digital environment based on the real surroundings. Through execution times of less than 40 ms for an average room utilised for AR and VR environments, we also proved that this algorithm can be used to provide the recreated room in real-time to the application at least every five frames. Not even the HoloLens can update the room geometry in a faster time. This allows our algorithm to be used for continues room recreation updates, as well as informing the player when he is about to collide into real-world objects. This is achieved by combining the room feature detection with a greedy meshing algorithm.

By testing the proposed approaches over multiple rooms, we could determine its strengths and weaknesses. For one, we proved the deterministic nature of our algorithm and the reliable detection of suitable areas and room bounds, provided the passed mesh geometry has a floor and ceiling parallel to the X-Z-plane, walls are perpendicular to the floor, the mesh contains a uniform distribution of triangles and vertices across all surfaces, and the floor and ceiling are at least 2.3 m apart. Through the two tested rooms, where our method was unable to produce any results, we proved that our algorithm needs to be able to detect the floor and ceiling and will fail to provide any results if at least one of both is not detected correctly.

5.2. Outlook

While providing an important proof of concept and a great foundation, many aspects of the approaches developed in this thesis can be improved upon. The impact of certain features can be tested further, while also analysing the performance over different hardware configurations. Despite many existing approaches, which could have been used to achieve the same, if not better results, these would have limited the availability of this plugin to only a few platforms and would not have fulfilled our requirements to be easily deployable over a variety of different software and hardware solutions. In order to not increase the scope of this work unrealistically, we focused on providing a solid foundation, which can be easily extended, used on different hardware specifications, and proofs the possibility of achieving all of this with good results and in real-time. Nevertheless, the provided approaches here did an excellent job at detecting many different features and suitable areas, where some NNs cannot perform better detections in the same time period.

However, there are still open topics for future work. For one, our solution needs to be tested over different hardware and software setups and a performance evaluation, especially for the slower hardware of the HoloLens and HoloLens 2, needs to be done. This can provide important insights if our approach is fast enough to count as a real-time application even for those hardware specifications. Moreover, the approaches presented in this thesis were only tested on a limited set of room scans. A test by first scanning a room with the HoloLens or ZED mini and then analysing the room afterwards can also provide important insights into the performance in a productive environment.

Another aspect, which can be improved, are the special cases which currently do not work with our approach, such as roof pitch, lower scan qualities, and different vertex densities across the room. Nevertheless, new algorithms can be developed to encompass those into our solution. As such measures will increase the execution time of the methods, an evaluation of each algorithm's impact on the final delivered results should be done. Moreover, the speed of the execution time of our implementation can be improved upon and a better scalability for especially larger rooms can be developed. A few ideas for such achievements could be to only analyse the immediate surrounding of the user and ignore areas, which can currently not be seen by the headset wearer. This would drastically increase the scalability, as most parts of the model are then ignored and not considered anymore. We could also improve the feature detection results, by using our created bounds matrix instead of only using the unmodified voxelized representation of the room geometry.

The long term goal of this work is to provide a fast and easy to implement solution over a diverse portfolio of hardware and software solutions to recreate virtual environments based on the real surrounding. Moreover, content should be placed automatically on predefined areas found in the scanned environment. Our provided implementation is seen as the foundation of this process and more research and work has to be provided to achieve this goal in the future. Methods and algorithms have to be developed for recreating new virtual environments over the returned room bounds. These need to place objects based on their shape and dimension over the boundaries of the room in a natural and realistic structure, without occluding the open movement areas of the user.

The second part, placing interactive content automatically in suitable locations, needs more research to build on the provided foundation through this work. Currently, only the areas are returned, but to achieve an automatic placement, these areas also have to be analysed on their range to the player, depending on how often or from where the wished interaction should occur. Especially when combining the placement with the newly created virtual environment, the interactive content should fit into its surrounding smoothly and not stand out through an odd placement.

Nevertheless, this thesis laid the foundation for allowing AR and VR environments to be dynamically filled by interactive content and provide users with a cord-free experience, especially for VR headsets. These users cannot see their environment and our approach can provide the surrounding room bounds in real-time to the application in an optimized representation for rendering and collision queries.

A. Results from Room Evaluation

A.1. Room 1

Room Understanding - Room 1

Run Number	Unity Editor	Unity Build	Unreal Editor	Unreal Build
1	169.94	171.27	173.05	180.79
2	171.68	171.83	173.09	181.06
3	169.67	175.22	172.67	181.36
4	169.44	177.93	181.05	164.98
5	169.19	174.37	181.09	181.24
6	173.08	173.45	172.93	180.98
7	170.22	169.65	172.42	165.64
8	175.82	173.56	181.04	164.58
9	174.31	171.56	180.99	181.20
10	174.46	169.85	178.09	181.00
Average	171.78	172.87	176.64	176.28

Table A.1.: This table documents the measured execution time of the room understanding algorithm for room one from the initialization until the received results. This was done ten times in each case for the Unity and Unreal Engine editor and build versions respectively.

Room Bounds - Room 1

Run Number	Unity Editor	Unity Build	Unreal Editor	Unreal Build
1	31.01	28.19	31.37	36.29
2	31.32	29.13	31.49	32.03
3	29.99	32.47	31.83	31.40
4	31.85	27.83	31.69	36.54
5	30.53	26.81	31.66	36.36
6	30.38	29.04	31.77	36.35
7	33.05	28.18	31.84	36.00
8	30.03	29.81	31.60	36.35
9	33.09	28.82	31.81	36.07
10	28.96	29.13	31.88	36.18
Average	31.02	28.94	31.69	35.36

Table A.2.: This table documents the measured execution time of the room bounds algorithm for room one from the initialization until the received results. This was done ten times in each case for the Unity and Unreal Engine editor and build versions respectively.

A.2. Room 2

Room Understanding - Room 2

Run Number	Unity Editor	Unity Build	Unreal Editor	Unreal Build
1	58.42	56.59	55.20	58.64
2	57.85	52.72	55.15	58.84
3	58.43	54.93	56.17	63.92
4	61.75	54.42	54.98	61.00
5	58.30	54.33	55.10	58.66
6	56.21	53.95	55.08	58.54
7	57.63	53.84	55.00	58.50
8	56.88	52.57	55.52	58.70
9	58.15	53.06	55.31	58.63
10	58.28	54.15	55.49	58.76
Average	58.19	54.06	55.30	59.42

Table A.3.: This table documents the measured execution time of the room understanding algorithm for room two from the initialization until the received results. This was done ten times in each case for the Unity and Unreal Engine editor and build versions respectively.

Room Bounds - Room 2

Run Number	Unity Editor	Unity Build	Unreal Editor	Unreal Build
1	18.38	19.19	22.04	31.10
2	19.50	19.26	21.90	30.58
3	18.78	18.64	22.12	25.31
4	19.53	18.74	22.02	24.99
5	18.80	19.87	21.96	30.92
6	19.08	19.58	22.11	30.31
7	19.13	19.18	22.18	26.42
8	19.31	19.42	21.92	25.36
9	22.60	19.18	22.36	30.14
10	19.02	18.53	22.03	25.31
Average	19.41	19.16	22.06	28.04

Table A.4.: This table documents the measured execution time of the room bounds algorithm for room two from the initialization until the received results. This was done ten times in each case for the Unity and Unreal Engine editor and build versions respectively.

A.3. Room 3

Room Understanding - Room 3

Run Number	Unity Editor	Unity Build	Unreal Editor	Unreal Build
1	1486.46	1442.07	1416.04	1443.88
2	1433.58	1461.28	1446.89	1410.58
3	1416.91	1430.66	1413.64	1443.82
4	1424.97	1432.09	1419.78	1393.72
5	1456.66	1462.13	1438.53	1393.67
6	1435.34	1466.19	1397.05	1460.03
7	1438.55	1430.18	1430.28	1393.90
8	1475.50	1423.48	1425.29	1427.70
9	1448.97	1444.48	1424.00	1410.79
10	1425.29	1433.84	1419.12	1427.40
Average	1444.22	1442.64	1423.06	1420.55

Table A.5.: This table documents the measured execution time of the room understanding algorithm for room three from the initialization until the received results. This was done ten times in each case for the Unity and Unreal Engine editor and build versions respectively.

Room Bounds - Room 3

Run Number	Unity Editor	Unity Build	Unreal Editor	Unreal Build
1	181.08	169.00	172.06	176.01
2	185.54	179.50	180.28	177.28
3	179.09	180.91	180.38	178.79
4	177.63	170.63	172.19	177.23
5	181.90	168.97	173.51	177.33
6	173.51	167.18	172.03	193.94
7	169.20	173.14	172.02	194.01
8	174.78	170.55	180.28	177.55
9	169.29	171.87	172.07	193.98
10	173.04	167.44	172.13	177.36
Average	176.51	171.92	174.69	182.35

Table A.6.: This table documents the measured execution time of the room bounds algorithm for room three from the initialization until the received results. This was done ten times in each case for the Unity and Unreal Engine editor and build versions respectively.

Abbreviations

2D	two-dimensional
3D	three-dimensional
AI	artificial intelligence
API	application programming interface
AR	augmented reality
BIM	Building Information Model
CAD	computer-aided design
cm	centimetre
CNN	Convolutional Neural Network
CPU	central processing unit
DLL	dynamic-link library
DNN	Deep Neural Network
FOV	field of view
FPS	
	frames per second
	first-person shooter
GB	gigabyte
GHz	gigahertz
GPS	Global Positioning System
GPU	graphics processing unit
HMD	head-mounted display
HPU	Holographic Processing Unit
IR	infrared radiation
LiDAR	light detection and ranging
LTS	Long Term Support

m	metre
MR	mixed reality
MRTK	Mixed Reality Toolkit
ms	millisecond
MVS	Multi-View Stereo
MW	Manhattan-World
NN	Neural Network
OS	operating system
PC	personal computer
PCD	Point Cloud Data
RAM	random access memory
RGB	red, green, and blue
RGB-D	red, green, blue, and depth
RGBA	red, green, blue, and alpha
SDK	software development kit
SfM	Structure from Motion
SLAM	Simultaneous Localization and Mapping
ToF	Time of Flight
UWP	Universal Windows Platform
VR	virtual reality
WMR	Windows Mixed Reality

List of Figures

List of Tables

Bibliography

[Adan, 2011] A. Adan and D. Huber. "3D Reconstruction of Interior Wall Surfaces under Occlusion and Clutter." In: *2011 International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission*. May 2011, pp. 275–281. DOI: 10.1109/3DIMPVT.2011.42.

[Amazon, 2020] Amazon. *Amazon Lumberyard*. Amazon Web Services, Inc. 2020. URL: https://aws.amazon.com/lumberyard/ (visited on 02/21/2020).

[Anagnostopoulos, 2016] I. Anagnostopoulos, V. Pătrăucean, I. Brilakis, and P. Vela. "Detection of Walls, Floors, and Ceilings in Point Cloud Data." In: *Construction Research Congress 2016*. American Society of Civil Engineers, 2016, pp. 2302–2311. DOI: 10.1061/9780784479827.229. eprint: https://ascelibrary.org/doi/pdf/10.1061/9780784479827.229.

[Avetisyan, 2019a] A. Avetisyan, M. Dahnert, A. Dai, M. Savva, A. X. Chang, and M. Niessner. "Scan2CAD: Learning CAD Model Alignment in RGB-D Scans." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[Avetisyan, 2019b] A. Avetisyan, A. Dai, and M. Niessner. "End-to-End CAD Model Retrieval and 9DoF Alignment in 3D Scans." In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2019.

[Axon, 2016] S. Axon. *Unity at 10: For better—or worse—game development has never been easier*. Condé Nast Inc. Sept. 27, 2016. URL: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/ (visited on 02/22/2020).

[Azad, 2019] K. Azad. *Vector Calculus: Understanding the Dot Product*. Better Explained. 2019. URL: https://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/ (visited on 04/05/2020).

[Bærentzen, 2012] J. A. Bærentzen, J. Gravesen, F. Anton, and H. Aanæs. "Polygonal Meshes." In: *Guide to Computational Geometry Processing*. Springer, 2012, pp. 83–97.

[Batchelor, 2016] J. Batchelor. *Unity dropping major updates in favour of date-based model*. Gamer Network. Dec. 14, 2016. URL: `https://www.gamesindustry.biz/articles/2016-12-14-unity-dropping-major-updates-in-favour-of-date-based-model` (visited on 02/22/2020).

[Becker, 2015] S. Becker, M. Peter, and D. Fritsch. "Grammar-supported 3D indoor reconstruction from point clouds for "as-built" BIM." In: *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences* 2 (2015).

[Benchmark, 2020a] Benchmark. *Compare Intel Core i7-5820K vs Intel Core i5-4590*. UserBenchmark. 2020. URL: `https://cpu.userbenchmark.com/Compare/Intel-Core-i7-5820K-vs-Intel-Core-i5-4590/2579vs2604` (visited on 04/11/2020).

[Benchmark, 2020b] Benchmark. *Compare Nvidia GTX 980 vs Nvidia GTX-6GB*. UserBenchmark. 2020. URL: `https://gpu.userbenchmark.com/Compare/Nvidia-GTX-980-vs-Nvidia-GTX-970/2576vs2577` (visited on 04/11/2020).

[Bourke, 1997] P. Bourke. *Implicit surfaces*. Paul Bourke. June 1997. URL: `http://paulbourke.net/geometry/implicitsurf/` (visited on 02/23/2020).

[Burgard, 2012] W. Burgard, C. Stachniss, K. Arras, and M. Bennewitz. *SLAM: Simultaneous Localization and Mapping*. Uni Freiburg. 2012. URL: `http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/12-slam.pdf` (visited on 03/02/2020).

[CC, 2020] CC. *Attribution 4.0 International*. Creative Commons. 2020. URL: `https://creativecommons.org/licenses/by/4.0/legalcode` (visited on 04/11/2020).

[CD Projekt, 2020] CD Projekt. *CD Projekt RED*. CD PROJEKT S.A. 2020. URL: `https://en.cdprojektred.com/` (visited on 02/21/2020).

[Chacos, 2020] B. Chacos. *Microsoft's Xbox Series X probably puts your gaming PC to shame: Full specs revealed*. IDG Communications, Inc. Mar. 17, 2020. URL: `https://www.pcworld.com/article/3532810/microsofts-xbox-series-x-full-specs-vs-gaming-pc.html` (visited on 04/11/2020).

[Cherdo, 2020] L. Cherdo. *The 10 best PC VR headsets of 2020*. Aniwaa Pte. Ltd. Jan. 13, 2020. URL: `https://www.aniwaa.com/best-of/vr-ar/best-pc-vr-headset-tethered-vr/` (visited on 02/17/2020).

[Cohen, 2007]	P. Cohen. *Unity 2.0 game engine now available*. IDG Communications, Inc. Oct. 10, 2007. URL: `https://www.macworld.com/article/1060484/unity.html` (visited on 02/22/2020).
[Dai, 2018]	A. Dai, D. Ritchie, M. Bokeloh, S. Reed, J. Sturm, and M. Nießner. "ScanComplete: Large-Scale Scene Completion and Semantic Segmentation for 3D Scans." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.
[Dai, 2019]	A. Dai and M. Niessner. "Scan2Mesh: From Unstructured Range Scans to 3D Meshes." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.
[Delta, 2020]	Delta. *Laser Scanning Overview*. Delta Engineers, Architects, & Land Surveyors, DPC. 2020. URL: `http://delta-eas.com/laser_scanning/` (visited on 03/02/2020).
[EA, 2020]	EA. *Empowering game creators to shape the future of gaming*. Electronic Arts Inc. 2020. URL: `https://www.ea.com/frostbite/engine` (visited on 02/21/2020).
[Ennafii, 2018]	O. Ennafii, A. Le-Bris, F. Lafarge, and C. Mallet. "Semantic evaluation of 3D city models." working paper or preprint. Sept. 2018.
[Epic Games, 2018]	Epic Games. *How to Link External C Libraries .dll .lib With Your Project & Package With Game, Fast And Easy*. Epic Games, Inc. June 28, 2018. URL: `https://wiki.unrealengine.com/How_to_Link_External_C_Libraries_.dll_.lib_With_Your_Project_%26_Package_With_Game,_Fast_And_Easy` (visited on 02/21/2020).
[Epic Games, 2020a]	Epic Games. *FBX Static Mesh Pipeline*. Epic Games, Inc. 2020. URL: `https://docs.unrealengine.com/en-US/Engine/Content/Importing/FBX/StaticMeshes/index.html` (visited on 02/29/2020).
[Epic Games, 2020b]	Epic Games. *Features*. Epic Games, Inc. 2020. URL: `https://www.unrealengine.com/en-US/features` (visited on 02/21/2020).
[Epic Games, 2020c]	Epic Games. *Make something Unreal*. Epic Games, Inc. 2020. URL: `https://www.unrealengine.com/en-US/` (visited on 02/21/2020).
[Epic Games, 2020d]	Epic Games. *Unreal Engine 4.23 Release Notes*. Epic Games, Inc. 2020. URL: `https://docs.unrealengine.com/en-US/Support/Builds/ReleaseNotes/4_23/index.html` (visited on 02/22/2020).

[Evans, 2018] T. Evans. *Leap Into VR/AR Design*. Toptal, LLC. 2018. URL: https://www.toptal.com/designers/ui/vr-ar-design-guide (visited on 02/17/2020).

[Fidel, 2019] A. Fidel. *Using Spatial Networks: Social VR & Social AR for Education & Remote Teaching*. A Medium Corporation. July 16, 2019. URL: https://arvrjourney.com/future-of-education-remote-teaching-in-social-vr-1c836df82274 (visited on 02/17/2020).

[Foley, 1996] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, J. F. Hughes, E. Angel, and J. Hughes. *Computer graphics: principles and practice*. Vol. 12110. Addison-Wesley Professional, 1996.

[Geeks, 2020] Geeks. *How to check if two given line segments intersect?* geeksforgeeks. 2020. URL: https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/ (visited on 04/05/2020).

[Girard, 2010] D. Girard. *Unity 3 brings very expensive dev tools at a very low price*. Condé Nast Inc. Sept. 28, 2010. URL: https://arstechnica.com/information-technology/2010/09/unity-3-brings-very-expensive-dev-tools-at-a-very-low-price/ (visited on 02/22/2020).

[Greenwald, 2020] W. Greenwald. *The Best VR Headsets for 2020*. ZiffDavis, LLC. Feb. 13, 2020. URL: https://uk.pcmag.com/virtual-reality/75926/the-best-vr-headsets (visited on 02/17/2020).

[Gregory, 2019] J. Gregory. *Game Engine Architecture*. 3rd. New York: A K Peters/CRC Press, 2019. DOI: https://doi.org/10.1201/9781315267845.

[Hackster, 2017] Hackster. *3D Scan a Room With This LIDAR Rig*. Avnet, Inc. 2017. URL: https://www.hackster.io/news/3d-scan-a-room-with-this-lidar-rig-a630c1a65087 (visited on 03/02/2020).

[Hansen, 2011] C. D. Hansen and C. R. Johnson. *Visualization handbook*. Elsevier, 2011.

[Hoiem, 2017a] D. Hoiem. *Human Body Recognition and Tracking: How the Kinect RGB-D Camera Works*. University of Wisconsin. 2017. URL: http://pages.cs.wisc.edu/~dyer/cs534/slides/17_kinect.pdf (visited on 03/02/2020).

[Hoiem, 2017b] D. Hoiem. *Structure from Motion*. University of Illinois. Mar. 7, 2017. URL: https://courses.engr.illinois.edu/cs543/sp2017/lectures/Lecture%2015%20-%20Structure%20from%20Motion%20-%20Vision_Spring2017.pdf (visited on 03/02/2020).

[Hong, 2015] S. Hong, J. Jung, S. Kim, H. Cho, J. Lee, and J. Heo. "Semi-automated approach to indoor mapping for 3D as-built building information modeling." In: *Computers, Environment and Urban Systems* 51 (2015), pp. 34–46. ISSN: 0198-9715. DOI: https://doi.org/10.1016/j.compenvurbsys.2015.01.005.

[Horvath, 2012] S. Horvath. *The Imagination Engine: Why Next-Gen Videogames Will Rock Your World*. Condé Nast Inc. May 17, 2012. URL: https://www.wired.com/2012/05/ff-unreal4/ (visited on 02/21/2020).

[Hou, 2019a] J. Hou, A. Dai, and M. Nießner. "3D-SIC: 3D Semantic Instance Completion for RGB-D Scans." In: *CoRR* abs/1904.12012 (2019). eprint: 1904.12012.

[Hou, 2019b] J. Hou, A. Dai, and M. Niessner. "3D-SIS: 3D Semantic Instance Segmentation of RGB-D Scans." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[Huang, 2018] Q. Huang, W. Wang, and U. Neumann. "Recurrent Slice Networks for 3D Segmentation of Point Clouds." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[Hudlicka, 2009] E. Hudlicka. "Affective game engines: motivation and requirements." In: *Proceedings of the 4th international conference on foundations of digital games*. ACM. 2009, pp. 299–306.

[IDC, 2020] IDC. *AR & VR Headsets Market Share*. IDC Corporate. Jan. 20, 2020. URL: https://www.idc.com/promo/arvr (visited on 02/17/2020).

[Jiang, 2018] M. Jiang, Y. Wu, and C. Lu. "PointSIFT: A SIFT-like Network Module for 3D Point Cloud Semantic Segmentation." In: *CoRR* abs/1807.00652 (2018). eprint: 1807.00652.

[Jung, 2014] J. Jung, S. Hong, S. Jeong, S. Kim, H. Cho, S. Hong, and J. Heo. "Productive modeling for development of as-built BIM of existing indoor structures." In: *Automation in Construction* 42 (2014), pp. 68–77. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2014.02.021.

[Khronos, 2013] Khronos. *Calculating a Surface Normal*. The Khronos Group Inc. Jan. 13, 2013. URL: `https://www.khronos.org/opengl/wiki/Calculating_a_Surface_Normal` (visited on 04/05/2020).

[Kim, 2019] Kim. *Top 34 AR/VR Development Tools and SDKs You Should Know About*. Blue Whale Apps. May 28, 2019. URL: `https://bluewhaleapps.com/blog/top-ar-vr-development-tools-and-sdks-you-should-know-about` (visited on 02/17/2020).

[Kuredjian, 2017] S. Kuredjian. *Static Libraries vs. Dynamic Libraries*. A Medium Corporation. May 15, 2017. URL: `https://medium.com/@StueyGK/static-libraries-vs-dynamic-libraries-af78f0b5f1e4` (visited on 03/04/2020).

[Lafarge, 2012] F. Lafarge and C. Mallet. "Creating Large-Scale City Models from 3D-Point Clouds: A Robust Approach with Hybrid Representation." In: *International Journal of Computer Vision* 99.1 (Aug. 2012), pp. 69–85. ISSN: 1573-1405. DOI: `10.1007/s11263-012-0517-8`.

[Lang, 2019] B. Lang. *How to Tell if Your PC is VR Ready*. Road to VR. Nov. 19, 2019. URL: `https://www.roadtovr.com/how-to-tell-pc-virtual-reality-vr-oculus-rift-htc-vive-steam-vr-compatibility-tool/` (visited on 04/11/2020).

[LES, 2020] LES. *GeoSLAM ZEB-CAM*. Levelling Equipment Services. 2020. URL: `https://www.lesirl.ie/products/geoslam-zeb-revo-with-zeb-cam` (visited on 03/02/2020).

[Liu, 2017] F. Liu, S. Li, L. Zhang, C. Zhou, R. Ye, Y. Wang, and J. Lu. "3DCNN-DQN-RNN: A Deep Reinforcement Learning Framework for Semantic Parsing of Large-Scale 3D Point Clouds." In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[Lowood, 2014] H. Lowood. "Game Engines and Game History." In: *Kinephanos (History of Games International Conference Proceedings)*. 2014.

[LSE, 2020] LSE. *Laser Scanning Overview*. Laserscanning Europe GmbH. 2020. URL: `https://www.laserscanning-europe.com/en/3d-effects-games-movies` (visited on 03/02/2020).

[Lysenko, 2012] M. Lysenko. *Meshing in a Minecraft Game*. Automattic Inc. June 30, 2012. URL: `https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/` (visited on 04/06/2020).

[Magic Leap, 2020] Magic Leap. *Reality is just beginning*. Magic Leap, Inc. 2020. URL: `https://www.magicleap.com/` (visited on 02/17/2020).

[MathWorks, 2020a] MathWorks. *pcread*. The MathWorks, Inc. 2020. URL: https://www.mathworks.com/help/vision/ref/pcread.html (visited on 02/29/2020).

[MathWorks, 2020b] MathWorks. *Structure from Motion*. The MathWorks, Inc. 2020. URL: https://www.mathworks.com/help/vision/ug/structure-from-motion.html (visited on 03/02/2020).

[Maturana, 2015] D. Maturana and S. Scherer. "VoxNet: A 3D Convolutional Neural Network for real-time object recognition." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 922–928. DOI: 10.1109/IROS.2015.7353481.

[McLeroy, 2008] C. McLeroy. *Improving "America's Army"*. U.S. Army. Aug. 27, 2008. URL: https://www.army.mil/article/11935/improving_americas_army (visited on 02/21/2020).

[Microsoft, 2017] Microsoft. *Second version of HoloLens HPU will incorporate AI coprocessor for implementing DNNs*. Microsoft Corporation. July 23, 2017. URL: https://www.microsoft.com/en-us/research/blog/second-version-hololens-hpu-will-incorporate-ai-coprocessor-implementing-dnns/ (visited on 02/15/2020).

[Microsoft, 2018a] Microsoft. *HoloLens Research mode*. Microsoft Corporation. May 3, 2018. URL: http://pages.cs.wisc.edu/~dyer/cs534/slides/17_kinect.pdf (visited on 03/02/2020).

[Microsoft, 2018b] Microsoft. *HoloToolkit.SpatialUnderstanding*. Microsoft Corporation. Nov. 7, 2018. URL: https://github.com/microsoft/MixedRealityToolkit/wiki/HoloToolkit.SpatialUnderstanding (visited on 02/15/2020).

[Microsoft, 2018c] Microsoft. *Spatial mapping*. Microsoft Corporation. Mar. 21, 2018. URL: https://docs.microsoft.com/en-gb/windows/mixed-reality/spatial-mapping (visited on 02/14/2020).

[Microsoft, 2019a] Microsoft. *MixedRealityToolkit (MRTK)*. Microsoft Corporation. Sept. 2, 2019. URL: https://github.com/microsoft/MixedRealityToolkit (visited on 02/15/2020).

[Microsoft, 2019b] Microsoft. *Scene understanding*. Microsoft Corporation. July 8, 2019. URL: https://docs.microsoft.com/en-gb/windows/mixed-reality/scene-understanding (visited on 02/15/2020).

[Microsoft, 2019c] Microsoft. *Scene understanding SDK overview*. Microsoft Corporation. July 8, 2019. URL: https://docs.microsoft.com/en-gb/windows/mixed-reality/scene-understanding-sdk (visited on 02/15/2020).

[Microsoft, 2019d] Microsoft. *Walkthrough: Create and use your own Dynamic Link Library (C++)*. Microsoft Corporation. Aug. 22, 2019. URL: https://docs.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=vs-2019 (visited on 03/04/2020).

[Microsoft, 2019e] Microsoft. *Walkthrough: Creating and Using a Static Library (C++)*. Microsoft Corporation. Apr. 25, 2019. URL: https://docs.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-static-library-cpp?view=vs-2019 (visited on 03/04/2020).

[Microsoft, 2020a] Microsoft. *HoloLens 2*. Microsoft Corporation. 2020. URL: https://www.microsoft.com/en-us/hololens (visited on 02/17/2020).

[Microsoft, 2020b] Microsoft. *IntPtr Struct*. Microsoft Corporation. 2020. URL: https://docs.microsoft.com/en-us/dotnet/api/system.intptr?view=netframework-4.8 (visited on 03/09/2020).

[Microsoft, 2020c] Microsoft. *What is the Mixed Reality Toolkit*. Microsoft Corporation. Feb. 15, 2020. URL: https://github.com/Microsoft/MixedRealityToolkit-Unity (visited on 02/15/2020).

[Mohabia, 2018] S. Mohabia. *Bitwise operators — Facts and Hacks*. A Medium Corporation. May 30, 2018. URL: https://medium.com/@shashankmohabia/bitwise-operators-facts-and-hacks-903ca516f28c (visited on 03/28/2020).

[Niessner, 2018a] M. Niessner. *3D Scanning & Motion Capture. 3D Concepts and Sensors*. University Lecture. Technische Universität München. 2018. URL: https://www.moodle.tum.de/pluginfile.php/1421973/mod_resource/content/1/1_3DConcepts.pdf (visited on 02/23/2020).

[Niessner, 2018b] M. Niessner. *3D Scanning & Motion Capture. Surface Representations*. University Lecture. Technische Universität München. 2018. URL: https://www.moodle.tum.de/pluginfile.php/1426345/mod_resource/content/1/2_Surface_Representations.pdf (visited on 02/23/2020).

[Niessner, 2018c] M. Niessner. *3D Scanning & Motion Capture. Overview of 3D reconstruction methods*. University Lecture. Technische Universität München. 2018. URL: https://www.moodle.tum.de/pluginfile.php/1429844/mod_resource/content/1/3_Reconstruction_Overview.pdf (visited on 03/02/2020).

[NOAA, 2020] NOAA. *What is LIDAR?* National Oceanic and Atmospheric Administration. Jan. 7, 2020. URL: https://oceanservice.noaa.gov/facts/lidar.html (visited on 03/02/2020).

[Ochmann, 2016] S. Ochmann, R. Vock, R. Wessel, and R. Klein. "Automatic reconstruction of parametric building models from indoor point clouds." In: *Computers & Graphics* 54 (2016). Special Issue on CAD/Graphics 2015, pp. 94–103. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2015.07.008.

[Oomes, 1997] S. Oomes, P. Snoeren, and T. Dijkstra. "3D shape representation: Transforming polygons into voxels." In: *International Conference on Scale-Space Theories in Computer Vision*. Springer. 1997, pp. 349–352.

[Özdemir, 2018] E. Özdemir and F. Remondino. "Segmentation of 3d Photogrammetric Point Cloud for 3d Building Modeling." In: *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences* (2018).

[Paul, 2012] P. S. Paul, S. Goon, and A. Bhattacharya. "History and comparative study of modern game engines." In: *International Journal of Advanced Computed and Mathematical Sciences* 3.2 (2012), pp. 245–249.

[Petrock, 2019] V. Petrock. *Virtual and Augmented Reality Users 2019*. eMarketer Inc. Mar. 27, 2019. URL: https://www.emarketer.com/content/virtual-and-augmented-reality-users-2019 (visited on 02/17/2020).

[Pickton, 2012] M. Pickton. *What is a voxel, anyway? Voxels vs. Vertexes in Games*. GamersNexus, LLC. Mar. 6, 2012. URL: https://www.gamersnexus.net/gg/762-voxels-vs-vertexes-in-games (visited on 02/29/2020).

[Purwar, 2019] S. Purwar. *Designing User Experience for Virtual Reality (VR) applications*. A Medium Corporation. Mar. 4, 2019. URL: https://uxplanet.org/designing-user-experience-for-virtual-reality-vr-applications-fc8e4faadd96 (visited on 02/17/2020).

[Qi, 2016] C. R. Qi, H. Su, M. Niessner, A. Dai, M. Yan, and L. J. Guibas. “Volumetric and Multi-View CNNs for Object Classification on 3D Data.” In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[Qi, 2019] C. R. Qi, O. Litany, K. He, and L. J. Guibas. “Deep Hough Voting for 3D Object Detection in Point Clouds.” In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2019.

[Reed, 2004] K. Reed. *Unreal Engine 3*. Gamer Network. July 2, 2004. URL: `https://www.eurogamer.net/articles/i_epicgames_june04` (visited on 02/21/2020).

[Robertson, 2015] A. Robertson. *Unity officially releases its new game engine: Unity 5*. Vox Media, LLC. Mar. 3, 2015. URL: `https://www.theverge.com/2015/3/3/8142099/unity-5-engine-release` (visited on 02/22/2020).

[Rouse, 2016] M. Rouse. *point cloud*. TechTarget, Inc. Oct. 2016. URL: `https://whatis.techtarget.com/definition/point-cloud` (visited on 02/29/2020).

[Sanchez, 2012] V. Sanchez and A. Zakhor. “Planar 3D modeling of building interiors from point cloud data.” In: *2012 19th IEEE International Conference on Image Processing*. Sept. 2012, pp. 1777–1780. DOI: `10.1109/ICIP.2012.6467225`.

[Schonberger, 2016] J. L. Schonberger and J.-M. Frahm. “Structure-From-Motion Revisited.” In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[Sketchfab, 2017a] Sketchfab. *Home*. Sketchfab. Mar. 14, 2017. URL: `https://sketchfab.com/3d-models/home-8975b37e9d4f453983a5728664aecb23` (visited on 04/11/2020).

[Sketchfab, 2017b] Sketchfab. *My Studio via Hololens*. Sketchfab. Mar. 15, 2017. URL: `https://sketchfab.com/3d-models/my-studio-via-hololens-73d59a432e3b48b5be8902f99b34b2b6` (visited on 04/11/2020).

[Sketchfab, 2017c] Sketchfab. *New Year’s on the Verge of Stokes Croft*. Sketchfab. Jan. 11, 2017. URL: `https://sketchfab.com/3d-models/new-years-on-the-verge-of-stokes-croft-a327090936db421ba966c96152a8c72a` (visited on 04/11/2020).

[Sketchfab, 2017d] Sketchfab. *Office room scan*. Sketchfab. Jan. 2, 2017. URL: `https://sketchfab.com/3d-models/office-room-scan-c97168ba159c402593ff502ad023f66d` (visited on 04/11/2020).

[Sketchfab, 2017e] Sketchfab. *White Hart House*. Sketchfab. Jan. 12, 2017. URL: `https://sketchfab.com/3d-models/white-hart-house-7f4084ed27634b0596061a6f9811cb64` (visited on 04/11/2020).

[Smith, 2019] F. Smith. *How to Reload Native Plugins in Unity*. Forrest Smith. Sept. 2, 2019. URL: `https://www.forrestthewoods.com/blog/how-to-reload-native-plugins-in-unity/` (visited on 02/23/2020).

[Soma, 2018] Soma. *How Fast is the Average Blink?* Soma Tech Intl. Apr. 12, 2018. URL: `https://www.somatechnology.com/blog/thursday-thoughts/fast-average-blink/` (visited on 04/12/2020).

[Stachniss, 2016] C. Stachniss, J. J. Leonard, and S. Thrun. "Simultaneous localization and mapping." In: *Springer Handbook of Robotics*. Springer, 2016, pp. 1153–1176.

[Stereolabs, 2020a] Stereolabs. *Bring your imagination to life*. Stereolabs Inc. 2020. URL: `https://www.stereolabs.com/zed-mini/` (visited on 02/17/2020).

[Stereolabs, 2020b] Stereolabs. *Depth Sensing Overview*. Stereolabs Inc. 2020. URL: `https://www.stereolabs.com/docs/depth-sensing/` (visited on 03/02/2020).

[Stereolabs, 2020c] Stereolabs. *Spatial Mapping Overview*. Stereolabs Inc. 2020. URL: `https://www.stereolabs.com/docs/spatial-mapping/` (visited on 02/14/2020).

[Sweeney, 2014] T. Sweeney. *Welcome to Unreal Engine 4*. Epic Games, Inc. Mar. 19, 2014. URL: `https://www.unrealengine.com/en-US/blog/welcome-to-unreal-engine-4` (visited on 02/21/2020).

[Tach, 2012] D. Tach. *Unity 4.0 available for download today with DX 11 support and Linux preview*. Vox Media, LLC. Nov. 14, 2012. URL: `https://www.polygon.com/2012/11/14/3645122/unity-4-0-available-download` (visited on 02/22/2020).

[Tchapmi, 2017] L. Tchapmi, C. Choy, I. Armeni, J. Gwak, and S. Savarese. "SEGCloud: Semantic Segmentation of 3D Point Clouds." In: *2017 International Conference on 3D Vision (3DV)*. Oct. 2017, pp. 537–547. DOI: `10.1109/3DV.2017.00067`.

[Thomson, 2015] C. Thomson and J. Boehm. "Automatic Geometry Generation from Point Clouds for BIM." In: *Remote Sensing* 7.9 (2015), pp. 11753–11775. ISSN: 2072-4292. DOI: 10.3390/rs70911753.

[Tuliper, 2019] A. Tuliper. *Introduction to the HoloLens, Part 2: Spatial Mapping*. Microsoft Corporation. Jan. 25, 2019. URL: https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/january/hololens-introduction-to-the-hololens-part-2-spatial-mapping (visited on 04/11/2020).

[UGC, 2020] UGC. *Alt- und Neubau: Deckenhöhe*. UGC GmbH. 2020. URL: https://www.wohnung.com/ratgeber/418/alt-und-neubau-deckenhoehe (visited on 03/25/2020).

[Unity, 2010] Unity. *Transforms*. Unity Technologies. Feb. 24, 2010. URL: https://docs.unity3d.com/Manual/Transforms.html (visited on 03/09/2020).

[Unity, 2019] Unity. *Managed plug-ins*. Unity Technologies. Feb. 14, 2019. URL: https://docs.unity3d.com/Manual/UsingDLL.html (visited on 02/21/2020).

[Unity, 2020a] Unity. *Build once, deploy anywhere, captivate everyone*. Unity Technologies. 2020. URL: https://unity3d.com/unity/features/multiplatform (visited on 02/22/2020).

[Unity, 2020b] Unity. *Example - Creating a quad*. Unity Technologies. Mar. 31, 2020. URL: https://docs.unity3d.com/Manual/Example-CreatingaBillboardPlane.html (visited on 04/06/2020).

[Unity, 2020c] Unity. *Extend the power of the world's most performant real-time development platform*. Unity Technologies. 2020. URL: https://unity.com/products (visited on 02/22/2020).

[Unity, 2020d] Unity. *Geometry in Unity*. Unity Technologies. 2020. URL: https://learn.unity.com/tutorial/geometry-in-unity (visited on 02/29/2020).

[Unity, 2020e] Unity. *Native plug-ins*. Unity Technologies. Mar. 31, 2020. URL: https://docs.unity3d.com/Manual/NativePlugins.html (visited on 04/07/2020).

[Unity, 2020f] Unity. *Unity 2019.3.0*. Unity Technologies. 2020. URL: https://unity3d.com/unity/whats-new/2019.3.0 (visited on 02/22/2020).

[Unity, 2020g] Unity. *Unity Core Platform*. Unity Technologies. 2020. URL: https://unity.com/products/core-platform (visited on 02/21/2020).

[Unity, 2020h]	Unity. *Unity Core Platform*. Unity Technologies. 2020. URL: `https://unity.com/products/core-platform` (visited on 02/22/2020).
[Wang, 2018]	W. Wang, R. Yu, Q. Huang, and U. Neumann. "SGPN: Similarity Group Proposal Network for 3D Point Cloud Instance Segmentation." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.
[Wasser, 2020]	L. A. Wasser. *The Basics of LiDAR - Light Detection and Ranging - Remote Sensing*. Battelle. 2020. URL: `https://www.neonscience.org/lidar-basics` (visited on 03/02/2020).
[Wienand, 2019]	I. Wienand. *Computer Science from the Bottom Up. Chapter 2. Binary and Number Representation*. 2019. URL: `https://www.bottomupcs.com/chapter01.xhtml` (visited on 03/28/2020).
[Wikimedia, 2019a]	Wikimedia. *Heightmap*. Wikimedia Foundation, Inc. Aug. 25, 2019. URL: `https://en.wikipedia.org/wiki/Heightmap` (visited on 02/23/2020).
[Wikimedia, 2019b]	Wikimedia. *Non-uniform rational B-spline*. Wikimedia Foundation, Inc. Dec. 27, 2019. URL: `https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline` (visited on 02/23/2020).
[Wikimedia, 2020a]	Wikimedia. *Constructive solid geometry*. Wikimedia Foundation, Inc. Feb. 2, 2020. URL: `https://en.wikipedia.org/wiki/Constructive_solid_geometry` (visited on 02/23/2020).
[Wikimedia, 2020b]	Wikimedia. *Polygon mesh*. Wikimedia Foundation, Inc. Jan. 7, 2020. URL: `https://en.wikipedia.org/wiki/Polygon_mesh` (visited on 02/29/2020).
[Wu, 2019]	W. Wu, Z. Qi, and L. Fuxin. "PointConv: Deep Convolutional Networks on 3D Point Clouds." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.
[You, 2018]	H. You, Y. Feng, R. Ji, and Y. Gao. "PVNet: A Joint Convolutional Network of Point Cloud and Multi-View for 3D Shape Recognition." In: *Proceedings of the 26th ACM International Conference on Multimedia*. MM '18. Seoul, Republic of Korea: Association for Computing Machinery, 2018, pp. 1310–1318. ISBN: 9781450356657. DOI: `10.1145/3240508.3240702`.