Technische Universität München

Algorithmic Subdivision of Gamespaces into Semantic Volumes using Delimiters

Wissenschaftliche Arbeit zur Erlangung des Grades

B.Sc. Informatik: Games Engineering

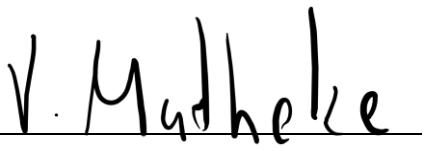
an der School of Computation, Information and Technology der Technischen Universität München.

Betreuer/-in	Daniel Dyrda, M.Sc.
	Lehrstuhl für Erweiterte Realität
Aufgabensteller/-in	Prof. Dr. rer. nat. David Plecher
	Lehrstuhl für Erweiterte Realität
Eingereicht von	Victor Matheke
	Kameterstraße 33
	85579 Neubiberg
	victor.matheke@t-online.de
Eingereicht am	Neubiberg, den 23.09.2024

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I confirm that this Bachelor's thesis is my own work and I have documented all sources and material used.

Neubiberg, 23.09.2024 

Ort, Datum, Unterschrift

Abstract

Video games are commonly played in a continuous space but evaluate their game logic based on the semantic location of entities instead of their raw position vectors in continuous space. Creating a mapping from the continuous into a semantic space to determine the semantic location can be a large effort for game developers and can be a source of bugs if not done correctly.

This thesis designs and implements a software solution to generate such a mapping from continuous into semantic space algorithmically based on input by the game developers. The mapping is generated based on spatial signatures, where the game developer provides a set of delimiters and anchors as input. Delimiters can be virtually extended inside the solution to decrease the overhead for the developers of preparing the input for the solution. Each anchor then grows a volume which extends as far as possible in all directions until a delimiter is hit. This volume then represents a spatial signature with semantic coherence. The generated mapping can then be queried to find which volume any point of the continuous space resides in.

The thesis assesses the proposed solution on the fulfillment of the requirements towards such a system, as well as stability, performance of the supplied implementation. A comparison to other approaches towards this problem is made. Finally, the thesis lists a set of features which have not been implemented yet but would improve the solution in the future.

The solution proposed in this thesis alleviates the workload of game developers while simultaneously improving the mapping from continuous into semantic space. It has been designed for easy integration into existing software solutions by being a lightweight, enclosed software solution.

Keywords: *spatial partitioning, semantic volumes, anchors and delimiters, game development, games engineering*

Acknowledgements

I would like to thank Jonathan Blow and Casey Muratori for (unknowingly) being great teachers, as well as M.Sc. Daniel Dyrda for many interesting discussions around this topic. I would like to especially thank my parents for all the help and support in- and outside of this thesis.

Contents

1. Introduction

Games throughout history commonly require a space to be played in [1, p. 65]. This space can either be discrete or continuous. An example for a discrete game space would be chess, where neither the actual “physical” size of the chessboard nor the “physical” position of an individual piece matters, but instead only the “semantic” position does (where the “semantic” position in chess would be the row / column tuple).

Continuous game spaces on the other hand are often found in sports, but they are also very common in digital games. Rules in these games however are often formulated in “semantic space”, which requires a mapping going from the “physical” continuous space to the “semantic” one. In soccer for example, it is important whether the ball is “fully inside the goal”, or whether a player is “inside the box”. These rules are formulated independently of the physical sizes of any particular soccer pitch, but evaluating them during a game does require knowledge of the continuous space the game is taking place in (where exactly is the ball in this specific situation, where and how big is the goal?). In a video game, it might be desirable to know which semantic space (e.g. a specific room) the player has just entered, and whether any other person is in that same space. Such information can also help in setting up a Pacing Graph [2], a more systemic approach to Game Design.

This mapping from continuous to semantic space is a large amount of effort, especially in video games containing large worlds with many relevant volumes. This is usually dealt with by a lot of manual work from video game developers to assign semantic meaning (in the form of IDs) to these relevant volumes in the continuous game space. These volumes may however be rather complex (for example mapping the different rooms of all house structures in a town), and the long iterative process of game development, along with the huge workload, can also lead to discrepancies between the visual geometry shown to the player and its semantic representation, in turn potentially leading to a worsened player experience.

The goal of this thesis is to propose, implement and evaluate an algorithm for automatically creating such a mapping based on input from the game developers, in order to lighten the workload on the developers while simultaneously improving the quality of the mapping.

1.1. Problem Statement

This mapping is created for a *world* in a three-dimensional space. All objects in one world can and will interact with each other, but objects from two different worlds are independent. In video games, the term *scene* is often used to describe this behavior [3]. Since the word *scene* is already quite overloaded with meaning (in- and outside of games), this thesis will stick to the term *world*.

The input for the algorithm consists of *anchors* and *delimiters* defined for the world. This input may be created manually by a game developer, or it may be (at least partially) automatically derived from the game world's geometry and passed on to the algorithm.

Anchors are objects in the three-dimensional space of the world which the developer considers to be the characteristic origin of a semantic volume, meaning a semantic volume will grow outward from this origin until its growth is stopped by *delimiters*. *Delimiters* are objects in continuous space which act as a border element between semantic volumes. As a real-world example, fences or walls would often be considered delimiters, and the altar in a church might be considered an anchor point.

The algorithm therefore must subdivide the input space by calculating a volume for every anchor fulfilling the restrictions listed in the next chapter. Afterwards, the generated volumes can be queried to check whether a specific point resides in them for evaluation of gameplay rules. This means that the proposed solution can be split into two phases: The subdivision of the world space, which needs to be done once, and the querying of the generated data structure during the run-time of the game.

1.2. Functional Requirements

The problem statement implies the following list of requirements which any algorithm attempting to solve the problem must fulfill.

1. The algorithm must be deterministic. This means that for the same input, the same output must always be produced. Otherwise, the user experience of the algorithm would suffer, because results might randomly change without any actual change to the input.

2. Querying the data structure after it has been generated must be possible in real time for the algorithm to be applicable in games. The build-up phase of the data structure may however be done "offline", because it is computationally expensive. Offline means that the data structure is created during the development of the

game, and then serialized and loaded during run time (which should be a lot faster than the generation for large inputs).

3. The volume calculated for any anchor must never intersect any of the defined delimiters in the world. Conversely, a delimiter must always stop a volume from growing in this direction.

4. The volume of any anchor must extend as far as possible while not violating the previous requirement. This means that these volumes must also perfectly represent complex shapes, by bending around corners or filling in tight polygonal shapes.

5. The volume of any anchor must always be completely enclosed and must always contain its anchor. It must not intersect itself or any other volume.

1.3. Non-functional requirements

1. The algorithm should be designed to be predictable for a human user to have a positive effect in real world application. The overhead required for a human user to achieve the desired outcome should be kept as low as possible.

2. The algorithm was designed with the goal to keep the error of the output as low as possible. The error of the output is defined as the discrepancy between the “ideal” desired output and the actual result.

3. The solution implemented should be easy to integrate into existing tools in real-world applications, such as game engines. The two distinct phases of the proposed solution (build-up of the internal data structure versus querying the data structure at run-time) are geared towards games with a static world which does not change (semantically) during run-time of the game, although re-computation is technically possible at run-time.

The goal of this thesis is to present a feasible algorithm and to illustrate its potential use in game development.

It focuses on the general design and on identifying potential problems in the design by providing an implementation. It does not claim to implement a fully functional solution which already fulfills all previously stated requirements.

2. Related work

2.1. Space Foundation System

A high-level overview of the problem space, as well as many terms and definitions used in this thesis, were introduced in the paper “Space Foundation System: An Approach to Spatial Problems in Games” [4]. The target of this thesis is to propose and implement a possible solution to the problem stated in the cited paper.

2.2. Thesis by Kerstin Pfaffinger

Another bachelor’s thesis about this topic has been written by Kerstin Pfaffinger in 2024 [5]. It pursues a different approach to the very same problem, which is based on the known Marching Cubes [6] algorithm to reconstruct surfaces from a voxel grid. This approach however suffers from an error in the constructed volumes that is proportional to the chosen size of the voxels. This thesis attempts to improve the error of the calculated volumes using a different approach.

2.3. Spatial Signatures

The paper “Spatial Signatures - Understanding (urban) spaces through form and function” [7] introduced the concept of *Spatial Signatures* in the analysis of urban spaces. The authors define a spatial signature as “a characterization of space based on form and function designed to understand urban environments” [7, p. 4]. They map out different spatial signatures in cities such as Barcelona or Singapore, to “provide unique insight into the ways human populations create and inhabit cities” [7, p. 9].

Their methodology takes in input data consisting of “delimiters” such as roads or rivers of a two-dimensional mapping of the city. Geometric cells are derived from this input data and combined into spatial signatures [7, p. 8].

The idea of creating a mapping of space into different signatures based on characteristics is related to the problem statement of this thesis. However, the practicality of the approach proposed in this thesis requires a well-specified input in a three-dimensional space, as well as additional features making integration into game software easier (see chapters 3.4.3 and 3.5)

2.4. Bounding Volume Hierarchies

An acceleration data structure that is commonly used in raytracing applications is called a *Bounding Volume Hierarchy*, or BVH for short [8]. It massively improves the performance of intersections tests between rays and a vast number of triangles by culling out sets of triangles using more efficient tests. It is essentially a tree of recursively shrinking bounding boxes around triangles.

Such a data structure has been described in the paper “A 3-Dimensional Representation for Fast Rendering of Complex Scenes” [9] and has since become the de facto standard in raytracing applications. In this thesis, it is used for accelerating the large number of ray-triangle intersection tests.

2.5. Flood Filling

The term “Flood Filling” describes an algorithm working on a graph structure that expands a shape from an origin point outward under pre-defined conditions. It is often used in raster graphics but can also be extended into three dimensions. A detailed description can be found in “Contour filling in raster graphics” [10]. The solution described in this thesis used Flood Filling for approximating the volume of anchors in three dimensions.

2.6. Definition of the term “Volume”

The term “Volume” is central to this thesis, as it describes one of the fundamental concepts of the proposed solution. In the context of this thesis the term “Volume” deviates from the mathematical concept usually described by it. The Cambridge Dictionary defines volume as “the amount of space that is contained within an object or solid shape” [11]. In video game technology, a Volume often does not only refer to the amount of space, but also to the three-dimensional shape of the referred space (e.g. Unreal Engine [12], Unity Engine [13]). Franics Ching defines the term “Volume” similarly in the paper “Architecture: Form, space, and order” [14].

The term “Volume” as used in this thesis therefore refers to the implicitly contained shape of an enclosed, three-dimensional region.

3. Approach

3.1. Overview

The proposed solution for this thesis has been designed with the previously listed requirements in mind. The approach implemented takes several common algorithms and data structures and combines them in a unique way to solve the problem at hand. This chapter will present the different intermediate steps of the proposed solution, the decisions that were taken during the implementation phase, as well as pointing out issues and potential improvements of the solution.

Spatial signatures as described in chapter 2.3 are the conceptual basis for defining a world consisting of anchors and delimiters. The developer using this solution creates the input for a world based on the spatial signatures inside said world. This input contains the list of anchors and delimiters present in the world.

Anchors, volumes and delimiters require a practical representation in software so that they can be manipulated internally. The representations for the different objects were chosen to fulfill the requirements listed in the problem statement, while providing necessary functionality to the following internal steps of the algorithm. The concept of delimiters as introduced in the problem statement is distinguished into two orthogonal parts in this approach, to aid in the integration of the solution into existing software. The representation of the different objects is based on triangles for the most flexibility, and is further described in chapters 3.2, 3.3 and 3.4.

The first step of the build-up stage of the solution is to preprocess the input that was passed in by the user. This preprocessing provides a feature called *Virtually Extending Delimiter Planes* to the user, reducing the amount of work the user has to put designing into the original input (see the chapters 3.4.3 and 3.5). This feature takes advantage of the triangulated nature of delimiter planes, by algorithmically expanding delimiter planes based on criteria.

After the preprocessing of the input, a Floodfilling algorithm is used to expand the outline for each volume as far as possible. Based on that intermediate result, the exact triangle representation of the volume will be derived using a BVH (see chapter 3.6).

After the build-up stage, the approach to querying a point is described. This step again takes advantage of the triangle representation of volumes to find the volume in which the point resides (see chapter 3.7).

3.2. Anchors

In the introduction, anchors were defined as “objects in space which the developer considers to be the characteristic origin of a semantic volume”. Anchors could therefore be represented using three-dimensional shapes, like a cuboid, a sphere or any polygonal mesh. This might however lead to some unwanted edge-cases; what should happen if two anchors intersect each other, or if a delimiter intersects with the anchor? Anchors are therefore defined as infinitely small points in the three-dimensional space of the world to avoid these issues and make the algorithm simpler.

3.3. Volumes

The goal of the algorithm is to subdivide a three-dimensional space into volumes that represent semantic spatial cohesion, based on user input. A subspace of any n-dimension space is again an n-dimensional space. In the problem statement, a world was defined to be a three-dimensional space, so volumes are represented as three-dimensional shapes in this approach.

1. A simple approach might be to approximate the result using some pre-defined three-dimensional shape, like a sphere or a cuboid (the latter often referred to as a *bounding box*). These shapes are fast to compute and have a smaller memory footprint than the following approaches, but there might be a huge discrepancy between the chosen shape and the expected output by the user (the *error* of the output). This is especially true for concave shapes (see the second image in Figure 1).

2. One might then consider using multiple of these “primitive” shapes to represent a volume, recursively making the shapes smaller and smaller, similar to a BVH tree. While this will decrease the error slightly, it will increase the computation and space complexity (exponentially for more levels of recursion in the hierarchy). Because computers cannot store an infinite number of bounding boxes, the result will always have some discrepancy to the desired output, which grows larger the tighter the memory constraints are. At the same time, the initial benefits of fast computation time and a small memory footprint are reduced the more recursion levels are used, until eventually the disadvantages outweigh the advantages.

3. An approach used in Pfaffinger’s thesis [5] is to discretize the continuous space of the world into a three-dimensional grid. A volume is then represented as a set of cells which are considered inside the volume (determined by the cell’s center,

see Figure 1, shape 4), and a triangle mesh is computed using the Marching Cubes [6] algorithm. While this approach does initially handle concave shapes better than recursive bounding boxes, the large error remains due to the discretization of the space into fixed intervals. Both error and computational complexity are proportional to the chosen cell size.

4. In computer graphics, *meshes* are usually represented by a list of triangles [15, p. 625]. The difference between meshes and volumes in this context is that meshes in general do not need to be enclosed, but a volume (in the sense of this thesis) is always mesh. Triangles are the simplest shape in three-dimensional space and can therefore be used as a piecewise approximation of linear approximation to a surface. These properties make triangle meshes a great general-purpose solution for representing complex shapes and surfaces, especially concave ones.
   Additionally, while the algorithm may be applied to any problem space, its main use is seen in video games. Both developers and tools are therefore used to triangle meshes, which can aid in the integration of the solution into other projects, or in future adaptations of the solution. Triangulated meshes have therefore been chosen for the representation of volumes (see Figure 1, shape 5).

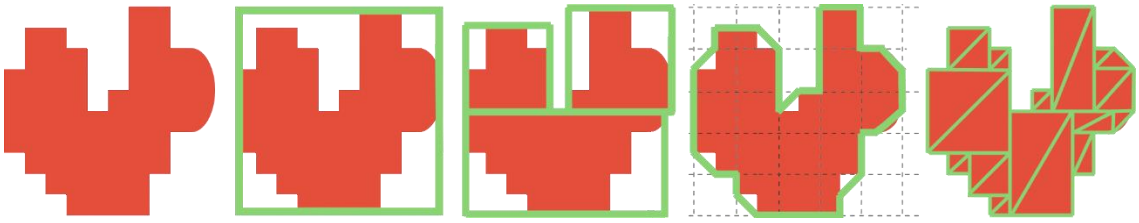


Figure 1: The four proposed representations of volumes (projected into two dimensions), from left to right, with the initial shape on the left. The error (white background inside the green shapes) decreases from left to right.

In the approach proposed in this thesis, volumes are not hierarchical. This means that at any point in the space of the world, either no or exactly one volume is present. The evaluation of game rules often assumes that a point in space is inside no or exactly one volume, making evaluation simple. A simple form of hierarchy can be implemented by the user by “logically” combining different volumes into a larger one. In Figure 3, a player might be considered inside the house if they are inside one of the three volumes present inside the house.

3.4. Delimiters

Delimiters were defined in the problem statement as “objects in continuous space which act as a border element between semantic volumes”. Their main characteristic is

therefore in determining whether a point is on one side of the border or the other. This idea can be represented using planes, where a point can be on either side of this plane (or exactly on it; this edge-case needs to be handled explicitly). Therefore, planes were chosen as the primitive for representing delimiters.

For the convenience of the user of this solution, the concept of a delimiter has been set up for higher flexibility by using two components, delimiter *objects* and *delimiter planes*. A delimiter object is a cuboid which can own between one and six delimiter planes. This grouping of multiple delimiter planes into delimiter objects can improve the integration of the solution into other software. As an example, a wall may be represented by two delimiter planes, which can be grouped together into a single delimiter object. This unit can then be treated as such, which reduces friction when modifying the input data, such as moving the wall. The two delimiter planes attached to the object will remain attached to the object and do not have to be moved independently.

3.4.1. Delimiter Planes

Planes can be represented in several different ways. A common one is the Hesse Normal Form [16], in which a plane is defined using an origin point and a normal vector. Solving a simple mathematical equation evaluates whether a point is in front, behind or exactly on the plane.

For this approach however, it is vital that delimiter planes are not infinitely large, which is not supported by such mathematical expressions like the Hesse Normal Form. Instead, the planes need to be clipped arbitrarily into planar regions of the original plane (see chapter 3.5). Therefore, similar to the reasoning on volumes, delimiter planes are represented using a list of finite triangles. All triangles are required to be coplanar and non-overlapping, and together they shape the delimiter plane. This list of triangles can then easily be adapted to change the area of the delimiter plane. These triangles will also be used to assemble to volumes later in the build-up stage (see chapter 3.6). While this imposes a larger memory (and execution time) footprint than other representations, the flexibility and accuracy provided by this approach is required to fulfill the requirements of the problem statement.

3.4.2. Delimiter Objects

A delimiter object is represented by a cuboid, which is positioned, rotated and scaled in the world. These three vectors make up what is typically called a Transform, a common concept in game engines [17]. This makes integrating the solution into such game engines easy, by making use of their manipulation tools.

A delimiter object therefore stores its transform (defining the cuboid), as well as up to six delimiter planes that are attached to it during input setup, and finally a *hierarchy level*. This hierarchy level will be used later in the build-up when solving intersections (see chapter 3.5).

When attaching delimiter planes to objects, the user can specify where to attach the plane. It can either be attached to one of the six faces of the cuboid, or to the center of the cuboid oriented along one of the three axes. An example for the latter option may be a doorframe, which should only have one delimiter plane centered inside the frame.

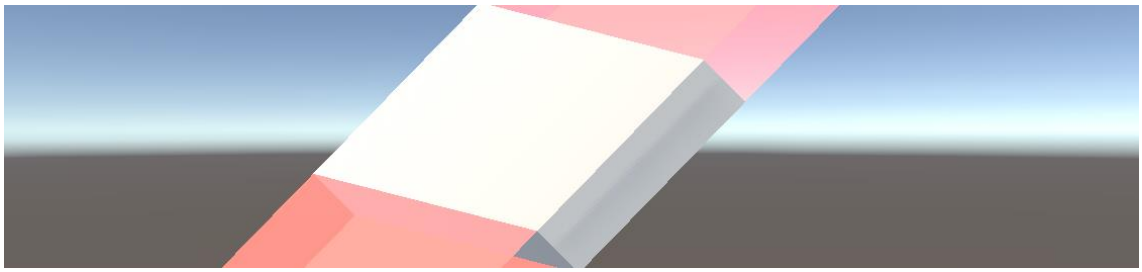


Figure 2: A delimiter object (in white) positioned and oriented in space, with two virtually extended delimiter planes (in red) created on the positive and negative Z faces of the cuboid.

3.4.3. Virtually Extending Delimiter Planes

The algorithm was designed with the goal of alleviating workload from game developers. To do that, the amount of overhead work required for setting up the input for the algorithm should be kept to the possible minimum. The solution therefore gives the option to virtually extend specified delimiter planes to automatically derive the actual size of delimiter planes before calculating the volumes for every anchor in the world, so that developers don't have to do this manually. This virtual extension is specified in the input supplied by the user of the solution.

As described in the previous chapter, delimiter planes must be attached to delimiter objects. If the plane should be virtually extended (which must be specified in the input), it will extend as far as possible in all directions until it intersects other delimiter planes. Otherwise, it will just take on the dimensions of the delimiter object's cuboid it was attached to (see Figure 4). This is helpful in cases where the geometry in the game world does not perfectly represent the requested semantic subdivision of the game space (e.g. a door frame in a wall, which should be ignored in the semantic mapping).

The game developer may manually set up the delimiter planes to always do exactly what they want at the cost of increased workload. The algorithm was however designed in a way that common cases should generate the expected result, so that developers only have to manually set up specific planes in edge cases.

In the following example, the blueprint of an apartment is shown from above. The apartment consists of a living room, a kitchen and a hallway. The left picture shows the geometry used for rendering and physics of the game. The gap in the lower wall allows the player to enter the living room coming from the hallway, and the open kitchen allows the player to roam there as well. The game developer requires knowledge on the whereabouts of the player for a game rule, so distinct volumes for all three rooms must be generated.

The next picture from the left shows the setup of delimiter planes, where each wall acts as a delimiter object owning a centered delimiter plane. This represents the input passed to the algorithm. The third picture then shows the intermediate representation of the extended planes inside the algorithm. The dotted lines are the virtual extensions of the planes where the developer has requested that feature. They are dotted purely for clarity in this picture, in the actual algorithm no difference is made between parts of the planes that have been extended or not. The last picture shows a possible manual adaptation of the input the developer might do if they want to achieve a different outcome of the algorithm, by adding a new delimiter object and plane not present in original geometry of the world.

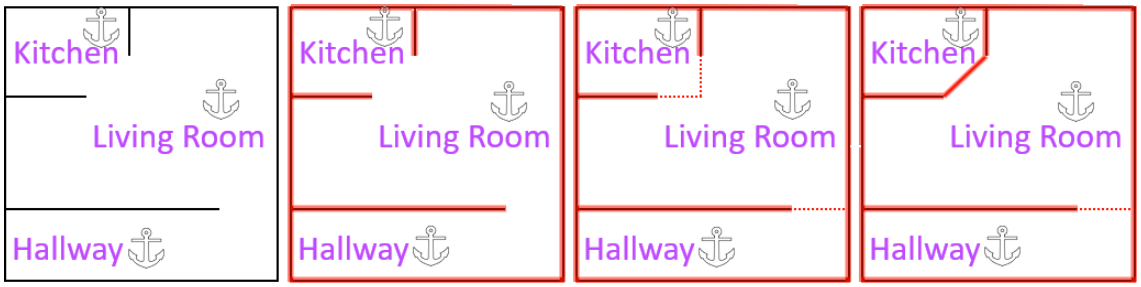


Figure 3: Visual Example of virtually extending Delimiter Planes. Delimiters are displayed from a top-down view.

This virtual extension can be requested for neither, either one, or both orthogonal axis of a delimiter plane.

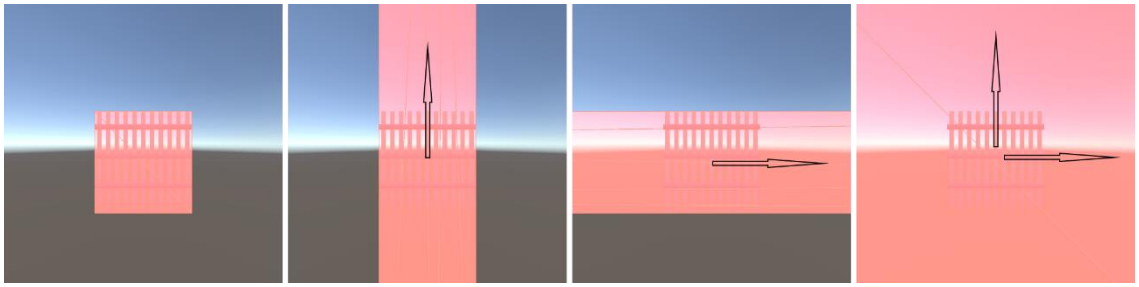


Figure 4: Virtual Extension can be specific for each orthogonal axis of a plane. This delimiter object was only assigned one delimiter plane.

3.5. Clipping Delimiters

As explained in the previous chapter, delimiter planes can be extended virtually to automatically resolve underspecification of the input by the developers. This reduces the amount of overhead work, as well as the size of the input data, required to get the desired output, assuming the automatic resolution described in this chapter matches the desired behavior by the developer. If that isn't the case, the developers must resolve this specification by adding more delimiters. See Figure 3 for an example of this.

Additionally, the later stages of the algorithm require that no triangles of two delimiter planes (whether virtually extended or not) intersect each other. This is further explained in the chapter 3.6, but the requirement is fulfilled in this stage.

When setting up delimiter planes, their area is usually assumed to be that of the face of the delimiter object it is created for. When extending the plane along an axis, however, it is first assumed to be infinitely large along that axis (or, in practice, large enough to cover the entire world area), and it will be cut down algorithmically to the expected size.

This means that initially, every delimiter plane consists of two triangles forming a rectangle (either representing the delimiter object's cuboid's face area, or large enough to cover the entire world area). The algorithm then iteratively finds intersections between two triangles of any two delimiter planes and solves it by subdividing the triangles along the intersection edge between them (see chapter 3.5.2), until no more intersections are present.

3.5.1. Cutting Delimiters down instead of growing them outward

The idea of clipping the plane down instead of growing it outward might seem unintuitive at first, but it can be reduced to the same problem. The latter approach would require a discrete step size at which the delimiter planes grow on each iteration, until an intersection is found, and the plane does not grow any further in that direction. With such a discrete step size, the plane can either stop before the intersection would happen (potentially leading to a small gap between where the plane stops and where it should be), or it can complete the step and stop after the intersection (in which case a small intersection might remain). Having a small enough step size so that any of these two options would be viable is non-practical, as numerical issues (see chapter 4.2) and run-time (even for the offline requirement) would grow unmanageable.

Therefore, instead of just keeping the plane as it was (before or after the step), the algorithm would need to solve the intersection between the two delimiter planes cleanly by

calculating the intersection edge and subdividing the planes so that no intersection remains.

Having a discrete step size also does not guarantee that only ever one intersection is found in each iteration. This also means that the algorithm must decide the order in which intersections are solved. The chosen step size therefore does not actually matter, and it can be set to the world size. In that case, only ever one step is needed, and the approach just extends all delimiter planes as far as possible and then cuts them down appropriately. The order in which intersections are solved will be explained later in this chapter.

3.5.2. Solving an intersection

This step of the algorithm goes through all delimiter planes in the world and checks for intersections between any possible pair of them. The triangle representations of both planes are then adapted to not intersect anymore.

When solving an intersection, the hierarchical levels of both delimiters that own the respective planes are compared. This gives the developer more control via the input to specify how two delimiters should interact when they intersect. If the delimiter objects involved in the intersection have the same level, then both planes are clipped (“stopped”) at that intersection. Otherwise, the delimiter with the higher numerical value as level is stopped, the delimiter with the lower level is not. This is helpful in scenarios where one delimiter has higher semantic priority for the developer. This also works for the case where two delimiter planes that belong to the same object intersect.

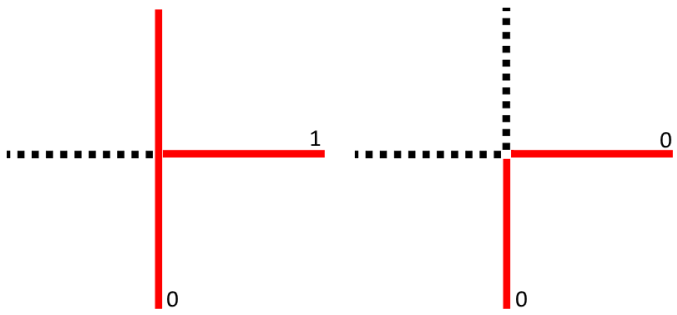


Figure 5: The black part is clipped away from the delimiters whose level is not lower than the other's. Delimiters are seen from a top-down view.

Figure 5 shows the two scenarios when two delimiter planes intersect from a top-down view, with the hierarchy level shown as a number. Another example of this behavior can be seen in Figure 3, where the outer walls of the house have a lower hierarchy level, and so the inner walls are stopped from going outside, but the outer walls essentially ignore the inner ones.

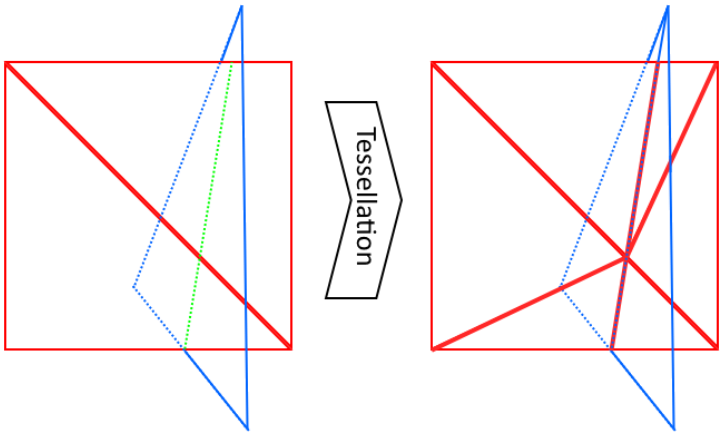


Figure 6: The two delimiters in red and blue intersect along the green axis. The red and blue delimiter planes are tessellated so that no two triangles intersect anymore.

Figure 6 shows an example of how the triangulation of the plane is adapted through a process called “Tessellation” (see chapter 4.5) so that no triangle intersects with any other triangle anymore. If the red delimiter object has a higher hierarchy level than the blue one, some of the triangles of the red delimiter plane would need to be removed.

3.5.3. Heuristic for ordering Delimiter intersections

The order in which the delimiter intersections are solved is vital to the predictability of the algorithm. Delimiters are conceptually expected to grow outwards until they are stopped, while in practice all intersections between delimiter planes are gathered at once (see chapter 3.5.1). To achieve the expected result, the intersections need to be resolved in the order in which they would’ve occurred conceptually. This requires that the intersections are sorted by some heuristic. Figure 7 shows an example result when a bad heuristic is used for sorting the intersections.

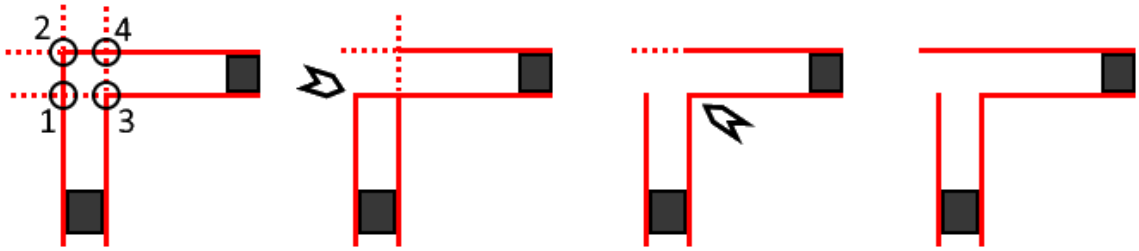


Figure 7: An example of unexpected results if the intersections are not ordered properly. Intersection 1 should be handled after intersection 3. Delimiters are seen from a top-down view, objects in black, planes in red.

The chosen heuristic should therefore reflect the idea that delimiters grow outward, meaning intersections that are “nearer” to the involved delimiters’ origin have higher priority in the resolution. This avoids the issue shown in Figure 7, where intersections result in “false” clipping because one of the delimiter planes should not actually extend far enough to reach the intersection point.

The heuristic calculates the shortest distance from each delimiter plane's origin to the opposite delimiter plane and sums the distances together. The intersections are then sorted so that the first intersection to be solved has the smallest value. A visual example of this is shown in Figure 8.

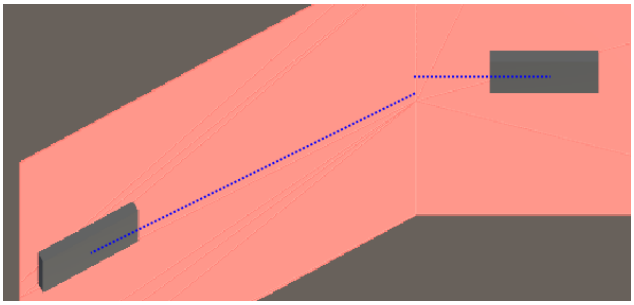


Figure 8: The blue dotted lines represent the heuristic by which intersections are sorted. The lengths of the two lines are summed together to find one distance value.

3.6. Calculating Volumes

After the input has been processed as described in the previous chapters, the next step is to calculate the volume for every anchor in the world. This step assembles a set of triangles that represent the "ideal" volume as close as possible, while fulfilling the requirements that were initially described in 1.2.

As described in chapter 3.3, a volume is represented by a list of triangles. Adapting the requirements from the problem statement to this assumption leads to the following implications:

1. A triangle of a volume must never penetrate a triangle of a delimiter (it is, by definition in point 2, identical to exactly one delimiter's triangle).

2. The triangles that make up a volume must be a subset of the triangles of the delimiters that border this volume.

3. The list of triangles that make up a volume must be completely enclosed and must always contain its anchor.

This stage of the algorithm therefore finds the delimiters that border an anchor and assemble the volume by copying triangles from these surrounding delimiters into the volume's triangle list. This fulfills all the three requirements listed above.

In the chapter 3.5, it is ensured that no triangle of a delimiter plane intersects with any other triangle of any other delimiter plane. When copying a triangle from a delimiter plane into a volume, it is therefore guaranteed to not penetrate any other triangle.

This approach also guarantees that the volume representation perfectly matches the expected output, because the chosen triangles are exactly the triangles that stop the volume from growing any further.

For the last requirement, the algorithm implicitly creates six additional delimiter planes which span around the entire world, internally called the *root* planes. Apart from being generated automatically, they behave the same as the delimiter planes that were specified in the input. They have a lower hierarchy level than the delimiter objects specified in the input, so that they are tessellated, but not clipped. This means that all points inside the world are completely enclosed by delimiter planes, which in turn means that any point inside the world can be completely enclosed using a set of triangles from delimiter planes.

The remaining challenge in this step of the algorithm is finding the set of triangles that make up an anchor's volume.

3.6.1. Assembling the triangles

To build up a volume, the algorithm looks at all delimiter planes and figures out which (if any) of the triangles making up the plane are actually delimiting that volume. If that is the case, the triangle gets added to the volume's internal representation. With the guarantees about the triangulation of delimiter planes, as well as the guarantees of the world's root planes, this ensures that a volume is always completely enclosed while extending as far as expected.

Determining whether a triangle is actually delimiting an anchor's volume is non-trivial. This algorithm applies another heuristic for this. For each triangle A, it checks a ray going from the triangle's center to the anchor position. If any other triangle B is intersecting with this ray, then triangle A cannot be a delimiting triangle of the anchor's volume as the volume should be stopped from growing in this direction by triangle B. Otherwise, the volume should grow in this direction until it hits this triangle A, making triangle A delimiting to the volume, and is added to the list. This heuristic initially has an obvious flaw.

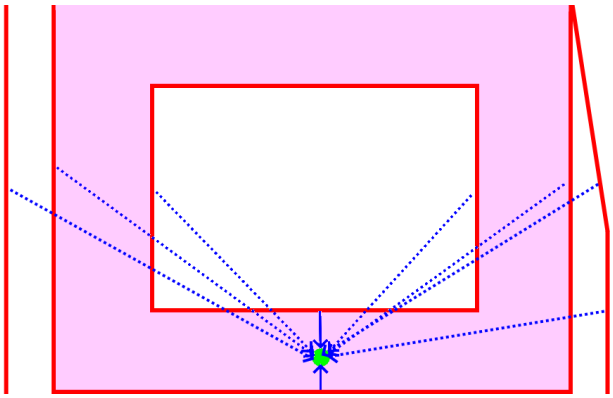


Figure 9: Top-down view of false negatives when calculating anchor volumes. The expected volume is indicated in pink, the anchor in green. Successful rays are indicated by a solid blue line, blocked rays by a dotted line.

In this setup, the triangles making up the delimiters on the left and right of the image would not be part of the anchor's volume, although they are expected to. They are therefore false negatives.

One approach to improve this is to use more than one *query point* when casting rays from the triangles' centers. This improves the output for concave shapes like the example in **Error! Reference source not found.**. With enough additional query points to check, all delimiting triangles should be found and added to the volume, as shown in Figure 10.

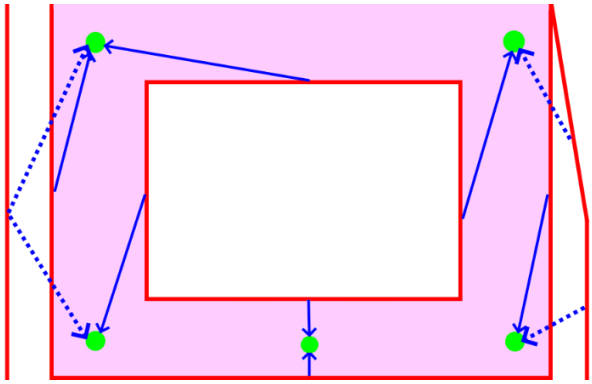


Figure 10: Using more query points (green) to check for delimiting triangles. All triangles now cast un-obstructed rays to at least one of the points. Some rays have been omitted for visual clarity.

This heuristic however requires enough points to be available so that no false negatives remain. More importantly, the points need to be positioned properly so that the volume gets represented properly.

A larger number of points obviously has a higher performance cost, both in memory and time consumption. This step of the algorithm is only required in the build-up of the data structure which may be done offline, as explained in chapter 1.2. Performance is therefore not a primary consideration.

3.6.2. Choosing query points

Choosing a set of query points used for querying delimiting triangles is also non-trivial, as they need to be inside the volume (which has not yet been determined), and they should be positioned and spaced out to be as useful as possible.

The approach used is derived from the concept of *Floodfilling* [10]. For this, the world is discretized into a three-dimensional grid with a specific uniform grid cell size. The center of each cell represents a candidate for a query point. This gives an easy uniform distribution of query points over the entire world space. Most of these points will be outside of the anchor volume however, in which case they should not actually be used as query points. The algorithm must therefore first determine which of the cells (characterized by their center) are actually inside the volume and should therefore be used as query points.

This is achieved by recursively marking cells as “reachable”, in which case they are inside the volume. The grid is transformed so that one cell’s origin is exactly the anchor’s position. This cell is initially marked as reachable. All neighboring cells are then checked on whether the ray segment between them and the reachable cell is obstructed by any delimiter plane. If not, then this neighbor is also marked as reachable, and the process continues recursively until no more additional cells can be reached. The centers of all cells that have been marked as reachable should therefore be used as query points when calculating the anchor’s volume.

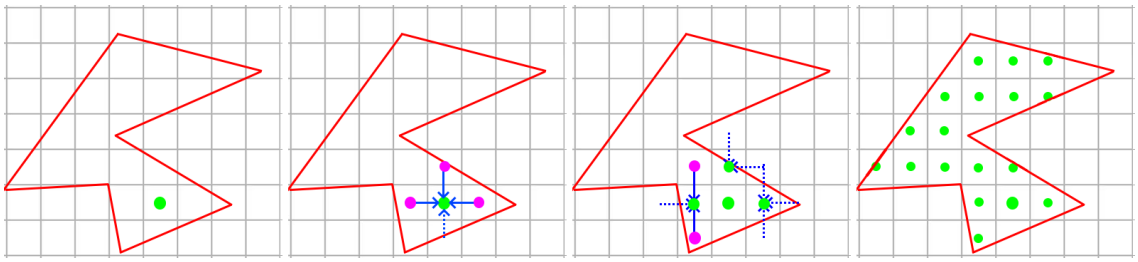


Figure 11: Four slices of the Floodfilling process. Green points indicate reachable cells of past iterations, purple ones indicate the reachable cells of this iteration. The final picture shows the output of the algorithm. All these points are query points.

With a large cell size, this approach can lead to issues where gaps between delimiter planes may not be recognized, leading to unexpected and unwanted results. Such a case is visualized in Figure 12. The cell size must therefore be small enough that such problems don’t happen for a specific world. On the other hand, smaller cell sizes lead to a higher time and space consumption of the algorithm, to the point of impracticality even on the most powerful machines (when combining large worlds with small cells). In practice, game developers already need to ensure that gaps in the geometry are either big enough for the player to walk through (like a door frame) or are not there at all, fighting against this issue. It is, however, a significant drawback of the Floodfilling algorithm.

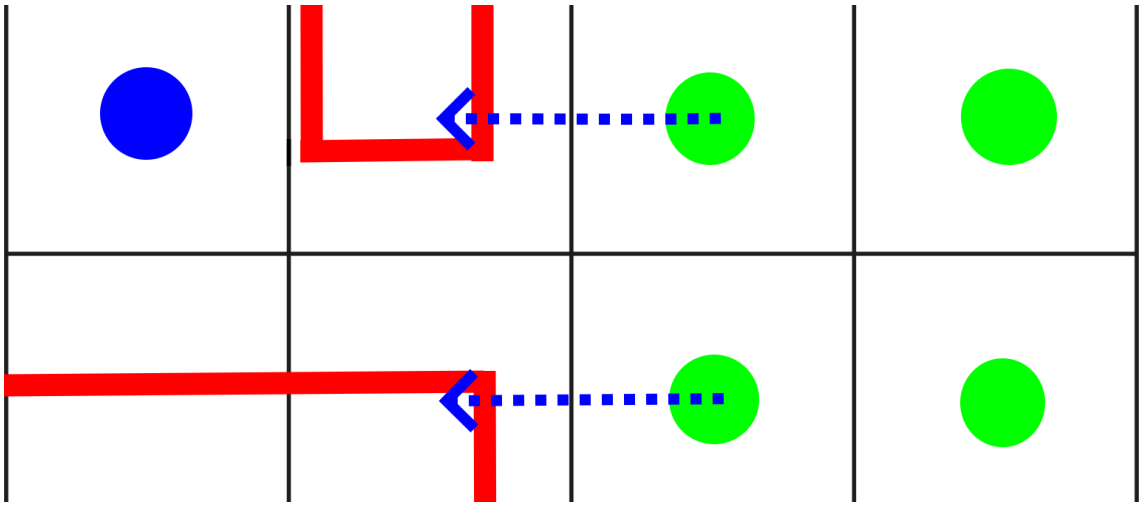


Figure 12: Issues with large grid sizes in the Floodfilling Algorithm. The cell in the top left (marked by the blue dot) will not be reached, leading to unexpected output.

3.6.3. Possible Improvements

In the future, the Floodfilling algorithm might be based on recursive cells, similar to a BVH tree. This would alleviate the issues observed if the cell size is big, as described in Figure 12. If a cell does not encounter any intersections, then the flood filling algorithm proceeds as normal. If there are intersections, the original cell is subdivided into smaller ones and the process repeats for these cells until a lower limit of the size is reached. This might make it practical to use extremely small cell sizes in the cases where it is needed while not wasting memory and time on parts of the world where such detail is not required.

As the goal of this thesis is to show the overall possibility and potential of such a solution and not provide a finished product, this feature has not been implemented.

3.7. Querying the World

After the first stage of building up the volumes for each anchor, the solution must also provide a way to query the semantic meaning of any point in the world by determining in which volume this point resides. The algorithm guarantees (for valid input) that any point in space resides in no or exactly one value, and that all volumes have been properly calculated (they are completely enclosed).

For such an enclosed mesh it is trivial to determine whether a point lies inside or outside of it. A ray is cast from the point of interest into infinity (in an arbitrary direction). Intersections between this ray and the triangles making up the volume are counted. If the number of intersections is odd, then the point is inside the volume, otherwise it isn't. This technique (often called the *Even-Odd Rule*) has been in use as early as 1974 [18], but has been adapted to three dimensions for this application.

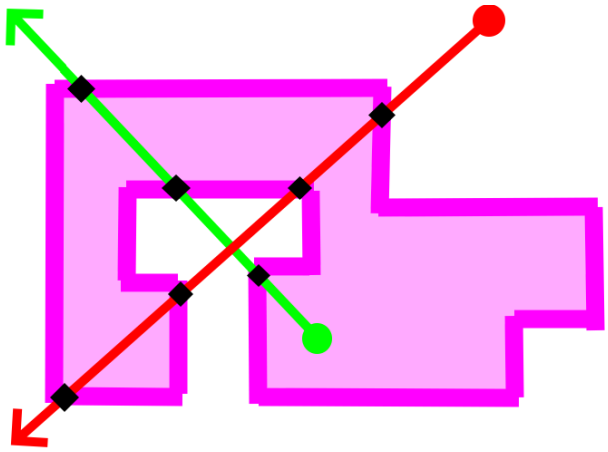


Figure 13: Visualization of the Even-Odd rule. The green ray intersects an odd number of times, so the point is inside the polygon, contrary to the red point.

3.7.1. Issues of the trivial approach

There are however several edge cases that need to be handled.

1. If the point lies exactly on a triangle, this thesis defines the point to be “inside” the volume. Should this case happen during evaluation of the Even-Odd Rule, then an early exit can be made as the point is considered “inside” the volume if it lies exactly on any triangle of the volume. Otherwise, it would be undefined how many intersections occur between the ray and the triangle, depending on the ray direction.

   If this triangle on which the point lies is shared by multiple volumes (e.g. “the other side of the fence”), then the resulting volume will be the one which is checked first and therefore not well-defined.

2. If the arbitrary ray direction is orthogonal to a triangle’s normal and the query point lies exactly on the triangle’s plane (but not on the triangle itself!), then the intersection test is undefined, as there are an infinite number of intersection points. Therefore, triangles that are orthogonal to the ray direction are ignored in this test, as they do not have any implication to the result. Because the volume is enclosed, other triangles must be connected to this orthogonal one, which with intersections can be calculated as normal.

3. If the ray passes exactly through a vertex, then there would be multiple intersections (as many triangles might share this one vertex). This can lead to false results (both false negatives and false positives). To improve this, the algorithm attempts to find “duplicate” intersections based on the distance from the query point to the reported distance. Since all intersections are from the same ray, then intersections with (approximately) the same distance should only be counted once. This requires a small numerical tolerance value, which is a very common concept in numerical programming.

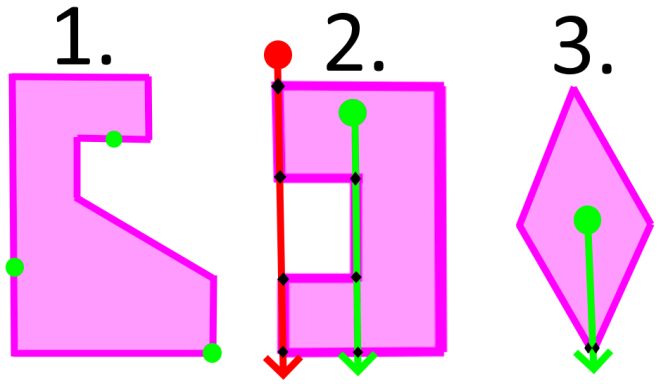


Figure 14: Visual Examples of the three described issues with the naive Even-Odd rule implementation.

3.7.2. Possible Performance Improvements

The described approach of casting rays against all volumes in the world is simple but inefficient. There are optimizations applicable to this problem to achieve a great improvement in performance. They have not been implemented as they were deemed unnecessary for this thesis due to the simplicity of the test cases. For more complex worlds in real-world applications, they might however be required to achieve real-time execution of queries.

The first optimization is to encapsulate every volume with an *Axis Aligned Bounding Box* (AABB for short) [19], which allows for easy rejection of volumes in which the query point definitely does not reside. Before ray-casting against all individual triangles of the volume, the algorithm first checks whether the point lies inside the AABB of the volume. If it does not, then the point cannot be inside the volume. If the point is inside the AABB, then the described ray-cast check is executed.

The second optimization is to use a Bounding Volume Hierarchy acceleration data structure. A BVH can massively improve the performance of intersection tests between rays and triangles. The BVH could either be constructed over all volumes by tagging each triangle with the volume it belongs to, or separately for each volume. The former would find all triangles in the world that intersect the ray and then count the number of times the ray intersects with any volume. The latter allows combining the BVH with the AABB optimization described before. See the chapter Bounding Volume Hierarchies for a more detailed description.

4. Implementation

4.1. Overview

This chapter will go into the more technical details of the implementation which have been omitted in the Approach chapter.

The implementation provided with this thesis has been programmed in C++ without any frameworks. This choice was made for the following reasons.

- C++ allows for highly optimized code, which was deemed important for meeting the performance goals.
- Being independent of any specific existing software enables easy integration into all other technologies, such as game engines like Unity or Unreal Engine, which is another goal of this thesis.

4.2. Dealing with Floating-point Precision

One of the primary technical challenges of implementing the proposed solution (or in fact implementing any numeric code) is dealing with numerical precision [20]. Numerical programming can use either Floating-point or Fixed-point number formats for computations. Floating-point is often supported directly in the hardware, making it easier and faster to use in the common case. Floating-point is also better at handling large ranges of values, since it does not overflow as quickly as fixed-point, especially when dealing with values in different magnitudes [21] [22].

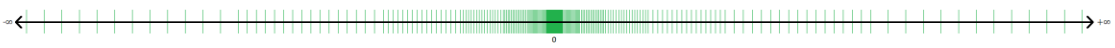


Figure 15: Precision of floating-point values at given intervals. [23]

Converting from one format to the other is also expensive. This implementation therefore also uses floating-point values for both the interface and the internal data storage.

Floating-point values are usually supported in two flavors: Single precision (using 32 bits in total) and double precision (using 64 bits in total) [22]. The latter one obviously offers higher precision at the cost of higher memory consumption. As this solution has been designed for use in games, which have a high requirement for memory hardware anyway, this trade-off has been made in favor of the double precision values.

Since the finite representation of (real) numbers in computer hardware cannot provide infinite accuracy, the code must always expect and correctly handle very small deviations from the “theoretical” result to the actual one in practice, which is caused by the hardware rounding this “theoretical” result to the nearest value that can be represented through the floating-point format.

One example for this is the intersection point between a ray and a triangle, which this solution relies upon heavily. The resulting intersection point is expected to lie exactly on the input triangle. In practice though, the calculated point may be slightly shifted if the exact point cannot be represented. During tessellation, this might lead to instability issues if an intersection point is not recognized as such, such as in the intersection for the triangles Figure 16. The intersection points might be calculated to be just slightly above the yellow triangle, in which case they would not be considered intersection points at all despite them being that. This would lead to no tessellation happening in this case, in case resulting in unexpected output.

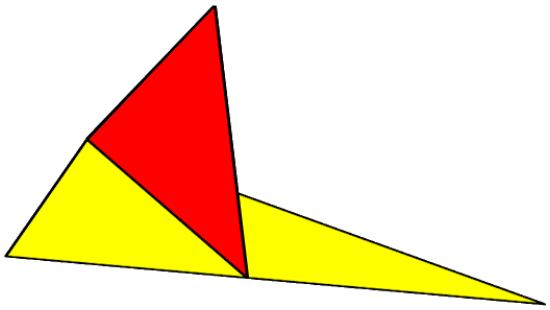


Figure 16: Two triangles touching each other, leading to potential Floating-point precision issues.

The default solution for this is to always factor in a small delta value (often called *Epsilon*) which acts as an interval of fuzziness around the theoretical results. As an example, if a distance along an intersection ray is expected to be less-than-or-equal-to one for it to be an intersection point, it should instead be checked against (one + Epsilon) to account for this fuzziness [24, pp. 19,169].

This can however also lead to false positives, where an intersection is found which might not be there at all. While an analytical analysis of such Epsilon values is theoretically possible, choosing a good Epsilon value is often done empirically as the analysis can be very complex [25]. The implementation uses an Epsilon value of 1^{-5}.

These instability issues are not solved by using the Fixed-point format, as it also suffers from imprecision (due to the limited number of bits available). The error (maximum discrepancy) between the theoretical expected value and the actual computed value is however fixed for the entire range of values, which is not the case for Floating-point. This can lead to higher stability, especially when working with a large range of numbers, as the Epsilon value can simply be set to this maximum discrepancy. For the reasons listed above however, Fixed-point numbers have not been implemented for this reason and remain a possibility for Future Work.

4.3. The interface

The interface (*API*) sits between the user program (usually the game software) and the internal parts of the solution. It enables communication between the two software parts. The interface should be kept as simple as possible to allow for easy integration into the user program. It should also be portable into different programming languages.

The interface only needs to support two high-level operations, supplying the input (the *world*) and querying the output. The user program does not need access to the internal state data of the algorithm.

A *world* is just exposed as an opaque handle through the interface. The user program can then create *anchors* and *delimiter objects* using this world handle. An anchor just takes in the position as three coordinate values, a delimiter object requires a transform (position, rotation and size) as well as the hierarchy level. Both anchors and delimiter objects are exposed as integer values (IDs) in the interface. The interface also allows for the attachment of *delimiter planes* to delimiter objects using this ID. Delimiter planes additionally require a parameter to specify its normal axis, whether it is centered or extruded onto the object's face, and which axis to virtually extend it on. After all anchors and delimiters have been created, the user program can finally calculate the volumes using one API call.

After the volumes have been calculated, the user program can query a point using the world handle and the coordinate values. The interface will return the integral ID of the anchor to which the point has been assigned, or a special value (-1) indicating that the point does not lie within any volume.

The interface also exposes *debug drawing* functionality. This means that the user program can request what it would like to visualize (anchors, volumes, delimiter planes and so on) and the solution will return a set of primitives that represent the requested data. Such primitives can be lines, triangles or cuboids. The user program must then draw these primitives themselves, to keep the implementation agnostic of any graphical API or game engine. This enables easy visualization of the results in any software environment (for example to debug the calculated volumes), as drawing primitives is trivial in most game engines [26]. Many figures in this thesis (such as Figure 21 and Figure 23) were created using this functionality.

Finally, the interface exposes functionality to destroy all data stored for this world after the user program no longer requires it.

4.4. The underlying data structure

The algorithm requires storing a lot of data about all objects in the world. This data does not need to be exposed through the interface and can therefore be suited exactly to the algorithm's needs.

All data is stored inside a *world* object. This world owns all active anchors and delimiter objects, as well as the root delimiter planes. A world may additionally store acceleration structures or helpers for memory management. Every anchor and delimiter object in the world stores its ID as well as the input specification that was passed through the interface for the creation of this object.

Every anchor stores its volume as a flat array of triangles, where each triangle consists of three vertices (points in three-dimensional space). This allows for fast iteration over all triangles when ray-casting against a volume.

Every delimiter object owns a list of up to 6 delimiter planes. A delimiter plane is represented by its normal, its origin (for distance heuristic calculations) and a flat list of triangles. Similarly to volumes, this allows for fast iteration over all triangles in the plane.

4.5. Tessellation

A big chunk of the complexity of this solution is clipping delimiter planes correctly. This part relies on splitting triangles to fulfil certain criteria, a process that is called *Tessellation*. The criterion in this thesis is that two triangles may not intersect along any edge. Tessellation happens while solving an intersection between two delimiter planes (see the chapter Solving an intersection). Every triangle of both planes is checked against any triangle of the respective other plane for intersection. The tessellation procedure therefore takes two triangles as input, the triangle T that is to be tessellated and the triangle C with which an intersection may occur.

Two triangles that are co-planar are considered invalid input in this scenario, leading to an early exit of the tessellation procedure, as that would imply two different co-planar delimiter planes in the world (which represents invalid input to the algorithm). Two triangles that are not co-planar but are parallel to each other cannot intersect and therefore also lead to an early exit of the tessellation.

There are three possible cases between two non-parallel triangles: They can either not intersect at all, intersect in a single point, or intersect along a line segment [27]. If they do not intersect at all, no tessellation is required. If they only intersect in a single point,

no tessellation is required either, as no area of the triangle T crosses the triangle C. If there is an intersection along a line segment (“intersection edge”), then the triangle T is split into two different areas by triangle C, therefore requiring tessellation.

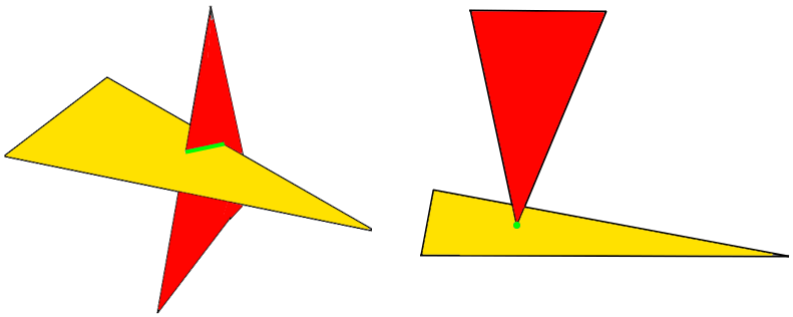


Figure 17: Two triangles on the left intersect along an edge, therefore requiring tessellation. The two triangles on the right only intersect in a single point.

4.5.1. Finding the intersection edge

The intersection edge can be determined by checking for intersections between all 6 edges against the respective other triangle to calculate all intersection points. After de-duplication of similar intersection points, the intersection type can be inferred. An intersection edge only exists if two intersection points remain, the other cases (none or only one point) can be dismissed at this stage. One (distinct) intersection point occurs if a vertex of a triangle exactly touches the other triangle, in which case two edges might report an intersection for a duplicate intersection point. As stated before, this case does not require tessellation.

4.5.2. Performing Tessellation

After the intersection points have been found, the tessellation’s job is to replace the input triangle T with new triangles which represent the original shape, but all meet the criterion of not penetrating the clipping triangle C. The assumption of one intersection edge means that between the input triangle is split up into between 1 and 5 new triangles, depending on the intersection edge.

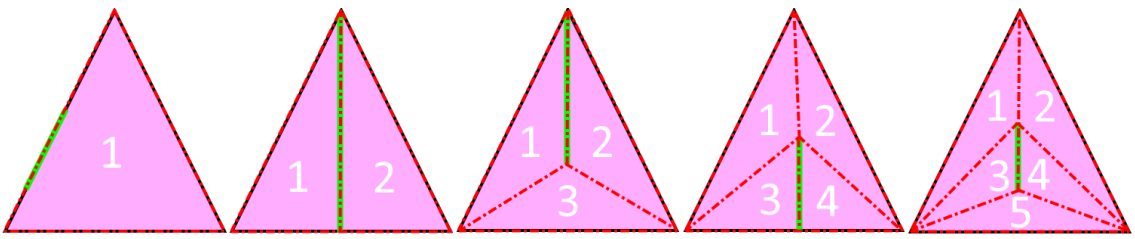


Figure 18: A triangle might be split up into up to 5 sub triangles. The intersection edge is indicated in green. The sub triangles are indicated by the dotted red lines.

The implementation may always enqueue five triangles for generation but can check whether the triangle has an area of (approximately) zero, in which case it would be

discarded. As an example, the third triangle in the center figure would be enqueued in its left neighbor figure as a triangle with zero height (and therefore zero area).

There is no unique solution for tessellating triangles in this scenario. The implementation in this thesis implements a simple approach. This approach is not optimal, as it may generate more triangles than necessary, or generate triangles that are extremely skewed and may lead to numerical precision issues. Research into more optimal tessellation procedures can be done in the future.

The tessellation algorithm finds the vertex that has the shortest distance to the intersection line (where the intersection line is the infinite expansion of the intersection edge). The algorithm then subdivides the input triangle by queuing five pre-defined triangles from the pool of points, such as seen in Figure 19.

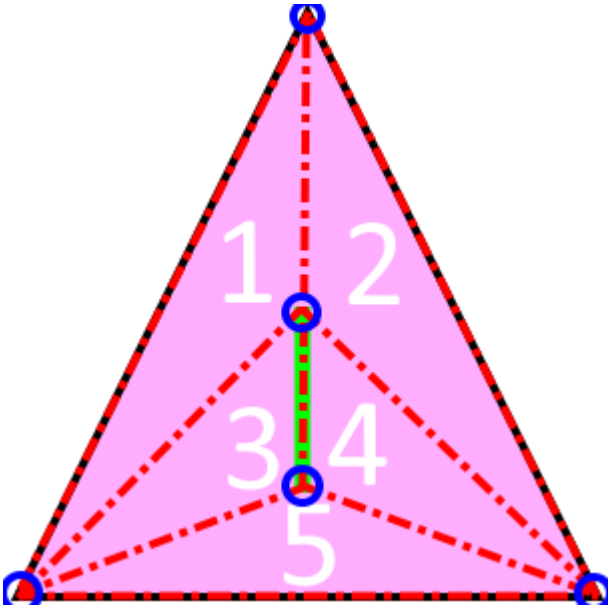


Figure 19: Showcase of the general tessellation algorithm.

This vertex pool consists of the three input vertices and the two intersection points. Connecting them in the fixed order (by extending the intersection edge towards an outer vertex), ensures that the generated triangles never overlap, always represent the original shape of the input triangle, and never penetrate the intersection edge. Of the five queued triangles, up to four may be discarded due to an empty surface area (such as in Figure 18).

4.6. Parallelization

One method for improving the time performance of an algorithm is to apply parallelization. For this, the workload gets divided into several jobs that can be run concurrently on different (hardware) threads. For good scalability, jobs should be independent from one another (so that they can actually run concurrently and do not have to wait on each other), and multiple cores must be available on the system. The job system implemented in this system is loosely based on the concept described by Gregory [15, pp. 549-558].

The build-up phase of the solution consists of two major tasks: Clipping all delimiters in the world (see chapter 3.5) and setting up the volumes using the clipped delimiters (see chapter 3.6).

Clipping delimiters is a task that does not offer good scalability through concurrency. As a reminder, it first needs to find intersections between all pairs of delimiters in the world. Then it sorts the intersections and finally solves them sequentially. While finding intersections could in theory be parallelized, it already takes up a very small percentage of the computation time (around 0.1% of the total build-up time) and the overhead of required synchronization may not actually pay off. For the same reasons, sorting the list of intersections has also not been parallelized. Finally solving the intersections must be done in a sequential order, meaning parallelization cannot happen at this stage (at least not without large additional effort of ensuring no side effects). Figure 7 shows an example of what happens if intersections are not solved in the correct sequential order that they have been sorted it.

While the task of clipping delimiters therefore has been deemed unviable for parallelization in this implementation, the task of calculating the volumes is a perfect fit. Every anchor modifies only its own volume using the existing delimiters as input. Two anchors do not depend on each other in any way, nor does the input change at this stage of the build-up. Two anchors can therefore be handled concurrently without any additional work required.

After the delimiters have been clipped, the solution spins up a number of desired hardware threads (typically the number of logical cores available in the system). Each thread then iteratively takes ownership of one of the remaining anchors and calculates its volume, until no more anchors are left. This ensures that the workload is spread as evenly as possible across all threads, since the execution time for an anchor can vary a lot depending on the input. After all anchors have had their volumes calculated, the threads are no longer used and can be shut down.

This effort of parallelizing the step of calculating volumes had a massive boost of performance, cutting the required time down by almost half for large words where most of the execution time is spent in assembling the volume triangles (see chapter 3.6.1).

4.6.1. Possible Improvements

As mentioned above, more work could be parallelized to decrease the required execution time even further, requiring an effort to ensure correct synchronization and behavior. Additionally, the overhead for assigning anchors to each thread might be improved using a lock-free data structure (instead of a queue using a mutex as is currently implemented). These micro-optimizations have however been deemed not worthy of implementing for this thesis, as the time gained is expected to be marginal compared to the performance of other sections of the code.

4.7. Floodfilling

A Floodfilling algorithm is used for finding the query points when calculating volumes (see chapter 3.6.2). Floodfilling is a very common algorithm, especially in computer graphics, and has been described as early as 1981 [10]. It can be implemented as follows:

The algorithm divides the world into a three-dimensional grid of even sizes. This grid is stored as a three-dimensional array. All cells of this grid are initially unmarked. The frontier is a stack of cells that should be expanded in the future. At the start of the algorithm, one origin cell is added to the frontier (in this solution, the origin is the cell that corresponds to the anchor position).

The algorithm then pops the first cell of the frontier and expands it as far as possible. This is repeated until the frontier is empty, at which point all cells that are reachable from the origin have been marked as such. Expanding one cell means checking if any of the six adjacent cells are reachable from this one, and if so, marking them and adding them to the frontier. Cell B is reachable from cell A in this thesis if a ray-cast from the world space position of A to the world space position of B does not intersect with any delimiters.

Additionally, all cells that have been marked are internally added to a list. This is not required for the Floodfilling algorithm itself, but it makes iterating over all marked cells much faster. This is required for assembling the triangles later, where all marked cells are *query points*.

4.7.1. Possible Improvements

The Floodfilling implemented is very simple and not at all optimized. A faster alternative may be to apply the Jump Flooding Algorithm [28]. This approach does not require a frontier, using less memory and making it applicable for computation on the GPU. The implementation for Jump Flooding is however much more complex than the naïve approach in this thesis and has therefore not been implemented.

4.8. Integration into the Unity Engine

As mentioned in the Overview of this chapter, this solution has been programmed in a way to be independent of any existing piece of software so that it can be integrated into other software solutions more easily.

One such existing software solution is the Unity Game Engine. It is one of the most common game engines and support has therefore been packaged into the implementation provided with this thesis. The interface, as described in chapter 4.3, already supports bindings for different programming languages (Unity uses C# for scripting). The supplied integration additionally provides pre-built Components for Delimiters and Anchors, as well as a manager script for setting up and querying a world for the currently loaded Unity Scene.

The component-based architecture of Unity's Game Object fits well into the solution's idea of Delimiters and Anchors, so that a world can be set up using just drag and drop elements of the Unity Editor. The developer can set up the requested delimiter planes via the Inspector and position them in the world using the Scene View. The manager script infers a delimiter's position, rotation and scale from the attached Game Objects transform and attached mesh. Similarly, anchors can be created and positioned in the scene. The manager script can then query the world and return the anchor component that a query point resides in.

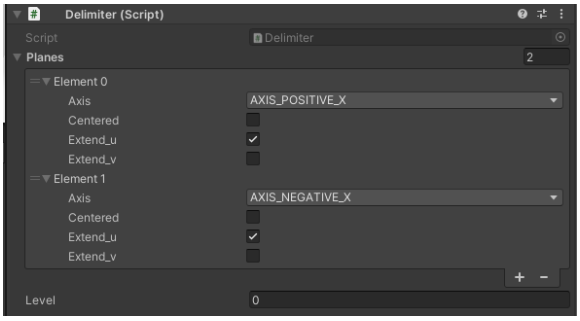


Figure 20: The Delimiter Component exposed in the Inspector.

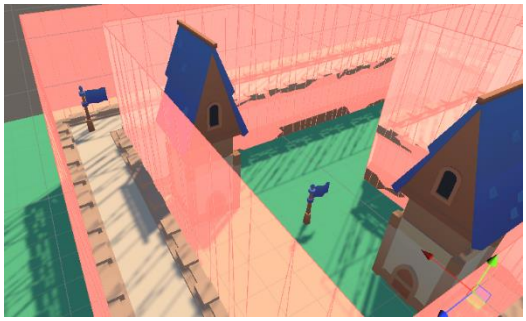


Figure 21: The Delimiter planes rendered in the Scene View.

4.9. Code

The actual code of the supplied implementation can be found on: https://github.com/surrealtm/Thesis/tree/main/Code/Core. Alternatively, an email can be sent to the author (victor.matheke@t-online.de) to obtain it.

5. Assessment

After the algorithm for building up the volumes and querying has been designed and implemented, this chapter will assess the solution on certain criteria and compare it with Pfaffinger's Thesis [5].

5.1. Test Cases

During development of the solution, many test cases have been developed to ensure the validity of the calculated volumes in certain edge cases. Some of these can be seen in Figure 22.

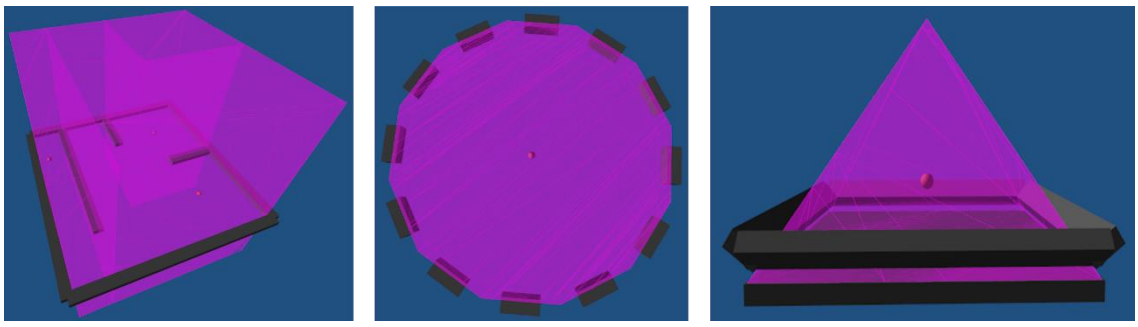


Figure 22: Predefined test cases to ensure the validity of the algorithm.

Additionally, a larger world has been set up inside the Unity Engine to test the behavior and performance of the solution with a larger input. The world contains 19 anchors and 330 delimiter objects. The algorithm calculates the expected output for this world.

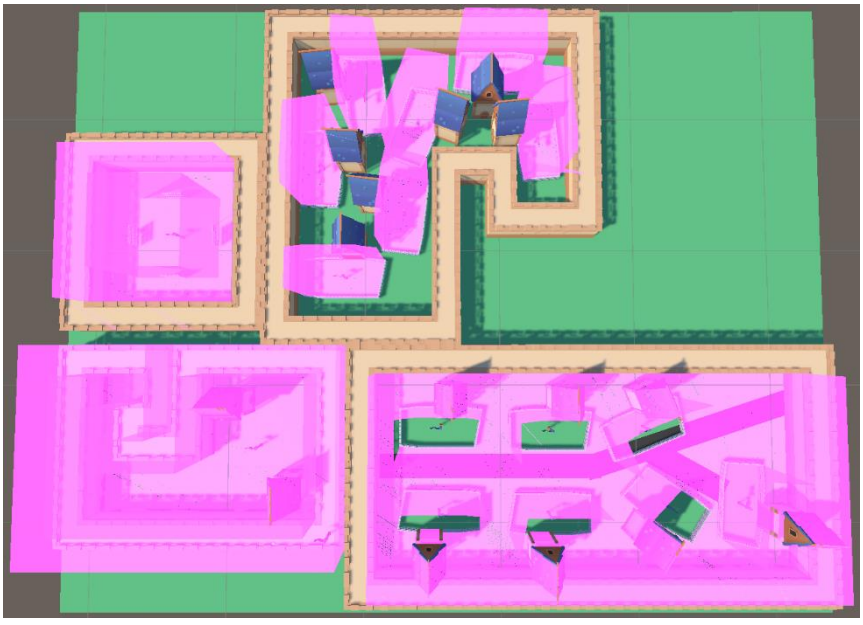


Figure 23: A complex world used for testing.

5.2. Fulfillment of the requirements

This chapter will go through the requirements that were initially stated in the introduction and check for their fulfillment in the proposed solution.

5.2.1. Determinism

The algorithm for calculating the volumes has been designed to be deterministic. Important steps for this are ensuring that the order in which delimiters are present in the world's internal storage is not important, instead intersections between delimiters are sorted before they are solved (chapter 3.5.3). Similarly, the order in which anchors are stored does not matter, as they do not interact with each other and only depend on the present delimiters.

However, the issue of floating-point representation may lead to instability issues, such as further described in chapters 4.2 and 5.4. This numerical instability can lead to issues with determinism when slightly changing input, as a change may lead to different results in tessellation, ray-casting or Floodfilling.

5.2.2. Performance

A more in-depth performance analysis is done in chapter 5.5. In short, the requirement of real-time execution of queries has been met for the complex world shown in Figure 23.

5.2.3. Requirements towards Anchor Volumes

The requirements towards an anchor's volume have been met in theory. The chapter 3 goes in the detail of fulfilling these requirements. In practice, numerical stability may lead to volumes not being fully enclosed, as very small triangles may be missing from the representation (if they have been rejected during tessellation). An example for this can be seen in Figure 24. This may be improved by either using Fixed-point calculations or applying mesh optimizations (see chapters 4.2 and 6.2). The chapter 5.4 goes into more detail on the issue of instability in this implementation.

Figure 24: Gaps (indicated in blue) in a calculated volume (indicated in pink) due to numerical imprecision. The blue gaps are caused by missing triangles in the volume's representation.

5.2.4. Predictability

The algorithm has been designed to be predictable to game developers. However, such predictability may be very subjective and would require an extensive field study in real-world application. Such a study has not been done in the scope of this thesis but may bring useful results in the future.

5.2.5. Output error

The discrepancy between the expected and the calculated output is reduced to floating-point imprecision. Volumes expand as far as the delimiters allow them, as the volume is made up of triangles from the delimiters. The output error is therefore minimal.

5.2.6. Integration into existing tools

The integration into the Unity Engine has been supplied with the implementation of the core solution (see chapter 4.8). Additionally, the solution has also been used in a program to visualize the results. The requirement of easy integration has therefore been deemed as fulfilled.

5.3. Malformed Input

There are constellations of input which are considered to be malformed, as no well-defined way of handling them can be conceived. The two major edge cases are the following:

1. Two delimiter planes are coplanar and overlap each other. As they do not intersect along a line, there is no well-defined strategy for clipping them against each other using tessellation (as described in chapters 3.5 and 4.5). Instead, an arbitrary intersection edge along which to solve the overlap would need to be found. This arbitrariness might however contradict the requirements of determinism and predictability. The user must instead ensure that such no coplanar delimiter planes overlap. Therefore, two delimiter planes that are coplanar do not result in an intersection to solve and their representations are not adapted. This can lead to anchor volumes storing overlapping triangles, which will result in incorrect queries due to the Even-Odd-Rule requiring non-overlapping triangles (see chapter 3.7).

2. Sets of anchors that are not fully delimited by delimiter planes will intersect with each other, as anchors grow their volume until it is stopped by delimiter planes, not by other anchors. This means that if the input does not contain enough delimiters to always separate two anchors, these anchors will overlap. This is one key difference to the approach pursued by Pfaffinger [5] (see chapter 5.6.1). This

leads to issues with querying the semantics of a point in space, as one point is expected to reside in (at most) one volume. With overlapping volumes, one point may reside in multiple volumes at once, which cannot be expressed through the interface of the implementation and only one of the multiple volumes would be reported back to the user program. In theory, the interface (and the query execution) could be adapted to return a list of volumes that the point resides in for this edge-case. However, one assumption in the design of this solution was that anchor volumes do not overlap, and therefore each point in space has either none or exactly one volume it belongs to, which usually fits nicely into game rule evaluation. Therefore, this case is considered malformed input as well and the user must ensure that no such input is passed to the solution.

Such malformed input should in the future be reported back to the user of the solution, so that they can change the input to fix these issues. This has, however, not been implemented yet.

5.4. Numerical Instability

A big issue when dealing with numerical problems (as in this thesis) is dealing with instability of floating-point numbers. A more in-depth explanation of this issue has been described in chapter 4.2, as well as potential improvements to this problem.

However, this implementation uses floating-point representation and therefore requires careful handling of floating-point precision. This has not always been achieved in the implementation, which can lead to instability of the algorithm. Such instability can manifest itself in missing triangles for anchor volumes (such as seen in Figure 24) or invalid representations of delimiter planes, which in turn can lead to point queries returning wrong results.

Thorough attempts have been made to improve stability for certain inputs with this implementation, but not all could be fixed, so that the provided implementation still suffers from this problem.

This issue of numerical instability is an implementation detail and therefore not a fundamental issue of the approach. Therefore, the theoretical approach as described in chapter 3 has therefore not been deemed unviable due to this, but instead more effort is required in the future to improve this part of the implementation.

5.5. Performance

The performance of the solution is split into two parts, the build-up of the volumes and the querying of a single point in the world. The performance of both parts, however, depends on many factors, such as the number of anchors, the number of delimiters, the step size of the Floodfilling algorithm, the size of the (calculated) volumes, the output created by the tessellation (as the number of triangles plays a factor in many parts), etc. It is therefore inconceivable to create a theoretical run-time analysis.

Additionally, the implementation provided with this thesis is far from optimized. Many possible algorithmic optimizations have been pointed out in chapters 4 and 6, and many more micro-optimizations are possible by improving the individual code. As stated in the introduction, performance (especially for the build-up stage) has not been a primary concern for this thesis.

Nevertheless, some performance statistics will be given in this chapter to give a rough impression of the run-time of this implementation. All performance figures were measured on a home computer with a CPU of 12 logical cores running at 3.70GHz and 16GB of 3200MHz main memory. The performance tests were all taken on the complex world shown in Figure 23.

Calculating the volumes for the complex world takes around 36 seconds, with a peaking memory usage of 34MB. The final output of the build-up stage occupies around 22MB of memory. As the build-up stage is expected to run offline, the run-time of 36 seconds is acceptable. Modern games can often require multiple gigabytes of memory anyway, so the memory cost is also deemed acceptable.

Querying the volumes for a single point takes between 0.05ms and 0.16ms, depending on which volume the point resides in (as the algorithm takes an early-exit once the volume has been found). This time can be heavily optimized by implementing the optimizations listed in 3.7.2, which should make it applicable for real-time games.

5.6. Comparison with Pfaffinger's Approach

A thesis for the same problem statement has already been referenced in the chapter 2.2 [5]. This chapter will compare the two implementations.

The two major differences in the approaches are that Pfaffinger's approach does not support Virtually Extending delimiter planes, and that volumes are calculated using Marching Cubes instead of finding triangles that represent the volume.

5.6.1. Discrepancy of the output

The implementation provided with this thesis also supports using Marching Cubes for volume calculation when setting the appropriate switch. Using Marching Cubes leads to a much higher discrepancy between the expected and the calculated output due to voxelization of the space. An example for this can be seen in Figure 25.

The Marching Cubes implementation in this thesis is not the same as in Pfaffinger's thesis, namely is there no Voronoi growing of anchors (meaning anchors without a delimiter between them would not stop each other from growing).

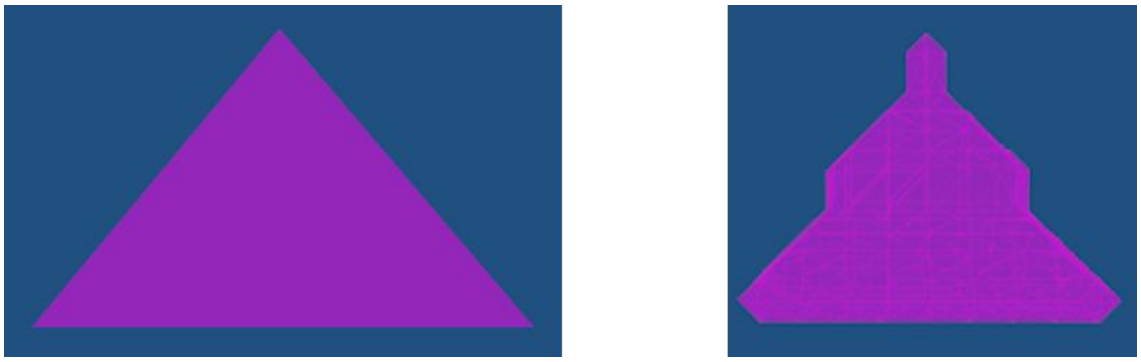


Figure 25: Comparison of the Marching Cubes approach versus the Assembling of triangles.

5.6.2. Resolving Underspecification

In Pfaffinger's approach of Marching Cubes, the volume of an anchor grows until it either hits a delimiter or another anchor, as a way of resolving underspecification of delimiters. This ensures that two volumes never grow inward of each other, effectively making all anchors simultaneously act as growing delimiters. Such a strong guarantee cannot be made in the implementation proposed in this thesis, as growing anchors until they hit one another is made possible by the discrete step size of Marching Cubes which does not exist in this thesis. Instead, underspecified input which leads to overlapping volumes is considered invalid and can lead to issues in the evaluation (e.g. violation of the requirement that volumes do not intersect each other).

In practice, one might prefer one or the other approach for resolving underspecification. Pfaffinger's approach guarantees that volumes stop each other from growing (even without any delimiters), their line of intersection however is determined by the algorithm. If that is not what the user of the solution desires (as may often be the case in practice), they would need to manually place delimiter anyways.

Figure 26: Volumes as calculated by Pfaffinger's approach. [2]

5.6.3. Performance

In theory, using Marching Cubes to calculate the volumes should be faster than the costly Assembling step as described in chapter 3.6, as the Assembly of the triangles includes a Floodfilling (which is the base for Marching Cubes as well), but then requires a lot of ray-triangle intersection tests (whereas Marching Cubes simply builds the mesh based on the Floodfilling results).

As mentioned before, the implementation provided with this thesis also supports Marching Cubes for calculating the volumes. Comparing the performance between using Marching Cubes and Assembling the triangles in a complex scene shows (Figure 23) that the build-up stage is around 15 times faster using Marching Cubes compared to Assembling the triangles. No comparison has been made to Pfaffinger's implementation, since there are slight differences in the approach, and performance measures highly depend on the code itself, as well as on the environment in which the code was run.

5.6.4. Trade-Off

This trade-off between discrepancy and performance is therefore a relevant point of discussion when using such an algorithm in real use cases. Games that have a dynamic component (such as the destruction of delimiters) may use the Marching Cubes approach to attempt to recalculate the volumes during run-time of the game. Games that require high accuracy of the calculated volumes may instead prefer the Assembling stage as proposed in this thesis.

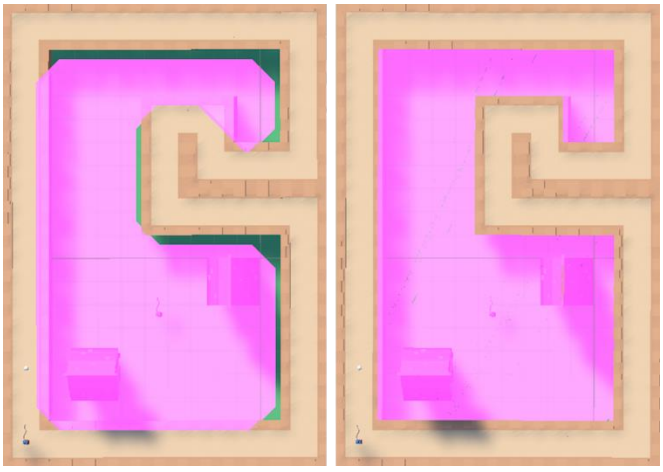


Figure 27: The volume on the left was calculated with Marching Cubes, approximately 15 times faster than the one on the right.

5.7. Feasibility of the approach

This thesis has shown that a solution for the described problem can be implemented, and that the proposed requirements are met by the design. The thesis provides potential

solutions to the issues that remain in the supplied implementation, as well as additional features which may need to be implemented in the future for real-world use.

However, games often have very specialized requirements for the software solutions that they use, to fulfill their specific needs which may be unique to their game. Therefore, a generalized software solution (such as this thesis) may not always prove useful in practice, if the downsides outweigh the upsides. The practicality of an algorithm such as this would therefore require an extensive user study with real world projects.

The two major points of interest for such a study would be whether the cost of integrating the solution into a project, and whether the input options allow for enough customization that the project can get the desired output. For some games, it might require less work to manually set up the anchor volumes than to integrate the solution into the project, set up the input so that the expected output is created, and then run the actual algorithm. Other games may employ additional logic to the calculation of volumes, such as a Voronoi algorithm (as in Pfaffinger's Thesis [5]), or more dynamic components.

In summary, the design is feasible and may act as a base for implementing such solutions (potentially tailored to more specific requirements) in the future.

6. Future Work

The goal of this thesis was to show the feasibility of a solution for the given problem statement. It has however not been thoroughly tested in real-world applications. The implementation is therefore not of commercial grade in terms of feature completeness. Such a study in the future would provide more insight into the fulfillment of the requirements (both functional and non-functional, such as the predictability).

In the previous chapters, many minor possible improvements have already been pointed out (for example using Fixed-point calculations in chapter 4.2). This chapter will list some more complex features that have not been implemented.

6.1. Serialization

In real-world applications, developers would like to serialize the calculated volumes after they have been calculated once to save start-up time. Loading the internal representations of all volumes in the world should be much faster than having to go through the entire build-up phase of the algorithm. It would also mean that the input data for the world (such as the requested delimiter planes) do not have to be stored in packaged builds.

A serialization system would also enable the game to stream in and out chunks of the world which are currently needed, as is often done in open world games. The world of such games usually does not fit into memory all at once, so the game streams in chunks (“parts”) of the world which the player should currently see, and streams out chunks that the player no longer sees. With a serialization system, this can easily be applied to the volumes calculated by this algorithm as well.

6.2. Mesh optimization

One major flaw of the implementation as provided is the massive number of triangles ending up in the representation of delimiter planes or volumes. This happens due to the tessellation of triangles during delimiter clipping. The triangles making up the delimiters are reused for the volume representation of anchors.

If the solution implemented a step of improving the delimiter plane representation by reducing the number of triangles while keeping the original shape of the plane, the results should have a much lower triangle count. This would lead to decreased memory consumption and better time performance due to less required ray-triangle intersection

calculations. It might also increase stability of the algorithm, as extremely small triangles (such as are present right now) might lead to numerical imprecision.

This optimization may either be implemented from scratch by attempting to combine multiple triangles together, or an existing software solution may be integrated into the solution for this purpose. A possible implementation is described in the paper *Mesh Optimization* [29].

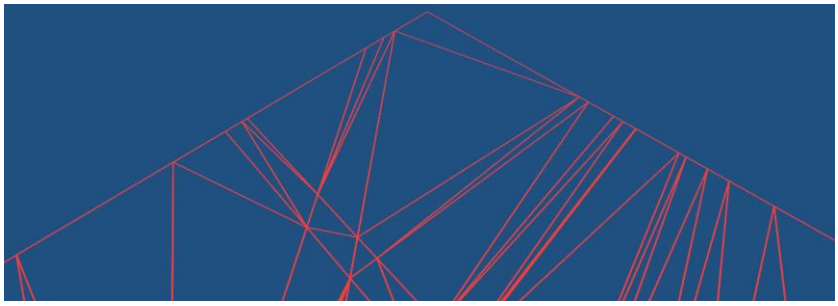


Figure 28: The unoptimized triangle representation of a delimiter plane.

6.3. Spatial Relations

The system proposed in the “Space Foundation System” [4] also includes storing adjacency between anchors. This enables the construction of a “Location Graph”, which can be used both for game design purposes but also for rule evaluation. This also enables pathfinding on the Location Graph, for example to find the shortest path between two non-adjacent volumes, which can then be shown to the player.

Spatial relations between anchor volumes can be determined by finding delimiter objects that are shared as a bordering delimiter by different anchors. Two anchors are adjacent if they are delimited in any direction by the same delimiter object. This information is stored as an edge between the nodes that represent the anchors in the location graph.

6.4. Partial Rebuilds

Games that have a highly dynamic component (such as the destruction of delimiters through gameplay) may require rebuilding the volumes for affected anchors at run-time. As the build-up of the data structure for large worlds has been deemed unviable in real-time, this might require building the complete world offline and then only rebuilding the parts of the world that have been affected by the change. This might decrease the time required for adapting the volume representation, making it viable for such dynamic games.

7. Conclusion

This thesis has proposed, implemented and evaluated a software system to algorithmically create a mapping from continuous space into a discrete semantic one. The solution takes in a set of anchors and delimiter objects as input for the build-up stage, in which the mapping is calculated. Afterwards, the solution allows for the querying of points in the continuous space in the second stage (usually during run-time), for which the mapped anchor of any given point is returned.

The two stages of the solution have been designed to fulfill the requirements that were listed in the problem statement, which in turn were derived from the desired properties of such a mapping in real world projects. The designed solution has been assessed as incomplete but feasible, due to several issues present in the provided implementation, as well as missing features that were listed in this thesis. The initial requirements towards the solution however have been fulfilled, and concepts for implementing the remaining work have been given.

The proposed software system aims at game developers but may be useful in other disciplines such as architecture or navigation. It has been implemented to be agnostic of any specific piece of software for ease of integration into real world projects. The implementation also provided a same game project inside the Unity Game Engine.

The approach proposed in this thesis is compared to the approach that was pursued by Pfaffinger [5], and a trade-off between the solutions is discussed. A major improvement was achieved in the high precision with which the volumes are calculated, although the provided implementation suffers from more instability.

The goal of the thesis has been shown by proving the feasibility of such a solution, and acts as a milestone in the development of a broader, more complex system by providing insight into remaining issues and potential improvements in the future.

List of figures

Bibliography

[1] C. W. Totten, Architectural Approach to Level Design, CRC Press, 2019.

[2] D. Dyrda and C. Belloni, "Game Engineering: Formalizing Experience Based on Pacing & the Topology of Play," *unpublished.*

[3] Unity Technologies, "Unity - Manual: Scenes," [Online]. Available: https://docs.unity3d.com/Manual/CreatingScenes.html. [Accessed 08 09 2024].

[4] D. Dyrda and C. Belloni, "Space Foundation System: An Approach to Spatial," 2024.

[5] K. Pfaffinger, "Anchors and Boundaries: Developing a Game Engine System for Hierarchical Spatial Partitioning of Gamespaces," 2024.

[6] T. S. Newman and H. Yi, "A survey of the marching cubes algorithm," *Computers & Graphics,* vol. 30, no. 5, pp. 854-879, 2006.

[7] M. Fleischmann and D. Arribas-Bel, "Spatial Signatures-Understanding (urban) spaces through form and function," *Habitat International,* vol. 128, 2022.

[8] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe and J. Bittner, "A Survey on Bounding Volume Hierarchies for Ray Tracing," *Computer Graphics Forum,* vol. 40, no. 2, pp. 683-712, 2021.

[9] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," *Proceedings of the 7th annual conference on Computer graphics and interactive techniques,* 1980.

[10] T. Pavlidis, "Contour filling in raster graphics," *Proceedings of the 8th annual conference on Computer graphics and interactive techniques,* 1981.

[11] Cambridge Dictionary, "Volume | definition in the Cambridge English Dictionary," [Online]. Available: https://dictionary.cambridge.org/us/dictionary/english/volume. [Accessed 8 8 2024].

[12] Epic Games, "Volumes Reference," [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/volumes-reference?application_version=4.27. [Accessed 8 8 2024].

[13] Unity Technologies, "Unity - Scripting API: Mesh.bounds," [Online]. Available: https://docs.unity3d.com/ScriptReference/Mesh-bounds.html. [Accessed 8 8 2024].

[14] F. D. Ching, Architecture: Form, space, and order, John Wiley & Sons, 2023.

[15] J. Gregory, "Game Engine Architecture," Boca Raton, Taylor & Francis, CRC Press, 2018.

[16] Wikipedia, "Hesse normal form - Wikipedia," 26 March 2024. [Online]. Available: https://en.wikipedia.org/wiki/Hesse_normal_form. [Accessed 30 08 2024].

[17] Unity Technologies, "Unity - Scripting API: Transform," Unity Technologies, [Online]. Available: https://docs.unity3d.com/ScriptReference/Transform.html. [Accessed 19 07 2024].

[18] I. E. Sutherland, S. F. Robert and R. A. Schumacker, "A characterization of ten hidden-surface algorithms.," *ACM Computing Surveys (CSUR),* vol. 6, pp. 12-13, 1974.

[19] L. Yu and C. Tu, "Research on collision detection algorithm based on AABB-OBB bounding volume," *First International Workshop on Education Technology and Computer Science,* vol. 1, 2009.

[20] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 18, no. 2, pp. 139-174, 1996.

[21] C. Inacio and D. Ombres, "The DSP decision: Fixed point or floating?," *IEEE Spectrum,* vol. 33, no. 9, pp. 72-74, 1996.

[22] C. Hecker, "Let's get to the floating point," *Game Developer Magazine,* pp. 19-23, 1996.

[23] Wikipedia, "Floating-point arithmetic," [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/b/b6/FloatingPointPrecisionAugmented.png. [Accessed 17 8 2024].

[24] R. E. Moore and C. T. Yang, "Interval Analysis I.," *Technical Document LMSD-285875, Lockheed Missiles and Space Division, Sunnyvale, CA, USA,* 1959.

[25] B. Mirtich, "Efficient algorithms for two-phase collision detection," *Practical motion planning in robotics: current approaches and future directions,* pp. 203-223, 1997.

[26] Unity Technologies, "Unity - Scripting API: Debug.DrawLine," [Online]. Available: https://docs.unity3d.com/ScriptReference/Debug.DrawLine.html. [Accessed 23 08 2024].

[27] C. L. Sabharwal, J. L. Leopold and D. McGeehan, "Triangle-Triangle Intersection Determination and Classification to Support Qualitative Spatial Reasoning," *Polibits,* vol. 3, pp. 14-15, 2013.

[28] R. Guodong, "Jump Flooding Algorithm on graphics hardware and its applications," 2008.

[29] H. Hoppe, T. DeRose, D. Tom, J. McDonald and W. Stuetzle, "Mesh Optimization," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 1993, pp. 19-26.