Technische Universität München

Algorithmic Subdivision of Gamespaces into Semantic Volumes using Delimiters

Untertitel

Wissenschaftliche Arbeit zur Erlangung des Grades

B.Sc. Informatik: Games Engineering

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

Betreuer/-in	Univ.-Prof. Dr. Dr. h. c. mult. Max Musterprofessor
	Lehrstuhl für Musterlehre
Aufgabensteller/-in	Univ.-Prof. Dr. Dr. h. c. mult. Max Musterprofessor
	Lehrstuhl für Musterlehre
Eingereicht von	Martin Mustermann
	Musterweg 20
	80999 München
	+49 89 123 456 89
Eingereicht am	Ort, den Datum

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.

Ort, Datum, Unterschrift

Abstract

This thesis will propose and analyze an algorithm for deriving semantic volumes from Gamespaces.

Acknowledgements

Contents

1. Introduction

Games commonly require a space to be played in. 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.

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 (usually 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. 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 space 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 under a few conditions which are 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 once by a developer, 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.

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 goal of this thesis is to show the plausibility of such an algorithm and its potential use in game development. The focus is therefore on identifying potential problems in the design, and not on implementing 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” [1]. 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 problem has been written by Kerstin Pfaffinger in 2024 [2]. It pursues a different approach to the very same problem, which is based on the known Marching Cubes 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. Bounding Volume Hierarchies

An acceleration data structure that is commonly used in raytracing applications is called a *Bounding Volume Hierarchy*, or BVH for short. 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” [3] 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.

2.4. 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” [4]. The solution described in this thesis used Flood Filling for approximating the volume of anchors in three dimensions.

2.5. 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” [5]. In video game technology (e.g. Unreal Engine [6], Unity Engine [7]), a Volume often does not only refer to the amount of space, but also to the three-dimensional shape of the referred space.

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. This chapter will present the different intermediate steps of the algorithm, the decisions that were taken during the implementation phase, as well as pointing out issues and potential improvements of the solution.

3.2. Representing 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. Representing Volumes

The goal of the algorithm is to subdivide a three-dimensional space into volumes that represent semantic 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 this algorithm needs to represent three-dimensional shapes, which in computer graphics is often referred to as bounding volumes.

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: The four proposed representations of volumes, from left to rightFigure 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. 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 benefits of fast computation time and a small memory footprint are reduced the more recursion levels are used, making the disadvantages bigger than the advantages.

3. An approach used in the other thesis [2] (referenced in Thesis) is to discretize the continuous the world on a three-dimensional grid. A volume is then represented as a set of grid points which are considered inside the volume. 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. The difference between meshes and volumes in this context is that meshes do not need to be enclosed. Triangles are the simplest three-dimensional shape and can therefore be used to approximate any other three-dimensional shape with less discrepancy than more complex shapes (like cuboids). This results in the least error in the output, especially for concave shapes. Additionally, while the algorithm may be applied to any problem space, its main use is seen in video games. Video games have been using triangulated meshes for hardware-accelerated rendering for the past decades. Both developers and tools are therefore used to triangle meshes, which can aid in the integration.

For the above reasons, a volume is represented by a list of triangles in this thesis.

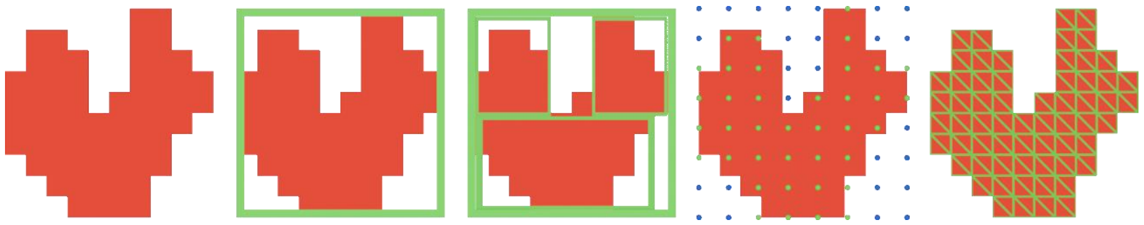
Figure 1: The four proposed representations of volumes, from left to right, with the initial shape on the left.

3.4. Representing Delimiters

Like volumes and anchors, delimiters need a representation inside the algorithm. The concept of a delimiter is divided into two parts in this solution. The first part, called the *delimiter object*, is described by a cuboid transformed in the world, using position, size and rotation vectors (known as a *transform*). Additionally, a delimiter object is assigned a hierarchical level in the world. This numerical level is used when resolving intersections between delimiter planes (explained in Clipping Delimiters).

The second part of a delimiter are the *delimiter planes*. The defining characteristic of a delimiter as described in the problem state is that it subdivides a space into two regions. The actual three-dimensional shape of the delimiter is therefore not the most important characteristic, but instead the evaluation of which side any arbitrary point lies in is. This idea can be represented by using a three-dimensional plane. A point can be on either side of or exactly on the plane, with the latter being an edge-case that must be handled explicitly.

A delimiter object can therefore own up to six delimiter planes, one for each of the faces of the delimiter object's cuboid. Alternatively, planes can also be created for an axis going through the center of the delimiter object, instead of lying on the face of the cuboid. This might be useful if the object is not supposed to have a three-dimensional depth in semantic space, but it does have depth in the continuous world. See Virtually extending Delimiter Planes for an example.

This concept of attaching delimiter planes to delimiter objects has two advantages. Firstly, it simplifies the integration of the algorithm into existing tools for game development, which commonly store a Transform [5] for every object in the world (which acts as a definition for a delimiter object). Secondly, it makes it trivial to model slightly more complex objects of the game world, such as walls. Walls often have an actual depth and would therefore require two delimiter planes to be modelled accurately. If delimiter planes were standalone objects in the world, the game developer would need to move both delimiter planes if they wanted to move the wall. With this setup, they only have to move the delimiter object, and the planes will remain attached to it.

3.4.1. Representing Delimiter Planes

There are a lot of ways to represent planes. A common one is the Hesse Normal Form, where a plane is defined using one origin point and the normal of the plane. Solving a simple mathematical equation gives information on whether an arbitrary point is in front, exactly on, or behind the plane (where “in front” means “in the direction of the plane normal”).

For this algorithm however it is vital that delimiter planes are not infinitely large. The exact reasoning for this is explained in the chapter Clipping Delimiters, but in short, the algorithm requires delimiter planes to be clipped arbitrarily to produce the desired output. Similarly to volumes, triangles have been chosen to represent arbitrary planes in three-dimensional planes. All triangles are guaranteed to be on the same plane, but the plane is no longer assumed to be infinitely large. Although imposing a larger memory footprint than simpler representations of planes, the flexibility provided by this triangulation of a plane is used and relied upon heavily by the implementation of this algorithm.

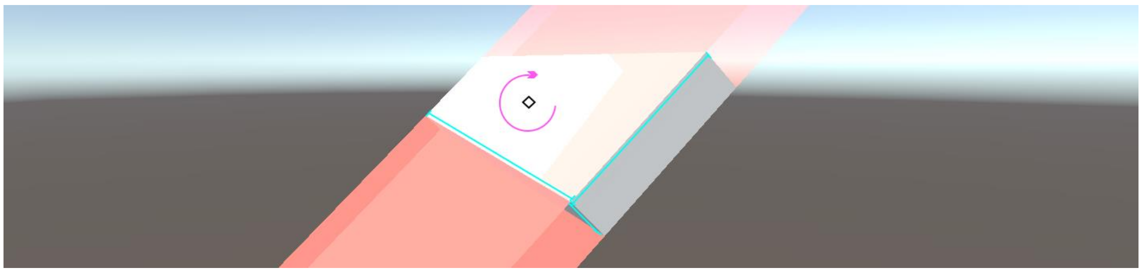

Figure 2: A delimiter object defined by position, rotation and scale, with two virtually extended delimiter planes created on the positive and negative X axis of the cuboid.

3.5. 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 of the algorithm should be kept to the possible minimum. The algorithm 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.

As described in the previous chapter, delimiter planes are attached to delimiter objects. The planes are specified as one of the six faces of the object cuboid. If the plane is not extended, then it will take on exactly the dimensions of the face. If it is extended, the plane will extend as far as possible in all directions until it hits other delimiter planes. 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).

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.

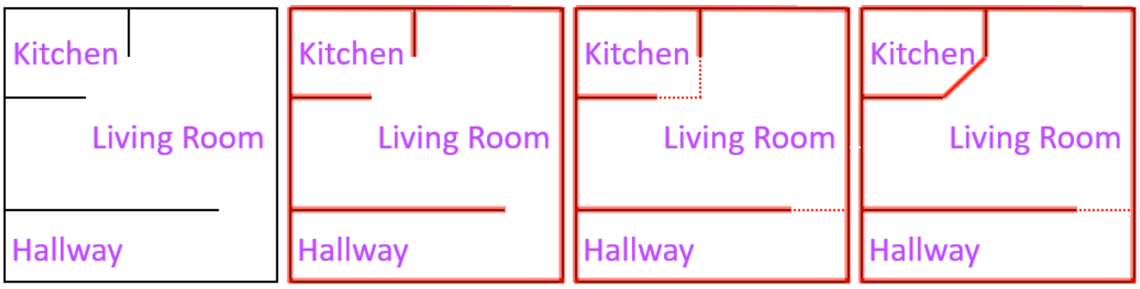

Figure 3: Visual Example of virtually extending Delimiter Planes. Delimiters are displayed from a top-down view.

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

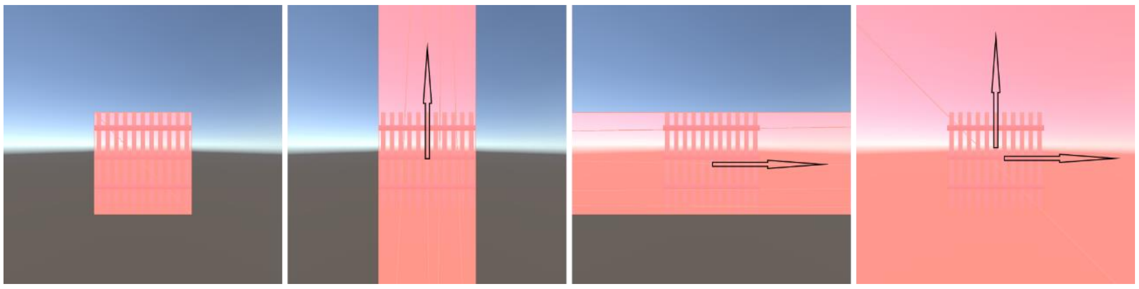

Figure 4: Virtual Extension can be specific for each orthogonal axis of a plane.

3.6. 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 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 manually resolve this specification by adding more delimiters. See Figure 3 for an example of this.

Additionally, the algorithm requires that no triangles of any two delimiter planes intersect each other for the later stages of the build-up. This is further explained in the chapter Calculating Volumes, 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 attached to. 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 to the expected size.

3.6.1. Cutting Delimiters down instead of growing them outward

This approach of clipping the plane down instead of growing it outward might seem un-intuitive 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. It would be impractical to use such a small step size that the plane could be left as it was before the step due to the gap between where the plane was and where it should be. Keeping the plane after the step would leave intersections between delimiters. Therefore, the algorithm would need to solve the penetration between the two delimiters by calculating the intersection edge, and clipping both delimiters so that neither extends beyond that intersection.

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.

3.6.2. 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.

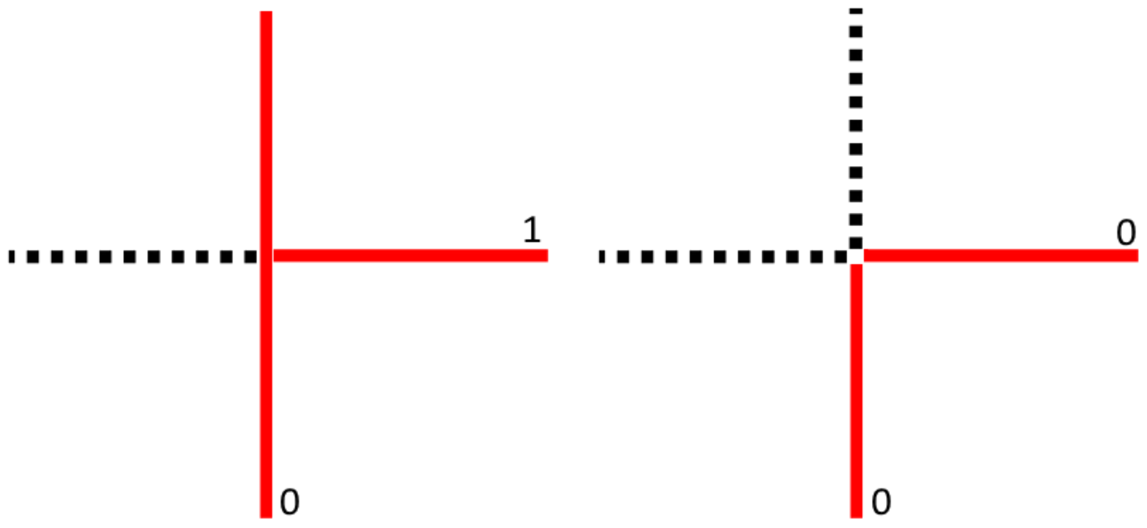

Figure 5: The black part is clipped away from the delimiters whose level is not lower than the other's. Delimiters are seen from a top-down view.

Figure 5 shows the two scenarios when two delimiter planes intersect from a top-down view. 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.

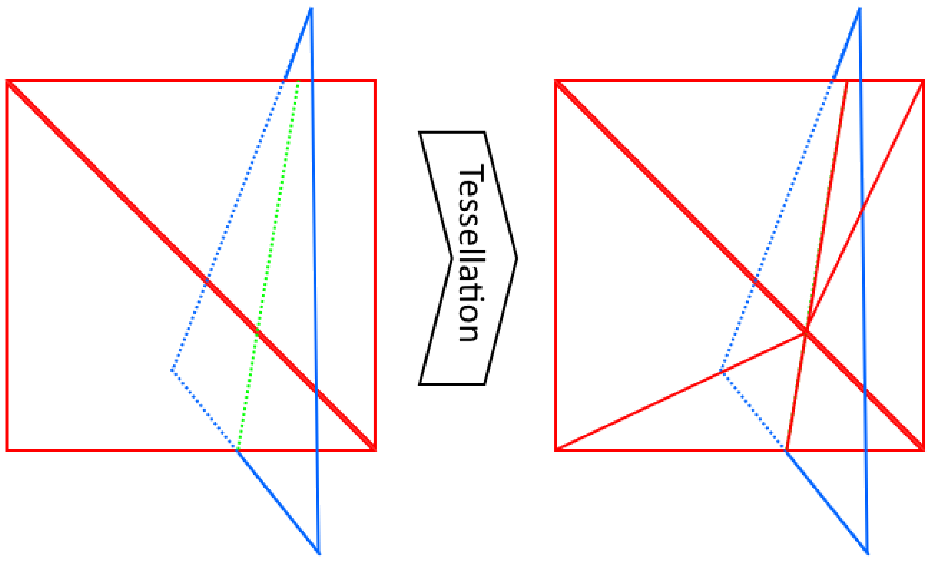

Figure 6: The two delimiters in red and blue intersect along the green axis. The red delimiter plane is tessellated so that no triangle of the plane intersects with the blue triangle anymore. The same must now happen for the blue triangle.

Figure 6 shows an example of how the triangulation of the plane is adapted through a process called “Tessellation” (see Tessellation) 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.

3.6.3. Heuristic for ordering Delimiter intersections

The order in which the delimiter intersections are solved is vital to the predictability of the algorithm. Delimiters are expected to grow outwards until they are stopped, which means we need to resolve the intersections in the order in which they would've occurred in theory. Because the algorithm does not actually grow them outward in practice, but instead cuts them down, this requires a heuristic to determine the sorting order. Figure 7 shows an example result when a bad heuristic is used for sorting the intersections.

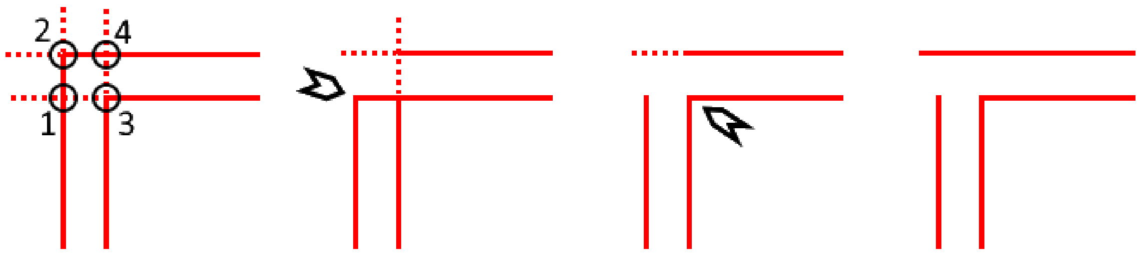

Figure 7: An example of unexpected results if the intersections are not ordered properly. Intersection 1 should be handled after intersection 3. Delimiters are seen from a top-down view.

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.

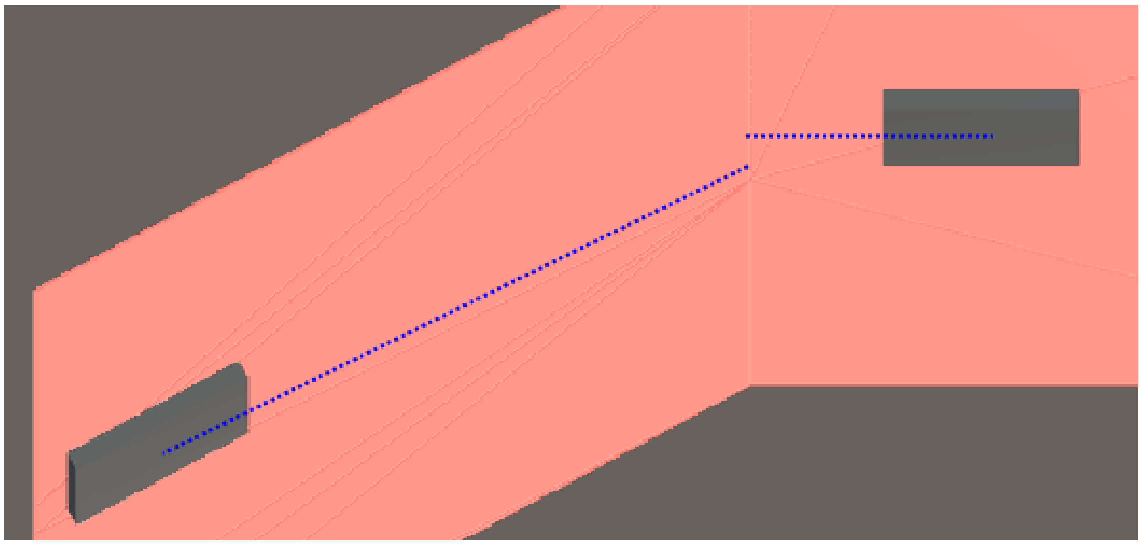

Figure 8: The blue dotted lines represent the heuristic by which intersections are sorted. The lengths of the two lines are summed together to find one distance value.

3.7. Calculating Volumes

After the input has been processed through 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 **Error! Reference source not found.**.

As described in Representing Volumes, a volume is a list of triangles. Adapting the requirements to this assumption leads to the following implications:

1. A triangle of a volume must never intersect a triangle of a delimiter.
2. The triangles that make up a volume must be the triangles of the delimiters that border this volume.
3. The 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 Clipping Delimiters, 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 intersect with any delimiter plane.

This approach also guarantees that the volume representation perfectly matches the expected output, because the volume's 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. 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.

3.7.1. 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. This heuristic initially has an obvious flaw.

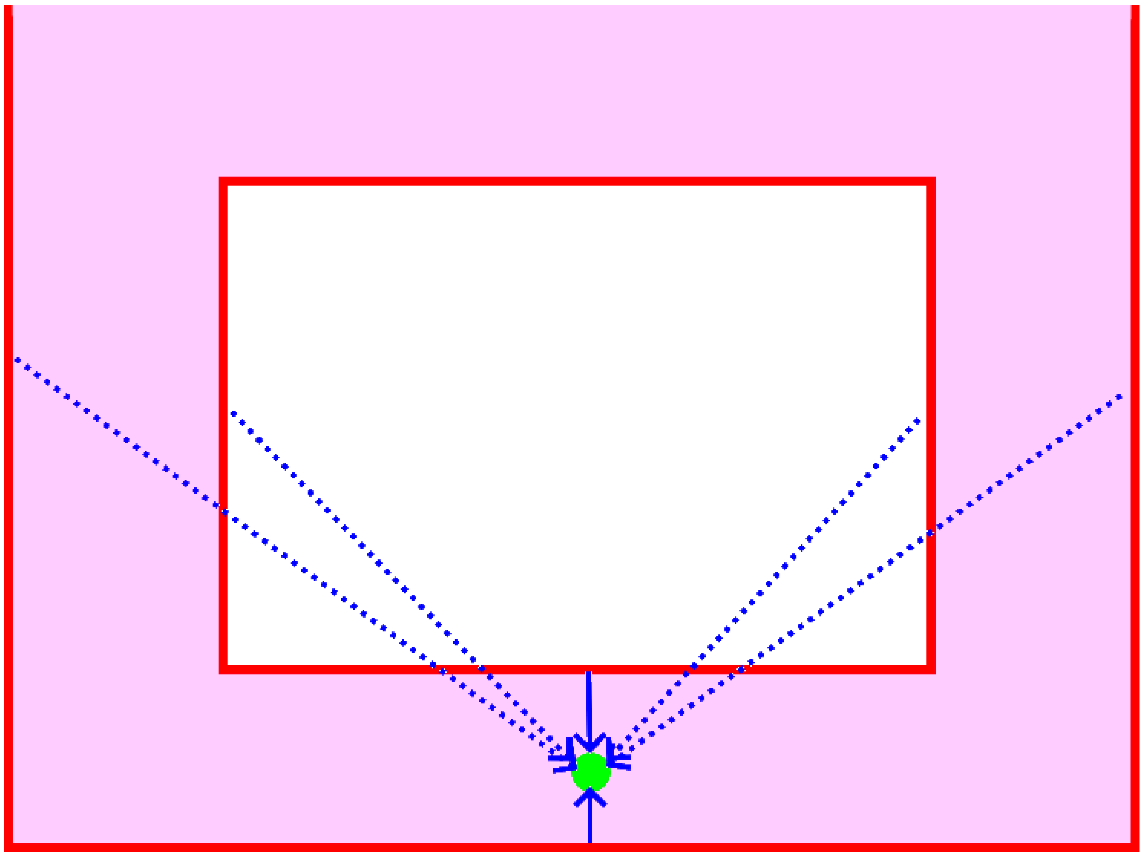

Figure 9: Top-down view of false negatives when calculating anchor volumes. The expected volume is indicated in pink, the anchor in green. Successful rays are indicated by a solid blue line, intersecting rays by a dotted line.

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 point when casting rays from the triangles' centers. This improves the output for concave shapes like the example in Figure 9. With enough additional *query points* to check, all delimiting triangles should be found and added to the volume, as shown in Figure 10.

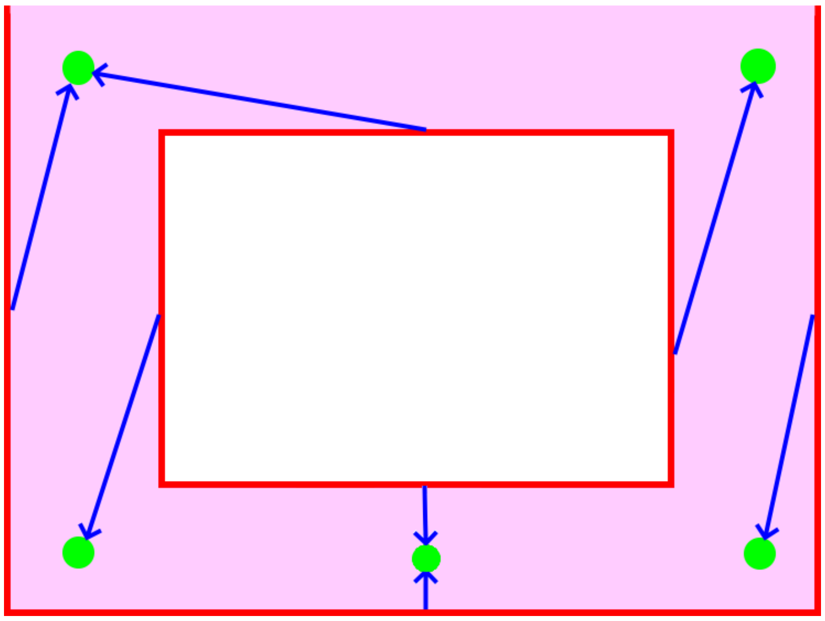
Figure 10: Using more points to check for delimiting triangles. All triangles now cast un-obstructed rays to at least one of the points.

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 the **Error! Reference source not found.** section of the introduction. Performance is therefore not a primary consideration.

3.7.2. Choosing query points

Choosing a set of 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 position and space to be as useful as possible.

The approach used is derived from the concept of *Floodfilling*. 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.

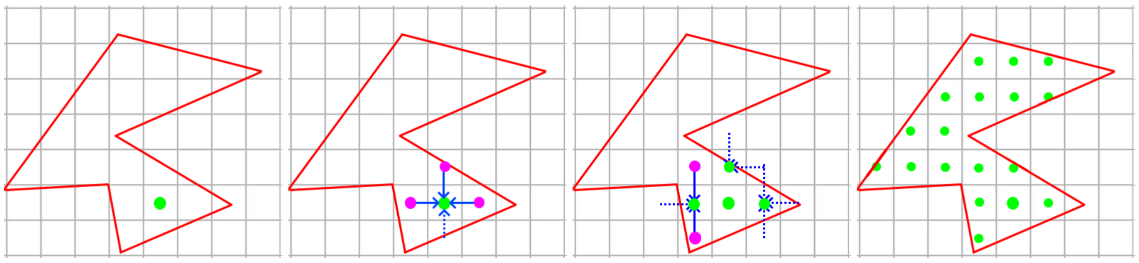
Figure 11: Four slices of the Floodfilling process. Green points indicate reachable cells of past iterations, purple ones indicate the reachable cells of this iteration. The final picture shows the output of the algorithm. All these points are query points.

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.

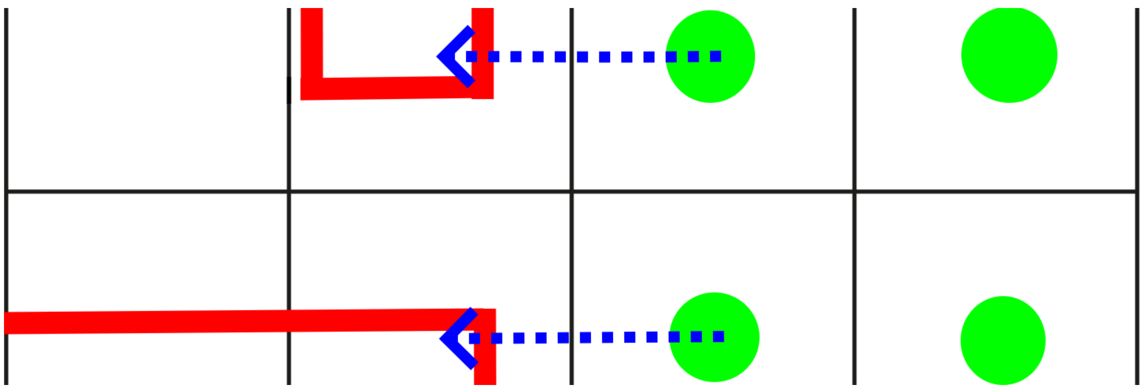
Figure 12: Issues with large grid sizes in the Floodfilling Algorithm

3.7.3. 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.

3.8. Querying the World

After the first stage of building up the volumes for each anchor, it must also be possible to query the semantic meaning of any point in the world by determining in which volume this point resides. This assumes that all volumes have been properly calculated, meaning 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 [6], but has been adapted to three dimensions for this application.

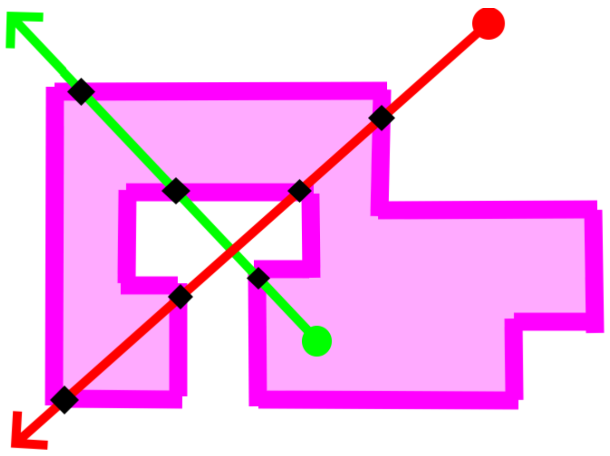

Figure 13: Visualization of the Even-Odd rule. The green ray intersects an odd number of times, so the point is inside the polygon, contrary to the red point.

3.8.1. 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.

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.

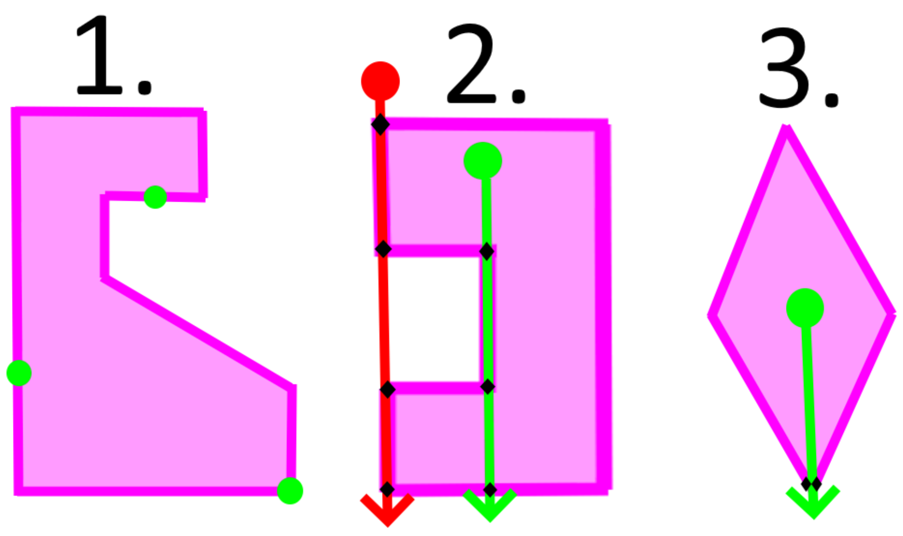

Figure 14: Visual Examples of the three described issues with the naive Even-Odd rule implementation.

3.8.2. 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), 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.

4.2. Dealing with Floating-point Precision

One of the primary technical challenges of implementing the proposed solution is dealing with numerical imprecision. Typical consumer hardware only supports floating-point values, fixed-point calculations must be emulated in software which is much slower than having hardware support. Floating-point is also better at handling large ranges of values, since it does not overflow as quickly as fixed-point, and it gives higher precision in the range of around one to ten.

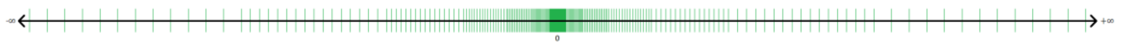

Figure 15: Precision of floating-point values at given intervals. [10]

Floating-point has become the de facto standard in (game-related) software to the point of unpracticality of deviating from that standard. 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). 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 additional precision.

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.

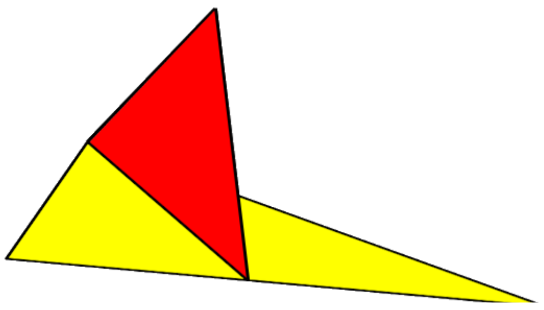
Figure 16: Two triangles touching each other, leading to potential Floating-point precision issues.

The default solution for this is to always factor in a small delta value (often called *Epsilon*) which acts as a 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.

This can however also lead to false positives, where an intersection is found which might not be there at all. Choosing a good Epsilon value is therefore usually done empirically as it also depends on the input data.

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 and the actual developed 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 ID of the anchor to which the point has been assigned, or a special value indicating that the point does not lie within any volume.

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 [10]. 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, then the triangle T is split into two different areas by the triangle C, therefore requiring tessellation.

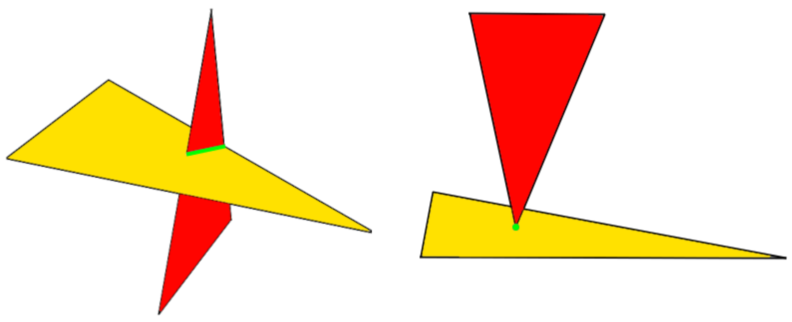

Figure 17: Two triangles on the left intersect along an edge, therefore requiring tessellation. The two triangles on the right only intersect in a single point.

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 deduplication 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.

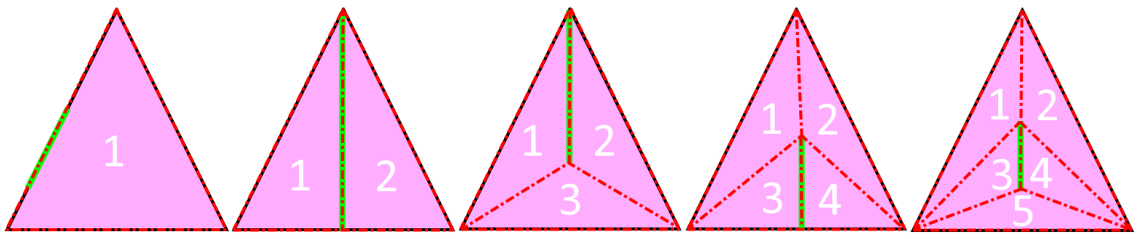

Figure 18: A triangle might be split up into up to 5 sub triangles. The intersection edge is indicated in green. The sub triangles are indicated by the dotted red lines.

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 one unique solution for tessellating triangles in this scenario. The implementation in this thesis follows a simple approach. This approach is not optimal, as it may generate more triangles than are necessary, or generate triangles that are extremely skewed and may lead to numerical precision issues.

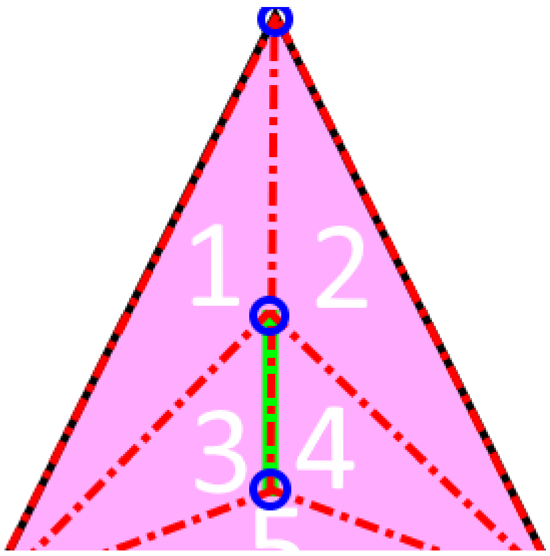

Figure 19: Showcase of the general tessellation algorithm.

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.

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 build-up phase of the solution consists of two major tasks: Clipping all delimiters in the world (see chapter 3.6) and setting up the volumes using the clipped delimiters (see chapter 3.7).

Clipping delimiters is a task that does not offer good scalability. As a reminder, it first needs to find intersections between all pairs of delimiters in the world. Then it sorts the intersections and solves them sequentially. While finding intersection 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 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 (usually 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 3.7.1).

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.

5. Assessment

5.1. Fulfillment of the requirements

5.2. Example Scenarios

5.3. Performance

5.4. Comparison with other approaches

6. Future Work

Many smaller details already listed inside thesis

Serialization

Mesh Optimization

Fixed point

7. Conclusion

List of figures

Bibliography

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

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

[3] 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.

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

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

[6] 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].

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

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

[9] 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.

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

[11] 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.