|  |
| --- |
| 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

[Eidesstattliche Erklärung 2](#_Toc173079532)

[Abstract 3](#_Toc173079533)

[Acknowledgements 4](#_Toc173079534)

[1. Introduction 7](#_Toc173079535)

[1.1. Problem Statement 7](#_Toc173079536)

[1.2. General Requirements 8](#_Toc173079537)

[1.3. Goals 9](#_Toc173079538)

[2. Related work 10](#_Toc173079539)

[2.1. Space Foundation System 10](#_Toc173079540)

[2.2. Thesis by Kerstin Pfaffinger 10](#_Toc173079541)

[2.3. Bounding Volume Hierarchies 10](#_Toc173079542)

[3. Approach 11](#_Toc173079543)

[3.1. Overview 11](#_Toc173079544)

[3.2. Representing Anchors 11](#_Toc173079545)

[3.3. Representing Volumes 11](#_Toc173079546)

[3.4. Representing Delimiters 12](#_Toc173079547)

[3.5. Virtually extending Delimiter Planes 14](#_Toc173079548)

[3.6. Clipping Delimiters 15](#_Toc173079549)

[3.6.1. Growing Delimiters outward vs. cutting them down 16](#_Toc173079550)

[3.6.2. Solving an intersection 16](#_Toc173079551)

[3.6.3. Heuristic for ordering Delimiter intersections 17](#_Toc173079552)

[3.7. Calculating Volumes 19](#_Toc173079553)

[3.7.1. Assembling the triangles 20](#_Toc173079554)

[3.7.2. Choosing query points 21](#_Toc173079555)

[3.8. Querying the World 22](#_Toc173079556)

[4. Implementation 23](#_Toc173079557)

[4.1. Overview 23](#_Toc173079558)

[4.2. The interface 23](#_Toc173079559)

[4.3. The underlying data structure 24](#_Toc173079560)

[4.4. Tessellation 25](#_Toc173079561)

[4.5. Flood filling 25](#_Toc173079562)

[4.6. Assembling 25](#_Toc173079563)

[5. Assessment 26](#_Toc173079564)

[5.1. Fulfillment of the requirements 26](#_Toc173079565)

[5.2. Comparison with other approaches 26](#_Toc173079566)

[6. Future Work 27](#_Toc173079567)

[7. Conclusion 28](#_Toc173079568)

[List of figures 29](#_Toc173079569)

[Bibliography 30](#_Toc173079570)

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. 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. General 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. The algorithm must be predictable for a human user to anticipate the results to reduce the friction between the human user and the algorithm. While this is a rather subjective, mathematically non-provable requirement, it is nevertheless an important one, as unpredictability will render the algorithm almost useless in practice. In short, changes to the input should only make a predictable change in the output.
3. 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).
4. 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.
5. 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.
6. The volume of any anchor must always be completely enclosed and must always contain its anchor.
   1. Goals

The algorithm was designed with the goal to keep the error of the output as low as possible, while simultaneously allowing for a simple integration of the solution into existing tools used in game development.

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.

1. Related work
   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.

* 1. 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.

* 1. 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.

@Incomplete: Insert a scientific source here.

1. Approach
   1. Overview

This chapter will go through all the decisions that went into the design of this algorithm, as well as giving an overview of the different internal steps that need to happen inside the algorithm for it to produce usable results.

* 1. 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.

* 1. 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, in computer graphics often referred to as bounding volumes.

1. A simple approach might be to approximate the volume 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 Figure 1).
2. One might then consider using a 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 (referenced in Related Thesis) is to discretize the continuous the world on a three-dimensional grid. A volume is then represented as a set of cells 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. 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 proposed representations of volumes, from left to right.

* 1. Representing Delimiters

Like volumes and anchors, delimiters need a representation inside the algorithm. The concept of a delimiter is split into two parts in this algorithm. The first part, called the delimiter object, is described by a cuboid transformed in the world, using position, size and rotation vectors. 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.

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.

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 [3] 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.



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.

* 1. 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.

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.

The game developer may manually set up the delimiter planes to always do exactly that 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.

* 1. 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.

* + 1. Growing Delimiters outward vs. cutting them down

This approach 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. It would be impractical to use such a small step size that the plane could be left as it was before the step. Instead, 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 discretized 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 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.

* + 1. Solving an intersection

After the initial setup based on the supplied input, the first step is solving intersections between delimiter planes. This ensures that extended planes only extend as far as they should (by the definition given in Virtually extending Delimiter Planes), and that no triangles in the internal representation of a delimiter plane intersects with any other triangles of any other plane.

This step of the algorithm goes through all delimiter planes in the world and checks for intersections with any other plane in the world. 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 also compared. This gives the developer more control via the input to specify how two delimiters should interact when they intersect. If the delimiter objects 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 clipped. This is helpful in scenarios where one delimiter seems to have higher semantic priority for the developer.



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.

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 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.

* + 1. 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 6 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. 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 6, where intersections result in “false” clipping because one of the delimiter planes should not actually extend far enough to reach the intersection point in its expected form.

The heuristic calculates the 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 7.



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.

* 1. 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 General Requirements.

As described in Representing Volumes, volumes are 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 match the triangles making up 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 encapsulate an anchor and assemble the volume by copying triangles from these surrounding delimiters. 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.

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.

* + 1. Assembling the triangles

The algorithm essentially looks at all delimiter planes and figures out which (if any) of the triangles making up the plane are actually delimiting any specific 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. 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 General Requirements section of the introduction. Performance is therefore not a primary consideration.

* + 1. Choosing query points

Choosing a set of points used for querying delimiting triangles is non-trivial. The approach used is derived from the concept of Floodfilling. The world is discretized into a three-dimensional grid with a specific uniform 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 actual 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. A Floodfilling algorithm is applied for this.

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 marked as reachable. All neighboring cells are then checked on whether the line 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 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.

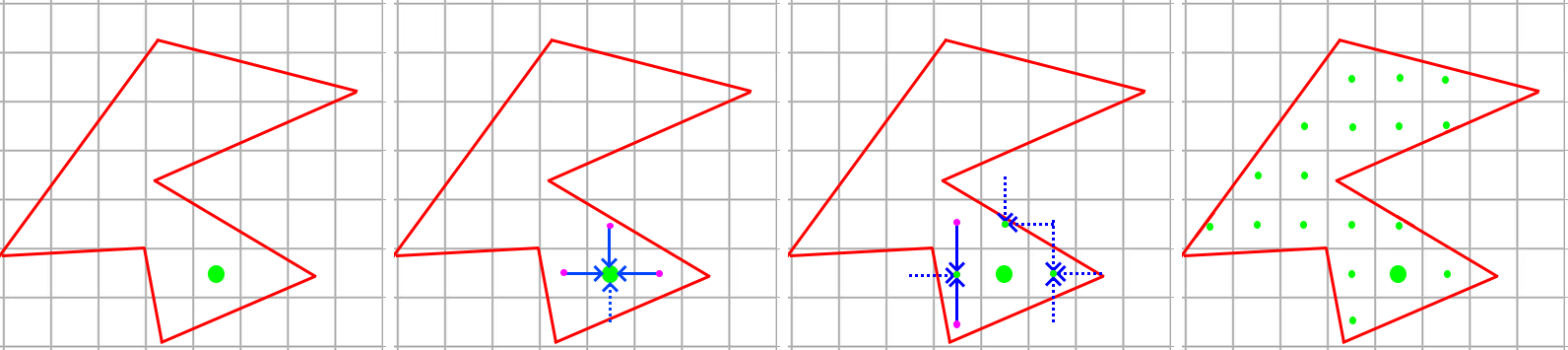


Figure 11: Four slices of the Floodfilling process. Green points indicate reachable cells of past iterations, purple ones indicate the reachable cells of this iteration. The final picture shows the output of the algorithm. All these points are query points.

@Incomplete: Add a Related Works to Floodfilling, with an actual source.

@Incomplete: Make the points in the figure bigger.

@Incomplete: Describe drawbacks of Floodfilling here.

* 1. Querying the World

1. Implementation
   1. Overview

The core algorithm proposed in this thesis consists of two different phases. The first part is building the actual data structure, representing the semantic volumes in the world. Building the structure represents the main challenge of this thesis, both in complexity and in runtime. This only needs to happen once for any given world though, as this data structure can be serialized and loaded when needed. If the world changes, this data structure needs to be recalculated. The second part is querying the created data structure to evaluate the semantic volume at any given point in the world. While being much less complex, this query might happen many times over the course of the user program, and should therefore be as lightweight as possible, especially for use in performance critical applications such as games.

The algorithm should also be independent of any specific application logic, to enable the integration in all kinds of user programs. This requires an interface between the algorithm and the user program to exchange data in a precise format. This interface should be as light and simple as possible, to minimize performance overhead and bug proneness.

* 1. The interface

This interface specifies how a user program can interact with the core algorithm. In the code, this interface is represented using a set of data structures and procedures.

This interface is split into two parts to accommodate both phases of the algorithm. This means that the user code must specify the world to the algorithm once so that it can build the underlying data structure. The interface provides a way to add anchors and delimiters to the world. The interface must also provide a way for the user code to create queries on the data structure, to get information about the semantics for any given point.

Since the algorithm is independent of user code (and user logic), it must associate semantic meaning to volumes in a very general way. Integral IDs have been chosen for this purpose, as they are fast, stable and easy to implement.

Positions and sizes for both anchors and delimiters are represented using three floating point numbers each, again for better compatibility.

The user program may create an empty world and start populating it with anchors and delimiters. Once all of them have been added, the user program can request the underlying data structure to be built.

From the interface’s perspective, an anchor is just a position and a semantic ID. The algorithm will return this semantic ID whenever a point is inside the semantic volume attached to an anchor.

Delimiters are cuboids with a position, scale and rotation. The interface will return a unique ID (i.e. the index) for each delimiter, so that the user code can then add delimiter planes to any created delimiter. A delimiter plane is created for a specific delimiter by specifying the axis on which to orient the plane, and whether to virtually extend this plane. Delimiters also have a hierarchical level assigned by the user program during delimiter creation.

After the world has been set up by the user program, and the underlying data structure calculated by the algorithm, the user program can then query for a point in the world. This essentially just returns a semantic ID for the position passed in (or some pre-defined “invalid” id, should the point not lie in any anchors’ volume).

* 1. The underlying data structure

The algorithm itself requires some additional data to be kept for computation. This data is not exposed to the user program.

The world owns a list of anchors and delimiters, as well as root delimiting planes. Each anchor stores its position, its semantic ID and a list of triangles which make up the anchor’s semantic volume. The former two attributes are passed in by the user program, whereas the latter is calculated during the first phase of the algorithm.

A delimiter stores the position, orientation and scale (commonly referred to as the transformation) of the cuboid it represents, as well as the set of clipping planes attached to it. The delimiter also keeps track of the hierarchy level it has been assigned by the user program.

Clipping planes are essentially a list of triangles. Note that a clipping plane is not actually an infinite plane in the mathematical sense, but instead a collection of adjacent triangles. This enables the clipping planes to be cut into any specific shape, which is required during the first phase of the algorithm. This applies to both the clipping planes of delimiters, as well as the root clipping planes of the world.

An implementation may also add more internal data structures for optimization purposes, such as an octree for space partitioning. These are however not required for the base implementation.

* 1. Tessellation
  2. Flood filling
  3. Assembling

1. Assessment
   1. Fulfillment of the requirements
   2. Comparison with other approaches
2. Future Work
3. Conclusion

List of figures

[Figure 1: The proposed representations of volumes, from left to right. 12](#_Toc173079502)

[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. 14](#_Toc173079503)

[Figure 3: Visual Example of virtually extending Delimiter Planes. Delimiters are displayed from a top-down view. 15](#_Toc173079504)

[Figure 4: Virtual Extension can be specific for each orthogonal axis of a plane. 15](#_Toc173079505)

[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. 17](#_Toc173079506)

[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. 17](#_Toc173079507)

[Figure 7: An example of unexpected results if the intersections are not ordered properly. Delimiters are seen from a top-down view. 18](#_Toc173079508)

[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. 18](#_Toc173079509)

[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. 20](#_Toc173079510)

[Figure 10: Using more points to check for delimiting triangles. All triangles now cast un-obstructed rays to at least one of the points. 21](#_Toc173079511)

[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. 22](#_Toc173079512)

Bibliography

|  |  |
| --- | --- |
| [1] | D. Dyrda and C. Belloni, "Space Foundation System: An Approach to Spatial," p. 8, 2024. |
| [2] | K. Pfaffinger, "Anchors and Boundaries: Developing a Game Engine System for Hierarchical Spatial Partitioning of Gamespaces," p. 69, 2024. |
| [3] | Unity Technologies, "Unity - Scripting API: Transform," Unity Technologies, [Online]. Available: https://docs.unity3d.com/ScriptReference/Transform.html. [Accessed 19 07 2024]. |