|  |
| --- |
| 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](#_Toc171870460)

[Abstract 3](#_Toc171870461)

[Acknowledgements 4](#_Toc171870462)

[1. Introduction 6](#_Toc171870463)

[1.1. Problem Statement 6](#_Toc171870464)

[1.2. Requirements 7](#_Toc171870465)

[2. Related work 9](#_Toc171870466)

[2.1. Space Foundation System 9](#_Toc171870467)

[2.2. Related Thesis 9](#_Toc171870468)

[2.3. Bounding Volume Hierarchies 9](#_Toc171870469)

[3. Theory 10](#_Toc171870470)

[3.1. Overview 10](#_Toc171870471)

[3.2. Representing Volumes 10](#_Toc171870472)

[3.3. Representing Delimiters 11](#_Toc171870473)

[3.4. Clipping Delimiters 11](#_Toc171870474)

[3.5. Calculating Volumes 11](#_Toc171870475)

[3.6. Querying Semantic Meaning 11](#_Toc171870476)

[4. Implementation 12](#_Toc171870477)

[4.1. Overview 12](#_Toc171870478)

[4.2. The interface 12](#_Toc171870479)

[4.3. The underlying data structure 13](#_Toc171870480)

[4.4. Tessellation 14](#_Toc171870481)

[4.5. Flood filling 14](#_Toc171870482)

[4.6. Assembling 14](#_Toc171870483)

[5. Assessment 15](#_Toc171870484)

[6. Future Work 16](#_Toc171870485)

[7. Conclusion 17](#_Toc171870486)

1. Introduction

All kinds of 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 the actual “physical” location of a piece doesn’t matter, but instead only the “semantic” location does (with the “semantic” location in chess being the row / column tuple). Continuous spaces are often found in sports, but they are also very common in digital games.

However, rules in games are often based on a semantic space, which requires a mapping from this continuous space into the semantic one. In soccer for example, it is important whether the ball has “fully crossed the goal line”, or whether a player is “inside the box” when committing a foul. These rules are independent of the physical layout of any particular soccer pitch, however evaluating them does require knowledge of the continuous space the game is taking place in (where exactly is the ball in this specific situation, what volume does it have, where is the goal line?). In a video game, it might be desirable to know which room the player has just entered, and whether any other person is in that same room. Similarly, the background music that is playing might depend on the district of the city in which the player currently is.

This mapping from continuous into semantic space is a large amount of effort, especially in video games containing large spaces with many (levels of) subspaces. This is usually dealt with using a lot of manual work by video game developers to assign semantic meaning to the different physical volumes. These physical 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 and semantic representation, in turn potentially leading to a worsened player experience.

This thesis will attempt to lighten the workload on game developers, while simultaneously improving the quality of the mapping from continuous into semantic space, by proposing, implementing and evaluating an algorithm to create this mapping based on input of the designers.

* 1. Problem Statement

The goal of this thesis is to implement an algorithm to generate a mapping from continuous to semantic space based on input from the designer, as well as query that mapping to return the semantics for any given point in the continuous space.

This mapping happens for a world. 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 objects in a world are the input of the algorithm. Objects can be anchors or delimiters. This input may be created manually by a game designer, or it may be (at least partially) automatically derived from the game world’s geometry and passed on to the algorithm.

Anchors are points in space which define a semantic volume, meaning the semantic volume belonging to this anchor will grow outward from this point until its growth is stopped by delimiters. Delimiters are objects in continuous space which act as a border between semantic volumes. As a real-world example, fences or walls would often be considered delimiters.

The algorithm should therefore deterministically subdivide the input space and calculate bounding volumes for every anchor, so that no volume intersects any delimiter, all volumes are extended as far as possible, and every volume contains its anchor point. The algorithm can then query all the volumes on whether they contain a specific point, to check in which semantic volume this point resides. This means that the algorithm can be split into two phases: The build-up of the data structure (the subdivision of the world space), which needs to be done once, and the querying of the data structure for evaluation.

* 1. Requirements

The problem statement implies a list of requirements which any algorithm attempting to solve the problem must fulfill, which are formalized here.

1. The algorithm must be deterministic. This means that for the same input, the same output must always be produced. If this wasn’t the case, 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. This means that a single query should be computed in a small number of microseconds on modern hardware. 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. A volume calculated for an anchor must never intersect any delimiter present in the world. Conversely, a delimiter must always stop an anchor from growing in this direction.
5. An anchor must extend as far as possible while not violating the previous requirement. This means that anchors must also perfectly represent complex shapes, by bending around corners or filling in tight polygonal shapes.
6. Related work
   1. Space Foundation System
   2. Related Thesis
   3. Bounding Volume Hierarchies
7. Theory
   1. Overview

This chapter will explain the decisions that went into the design of the algorithm, in order to fulfill the problem statement and requirements described in the first chapter.

* 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. The first challenge is therefore choosing how to represent these arbitrary n-dimensional subspaces, or in the case of three dimensions, 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, arbitrary volumes (in rendering often referred to as 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, delimiters need a representation inside the algorithm. As part of the three-dimensional world, they must also be three-dimensional shapes. It may seem plausible to choose cuboids for this, as most delimiters in the real world seem to be cuboids (fences, walls, et cetera).

The defining property of a delimiter is the splitting of a space into two parts (which may be referred to as “left” and “right” of the delimiter, “inside” and “outside”, “front” and “back”). The actual shape of the delimiter is therefore not the most important characteristic, instead the mapping of which side a point belongs to is. This mapping can be perfectly represented by a three-dimensional plane. A point can be on either side on exactly on the plane, which fits the idea of a delimiter. Therefore, planes were chosen for representing delimiters (from now on referred to as 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 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 on heavily by the implementation of this algorithm.

* 1. Clipping Delimiters
  2. Calculating Volumes
  3. Querying Semantic Meaning

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
2. Future Work
3. Conclusion

Table of figures

[Figure 1: The proposed representations of volumes, from left to right 9](#_Toc170059865)