|  |
| --- |
| 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

I would like to thank my keyboard for always being there for me.

Contents

[Eidesstattliche Erklärung 2](#_Toc170059886)

[Abstract 3](#_Toc170059887)

[Acknowledgements 4](#_Toc170059888)

[1. Introduction 6](#_Toc170059889)

[1.1. Problem Statement 6](#_Toc170059890)

[2. The Algorithm 8](#_Toc170059891)

[2.1. Overview 8](#_Toc170059892)

[2.2. Representing Volumes 8](#_Toc170059893)

[3. Implementation 10](#_Toc170059894)

[3.1. Overview 10](#_Toc170059895)

[3.2. The interface 10](#_Toc170059896)

[3.3. The underlying data structure 11](#_Toc170059897)

[Table of figures 13](#_Toc170059898)

1. Introduction

Games in general usually 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, the are often rules in games based on a continuous space which require a mapping from this continuous space into semantic volumes. 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 in principle independent of the physical layout of the soccer pitch, however evaluating them does require knowledge of the continuous space (where exactly is the ball, what volume does it have, where is the goal line?).

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 game developers to assign semantic meaning to the different physical volumes. These physical volumes may however be rather complex (for example attempting to map a house structure into one volume), 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 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 consists of anchors and delimiters. Anchors are points in space which define a semantic volume, meaning this semantic volume will grow outward from the anchor, until it hits a delimiter. Delimiters are planes in continuous space which act as a border between semantic volumes.

The algorithm should therefore deterministically subdivide the input space and calculate bounding volumes for every anchor, so that no volume is cut through by any delimiter 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.

The user should therefore simply give a list of points as anchors, as well as a list of planes which will act as delimiters. The algorithm will then calculate and return a data structure which contains the volumes of each anchor, which can then be queried.

The algorithm needs to be deterministic so that developers can rely on the algorithm producing the same results at different times on different machines (potentially even user machines) given the same input. It should also be relatively fast, so that the user of this algorithm can adapt the input if the output does not match the desired result.

The algorithm should also be very transparent in its calculations, so that the human user can anticipate the results. If the algorithm produces unexpected results, it causes more work for the human user to adjust the input until the expected result is generated. Depending on the amount of friction this caused, it might be more efficient to just do the work manually.

1. The Algorithm
   1. Overview

This chapter will explain the idea behind the algorithm. Further details can be found in the implementation chapter.

The problem statement requires the algorithm to have two phases: the build-up of the internal data structure, and the querying of said data structure. While it is possible to merge these two phases into a single, by implicitly building the data structure for each new query, it is recommended to only do the heavy build-up phase just once, since it might take a lot of time, and the results will not change for a static world setup. This initial calculation may even be done offline, meaning once by developer, and the data structure is then serialized for the shipped software.

* 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-dimension space. The first challenge is therefore choosing how to represent these arbitrary three-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 cuboid or a sphere (often referred to as a bounding box). This might be faster 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.
2. One might then consider using multiple “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. Since 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.
3. An approach used in another thesis is to base 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 essentially discretizing the space into fixed intervals. Both error and computational complexity are proportional to the chosen interval.
4. In computer graphics, arbitrary volumes are usually represented using a list of triangles. Triangles are the most primitive 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 less 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. 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.

A world owns a list of anchors and delimiters, as well as the root clipping 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.

Table of figures

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