|  |
| --- |
| 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](#_Toc169967806)

[Abstract 3](#_Toc169967807)

[Acknowledgements 4](#_Toc169967808)

[1. Introduction 6](#_Toc169967809)

[1.1. Problem Statement 6](#_Toc169967810)

[2. Implementation 8](#_Toc169967811)

[2.1. Overview 8](#_Toc169967812)

[2.2. The interface 8](#_Toc169967813)

[2.3. The underlying data structure 9](#_Toc169967814)

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.

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 user can anticipate the results, leading to less friction and less work.

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 assign a unique ID for each delimiter, so that the user code can then add delimiter planes to a 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.

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