School of Computation, Information and Technology
Technische Universität München

Bachelor's Thesis in Informatics: Games Engineering

Anchors and Boundaries: Developing a Game Engine System for Hierarchical Spatial Partitioning of Gamespaces

Kerstin Andrea Pfaffinger

School of Computation, Information and Technology
Technische Universität München

Bachelor's Thesis in Informatics: Games Engineering

Anchors and Boundaries: Developing a Game Engine System for Hierarchical Spatial Partitioning of Gamespaces

Anker und Begrenzungen: Entwicklung eines Game-Engine-Systems für die hierarchische räumliche Partitionierung von Gamespaces

Author:	Kerstin Andrea Pfaffinger
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisors:	Daniel Dyrda, M.Sc.
Submission Date:	March 15, 2024

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Bachelor Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this Bachelor's thesis is my own work and I have documented all sources and material used.

Munich, March 15, 2024

KERSTIN ANDREA PFAFFINGER

Abstract

In recent research, there are several approaches for formalizing games and gamespaces. One stream aims at providing knowledge about the current location and its related meaning within the game to its entities.
This paper proposes a solution to said challenge of creating a spatial understanding of an object's semantic location in the gamespace. It presents an editor tool that allows level designers to place so-called *anchor* points in the gamespace. These then grow a volume as far as possible using the Voronoi algorithm as a baseline. Additionally, *boundary* elements can be inserted to influence the partition. Eventually, the gamespace is fully partitioned into anchor subspaces. The system also supports nesting anchors so that they only extend within another anchor's volume. The outcome of the partition is assessed by defining custom test setups for all required features and comparing them to the expected results. That revealed that the system produces the desired space partition, but it can be improved in both performance and cleanliness in future extensions.
The tool assists the user in partitioning the gamespace automatically following rules set by placing anchor and boundary elements. The outcomes can be used for several purposes and specifically tailored towards the user's needs. The resulting volumes are stored in the scene, so they can be generated once and reused from that point on. Furthermore, they will not conflict with other game elements, as they are hidden in the game and can only collide with a custom object provided by the tool. This way, they are clearly separated from the game logic.

Keywords: *spatial partitioning*, *hierarchical spatial partitioning*, *voronoi*, *anchors and boundaries*, *game development*, *games engineering*

Acknowledgements

I would like to thank Michael Grupp for the LaTeX template used for the thesis.
A big thanks to Justin Hawkins, too, for developing a C# Unity package for Marching Cubes, which yielded stable and high-quality results.
Additionally, thanks to my boyfriend Maxi for supporting me on this journey and keeping an eye on me during the past four months.
Thanks to Jakob Rott and the entire team holding the Software Quality seminar for teaching me how to research, structure, and conduct a thesis. Their course equipped me with both a very helpful toolkit and some confidence to tackle this first major thesis more professionally.
Furthermore, I say thanks to all the other students participating in the thesis co-working sessions and helping me out with feedback, ideas, experiences, and useful research hints.
I would like to thank M. Sc. Daniel Dyrda for providing valuable feedback and hints all the time and offering the possibility of these co-working sessions in the first place.
Thanks to Prof. Dr. Gudrun Klinker for enabling this thesis and registering it so quickly.
Finally, I want to thank all the people who were listening to me talking about my thesis and its progress, problems, and milestones.

Contents

1 Introduction

For a series of games, it would be beneficial to have some sort of location structure so that an object in the game can tell where it is located in the logical game space. This way, any game object like a box, a player, or an enemy not only knows its global position in the game world but additionally has access to some kind of semantic meaning related to that very position. The ultimate goal is to create a location awareness accessible to all objects. This way, spatial knowledge can be abstracted and reused for a set of entities in a common space. This knowledge can include general information about the enclosing gamespace's structure. Additionally, an entity could potentially tell which subspaces are accessible or viewable from their current position which, in turn, can define possible follow-up actions. Essentially, any game object should be able to retrieve its semantic location in the gamespace from its current subspace.
A central term related to the field of spatial partitioning is the so-called *integrity of space*. Christopher W. Totten talks about a similar idea in their cornerstone book *Architectural Approach to Level Design*, but they call it the "spirit of a place" [39, p.4]. They state that spaces can be clustered by analyzing common properties connecting them. Those characteristics can range from simple traits like color, purpose, materials, or architectural style to rather interpreted qualities like feelings or mood associated with a place. [39]
This paper takes the first step towards this higher-level goal and proposes a solution to a subproblem of the described problem. Concretely, it tackles the challenge of partitioning the gamespace spatially based on user-defined center points, called *anchors*. Regarding the integrity of space, that means that the only property connecting subspaces is their distance to said points. The algorithm additionally takes custom *boundary* elements into account which influence the outcome of the partition. Furthermore, it supports nesting subspaces and therefore realizes hierarchical partitioning as well. Refer to section 1.2 for a detailed description of the goal of this thesis.

1.1 Applications

A system for location awareness could be useful in a variety of applications. Most of them were mentioned in the previous section, namely providing knowledge of viewable and accessible spaces, but also storing data about the current location's structure and meaning in the game's context. More general applications of spatial partitioning are load balancing [12], ray tracing [20], global illumination as well as realtime rendering [22], and 3D layout reconstruction [42].

1.2 Problem Statement

The goal of this thesis is to implement a game engine system that takes a set of space centers and assigns a subspace within the gamespace to each of them. These center points

are called *anchors* in this thesis. The algorithm additionally takes so-called *boundaries* into account which stop the expansion of any subspace and therefore influence the overall division of the space. Based on the problem statement, the most important requirements were defined. The following list contains the desired characteristics that make up the goal for this thesis:

- The user places anchors in the game world to which the system assigns subspaces.
- The user can additionally mark objects as boundaries that affect the partition by hindering the anchor volumes from growing.
- The algorithm recursively traverses subvolumes; anchors can be nested.
- Boundaries of a certain layer must only influence anchors of an equal or larger layer.
- The outcome can be visualized, but must not interfere with the game logic.
- Slight variations in the anchor placement must not influence the results.

Note, that these requirements are represented by a series of test setups, which can be found in chapter 3. Their results are discussed in chapter 9.

1.3 Methodology

To tackle the challenge described in the previous section, existing approaches related to this thesis' challenge were investigated first. As there is little to no research addressing the exact problem statement, the problem was broken down into smaller parts which were then researched individually. These are especially spatial partitioning and mesh creation. Refer to chapter 2 for an in-depth description of the research results.
Before actually developing an algorithm, the desired characteristics were turned into test cases, to be able to continuously evaluate the progress and performance of the system. They are introduced in chapter 3. Additional test setups were added throughout the tool's creation. Based on the prior research, a custom approach was worked out and implemented. This was done with the help of external packages solving subproblems that were not part of the problem statement. Utilizing the aforementioned test setups, weaknesses and shortcomings of the algorithm were identified and revised. Finally, the outcomes were reviewed using said tests.

1.4 Overview

This section presents the structure of this thesis by briefly elaborating on the purposes and contents of the upcoming chapters. The thesis continues with the upcoming **Related Work** chapter. It evolves around the current state of research within this and similar fields and sums up the main takeaways and approaches others have taken to solve similar problems as this thesis. After that, the desired characteristics of the algorithm to develop are presented by showing all test setups and defining how the algorithm should handle these in the **Requirements** chapter. Then, in the **Theory** chapter, the reader is introduced to the terms and concepts that are crucial to understanding the **Implementation** chapter. Said chapter explains concrete implementation and presents the system's most important features in detail. The Implementation chapter is preceded by the **Tools** chapter, presenting the used third-party software assisting the development of the system.

As optimizing the algorithm was a huge part of developing the tool, there is a dedicated **Optimizations** chapter covering the process of finding the potential problem locations as well as the solutions for those. To review the game engine system created to solve the problem statement, the **Assessment of the Goals** chapter again goes over the all test setups presented in chapter 3 to decide whether the partition yields the expected results. Furthermore, it comments on how the outcomes could be improved and concludes the review with a short discussion. Said discussion leads to the **Future Work** chapter, which hints at actions to take in the future and possible expansions to add to the algorithm. Finally, the thesis is summed up in the **Conclusion** chapter.

2 Related Work

The idea for this thesis is based on a Bachelor's thesis written by Claudio Belloni in 2023, proposing a general approach to formalizing gamespaces [5]. They present a human-aided algorithm for partitioning gamespaces using anchors, boundaries, and other concepts. The former two terms and their function related to spatial partitioning highly influenced the approach used in this thesis. The problem statement as presented in the introduction makes up a small part of the algorithm proposed by Belloni because it focuses on partitioning the gamespace spatially using anchors and boundaries that can be placed by the user. As this area is poorly researched, this chapter breaks down the problem statement into subproblems and analyzes how existing research has tried to solve those.

2.1 Spatial Partitioning

As mentioned in the introductory section 1.1, spatial partitioning can be useful in a variety of fields. There is a plethora of options on how to produce a partitioned result in the first place, though. The following paragraphs present some of the basic spatial partitioning methods.

2.1.1 Voxel-Based Approaches

Yang et. al aim to create highly detailed and complex "[d]igital representations of urban architecture", especially "three-dimensional [...] layout reconstruction of indoor environments" [42, p. 1]. To do so, they combine a voxel-based approach with space partitioning [42]. Dorn et al. also make use of voxelizing the space, coupled with the so-called *neighbor propagation* technique. This means that each voxel tells its neighbors which cell it belongs to and assigns them to said cell if they are not already claimed by another center point. To keep memory usage low, they only store the outer ring of voxels, as they are needed to spread the center's cell assignment. [10]
The approach used in the present thesis builds on Dorn's idea. It uses both voxels and neighbor propagation, which will be explained in detail in chapter 6. Further information on voxels can be found in section 4.2.1.

2.1.2 BSP and Extensions

An old and rather slow method is *Binary Space Partition*, or *BSP*, respectively. William C. Thibault and Bruce F. Naylor define it as "recursive, hierarchical partitioning, or subdivision, of d-dimensional space" [38, p. 154]. For a d-dimensional space, the division can take up to $\mathcal{O}(n^d)$ where n denotes the number of faces included in the input. BSP iteratively splits the space along a hyperplane, which can be associated with geometry in

the scene. It often produces a tree containing the partition's result. [38]
Meneveaux et al. improved that process in 1998. They did so by projecting vertical polygons onto the plane defined by both the x and y axes and therefore creating lines on this horizontal plane. By analyzing the created line segments, they can identify neighboring polygons. These in turn define the surfaces used to split the gamespace into volumes. [22]

2.2 The Voronoi Algorithm

The *Voronoi algorithm* acts as a baseline for this thesis' implementation. In the following, both the abstract idea as well as some implementation approaches are presented.

2.2.1 General Idea

The basics behind the Voronoi algorithm are explained in the chapter *Voronoi Diagrams* of Mark de Berg et al.'s book *Computational Geometry*. The general approach is to subdivide a space into a set of *Voronoi cells* according to a given set of center points. Each cell hereby contains all the points in space that are closer to its center than to any other center. *Closer* is thereby defined as the smallest Euclidean distance as described in the equation in figure 2.1, which is adapted from De Berg as well. [9]

$$d(p,q) := \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Figure 2.1: Distance Between Two Points p and q [taken from [9]]

A central property of Voronoi diagrams is that each cell only contains one center point [34]. Center points are commonly also referred to as *sites* [1, 9, 13, 17, 34, 43] or *seeds* [15, 31, 34]. In this thesis, the term of choice is center or anchor point, respectively.

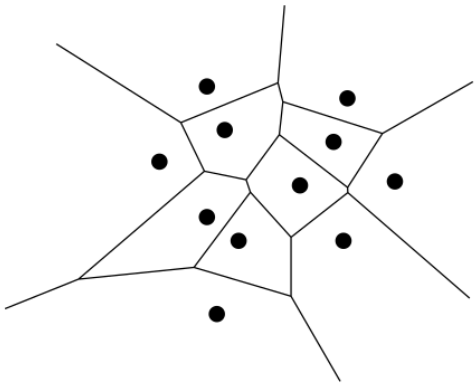


Figure 2.2: Example of a Voronoi Diagram [taken from [9, p. 149]]

An example of a Voronoi diagram can be found in figure 2.2. Refer to figure 2.3 for an exemplary application. In there, the Voronoi partition was used to subdivide the trading areas of the capitals of the twelve provinces in the Netherlands [9]. This is very similar to what this thesis wants to achieve for gamespaces. Note that the problem statement requires the partition of a 3D space. The three-dimensional cells may be referred to as *volumes* in the sequel.

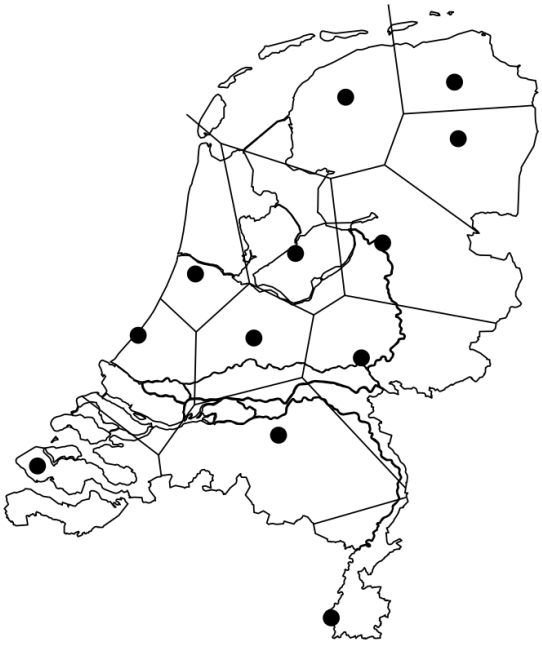


Figure 2.3: Using Voronoi Diagrams to Partition a Country [taken from [9, p. 147]]

The underlying system uses a custom voxel-based implementation utilizing breadth-first graph search to compute the Voronoi volumes. This technique will be introduced in chapter 4 and works similarly to flood fill algorithms which will be elaborated on later in section 2.2.2. However, there is a plethora of alternative approaches, which are presented in the following paragraphs.

2.2.2 Implementation Approaches

The most famous Voronoi algorithm was presented by Steven Fortune in 1986, falling into the category of sweep line algorithms [13]. These algorithms make use of a line traversing the Euclidean space and triggering data updates when hitting some kind of geometry [45]. Besides that, there are four main ideas for implementing a Voronoi algorithm which are explained in the following paragraphs.

Incremental algorithms. The first group of algorithms processes the set of center points in a given order [3]. Some of these approaches, like the one proposed by Aurenhammer et al., additionally randomize the sequence of the input set prior to traversing it [3].

Divide-and-conquer algorithms. Other approaches divide the space into small regions and perform the partition in these regions independently. Aichholzer et al. base their approach on the so-called *edge graph* which consists of points that are equally distant to multiple center points [1]. Smith et al. limit the number of computations due to simplifications they can make for patches of points belonging to the same subspace [34]. They recursively subdivide the resulting chunks until all of their corners belong to the same center point.

Flood fill algorithms. To speed up the computation, the GPU can be employed to carry out costly operations. Kenneth E. Hoff III et al. make use of the linear interpolation and Z-Buffer capabilities of the graphics hardware in their research from 1999 [17]. More approaches considering GPU acceleration arose from that point on. A lot of representatives of the flood fill algorithm family are also incorporating GPU accelerations. They start from the center point pixels and iteratively assign their neighboring pixels to the cell related to that very center [40]. Papers in that area are among others [15] and [40]. Flood fill approaches are especially useful for the present problem statement because they allow for flowing around geometry like the boundaries.

Jump flood algorithms. A special version of flood fill algorithms is the so-called *jump flood algorithm* or, in short, *JFA*. The idea originated from Rong and Tan in 2006. The difference to the regular flood algorithm is that JFAs assign the pixels with a decreasing distance l to their center's cell instead of propagating the cell to their direct neighbors only, which saves a lot of computation time. [31]
An exemplary illustration of a basic JFA process is presented in figure 2.4. Here, the propagation distance l starts at a value of 4 and is halved in each step. Further research on JFAs can be found in [21], [32], and [43].

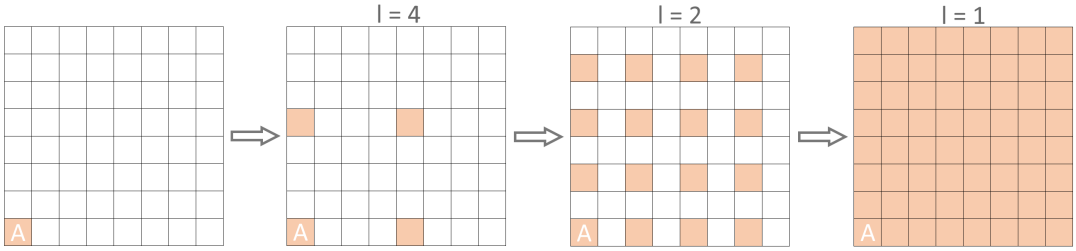


Figure 2.4: Illustration of a JFA propagation in 2D [adapted from [31, p. 111], [32, p. 2], [43, p. 77]]

Delaunay tesselations. Another widespread approach is to compute the *Delaunay tesselation* and map its outcome to a Voronoi partition. It divides the space into tetrahedra, or triangles, respectively. Hereby, a triangle vertex is a Voronoi center point. The triangles are only valid if no input center point is enclosed by the sphere defined by the triangle vertices. Delaunay tesselations are helpful because their structure is dual to the one of the Voronoi diagram and therefore they can be used for deriving the Voronoi partition. [18]

2.3 Surface Reconstruction

Surface reconstruction or *isosurface construction* aims at converting scalar data points into a digital model. It is based on the concept of an *isosurface*, which is a "set of points with identical scalar values" [7, p. 130]. Isosurface construction techniques try to identify said isosurface and create a three-dimensional model defined by it. The surface reconstruction method of choice for this thesis is an algorithm called *Marching cubes*. However, there exist alternative approaches for converting a density field to a triangle mesh, namely *Surface Nets* and *Convex Hulls*. These three ideas are presented in the following subsections.

2.3.1 Marching Cubes

Marching cubes is a surface reconstruction algorithm using three-dimensional data [19]. It can approximate surfaces from regular 3D grid point samples [28]. The idea together with the name originated from research by William E. Lorensen and Harvey E. Cline about creating three-dimensional models out of medical data [19]. In their paper named *Marching cubes: a high resolution 3D surface construction algorithm* [19], they introduce a divide-and-conquer algorithm working on 2D slices of data. The algorithm processes these lattice slices in a "scan-line order and calculates triangle vertices using linear interpolation" [19, p. 347]. For processing, a cube is positioned between two lattices, where four corners of the cube stem from one slice each as shown in figure 2.5. These cubes are processed independently [28].

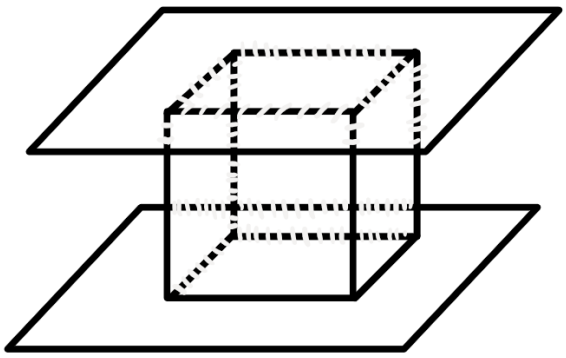


Figure 2.5: Illustration of Slices and Cube in the Marching Cubes Algorithm [adapted from [19, p. 348]]

Each of the vertices has a scalar value assigned, indicating whether it is part of the surface or not. Vertices inside or on the surface have a value of 1, and those outside the surface are 0 [19]. In their *A survey of the marching cubes algorithm*, Timothy S. Newman and Hong Yi also talk about marked and unmarked vertices, respectively [27]. This classification of vertices allows for determining the border of the surface. If an edge of the cube contains two vertices with different values, then this edge intersects the surface [19]. Newman and Yi call such an edge "active" [27, p. 856].
Depending on the configuration of marked and unmarked vertices together with intersecting edges, the resulting triangulation is derived. Triangulation refers to subdividing a polygon into triangles. The triangulation of the model is achieved by performing a lookup in a pre-created table. As a cube consists of eight vertices and each of these can either be marked as 0 or 1, there are a total of $2^8 = 256$ possible configurations. Exploiting symmetries and reflections, this number can be reduced to 14 base cases plus an extra trivial case where all vertices are 0. See figure 2.6 for an overview of these 15 cases. [19]
Upcoming adaptions of the Marching cubes approach use 15 [27] or even 22 [28, 29] base cases apart from the trivial case to counteract possible artifacts of the original algorithm. Gregory M. Nielson additionally uses quads instead of triangles for mitigating the problem of poorly shaped triangles [29]. Apart from these, Newman and Yi point out artifacts like loss of small features of the geometry data, aliasing at sharp corners, and erroneous shading [27].

In this thesis, the method of choice to produce meshes is, indeed, Marching cubes because the input data structure for the Marching package has a similar format to the output of the

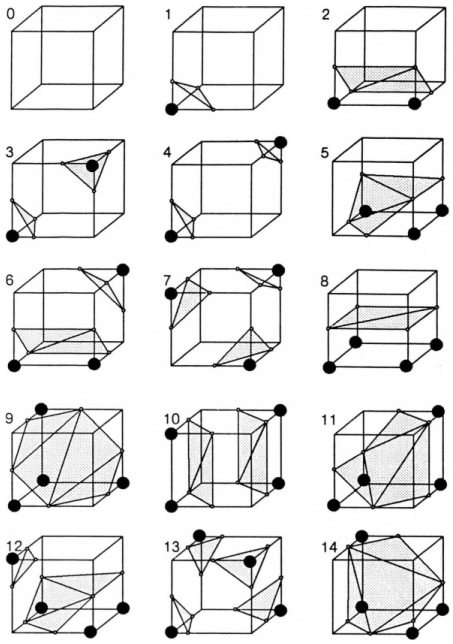


Figure 2.6: Base Triangulation Cases in the Marching Cubes Algorithm [taken from [19, p. 349]]

first step in the algorithm. Additionally, the marching cubes are very similar to the cuboid voxels used in the first step of the algorithm. The problems addressed by Newman and Yi, among others, are not problematic in the application of this thesis. Mainly, because the resulting meshes are not visible to the user of the system but are rather used for internal logic like calculating collisions.

2.3.2 Surface Nets

Sarah Gibson researched a technique called *Surface Nets* in *Constrained elastic surface nets: Generating smooth surfaces from binary segmented data* [14]. This section is heavily based on their publication. As with many surface reconstruction methods, their approach was involved in creating digital models out of medical data generated by Computer Tomography or Magnetic Resonance Imaging. To do so, it is crucial to group data points before converting the resulting data into a model. Surface nets aim to keep fine details from the original data while creating a smooth surface at the same time.
First, points lying on the surface have to be identified. This is done by approximating the geometry with cubes and assigning values to their vertices. Similar to the Marching cubes technique, a cube with both inside and outside corners is part of the isosurface. To produce smooth, highly detailed results, said surface cube centers are then connected and their positions are adjusted to reduce sudden jumps or similar artifacts in the resulting model. This relaxation is repeated to reach the desired smoothness. To retain fine details like cracks at the same time, a point can never be moved out of its enclosing cube. After the surface net has been constructed, it is then triangulated to create a surface mesh. Each center and its linked neighbors are processed sequentially. Finally, distance values from center points to surface triangles are used to correctly render the result. [14]
Later, Zefass et al. presented an extension that lets the user deform the resulting surface net to define masks and thresholds with ease [44].

2.3.3 Convex Hulls

Further, Bhaniramka et al. present an algorithm for creating surfaces in arbitrary dimensions leveraging convex hulls in *Isosurface construction in any dimension using convex hulls* [7], which works similarly to Marching cubes explained previously. Barber, Dobkin, and Huhdanpaa define a convex hull as "the smallest convex set that contains the points" [4, p. 469] of an object. There exist many algorithms for constructing the convex hull of a given set of points. Besides Barber et al., who present their so-called *quickhull algorithm* [4], Bentley and Faust investigate various approaches approximating convex hulls [6]. Additionally, Preparata and Hong propose a divide-and-conquer method for constructing them [30]. Note, that the following paragraphs are fully based on the research conducted by Bhaniramka et al. [7].
Their algorithm starts with dividing the d-dimensional space by a regular grid. This means that the space is now separated into hypercubes of the same size. Then, they determine where the isosurface intersects the grid edges. Similar to the previous approaches, intersections are detected if the two endpoints of a grid edge have certain scalar values, i.e. one of them lies inside the object and the other one is located outside of it. The exact point of intersection is then calculated using linear interpolation. The next step is to construct the surface within each of the hypercubes. This is done by retrieving the points enclosed by the cube, but outside the isosurface and intersecting their convex hull with the cube's volume. The resulting intersection in turn purely contains the surface. Hereby, the construction of the convex hull can be sped up by using a lookup table. This, however, is only applicable to dimensions $\mathcal{R}^d$ with $d \leq 4$.[7]

3 Requirements

As described in section 1.2, the user of the game engine system developed for this thesis can place both anchor and boundary objects anywhere into a gamespace and configure them to their needs. These configurations involve the layer on which anchors and boundaries operate as well as the radius to which the anchor's volume should grow if nothing is in its way. It is also possible to create a hierarchical partition using these layer properties. To define the goals more concisely, the tool was developed against a set of test setups. These act similarly to a test suite and were used to evaluate both the performance and accuracy of the algorithm. All of the test setups together with their justification and applications are presented in this section. Be aware that some of these came up during development as additional requirements became apparent and desirable.

3.1 Test Setup Categories

The setups can be divided into six major categories, starting with a single anchor without any boundaries and ranging to more complex combinations and arrangements. The categories are as follows:

- Single anchor of different sizes (section 3.2).
- Combination of anchors without boundary elements (section 3.3).
- One to two anchors with varying boundary configurations (section 3.4).
- Single anchor in combination with shapes made up of boundaries (section 3.5).
- Hierarchical anchors without boundaries (section 3.6).
- Hierarchical anchors with boundaries of different layers (section 3.7).

Those listed setups are further accompanied by a complex gamespace example combining all challenges in a mock application to test the system in a more realistic environment. Refer to section 3.8 for that. The setups are now described in more detail and illustrated by screenshots.

3.2 Single Anchor of Different Sizes

For the first test, the algorithm was fed with six setups, each containing a single anchor. Those vary in size of the anchor's radius of influence which is from now on referred to as *maximum distance*. More on that can be found in section 6.1.1 in the Implementation chapter. The exact maximum distances were 0, 1, 4, 10, 40 and 100. Sizes 0 and 1 were included to test how the algorithm handles edge cases. The others should give an estimation of the algorithm's performance in terms of run time. Figure 3.1 shows the setup for all tests in their basic form. There is only one figure because the setup is visually identical for all of the tests in this category. Note, that the anchor objects have neither a

Figure 3.1: Illustration of a Single Anchor Setup

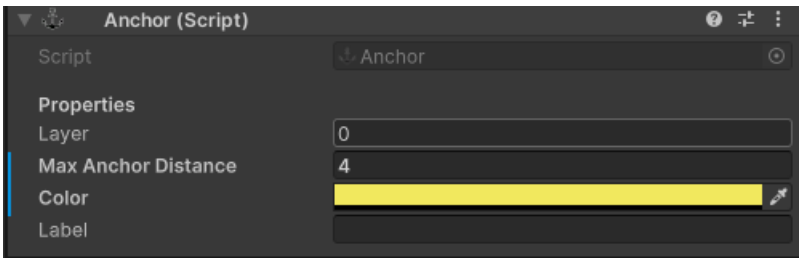
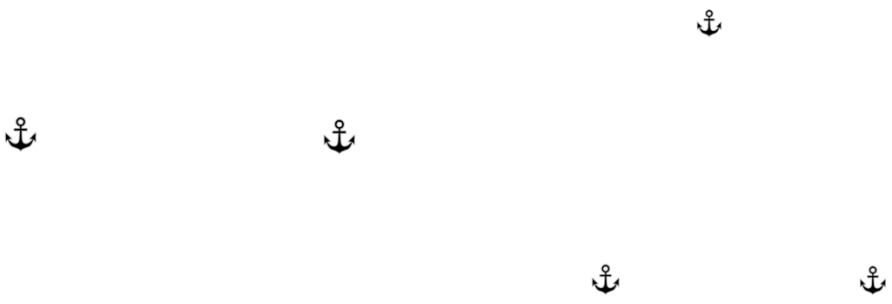


Figure 3.2: Inspector Values for the Single Anchor Setup With Maximum Distance 4

volume nor a geometry, which is why they are represented by an anchor symbol. Only the value of their distance variable is adjusted. For an example configuration, refer to figure 3.2, which shows the *Anchor* component in the anchor object's inspector. Here, the maximum distance is set to 4. The algorithm must be able to handle a single anchor within a reasonable time frame.

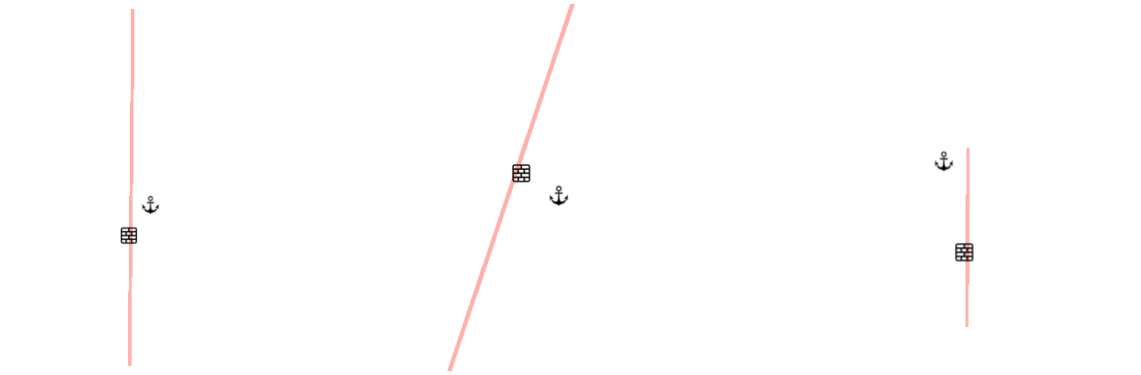
3.3 Combination of Anchors without Boundaries

Now it is time to state how the anchor volumes should interact with each other. To test this, two exemplary cases were set up. The first one contains two anchors whereas the second one is comprised of three anchors. The reader can have a look at both of them in figure 3.3a and 3.3b. As one imagines, the spaces related to the anchors must grow uniformly but stop in certain directions when hitting another anchor's volume.



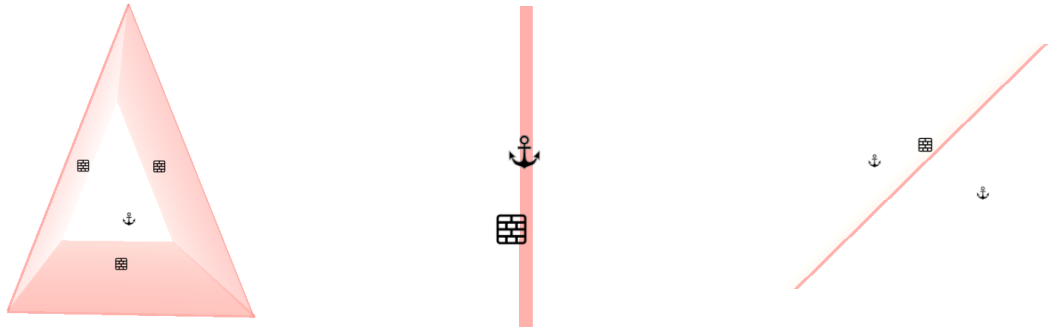
(a) Two Anchor Setup (b) Three Anchor Setup

Figure 3.3: Combination of Anchors



(a) Anchor Next to Vertical Boundary (b) Anchor Next to Diagonal Boundary (c) Anchor that Grows over Boundary

Figure 3.4: Simple Boundary Setups



(a) Anchor in Prism (b) Anchor in Boundary (c) Anchors with Boundary

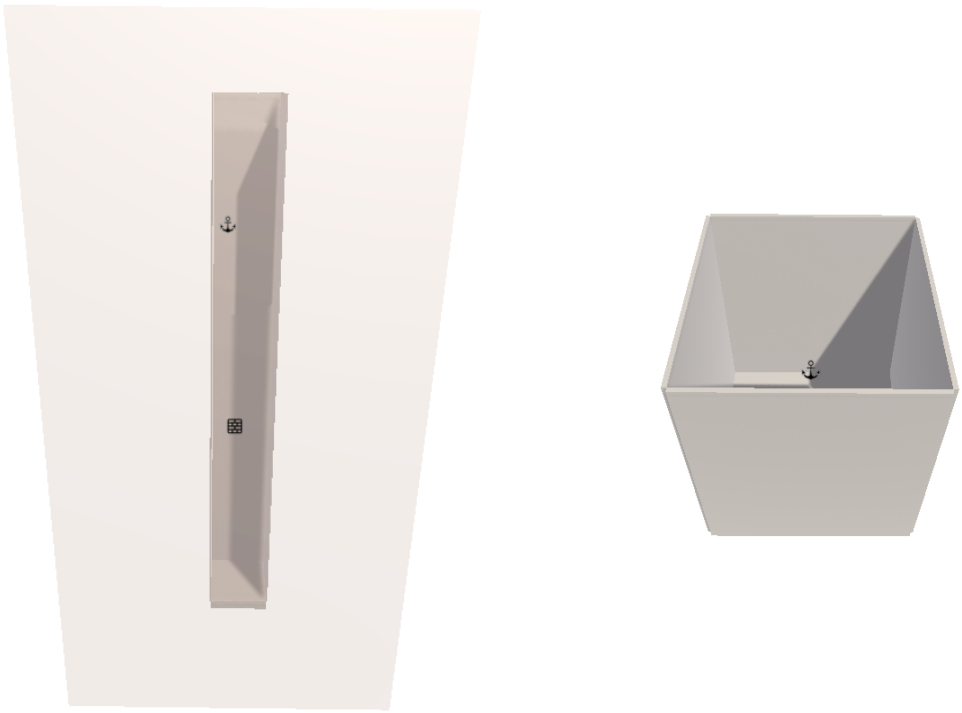
Figure 3.5: More Advanced Boundary Setups

3.4 Anchors with Different Boundary Configurations

As the title of this thesis implies, boundary elements should also affect the partition. The boundaries in this category are represented by planes. However, later categories will elevate the complexity of the boundary elements to make the tests more realistic.
The first and most simple setup here is comprised of a single anchor with a maximum distance of 7, accompanied by a vertical boundary that limits the anchor's expansion on one side. It is shown in figure 3.4a. The other setups use the same anchor properties but change either the boundary's position relative to the anchor or its orientation. Note, that all boundary elements are marked with a wall icon. Figure 3.4b depicts a slanted boundary wall, while figure 3.4c visualizes a boundary that allows the volume to grow over it. These three tests are accompanied by more advanced setups. Figure 3.5a shows one of them. Here, an anchor is boxed in by three boundaries forming a prism shape. A later added edge case scenario places the anchor directly at a boundary, as seen in figure 3.5b. Lastly, figure 3.5c adds a second anchor to the boundary setup.
The interaction between anchor volumes and boundary elements should be as follows: A boundary should stop the anchor's expansion on that side while allowing the volume to flow around it if possible. In the special case where the anchor starts at a boundary, there are a few possible solutions. The anchor can either produce a volume of size 1 or no volume at all. How this is handled in this thesis' algorithm will be discussed in chapter

6.

3.5 Single Anchors in Combination with Boundary Shapes



(a) Anchor Starting in Long Bucket (b) Anchor Growing out of Bucket

Figure 3.6: Bucket Setups

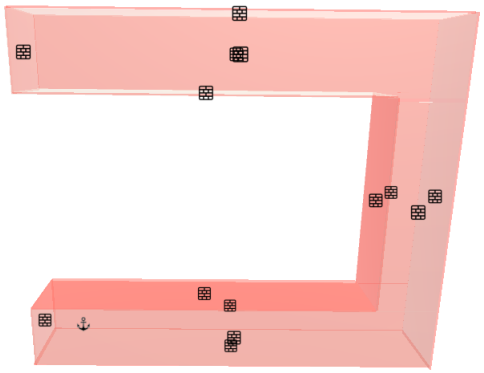


Figure 3.7: Anchor in Tube

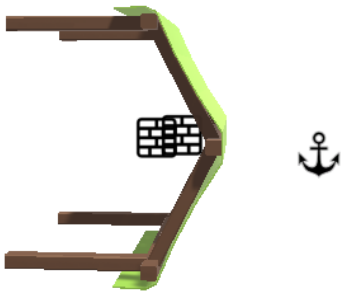


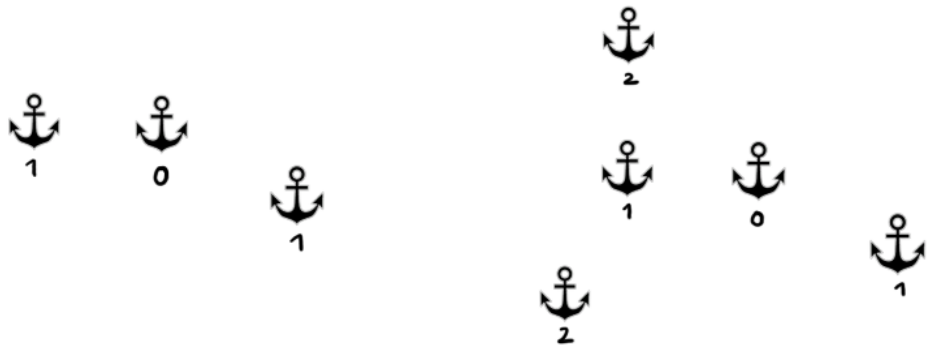
Figure 3.8: Complex Boundary Element

Boundaries are not always as simple as walls. There are more complex cases to take into account which are tested in this category. The term *shape* hereby refers to any boundary object that is more complex than the walls or planes present in section 3.4. Some of the shapes here are made up of several planes while others use actual geometry.
The first two setups use a shape that is referred to as *bucket* in this thesis. It is essentially

a cuboid with a missing face. Examples are shown in figure 3.6a and 3.6b, presenting the test setups. They, again, use a single anchor that is now placed inside a bucket. To increase the readability of the first setup, the front face was replaced by a transparent boundary so that the inside became visible. Another interesting shape is the *tube*. As shown in figure 3.7, it has several corners and winds instead of defining a straight path. The final setup here then places some objects with complex geometry next to the anchor. Figure 3.8 depicts a setup that uses a tent object marked as boundary.
The goal of this category is to assess the phenomenon of flowing around some kind of border in more difficult situations. The anchor should adhere to the bucket borders where present and flow freely outside them. Its volume should also correctly approximate the complex geometry that blocks its way. Furthermore, it should follow the path defined by the tube until it hits its maximum distance.

3.6 Hierarchical Anchor Setups without Boundaries

Figure 3.9: Inspector Values of Layer-1 Anchor



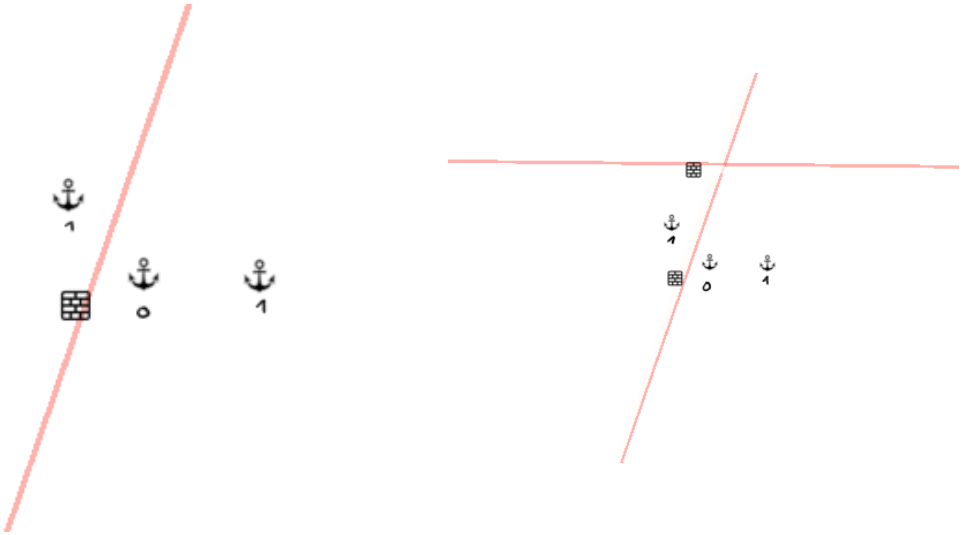
(a) Two Layers (b) Three Layers

Figure 3.10: Hierarchical Anchor Setups

All the previous setups assume that all anchors and boundary elements operate on the same hierarchical layer. Yet, the problem statement postulates that both anchors and boundaries can be nested and configured to only influence certain layers within the partition. This is put to the test with the tests in the next category. In theory, the user can infinitely nest anchors. To keep the results readable, the presented setups have a maximum of three layers and inner layers have at least two anchors per higher-order layer to show any impact on the partition. These observations led to a set of test setups.

Initially, no boundary elements are involved. Figure 3.9 shows the inspector values of the centers' *Anchor* component. In this case, the anchor is part of layer 1, which implies that this anchor must be nested within another one. The outermost layer is associated with the value 0, and higher layer values refer to anchors further down in the hierarchy.
The first setup, depicted in figure 3.10a, includes a layer-0 anchor and two anchors of layer 1. Note, that the layer numbers were added to the screenshot afterward and are not visible in the gamespace. The remaining scenario expands on the first one. As seen in figure 3.10b, it involves a third hierarchy layer, represented by a value of 2. The requirements here are that none of the lower-level anchors' volumes exceeds the border of the enclosing one. Furthermore, anchors of the same layer should influence each other's growth as demanded by the setups in section 3.3.

3.7 Hierarchical Anchor Setups with Boundaries



(a) A Layer-1 Boundary Influencing two Layers of Anchors

(b) Additional Layer-0 Boundary Cutting Across at the Top

Figure 3.11: Hierarchical Anchor Setups With Boundaries

Last but not least, the boundaries of different layers have to be tested as well. This category includes only two setups. The first is a copy of the first setup shown in section 3.6, meaning it contains a layer-0 anchor enclosing two layer-1 anchors. The major difference now is that a layer-1 boundary plane is added. As an illustration for that, refer to figure 3.11a. The second setup additionally has a boundary wall of layer 0 cutting the top of all anchors' volumes. Figure 3.11b shows said test setup. The expected outcome here is that a boundary element configured with layer n must only influence the partition of anchors of layer n or greater. Concretely, in the first case, the layer-1 plane should only restrict the inner anchors. Furthermore, The second scenario is fulfilled if the layer-0 boundary additionally restricts all anchor volumes.

3.8 Complex Environment

The complex environment used to test the partition algorithm in a realistic scenario is a desert village. It was specifically created for this thesis using a combination of free asset packs from the Unity Asset Store. A top-down view of it can be seen in figure 3.12.

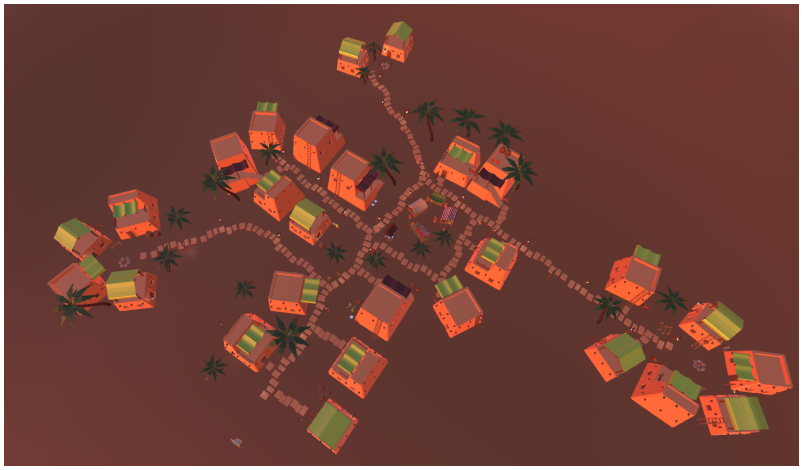


Figure 3.12: The Desert Village

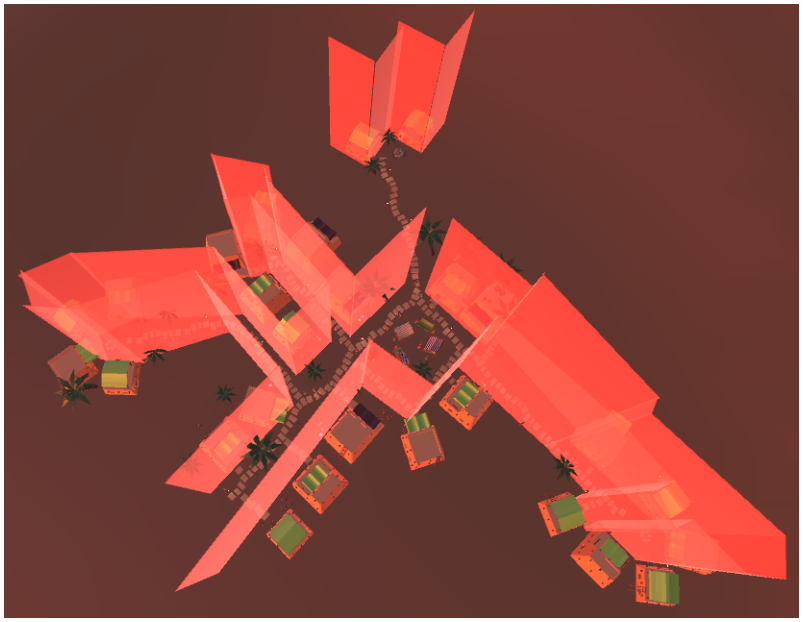


Figure 3.13: Adding Delimiters to the Desert Village

Below it, figure 3.13 shows how the village was prepared for the partition. Here, the reader can notice the presence of a bunch of boundary planes. These are utilized to customize the partition's outcome to the designer's needs. In this case, they are used to restrict the anchor's growth and therefore retain a walkable area with separate volumes. Functionally, these boundaries take the role of delimiters. These will be introduced in section 4.1.4 of the Theory chapter. Essentially, they are helper objects that block the expansion of the anchors but are not visible in the final game.
Groups of village houses were assigned to a layer-0 anchor. Additionally, each house received an individual anchor of layer 1, as well as some important points like the market stands or the well. The expected outcome is that the group anchors only mutually interact with one another and that houses belonging to a common group influence each other but do not expand out of their enclosing volume. The delimiters must block the house anchors from interfering with the walkways.

4 Theory

Before explaining the approach and the algorithm's exact inner workings, it is important to introduce some important terms and concepts of both the thesis' problem and solution domain. These are split into three categories: Those that lie in the context of spatial partitioning, those that are relevant to the description of the algorithm, and those stemming from the field of searching.

4.1 Terms Related to Spatial Partitioning

As spatial partitioning is the central field of this thesis, it is important to define the core concepts related to that area. These are mainly based on the definitions by Daniel Arribas-Bel and Martin Fleischmann in *Spatial signatures - understanding (urban) spaces through form and function* [2]. Especially the third chapter serves as a fundamental reference for the meaning and usage of these terms.

4.1.1 The Voronoi Algorithm

The idea behind the Voronoi algorithm was introduced before in chapter 2. This paragraph repeats the significant basics of the algorithm, especially those that contribute to the spatial partition performed in the system developed for this thesis. Essentially, the Voronoi algorithm can be utilized to partition any space by defining center points that expand their volume until they hit another center's volume. In the end, the space is fully divided by the centers. Hereby, any point in the input space is assigned to the center it has the shortest distance to.

4.1.2 Anchors

The Voronoi centers are referred to as *anchors* in the context of this thesis. They define the root of a volume's expansion during the partition. Note, that volumes are also called *subspaces* in the sequel.

4.1.3 Boundaries

The anchor's counterpart is the so-called *boundary element*, or *boundary* in short. Boundaries act like blockers for the expansion of the anchor volumes. Any time a subspace encounters a boundary, it will stop expanding further in that direction. Note that subspaces can flow around boundaries, meaning they only stop the growth in the space occupied by their mesh.

4.1.4 Delimiters

Delimiters mark a special kind of boundary. These are boundary elements that are only used to influence the partition but do not affect the entities in the actual game. More precisely, they block the expansion of anchor volumes but do not hinder game objects from moving through them. Delimiters are a great tool for designers to tweak the outcome of the partition to their needs by limiting the growth of subspaces in a certain direction.

4.1.5 Subspace

This term is presumably already familiar to the reader, but it will be used frequently in the upcoming pages, especially in chapter 6, so it is important that its meaning in the context of this thesis is explained explicitly. The *gamespace* describes the whole connected space a game world or parts of it are built in. A scene in Unity like the desert village from section 3.8 would be clarified as a single gamespace. A *subspace* now is any part of the gamespace assigned to an anchor during the partition phase. Therefore, they may also be referred to as *anchor volumes* in this paper. Subspaces can again be divided into even smaller subspaces to include the hierarchical aspect of the problem statement.

4.2 Terms Related to the Implementation

This section contains terms that are used in the Implementation chapter later. They help to explain the algorithm and understand all of its details.

4.2.1 Voxel

A pixel is a unit in the two-dimensional space. Likewise, a voxel describes a pixel's equivalent in the 3D space as it represents a three-dimensional unit "with pre-defined volumes, positions, and attributes, which can be used to structurally represent discrete points in a topologically explicit and information-rich manner" [41, ch. Abstract]. They are shaped like cubes.

4.2.2 Bounding Box

Each object in space occupies a portion of this very space with its geometry. There are several measures to compute and visualize this portion. Some are very exact approximations of the actual geometry and some are rather coarse, but in turn are much easier to calculate and to work with. Depending on the use case, the one or the other option is preferred. The following paragraph is based on Chang, Wang, and Kim's paper about *Efficient collision detection using a dual OBB-sphere bounding volume hierarchy* [8].
When considering three-dimensional objects, there are two types of simple shape approximations, namely *bounding boxes* and *bounding spheres*. The former takes the minimal cuboid enclosing all vertices of the geometry as an approximating shape whereas the latter uses the smallest sphere containing the shape. Depending on the geometry of the object to approximate, one or the other can represent the object more tightly.

Bounding boxes are either axially aligned bounding boxes, or *AABBs* for short. Or they are oriented bounding boxes, also referred to as *OBBs*. While OBBs usually are tighter because they can rotate to fit the shape best, AABBs have significant advantages when it comes to computing overlaps. Spheres can also determine overlaps very quickly. The algorithm used in this thesis employs both AABBs and bounding spheres because they come with efficient overlap calculations. Moreover, the voxels used for the partition are axis-aligned in any case, so OBBs would not grant any benefits in tightness.

4.3 Terms Related to Searching

Searching is a technique used in domains of artificial intelligence to find one or several solutions to a problem. Algorithms tailored to searching often utilize a so-called *search tree* they traverse [33]. This part is heavily based on the fourth edition of the cornerstone book *Artificial Intelligence, A Modern Approach* by Stuart Russel and Peter Norvig [33], especially the chapter about search algorithms. The algorithm created for this thesis does not use a search tree but carries over many concepts of searching techniques as described by Russel and Norvig. These are elaborated on in the following subsections.

4.3.1 FIFO and LIFO Queues

Before digging further into the field of searching, it is crucial to introduce the abbreviations *FIFO* and *LIFO*. These concepts are central to understanding the upcoming definitions. Both of them define the order in which items are retrieved from a set. Hereby, FIFO stands for *first-in-first-out* whereas LIFO means *last-in-first-out* [33]. This means, that in FIFO the first element to take out from a queue is the one that was added first, while in LIFO, it would be the element that was added last.

4.3.2 BFS and DFS Techniques

This section introduces two of the most basic search algorithms presented by Russel and Norvig. These two approaches are *BFS* and *DFS*, which are abbreviations for *breadth-first search* and *depth-first search*, respectively. BFS and DFS are so-called *uninformed search strategies*, as they do not have any information about the environment whatsoever. Especially, since they cannot tell how close they are to finding a solution. They essentially differ in the type of queue they use for traversing the search tree. BFS employs a FIFO queue, whereas DFS utilizes a LIFO one. This being said it becomes apparent that breadth-first search tries to find a goal by traversing the search tree layer by layer. It starts at the root node and looks at all its successors, followed by their successors, and so on. Depth-first search, on the contrary, follows one path until it arrives at a dead end. So, instead of expanding one layer at a time, it goes as deep as possible as fast as possible. [33]
The system developed for this thesis is based on BFS, so it uses a FIFO queue. That is due to the nature of the Voronoi algorithm, where anchor volumes grow one step in every direction per iteration. Using DFS would defeat the purpose of assigning a point to the closest center because one center would expand fully before the others started growing one step.

4.3.3 Tree-Based and Graph-Based Search Algorithms

When talking about search algorithms, two major approaches come into play. These can be distinguished by a central property indicating if the search keeps track of already visited paths to eliminate redundancy or not. A search algorithm that takes redundant paths into account uses *graph search*. Those, that ignore duplicates can be collectively referred to as *tree-based* search algorithms. They both keep track of the current set of tree nodes they have to visit next. As mentioned before, these can be organized in a FIFO or LIFO queue, but alternatives also exist. The queue is called *frontier queue*. In addition to said frontier, graph search algorithms store a set of nodes that were already reached. This set is appropriately called *reached set*. [33]
For this thesis' system, graph search was employed because eliminating duplicated paths is crucial for the algorithm to terminate. The concrete use of the previously mentioned data structures, frontier and reached, will be explained in detail in chapter 6.

5 Tools

This chapter gives a short overview of which tools were used to develop the game engine system for this thesis.

5.1 Unity Editor Scripting

The game engine used for developing the spatial partitioning tool is Unity. This engine offers the possibility to extend its editor by writing code that does not execute in the game but rather during the editor's runtime. That is perfect for the goal of this thesis because it allows to create a tool that can assist level designers in their workflow of building gamespaces.

5.2 External Packages

To convert the result of the partition to a mesh, the system uses an external package from GitHub. It takes care of converting a set of positions into a connected mesh using a technique called *Marching cubes*, which was introduced in chapter 2. The package was created by Justin Hawkins in 2015 [16].

5.3 Tools Used for Optimizing the System

In chapter 7, changes made to optimize both the algorithm's memory and CPU usage will be described. Tools leveraged to guide and support the identification of problematic locations in the code itself are, among others, the Rider IDE and the Unity Profiler. All of them will be presented in more detail in section 7 in the aforementioned chapter.

6 Implementation

In this chapter, the tool's inner functionality is presented. As mentioned in the introductory chapters, it uses a voxel-based partitioning method with neighbor propagation while adopting some characteristic ideas from the realm of flood fill techniques. The algorithm performs the following main steps:

1. Collect all anchors and boundaries that were placed in the gamespace by the user (sections 6.1.1 and 6.1.2).
2. Start from the anchors' positions and let them grow iteratively until they reach a boundary element or another anchor's volume. This will result in a set of voxels defining each anchor's volume border (section 6.2.3).
3. Convert each anchor's voxel set into a connected mesh (section 6.3).

Before each of the steps is described in more detail, the individual parts used in the partition are introduced first. Only then, the inner workings of the partitioning method itself can be laid out.

6.1 Anchors and Boundaries

The algorithm can only work if the system provides a representation of anchor and boundary elements and the necessary data to process them correctly. The following paragraphs introduce both terms and how they are represented in the tool.

6.1.1 Anchors

For simplicity, an anchor is represented by an infinitely small point in space that has no volume. The user can define to which extent the anchor grows by setting its maximum distance. An example of this is shown in figure 3.2 in the Requirements chapter. To enable nested anchors and hierarchical partitions, an anchor has a *layer* property. Besides the enclosing anchor, only anchors of the same layer will interact directly with each other. In order to visualize the resulting volume and be able to distinguish different anchors, the user can additionally choose a color for each anchor. Apart from these properties, anchors store a set of positions that are part of their subspace which is called *subspacePositions*. That set is continuously filled with voxel positions until the partition ends. It is later used by other components to create the mesh associated with the anchor's subspace. Anchors can be added by using the provided prefab or by adding the *Anchor* component to an existing object. This allows the system to find it when collecting the anchors.

6.1.2 Boundaries

The anchors' counterparts are boundaries. They also have a *layer* property, because the algorithm needs to know when to take a boundary element into account. Boundaries are used to stop an anchor's expansion. Apart from that, they store no additional data. A boundary can be any object. The system provides simple boundary prefabs in the form of planes which can also be used as *delimiters* as they are configured to be invisible in the final game. They already have the *Boundary* component attached so that the tool can gather them. However, the user can add this script to any object they want to turn into a boundary.

6.2 Voxel Creation

Now that the core components are known, the steps of the algorithm are outlined and their implementation is explained. Note that the following sections ignore hierarchical setups for the beginning and therefore describe what happens when processing a single layer. Section 6.4 will then introduce the concept of nested partitions and how these are handled.

6.2.1 Partition Data Structures

As explained before, the algorithm uses a basic graph-based breadth-first search starting at each anchor's position and expanding as far as possible. This search technique was explained in section 4.3. Remember, that in order to realize any graph-based search algorithm, the system needs two data structures: a *frontier* queue and a *reached* set. Additionally, the voxels used to define an anchor's subspace have to be represented somehow.

Frontier. A queue that holds all voxels that are not yet processed but will be part of the partition. In order to maintain the correct order of its elements, the frontier is realized by a FIFO queue as described in section 4.3.1.

Reached. A set that contains all positions that were reached until now. Its purpose is to avoid redundant insertions of positions to the frontier, which in turn would create an endless loop in the voxelization phase. A position is added to the reached set when it is enqueued to the frontier. Contrary to the frontier queue, the order of items in the set is not important as the system only needs to know *if* a position was already reached and not when the insertion happened.

VoxelData. Objects that store data for a voxel. For the underlying algorithm, these are the following properties:

- The voxel's position in world space.
- The anchor it belongs to.
- The voxel's *creator*. This is the position of the voxel which added this voxel to the queue.

VoxelData objects are used for the frontier queue in the partitioning phase. Whenever the position stored in the frontier is mentioned, this refers to the corresponding voxel data. The anchor subspaces and the reached set, however, truly only store the positions as these are the only information needed for the later mesh creation.

6.2.2 Preparing the Partition

Now that the core components are known, this section can briefly summarize what setup tasks have to be done prior to the main loop of the partition algorithm developed for this thesis. The preparation phase consists of five parts:

1. Reset all data structures to clean up previous results.
2. Collect all anchor and boundary objects present in the gamespace.
3. Initialize all anchors.
4. Adjust the *frontier collider* object, which is used to check for collisions in the main loop of the partition.
5. Initialize the frontier queue and the reached set.

Steps 3 and 5 require further investigation and are therefore explained in more detail in the following paragraphs.

Initializing the anchors. The partitioning algorithm takes three parameters as inputs. The first one is the set of anchors to include in the partition. The remaining two values are used to define if the partition's results will be visualized by meshes and whether these will be transparent or not. Each anchor will conduct the following steps to set itself up for the partition:

1. Standardize its position so that it can be represented by three integers.
2. Add that rounded value to its set of subspace positions.
3. Create a material for visualizing its produced subspace mesh after the partition.
4. If the layer is not 0, check if the anchor is enclosed by another anchor's volume.

Standardizing simplifies the calculations immensely and decouples the partition's outcome from the exact anchor placement as postulated by the problem statement. For more information on the fourth step, consult section 6.5.

Initializing the frontier queue and the reached set. Before actually starting the partitioning's main loop, both the reached set and the frontier queue are properly set up. They are initialized with the positions of the anchors included in the input set, or their corresponding voxel data, respectively. The anchors' positions therefore become the Voronoi centers.

6.2.3 The Main Loop of the Partition

For realizing breadth-first graph search, the algorithm sequentially processes frontier elements until the queue is empty. As explained in section 6.2.2, the frontier initially con-

tains voxels located at the anchors' standardized positions. The following sections focus on how the voxel data is processed.

Processing voxels. Processing a voxel means removing it from the queue and checking if it can expand further. If so, all of its neighboring voxels are added to the queue as well. *Neighboring* hereby refers to the voxels that are right to, left to, over, under, behind, or in front of the current voxel. However, it is important to not add the creator of the currently processed voxel to the queue again. Otherwise, the partition will run into an endless loop and therefore never terminate. The expansion follows the neighbor propagation principle employed by Dorn et al. [10] and explained in chapter 2. An illustration of how neighbor propagation sequentially traverses a two-dimensional space can be found in figure 6.1.

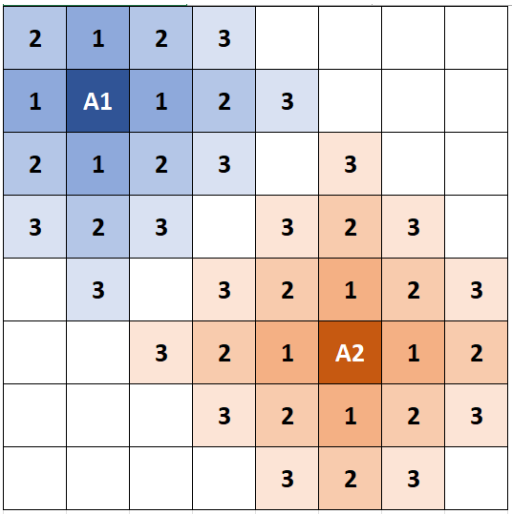


Figure 6.1: Illustration of the Neighbor Propagation Principle in 2D With Two Anchors [adapted from [40, p. 715], [15, p. 493]]

A voxel cannot expand in the following cases:

- There is a boundary element at the voxel's position.
- There is a voxel of another anchor at its position.
- The voxel detects the border of an enclosing anchor.
- The distance to its corresponding anchor exceeds this anchor's maximum distance.

In these cases, the voxel has to be *fixed*. This means it will not add its neighbors to the frontier so the anchor's expansion will stop at this position. Note, that checking the distance only matters if the anchor's layer is 0 because inner anchors will grow as far as they can within their enclosing volume. The upcoming paragraphs further go into detail on how the first three cases are handled.

Detecting boundary elements and enclosing anchors. Detecting boundary elements and anchor borders is done by moving the previously created frontier collider to the voxel's position and leveraging it to check for collisions with those objects. The frontier collider is a simple cube with the same dimensions as the voxels. For collision detection, it uses an AABB as defined in section 4.2.2. A bounding box is sufficient here, as a voxel

itself will always be a cube, so it can be perfectly approximated by a box. Voxels also retain the default rotation, so they are axis-aligned.

Detecting other anchors. Subspaces of anchors that are currently turned into a set of voxels cannot collide with the frontier collider at this point because their meshes are not yet constructed and therefore neither colliders nor knowledge of the final geometry exists. However, there is evidence of positions that already belong to another anchor in the form of their subspace positions. This means, checking if a position can be claimed can be done by performing lookups in all of these sets.

6.3 Creating Meshes

After every anchor is associated with a set of positions making up its subspace, these voxels have to be converted into a single, combined mesh. Of course, this step is only executed if the user configures the subspaces to be visible. The mesh construction employs the called *Marching cubes* technique, which was explained in section 2.3.1. It is performed for each anchor individually. Both the whole mesh and a version containing only the border are created. The latter is necessary for detecting mesh borders of enclosing anchors from the inside. Both of them use the material created by the anchor in its initialization phase. As said in the Tools chapter, the system utilizes functionality provided by an external package realizing the Marching cubes method. That existing implementation was slightly adjusted to fit the system's needs. The following paragraphs give extra insight into working with the package.

Creating the input array. The first step in the process of converting a set of voxels to meshes is to create a three-dimensional array representing the unit-sized voxels. Hereby, the array element's value is set to 1 if the corresponding voxel was inside the mesh to be created and 0 otherwise. This can directly be inferred from the anchor's set of subspace positions.

Choosing a pivot. A thing to consider when accessing array elements is that the positions stored in the anchor's set are global whereas the array indices always start at 0. This means the global positions have to be correctly mapped to the array's relative space to prevent any `IndexOutOfBounds` exceptions. This is where the so-called *pivot* comes into play. It marks the global position that is mapped to the array element at indices $(0, 0, 0)$. To achieve this, it resides at the subspace voxel with the smallest position. Each time a position from the anchor's subspace set is written to the array, the pivot position is subtracted from it first. This converts it to a position relative to the pivot and therefore within the array's dimensions.

6.4 Realizing Hierarchy

The previous steps are only applicable to anchors and boundaries sharing a common layer. In order to allow nested partitions, both voxelization and mesh creation have to be

executed several times while including the correct objects. To do so, anchors and boundaries collected from the gamespace are grouped by layer. Anchors have to be partitioned layer by layer, starting at layer 0 and treating the remaining layers in ascending order. A boundary element of layer n only influences layers n and greater. This means, that initially only the layer-0 boundaries are included in the partition, and in each iteration, a new layer of boundaries is added.

6.5 Quality of Life Features

This section describes additions that contribute to the overall appearance, usability, and user-friendliness of the system. Several features have to be mentioned here.

6.5.1 Tool Window

To offer a graphical interface for the user to configure and start the partition, the system provides a custom window that can be opened inside Unity. It works like any built-in window and can be used both in a standalone window and attached to the editor itself.

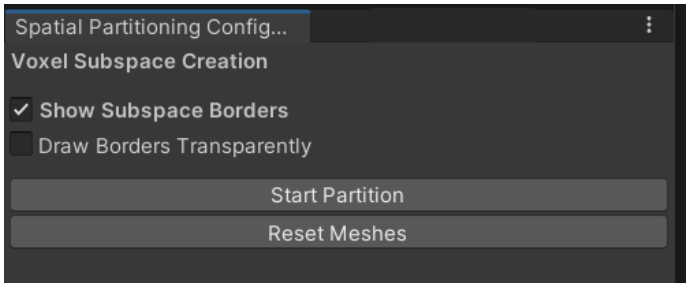


Figure 6.2: Partition Tool Configuration Window

It consists of two checkboxes influencing the visualization of the spatial partition, as well as two buttons for creating and removing the anchor volumes, respectively. A screenshot of the configuration window can be found in figure 6.2. It can be opened from the toolbar under the *Spatial Partitioning* tab.

6.5.2 Shortcuts

The methods for starting and resetting the partition can also be triggered using defined shortcuts. This facilitates a faster and more efficient workflow for experienced users. The shortcuts are *Ctrl + W* for starting the partition and *Ctrl + Shift + W* for deleting any produced meshes.

6.5.3 Gizmos

The Unity Manual describes gizmos as "graphics associated with GameObjects in the Scene" [35, ch. Gizmos]. As this excerpt implies, gizmos are not visible in the final game but can help to illustrate a game object's purpose during development. Single components like scripts can have custom gizmos, too. In the system created for this thesis,

custom gizmos visualize anchors and boundary elements. Anchors have no geometry as only their associated subspace is visualized after the partition. This means the user would not be able to see where they have placed them before executing the partition. Gizmos for boundaries additionally support comprehending which objects will contribute to the spatial partition. All in all, displaying said icons drastically improves the tool's usability and readability.

6.5.4 Hiding Tool Objects in the Game

The goal of the tool described in this thesis is to support the level design process as well as to provide location semantics for the actual game. It is important to separate the parts that are only used during the design process from those that are present in the final game. The system provides two scripts to support that separation. Firstly, it includes a component that deactivates the game object as soon as the game starts. For convenience, this component is already added to the boundary plane prefabs. But the user is free to attach it to any game object. Secondly, another script can disable the object's renderer instead of destroying the object holding the component. This is crucial for anchors, as it retains their volume's ability to contribute to collision detection. Disabling the renderer component only makes an object invisible, but not intangible.

6.5.5 Skipping Poorly Configured Anchors

A central characteristic of lower-level anchors is that their maximum distance will be ignored in the partition. This decision results in them being able to fill the enclosing anchor's volume. However, this means that anchors with a layer other than 0 would expand endlessly if they were not enclosed by an anchor with a higher layer. This, in turn, would lead to the partition never being terminated. To mitigate this, each anchor with a non-zero layer searches for subspace volumes in its initialization phase as pointed out in section 6.2.2. If an inner anchor is not enclosed, it will be skipped by the partition to avoid the aforementioned endless loop. Simultaneously, a warning is displayed to inform the user about the poorly configured anchor being left out. The warning contains the name of the anchor in question so that the user can solve the problem with ease. The reader can find an example warning in figure 6.3.

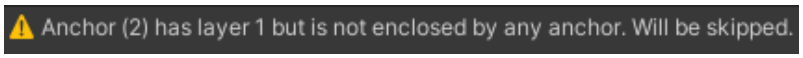


Figure 6.3: Warning Displayed by the Tool due to Poor Configuration

7 Optimizations

The initial implementation could barely handle anchors with a maximum distance of $m > 40$. Yet, the algorithm should be able to process big gamespaces with a plethora of boundary elements and anchors with a distance up to $m = 100$. That is why it needed some further inspection regarding performance, especially concerning CPU usage. The process of optimizing the code started with identifying problematic code blocks that produced a spike in CPU usage in Unity's profiler. At this point, the C# and .NET documentations came in very handy because they helped to identify expensive operations.

7.1 Problem Locations

The main focal point for identifying performance and memory issues was the main loop of the partition because it is the most frequently executed part of the algorithm. The time-consuming operations could be traced back to poor use of data structures. Initially, lists were used to store both reached positions as well as each anchor's subspace positions. The use of lists here drastically slowed down the partition process because the loop performs element queries on them in each iteration. According to the C# documentation, checking if an element is contained in a list is a linear operation, meaning it has to traverse all contents of the list for each query [24].
The second big bottleneck was checking if a similar voxel already existed in the queue when adding a new one. The idea was to avoid an endless loop by re-inserting the same positions back and forth. Similar to lists, checking for an element is linear in the number of contained elements as stated by the documentation [25]. As the number of voxels grows very fast, especially for big maximum distances, the number of elements in the queue gets rather high very quickly, which makes this operation very expensive.

7.2 Solutions

The solution to the first problem is to replace the lists with other data structures that bring better performance in operations that are frequently used in the underlying algorithm. Mainly, these are the `Contains()` and `Add()` methods. As the Microsoft Learn site proposes, hash sets are well suited for this purpose because querying if an element is present in the set is an $\mathcal{O}(1)$ operation, which means that the runtime increases linearly in the number of elements stored in that container. If a hash set has enough capacity, adding an element is equally performant [23]. The set can be initialized with a sufficiently high capacity to ensure that. Turning both anchor subspace position lists and the reached set into hash sets enhances the performance immensely. This will be backed up by concrete figures in section 7.3.
However, using a hash set instead of the frontier queue is not possible, because this data structure does not preserve the order in which elements are inserted. Yet, BFS relies on a

FIFO queue. Fortunately, the operation can be omitted by checking whether the voxel's position was already in the reached set, rather than in the queue. That is possible because any time a voxel gets enqueued, it is added to the reached set at the same time.
Smaller improvements are reusing as much data as possible, especially for frequently used objects like the voxel data. Also, to avoid copying large amounts of data, the `ref` keyword is used in most of the functions, because it forces the parameters to be passed by reference as pointed out by the dedicated documentation [26]. Another helpful change to reduce memory consumption is to use non-allocating alternatives where possible.

7.3 Performance Improvement in Figures

Together, these changes heavily decreased the computation time while improving memory usage. This section focuses on the computation time, which was measured using the C# `Stopwatch` class. The benchmarking was carried out on a Windows x64-based system with an Intel i7-1185G7 processor, 3GHz, an Intel Iris Xe graphics card, and 16GB RAM. The tests included single anchors of radius 10, 40, and 100, two size-40 anchors, an anchor with radius 40 next to a boundary, and the first hierarchical test setup from figure 3.10a. Each test was performed in isolation and repeated 20 times to obtain the average computation time. In the plot in figure 7.1, the red bars show the run-time before switching to hash maps and removing the frontier query, and the blue ones represent the algorithm after applying the changes.

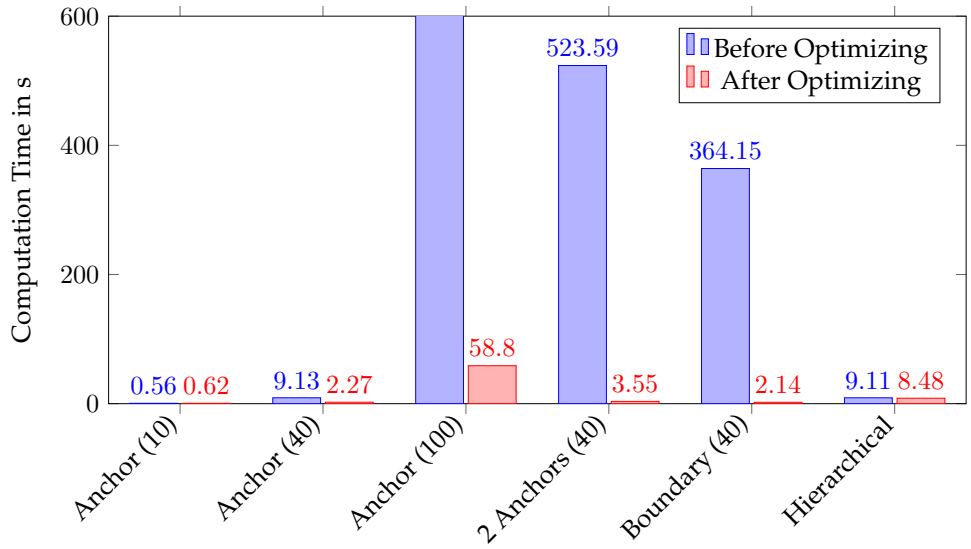


Figure 7.1: Average Computation Time Before and After Optimizations

From the plot, one can tell that, especially when dealing with large anchor distances, the optimizations drastically speed up the process of partitioning the space. Note that the test with the anchor of radius 100 was aborted in the variant without optimizations because it took over an hour. There is still room for improvement, as will be shown in chapter 10.

8 Use Case Example: Labeling System

A potential use case of the spatial partitioning tool is creating a subspace labeling system. The system provides a prefab that can collide with the anchor volumes. It is an empty game object which can be parented by any other game object. Said prefab can retrieve the current subspace. Here, it is used to display text configured in the anchor volume when the player enters it, but the user is at liberty to change or extend this behavior.

8.1 Realizing Non-Blocking Detection

At this point, it is helpful to explain the decision behind using an extra object for collision detection with the volumes. The problem can be traced back to a central property of the anchor volumes: Their geometry is not necessarily convex, because they can wind and flow around corners. Yet, the anchor volumes have to use exact colliders to avoid false positives in collision detection later in the game. The player should never collide with a subspace they have not yet entered. In Unity, only convex colliders can be *trigger* colliders, meaning the subspace volume hinders objects from passing it [36]. This means preserving the exact collision a concave mesh collider provides would prohibit other objects from entering said collider. The latter is important because otherwise, the player could not move around in the partitioned gamespace.
As said before, the solution here is to employ an empty game object to detect the collision. The anchor volumes, indeed, have non-trigger concave mesh colliders, but they are configured to only collide with the empty object. This is achieved using Unity's layer collision matrix which defines "which GameObjects can collide with which [l]ayers" [37].

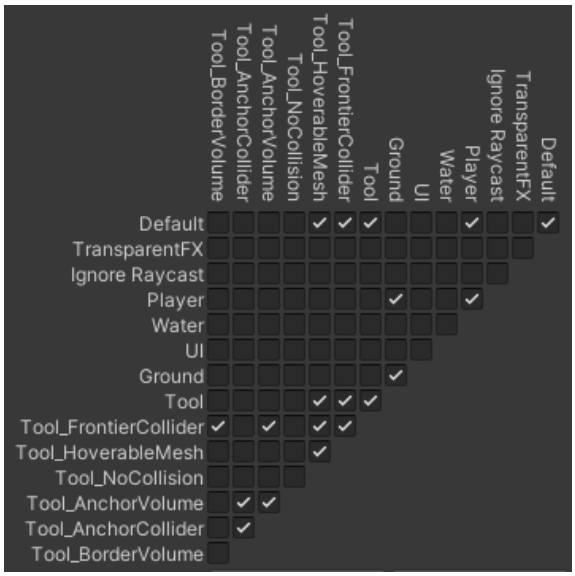


Figure 8.1: Layer Collision Matrix Used for the Tool

Figure 8.1 shows the layer matrix set up for the system. From the matrix, the reader can grasp that only the anchor collider and the frontier collider can collide with the anchor volumes. The empty game object must have any type of convex collider that can then be configured as a trigger, so it can detect volumes while passing through them at the same time.

8.2 The Actual Labeling System

Now, that the challenges for subspace collisions are out of the way, it is time to show what a labeling system could look like. Each anchor now has an additional *label* variable defining what to display on collision. The user can configure it in the inspector as shown in figure 8.2. When the anchor collider prefab detects an anchor volume, the respective

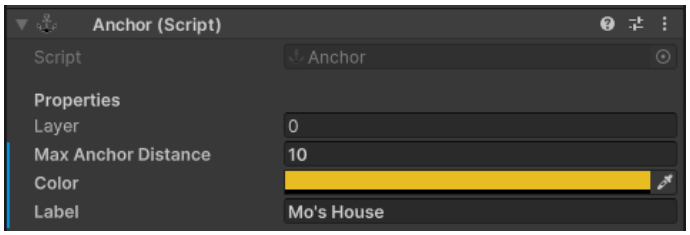


Figure 8.2: Anchor Label Value in the Inspector

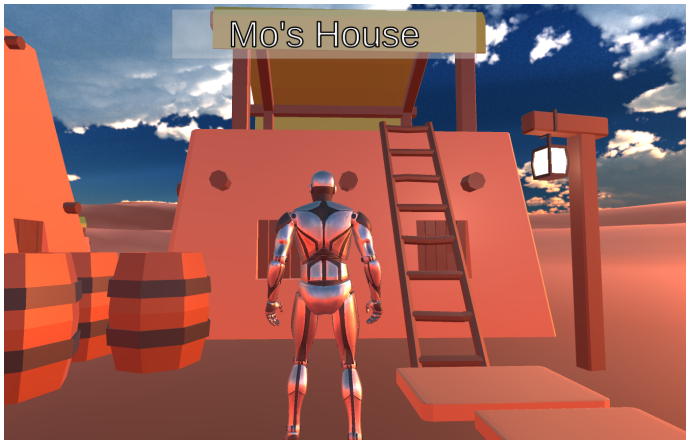


Figure 8.3: Simplified Labeling System

label is shown at the top of the screen. A simple realization of this can be seen in figure 8.3. A possible alternative would be to invoke an event on player collision. These events could then for example spawn a boss, play music, or change the weather in that region. Using events would enable the designers to utilize the space partition in any way that suits their software the best.

9 Assessment of the Goals

Chapter 3 presented some test setups the algorithm should be able to handle correctly. The current chapter assesses whether the requirements for each setup were met. For the failing ones, it explains why the algorithm failed the postulation and lists possible solutions to these problems. Furthermore, it covers how the algorithm managed to partition the complex example scene.

9.1 Test Setup Results

The reader might not remember all of the test setups described in chapter 3 in detail. To avoid going back and forth between chapters, the more complicated setups are repeated here by showing them side to side with their respective outcome. Again, there is a subsection for each setup category.

9.1.1 Single Anchor of Different Sizes

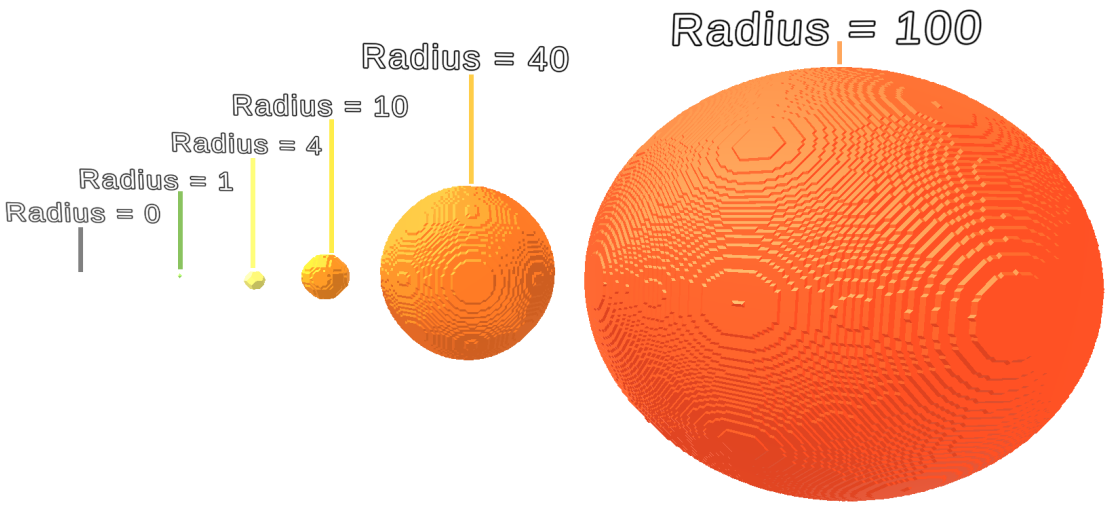


Figure 9.1: Single Anchors with Increasing Maximum Distances

The first setup contains a set of single anchors with different maximum distances. All of the produced meshes match their expectations. The reader can find the results in figure 9.1. Most importantly, the anchor with radius 0 did not produce a mesh at all. This test set's secondary purpose was to evaluate the performance of the algorithm. As benchmarking was conducted and presented in section 7.3, that information is omitted here.

9.1.2 Combination of Anchors without Boundaries

The next setup set revolves around combining anchors and assessing how their volumes interact with each other. Figures 9.2 and 9.3 depict the results for two or three anchors, respectively. As the reader can tell from these results, the algorithm partitioned the space correctly. However, the volumes do not stop early enough. Meaning, the subspaces overlap by a small amount. This is due to the use of uniformly sized voxels and `OverlapBox()`, causing the expansion to stop at the border, not the step before. A potential fix for this would be to leverage mesh boolean operations and cut the resulting anchor meshes so that they have a clean, non-overlapping border. The anchor subspaces expand one step too far by design, to enable mesh cutting in the future. More explanations on that can be found in section 10.5 of the Future Work chapter.

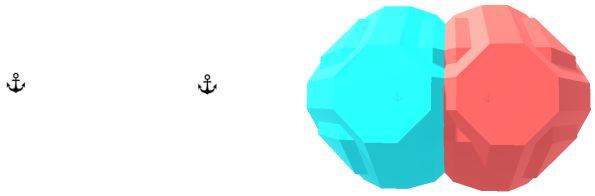


Figure 9.2: Three Anchors without Boundaries

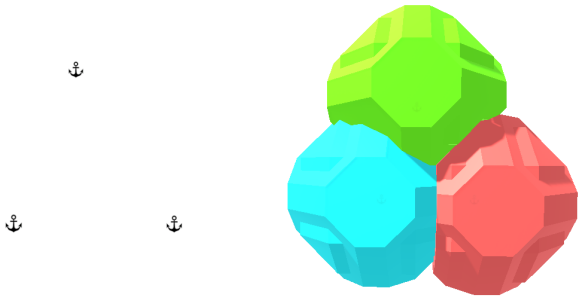


Figure 9.3: Two Anchors without Boundaries

9.1.3 Anchors with Different Boundary Configurations

Figures 9.4 to 9.9 show the results of the different boundary configurations described before. Similar to the previous setups, the results look as intended, except for the volumes reaching through the boundary by a tiny amount. Again, the expansion stops when colliding with the boundary element and not in the previous step. From this point on, the fact that volumes are slightly overlapping is not explicitly stated for each setup, as it applies to all cases and can be changed in future iterations of the system. For the special case where the anchor itself already touches a boundary, some people might prefer that no volume is drawn. This could be achieved by additionally checking if an anchor is blocked in its initialization, and skipping those anchors as well. In the underlying implementation, the solution shown by figure 9.8 was preferred because it provides informative feedback to the user. If no volume is created at all, a warning should be displayed.

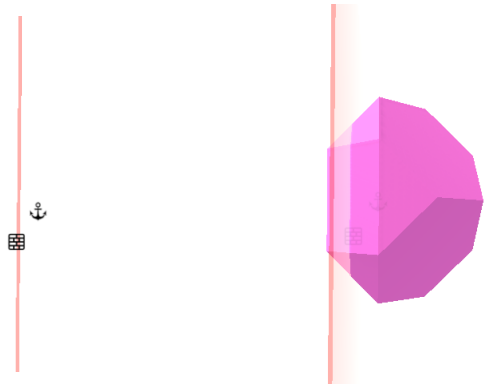


Figure 9.4: Anchor Next to Vertical Boundary

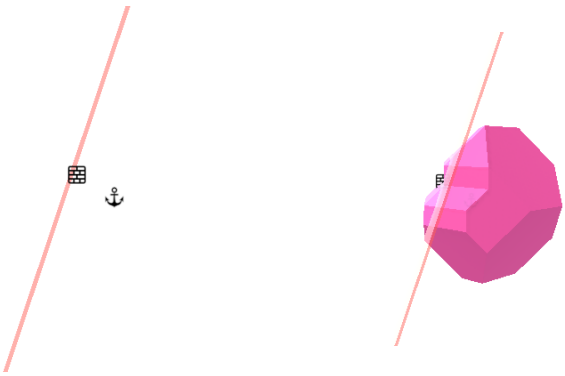


Figure 9.5: Anchor Next to Diagonal Boundary

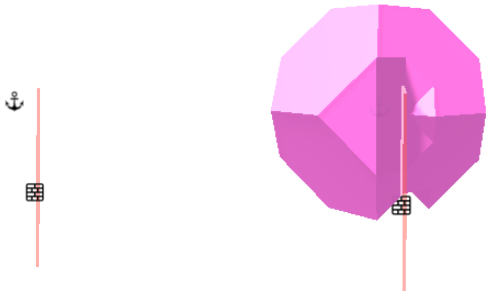


Figure 9.6: Anchor Growing over Boundary

9.1.4 Single Anchors in Combination with Boundary Shapes

Investigating how anchors expand with different kinds of shapes nearby is the first step toward testing the algorithm in a more realistic scenario. The shapes used for this were buckets, tubes, and a tent asset. The outcomes of these space divisions can be perceived in figures 9.10 to 9.13. At this point, the reader is reminded that the subspace volume in the long bucket test is blocked by an invisible boundary to be more readable. Regarding the results, the partition is influenced correctly by any shape, no matter how complex. Both the tube test, present in figure 9.12, and the long bucket test, observable in figure 9.10, require additional remarks because they show some more interesting characteristics. The tube anchor's volume does not change when increasing its maximum distance because

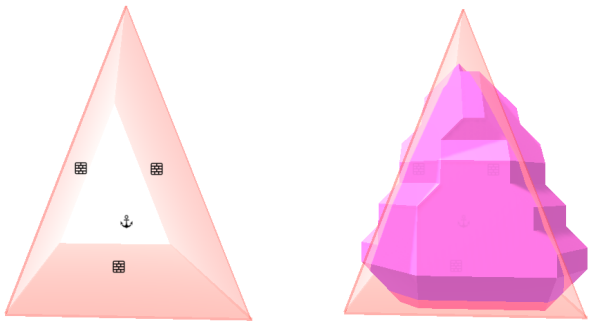


Figure 9.7: Anchor Boxed in by Boundaries

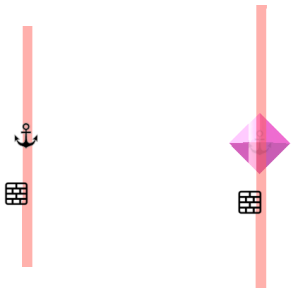


Figure 9.8: Anchor at Boundary

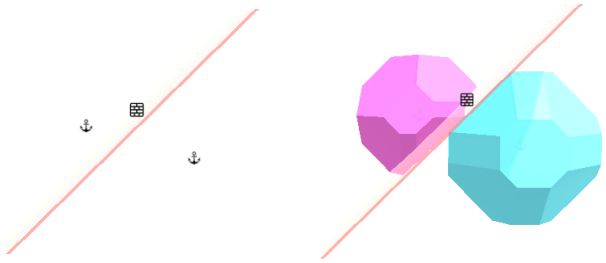


Figure 9.9: Anchors Next to Diagonal Boundary

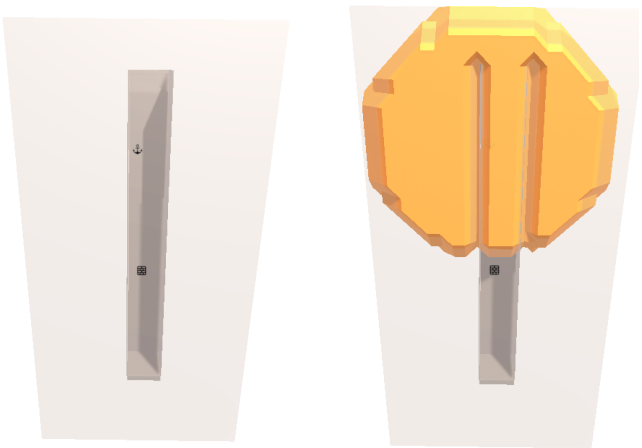


Figure 9.10: Anchor in Long Bucket

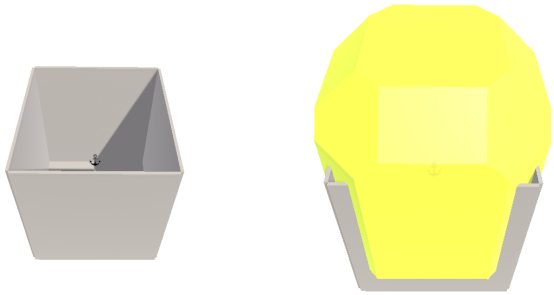


Figure 9.11: Anchor in Bucket

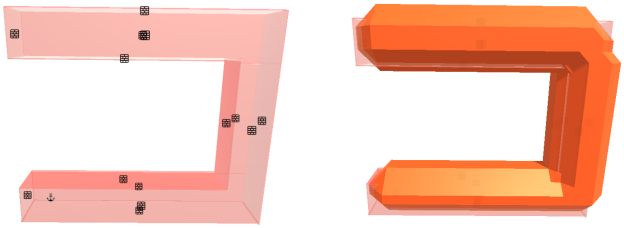


Figure 9.12: Anchor in Tube

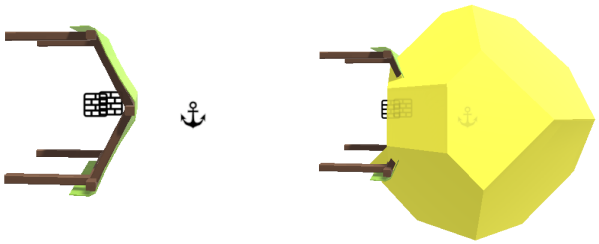


Figure 9.13: Anchor with Complex Boundary

it is completely boxed in by the tube boundaries. Similarly, the long bucket anchor's subspace mesh only goes so far into the bucket, as it sits near the top of the bucket and might not reach the bottom, depending on the configured maximum radius.

9.1.5 Hierarchical Anchor Setups without Boundaries

The following paragraphs focus on the hierarchical aspects of the partition. Hereby, the partition was executed with the option to draw the borders transparently, so it is possible to see inside the subspace meshes. Additionally, the outer anchor's color is white to increase the legibility of the inner volumes. As figure 9.14 shows, the inner anchors only grow inside the enclosing volume. Within it, they expand as far as they can. Moreover, figure 9.15 presents that this holds for the lower layers as well, as the layer-2 anchors are now restricted by the respective layer-1 volume.

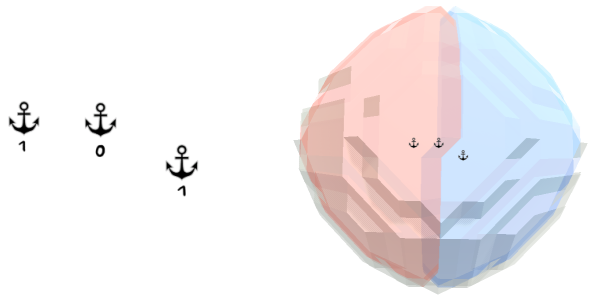


Figure 9.14: Hierarchical Setup Containing two Layers

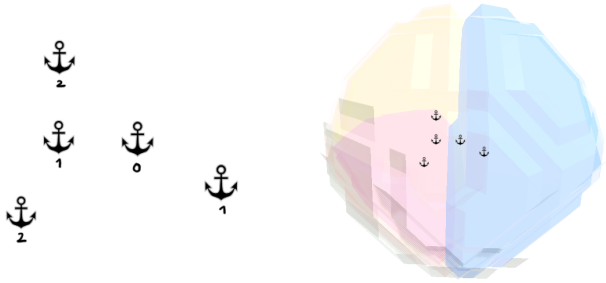


Figure 9.15: Hierarchical Setup Containing three Layers

9.1.6 Hierarchical Anchor Setups with Boundaries

The final test set now combines the challenges of the previous categories with hierarchical boundary elements. The first setup, represented by figure 9.16, hereby adds a layer-1 boundary cutting through the anchors diagonally. As intended, this plane only affects the inner volumes. That can be inferred from the observation that the red anchor cannot expand fully to the right anymore. This especially becomes clear in comparison to the results shown in figure 9.14. As postulated in this test's description in chapter 3, the outer anchor remains unchanged. Likewise, adding another boundary of layer 0 influences all anchors of this partition. Figure 9.17 illustrates that, presenting the outcome of the second hierarchical boundary test. Therefore, the algorithm also fulfills the requirement that a layer-n boundary element must influence all anchors of layer n and higher.

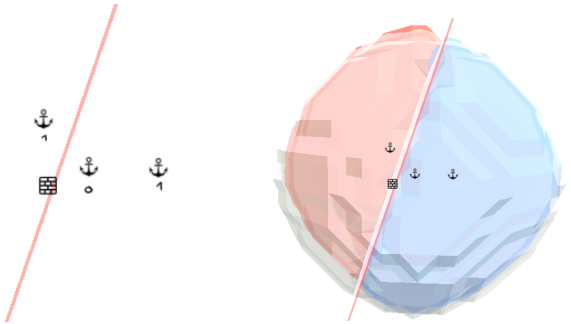


Figure 9.16: Hierarchical Boundary Setup

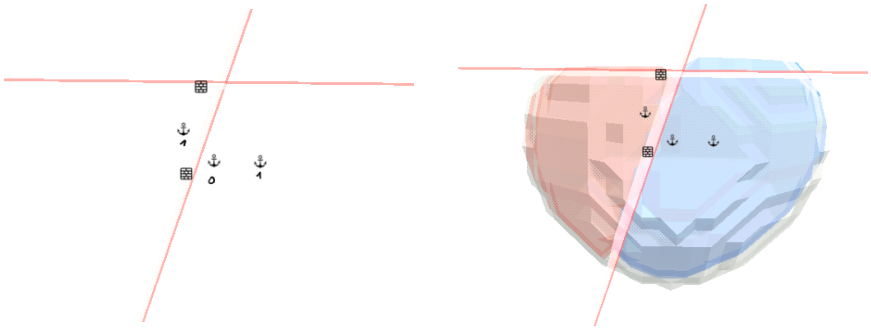


Figure 9.17: Hierarchical Setup With Two Boundaries

9.2 Complex Environment

Last but not least, this section comments on how the system handles the desert village. With transparent borders, the algorithm produced the subspace meshes depicted in figure 9.18. The magenta shapes each correspond to an anchor of layer 0. Remember, that these are the anchors that were placed at groups of houses.

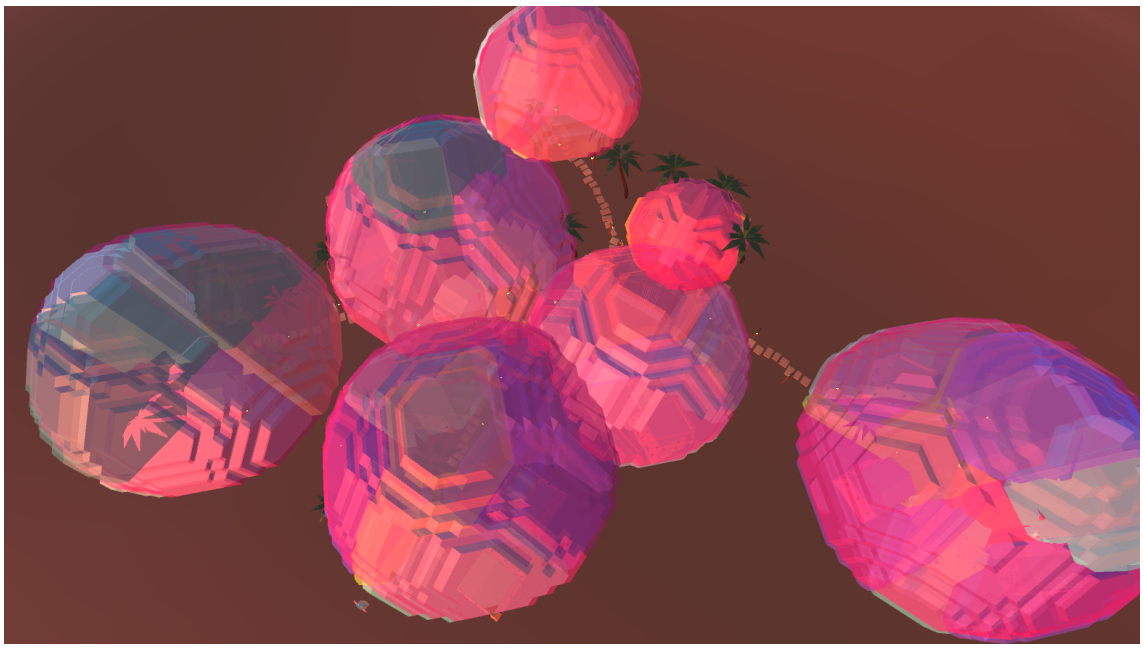


Figure 9.18: Hierarchical Spatial Partition of the Desert Village

For better readability, figure 9.19 shows the partition without the outer anchors' volumes so that the lower-level anchors become visible. The village's partition only uses two layers of anchors. The delimiter boundaries heavily influenced the partition, as the house anchors did not grow past them. This can especially be observed in the lower middle group of figure 9.19 where the yellow space was defined by the delimiters. Partitioning this complex environment took about 45 seconds on average.

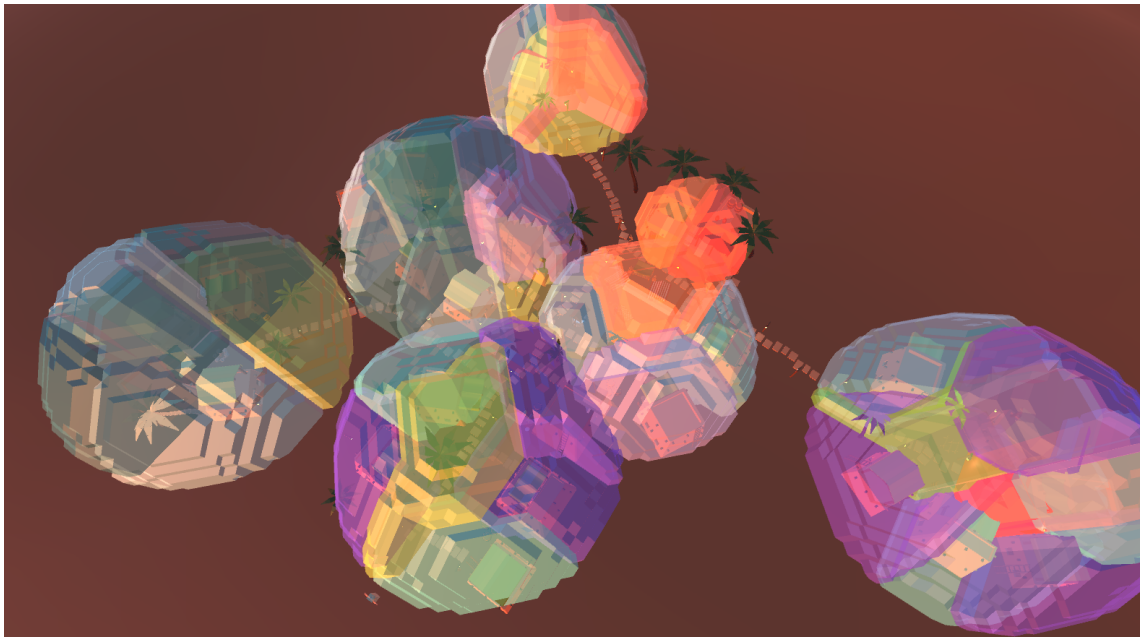


Figure 9.19: Hierarchical Spatial Partition of the Desert Village without outer Anchors

9.3 Discussion of the Algorithm

Overall, the system yields the expected results and is simple to use. The problem of subspace meshes overlapping slightly with each other or with boundaries, respectively, can be fixed in future extensions of the algorithm. To do so, the resulting meshes have to be cut accordingly. This will be elaborated on in section 10.5 in the upcoming chapter. However, there still is an edge case that causes an endless loop in the partition. If a certain combination of nested anchors and a layer-0 boundary occurs, the voxels of the inner anchors grow through the border of the enclosing anchor on the opposite side of where the boundary cuts the mesh of the outer anchor. This has to be further investigated in the future.
The performance can be assessed as good, yet the computation time increases drastically with the number of anchors with a large maximum distance in the scene. But, this is tolerable because the partition will likely only run once or twice for a level. With the inclusion of a chunk-based update system, the waiting time could be decreased even further. Refer to section 10.4 for a more detailed outlook on that.
Another thing to consider is that the partition depends on the order in which the anchors are processed. When collecting the anchor objects as described in chapter 6, the order of anchors is random. However, this only affects the voxels lying in regions where two anchor borders are touching. These border voxels will be assigned to the anchor that expands first and is involved in this region. The rest of the partition will produce the same outcome. Therefore, this inconsistency only affects a very small portion of the partition.

10 Future Work

The system developed in the context of the underlying thesis is far from being optimal. However, it can serve as a foundation for future research and implementation approaches in this area. In the following paragraphs, future additions and important considerations are presented and investigated.

10.1 Volumetric Anchors

In the presented version of the partition algorithm, anchors are modeled by infinitely small points with no volume. However, there could be the desire to use arbitrarily shaped objects as anchors. Just like boundary elements that can have any shape. Turning any object into an anchor would require being able to work with centers that have a certain volume. The algorithm could be expanded to accept these as anchors in the future.

10.2 Automatically Define Delimiters

In the complex environment shown in figure 3.13, there is a strong presence of delimiter elements. In the case of the desert village level, these must be placed manually by the level designer to tweak the partition's outcome to their needs. They are positioned in special, custom positions to achieve the desired effect. In other cases, however, the location of delimiter objects is rather obvious. An example of this would be an open fence gate which should block the anchor volume from growing out of the fenced-in area. Another one would be between the pillars of a temple. These are situations in which the tool could automatically insert delimiter objects to assist the designer even more. Nevertheless, the option to manually place and edit delimiters should be retained.

10.3 MipMap Levels for Voxels

MipMapping is a term originating from the field of computer graphics. It is a texture mapping technique where a single texture stores several levels of detail which are created by merging neighboring pixels into one by averaging texture values like color [11]. The characteristic that could be leveraged in this thesis' application is the use of different levels in different contexts. When creating voxels for the spatial partition, it would be beneficial to adapt to the state of the surrounding anchors and boundaries. Concretely, the voxels should start big and get increasingly smaller in the vicinity of other anchor volumes or boundary elements. If a big voxel competes with another, this part of the space should be divided using smaller voxels recursively.

10.4 Chunks and Dirty Flags

The present version of the system recalculates all anchor volumes when it is executed. This can lead to redundant computations if only a few anchors were edited. To improve that, the gamespace could be divided into so-called *chunks*. Chunks are uniformly sized subparts of the gamespace. If anchors located within their area have changed compared to the previous execution, the whole chunk is marked as *dirty*. Additionally, chunks adjacent to dirty ones and affected by updated anchor subspaces are flagged as well. Restarting the partition algorithm would then only recompute dirty chunks. This way, the majority of the gamespace remains untouched and the computation time for completing the spatial partition would heavily decrease.

10.5 Cut Anchor Volumes

As pointed out in previous chapters, the subspace meshes extend one step too far. This way, the system guarantees that any anchor volume touches boundary elements or other anchor meshes, respectively, to avoid any unassigned space. However, this can lead to an improper separation of the gamespace as pointed out in chapter 9. To achieve a clean space division, the protruding part of the subspace mesh has to be cut according to boundaries and other anchor volumes. For boundaries, their geometry can simply be subtracted from the anchor mesh. The situation becomes more difficult when multiple anchors overlap because here the shared space has to be divided equally. Especially the second case has to be further investigated and properly solved in the future.

10.6 Improvements to the Configuration Window

Lastly, several improvements can be made to enhance the usability of the configuration window. As a start, the system could remember the visualization settings made in the tool window so that they influence partitions produced by using the shortcut as well. Further, the window could present a readable list of all the anchors and boundaries that will be included in the partition. Such an overview could be further extended by providing the possibility to add and remove anchors or boundaries via the window. This would remove the task of finding the game object in either the scene or the hierarchy and disabling it in order to exclude it from the next partition. On top of this, a custom inspector could be integrated into the configuration window, so that the user can edit the values of the object's *Anchor* or *Boundary* components directly from the overview list. All in all, the tool window has room for improvement to enhance both user-friendliness and efficiency.

11 Conclusion

This paper proposes a system to achieve a spatial understanding of an object's semantic location in a gamespace. Hereby, the Voronoi algorithm is used as a role model. The tool enables the user to place anchor points in the gamespace that grow as far as they can to partition the space accordingly. Additionally, boundary elements can be positioned to influence the partition. For better usability, a configuration window is provided to the user, allowing them to manipulate the result's visualization. For more experienced users and faster execution, shortcuts for the basic functionality are included.
The tool could be beneficial for applications like pathfinding, rendering, loading, or labeling systems. A simplified example of the latter was presented in chapter 8.
To solve the challenge of partitioning a gamespace, the algorithm roughly performs three major steps. At the start, it collects all anchors and boundaries present in the current gamespace. As mentioned earlier, the anchors then extend as far as possible, while taking other anchors and boundary elements into account. Finally, a single, connected mesh is created for each anchor's computed subspace. To avoid interferences with any game logic, the results are not visible in the later game, yet, their data can be used in code. The system provides a mechanism to detect these meshes and retrieve information about their corresponding subspace. To achieve a better performance regarding computation time and memory usage, the program code was analyzed by leveraging different tools and language-specific documentations.
The algorithm was evaluated using a set of toy setups as well as a more realistic level example to define the desired requirements. Overall, the partitioning system produces the intended outcomes. Yet, the mesh clarity can be enhanced by cutting them in a later extension of the algorithm. Other possible future additions contain an improved configuration window providing more options to include and exclude anchors and boundaries from the partition. Apart from that, further optimizations like chunks and dirty flags could be considered. Finally, applying MipMap techniques, as well as supporting volumetric anchors are additional wishes for the following versions of the partition tool.

List of Figures

Bibliography

[1] Oswin Aichholzer, Wolfgang Aigner, Franz Aurenhammer, Thomas Hackl, Bert Jüttler, Elisabeth Pilgerstorfer, and Margot Rabl. Divide-and-conquer for voronoi diagrams revisited. In *Proceedings of the twenty-fifth annual symposium on Computational geometry*, SoCG '09. ACM, June 2009.

[2] Daniel Arribas-Bel and Martin Fleischmann. Spatial signatures - understanding (urban) spaces through form and function. *Habitat International*, 128:102641, 2022.

[3] Franz Aurenhammer and Otfried Schwarzkopf. A simple on-line randomized incremental algorithm for computing higher order voronoi diagrams. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 142–151, 1991.

[4] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, dec 1996.

[5] Claudio Belloni. Games engineering: a formal approach to gamespaces, 2023.

[6] Jon Louis Bentley, Franco P. Preparata, and Mark G. Faust. Approximation algorithms for convex hulls. *Communications of the ACM*, 25(1):64–68, January 1982.

[7] Praveen Bhaniramka, R. Wenger, and R. Crawfis. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):130–141, 2004.

[8] Jung-Woo Chang, Wenping Wang, and Myung-Soo Kim. Efficient collision detection using a dual obb-sphere bounding volume hierarchy. *Computer-Aided Design*, 42(1):50–57, 2010. Advances in Geometric Modelling and Processing.

[9] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Voronoi Diagrams*, pages 147–171. Springer Berlin Heidelberg, 2008.

[10] Sebastian Dorn, Nicola Wolpert, and Elmar Schömer. Voxel-based general voronoi diagram for complex data with application on motion planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 137–143, 2020.

[11] J.P. Ewins, M.D. Waller, M. White, and P.F. Lister. Mip-map level selection for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 4(4):317–329, 1998.

[12] Umar Farooq and John Glauert. Faster dynamic spatial partitioning in opensimulator. *Virtual Reality*, 21(4):193–202, February 2017.

[13] Steven Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, page 313–322, New York, NY, USA, 1986. Association for Computing Machinery.

[14] Sarah F. F. Gibson. *Constrained elastic surface nets: Generating smooth surfaces from binary segmented data*, pages 888–898. Springer Berlin Heidelberg, 1998.

[15] Licai Guo, Feng Wang, Zhangjin Huang, and Naijie Gu. A fast and robust seed flooding algorithm on gpu for voronoi diagram generation. In *2011 International*

Conference on Electrical and Control Engineering. IEEE, September 2011.

[16] Justin Hawkins. Marching-cubes. https://github.com/Scrawk/Marching-Cubes, 2017. [Online; accessed January 08, 2024].

[17] Kenneth E Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, 1999.

[18] Hugo Ledoux. Computing the 3d voronoi diagram robustly: An easy explanation. In *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, pages 117–129, 2007.

[19] William E. Lorensen and Harvey E. Cline. *Marching cubes: a high resolution 3D surface construction algorithm*, pages 347–353. ACM, July 1998.

[20] Maxime Maria, Sébastien Horna, and Lilian Aveneau. Topological space partition for fast ray tracing in architectural models, spatial partitioning. In *2014 International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 1–11, 2014.

[21] Talha Bin Masood, Hari Krishna Malladi, and Vijay Natarajan. Facet-jfa: Faster computation of discrete voronoi diagrams. In *Proceedings of the 2014 Indian Conference on Computer Vision Graphics and Image Processing*, ICVGIP '14, New York, NY, USA, 2014. Association for Computing Machinery.

[22] D. Meneveaux, K. Bouatouch, E. Maisel, and R. Delmont. A new partitioning method for architectural environments. *The Journal of Visualization and Computer Animation*, 9(4):195–213, October 1998.

[23] Microsoft. Hashset<T>.Add(T) Method. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1.add?view=net-8.0, 2024. [Online; accessed February 17, 2024].

[24] Microsoft. List<T>.Contains(T) Method. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.contains?view=net-8.0, 2024. [Online; accessed February 17, 2024].

[25] Microsoft. Queue<T>.Contains(T) Method. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1.contains?view=net-8.0, 2024. [Online; accessed February 17, 2024].

[26] Microsoft. ref (c# reference). https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref, 2024. [Online; accessed February 17, 2024].

[27] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, 2006.

[28] G.M. Nielson. On marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):283–297, July 2003.

[29] Gregory M. Nielson. Dual marching cubes. In *Proceedings of the Conference on Visualization '04*, VIS '04, page 489–496, USA, 2004. IEEE Computer Society.

[30] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, feb 1977.

[31] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to

voronoi diagram and distance transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 109–116, New York, NY, USA, 2006. Association for Computing Machinery.

[32] Guodong Rong and Tiow-Seng Tan. Variants of jump flooding algorithm for computing discrete voronoi diagrams. In *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, pages 176–181, 2007.

[33] Stuart Russell and Peter Norvig. *Artificial Intelligence, Global Edition A Modern Approach*. Pearson Deutschland, 2021.

[34] Elijah Smith, Christian Trefftz, and Byron DeVries. A divide-and-conquer algorithm for computing voronoi diagrams. In *2020 IEEE International Conference on Electro Information Technology (EIT)*. IEEE, July 2020.

[35] Unity Technologies. Gizmos and icons. `https://docs.unity3d.com/560/Documentation/Manual/GizmosMenu.html#GizmosIcons`, 2017. [Online; accessed January 20, 2024].

[36] Unity Technologies. Introduction to collision. `https://docs.unity3d.com/Manual/CollidersOverview.html`, 2024. [Online; accessed January 30, 2024].

[37] Unity Technologies. Layer-based collision detection. `https://docs.unity3d.com/Manual/LayerBasedCollision.html`, 2024. [Online; accessed January 30, 2024].

[38] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph.*, 21(4):153–162, aug 1987.

[39] Christopher W Totten. *Architectural Approach to Level Design*. CRC Press, 2019.

[40] Jue Wang, Fumihiko Ino, and Jing Ke. Prf: A fast parallel relaxed flooding algorithm for voronoi diagram generation on gpu. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 713–723, 2023.

[41] Yusheng Xu, Xiaohua Tong, and Uwe Stilla. Voxel-based representation of 3d point clouds: Methods, applications, and its potential use in the construction industry. *Automation in Construction*, 126:103675, 2021.

[42] Fan Yang, You Li, Mingliang Che, Shihua Wang, Yingli Wang, Jiyi Zhang, Xinliang Cao, and Chi Zhang. The polygonal 3d layout reconstruction of an indoor environment via voxel-based room segmentation and space partition, spatial partitioning. *ISPRS International Journal of Geo-Information*, 11(10):530, October 2022.

[43] Zhan Yuan, Guodong Rong, Xiaohu Guo, and Wenping Wang. Generalized voronoi diagram computation on gpu. In *2011 Eighth International Symposium on Voronoi Diagrams in Science and Engineering*, pages 75–82, 2011.

[44] Peter Zerfass, Christian D. Werner, Frank B. Sachse, and Olaf Dössel. Deformation of surface nets for interactive segmentation of tomographic data. *Biomedizinische Technik/Biomedical Engineering*, 45(s1):483–484, 2000.

[45] Krista Rizman Žalik and Borut Žalik. A sweep-line algorithm for spatial clustering. *Advances in Engineering Software*, 40(6):445–451, 2009.