Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Interactive room detection

Bc. Martin Němec

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Němec Martin**	Personal ID number:	**493248**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Graphics**		

II. Master’s thesis details

Master’s thesis title in English:

Interactive Room Detection

Master’s thesis title in Czech:

Interaktivní detekce místností

Guidelines:

Review the existing algorithms for scene partitioning into cells and portals. Implement the Breaking the Walls algorithm [1] to detect rooms and portals connecting them in 2D scenes. The implementation will be done inside the Unity framework. The input of the algorithm will be a set of walls defined as 2D line segments. An important requirement for the implementation is a fast response and the possibility of editing individual walls with the help of local data modifications. Suggest a way to apply the algorithm to 3D data. Evaluate the implementation for computational complexity on at least five scenes of different sizes. Use the acquired room information in a simple application using potentially visible sets (PVS).

Bibliography / sources:

[1] Lerner, Alon, Yiorgos Chrysanthou, and Daniel Cohen-Or. "Breaking the walls: Scene partitioning and portal creation." 11th Pacific Conference on Computer Graphics and Applications, 2003. Proceedings. IEEE, 2003.
[2] Lerner, Alon, Yiorgos Chrysanthou, and Daniel Cohen Or. "Efficient cells and portals partitioning." Computer Animation and Virtual Worlds 17.1 (2006): 21-40.
[3] Haumont, Denis, Olivier Debeir, and François Sillion. "Volumetric cell and portal generation." Computer Graphics Forum. Vol. 22. No. 3. Oxford, UK: Blackwell Publishing, Inc, 2003.
[4] Lefebvre, Sylvain, and Samuel Hornus. Automatic cell-and-portal decomposition. Dissertation thesis, INRIA, 2003.
[5] Oliva, Ramon, and Nuria Pelechano. "NEOGEN: Near-optimal generator of navigation meshes for 3D multi-layered environments." Computers & Graphics 37.5 (2013): 403-412.

Name and workplace of master’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **15.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

doc. Ing. Jiří Bittner, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank Doc. Ing. Jiří Bittner, Ph.D. for leading this thesis and for all the consultation and the help provided.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 24, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. května 2024

Abstract

The aim of this thesis is an implementation of an 2D algorithm for room detection as well as creation of a Cells and Portals partition using the Breaking the Walls algorithm. The input consisting of separate walls defined as line segments is first processed into a half-edge data structure from which the resulting Cells and Portals partition is created. Finally, potentially visible sets are generated for each room. An important part of the algorithm is also the ability to locally update the partition without having to rebuild the whole data structure. Using Unity engine, a prototype has been implemented, allowing for easy manipulation and creation of the scenes, as well as control over different steps and values of the algorithm or visualisation of the potentially visible sets. This prototype works both in the Unity editor and in the standalone version.

Keywords: Breaking the Walls, Cells and Portals, Room detection, Potentially visible sets, Unity

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Praha 2,
Karlovo náměstí 13,
E-420

Abstrakt

Cílem této práce je implementace 2D algoritmu pro detekci místností a vytvoření rozdělení na Buňky a Portály pomocí algoritmu Breaking the Walls. Vstupem jsou samostatné stěny definované jako úsečky, které jsou nejprve zpracovány do datové struktury half-edge. Z této struktury je následně vytvořeno konečné rozdělení na Buňky a Portály. Nakonec jsou pro každou místnost vygenerovány potenciálně viditelné množiny. Důležitou součástí algoritmu je také možnost lokální změny bez nutnosti přestavby celé datové struktury. V rámci Unity engine byl implementován prototyp umožňující snadnou manipulaci a tvorbu scén, stejně jako kontrolu nad různými kroky a hodnotami algoritmu či vizualizací potencionálně viditelných množin. Tento prototyp funguje jak v Unity editoru, tak i ve samostatné verzi.

Klíčová slova: Breaking the Walls, Buňky a Portály, Detekce místností, Potencionálně viditelné množiny, Unity

Překlad názvu: Interaktivní detekce místností

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Goals 2

2 Analysis **3**
2.1 Data structures 3
2.1.1 Indexed triangle mesh 3
2.1.2 Winged edge 4
2.1.3 Half-Edge data structure 5
2.2 Visibility culling 7
2.2.1 Potentially visible sets 8
2.2.2 From-region Cells and Portals 10
2.2.3 Point-based Cells and Portals 11
2.2.4 Volumetric Cells and Portals 11
2.2.5 Voronoi segmentation 13
2.2.6 Morphological segmentation . 13
2.3 Breaking the walls 16
2.3.1 Construction using BSP tree 16
2.3.2 Quality of the partition 17
2.3.3 Breaking the walls algorithm 18
2.3.4 Initial Partition 18
2.3.5 Refinement 19
2.3.6 Breaking the walls 20

3 Implementation **21**
3.1 Interactive room detection 21
3.2 Game objects 22
3.3 Steps of execution 23
3.3.1 Splitting the walls 23
3.3.2 Calculating half-edges 27
3.3.3 Generating rooms 28
3.3.4 Mesh generation 30
3.4 Breaking the Walls 30
3.4.1 First pass 32
3.4.2 Second pass 34
3.5 Potentially visible sets 36
3.6 Usage for 3D 37

4 Results **39**
4.1 Half-edge structure 40
4.2 Breaking the Walls algorithm . . 42
4.3 Potentially visible set 45
4.4 Final time 48
4.5 Local modification 49

5 Conclusion **51**

Bibliography **53**

A Additional Test Results **55**
A.1 Cross 56
A.2 Houses 58
A.3 FEL 60
A.4 Vienna 62
A.5 Pompeii 64
A.6 Soda Hall 66

B User Manual **69**
B.1 Editor mode 69
B.1.1 Parameters 70
B.1.2 Preprocess 71
B.1.3 Algorithm execution 71
B.1.4 Mesh generation 72
B.1.5 Structure deletion 72
B.1.6 Input and output 73
B.2 Run-time mode 73

C Content of attached medium **75**

Figures

2.1 Description of a single winged edge with all associated information. 4
2.2 Example of face ambiguity of winged edge structure, where faces can have differently directed edges. . 5
2.3 Description of a single half-edge with all associated information. 6
2.4 Pseudocode for acquiring the vertices and faces of a selected edge. 6
2.5 Pseudocode for acquiring all faces and edges that use a selected vertex. 6
2.6 Pseudocode for acquiring all faces and edges adjacent to a selected face. 7
2.7 Potentially visible sets of cell A: cell A is shaded dark gray, visible cells are light gray, and non-visible cells are white. 8
2.8 Binary space partition for a scene (left) shown as a tree (right). Planes a[1-4] are used as splitting planes for the BSP tree. Scene is split into cells A[1-3], B, and C. 9
2.9 Stab tree [left], potentially visible (light grey), visible sets (grey) of cell A [middle] and graph depicting connectivity of cells in the scene [right]. 10
2.10 Distance field representation of an architectural model, colour represents distance to closest geometry. Source of the image: [HDS03] 11
2.11 Different steps of the watershed algorithm. Source of the image: [HDS03]. 12
2.12 Stages of the Voronoi graph-based segmentation algorithm: (i) computation of the Voronoi graph, (ii) set of extracted critical points, (iii) critical lines, and (iv) segmentation after merging Voronoi cells. Source of the image: [BJL+16]. 14
2.13 Stages of the morphological segmentation algorithm: (i) initial floor map, (ii) iteratively eroded map, (iii) initial labelling of separated rooms, and (iv) segmentation after wavefront propagation. Source of the image: [BJL+16]. 15
2.14 Example of a result of morphological segmentation. Source of the image: [BJL+16]. 15
2.15 Step-by-step creation of a BSP partition and a tree over a scene. At each step a splitting plane is chosen and subspace is split. 16
2.16 Example of scene for a portal creation (left), created BSP portals (middle), ideal portal (right). 17
2.17 Traversal of an edge during Breaking the Walls algorithm of wall W. Wall W_{adj} is next traversed wall. Wall W_{cls} can potentially be connected by a portal (doted line). Only walls in grey subspace are tested considered to be potentially connected. 19
2.18 Enclosed geometry (a) gets connected by itself. Geometry (b) needs to be manually connected with the shortest portal. 19
2.19 (a) Input for BW algorithm. (b) White cells are underestimated, light grey are the ideal and dark grey are overestimated. (c) Ideal partitions. 20

3.1 A HalfEdge *test* is can be added as next half-edge of current one *curr* by testing it against both *test.prev* and *curr.next*. 23
3.2 Pseudocode for generation of SkipIndex and Vertices arrays. 25
3.3 Example of the Vertices and SkipIndex arrays for a multiple intersection. 26
3.4 Pseudocode for the splitting of line into line segments. 26

3.5 Pseudocode for validation and preparation of line segments for the half-edge connecting. 27
3.6 (a) Walls to intersect. (b) Intersected walls with half-edges. (c) Groups of half-edges to be tested. . 28
3.7 Pseudocode for half-edge connecting of two adjacent line segments. 28
3.8 Pseudocode for the Shoelace algorithm for testing CW or CCW direction of a polygon. 29
3.9 Detection of surrounding room for room A.out. Each room has inner and outer part of itself. First inner room intersected room without having the outer room being intersected is room B. 30
3.10 Testing the position of connecting points for a half-edge. On point lies of black line segment, the other point can only lie in the gray area. 31
3.11 An example of input case, where portal may be incorrectly constructed without using linecast. 31
3.12 Pseudocode for testing the valid portal between half-edges A,B. Points A, B are the closest points on each half-edge. 32
3.13 Pseudocode for the first pass of Breaking the Walls algorithm. The first pass generates partition by creating portals between two valid half-edges. 33
3.14 Pseudocode for the first part of the second pass of Breaking the Walls algorithm, which removes invalid portals. 34
3.15 Pseudocode for the second part of the second pass of Breaking the Walls algorithm, which generates the best valid portal in the whole room. 35
3.16 Computation of length of a split room. The algorithm calculates distance of each point from the selected point (0). Then the length of split room can be calculated as: (distance of 3 - distance of 1) + distance of portal along lines 1 and 4 + portal length. The gray lines represent distances for each vertex. 36
3.17 Pseudocode for the recursive generation of PVS for a single room. 36

4.1 PVS generation time depending on amount of rays. The tested values were 10, 30, 60, 100, 250. 45
4.2 Resulting times of the partition, split into half edge generation, Breaking the Walls first and second pass and PVS generation. 48

B.1 The room detection controller window used in editor mode. 70
B.2 Parameters in the controller window. 71
B.3 Algorithm execution in the controller window. 71
B.4 Example of a visualised PVS of a room. Dark red is selected room, light red are visible rooms. 72
B.5 Visual and deletion functions in the controller window. 72
B.6 I/O functions in the controller window. 73
B.7 The UI controller for the run-time mode. 73

Tables

4.1 Preview of tested scenes. 39
4.2 Amount of geometry and computation time for the half-edge structure generation. 40
4.3 Resulting half-edge partitions and resulting rooms. Red and blue arrows represent next half-edge of front and back half-edge for each wall. Green represents the direction of a wall. . 41
4.4 Created and deleted portals for each Breaking the Walls algorithm phase as well as their computation times. 42
4.5 Resulting Breaking the Walls partition. Second pass with portal ratio of 3. 43
4.6 Breaking the Walls partition with different portal ratios. 44
4.7 Tested values of Potentially visible set for different portal ratios. 46
4.8 Resulting PVS for selected (red) cells. The visible cells are shown as light red cells. 47
4.9 All measured times of the algorithm in seconds, along with the total time. 48
4.10 Total time for half-edge data structure and Breaking the Walls partition generation compared to the local update. Potentially visible sets generation is not part of the total time. 49

A.1 Measured values for scene Cross 56
A.2 Step-by-step partition of scene Cross. 57
A.3 Measured values for scene Houses 58
A.4 Step-by-step partition of scene Houses. 59
A.5 Measured values for scene FEL . 60
A.6 Step-by-step partition of scene FEL. 61
A.7 Measured values for scene Vienna 62
A.8 Step-by-step partition of scene Vienna. 63
A.9 Measured values for scene Pompeii 64
A.10 Step-by-step partition of scene Pompeii. 65
A.11 Measured values for scene Soda Hall 66
A.12 Step-by-step partition of scene Soda Hall. 67

Chapter 1
Introduction

1.1 Motivation

In this technological age, rendering is an important aspect of computer science. With faster computers and better hardware, there are attempts to increase the quality of renderings, but even with the fastest hardware, it is often impossible. If a scene with millions of different objects were to be rendered, the time needed for a realistic render may be impossibly high. Therefore, there is an attempt to decrease the number of objects needed to be rendered at any point.

There are many methods to speed up rendering, but most of them are trying to do one thing: reduce the amount of objects or geometry that needs to be rendered by skipping the non-visible geometry. The technique that accelerates rendering by removing unnecessary objects or geometry is called occlusion culling [COCS01]. One of the methods is the Cells and Portals [LG95] algorithm, which divides the scene into separate cells, from which the visibility between them can be calculated. Cells are basically rooms, defined as enclosed polygons. The portals then connect these rooms, similar to doors or corridors. On top of this partition, an adjacency graph can be built which represents connectivity of these rooms. The adjacency graph can be used to determine visible cells for each room, which are called potentially visible sets [Air90]. For potentially visible sets, it is typical to add any cell that can be seen from any point in that specific room. Then during the rendering, only the visible cells from the room in which the camera lies are drawn. This approach is advantageous, since the visibility for static scenes can be precomputed. However, the division into cells is only useful in enclosed scenes, such as architectural plans of buildings. If this algorithm were to be used on open scenes, where visibility would typically be very high, the more efficient choice would be just rendering everything.

Cells and Portals only usage is not for occlusion culling. Splitting the scene into smaller, more manageable cells makes the scene more natural and logical. Instead of having large rooms, having a greater number of smaller distinct rooms, where each room has its function and purpose, can be more understandable. Other usage is for some advanced navigational algorithms, such as the generation of navigational meshes [OP13]. By splitting the scene into smaller areas, navigation can be calculated for smaller rooms and connected together using the Cells and Portals adjacency graph, enabling faster and more precise navigation.

Additionally, when working with meshes and geometry, a quick way to traverse its topology is also needed for many of the algorithms. Often, during an algorithm's run, the data structure must be traversed in an organised, predetermined manner. For that reason, multiple data structures are created to speed up the traversal by keeping important information about their surroundings, which allows quick traversal over the geometry. These structures are described in the beginning of the analysis chapter.

1.2 Goals

The main goal of this thesis is the implementation of a Cells and Portals algorithm, specifically the Breaking the Walls [LCCO03] algorithm, which generates the partition from a half-edge data structure instead of a typical BSP tree. At the beginning, different data structures for storing geometry and various implementations for the Cells and Portals are studied. Afterwards, an algorithm for both the half-edge data structure generation and the Breaking the Wall algorithm is proposed and implemented in the Unity engine. The implementation also contains a method to update the entire structure without rebuilding it, which both the half-edge structure and the Breaking the Walls algorithm enable. For that, the half-edge data structure implementation is also reworked into a dynamic one using the Unity physics system. The algorithm will also contain a potential visible set generation, which will be visualised in the simple application. A way to apply this algorithm on 3D data will also be suggested. Lastly, the implementation will be tested on six different scenes with different parameters for its computation complexity.

Chapter 2
Analysis

This chapter contains the analysis of different data structures used to save the scene topology. It also provides an overview of the visibility culling technique for decreasing the amount of geometry needed to be rendered in the scene. Finally, different algorithms for generating the Cells and Portals partition are presented along with an idea behind potentially visible sets, also used in this thesis.

2.1 Data structures

There are multiple data structures, each useful for some different problem. Some structures may not store information about topology, which can be useful for the simple rendering of triangles. On the other hand, some structures keep information about all neighbouring elements of each element, which is useful for operations over the mesh itself. In general, every structure was created for a specific use. These data structures vary in stored information and space complexity, as they can save many times more data than some simpler ones. Still, they compensate for their requirements in their computation speed.

2.1.1 Indexed triangle mesh

The indexed triangle mesh [HvDM+13] is the easiest way to store information about the mesh, by using two separate arrays, one array of vertices and another of faces, usually called the list of indices. While the vertex array saves simply coordination of each vertex, the list of indices represents polygons, usually triangles, as sequences of indices of the vertex array. There are different types of indexation depending on the usage and efficiency of the mesh. Some of those methods described for the amount of n vertices are as follows:

- **Triangle list.** Sometimes also called *Triangle soup* is method of storing independent polygons. Each vertex is used exactly once by polygons, resulting in $3n$ vertices and indices.
- **Triangle strip.** Every triangle (except for the first one) shares exactly two vertices with the previous triangle, resulting in a total of $n + 2$ indices.

- **Shared vertex.** Similar to the triangle list, this method represents each triangle from three independent vertices but removes the redundancy of the vertices. In this way, each vertex can be used by multiple triangles at once. The number of indices is still *3n*, but the number of vertices can decrease considerably.

Although the first three methods are not often used for their redundancy or specific usage, the shared vertex method can be seen quite often, as it allows for a spatially effective and simple representation. The indexed representation can also easily be expanded with additional arrays, such as a normal or a colour array.

2.1.2 Winged edge

Winged edge [Bau72] is an edge-centred structure that allows for a fast traversal along the mesh of the object. This structure saves the topology of the mesh which is centred around edges of the mesh. The structure saves tables for every vertex, edge, and face. Although the vertices and faces tables are simple, as they only need information about an edge, which uses them, the edge contains information about:

- *PVT(E)*: previous (origin) vertex where edge E begins.
- *NVT(E)*: next (destination) vertex where edge E ends.
- *PCW(E)*: previous (left) edge in clockwise direction from edge E.
- *NCW(E)*: next (right) edge in clockwise direction from edge E.
- *PCCW(E)*: previous (left) edge in counterclockwise direction from edge E.
- *NCCW(E)*: next (right) edge in counterclockwise direction from edge E.
- *PFACE(E)*: previous (right) face of edge E.
- *PFACE(E)*: next (left) face of edge E.

The winged edge can be seen in Figure 2.1.

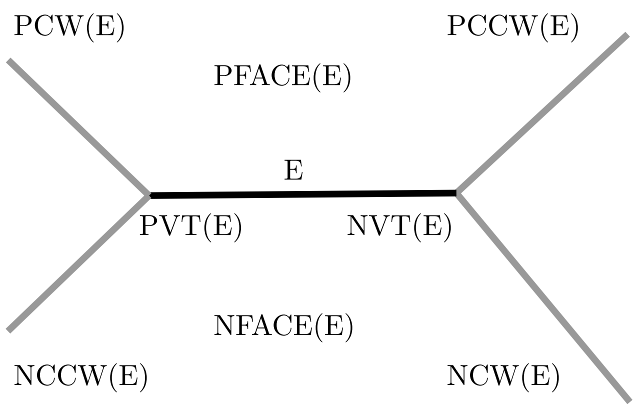


Figure 2.1: Description of a single winged edge with all associated information.

The stored information allows for fast execution of different topological queries or algorithms, such as the subdivisional surface or triangulation. Although the structure is useful for many algorithms, it may contain some inaccuracies. Depending on how the structure is designed, a problem may arise when three edges share the same vertex. In that case, the orientation of the faces will become ambiguous, visible in Figure 2.2. To ensure correctness, it is necessary to choose a correct vertex during the traversal.

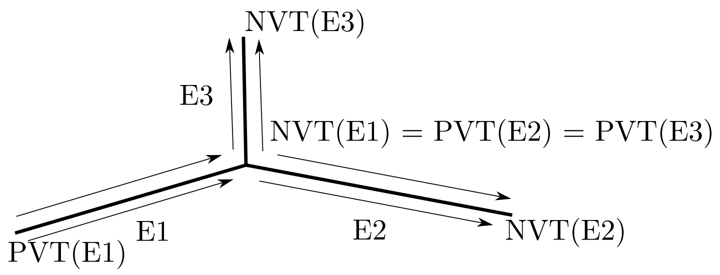


Figure 2.2: Example of face ambiguity of winged edge structure, where faces can have differently directed edges.

2.1.3 Half-Edge data structure

The half-edge data structure, also known as „Doubly connected edge list“ or „DCEL“, is an edge-centred data structure used to maintain information about the mesh topology, including information on the incidence of the vertex, edges, and faces. Every edge is split into two half-edges, each having an opposite orientation. Each half-edge contains information about:

- *Vertex* representing the end of half-edge.
- *Next* half-edge representing the neighbouring edge in counterclockwise order.
- Optionally *Previous* half-edge representing the neighbouring edge in clockwise order.
- *Opposite/Sibling* half-edge representing the other half-edge in which the original edge was split.
- incident *Face*, which the half-edge defines (if any).

visible in Figure 2.3.

Each face and vertex stores information only about one of its incident half-edges *edge*, since all other parameters are easy to find from the topology [Ket23]. Note that not every half-edge needs to have an incident face. Those edges would represent a hole in the mesh in 3D or a border of the mesh in 2D.

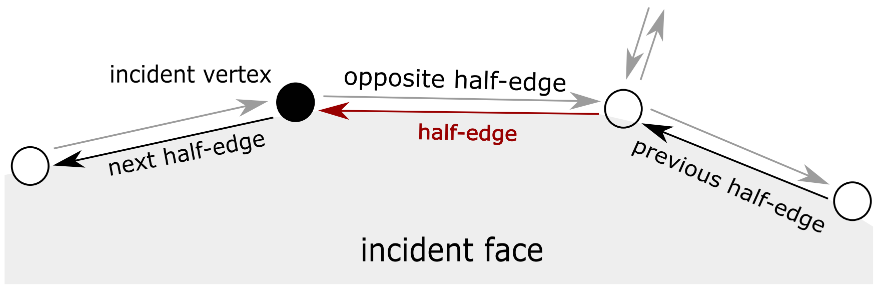


Figure 2.3: Description of a single half-edge with all associated information.

Additionally, due to the half-edge structure, some useful queries are stored directly within the structure [McG00].

- Which faces/vertices border an edge, see Figure 2.4.

```
    face1 = edge.face;
    face2 = edge.opposite.face;

    vertex1 = edge.vertex;
    vertex2 = edge.opposite.vertex;
```

Figure 2.4: Pseudocode for acquiring the vertices and faces of a selected edge.

With additional calculations, some more complex queries can be detected, [McG00] such as:

- Which faces/edges contain a vertex, see Figure 2.5.

```
    starting_edge = vertex.edge
    edge = starting_edge
    do:
        print(edge.face)    # face using vertex
        print(edge)         # edge using vertex
        edge = edge.next.sibling
    while( edge != starting_edge ):
```

Figure 2.5: Pseudocode for acquiring all faces and edges that use a selected vertex.

- Which edges/faces are adjacent to a face, see Figure 2.6.

```
starting_edge = face.edge
edge = starting_edge
do:
    print(edge)              # adjacent edges
    print(edge.opposite.face) # adjacent face
    edge = edge.next
while( edge != starting_edge ):
```

Figure 2.6: Pseudocode for acquiring all faces and edges adjacent to a selected face.

2.2 Visibility culling

When rendering a scene, drawing geometry, which the viewer cannot see is unnecessary. For that purpose, an attempt is made to determine whether a specific object is visible from the current view. Visibility culling aims to quickly detect invisible geometry that does not need to be rendered.

Typical visibility culling starts with two basic culling methods:

- **Back-face culling** which avoids rendering triangles not directed to the viewer.
- **Viewing-flustrum culling** which avoids rendering geometry outside the viewing flustrum.

The next technique, the occlusion culling technique, is used to avoid rendering primitives occluded by other geometry. Their designs are more complex, requiring knowledge of relationships between geometries of different objects.

What these algorithms have in common is that they use the concept of conservative visibility, which states that the output set must contain at least all visible objects. That is, the set can also include invisible objects, but there is an attempt to minimise their amount. Including that, this algorithm cannot label any visible object as invisible, which guarantees the accuracy of the output image [COCS01].

What makes visibility a complex problem is that in more extensive scenes, a slight movement of the viewpoint can cause significant changes in the scene's visibility. Additionally, with more extensive scenes, there are more objects that need to tested, making an ideal algorithm output sensitive. That means that the speed of the algorithm depends on the amount of visible geometry from the viewpoint.

Multiple algorithms are used to correctly and quickly calculate the problem of occlusion culling. Their speed is typically tied to the number of additional invisible objects they produce on output or the scene in which they are used. Some algorithms can be beneficial for some scenes, but not for others. An example can be large convex occludes for scenes with large faces or Cells and Portals for closed scenes.

2.2.1 Potentially visible sets

The method of potentially visible sets (shortened PVS) [Air90] splits the scene into separate areas (usually much smaller than the whole scene) and calculates which cells can be viewed from any point inside of this area, see Figure 2.7.

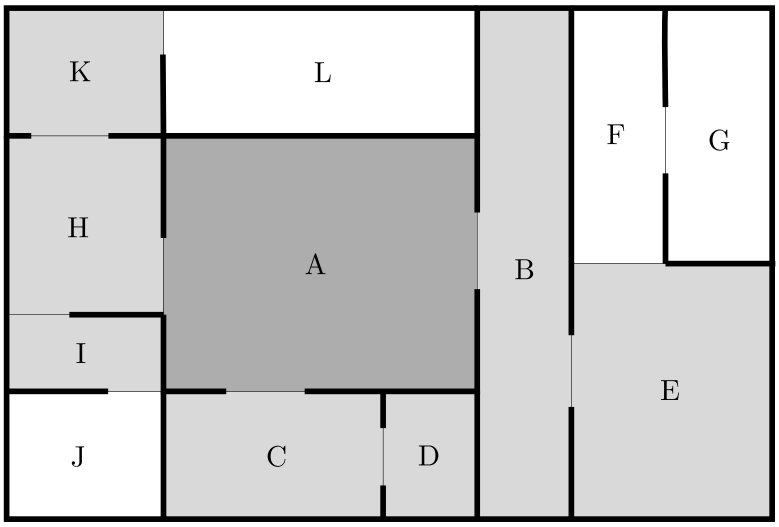


Figure 2.7: Potentially visible sets of cell A: cell A is shaded dark gray, visible cells are light gray, and non-visible cells are white.

By doing that, many of these cells, as well as all the objects lying inside of these cells, can be automatically removed from rendering, since there is no way for them to be visible.

There are two main goals for an ideal PVS:

- Minimise size of potentially visible sets.
- Minimise the number of cells. Split cells only when the resulting child cells have significantly smaller PVS.

In addition to that, there are other restrictions:

- It needs to be easy to determine the cell for current viewpoint.
- It must be generated automatically.

The restrictions indicate that a suitable data structure for range searching is necessary, such as binary space partition, kd-tree, or regular grid. Binary space partition being the most common choice, since it allows the splitting plane to be any general plane [Tót05]. The ideal splitting plane would be one that is mostly opaque, meaning that it fully represents some wall or obstacle in the model. An example of a split scene can be seen in Figure 2.8.

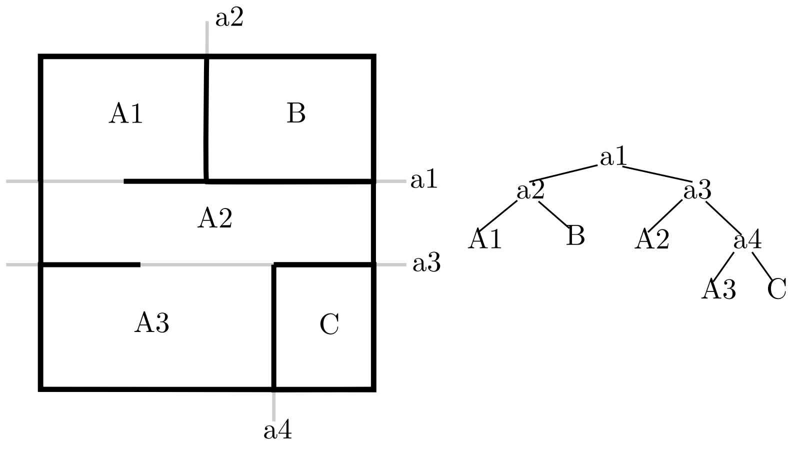


Figure 2.8: Binary space partition for a scene (left) shown as a tree (right). Planes a[1-4] are used as splitting planes for the BSP tree. Scene is split into cells A[1-3], B, and C.

After the BSP is constructed, the potentially visible cells can be calculated. If a cell is completely sealed, meaning that its boundary is composed of opaque surfaces, only the cell itself is included in its PVS. However, if there is an opening on its boundary called a portal, the PVS calculation becomes more difficult. A portal can be, for example, a hole in a wall, an open door, or a window. It's geometry is defined as an absence of polygons on the boundary, meaning it can be calculated using boolean operation on the cell boundary and polygons lying on the cell's boundary planes. The portals then connect two different cells that are visible to each other, but if a sequence of cells is selected, each connected by a portal, all the cells do not have to be visible from the beginning. This problem can be reduced to the polygon visibility problem, where polygons represent portals.

Calculating a PVS is equivalent to identifying the polygons that receive direct illumination from an area light source, where the portal acts like a light source. Overall, the exact solution for PVS is very complex, there exist many approximate ones. The most common ones are:

- **Point sampling algorithm** samples the polygon by a set of points and tests point-polygon visibility, usually by ray casting. Although the algorithm is simple, it can often underestimate the set of PVS cells.
- **Occlusion relations polygons** are based on the shadow volume algorithm [AAM04]. It used only part of all scene polygons (usually the largest ones) to test the visibility, which may overestimate the potentially visible set of polygons.

Since scenes need to be mostly occluded, this method is not ideal for open scenes. The best usage of this algorithm is for closed scenes, such as buildings, cities, and other architectural scenes. The calculation will also return geometry that can be invisible, meaning it uses the concept of conservative visibility.

Compared to other methods, this algorithm is usually used as a preprocess, which means that it is calculated on a static scene beforehand. This has significant advantages. Not only does the PVS information save computational resources during the walk-through, but it also allows for the precalculation of information about adjacent cells.

2.2.2 From-region Cells and Portals

The concept of Cells and Portals algorithm is similar to the potentially visible sets. The scene is subdivided into convex cells using a BSP tree and the main surfaces, such as the walls, are used as partitions to separate the cells. Non-occluding or non-opaque spots, such as doors or entrances, are used as the boundaries between the cells, called portals, to form the adjacency graph. Other non-occluding objects inside cells are generally ignored [Tel92].

There are two structures that are used to represent the visibility of a cell. First, an undirected graph that illustrates the connectivity between cells. Secondly, the so-called „stab tree“ expresses the direction of traversal of cells whose root contains the starting cell.

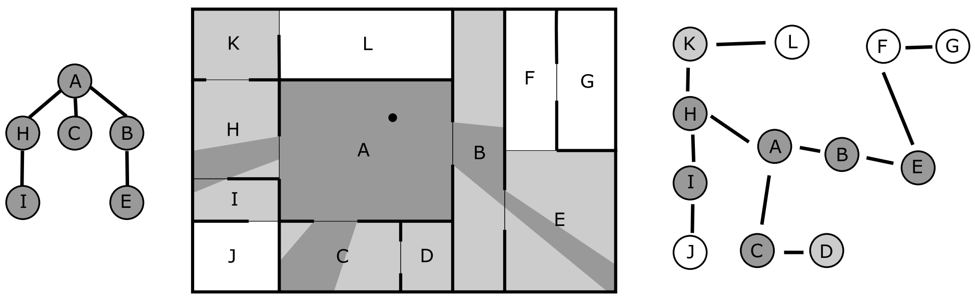


Figure 2.9: Stab tree [left], potentially visible (light grey), visible sets (grey) of cell A [middle] and graph depicting connectivity of cells in the scene [right].

The resulting hierarchy can be represented as an adjacency graph, where nodes represent a cell and edges connectivity between respective cells.

The visibility between the cells is determined by testing if there is a non-colliding line between two points, each in their respective cell. If such a line exists, it passes through the portals between these two cells. Therefore, all that is needed to determine is whether the portals are visible for the cell to be visible. Additionally, it is known that all visible cells are contained in the potentially visible set. Putting all this together, when the cell is visible, all cells on the path from the root to this cell inside the stab tree are also visible, and a sightline exists through all the portals bordering these cells.

2.2.3 Point-based Cells and Portals

Another approach to the Cells and Portals algorithm is that instead of precomputing the potentially visible geometry for each cell, the visibility can be calculated during the runtime using the recursive depth-first traversal of cells [LG95].

The algorithm works by first rendering the cell that the viewer is inside. Then, the visibility of connected portals is tested by projecting them into screen space and comparing them to an axis-aligned 2D bounding box containing the portal projection. Objects outside this bounding box cannot be visible and can be safely culled. The object inside the bounding box can possibly be seen through the portal, so they can be visible. As each successive portal is traversed, its bounding box intersects the previous bounding box, and the resulting box is used to cull the cell. During the traversal, objects inside the cell can only be tested using the current bounding box, which represents visible space inside their respective cell.

2.2.4 Volumetric Cells and Portals

This method for creating the Cells and Portals partition comes from a flooding simulation as an adaptation of the 3D watershed transform. Flooding originates from local minima, which is opaque geometry, such as walls or other obstacles. As the flooding regions expand, they connect. Portals are created in a way to avoid merging such regions. In a different way, a portal is created at the point where two obstacles are closest to each other, since that is where the flooded regions connect [HDS03]. Thanks to the distance field representation, this method is not limited to only geometric representation but also supports parametric, implicit, or volumetric representations.

The main idea of the algorithm is to sample the scene as a distance field, which is a discrete scalar field, where each sample point stores the distance to the closest geometry in the scene, visible in Figure 2.10.

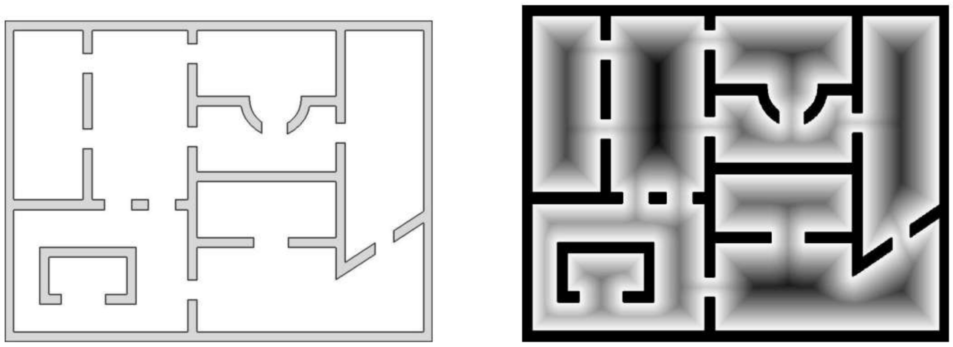
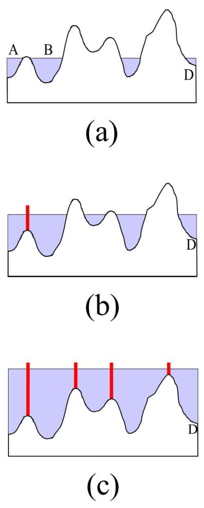
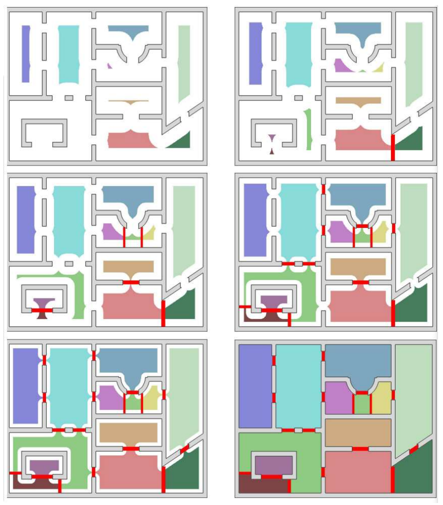


Figure 2.10: Distance field representation of an architectural model, colour represents distance to closest geometry. Source of the image: [HDS03]

The watershed transformation [MB90] is then used. The watershed method, often used in image processing, is a method used to separate different areas of a field on the basis of its gradient, such as grey-scale colour of an image. The watershed method can have different implementation approaches. The one described in this section is the watershed by flooding [MB90]. The basic idea consists of placing water sources in each regional minimum (walls in our case) to flood the surrounding area and then building barriers when different water sources meet. The watershed method proceeds at intervals, during which the flooding of the scene increases, visible in Figure 2.11. The resulting barriers then represent portals and a Cells and Portals partition can be created.



(a) : Representation of steps as flooding



(b) : Two basins connecting reveals an opening. A portal is built to separate them

Figure 2.11: Different steps of the watershed algorithm. Source of the image: [HDS03].

The disadvantage of a scalar field representation is it's memory needs. For that purpose, the hierarchy representation was proposed, which allowed for the decrease of memory required by combining blocks with the same values together.

2.2.5 Voronoi segmentation

Another way to detect a Cells and Portals partition is by using the Voronoi diagram [BJL$^+$16]. Although this is not necessarily a Cells and Portals algorithm, but instead an algorithm for a room segregation, both cells and portals can be extracted from the resulting partition.

A Voronoi diagram [O'R98, p. 155] is a geometric structure typically built on a set of points. The diagram divides a 2D plane into the so-called sites, where for each site S of point P is P the closest point of all the points in the diagram. The boundaries between those sites are defined by line segments. In the usual partition, lines, rather than points, are typically used for a representation of walls. That is why instead of using the point Voronoi diagram, the line-segment Voronoi diagram, where sites represent the set of closest points to the line-segment, is used. The difference between point and line-segment Voronoi graph is that, for the line-segments, the boundaries between sites are not only line-segments, but also parabolic arcs, which are created as a set of points equidistant from a line and a point.

After creating the Voronoi graph from the input lines, its leaves, which are lines or arcs connecting to any of the input walls, collapse into the node of their origin, visible in Figure 2.12, upper left image. Afterwards, every point in the pruned Voronoi graph is tested to see whether it has exactly two closest obstacle pixels, meaning that they lie between exactly two input walls. If such a point exists, it is considered a candidate for a critical point. The real critical points are then extracted from these candidates as the closest point to an obstacle within a certain neighbourhood. These points are visible in Figure 2.12, upper right image. The critical lines are then drawn into the partition so that the critical line for each point connects the point to its two obstacle points, visible in Figure 2.12, lower left image. The lines with a small angle between the critical point and obstacle point are usually removed, as they often lie in the corners of cells. The resulting partition can be seen in Figure 2.12, lower right image.

The easiest way to transform this room segregation algorithm into a Cells and Portals partition could be to instead of connecting the critical points with obstacle points just connecting the two obstacle points together and creating a single portal there. As the borders between rooms are close to straight line segments, the partition would usually stay very similar.

2.2.6 Morphological segmentation

Another way to segregate the area into rooms is by using morphological segmentation. By transforming the input into a grid map *M1*, whose pixels are labelled accessible or inaccessible. At first, the only inaccessible pixels in the map are those which lie on the input walls and pixels outside of the tested area, visible in Figure 2.13, top left image. Afterwards, the walls of *M1* are iteratively grown by one pixel. In each step, it is analysed whether the previously connected areas have become separated, visible in Figure 2.13, top right image. This process is repeated until a region has a certain

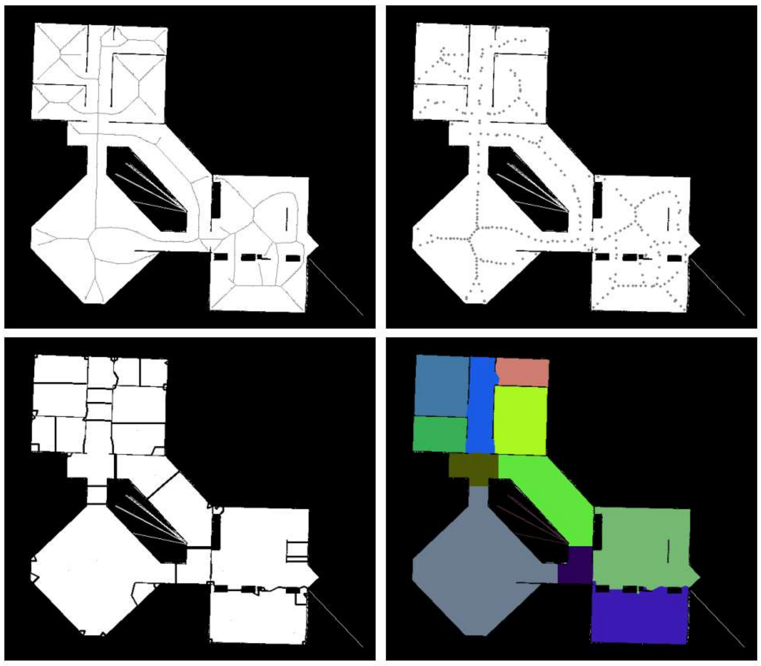


Figure 2.12: Stages of the Voronoi graph-based segmentation algorithm: (i) computation of the Voronoi graph, (ii) set of extracted critical points, (iii) critical lines, and (iv) segmentation after merging Voronoi cells. Source of the image: [BJL^{+}16].

size between the lower and upper thresholds, which represents the desired segment size. When that happens, all its cells are labelled as an individual room in the second map *M2*, which is a copy of the original *M1* map before the morphological iterations. The labelling procedure is repeated until all accessible cells are labelled as a room, visible in Figure 2.13, lower left image. Afterwards, when all rooms are labelled, the rooms are iteratively expanded using the wavefront propagation by usually expanding one pixel at a time into the surrounding accessible pixels. The algorithm ends when there are no accessible pixels in the map *M2*, visible in Figure 2.13, lower right image.

Compared to Voronoi segmentation, the morphological segmentation may be harder to transform into a Cells and Portals partition, as the borders between rooms are not line segments, but have some general shapes, visible in Figure 2.14. It may be possible by tracing the boundary and connecting the two endpoints, but even that will result in errors, meaning that some additional transformation would be necessary. Another way to detect portals may be during the room expansion, when if two separate rooms meet, the breath-first search through the accessible pixels may find two closest wall pixels, each on separate side of the connection pixel, where the rooms met. These two pixels could be connection points for a portal.

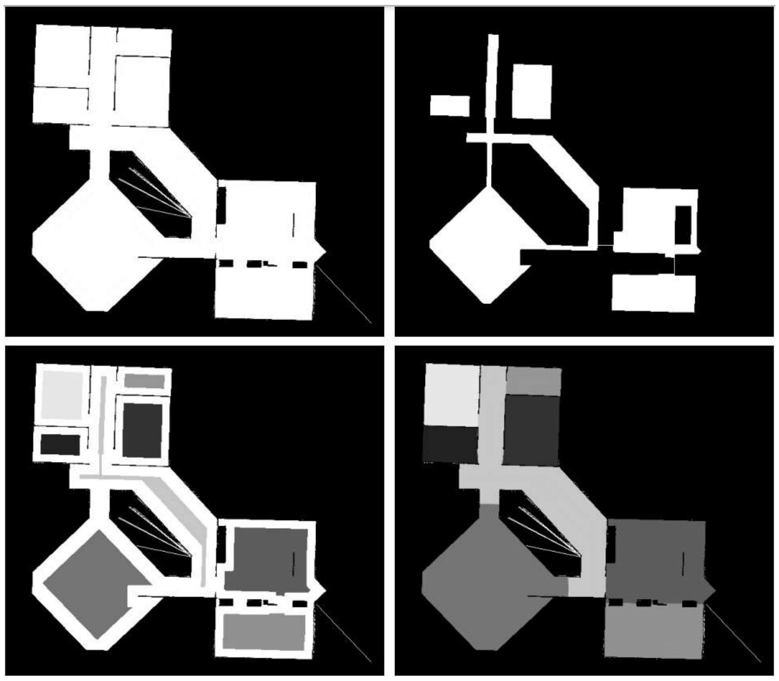


Figure 2.13: Stages of the morphological segmentation algorithm: (i) initial floor map, (ii) iteratively eroded map, (iii) initial labelling of separated rooms, and (iv) segmentation after wavefront propagation. Source of the image: [BJL$^+$16].

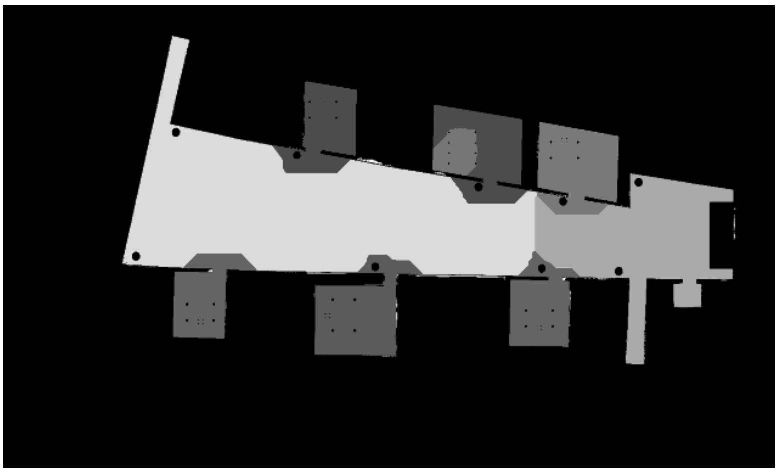


Figure 2.14: Example of a result of morphological segmentation. Source of the image: [BJL$^+$16].

2.3 Breaking the walls

Cells and Portals effectively decrease the amount of geometry to be rendered inside architectural scenes, but the problem is to build a graph defining the partition. These are two approaches:

- Manual partitioning.
- Automatic partitioning.

Manual partitioning is often used in the game industry. The reason is that the design has complete control over the partition and can ensure that the portals are placed the best way. This takes a lot of time and effort to create a suitable partition for a game designer, further scaling with the size and number of scenes. Furthermore, even the best designer cannot be sure if the partition created is optimal, as a single portal can drastically change the quality of a partition.

However, creating automatic partitions is a complex problem. The other problem is how the quality of the partition is defined, which can be viewed in many ways. What is a good number of portals, how do you determine what is a good position for a portal, etc.

2.3.1 Construction using BSP tree

The standard solution for automatic partitioning is a BSP tree, used in the from-region Cells and Portals algorithm [Tel92]. BSP is a binary tree created by recursively splitting the space into two subspaces using a splitting plane, visible in Figure 2.15. A splitting plane is defined for each non-leaf node and is chosen based on the defined walls. The tree is finished when every wall is used as a splitting plane. Portals are represented as polygons on the splitting planes and are created by recursively splitting themselves by traversing the tree. After reaching the leaf, a valid portal is built on the boundary of two leaf nodes that it has reached successfully [LCCO03].

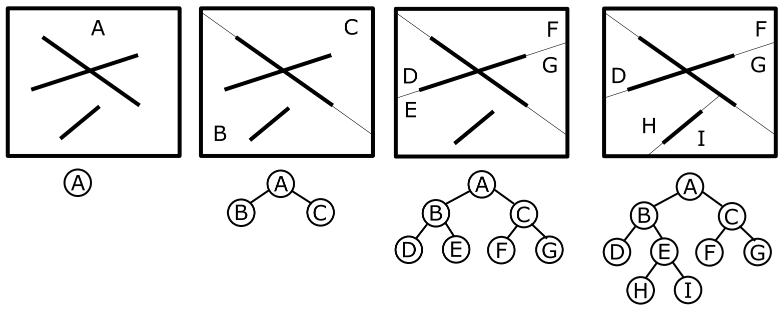


Figure 2.15: Step-by-step creation of a BSP partition and a tree over a scene. At each step a splitting plane is chosen and subspace is split.

The biggest problem is choosing the splitting plane during the BSP tree construction. The standard solution is to create a balanced tree and avoid over-splitting the space. Since the BSP tree is not used after the Cells and Portals graph is created, it is unnecessary to balance it. However, there are no splitting criteria to choose the splitting plane. The most used criteria are choosing the median splitting plane (plane with the same number of planes on the left and right of itself), a plane splitting the least amount of walls, or another similar approach.

The problem with the BSP tree is that the portal must lie on the splitting plane, which results in it not capturing the natural partitions induced by the model. In other words, the portal that could be placed naturally in a doorway cannot be created as a portal by BSP tree, resulting in the problem seen in Figure 2.16.

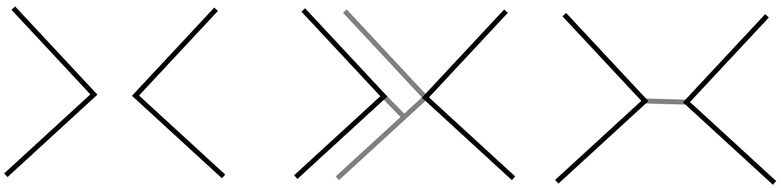


Figure 2.16: Example of scene for a portal creation (left), created BSP portals (middle), ideal portal (right).

2.3.2 Quality of the partition

As previously stated, the partition quality is not well defined. Ideally, the number of portals is proportional to the number of normal cells. This means that most viewpoints will have to render fewer polygons. But how to determine the specific amount? Too few portals and the cost of rendering hidden polygons will be the cost of polygon culling. Too much and the cost will outweigh the contribution of testing fewer portals.

According to [vdPS99], two cells with similar PVS can be merged while maintaining nearly the same quality. That means that they are most likely part of the bigger cell, which was split. It is assumed that the effective partition contains cells that do not have similar PVSs and therefore cannot be compressed. This knowledge can indicate whether a partition can still be improved.

The measurement suggested to evaluate the portals by [LH03] is defined for each portal as *score (p) = visible volume (p) * balance (p)*. *VisibleVolume* is an area within the model from which the portal is visible, and *balance* is defined as *balance(p) = |cost(cellA(p) - cellB(p)|*, *cost* being cost of rendering the cell. *CellA/B* are cells lying on each side of the portal *p*. The goal is to minimise the score function for each portal. This can lead to problems, mainly for the balance functions. Since the cost function is computed per portal, it can quickly reduce the cost to zero at the expense of creating a non-optimal portal.

2.3.3 Breaking the walls algorithm

Input for the Breaking the Walls algorithm is a set of half-edges. These edges can be extracted from a 2D floor plan. It is assumed that all walls are vertical, as non-vertical walls are ignored. The portals are half-edged created during the execution of the algorithm. For each portal, two opposite-orientated half-edges are created [LCCO03].

The scene is divided into cells. A cell is defined as a polygonal region with a boundary made up of a sequence of walls and valid portals. A cell is a valid if the polygonal region does not contain walls inside. The cell is considered optimal if it is valid and cannot be divided into several valid cells.

The Breaking the Walls algorithm works in two passes. The first pass creates cells by traversing the wall and choosing the shortest possible step. It is achieved by traversing or skipping some walls by creating a new portal and enclosing the current cell. Since there is no previous knowledge, the decision is only based on the length of the walls. For that reason, only relatively short portals are created. The second pass is used to refine the cells. As the size and shape of the cells is known, the walls can be split or merged to create high-quality partitions. It is achieved by creating short portals relative to the cell boundary length.

2.3.4 Initial Partition

The initial partition is created during the first pass of the algorithm. Most formed cells will be valid during this pass but usually not optimal. The algorithm starts at an arbitrarily half-edge. At each step of the traversal it continues either to an adjacent half-edge or to the closest disjoint wall, in which case a portal is created to connect those edges. When an already visited wall is reached, a new cell is made with the correctly traversed path as its boundary. Traversal continues until all half-edges are part of a cell.

The most important part of this algorithm is the selection of the next wall traversed. Suppose that W is the current wall. According to [LCCO03], all adjacent edges should be considered to the wall W. But since the half-edge data structure is used, it is already known that the only possible edge to traverse would be the next one W.next, which will be called W_{adj}. Note that W_{adj} could be a wall or a portal defined previously. W_{adj} does not necessarily have to be the best option. It may be possible to create a new portal between W and the closest disjoint wall W_{cls}, so that the distance is shorter than W_{adj}. Furthermore, for a wall to be valid, it must be located in the subspace defined by the supporting lines of W and W_{adj}, visible in Figure 2.17.

Sometimes, a cell that encloses other geometry can be created when creating partitions using this method. Such a cell is not allowed and needs to be resolved. Ideally, such a cell will be resolved as the enclosed geometry is traversed, connecting itself to the surrounding cell, as shown in Figure 2.18(a). However, it is possible that such a connection will not be created, as in the case in Figure 2.18(b). In that case, this is dealt with separately by connecting it to the surrounding cell using the shortest portal.

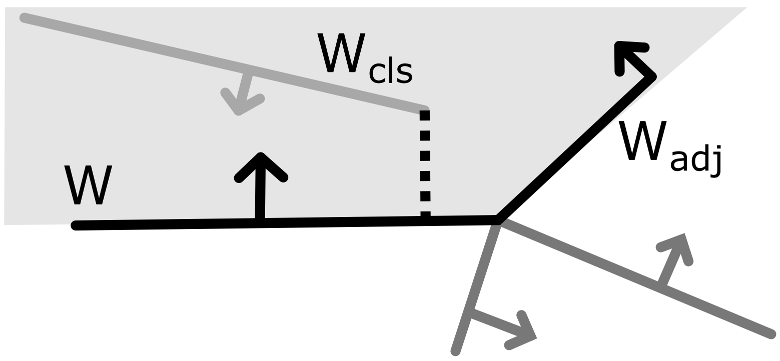


Figure 2.17: Traversal of an edge during Breaking the Walls algorithm of wall W. Wall W_{adj} is next traversed wall. Wall W_{cls} can potentially be connected by a portal (doted line). Only walls in grey subspace are tested considered to be potentially connected.

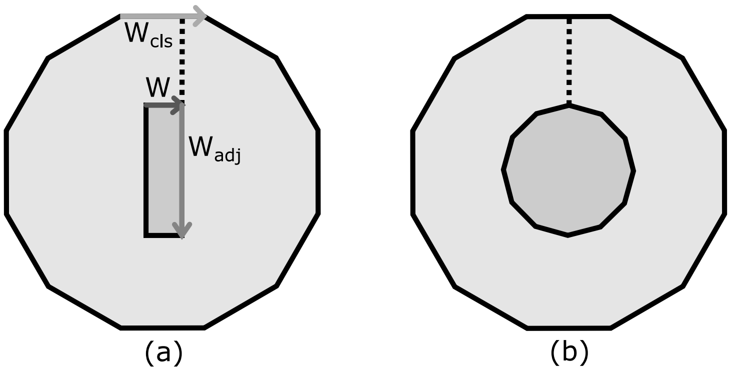


Figure 2.18: Enclosed geometry (a) gets connected by itself. Geometry (b) needs to be manually connected with the shortest portal.

2.3.5 Refinement

After the first pass, a Cells and Portals partition is created, which is probably not ideal. An ideal partition is a partition that contains only portals with a ratio of their length and the length of the cell boundary to which they belong less than the predefined value *alpha*. Using this definition, the algorithm controls which cells should be split or merged to create a good partition.

There are three types of cells, visible in Figure 2.19:

- Underestimated cell.
- Ideal cell.
- Overestimated cell.

The ideal cell is a cell that meets the criteria of ideal portals for all the portals it contains. Such a cell is correct, and it is not necessary to do anything with it.

The underestimated cell contains an invalid portal. This portal is too long and can be safely removed. In other words, cells on opposite sides of the portal can be merged.

The overestimated cell can be split into two cells with a valid portal. This type of split can occur only when two consecutive edges share an angle greater than 180°. To find the splitting portals, the algorithm traverses all edges of the cell and, for each right-turn edge, detects the shortest portal. The shortest portal is then picked and the cell is split using it. The two new split cells can be further split as they may not be optimal. The optimal cell does not have a potential portal that could split it.

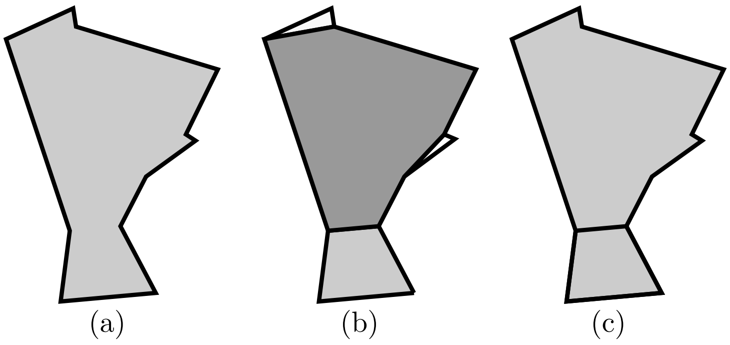


Figure 2.19: (a) Input for BW algorithm. (b) White cells are underestimated, light grey are the ideal and dark grey are overestimated. (c) Ideal partitions.

2.3.6 Breaking the walls

The algorithm is ideal for localised changes in the structure, since all computations depend on the topology of the half-edge data structure, allowing us to change only part of the geometry without recalculating the whole system. Using this knowledge, it is possible to easily add or remove walls from the current partition.

If an edge needs to be removed, the cells to which this edge is connected will be merged, the connected portals deleted, and the influenced edges will be added for processing. For adding, the edges are added as unclassified edges for processing, and in both cases the algorithm is run only for these changed edges.

Chapter 3

Implementation

This chapter first describes the input and output of the whole algorithm as well as its overall execution. Afterwards, the implementation of the „Interactive room detection“ algorithm in the Unity Engine is explained. The engine was used both for its functionality and for its usage in practice. Furthermore, the algorithm uses the Unity physics engine to detect intersecting walls instead of building the data structure as well as ray casting during specific parts of the algorithm.

3.1 Interactive room detection

The steps for the algorithm, which, from the input of walls, detects rooms and creates partitions using the Breaking the Walls algorithm, are explained in this section. This algorithm was implemented in the Unity engine in a way that allows it to be run in both the editor window and at runtime.

- **Initial data**

  The set of walls defined by their endpoints represents an input. These walls are added to the Unity scene as predefined objects. The user can create, delete, and move these objects. In addition, these objects can also be saved along with the scene. Since precise movement of walls can be complicated, users can use snapping of wall to other walls as well.

  For simplification of adding many walls, the algorithm can load and write walls from an input file. The file contains simple endpoints in the format *x1, y1, x2, y2* for a wall on each line. Users can start the algorithm in the editor using a specific script or button in the user interface inside the game.

- **Resulting data**

  The output of the algorithm are the individual objects that define the rooms. A room is defined as a polygon that surrounds a given room. The algorithm will visually create these rooms as walls and floors made of polygons by triangulation.

  The resulting partition will contain a potentially visible sets partition using the result of the Breaking the Walls algorithm. This partition splits rooms using portals that are used to simplify the rendering of the scene based on the visibility of the rooms.

- **Algorithm execution**

  The script has two launch options, either in the editor or at runtime.

  In the editor, the user can add objects directly to the editor and launch the script for the room creation algorithm afterwards, and the output of the algorithm will be inserted into the scene. The advantage of this method is that both input and output can be saved in the scene, offering acceleration of the game by preprocessing this information beforehand.

  At runtime, users can work the same as in the editor. They can add and edit wall objects and launch the algorithm for room creation. This approach can read input created before launching the game from the scene and data added during runtime, but the algorithm's output cannot be saved back into the scene. This approach is advantageous in games or applications that allow building or room creation and require room detection and partition features.

3.2 Game objects

The algorithm contains some predefined scripts and objects that can be added to the scene anytime. Most used objects can be compared to half-edge data such as vertices, edges, and faces. These objects consist of:

- **Wall Vertex** represents a vertex of normal half-edge data structure. It contains information on its position and reference to a single wall that uses it.
- **Wall Edit Object** is an object used to define the wall on the input. This object can only be created before the execution of an algorithm as it is not used by the algorithm after the creation of the half-edge data structure. It can only be used by the user to create, edit, or delete edges of the data structure.
- **Wall Half Edge Object** represents a wall, which is a part of half-edge structure, unlike the *Wall Edit Object*. These objects are created at runtime and are used to store two instances of *Half Edge* class that represent the front and back half-edge.
- **Room Mesh Generator** generates and manages a single room. Based on a single half-edge, this script detects an enclosed polygon from which it can generate a mesh using Unity procedural mesh. The mesh consists of rectangles as walls and triangulated mesh for the ground. In addition, it contains a function for the Breaking the Walls algorithm, since it is usually detected once per room.
- **Room Detector Controller** is an object used as an interface to the whole algorithm. It contains multiple functions that are used to start each

phase of the whole algorithm, such as half-edge data structure generation or Breaking the Walls passes. In addition, it stores information about all objects used, which this script can create and delete.

On the other hand, the *HalfEdge* class represents a single half-edge of a half-edge data structure. The data it contains include:

- **Origin:** instance of Wall Vertex representing an end of the half-edge.
- **Sibling:** instance of Half Edge representing half-edge on the opposite side of the wall.
- **NextHalfEdge:** instance of HalfEdge in a counterclockwise direction.
- **PreviousHalfEdge:** instance of HalfEdge in a clockwise direction.
- **Room:** object representing a room this HalfEdge defines.

Furthermore, it contains a function to set the *NextHalfEdge*, which tests whether the HalfEdge on the input should be added as *NextHalfEdge* by testing the angles between the current *NextHalfEdge* and the *PreviousHalfEdge* of the tested one, visible in Figure 3.1.

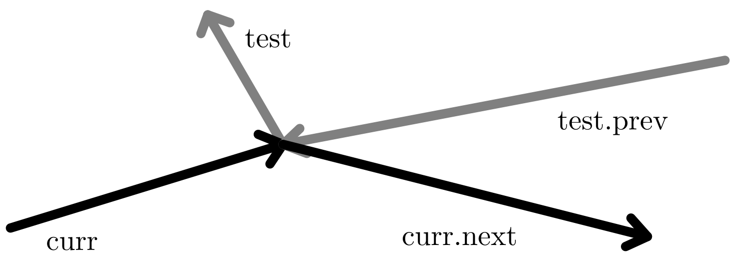


Figure 3.1: A HalfEdge *test* is can be added as next half-edge of current one *curr* by testing it against both *test.prev* and *curr.next*.

3.3 Steps of execution

The algorithm performs multiple steps to first separate the input walls into non-crossing lines during which it creates a half-edge structure over the data. Afterwards, it generates the partition using the Breaking the Walls algorithm. The steps are described in this section and later in the implementation section in detail. Note that the algorithm execution may not return the same output even for the same input walls. First off, the algorithm is order-dependent, as the half-edge data structure generation takes input walls iteratively. Additionally, there may be some cases, such as when multiple walls intersect at one point, where the algorithm may process them at different order each time, slightly changing the output partition.

3.3.1 Splitting the walls

The first step is to detect the intersection of the walls and to divide them into non-crossing lines. During this step, the half-edge structure is slowly being built, as it is already known which walls are neighbouring each other.

During the whole splitting a simple rule is held: „**The wall is processed, when it is split into correct wall segments and each segment has correctly set half-edges**“. If this rule is held, after a wall is processed, it does not need to test against it in any of the next iterations, as it is already known that it cannot be improved in any way.

At first, all input walls are inserted into a queue. This queue contains all walls that need to be tested at any time. These walls are processed until the queue is empty. Every wall is tested to determine which other walls it intersects. It is tested not only for a full intersection but also for whether the walls only touch each other, as these walls are also used to split the current one. It is achieved by using the Unity physics engine by looking for all objects in a specific layer inside an orientated bounding box with the sizes of a wall (it is made it a little wider in case of mistakes). From this point on, the walls are considered as line segments.

For every intersection the walls have, they are split using the same method. First, every intersection point and its distance along both walls is calculated. Then these walls are sorted according to their intersection distance. By doing this, every intersection of our current wall can be calculated once instead of splitting it into two and repeatedly testing them for intersections. During the next step, new vertices are detected and created to split the wall. For the detection, two new arrays are created:

- Vertices.
- Skip index.

These arrays can be created at once by iterating the array of intersected walls. The *Vertices* array contains instances of WallVertex for every unique intersection point. It needs to be done like this, since multiple walls that intersect simultaneously are supported. For every vertex, an already existing vertex that could be used is detected first, which could be a vertex from both the currently tested wall and any of the intersected ones. If none exists, a new one is created.

The skip index array detects which walls intersect at which intersection point. The idea behind it is pretty simple. For a skipIndex at position *k*, it is known that the walls from *intersectedWalls[k]* up to, but excluding *intersectedWalls[k+1]* all intersect at *vertices[k]*. Knowing this, the intersection can be tested quickly and the already created vertex can be used without trying to make a new one.

To create these arrays, iteration through the sorted intersected walls is needed. The pseudocode for the algorithm can be seen in Figure 3.2

At the beginning, the starting vertex is inserted into the vertices array and 0 for skipIndex. The distance along the wall for the correctly tested wall, initially 0, is also remembered. Every intersected wall is tested to determine whether it is further than the previous. The algorithm ends with the vertex at the end of the current wall and the number of walls as the last *skipIndex*. The resulting arrays can be seen in Figure 3.3. It can be seen that there are two vertices at the end of the line. This is caused by the way the segments

```
    skipIndex, vertices = []
    currentT = 0

    skipIndex.Insert(0)
    vertices.Insert(wall.startPosition)

    for i = 0 to intersectedWalls.Length - 1 do:
        if currentT < intersectedWalls[i].distanceT:
            skipIndex.Insert(i)
            vertices.Add(intersectedWalls[i].intersectPosition)

            currentT = intersectedWalls[i].distanceT

    skipIndex.Insert(intersectedWalls.Length)
    vertices.Insert(wall.endPosition)
```

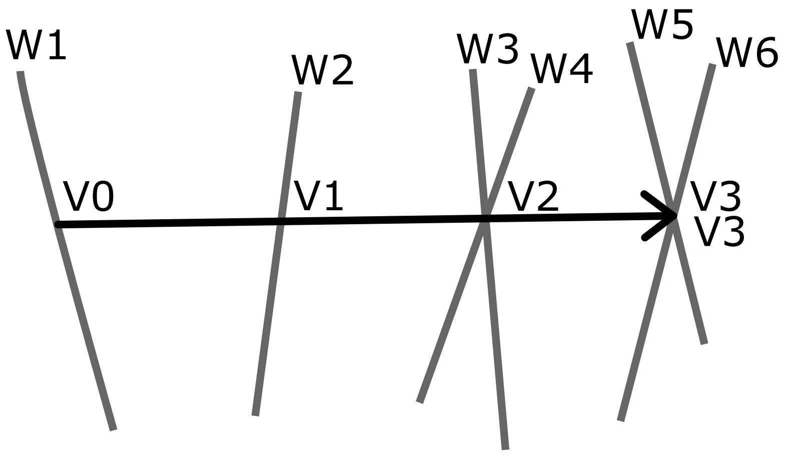
Figure 3.2: Pseudocode for generation of SkipIndex and Vertices arrays.

are programmed. It is necessary to keep both *skipIndex* and *vertices* arrays the same length since, at each vertex, the intersected wall is processed and the next line segment is created. Because of this, there can be two vertices in the same position at the end. This can be solved by detecting whether there is an intersection at the end. If there is, the ending vertex is inserted twice. It is inserted twice, since the ending vertex can be a previously created vertex that some other wall may share. This is done mainly because the last vertex should be used only to finish the line segments and should not have any intersections.

After creating the structure, the segments can be created by simply iterating over them. The iteration can be seen in pseudocode in Figure 3.4.

The two tested walls are divided into four segments (previous, next, left, right), two for each wall. The previous and next segments are two line segments of the wall that is being tested. The left and right walls are two line segments in which the intersected line was split. Any of these parameters can be undefined, but at any time, at least two must be valid objects. Otherwise, there cannot be an intersection. Additionally, it is assumed that the left[right] wall is to the left[right] of the tested wall in its direction. When they are created, it is not known if this rule is held, so it is necessary to test and eventually swap them to keep this rule.

This approach is not ideal, as walls that only touch the current wall will also be split. Walls with a length close to zero are not created, but the code will still make a copy of a wall that could be used. It can be easily detected whether there is a need to create a copy of a wall depending on the distance of the intersection point along the line. If the intersection distance is 0 or the full length of the line segment, one of the split walls will not be created, and the other can use the original wall instead.



walls = [W1, W2, W3, W4, W5, W6]
skipIndex = [0, 1, 2, 4, 6]
vertices = [v0, v1, v2, v3, v3]

Figure 3.3: Example of the Vertices and SkipIndex arrays for a multiple intersection.

```
nextSegment, prevSegment,
leftSegment, rightSegment = null;

for v = 0 to length(vertices) - 2 do
    nextSegment = Wall(vertices[v], vertices[v+1])

    i = skipIndex[v]
    j = skipIndex[v+1]

    for e = i to j - 1 do
        leftSegment = Wall(walls[e].start, vertices[k])
        rightSegment = Wall(vertices[k], walls[e].end)

        SetHalfEdges(vertices[k],
            prevSegment, leftSegment,
            nextSegment, rightSegment)

    previousSegment = nextSegment
```

Figure 3.4: Pseudocode for the splitting of line into line segments.

3.3.2 Calculating half-edges

Now that the wall is correctly split, the neighbouring half-edges can be connected. The primary function that manages the computation of half-edges is seen in the pseudocode 3.4 called *SetHalfEdges*.

This function holds an array of the input *edges* in counter-clockwise order around the input *vertex* as well as another array that defines whether that specific edge starts or ends in the *vertex*, saved in array *swapped*. First, it must be detected whether the wall on the input is valid and to see if they start in the *vertex*. This can be tested using the simple comparison seen in Figure 3.5.

```
size = 0
swapped = bool[4]
edges = Wall[4]

// for each input Wall segment
if segment is not null then
    swapper[size] = segment.startVertex != vertex
    edges[size] = segment
    size++
```

Figure 3.5: Pseudocode for validation and preparation of line segments for the half-edge connecting.

Two comparisons can be simplified because the previous segment never starts in the *vertex*, while the next segment always starts there.

There was a need to test whether the edge was swapped, since it is not known whether the front or back half-edge of the wall should be tested against. Now that it is known how each wall segment is rotated, it can be decided which half-edges to test against each other. Two adjacent walls are always tested, as shown in Figure 3.6.

Testing of half-edges can be split into four separate categories depending on which half-edge from every wall is used. This test can be seen in pseudocode 3.7. The front half-edge is one to the right from the start of the wall.

As described previously, the *setNext* function also tests whether the other half-edge should be added next to the current one. For that, it is necessary to test not only against the current next half-edge, but also against the previous half-edge of the second one, visible in Figure 3.1. This is tested to ensure that the half-edge prediction does not worsen and can only be improved in the next iterations. The test of whether the half edge should be changed is calculated as the difference in angles between the two edges tested using the *atan2* function.

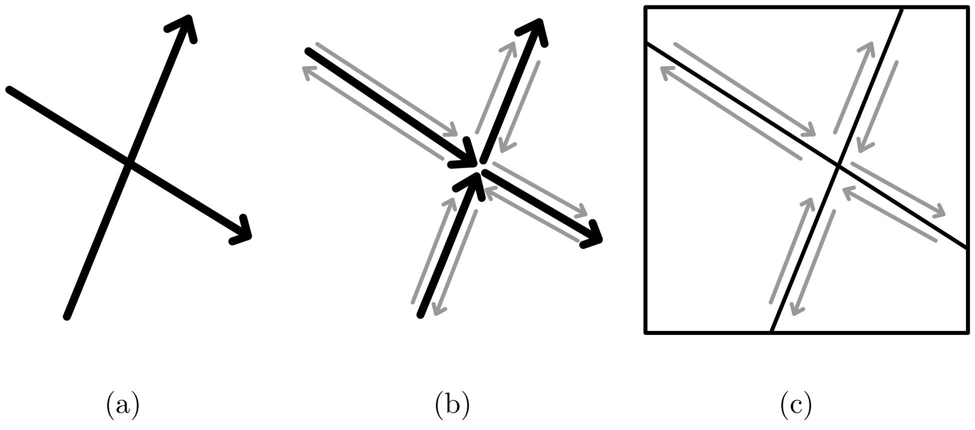


Figure 3.6: (a) Walls to intersect. (b) Intersected walls with half-edges. (c) Groups of half-edges to be tested.

```
    for i = 0 to size - 1 do
        w1 = i
        w2 = (i + 1) % size

        if swapped[w1] and swapped[w2] then
            setNext(edges[w1].frontHE, edges[w2].backHE)

        else if not swapped[w1] and swapped[w2] then
            setNext(edges[w1].backHE, edges[w2].backHE)

        else if swapped[w1] and not swapped[w2] then
            setNext(edges[w1].frontHE, edges[w2].frontHE)

        else // (not swapped[w1] and not swapped[w2])
            setNext(edges[w1].backHE, edges[w2].frontHE)
```

Figure 3.7: Pseudocode for half-edge connecting of two adjacent line segments.

3.3.3 Generating rooms

After the scene was successfully split into segments and the whole half-edge data structure was computed, the rooms can be easily detected by iterating through the half-edge structure. To detect every room, every non-previously processed half-edge is iterated over. From such a half-edge, the room can be detected using the algorithm in Figure 2.6, taking the input half-edge as the starting edge. Every edge visited is set as processed. When the original half-edge is reached again, all the traversed vertices form a room.

During execution of this algorithm, it is also needed to test when the polygon is in clockwise or counterclockwise order, since that would determine whether the room represents the inner or outer surface of the room. Although the outer surface is not typically considered a room, this paper assumes that it is one, as it is generated in the same manner. For that, the Shoelace algorithm [O'R98, p. 21] is used with pseudocode in Figure 3.8.

```
function IsCCW(polygon):
    area = 0
    for i = 0 to len(polygon) - 1 do:
        x1, y1 = polygon[i]
        x2, y2 = polygon[(i + 1) mod len(polygon)]
        area += (x2 - x1) * (y2 + y1)

    return area > 0
```

Figure 3.8: Pseudocode for the Shoelace algorithm for testing CW or CCW direction of a polygon.

The result is quite important, since each room works differently depending on whether it is an outer or inner room.

- **Outer room** has no floor, meaning it shouldn't be triangulated. Furthermore, the room that closely and fully encloses this room needs to be detected, as that room will be a candidate for a portal connection during the Breaking the Walls algorithm.
- **Inner room** has a volume, so the triangulation should be calculated correctly. The surrounding room does not need to be calculated since this room can have only rooms inside of itself. Those inside rooms must be the outer rooms, which are detected by themselves.

To detect the surrounding room, linecast is used to find intersections with every wall starting at the boundary of the tested room, as it is not necessary to test rooms lying inside of the tested room. For each intersected wall along the line, there are two rooms. The first one lying closer to the point from which the linecast started, the *closer room*, and the second lying on the opposite side of the wall, the *further room*. The searched room is the one of which the player is inside of, meaning a room that is found as *closer room* without finding it as *further room* before, as seen in Figure 3.9. If such a room is found, it is known that it fully encloses the currently tested room, as well as that it is the closest fully enclosed wall. The outer rooms then remember their surrounding rooms, and the inner rooms remember all the rooms that are closely inside of them.

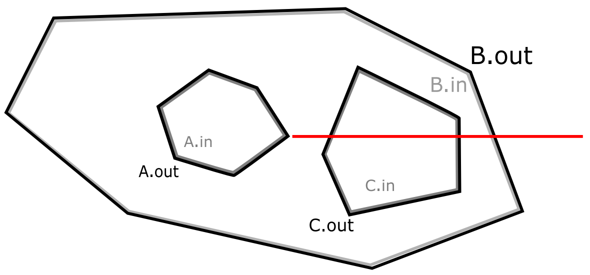


Figure 3.9: Detection of surrounding room for room A.out. Each room has inner and outer part of itself. First inner room intersected room without having the outer room being intersected is room B.

3.3.4 Mesh generation

After room generation, it is possible to also generate a mesh for the rooms. The mesh is made up of two parts: walls and floor. Walls can be generated by iterating through the vertices of the room and generating them as two vertical triangles between two adjacent vertices. Floor, on the other hand, needs a triangulation algorithm. For that purpose a simple ear-cutting triangulation algorithm is used.

The ear cutting algorithm works by iterating through the vertices, while at every step, a triangle is created from three adjacent vertices. Then, it is tested whether any other vertex in the room lies inside this triangle. If not, it can be called it a valid triangle and the middle vertex is removed from the tested vertices. If a vertex is located inside this triangle, another triangle needs to be tested. This continues until there are only three vertices, which create the last triangle of the triangulation.

A slight difference between the standard ear-cutting algorithm is how the point inside the triangle is detected. Usually, points that lie on the border of the triangle are considered inside. For this algorithm, it is necessary to make this rule a little more complex because our room allows walls without width. Because of this, there may be a vertex on the boundary of the triangle but theoretically on the other side of the wall. For this reason, the vertices in one of the vertices of a triangle are not considered inside but outside and are also accepted. This simple change allows for the triangulation of rooms with zero-width walls.

3.4 Breaking the Walls

Breaking the Walls algorithm consists of two passes. While the first pass only creates locally good portals, the second pass validates the result by both deleting the non-ideal portals and creating the ideal ones.

Given a pair of half-edges on the input, a pair of points, where each of them lies on a separate half-edge, is searched for to create a shortest portal. The process can be simplified by finding the distance between the endpoint of one line and the other line, which results in four cases for each pair of walls.

In addition, some constraints must be specified on these points to ensure that a valid portal can connect them. The valid portal is a portal that does not cross any other wall. These constraints consist of the following.

- Distance has to be nonzero, since zero length portal cannot be created.
- The other connection point for a half-edge needs to be between the half-edge wings (previous and next half-edge). This can be ensured by testing the orientated angle between each of the wings, visible in Figure 3.10. Furthermore, whether the closest point lies between endpoints is also tested, as it is necessary to ensure that the other connection point lies on the same side as the half-edge in the case of a wall with zero width. This step is performed for both the half-edges and their respective connecting points.

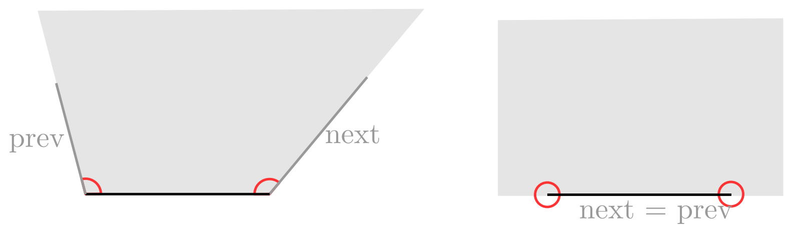


Figure 3.10: Testing the position of connecting points for a half-edge. On point lies of black line segment, the other point can only lie in the gray area.

- Unfortunately, this is not enough to guarantee the validity of the portal, as the portal can still cross some other half-edge. The example of such a problem can be seen in Figure 3.11. As a result, the last step is using a linecast. Linecast is performed between the pair of closest point as the last step to ensure there is no other object between the points.

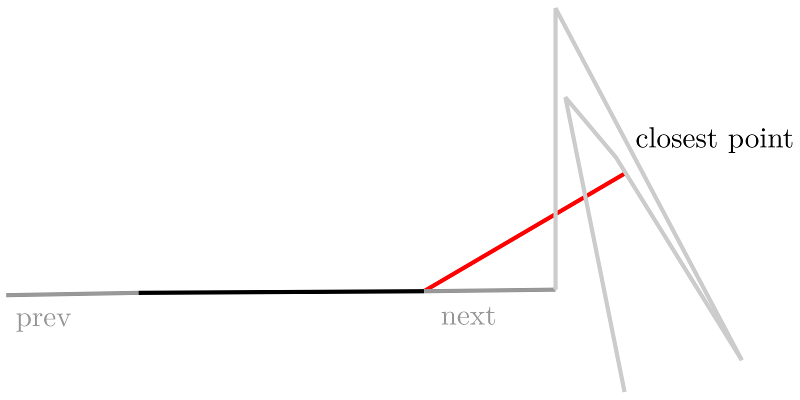


Figure 3.11: An example of input case, where portal may be incorrectly constructed without using linecast.

In general, the requirements can be summarised as a function in Figure 3.12.

```
function isValid(halfEdgeA, pointA, halfEdgeB, pointB):
    return Dist(pointA, pointB) > 0 AND
           PointBetweenWings(halfEdgeA, pointB) AND
           PointBetweenWings(halfEdgeB, pointA) AND
           NOT LineCast(pointA, pointB);
```

Figure 3.12: Pseudocode for testing the valid portal between half-edges A,B. Points A, B are the closest points on each half-edge.

3.4.1 First pass

The first pass of the Breaking the Walls algorithm creates a simple partition. This partition is created by detecting the best portal locally. At each step a half-edge is selected and tested against every other half-edge while looking for a single shortest valid portal. If such a portal exists, it is created and, if not, next half-edge is selected and the process is repeated. The end of this step occurs when no new portal can be created.

During the algorithm, it is not necessary to test every half-edge, as most of them may not be connected with valid portals. Since portals are generated per room, a pair of half-edges that can be connected must lie in the same room. That may not be enough, as during this step there are also room areas that are split into multiple parts, such as the rooms in Figure 3.9. A valid portal can be created between these rooms. In general, tested rooms can be detected based on whether the original room is an outer or inner room.

- **Inner room**
  - Itself
  - Inside rooms
- **Outer room**
  - Outside room
  - Inside rooms of the outside room (which contains itself)

When the tested rooms are detected, one half-edge from the original room is selected and the search for the shortest valid portal is started. Another constraint on a valid portal is also that the length of the portal must be shorter than the length of the next half-edge, which must be updated every time the original half-edge is changed.

When a portal is created, the current room splits into one or two new rooms, which needs to be tested again for a new potential portal. As mentioned above, the first pass ends when no room can create a valid portal. The whole pseudocode of the first pass can be seen in Figure 3.13.

```
function FirstPass(rooms):
    Queue roomsQueue = new Queue(rooms)
    while (roomsQueue.count > 0):
        room = roomsQueue.dequeue()

        testedRooms = room.GetRoomsToTest()

        currentHE = room.halfEdge
        do:
            distance = length(currentHE.next)
            bestPortal = null

            foreach otherRoom in testedRooms:
                testedHE = otherRoom.halfEdge
                do:
                    portal =
                       FindClosestPortal(currentHE, testedHE)
                    if portal.distance < bestPortal.distance:
                        bestPortal = portal

                    testedHE = testedHE.next
                while (testedHE != otherRoom.halfEdge)

            if bestPortal != null:
                room1, room2 = CreatePortal(bestPortal)
                roomsQueue.enqueue(room1)
                roomsQueue.enqueue(room2)

            currentHE = currentHE.next
        while (currentHE != room.halfEdge)
```

Figure 3.13: Pseudocode for the first pass of Breaking the Walls algorithm. The first pass generates partition by creating portals between two valid half-edges.

After completion of the first pass, there may still be some outer rooms that are not connected to any other room, such as the room in Figure 2.17. Every outer room needs to be connected, so that the validation pass does not need to connect to multiple rooms as in the case of the first pass. For that purpose, the first pass is executed for outer rooms again, while testing only against other rooms, not itself, to force a connection while allowing a portal of any size to be created. As before, the algorithm continues until there is no new portal possible, which in this case is when all outer rooms are connected and only inner rooms are left.

3.4.2 Second pass

Second pass or validation pass is a refinement of the first initial pass. Compared to the local approach of the first pass, which is looking for the best portal for a half-edge, the second pass looks for the best portal for the whole room. The same as in the first pass, the new portal is detected per room, but this time only the chosen room is tested, as no valid portal can be created with any of the other rooms.

First, the portals created before are tested, whether they are good enough for the partition, which is determined by comparing the length of the portal with the length of the whole room. The portal is valid if:

$$\frac{\text{Room length}}{\text{Portal length}} > \text{Portal ratio}$$

With a smaller portal ratio, more portals and denser partitions will be generated. If a portal is deemed invalid, it is deleted from the structure, the rooms it splits are connected, and the new room is added again for validation. The pseudocode for this step can be seen in Figure 3.14.

```
function SecondPassDelete(room):

    portals = room.GetPortals()
    roomLength = room.GetLength();

    foreach portal in portals:
        if roomLength / portal.length < portalRatio:
            mergedRoom = portal.Delete()
            AddToTestingQueue(mergedRoom)
            return
```

Figure 3.14: Pseudocode for the first part of the second pass of Breaking the Walls algorithm, which removes invalid portals.

If there are no invalid portals, new valid portals can possibly be added. Such a portal needs to be found in a way similar to the first pass, but compared to the first pass, there are some differences. First, only two consecutive half-edges that have a right turn are tested, or in other words, share an orientated angle larger than 180 degrees. Another difference is that the algorithm does not stop when a valid portal is found, but continues until it tests every pair of half-edges in a single room. Then only the best portal is accepted. If there is no such portal, the algorithm ends. The pseudocode for the detection of new portals can be seen in Figure 3.15.

Although the algorithm will find the best portal for a single room, it may still not be a valid one, as it could break the length requirement. For that reason, it is necessary to test every portal for its validity before accepting it as

```
function SecondPassAdd(room):
    currentHE = room.halfEdge
    bestPortal = null

    do:
        if hasRightTurn(currentHE, currentHE.next):
            testedHE = currentHE

            do:
                portal =
                    FindClosestPortal(currentHE, testedHE)
                if portal.distance < bestPortal.distance:
                    bestPortal = portal

                portal =
                    FindClosestPortal(currentHE.next, testedHE)
                if portal.distance < bestPortal.distance:
                    bestPortal = portal

                testedHE = testedHE.next
            while (testedHE = currentHE)

        currentHE = currentHE.next
    while (currentHE = room.halfEdge)

    if bestPortal != null:
        room1, room2 = CreatePortal(bestPortal)
        roomsQueue.enqueue(room1)
        roomsQueue.enqueue(room2)
```

Figure 3.15: Pseudocode for the second part of the second pass of Breaking the Walls algorithm, which generates the best valid portal in the whole room.

a possible candidate. This can be simply tested by calculating the lengths of the two rooms to which this room would be split by such a portal. However, calculating the lengths for each portal can be ineffective since the same values would be calculated every time. It is useful to precompute the portal distance at each half-edge to quickly recompute the new lengths. The idea behind this computation can be seen in Figure 3.16. If the portal in the figure was being created, the new room length can be simply calculated by getting the distance between half-edges 1 and 3 and adding to it the portal length and the distance along the half-edges of 1 and 4, which is known from the portal calculation. Then if the new portal has a valid length, it can be considered a candidate for a newly created portal. Obviously, the new portal must be valid for both created rooms, so the other room is also tested.

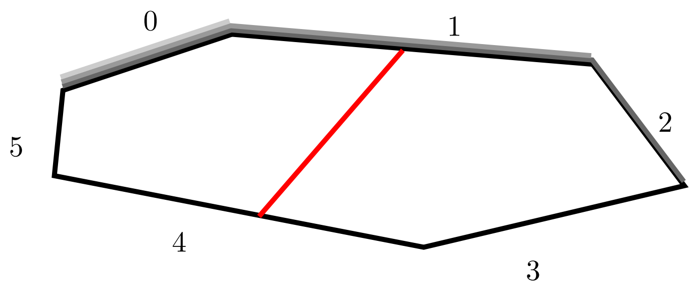


Figure 3.16: Computation of length of a split room. The algorithm calculates distance of each point from the selected point (0). Then the length of split room can be calculated as: (distance of 3 - distance of 1) + distance of portal along lines 1 and 4 + portal length. The gray lines represent distances for each vertex.

3.5 Potentially visible sets

Final step of the algorithm is the generation of potentially visible sets. An approximate algorithm based on line casting between portals is used to determine their visibility. A graph is built over the partition by detecting which rooms are visible from the selected room. The idea of the PVS algorithm is to recursively test the visibility of portals. Whether a portal along the path is visible, the room on the other side of the portals is added to the PVS of the tested room, and all the portals in the room added to be tested against. The pseudocode can be seen in Figure 3.17.

```
function GenerateRoomPVS(room):
    PVS[room].Insert(room)

    Queue portalQueue = new Queue()
    foreach p in room.portals:
        portalQueue.enque([p, p])

    while ( roomQueue.count > 0 ):

        portalFrom, portalTo = portalQueue.deque()
        if portalTo.toRoom NOT IN PVS[room] AND
           TestVisibility(portalFrom, portalTo):

            PVS[room].Insert(portal.toRoom)

            foreach p in portalTo.toRoom:
                portalQueue.enque[portalFrom, p]
```

Figure 3.17: Pseudocode for the recursive generation of PVS for a single room.

The visibility test is a necessary part of the function. The result indicates whether there are any obstacles, walls in our example, in the way. In our case, a simple approximate line cast from points lying on one portal to point lying on the second portals is performed casting a predefined amount of rays along the portals at uniform distance.

After the generation of the PVS, the resulting visible rooms can be easily given for every room. To detect in which room user lies, a ray cast is executed, which returns the room. The PVS is then returned as a list of rooms that the user can potentially see.

3.6 Usage for 3D

This algorithm is intended for a 2D scene made from lines, but it would be ideal if there were a way to also use it for a 3D scene in such a way that the 3D scene could be transformed into 2D, a PVS partition being generated and mapped back to the 3D scene.

The first necessary step would be to project a 3D model into a 2D one. Techniques like orthographic projection from above would probably be most useful, as the walls for PVS generation are most likely vertical. This may bring it's problems, as other problems as if the 3D mesh contains other objects, such as furniture, since those objects would also be projected. It is necessary to preprocess them, so they do not show in the model either by manually hiding them or possibly distinguishing them from walls by their space, such as the furniture would be shorter or overall smaller than the architecture. A different approach, the one this thesis used for preprocessing the models, is to intersect the models with a horizontal plane. By intersecting the model at the correct height, ideally where there is no furniture, the correct floor plan can be quickly generated. While it is a simple and easily controllable approach, as it may contain less unnecessary objects, it is not ideal for dense scenes or scenes with different room heights, as a more complex objects for intersecting would be necessary.

All of the mentioned projection techniques are still not perfect, as the projection may leave errors such as duplicated or parallel edges, which are necessary to be fixed before generating the PVS partition. When the valid lines are created, the PVS partition can be generated using this thesis algorithm.

The last step is to assign the separate PVS cells to the geometry of the 3D mesh. A way to do it may be to use the orthogonal projection from the first step. As PVS cells are considered 2D polygons, they can be tested to see which cell a piece of geometry belongs after it has been projected in 2D. By simplifying this 3D problem into 2D, the cell in which the geometry belongs may be detected noticeably faster than in 3D. Each piece of geometry will then know which cell it belongs to and during the rendering it may be decided not to be rendered based on PVS of the currently entered cell.

This process may not be the best, as there may be problems with geometry at some steps. The process would largely depend on the 3D model, but the idea may, with some additional refinement, work.

Chapter 4 Results

The algorithm was tested on six scenes. Three of those scenes were extracted from 3D models, namely, Vienna, Pompeii, and Soda. As the algorithm is capable of loading wall from file line by line, the floor plan was needed to be extracted first. This extraction was performed by bisecting the models in Blender, a 3D modelling software. The result of the bisect were lines representing the floor plan. All of these scenes still had multiple inaccuracies after extraction, such as parallel intersecting walls, so algorithms such as *Merge by distance*, which merges close vertices or *Decimate* simplifying geometry were used. The scene FEL was created by hand based on the first floor of the Czech Technical University in Prague, situated at Technicka 2. Lastly, scenes Cross and Houses were generated using a simple Python script.

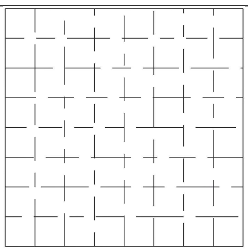
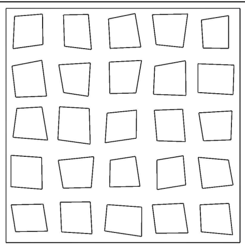
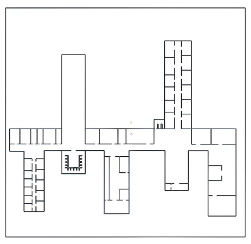
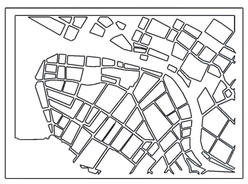
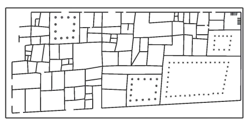
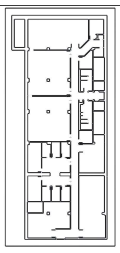
 Cross	 Houses	 FEL
 Vienna	 Pompeii	 Soda Hall

Table 4.1: Preview of tested scenes.

Each scene was tested with multiple values with different algorithm parameters. The common parameter for every test is the width of the wall as 0.01, as this parameter can slightly change the partition. These values and measured results can be seen split in the following tables based on the phase of the algorithm described in previous chapter. The input images for every scene can be seen in the table 4.1. This chapter will contain only images of three selected scenes (FEL, Vienna, Soda). All scenes can be seen in the appendix A, where all values and images can be found split based on the scenes compared to the values divided by algorithm phases described here.

4.1 Half-edge structure

First phase of algorithm consists of splitting the walls into segments and creating the half-edge data structure. The measured values in table 4.2 consist of:

- **Walls:** The amount of walls in the input.
- **Edges:** The amount of wall segments created, each representing two half-edges.
- **Rooms:** The amount of rooms created.
- **Intersections:** The number of times intersections were calculated for a single wall.
- **Time:** The calculation time of the half-edge structure.

The calculated half-edge data structure can be seen in table 4.3. There are two types of scenes, the first type that was created by hand (FEL) or extracted from a 3D model (Vienna, Pompeii, Soda). These scenes contain barely any wall intersections, as the model was already separated into non-intersecting lines. For that reason, the number of edges is close to the number of input walls. On the other hand, scene like Cross was procedural generated, so it contains many intersecting lines, meaning many of the input wall needed to be split into segments, hence the larger amount of edges in the data structure. The same applies to the intersection value, since this value must be bigger than the input wall value, since each wall must be tested for intersection at least once, but must be lower than the number of edges, since every edge can only be tested a maximum of once. The computation time does not depend as much on the input walls as on the amount of edges or intersections created.

Scene	Walls	Edges	Rooms	Intersections	Time (s)
Cross	97	163	48	136	0.0799
Houses	104	104	52	104	0.05927
FEL	299	300	11	300	0.1899
Vienna	658	660	186	659	0.437
Pompeii	755	758	229	757	0.4783
Soda Hall	759	777	142	768	0.499

Table 4.2: Amount of geometry and computation time for the half-edge structure generation.

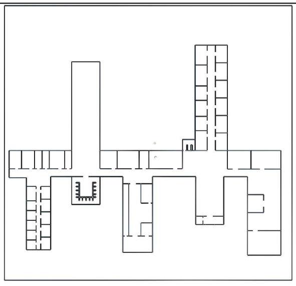
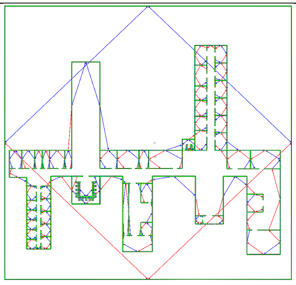
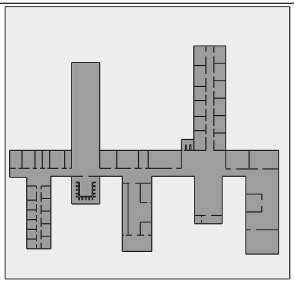
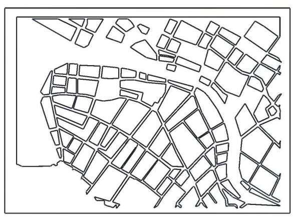
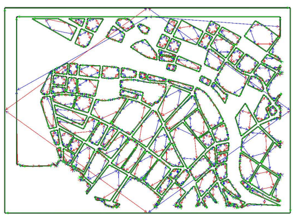
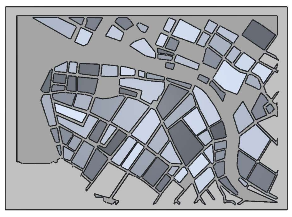
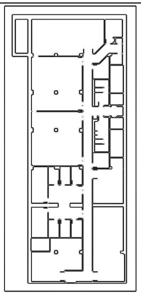
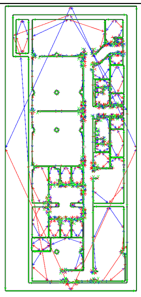
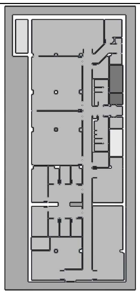
Input walls	Calculated half-edges	Resulting rooms
		
	FEL	
		
	Vienna	
		
	Soda Hall	

Table 4.3: Resulting half-edge partitions and resulting rooms. Red and blue arrows represent next half-edge of front and back half-edge for each wall. Green represents the direction of a wall.

4.2 Breaking the Walls algorithm

Next step is the Breaking the Walls algorithm. As stated above, the algorithm has two passes. The first pass generates a simple partition by creating portals, which the second pass validates by both deleting invalid portals and creating new portals into a final one. The portal ratio for the validation pass was 3. The values measured in the table 4.4 are:

- **P1 Ptl+:** Number of portals created in the first pass of the algorithm.
- **Time P1 (s):** The calculation time of the first pass.
- **P2 Ptl-:** Number of portals deleted in the second pass of the algorithm.
- **P2 Ptl+:** Number of portals created in the second pass of the algorithm.
- **Time P2 (s):** The calculation time of the second pass.

The first pass usually creates a large number of portals. As seen in the table, the large amount, usually around 60-70%, of those portals stays in the partition. It should be noted that those portal created in the first pass may not be the best ones, as they are only good enough to be in the partition based on the portal ratio selected. These portals were created as the best of the local area, meaning the best from one half edge. If the algorithm were looking for a globally best portal, meaning the best between every pair of half-edges, the partition would most likely look different, but the generation would take noticeably longer time. Although exact values of the optimal partition were not measured, as the algorithm would need a change, the time would change from $O(n^2)$ to $O(n^3)$. Although the validation pass also has the complexity of $O(n^3)$, the n in this case is considerably lower, due to the first pass splitting large rooms into smaller ones.

Scene	P1 Ptl+	Time P1 (s)	P2 Ptl-	P2 Ptl-	Time P2 (s)
Cross	135	0.2859	52	26	0.06401
Houses	123	0.1689	18	7	0.01673
FEL	492	0.5234	51	83	0.1546
Vienna	921	2.2373	316	121	0.33531
Pompeii	397	1.5192	162	82	0.8029
Soda Hall	481	1.3293	137	142	0.40878

Table 4.4: Created and deleted portals for each Breaking the Walls algorithm phase as well as their computation times.

The generated partition of both passes can be seen in the table 4.5. The partition for different values of the ratio of the portal to room size, used in the validation, can be seen in the table 4.6.

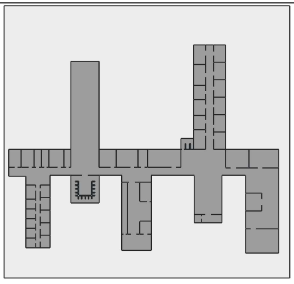
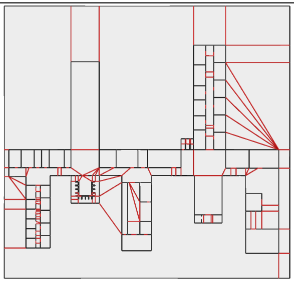
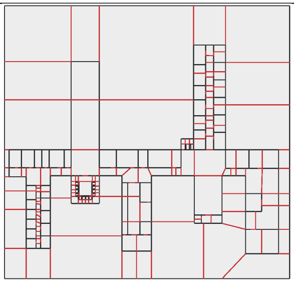
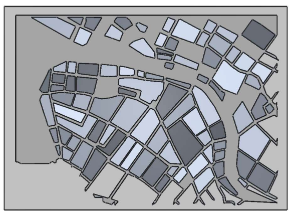
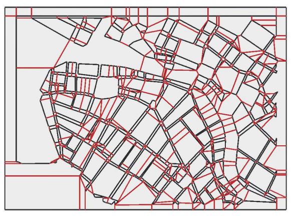
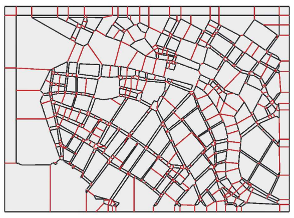
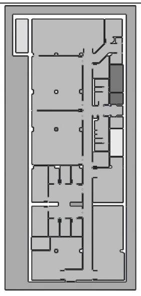
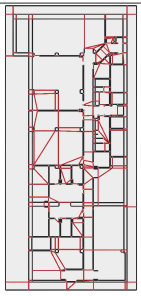
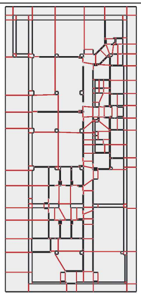
Rooms	First BtW pass	Second BtW pass
		
	FEL	
		
	Vienna	
		
	Soda Hall	

Table 4.5: Resulting Breaking the Walls partition. Second pass with portal ratio of 3.

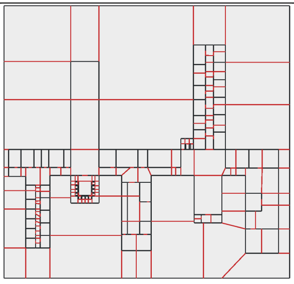
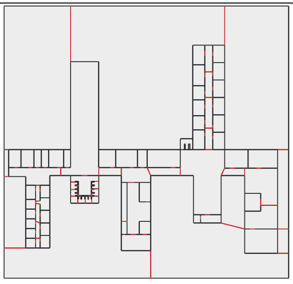
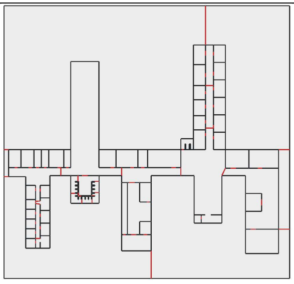
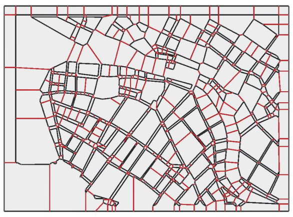
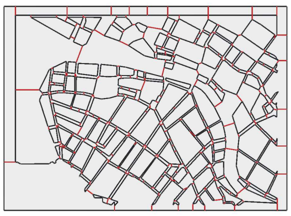
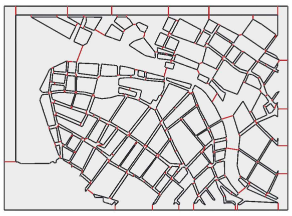
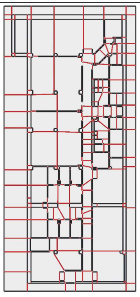
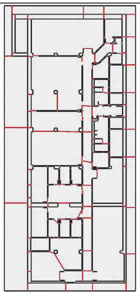
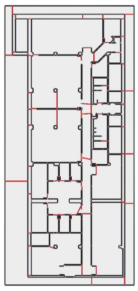
		
Ratio 3	Ratio 6	Ratio 10
	FEL	
		
Ratio 3	Ratio 6	Ratio 10
	Vienna	
		
Ratio 3	Ratio 6	Ratio 10
	Soda Hall	

Table 4.6: Breaking the Walls partition with different portal ratios.

4.3 Potentially visible set

The last step of the algorithm is the generation of potentially visible sets. As PVS during run-time only returns precomputed values, only on the generation of PVS itself is tested and measured. Since PVS generation is approximate, being sampled by a number of rays, the times of generation closely depend on the number of rays. As seen in Figure 4.1. The times were tested for amounts of 10, 30, 60, 100 and 250 rays.

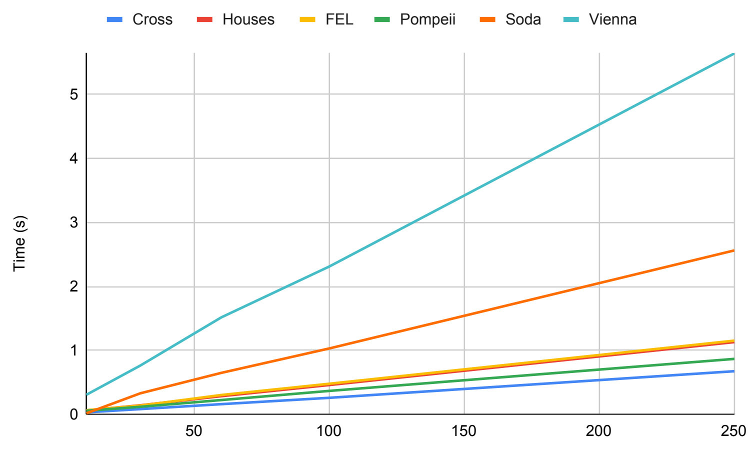


Figure 4.1: PVS generation time depending on amount of rays. The tested values were 10, 30, 60, 100, 250.

From the figure it can be noticed that the time is nearly linear to the amount of tested rays. It can also be noticed that scenes differ in how quickly the time rises, such as scenes Vienna and Cross. The scene Vienna rises considerably faster than the scene Cross. The main reason for this is the number of rooms and portals in the resulting partition. The Cross scene contains only 109 portals compared to the 726 portals of the Vienna scene. Since there are more portals, the visibility test will be tested many more times, resulting in an increase in computation time. In addition, there are other reasons, such as how the scene is open. An example of this are scenes Cross and Pompeii. These scenes have similar time, even if the Cross scene has 109 portals and Pompeii 317. Although one could assume that the scene Pompeii would take longer to generate PVS, the times are very close. The reason lies in the layout of the scenes. The scene Pompeii is mostly closed, meaning that most of the rooms will have very small PVS that does not need to be tested for very long. On the other hand, the scene Cross is very open, so more visible rooms need to be tested. In general, both the number of portals and the layout of the scene are important factors that determine the computation speed of PVS generation.

The PVS quality is determined by how much area is visible from a selected cell compared to the whole scene. The less visible area that is detected, the smaller part of the scene must be rendered. For that reason, the two main parameters tested are the average visible area (see equation 4.1)

$$\frac{\sum \text{visibleArea}}{\text{roomAmount} \cdot \text{totalArea}} \tag{4.1}$$

and the average weighted visible area (see equation 4.2),

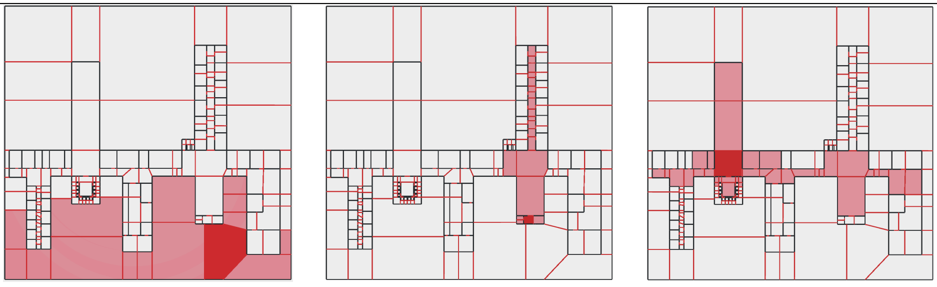
$$\sum \left(\frac{\text{roomArea}}{\text{totalArea}} \cdot \frac{\text{visibleArea}}{\text{totalArea}} \right) \tag{4.2}$$

where the values are summed over all cells in the partition. While the average visible area on describes how much percent of the scene is visible from any scene on average, the weighted visible area takes into account the size of the tested cell while calculating the average. The value of weighted average range from zero to one, where zero indicates that every cell visibility is only that cell itself and one indicates that every cell can see whole scene. Both the average and weighted areas can show how much a scene is opened. For scenes such as houses, which have a high percentage of the average area, it can be assumed that it is very open. On the other hand, a scene like Vienna should have many cells that can see very little. The measured values can be seen in Table 4.7. Furthermore, examples of PVS of a selected cell for selected scenes can be seen in Table 4.8.

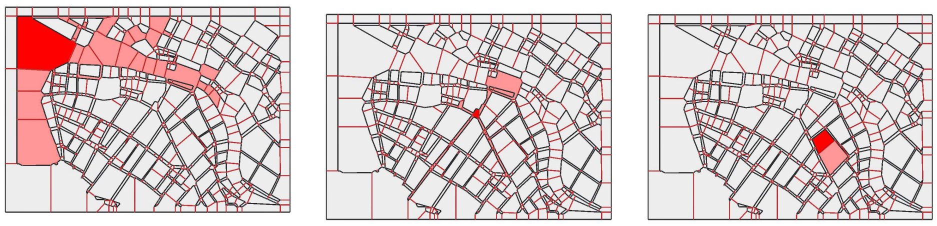
	Ratio 3		Ratio 6		Ratio 10	
Scene	Area	AreaW	Area	AreaW	Area	WArea
Cross	26.64%	0.3	30.45%	0.41	31.85%	0.44
Houses	20.01%	0.16	21.17%	0.21	19.37%	0.24
FEL	6.65%	0.25	8.29%	0.37	8.34%	0.44
Vienna	2.76%	0.06	3.51%	0.08	4.38%	0.09
Pompeii	4.07%	0.06	3.19%	0.064	2.39%	0.74
Soda Hall	3.64%	0.13	3.34%	0.17	3.27%	0.21

Table 4.7: Tested values of Potentially visible set for different portal ratios.

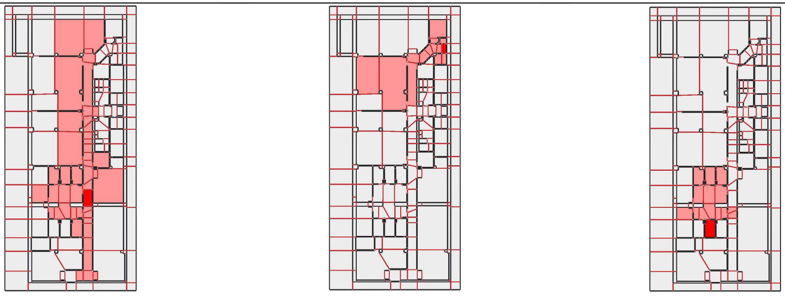
After the PVS is generated, there are multiple ways it can be used. The first is the intended use as a tool to decrease rendering times by hiding the non-visible objects. That is a simple optimisation technique, but it is only useful for a first-person application. If the application was used for some strategy or building game from a third-person view, this would not be useful. On the other hand, for these types of game, the PVS can find its own purpose. Let us say there is a game that lets you build and decorate buildings. The whole application, not just the PVS, would be very useful for such a task. Not only can a user create a building, it can also generate a PVS partition, which would be useful for queries such as detecting in which room an object lies. There may be a mechanic for which some objects need to lie in specific rooms. A PVS partition can then be used for a quick query, where it is not necessary to test against a whole room, but only a small part.



FEL



Vienna



Soda Hall

Table 4.8: Resulting PVS for selected (red) cells. The visible cells are shown as light red cells.

4.4 Final time

The graph 4.2 represents separate times of the partition. Each column consists of times for half edge generation, Breaking the Walls first and second passes, and PVS creation, as well as the total summed time. Furthermore, these number values can be seen in Table 4.9.

It can be seen that the amount of input walls is not the main identification of the computation time. Scene Vienna, while having fewer walls than the Pompeii or Soda Hall scene, has the computation time higher. The main reason is the Breaking the Walls algorithm. The scene Vienna has many long walls that are close to each other. The way the Breaking the Walls algorithm works is by connecting close walls, which results in creating many portals and a longer computation time.

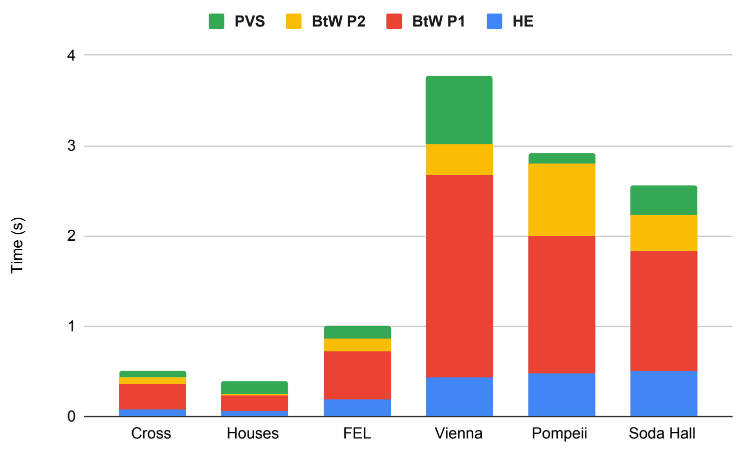


Figure 4.2: Resulting times of the partition, split into half edge generation, Breaking the Walls first and second pass and PVS generation.

Scene	HE	BtW1	BtW2	PVS	Total
Cross	0.0799	0.286	0.064	0.08	0.51
Houses	0.059	0.169	0.167	0.14	0.385
FEL	0.1899	0.5234	0.155	0.135	1
Vienna	0.437	2.237	0.335	0.762	3.77
Pompeii	0.478	1.519	0.803	0.116	2.92
Soda Hall	0.498	1.329s	0.409	0.327	2.24

Table 4.9: All measured times of the algorithm in seconds, along with the total time.

4.5 Local modification

Additionally to the partition generation a local update of this partition was also implemented. In this way, the used can change a wall without having to recalculate the whole partition. The local update can change the Breaking the Walls partition only in a local area by first removing the selected wall and then inserting it at a different place. By using this, all unchanged walls do not change inside of the partition, and only the changed ones are recalculated. Note that the potentially visible sets are not updated. The reason being that a simple change can drastically change the potentially visible sets, and recalculation would be complicated.

The average time for a local update can be seen in the table 4.10 compared to the total time for the half-edge data structure and the generation of the Breaking the Walls partition.

Scene	Total	Local update
Cross	0.43	0.038
Houses	0.24	0.01
FEL	0.87	0.024
Vienna	3	0.086
Pompeii	2.8	0.19
Soda Hall	2.24	0.11

Table 4.10: Total time for half-edge data structure and Breaking the Walls partition generation compared to the local update. Potentially visible sets generation is not part of the total time.

Chapter 5
Conclusion

The goal of the thesis was the implementation of the Breaking the Walls algorithm used for generation of Cells and Portals partition. For that purpose different data structures and Cells and Portals implementation were explored. Next, the algorithm for both the half-edge data structure used by the Breaking the Walls algorithm and the Breaking the Walls algorithm itself was then proposed and implemented in the Unity engine. The implementation also contained a feature for a local edit of the partition without the need to generate it from the beginning. Lastly, the implementation was evaluated on six different scenes for its time complexity and different measured values.

Although there were some problems during implementation, such as multiple stages of valid portal detection, which changed many times during implementation, the algorithm prototype was able to generate a complex Cells and Portals partition, as well as valid potentially visible sets. The local change was also completed successfully, as the measured times showed considerable improvement in the time it took to update the partition instead of generating it from the beginning. While the algorithm isn't able to process every scene, since some scene with degeneracies such as multiple parallel overlapping walls or very close walls, can cause errors, most of the scenes very computed correctly. The application in Unity allowed easy control over the algorithm and the scene itself, as it allowed editing, deleting, and saving scenes.

Some possible expansions of this work could contain different types of half-edge generations, as this one was based on the Unity physics engine instead of building a dynamic data structure. Or possibly a better detection of valid portal placement between two half-edges. The current implementation is valid, as it created a valid portal and does not create too many of them, which would be deleted in the validation pass, but a different approach to creation could be more beneficial.

Overall, the implementation was a success, as it brought a dynamic algorithm for generation of Cells and Portals and potentially visible sets, which can be used for additional games, as the prototype was implemented in the Unity game engine.

Bibliography

[AAM04] Ulf Assarsson and Tomas Akenine-Möller, *Occlusion culling and z-fail for soft shadow volume algorithms," the visual computer*, The Visual Computer **20** (2004), 601–612.

[Air90] John M. Airey, *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*, Ph.D. thesis, NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE, 1990.

[Bau72] Bruce G Baumgart, *Winged edge polyhedron representation*, Stanford University Stanford, California, 1972.

[BJL$^+$16] Richard Bormann, Florian Jordan, Wenzhe Li, Joshua Hampp, and Martin Hägele, *Room segmentation: Survey, implementation, and analysis*, 2016 IEEE International Conference on Robotics and Automation (ICRA), 2016, pp. 1019–1026.

[COCS01] Daniel Cohen-Or, Yiorgos Chrysanthou, and Cláudio Silva, *A survey of visibility for walkthrough applications*, Proceedings of SIGGRAPH (2001).

[HDS03] D Haumont, O Debeir, and F Sillion, *Volumetric cell-and-portal generation*, Comput. Graph. Forum **22** (2003), no. 3, 303–312 (en).

[HvDM$^+$13] John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven Feiner, and Kurt Akeley, *Computer graphics: Principles and practice*, 3 ed., Addison-Wesley, Upper Saddle River, NJ, 2013.

[Ket23] Lutz Kettner, *Cgal 5.6 - halfedge data structures: User manual*, Jul 2023.

[LCCO03] A. Lerner, Y. Chrysanthou, and D. Cohen-Or, *Breaking the walls: scene partitioning and portal creation*, 11th Pacific Conference on Computer Graphics and Applications, 2003. Proceedings., 2003, pp. 303–312.

[LG95] David Luebke and Chris Georges, *Portals and mirrors: Simple, fast evaluation of potentially visible sets*, Proceedings of the 1995 Symposium on Interactive 3D Graphics (New York, NY, USA), I3D '95, Association for Computing Machinery, 1995, p. 105–ff.

[LH03] Sylvain Lefebvre and Samuel Hornus, *Automatic Cell-and-portal Decomposition*, Tech. Report RR-4898, INRIA, July 2003.

[MB90] F MEYER and Serge Beucher, *Morphological segmentation*, Journal of Visual Communication and Image Representation - JVCIR **1** (1990), 21–46.

[McG00] Max McGuire, *The half-edge data structure*, Website: http://www. flipcode. com/articles/article halfedgepf. shtml (2000).

[OP13] R. Oliva and N. Pelechano, *NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments*, Computers & Graphics **37** (2013), no. 5, 403–412.

[O'R98] J. O'Rourke, *Computational geometry in c*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[Tel92] Seth Jared Teller, *Visibility computations in densely occluded polyhedral environments*, Ph.D. thesis, EECS Department, University of California, Berkeley, Oct 1992.

[Tót05] Csaba D Tóth, *Binary space partitions: recent developments*, Combinatorial and Computational Geometry **52** (2005), 525–552.

[vdPS99] Michiel van de Panne and A. James Stewart, *Effective compression techniques for precomputed visibility*, Eurographics Workshop on Rendering, June 1999, pp. 305–316.

Appendix A

Additional Test Results

This chapter contains all the results of the tests compacted into separate sections based on the scene. Compared to the information presented in the results chapter, where all the data was written in order of execution of the algorithm, this chapter contains the data separated by scenes together. In addition, images of scenes not presented before can be seen.

The tested values are the same as in the result chapter, ranging from the number of input walls *Walls*, amount of newly generated *Edges*, *Rooms*, the amount of intersected executed *Intersections* and time *Time* for half-edge data structure. *Portal[+,-]* for portals added or removed during *P[1,2]* first or second pass. Average visible area *Area* and weighted average area *AreaW* for portal ratio *Ratio[3,6,10]* and lastly measured times for each parts of the algorithm additionally with local edit of one input wall *Edit*.

A.1 Cross

The Cross scene was created by a simple Python script by generating random length line segments with spaces between them in a grid-like pattern.

Cross Half-edge data structure					
Walls	Edges	Rooms	Intersections	Time (s)	
97	163	48	136	0.0799	
Breaking the Walls					
Portal+ P1	Time P1 (s)	Portal- P2	Portal+ P2	Time P2 (s)	
135	0.2859	52	26	0.06401	
Potentially visible sets					
Ratio 3		Ratio 6		Ratio 10	
Area	AreaW	Area	AreaW	Area	WArea
26.64%	0.3	30.45%	0.41	31.85%	0.44
Total times					
HE	BtW1	BtW2	Total	Edit	PVS
0.0799	0.286	0.064	0.8	0.039	0.08

Table A.1: Measured values for scene Cross

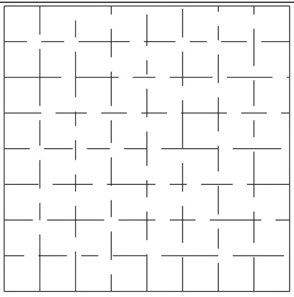
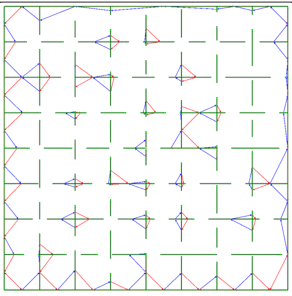
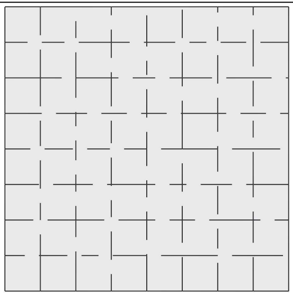
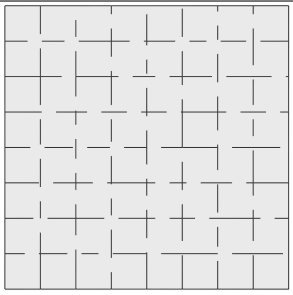
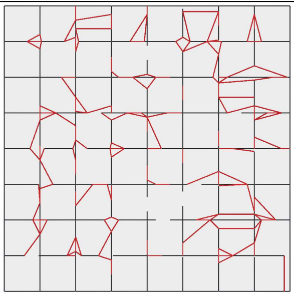
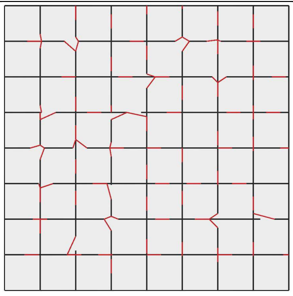
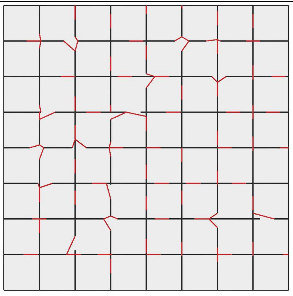
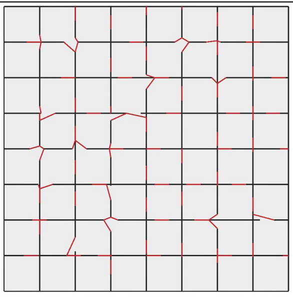
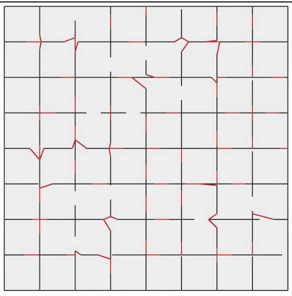
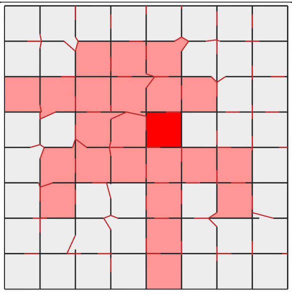
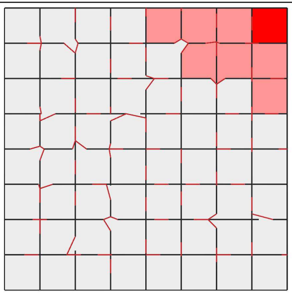
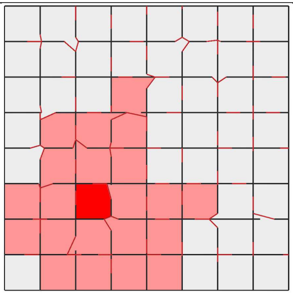
		
Input	Half-Edge	Rooms
		
Rooms	BtW first pass	BtW validation pass
		
	Breaking the Walls validation pass - ratio 3, 6, 10	
		
	PVS Examples	

Table A.2: Step-by-step partition of scene Cross.

A.2 Houses

The Houses scene was also created by a Python script as squares with random point offset in a grid-like pattern. This scene simulates the algorithm's behaviour in a sparsely developed area.

Houses Half-edge data structure									
Walls 104		Edges 104		Rooms 52		Intersections 104		Time (s) 0.05927	
Breaking the Walls									
Portal+ P1 123		Time P1 (s) 0.1689		Portal- P2 18		Portal+ P2 7		Time P2 (s) 0.01673	
Potentially visible sets									
Ratio 3			Ratio 6			Ratio 10			
Area 20.01%	AreaW 0.16		Area 21.17%	AreaW 0.21		Area 19.37%	WArea 0.24		
Total times									
HE 0.059	BtW1 0.169		BtW2 0.167	Total 0.14		Edit 0.01	PVS 0.14		

Table A.3: Measured values for scene Houses

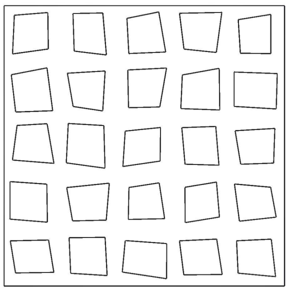
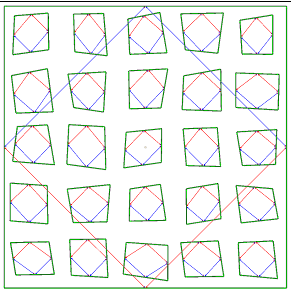
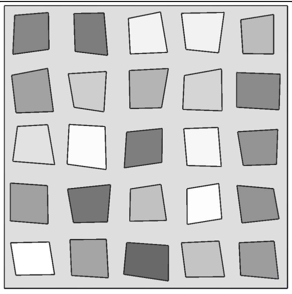
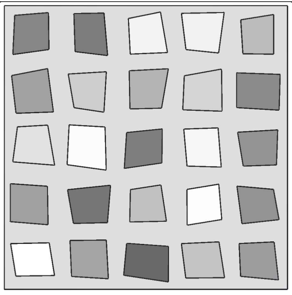
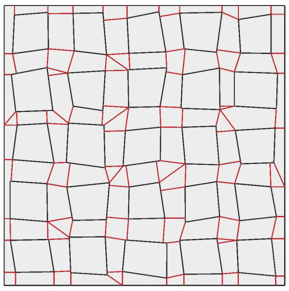
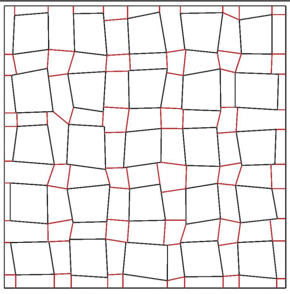
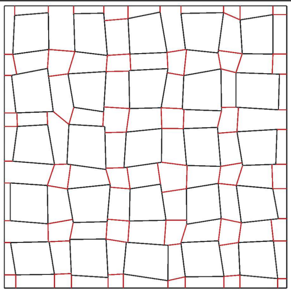
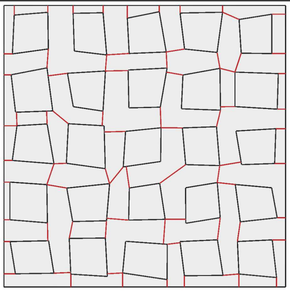
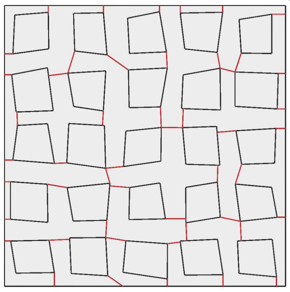
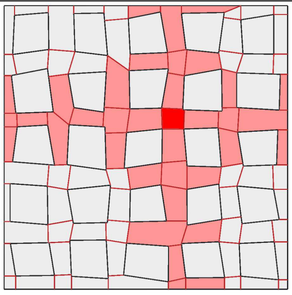
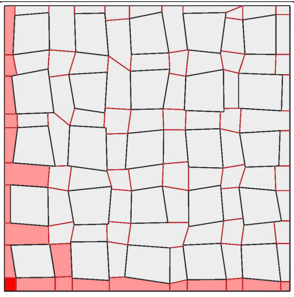
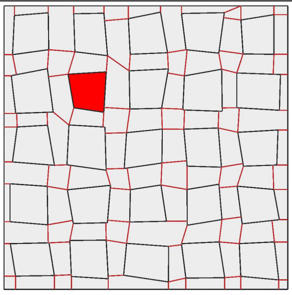
		
Input	Half-Edge	Rooms
		
Rooms	BtW first pass	BtW validation pass
		
Breaking the Walls validation pass - ratio 3, 6, 10		
		
PVS Examples		

Table A.4: Step-by-step partition of scene Houses.

A.3 FEL

This scene was created by hand based on the first floor of the Czech Technical University in Prague, located at Technicka 2.

FEL Half-edge data structure				
Walls	Edges	Rooms	Intersections	Time (s)
299	300	11	300	0.1899
Breaking the Walls				
Portal+ P1	Time P1 (s)	Portal- P2	Portal+ P2	Time P2 (s)
492	0.5234	51	83	0.1546

Potentially visible sets					
Ratio 3		Ratio 6		Ratio 10	
Area	AreaW	Area	AreaW	Area	WArea
6.65%	0.25	8.29%	0.37	8.34%	0.44
Total times					
HE	BtW1	BtW2	Total	Edit	PVS
0.1899	0.5234	0.155	0.868	0.024	0.135

Table A.5: Measured values for scene FEL

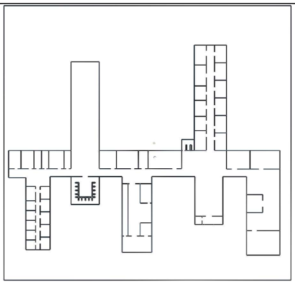
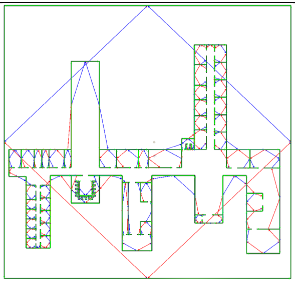
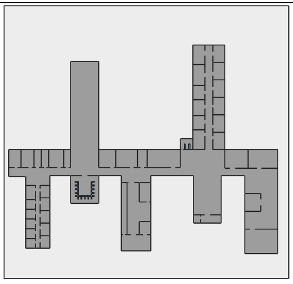
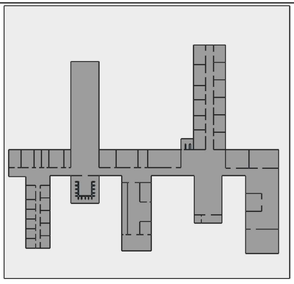
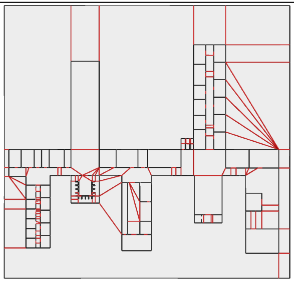
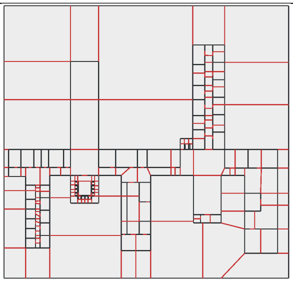
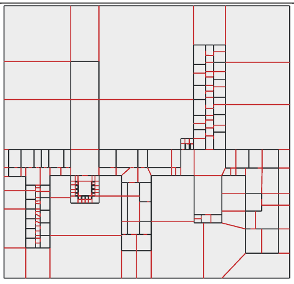
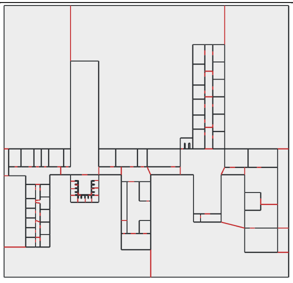
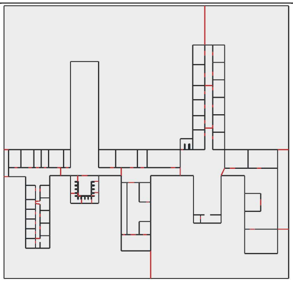
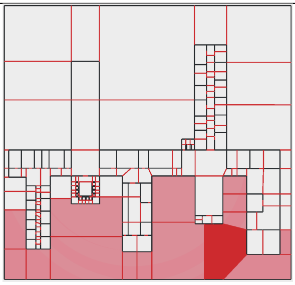
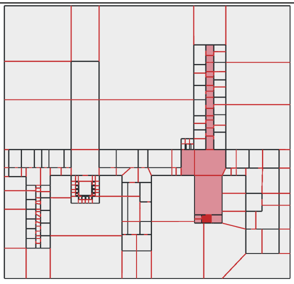
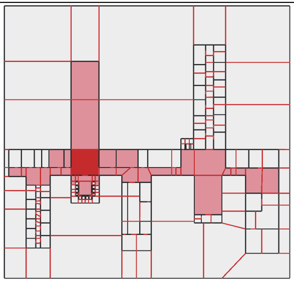
		
Input	Half-Edge	Rooms
		
Rooms	BtW first pass	BtW validation pass
		
	Breaking the Walls validation pass - ratio 3, 6, 10	
		
	PVS Examples	

Table A.6: Step-by-step partition of scene FEL.

A.4 Vienna

The Vienna scene is extracted from the part of the 3D model of the city of Vienna. The extraction, as for all scenes based on 3D models, was done in 3D software Blender. The model was bisected to find the floor plan. As the bisection returned a rough model with errors, a final floor plan was extracted by hand and simplified.

Vienna					
Half-edge data structure					
Walls	Edges	Rooms	Intersections	Time (s)	
658	660	186	659	0.437	
Breaking the Walls					
Portal+ P1	Time P1 (s)	Portal- P2	Portal+ P2	Time P2 (s)	
921	2.2373	316	121	0.33531	
Potentially visible sets					
Ratio 3		Ratio 6		Ratio 10	
Area	AreaW	Area	AreaW	Area	WArea
2.76%	0.06	3.51%	0.08	4.38%	0.09
Total times					
HE	BtW1	BtW2	Total	Edit	PVS
0.437	2.237	0.335	3.009	0.085	0.762

Table A.7: Measured values for scene Vienna

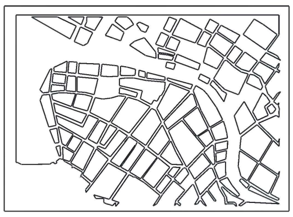
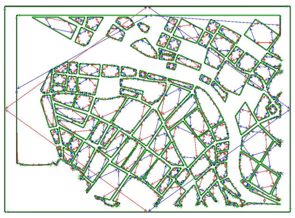
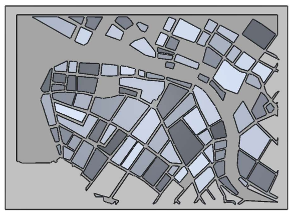
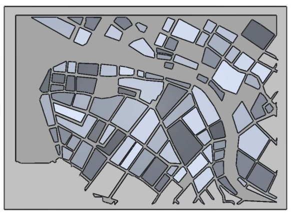
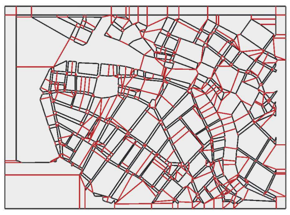
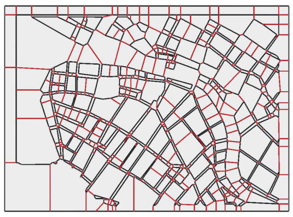
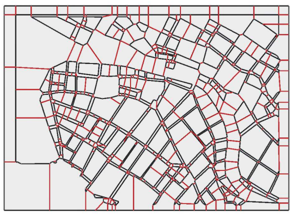
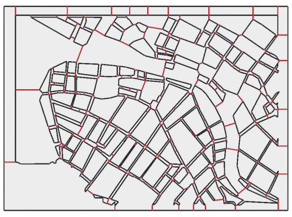
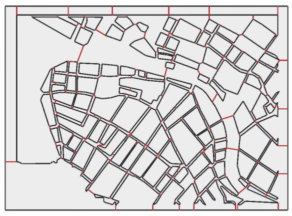
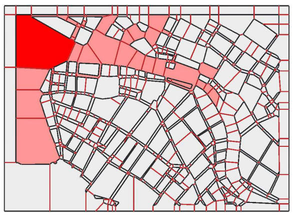
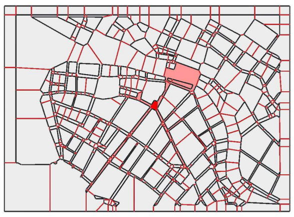
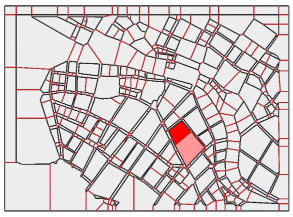
		
Input	Half-Edge	Rooms
		
Rooms	BtW first pass	BtW validation pass
		
	Breaking the Walls validation pass - ratio 3, 6, 10	
		
	PVS Examples	

Table A.8: Step-by-step partition of scene Vienna.

A.5 Pompeii

Pompeii scene is another scene based on a 3D model. This model is based on the reconstruction of the city of Pompeii. The scene is only a small part of the whole 3D model. In addition, the scene is extremely simplified, as the original model contained many problematic parts, such as parallel lines, that needed to be removed. The removal was automatic by simplifying the mesh as well as by hand, as some errors remained.

Pompeii					
Half-edge data structure					
Walls	Edges	Rooms	Intersections	Time (s)	
755	758	229	757	0.4783	
Breaking the Walls					
Portal+ P1	Time P1 (s)	Portal- P2	Portal+ P2	Time P2 (s)	
397	1.5192	162	82	0.8029	
Potentially visible sets					
Ratio 3		Ratio 6		Ratio 10	
Area	AreaW	Area	AreaW	Area	WArea
4.07%	0.06	3.19%	0.064	2.39%	0.74
Total times					
HE	BtW1	BtW2	Total	Edit	PVS
0.478	1.519	0.803	0.116	0.189	0.116

Table A.9: Measured values for scene Pompeii

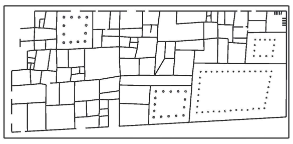
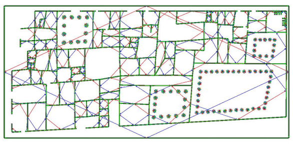
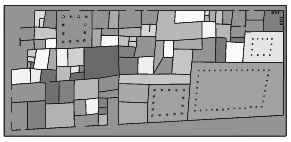
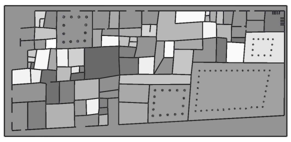
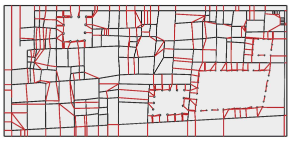
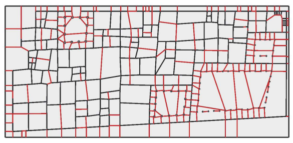
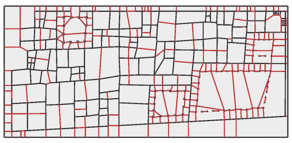
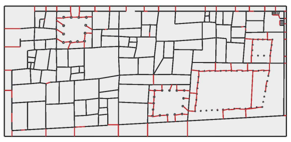
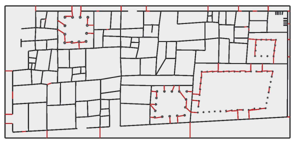
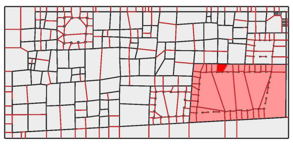
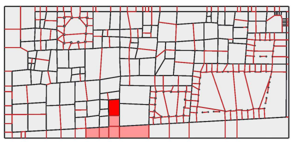
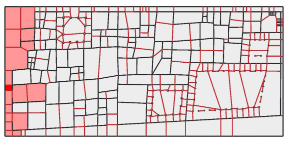
		
Input	Half-Edge	Rooms
		
Rooms	BtW first pass	BtW validation pass
		
Breaking the Walls validation pass - ratio 3, 6, 10		
		
PVS Examples		

Table A.10: Step-by-step partition of scene Pompeii.

A.6 Soda Hall

The last scene, Soda Hall, is also generated from a 3D model based on the Soda Hall building of the University of California in Berkeley. The scene is extracted from the first floor of the building in the same way as the previous 3D models.

Cross					
Half-edge data structure					
Walls	Edges	Rooms	Intersections	Time (s)	
759	777	142	768	0.499	
Breaking the Walls					
Portal+ P1	Time P1 (s)	Portal- P2	Portal+ P2	Time P2 (s)	
481	1.3293	137	142	0.40878	
Potentially visible sets					
Ratio 3		Ratio 6		Ratio 10	
Area	AreaW	Area	AreaW	Area	WArea
3.64%	0.13	3.34%	0.17	3.27%	0.21
Total times					
HE	BtW1	BtW2	Total	Edit	PVS
0.498	1.329s	0.409	2.237	0.106	0.327

Table A.11: Measured values for scene Soda Hall

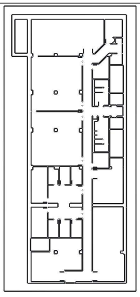
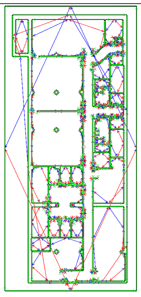
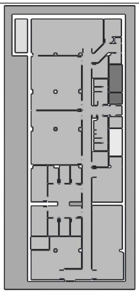
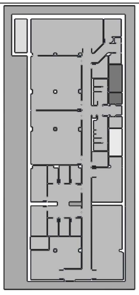
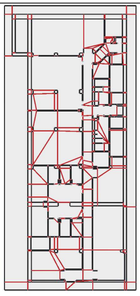
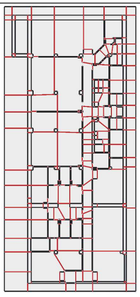
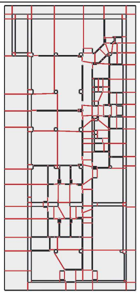
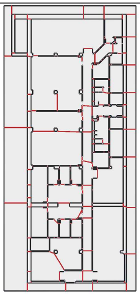
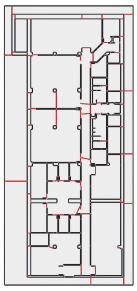
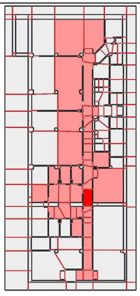
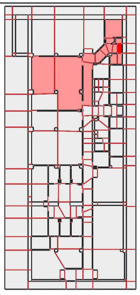
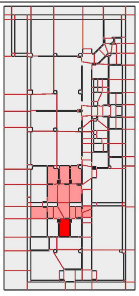
		
Input	Half-Edge	Rooms
		
Rooms	BtW first pass	BtW validation pass
		
	Breaking the Walls validation pass - ratio 3, 6, 10	
		
	PVS Examples	

Table A.12: Step-by-step partition of scene Soda Hall.

Appendix B
User Manual

The application was implemented in Unity 2022.3.4f1. It may not work properly in other versions. To open the app, it is recommended to use Unity Hub, from where users can easily manage Unity projects. To select the project, the user needs to select the *Add* button and choose the project folder. The project will then show and can be opened just by clicking on it.

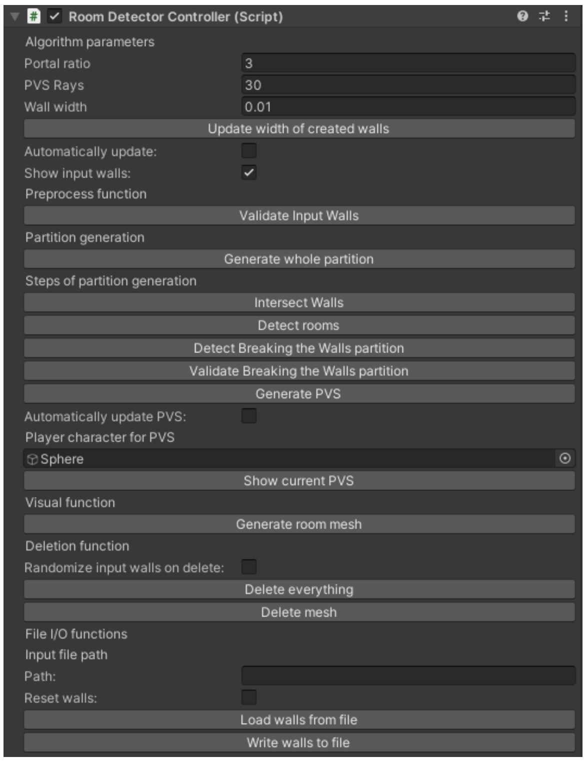
The application contains multiple folders for scripts, material, etc. For the user, the only necessary folder is folder *Scenes*, which contains the tested scenes, as well as an empty scene *Empty*, which is an empty scene only with the algorithm controller. As stated before, the algorithm has two run options, in the editor or at run-time. These options will be explained individually as they have different controls, while allowing the same algorithm execution.

B.1 Editor mode

In editor the main object is *RoomDetectorController*, which can be found in scene hierarchy. If the scene does not contain one, it can be added by right-clicking the hierarchy or in the menu bar in the *GameObject*. Then in both ways, it can be added by choosing the option *RoomDetect/RoomController*. None that scene cannot contain two controllers, as one would stop working.

To add a new input wall, the user needs to find the *InputWall* button, which is located in the same path as the room controller. From there it is possible to select either of the wall vertices and move them.

After selecting the controller object, a window will appear in the inspector, visible in Figure B.1. This window contains values for the algorithm, as well as buttons that start the different parts of the algorithm.



Room Detector Controller (Script)

Algorithm parameters

Portal ratio: 3

PVS Rays: 30

Wall width: 0.01

Update width of created walls

Automatically update: ☐

Show input walls: ✓

Preprocess function

Validate Input Walls

Partition generation

Generate whole partition

Steps of partition generation

Intersect Walls

Detect rooms

Detect Breaking the Walls partition

Validate Breaking the Walls partition

Generate PVS

Automatically update PVS: ☐

Player character for PVS

Sphere

Show current PVS

Visual function

Generate room mesh

Deletion function

Randomize input walls on delete: ☐

Delete everything

Delete mesh

File I/O functions

Input file path

Path:

Reset walls: ☐

Load walls from file

Write walls to file

Figure B.1: The room detection controller window used in editor mode.

B.1.1 Parameters

The first part of the window is the algorithm parameters, visible in Figure B.2. These are *Portal ratio* used for Breaking the Walls algorithm seconds pass, amount of *PVS Rays* which are used to sample the visibility of potentially visible sets and *Wall width* representing the width of the shown walls. Note that the *Wall width* is tied to the algorithm, as the width of the walls depends on which walls are considered as intersected. With a bigger *Wall width* the algorithm can give incorrect results, which is why it is recommended to keep the number low. The next parameter is *Automatically update* toggle, which, if activated, allows the algorithm to update the partition locally when any input wall is changed. Note that the automatic update will update the partition by completing the Breaking the Walls partition. If used before the algorithm finishes, such as after the half-edge generation phase, it will automatically complete it. Also, by selecting the vertex, the wall will get removed from the structure, and by deselecting the vertex, it will get added back. In the editor mode, only the first selected wall vertex will be updated. Last parameter is *Show input walls* which if togged will either hide or show the input wall of the algorithm. The walls are automatically hidden after the half-edge generation phase, after which they are not used anymore. The user can reveal them to update the partition locally.

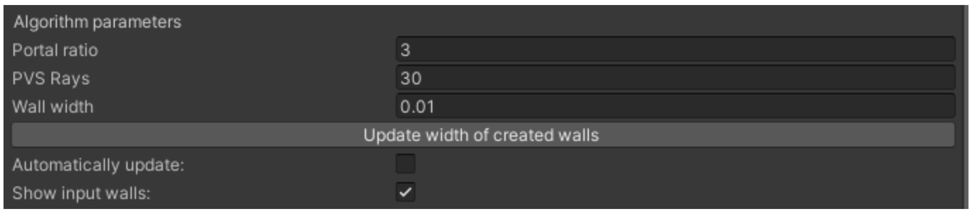

Algorithm parameters
Portal ratio 3
PVS Rays 30
Wall width 0.01
Update width of created walls
Automatically update: ☐
Show input walls: ✓

Figure B.2: Parameters in the controller window.

B.1.2 Preprocess

The next set of buttons is for the algorithm itself. The first button is *Validate Input Walls* for preprocessing. The input may contain parallel walls, which are illegal and would most likely break the algorithm. This button attempts to merge the overlapping walls into one. Additionally, it will also merge parallel adjacent lines into one to eliminate unnecessary calculations during the half-edge generation step. Note that this algorithm may not detect all input problems and the partition may still be generated incorrectly.

B.1.3 Algorithm execution

The next set of buttons is for different steps of the whole algorithm. The button *Generate whole partition* is used to execute all the steps of the algorithm at once, except the PVS. For a step-by-step update, the user can use buttons *Intersect walls*, *Detect rooms*, *Detect Breaking the Walls partition*, *Validate Breaking the Walls partition*, which all represent the different steps, all visible in Figure B.3

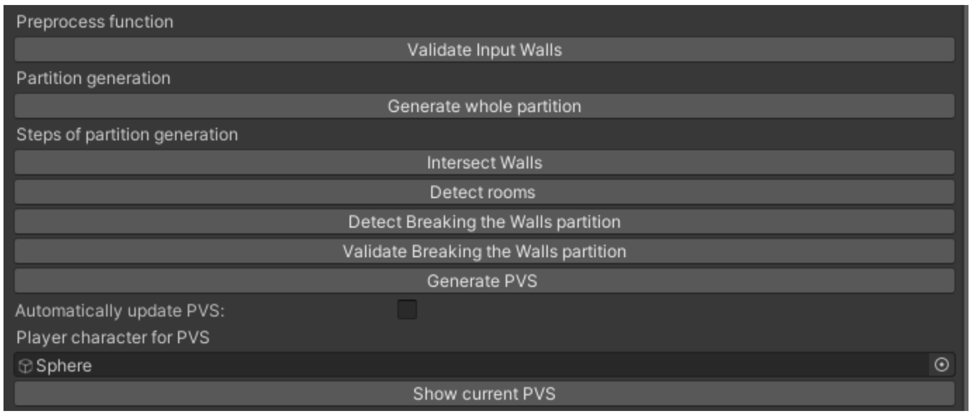

Preprocess function
Validate Input Walls
Partition generation
Generate whole partition
Steps of partition generation
Intersect Walls
Detect rooms
Detect Breaking the Walls partition
Validate Breaking the Walls partition
Generate PVS
Automatically update PVS: ☐
Player character for PVS
Sphere
Show current PVS

Figure B.3: Algorithm execution in the controller window.

PVS can be generated using the *Generate PVS*. It's visualisation can be controller by the user using the *Automatically update PVS* to show PVS of the room in which the object chosen in *Player character for PVS* is continually or just once using the *Show current PVS* button. The PVS is shown as red tinted rooms, where the selected rooms is red and the visible rooms light red. An example of this can be seen in Figure B.4.

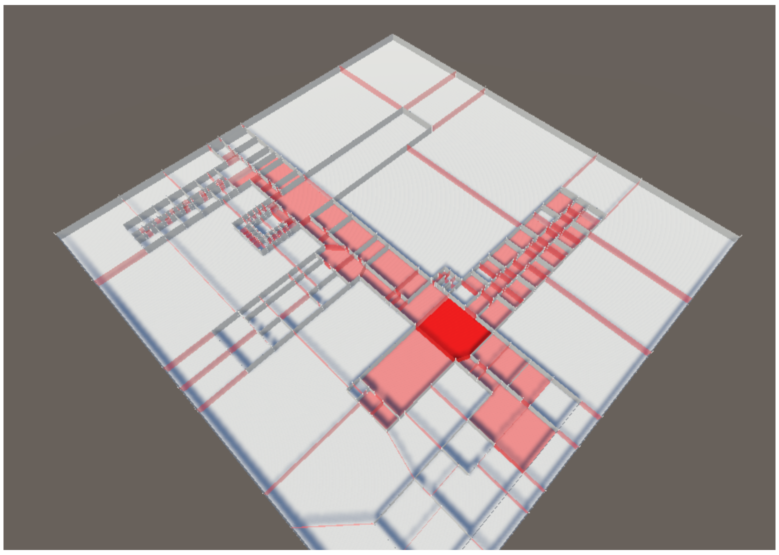


Figure B.4: Example of a visualised PVS of a room. Dark red is selected room, light red are visible rooms.

B.1.4 Mesh generation

The next button is *Generate room mesh* for mesh generation since room mesh is not automatically generated until PVS is created, as the algorithm does not use it for anything. Note that the room mesh can be deleted at any step of the Breaking the Walls algorithm.

B.1.5 Structure deletion

Another set of buttons, which can be seen in Figure B.5, is used to remove the structure. The buttons *Delete everything* and *Delete mesh* are to remove the structure or just the mesh in particular. Furthermore, the user can toggle the *Randomise input walls on deleting* field to change the order of the input walls that the algorithm takes. This may change the resulting partition and sometimes help with the algorithm, as some input order may work better than another.

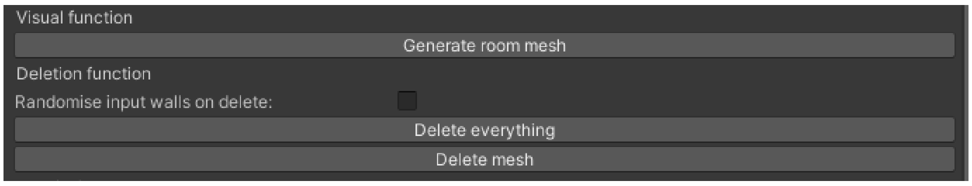


Figure B.5: Visual and deletion functions in the controller window.

B.1.6 Input and output

Input and output (I/O) functions, visible in Figure B.6, are used to load and save walls on the input. The user needs to input the save path in the *Path* field. This path can be either absolute or relative to the project folder. Then the buttons *Load/Write walls* can be used to load or write the walls in a scene in/from a file. During file loading, the user can toggle the *Reset walls* button to delete all walls in the scene and replace them with those in the input file.

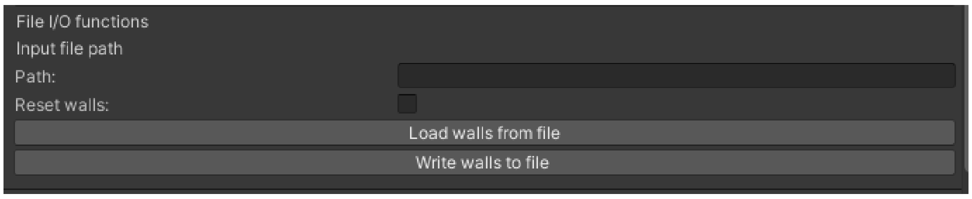


Figure B.6: I/O functions in the controller window.

B.2 Run-time mode

Similar to the editor mode, the run-time mode is also controlled by a set of buttons. These buttons are embedded in the scene user interface and can be controlled in the same way. The UI can be seen in Figure B.7.

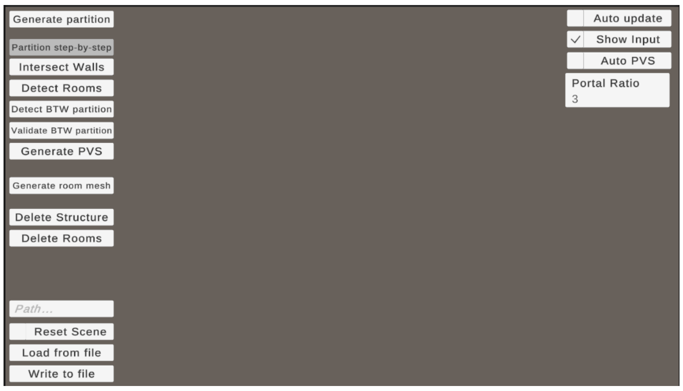


Figure B.7: The UI controller for the run-time mode.

The buttons have the same function as the ones in the editor controller described in the previous chapter. The main difference between the editor mode is first the controls. The user can move around the scene using the *wasd* keys and mouse. To rotate the camera, it is necessary to press the right mouse button. All buttons can be controlled simply with the mouse. The other difference is in PVS visualisation, compared to the selected object, which was used in edit mode, in execution mode the user can simply point on the room to show it's PVS. To automatically see PVS, it is necessary to toggle the *Auto PVS* option.

The run-time mode currently does not contain all the buttons of edit mode, as the amount of the button would be overwhelming, so only the necessary ones are chosen.

To add a new input wall to the scene, the user must press a *Q* key to place the starting vertex. While still holding the *Q* button, the other vertex can be moved before it is placed again. The placement is finished when the user presses the *Q* key. In addition, the user can easily move the wall vertices simply by clicking and dragging them into the scene. To delete a wall, the user needs to have one of it's vertices selected and press *Delete*. The wall with both it's vertices will be deleted.

As it is not possible to change scenes, the user can press the *Escape* button to enter a scene selection menu, where he can load the scenes which were tested in this paper.

Appendix C
Content of attached medium

`readme.txt`	Description of the content of attached medium.
`src`	Archive with implementation in the Unity engine.
`Assets`	
`Scenes`	Folder with tested scenes.
`Scripts`	Folder with algorithm scripts.
`Packages`	
`ProjectSettings`	
`bin`	
`RoomDetection.exe`	The executable application.
`latex.zip`	Source of thesis in LaTeXformat.
`imgs.zip`	Images presenting the implementation.