Computer Science • 17 (3) 2016 http://dx.doi.org/10.7494/csci.2016.17.3.321

Piotr Turecki
Tomasz Pędzimąż
Szymon Pałka
Bartosz Ziółko

AUTOMATIC PORTAL GENERATION FOR 3D AUDIO – FROM TRIANGLE SOUP TO A PORTAL SYSTEM

Abstract *The purpose of this paper is to investigate an algorithm for generating an automatic portal system. This has been accomplished based on a given set of triangles. The proposed solution was designed to enhance the performance of a sound beam-tracing engine. This solution can also be used for other areas where portal systems are applicable. The provided technical solution emphasizes the beam tracing engine's requirements. Our approach is based on the work of Haumont et al. (with additional improvements), resulting in improved scene segmentation and lower computational complexity. We examined voxelization techniques and their properties, and have adjusted these to fit the requirements of a beam-tracing engine. As a result of our investigation, a new method for finding portal placement has been developed by adjusting the orientation of the found portals to fit the neighboring scene walls. In addition, we replaced Haumont et al.'s prevoxelization step, which is used for erasing geometrical details (for example, thin walls). This was done by smoothing the distance field that, in effect, eliminated incorrectly positioned portals. The results of our work remove the requirement for walls that separate rooms to have a particular thickness. We also describe a method for building a structure that accelerates real-time queries for determining the area where a given point is located. All of the presented techniques allow for the use of larger sized voxels, which increases performance and reduces memory requirements (not only during the preprocessing phase but also during real-time usage). The proposed solutions were tested using scenarios with scenes of varying complexity.*

Keywords voxelization, portal generation, geometry occlusion, watershed transform

Citation Computer Science 17 (3) 2016: 321–352

1. Introduction

Because of the growing size of the worlds presented in video games and other real-time simulations, automatic methods for generating structures used to detect visible objects are becoming increasingly important. They significantly accelerate scene processing in video and audio simulations and allow for the use of more-complicated scenes. Currently, two solutions are popular: Auto-generated portals (based on a given level input geometry) and Occlusion Culling (OC) techniques [15]. In the first case, a very popular solution is middleware Umbra [28]. Simpler solutions such as a Binary Space Partitioning (BSP) tree [40] or Bounding Volume Hierarchy (BVH) [27] are also known. The OC solution is popular and has been implemented in a number of game engines (e.g., CryENGINE). An algorithm for automatic portal placement was already suggested by D. Haumont [13, 14]. Automated solutions, which allow for eliminating errors caused by oversights during manual editing, can limit the time an artist spends on this purpose and facilitate changes in the projects levels. Our work extends some of the previous solutions for automatic portal generation by increasing their performance and accuracy. To validate the accuracy, scenes with different geometrical complexity were tested.

Some systems for simulating architectural acoustics based on virtual scenes already exist [21, 30]. Actual acoustic measurements are also available and can be used to validate sound-simulation outcomes [19]. However, even on the fastest personal computers, the computations needed to accurately simulate sound cannot be performed at interactive rates, and there is a growing need to conduct similar tasks in real time. A possible way to achieve this is to use the beam-tracing technique [7, 25, 36]. This solution works by determining visibility from a point or from an area and conducting it thousands of times per second. The application of a scene partitioning with a portal system can significantly reduce the size of geometry needed to be processed for each visibility query. In this approach, processing is done per room instead of per scene. This is due to each room being completely separated from others, either by its walls or portals. Before beam tracing starts, the room in which the receiver is positioned must be determined. Application of a portal system guarantees that each area is separated from one another, either by walls or portals. This means that the visibility determination from a receiver position gives a subset of walls and portals in the searched area. A beam can be reflected off, transmitted, or diffracted on any of the visible walls in this set. It can also be transmitted through a portal. In either of these cases, it can be easily determine which areas intersect the newly created beam. As a result, a set of walls possibly visible for a single beam is significantly reduced due to the usage of a portal system.

The goal of the system is to accelerate the geometric subsystem sound engine RAYA. In particular, its beam-tracing algorithm [36], which imposed some additional constraints on dividing a scene into separate cells. It also requires additional functionality, such as a method for determining the location of a particular point within a cell. The nature of the beam-tracing algorithms allows for the calculation of a set

of objects visible from many areas and in different directions during the processing of one frame [7, 25]. OC methods have proven to be ineffective in such a case. OC algorithms were designed for the fast elimination of large quantities of visible triangles in order to reduce the number of primitives rendered for one view frustum (e.g., to determine visibility for one frustum) per frame. In relation to beam tracing, scenes are used for processing several frusta per frame. Clearly, a set of possibly visible triangles (potentially a visible set) must be approximated very quickly for each processed beam. For its application in this scenario, an area-portal graph (which partitions an input scene into a number of areas) can be generated. The created rooms should contain a similar number of triangles to avoid large discrepancies between different scene areas (resulting from the different times of processing depending on the number of primitives in these areas). This requirement is not necessary in the case of using a portal system for accelerating visibility testing.

Haumont et al. [13] found that the best way to fulfill the requirements of a beam-tracing system was to use a method based on BSP trees and other types of hierarchical scene representation. This method does not reflect the shape of areas and requires splitting the input geometry, which increases the complexity. The method used in Umbra [28] is also less effective. In the case of applying Umbra, only convex areas connected by portals are generated (see Fig. 1). A supporting query is needed to determine the area containing the point. It is necessary to combine Umbra areas (which are not necessarily constrained only by scene triangles and portals) with an exact input of a geometrical structure (i.e., rooms, corridors, halls, open areas). Unfortunately, this is not a trivial issue (sometimes even for a human level designer). The method described in this paper generates areas of any shape, and thus there is no need to adjust them to the shape of the input geometry.

Scenes used in computer simulations are often made of areas connected by corridors or passages. A portal system is used for the segmentation of such scenes into well-defined areas. Each area is constrained by walls and portals in such a way that there is no pair of points in two separate areas that can be connected by a line which does not intersect any wall or portal.

2. Related work

There are many techniques for the accelerated approximation of a set of visible objects in the scene. One such technique is the classical visibility acceleration technique. This uses hierarchical scene representation. If we operate at the level of a triangle soup only, it can be a BSP tree [40], for example, which is created by recursively dividing the space using a plane defined by a triangle being currently added to the tree. BSP divides the list of input geometry triangles into quantitatively equal parts. If a triangle is cut by the plane splitting the tree, either plane divides the triangle into two smaller parts (since one of the parts can be quadrilateral, up to three triangles can be created) – and then each part is assigned to a corresponding set – or added unmodified to both sets. But then, repetitions of triangles found during visible-object lookup have to be

eliminated. Hybrid solutions are also possible if the graphic scene operates on whole objects. Another very good solution is to use some form of BVH tree [27]. This tree does not require the splitting of objects into smaller parts and can be built according to various heuristics (depending on the requirements). Also, it is not necessary to divide nodes of the constructed tree into two parts. Instead, one way to make better use of the vector instruction in modern processors is to split the node into four parts, allowing for parallel processing. This type of structure works best on static open scenes (e.g., city model, etc.) and architectural scenes (e.g., interior of a cathedral, home, etc.). Unfortunately, performance is significantly decreased if the scene has a lot of dynamic objects. With a relatively small cost, the tree can be modified to make it better-suited for a dynamic geometry.

The triangle splitting issue does not exist in the Occlusion Culling technique [15, 39], where an assumption is made that some number of large objects (occluders) occlude a significant portion of scene objects (occludes). A determined set of occluders allows for removing occludes from processing, thus speeding up the rendering process. There are many versions of this algorithm that use the depth buffer rendered by the GPU from the current frame, previous frame, or even multiple frames back. During implementation, it must be taken into account that such a buffer can provide no usable data after rapid camera movements.

Rendering the depth buffer on the CPU is also popular. A simplified version of the scene is rendered to the buffer, which means that only the large objects occupying a significant amount of space on the screen are drawn. The reason for this is that there is little chance that small objects will occlude a large number of other objects. This method works very well for dynamic scenes. However, using this technique alone is not suitable for beam tracing where many visibility tests must be performed per one frame. Depth buffer techniques require that new depth buffer must be computed for each query when used for determining the visibility in different places or directions. This is a computationally intensive process and cannot be performed as frequently as needed in beam tracing to achieve real-time performance (unless other structures such as the portal system have been used to decrease the number of primitives rendered in the depth buffer).

The general principle of the portal system-generation algorithm is the division of a scene into areas connected by portals. This is done in a way that ensures that adjacent areas can be seen from a neighboring area only when portals leading to them can be seen. Such an approach greatly reduces the number of processed objects. If some area is visible, it is assumed that all of the objects contained by it could be visible. As an option, more-accurate visibility testing of these objects can be performed. Although portals are used for a very long time (especially in computer games), it was the third generation (the latest) of visibility system Umbra [28] when a solution allowing the automatic generation of portals was introduced to the market. Scene segmentation can be achieved using Voronoi Diagrams, even with dynamic scenes [42]. Unfortunately, these techniques do not create three-dimensional scene segmentation at the moment.

Earlier solutions for portal usage required a manual setup (with a various amount of work needed for scene preparation). For example, in the second version of the Unreal Engine [16], portal construction was done with the help of a level editor built into the engine. Although a method of generating portals for arbitrary scenes was described in 2002 [14], use of the automatic generation of portals is not widespread yet in the video game industry due to difficulties associated with the proper implementation of such techniques. In the literature, a similar algorithm for automatic portal generation has been described [8, 32]. We have decided to base our work on that described by Haumont et al. [13]. This is because portals generated by this method fit the shape of the input geometry, which was a strong requirement for a portal system to be compatible with a soundtracer.

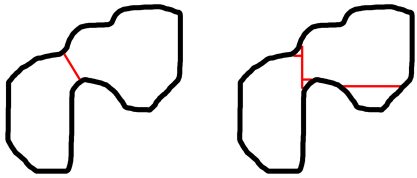

Figure 1. Illustrative comparison of a set of portals generated by our method (left) and the method used in Umbra (right). Umbra creates only convex areas; thus, many small areas are created. The proposed approach generates a lower number of areas with a similar number of triangles in each area. The outcome is more consistent with the human perception of a room.

3. Algorithm overview

The proposed method is based on the voxelization of the input geometry [14] and then computing the distance field [4]; i.e., assigning the shortest distance to any wall in the scene for each voxel. It is worth mentioning that the same voxelization can also be used to automatically generate a simplified geometry (through the so-called re-meshing), which can then be used in other algorithms (such as OC visibility [41]). In the vast majority of cases, Euclidian metrics are used for calculating the distance field. However, its low time efficiency makes it unacceptable for large scenes.

There are many algorithms that can accelerate distance-field calculations; for example, Distance Transform [29] or a solution using the GPU [10]. We chose Manhattan metrics due to their reasonable efficiency and relatively easy implementation. The Watershed segmentation algorithm itself was initially designed for the segmentation of 2D images [6], but it can be successfully adapted to 3D [13].

A method that allows for the performance of fast tests to resolve whether a given point is inside a calculated voxelized area is an important constraint imposed on the portal system being described. For a mesh construction from voxel data, a cube-marching algorithm [26] can be used. However, in its simplest form, this algorithm

generates a mesh consisting of a large number of triangles, which makes fast testing impossible. There are a number of more-advanced algorithms for this purpose [1, 33].

Partitioning of a scene into areas is done by using a three-dimensional extension of a well-known image-segmentation algorithm – Watershed [6]. This algorithm operates on an intermediate form of geometry that is distance-field generated from the input of a scene [4, 10, 18, 29]. It is worth mentioning that this generation of portals is not the only the application of the above-mentioned algorithm. For example, it can be used to automatically generate the navigation of a mesh that is used by artificial intelligence in games for automatic movement in a three-dimensional environment [8, 32, 42].

The presented algorithm consists of three stages. The first is where the input geometry is voxelized and distance field is computed. Also, the local maxima in the distance field are smoothed to minimize the number of incorrectly generated portals (subsection 3.1). A scene prepared in this way becomes the input for the second stage, which is a portal-detection algorithm (subsection 3.2). In the last stage (subsection 3.3), the data necessary for further usage of portals in real-time applications is calculated.

3.1. Preparation of input data

Input geometry voxelization is conducted by calculating a bounding box large enough to contain all of the objects in a scene and expanding it on each side by the size of a voxel. In the next step, the bounding box is filled with a three-dimensional grid of voxels (with a predetermined size). In such a grid, we specify two voxels as adjacent if they share a common wall. The distance between such voxel pairs is equal to 1. Voxels with a shared edge but no shared wall have a distance equal to 2.

Voxels intersecting any triangle are marked with a distance value of 0 (which is the initial step in building the values for the distance field). A difference between the proposed algorithm and Haumont's approach [13] lies in the fact that, in our case, voxels intersected by the geometry create an outline of a given geometry – the rest of the voxels create a space where portals and areas will be built. Voxels intersected by the geometry are ignored by the Haumont algorithm [13] (the authors do not specify whether these voxels are totally ignored or included in process of building portals and areas). Instead, voxels that are completely outside the reachable space (e.g., fills the wall) are detected, and they become an outline of the geometry – see Figure 2. It is worth mentioning that our voxelization leaves thin walls (thinner than the voxel size) while, in the case of Haumont's algorithm [13], these walls will disappear after the voxelization process (they are ignored by the algorithm).

To compute a distance field usable in practice, two versions of metrics can be used (Manhattan and Euclidean). The Manhattan distance is calculated using a floodfill algorithm starting from voxels with a distance equal to 0 (that is, voxels intersected by the geometry). Each voxel has a distance assigned that is defined as a minimal count of other voxels to be crossed until the nearest voxel which intersects the scene geometry is reached.

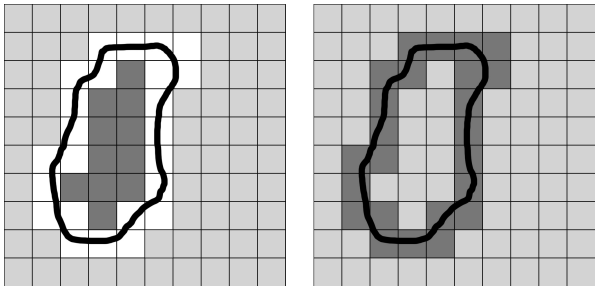

Figure 2. On the left – the concept of Haumont et al. [13]; on the right – the approach applied in our algorithm. Light gray indicates voxels that will be used for portal generation. Dark gray indicates voxels marked in our work as intersecting the geometry or (in the case of Haumont's algorithm [13]) being inside the geometry (e.g., fills the walls). Portals will be not generated in these places. The black lines mark the geometry walls.

Each 3D voxel has six neighboring voxels (which we call border voxels). After calculation of the distance field, each border voxel is marked with the maximal distance found in the distance field incremented by one. This ensures that an outdoor area will typically be created initially. Indoor areas will be separated from it later, during the portal-search phase. Euclidean-metrics usage involves the calculation of a distance between the center of each voxel and the nearest part of the input geometry.

The final step in this phase is smoothing the local maxima in a distance field. Each of these maxima is a source of new area and, as a consequence, a source of portals separating these areas. Smoothing the local maxima results in combining many sets of voxels with high distance to the geometry into one set with a common distance. As a result, they do not generate unnecessary and often erroneous portals in narrow passages.

3.2. Watershed portal search algorithm

At this stage of the algorithm, it is possible to use manually inserted portals provided with input geometry. Such portals should be provided in places that are problematic for the automatic-generation algorithm. Manual portals are voxelized; that is, each voxel crossed by a manual portal and not intersected by the geometry is marked as a portal voxel. Voxels intersected by the geometry remain unchanged.

Later in the algorithm, portal voxels are treated in the same manner as voxels that intersect walls in the scene. Also, it should be noted that the described 3D extension of the Watershed algorithm is referring to the maxima, while in the original Watershed algorithm [6, 14], minima are used (in our example, minima can be used by inverting the distance sign).

Given the distance field computed in the previous stage, the greatest value in the distance field is found. We denote it as $curDist$ and two operations are performed in a loop: expanding the existing areas and creating new areas. After these two

operations, the *curDist* value is decremented by one, and the whole process starts again (as long as any of these operations can possibly have some effect, or *curDist* is greater than zero).

The procedure of expansion causes all already-found areas to be enlarged by the adjacent voxels that are distanced from the geometry by a *curDist* distance. The procedure is continued until there is no longer any neighboring voxels to join, or if an attempt is made to join a voxel that is already in another area. If an attempt to enlarge an area by a voxel assigned to another area is made, such a voxel becomes a starting point for portal generation – the collision stage is entered. During the collision stage, the currently processed area is restored to its original state (before failing the expansion step). The center of the colliding voxel is the starting point where we begin the construction of the portal. After calculating the orientation of the portal and its size, all voxels that are intersected by recently created portals are marked as portal voxels. As a result, collision is eliminated by separating the two colliding areas by a portal.

If the voxelized portal overlaps with voxels already assigned to some other area, the construction of a portal fails, and the conflicting areas are merged. Similarly, if there is a collision with another portal, building fails, and the colliding areas are merged. After insertion of a portal, the current step of expanding the area is continued. The expanding operation ends when there is no new portal and no area that can be enlarged by new voxels.

When the area-expanding stage of the algorithm is finished, voxels that are not yet in any area and have a distance equal to *curDist* participate in the creation of new areas. The first of the remaining voxels creates a new area that consists only of this voxel. The new area is then expanded (as long as it is possible). After all of the voxels that could be joined to the area have been added, the next voxel not assigned to any area is chosen, and the procedure is repeated. Checking collisions is unnecessary, because all possible collisions have been found in the earlier stages; so, they cannot occur at this stage. New areas are created as long as there are voxels with distance *curDist* that are not yet assigned to any area.

After the end of the calculations, areas are assigned to manually chosen portals. If for a manually chosen portal there are more than two areas adjacent to it, then the construction process fails and the manual portals need to be corrected. In the end, the algorithm generates a graph of areas and portals where the areas become nodes and portals become edges connecting these nodes.

3.3. Preparation of structures for usage in real-time applications

The result of the portal-finding algorithm is a set of voxel areas and a set of portals connecting them, where areas are represented as a set of voxels and portals are represented by rectangles. For proper integration with a beam-tracing engine, there needs to be a method that allows us to determine in which area a given point is loca-

ted. Then, information about which objects are contained within a particular area is required.

A structure representing an area (used for determining if a given point is inside) is created by calculating a set of cuboids completely covering the voxel area. Additionally, input geometry triangles and portals connected to the area are stored (they form a hull of the area). A point inside any of these cuboids is certainly within the area; otherwise, all of the cuboids from this area that are in a distance smaller than the size of one voxel from this point are examined. This involves connecting the examined point with the center of the cuboid by a line segment and checking if this line segment intersects any of the stored triangles. If, for any cuboid, there is a line not intersecting the hull of the scene, then the point is inside the area.

4. Details

4.1. Smoothing the local maxima

The procedure of distance-field generation can create local maxima in the distance field. Local maxima can occur in voxels positioned further from the area hull than its neighbors. Such maxima can generate erroneous results, because each local maximum is the source of a new area and potentially some portals. To eliminate erroneous portals created in the proximity of small corners, a local maxima-smoothing procedure is performed – especially if there are many walls not aligned to the coordinate system of the map (see Fig. 4).

For two given distances A and B (where $A > B$), the local maximum is defined as a set of connected voxels for which distance values are within range $[B, A]$. None of these voxels can be connected to any voxel that has a distance value greater than A. The smoothing procedure sets distance values for all voxels in this set to value A. Input parameter $minSmoothSize$ is used to calculate the value of B as $A{*}minSmoothSize$, where $0 < minSmoothSize \leqslant 1$. When $minSmoothSize = 1$, the result is the same as when no smoothing was applied. Setting a value of distance for all voxels in this set makes all local maxima (for given distance A and calculated distance B) inside this set merged into one maximum with a larger volume. The entire step is designed to assign the same value of all of the local maxima for each value of A (i.e., for each distance value in a distance field). If the size of the smoothed maximum is set too high ($minSmoothSize$ is very close to 0), this can result in smoothing all of the small spaces (as shown in Figure 3). This selected area is a local maximum, but it can have smaller local maxima inside. If $minSmoothSize$ is very small, such small local maxima will be flattened (overwritten with one value). Because the algorithm separates local maxima in a distance field with portals, there will be no portals between small rooms if these small maxima are flattened. On the other hand, too small a value may not smooth all of the peaks; in effect, erroneous portals could be generated.

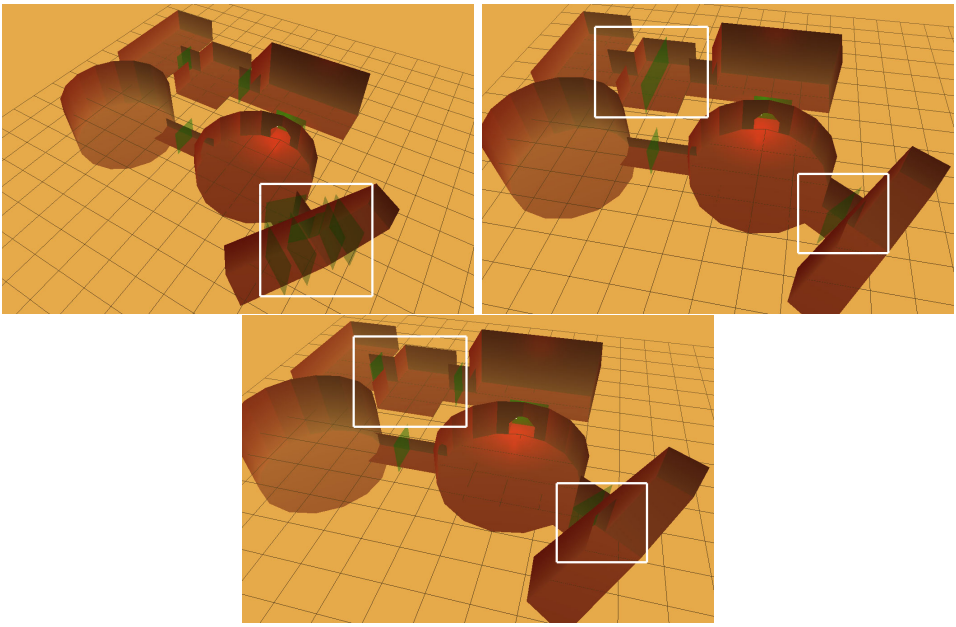

Figure 3. On the top left, a situation in which the smoothing procedure was not applied: unwanted portals appear near diagonal wall pairs (due to local maxima appearing in such places). On the top right, a situation with an incorrectly chosen smoothing size – *minSmoothSize* is too large – a portal in the middle of the room is created instead of two portals in corridors between the rooms, but no incorrect portals are created near the diagonal wall pairs. At the bottom, a situation with a properly adjusted smoothing size: small maxima are successfully eliminated, and small rooms are correctly separated by portals.

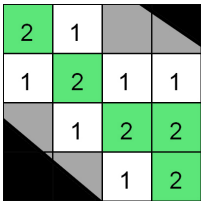

Figure 4. Fragment of geometry containing two walls positioned diagonally (black) and a distance field generated in this fragment. Gray voxels are intersected by geometry and do not participate in portal generation. Local maxima are marked in green. Each of the maximum is the source of a new area, and as a consequence, a potential source of portals separating these areas. Given such conditions, many incorrect portals would be generated. Smoothing a distance field (in this case – resulting in merging voxels with a distance value between 1 and 2 into one set with a distance equal to 2) solves the problem. For simplicity, a Manhattan distance field is presented, but the same problem arises for the Euclidean metric.

4.2. Portal placement

Each collision between two areas is a potential source of a new portal. Computation of the point for which portal building starts is conducted in a separate pass. This pass consists of alternately expanding both areas with one voxel. As a result, when a collision between two areas occurs, the collision voxel is located in the middle of the shortest path between these areas (there may be other such paths [multiple shortest paths with proximity to a voxel] that are omitted, the first found is taken). The center of a colliding voxel is the starting point from which the construction of the portal should begin. With a larger voxel size creating a denser voxel grid, the possibility increases that the initial point for portal generation is not optimal.

4.3. Elimination of portals that are large compared to area size

When an area is created, the distance value of a voxel from which the area-construction process started is saved as $initialDistance$. This value is the greatest distance values of all of the voxels in this area. Later, when two areas are found to collide and the portal building process starts, the higher of the $initialDistance$ values from these areas are computed and scaled by a factor $f(0 < f \leqslant 1)$, introduced as an input parameter to the algorithm. The value computed in this way is denoted as $maxAreaSize$. Then, the distance value of the voxel from which the portal building process started, denoted as $portalSize$, is compared to the previously computed value of $maxAreaSize$. The $initialDistance$ value for each area is treated as a rough approximation of area size, because the exact value is hard to compute before portal-system building is completed.

If $portalSize$ is bigger than $maxAreaSize$, the portal will not be created. This prevents the creation of portals as big as a cross-section of the colliding areas. Setting the scale factor to a small value can result in removing many well-placed portals (default value is 0.9). When two areas are being merged, the smaller area is always joined with the bigger one; thus, the saved $initialDistance$ value for the merged area is still valid after this step.

4.4. Portal orientation

Calculating the orientation of a portal is performed differently than in the Haumont method [13]. The shortest paths, defined as a set of connected voxels (see Fig. 5) between the collision point (voxel) and the center voxels of both areas are calculated. The direction vector of the path is calculated as the average of all of the line segments connecting two voxels on this path. The initial normal of a portal is calculated as the average of the direction vector of the path to the larger area and a reversed direction vector of a path to the smaller area. There can be more than one different path with the shortest length; but for maximum performance, the first found is used.

The initial orientation of the portal is adjusted to the geometry. This is especially important in scenes with walls and corridors that are not aligned to the axes of the coordinate system. It is frequently the case that the initial orientation of the portal is

incorrect. When the voxel size is small compared to the area size, the portal is placed almost correctly.

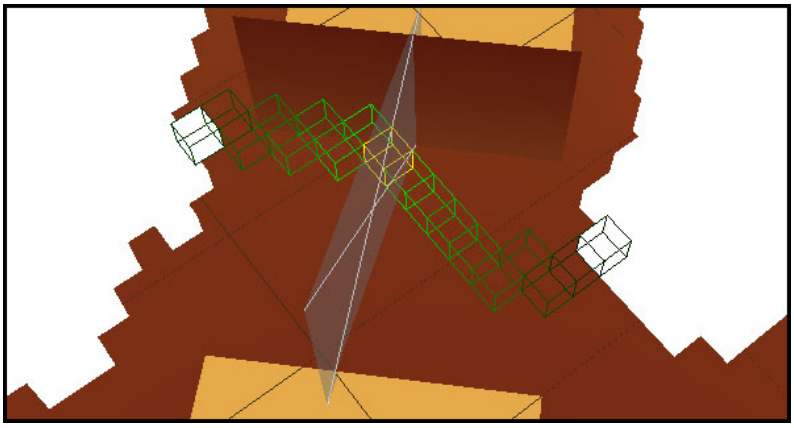

Figure 5. Two voxel areas (white) are colliding in a voxel marked in yellow. A portal built from this voxel is shown in gray. This portal does not have the correct orientation yet. From the voxel in which a collision between two areas was found, the shortest voxel path is created to each of the colliding areas (green voxels). Note the importance of ensuring that these paths are at least a few voxels in length. If the paths were too short, the directions calculated from them would not be accurate. Smoothing the distance field helps in correcting portal placement.

The left side of Figure 6 represents a portal generated in a corridor. The walls of the corridor are not aligned to the coordinate system, which prevents the portal from being oriented properly. Ideally, the portal should be aligned to the corridor walls. For simple geometry, the following procedure is accurate enough to achieve good results. Our solution is designed for audio, and this is why complex geometries are never considered. Four rays lying in a portal plane (and perpendicular to the portal normal) are calculated every 90 degrees from the center of the portal. Each of these rays hits some triangle, and the normals of these triangles are stored as $(N1, N2, N3, N4)$. Using these stored normals, better alignment of a portal can be calculated. If the normals in pair $N1$ and $N3$ have exactly the opposite direction, the normals in pair $N2$ and $N4$ have the opposite direction, and (additionally) these directions are not parallel (see Fig. 6), then the normal calculated as a cross product of these directions is saved in a set named $PortalCorrectionNormals$. This procedure is repeated ten times; each time, the rays are rotated clockwise by 4.5 degrees (the dashed lines represent the second rotation). If the number of normals saved in $PortalCorrectionNormals$ is greater than five, then a new normal of the portal is calculated as the average of normals in $PortalCorrectionNormals$. This method has proven to be precise and fast enough in all of the tested scenes.

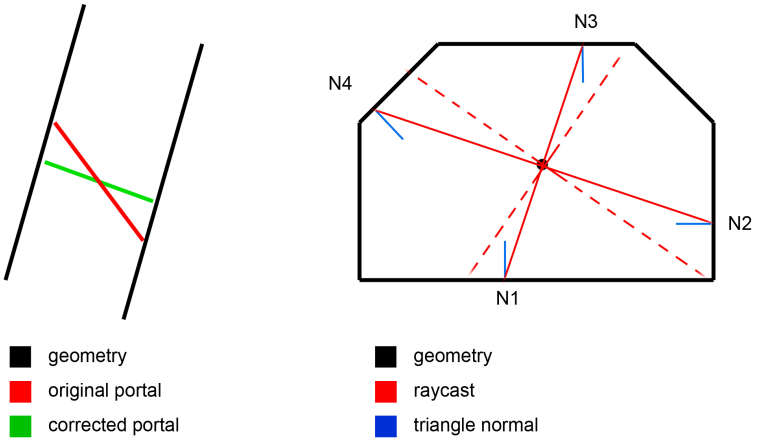

Figure 6. On the left – correcting the orientation of a portal; on the right – cross-section of passing.

4.5. Portal shape

A portal is represented as a rectangle. After calculating the starting point and determining the correct orientation of the portal, the portal-voxelization step follows. All empty voxels (e.g., voxels not intersecting any wall in the scene) that intersect the portal are marked as portal voxels. Then, the centers of these voxels are projected onto the plane of this portal, and the rectangular bound of these projections is calculated. The calculated rectangle is the geometrical shape of the portal. The center of the portal is adjusted to be located in the middle of the rectangle.

After calculation of an initial portal shape, portal fitting to the surrounding geometry follows. Starting from the center of the portal, rays in the plane of the portal are cast to the surrounding geometry. The collision points between rays and triangles surrounding the portal are part of a set of points that approximately describe the exact shape of the portal. Additionally, each triangle visible through the raycasts is processed by computing the endpoints of its intersection with the portal plane. For endpoints where the line connecting them with the middle of the portal does not intersect the geometry, they are additionally included in the set that approximates the portal shape. If the line does not intersect the geometry, it is ignored in further calculations. If there are several such lines, the whole portal is ignored. For a given set of points (guaranteed to be in one plane), the best-possible bounding rectangle is generated. A rectangle computed this way becomes a portal if and only if it has a smaller area than the portal generated by using voxels, because the existence of a smaller portal indicates its better adjustment to the input geometry.

4.6. Bounding geometry for areas and finding an area for a point

The structure representing an area used for determining if a given point is inside is created in the following way: A set of cuboids completely covering the voxel area is calculated. The already-calculated distance field is used while generating the cuboids. This allows for more optimal results (fewer cuboids) than generating from a random voxel. From a set of not-yet-covered voxels, one with the greatest distance value is searched for. After that, construction of the limiting cuboid is started. This cuboid is expanded along each axis for as long as possible (i.e., not outside the covered area). Next, all voxels covered by the cuboid are removed from the active set, and the cuboid is added to the list of cuboids overlapping the area. This procedure is repeated until all voxels within the area are covered (i.e., the set of active voxels is empty). Cuboids and triangles are stored in a continuous memory container.

For each area, triangles that intersect any of the previously numbered cuboids enlarged by a voxel size from each side are also stored. Checking whether the point is in the area is based on checking whether any of the cuboids contain the point. If so, it is certainly within the area. In the other case, for each cuboid for which the distance from the given point is smaller than the voxel size, it is determined whether a segment that starts at the given point and ends at the center of the cuboid intersects any of the stored triangles. If there is no collision with the geometry, the point lies inside the area, and the test can be stopped. If testing of all of the cuboids results in a collision, the point is outside the area (see Fig. 7). Time complexity is linearly dependent on the number of cuboids and mesh triangles assigned to each area. Space complexity is linearly dependent on the number of voxels.

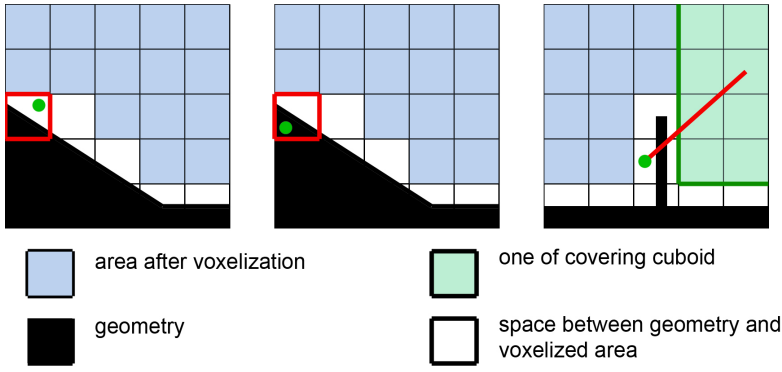

Figure 7. On the left – a case where a point is inside a given area. In the middle – a point is in the same voxel but is actually positioned outside the area (there is no line connecting this point with any cuboid in the area that does not intersect the scene geometry). On the right – a point is inside the area even though the line connecting it with one of the area cuboids intersects the scene geometry. This situation cannot occur if a special type of voxelization is used [14] or when thin-wall usage in the input geometry is forbidden.

5. Scenes and results

The algorithm was tested on three sets of scenes (see Table 1), featuring various sizes and characteristics. All tests were run on a computer with an Intel Core 2 Duo 2.20 GHz processor and 2GB of RAM. The results presented in this chapter were generated using a single-threaded version of the application. The results are divided into two different scenarios. The first scenario focuses on memory usage and overall scene statistics, such as the number of triangles, number of created voxels, etc. The second scenario provides metrics for comparing the quality of a result between automatic and manual portal placement.

5.1. Time efficiency and memory usage tests

Table 1
Complexity of scenes.

	scene 1	scene 2	scene 3
size [m]	91×11×82	65×15×54	130×25×110
number of triangles	540	386	759

Figure 8 shows the scenes used for tests whose results are provided in Tables 2 and 3. "Memory" in Table 2 refers to the total amount of memory used by the algorithm (not only for voxel storage but also all intermediate structures). It is clear that the memory requirements increase rapidly with a decreasing voxel size – with $O(n^3)$, where n is the inverse of the coefficient by which the voxel size was scaled. "Real-time memory" refers to the amount of memory used by the portal system for structures in a real-time application. Differences depending on the size of the voxel are due to the method used to define the areas: the smaller the voxel size, the more cuboids are needed to cover the area (middle and bottom of Figure 9). For the first scene, this size is constant, because the map is made up of the same cuboids with no slanted walls, so these areas are covered by the same number of cuboids regardless of the size of the voxel (top of Figure 9).

Building structures for real-time portal usage with large scenes and small voxel sizes can easily consume the biggest portion of time (scene 3), which suggests that there are still many places for possible optimization.

5.2. Measuring correctness of automatic portal generation

In order to automatically test the correctness of automatic portal generation, the following measure was implemented and applied on various scenes (see Figs. 10, 11, and 12). We refer to it as the Correctness Measure. With two scenes (one with portals inserted manually and the other with portals computed by our algorithm), we can provide one value indicating the correctness of the generated portals.

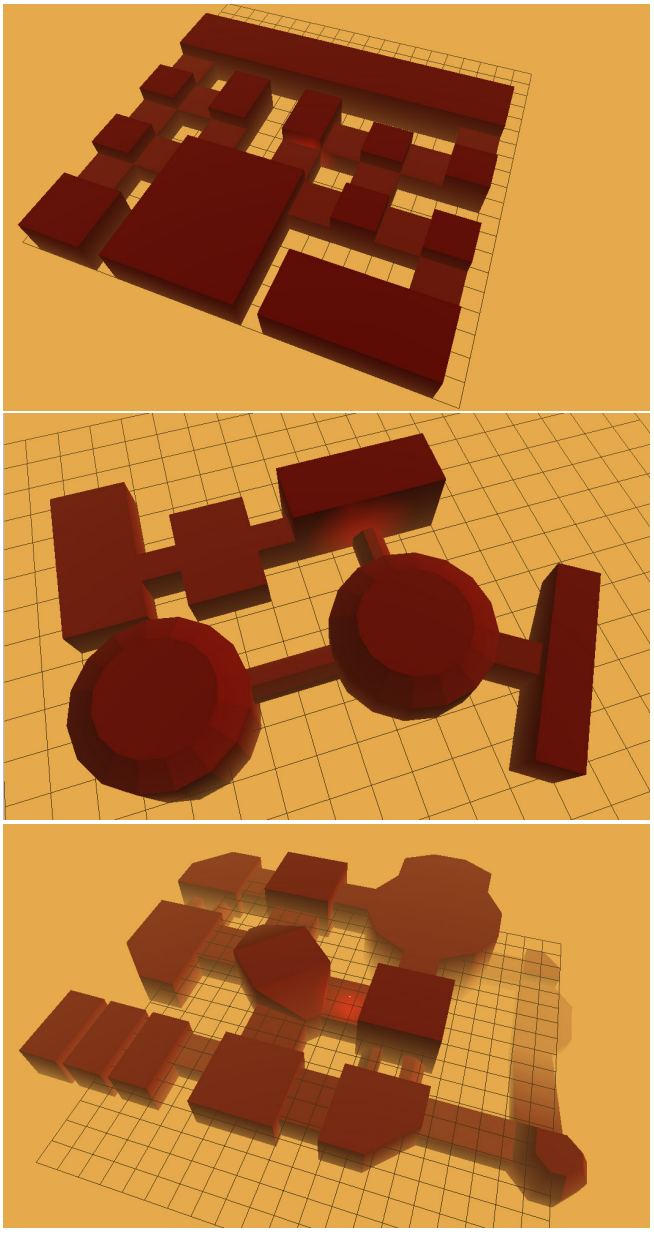

Figure 8. The scene on top is composed of cuboids only. The middle scene contains a variety of shapes and one room positioned diagonally to coordinate system axes. The last scene combines rooms of cuboid shape and more-complex rooms all connected with various corridors. All presented scenes have been prepared specifically for the task of portal generation; i.e., all small geometric elements have been removed.

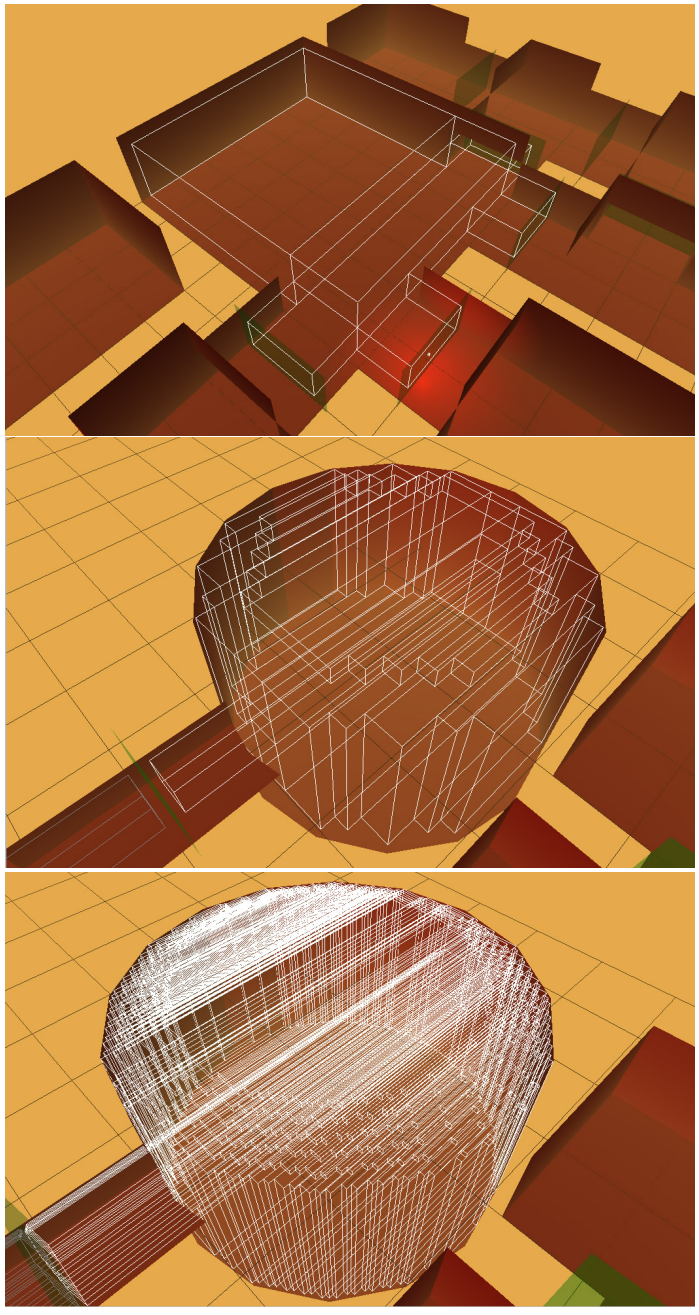

Figure 9. The smaller the size of the voxels, the larger number of cuboids necessary to cover the irregular areas (middle and bottom).

Table 2

Memory usage for various voxel sizes for scenes from Figure 8.

Scene 1

	voxel = 1m	voxel = 0.5m	voxel = 0.25m
memory for offline precalculation	1,6 MB	13,5 MB	106,3 MB
memory for real time application use	38 kB	38 kB	38 kB
number of voxels	93×13×84 = 101 556	183×23×164 = 690 276	363×43×324 = 5057 316

Scene 2

	voxel = 1m	voxel = 0.5m	voxel = 0.25m
memory for offline precalculation	0,9 MB	7,3 MB	57,6 MB
memory for real time application use	23 kB	26 kB	40 kB
number of voxels	67×17×56 = 63 784	130×32×109 = 453 440	256×62×216 = 3428 352

Scene 3

	voxel = 1m	voxel = 0.5m	voxel = 0.25m
memory for offline precalculation	6,6 MB	50,4 MB	411 MB
memory for real time application use	50 kB	64 kB	101 kB
number of voxels	132×27×112 = 399 168	261×51×221 = 2941 731	519×99×440 = 22 607 640

Let us denote a set of all vertices constituting manually created portals as v^m and a set of all vertices constituting automatically created portals as v^a. First, we add up the distance between each pair of vertices v^m. Then, we add up the distance between each vertex of each automatically found portal and each vertex of manually created portal. The quotient of these two numbers is the measure of accuracy of the generated portals. We assume that the manually created portals are in optimal positions; therefore, optimally positioned automatic portals will yield a value of 1 because they will be in the same positions as the manually placed portals. Any differences in portal placement will decrease the value of the correctness measure (because the distance between the vertices of manually and automatically placed portals will increase).

$$CorrectnessMeasure = \frac{\sum_{u,v \in v^m} |u - v|}{\sum_{\substack{u \in v^m \\ v \in v^a}} |u - v|} \,. \tag{1}$$

With the Correctness Measure close to 1, the automatically found portals are more similar to the portals created manually. With a value of 0.2, five-times-too-many

Table 3

Calculation times for various voxel sizes. The time given for voxelization and distance-field computation also includes time spent on smoothing the local maxima. Time named "portals" is the time spent on seeking portals.

Scene 1

	voxel = 1m	voxel = 0.5m	voxel = 0.25m
voxelization + distance field	0.01 s	0.4 s	2.6 s
portals	0.3 s	2.5 s	20.5 s
structures for real-time computation	0.05 s	0.04 s	3.5 s
total	0.5 s	3.5 s	27 s

Scene 2

	voxel = 1m	voxel = 0.5m	voxel = 0.25m
voxelization + distance field	0.04 s	0.3 s	2.4 s
portals	0.07 s	0.3 s	3.6 s
structures for real-time computation	0.04 s	0.4 s	4.2 s
total	0.16 s	1 s	10.7 s

Scene 3

	voxel = 1m	voxel = 0.5m	voxel = 0.25m
voxelization + distance field	0.27 s	2.6 s	30 s
portals	1.3 s	9.5 s	3 min
structures for real-time computation	0.4 s	3.5 s	11 min
total	2 s	16.4 s	14.5 min

portals were generated, and with values above 1, too few portals were generated. The proposed measure provides sufficient evaluation of the generated portals. The results for ten different maps for the different metrics used to generate the distance field are presented in Tables 4 and 5.

A Correctness Measure with a 0 score means that no automatic portals were generated. For test scene number 4 (see Fig. 10), the lack of generated portals is caused by the fact that all of the rooms in the scene have the same height, and the algorithm is unable to cope with such scenes (more explanations below). The absence of generated portals may also happen due to the large size of the voxel. In this case, some small passages between rooms cannot be detected, or the smoothing step removes the information about such passages.

Large and small values of the Correctness Measure can be observed most frequently when the number of automatically generated portals is very different from the

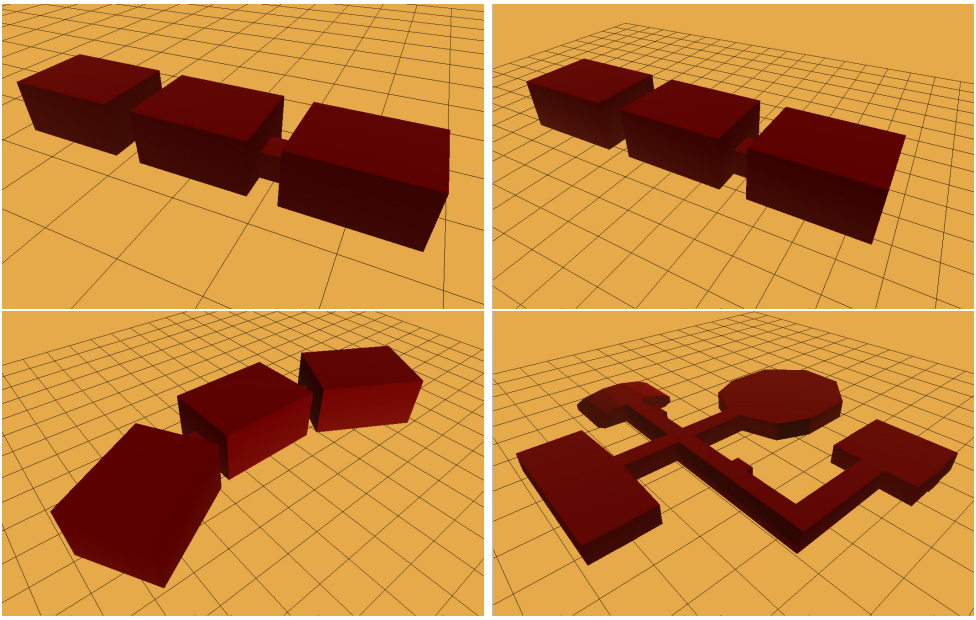

Figure 10. Scenes with various levels of complexity and different architectural properties used in tests. In order from the top: Scene 1 (on the left) and 2 (on the right) are similar and composed of cuboids with different size. Scene 3 (left, bottom) is made with diagonal passages and rooms. Scene 4 has four rooms and a corridor; each room and corridor have the same height.

Table 4

Manhattan metrics tests (man. – manual, calc. – calculated, * means that the computation did not finish within the given time constraint of one hour). Test scenes are presented in Figures 10, 11 and 12.

test sc.	man. portals	voxel = 0.5	calc. portals	voxel = 0.25	calc. portals	voxel = 0.1	calc. portals
1	2	1.007	2	1.004	2	0.991	2
2	2	1.003	2	1.002	2	1.001	2
3	2	0.969	2	0.967	2	0.965	2
4	4	0	0	0	0	0	0
5	14	0.273	14	1.000	14	1.117	15
6	3	0	0	0.363	1	1.015	3
7	11	0.538	6	0.639	7	1.256	14
8	9	0.161	1	0.786	7	1.561	14
9	39	0.815	28	0.908	35	error*	—
10	21	0.308	6	0.431	9	0.643	13

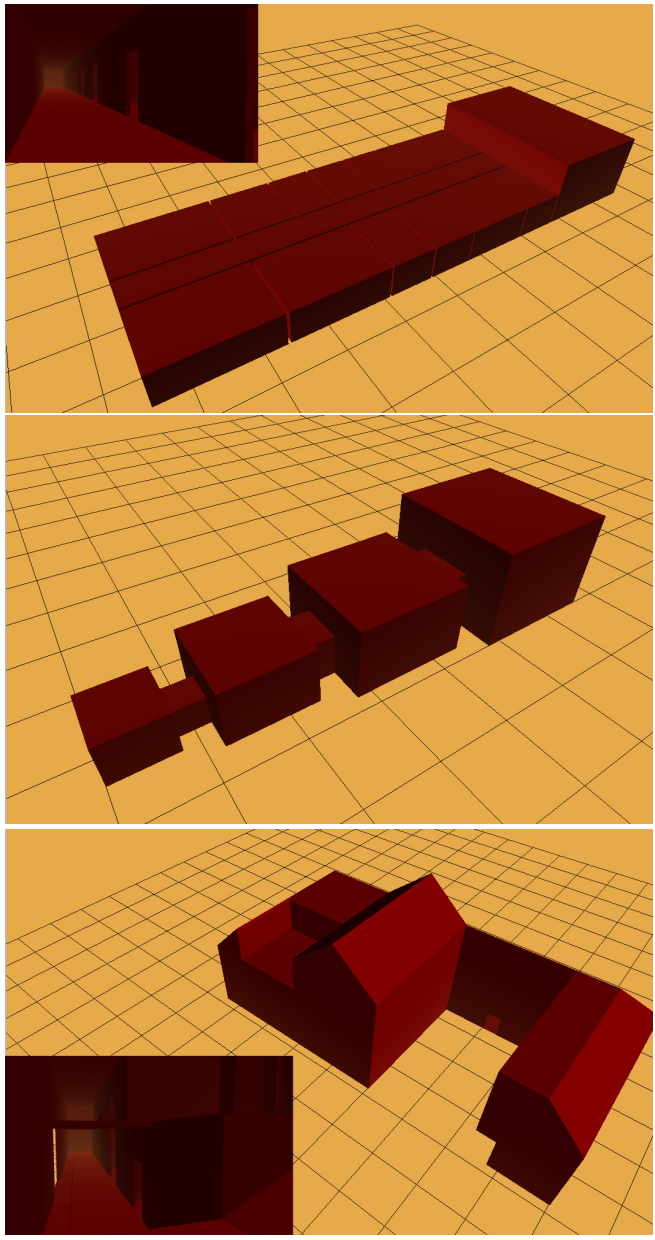

Figure 11. Scenes with various levels of complexity and different architectural properties used in tests. In order from the top: Scene 5 is made from cuboid-shaped rooms separated by thin walls and doors. Scene 6 is made from rooms shaped like cuboids of different sizes. Scene 7 is a two-story house with rooms connected by various corridors (sloped, horizontal, and verticals).

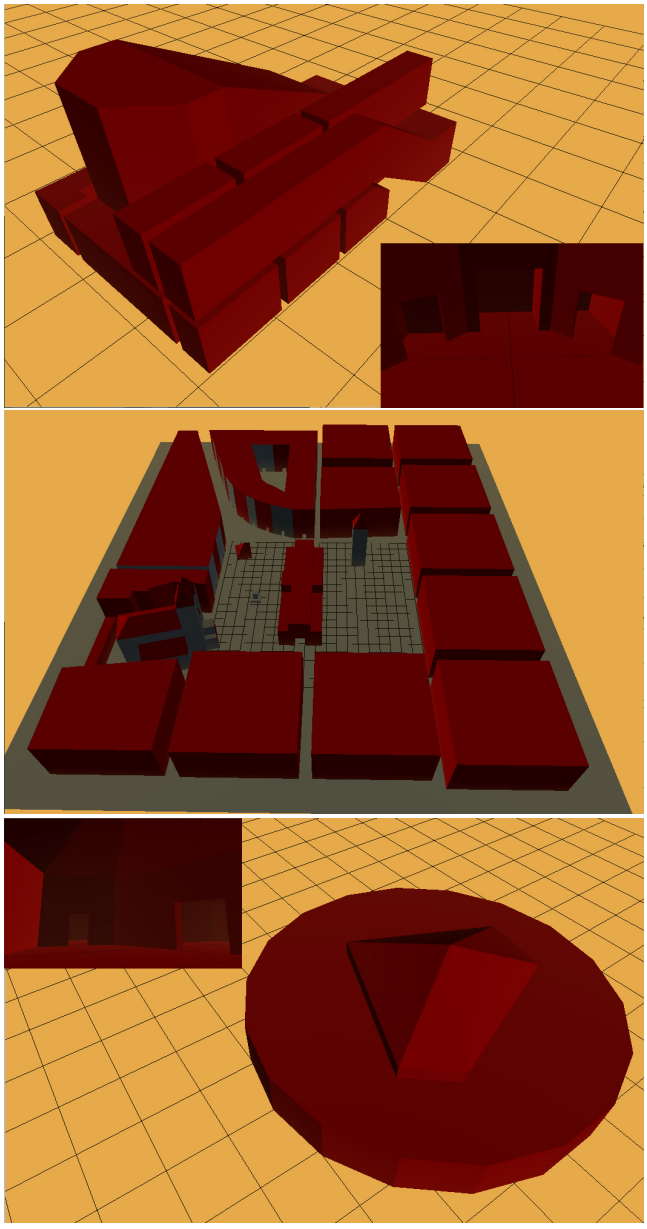

Figure 12. Scenes with various level of complexity and different architectural properties used in tests. In order from the top: Scene 8 has one floor with many rooms of different shapes. The rooms are connected by many passages (sloped and horizontal). Scene 9 is a model of Krakow's Main Square. This is the scene with the highest volume. Some of the buildings have simple interiors where portals should be built. Scene 10 is similar in its characteristics to Scene 8.

Table 5

Euclidean metric tests (man. – manual, calc. – calculated)

test sc.	man. portals	voxel = 0.5	calc. portals	voxel = 0.25	calc. portals	voxel = 0.1	calc. portals
1	2	1.007	2	1.004	2	0.991	2
2	2	1.003	2	1.002	2	1.001	2
3	2	0.969	2	0.967	2	0.965	2
4	4	0	0	0	0	0	0
5	14	0	0	1	14	1.001	14
6	3	0	0	0.363	1	1.018	3
7	11	0.524	5	0.639	7	1.025	12
8	9	0	0	0.595	5	0.905	8
9	39	0.849	33	0.850	32	0.850	32
10	21	0.308	6	0.665	12	0.708	15

number of portals inserted manually. With the same number of portals, the measure is in most cases very close to 1.

It is worth mentioning that, in the case of the Euclidean metric (for almost every scene), the smaller the voxel size, the closer the portal's correctness value is to 1. However, this is not the case when the Manhattan metric is used. So, in this particular implementation, the Euclidean metric is more predictable in the sense that voxel-size reduction results in better portal placement.

In the case of more-complex scenes (7 – see Figs. 11, 8, 9 – see Fig. 12), it can be seen that decreasing voxel size results in the creation of more portals. The reason is that these scenes contain many small rooms often separated by thin passages that are better separated when smaller voxels are used (see Fig. 13).

Another reason is that smaller voxels generate more-correct portals for scenes with thin passages between rooms. It is more probable that such portals will not be rejected during validation (see Fig. 14).

Very simple scenes (1, 2, 3 – see Fig. 10) have passages between rooms so wide and long that the algorithm can find the correct portals even with the largest-tested size of a voxel. In Scene 6 (see Fig. 11), it can be seen that the algorithm finds portals in smaller and smaller passages, along with the decreasing size of the voxels.

5.3. Issues encountered during the portal-generation process

As mentioned before, no portals are generated in the areas where rooms and corridors connected to them have the same height, and additionally, the width of the rooms and corridors is greater than their height. In this case, local maxima will not be generated, because the minimum distance between the center of the room to the surrounding geometry is always smaller to the ceiling or floor than to far-away distant walls (Fig. 15).

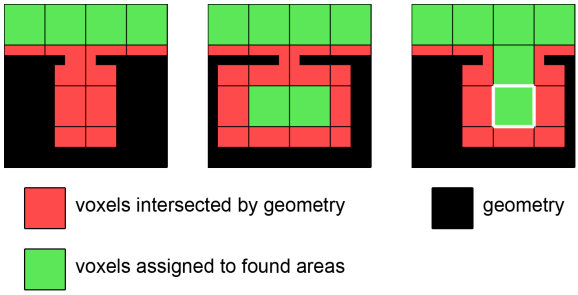

Figure 13. Three cases when a portal separating a small area is not found. This can happen if an inadequate voxel size is used. On the left, a small area is completely covered with voxels intersected by the geometry and, in effect, a room is omitted by the algorithm. In the middle, the area is found, but it is separated from the other area by voxels intersected by the geometry. No portal will be built, but a small area without pa ortal connection with the rest of the scene will be generated. On the right, an area and a passage are found; however, in the case of the Manhattan metric, no local maximum exists in this area, and a portal will not be created. In the case of the Euclidean metric, there might be a one-voxel-size local maximum found (white rectangle), and portal construction will succeed. For this reason, the Euclidean metric is more accurate.

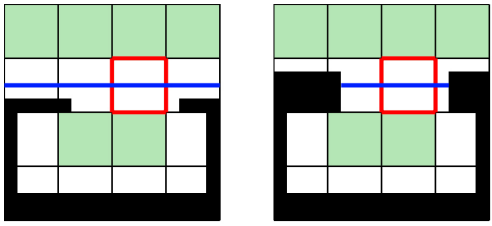

Figure 14. The red rectangle marks a collision voxel between two areas (areas are marked in green). A portal-building procedure starts in the center of a collision voxel. On the left side, the generated portal is incorrect because the passage is very short compared to the voxel size. Portal shape computed only from voxel representation will leave an empty space between a portal and the geometry. On the other hand, a portal generated using exact geometry information will be too big, because raycasts can miss the geometry. On the right side, a similar geometry but with a long passage is presented. In this case, a correct portal is created.

Similar cases occur for long, bent corridors with a constant cross-section. In such a geometry, one large maximum will be generated. In effect, portals which separate corridor sections between bent walls will not be created.

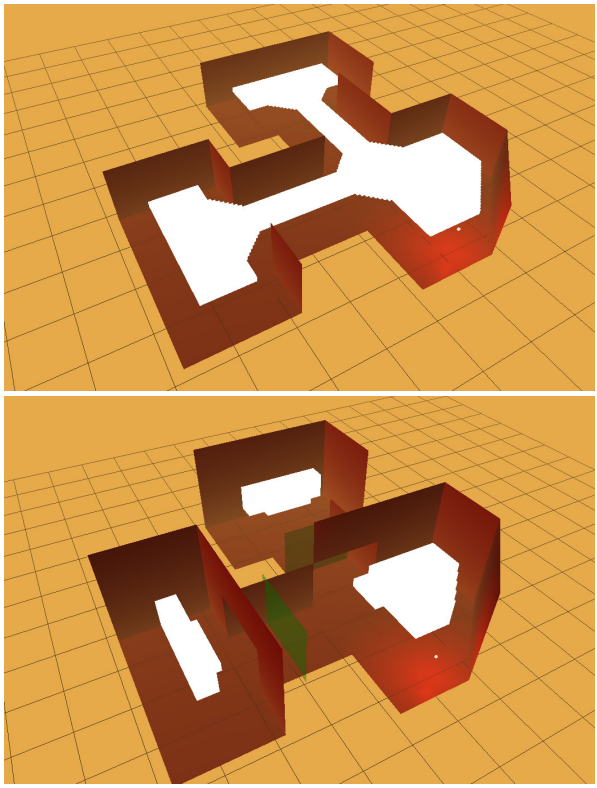

Figure 15. White indicates local maxima. On the top, a single maximum is found in a scene in which all rooms and corridors have the same height. No other portals can be created. On the bottom, the scene has been modified by raising the ceiling in the corridors, so local maxima are formed there, and as a consequence, portals between these spaces are created (green).

6. Discussion over implemented solutions

In order to improve the algorithm's time efficiency, it is possible to implement multiple optimizations. We did not decide to use any hierarchical structure for distance-field computation. The amount of memory required (discussed in the method – each voxel takes 4 bytes) becomes a problem when the voxel size is small. Haumont states that the hierarchical structure works well only for the simplest scenes. On more-complex scenes, a tradeoff between memory-requirement reduction and the significantly longer time required to navigate through the hierarchical structure has to be made [14].

Even with a hierarchical structure applied, memory requirements can still be a problem. For example, if such a structure can optimize memory usage by a factor of 64, either the size of the voxel can be four times smaller or the scene can be four times

larger. This is still far from satisfying, if we take into consideration the requirements of large scenes. For comparison, the best memory-optimizing factor achieved earlier is 30 [13] and, therefore, will not allow for a reduction of a voxel size by a factor of 4.

The only feasible solution seems to be analogous to the Umbra system [28]) dividing the scene into independent parts. Then, each of the parts can be processed separately, and the results could be merged together.

This is the reason for the use of large scenes. The algorithm is always used with a large voxel size, compared to the visibility applications of portals (where voxels must be very small). This obviously imposes limitations on the granularity of the geometry, but sound simulations are much less sensitive in this respect than light (due to the length of sound waves). With such a voxel size, working with holes in walls of the size of a voxel is problematic. Sound should be able to pass freely through them. This, however, will not be taken into account in our model. Portals inserted manually do not solve the mentioned issue because, after the voxelization step, such a hole will be fully covered by the 'blocking' voxels (i.e., crossed by the geometry) of the surrounding walls. This would require writing special code for this case. At the moment, running an algorithm with a smaller voxel size will take such a hole into account. Despite the large voxel size, the point's position needs to be checked for accuracy with respect to the input geometry. Using only voxel representation in this situation is not accurate enough for this task. Video games restrict possible camera movement close to the room walls to avoid erroneously showing parts of the scene that should be obstructed. This restriction is achieved by imposing a minimal distance between the camera and any wall. Let's assume that a voxel size is smaller than this distance. If the test point is no closer than the voxel size to any wall such a test could be based on data generated only using voxelized representation of the generated areas.

An additional difficulty is that the generated areas are not necessarily convex. A solid outline created by the well-known marching cube algorithm would have multiple triangles. A more-sophisticated algorithm is needed. The solution adopted by us is easy to implement, but it has the following drawbacks: the need to iterate over many cuboids overlapping the area, and the fact that, along with decreasing voxel size, the number of cuboids in an area increases. Using large voxels is also a problem in the case of locating portals in narrow passages. Such a passage will be found, but the portal will not be properly aligned to it (since its construction starts from the center of the initial voxel, and it is unlikely that this center is in the center of the passage). Due to implementation simplicity and efficiency, the Manhattan distance metric was used to calculate the distance field. However, an Euclidean metric was added later, and it proved to be less sensitive to $curDist$ changes. Both metrics have linear efficiency $k\dot{O}(n)$, but k is much lower when the Manhattan metric is applied.

Using the Manhattan metric does not cause performance problems when larger scenes are used. In contrast, the time of brute-force distance-field generation with the Euclidean metric becomes unacceptable when the size of a scene is large. Portals generated with Manhattan metrics are correct, but tests show that the Euclidean metric is more accurate. The current solution does not recognize which voxels are in

the part of the geometry that are inaccessible by users (e.g., filling a wall), and which are outside. To be able to recognize such voxels, triangle normals must be assigned consistently. For example, normals of triangles should always point to the outside of the geometry (that is, towards the empty space) where potential player movement is impossible. This assumption further implies the absence of thin walls built from triangles lying in a single plane, because such walls can not have the normal "on the outside" on both sides. Alternatively, such triangles would have to be marked as two-sided. Voxels in unreachable space are not recognized, which causes portals and areas to be computed and created in such space. Their presence does not affect the outcome (beyond slowing down the calculations), because they are completely isolated from the correct areas. On the other hand, they unnecessarily occupy memory.

7. Conclusions

This paper presents an algorithm for the automatic construction of portals so that the resulting area-portal graph is closely linked to the geometry structure. The manner of its use in practice is also described.

Our improvements to Haumont's approach [13] conclude that it is necessary to increase the number of voxels (briefly estimating about 10x-40x larger voxels) and use the Manhattan metric to speed up calculation. Furthermore, the methods used for searching for collision points and placing portals have been developed to better fit the geometry. Applying this method ensures that the automatically generated portals are correct and suitable for practical use in the sound engine RAYA.

Future work in this area would focus on enhancing the voxelization with capabilities of detecting unreachable areas. Another possible improvement would be to develop a method for the correct localization of portals in short passages when voxels size is bigger than the length of the passage. A number of optimizations (especially in terms of memory requirements) are also possible, since it seems to be a more-important limitation than time efficiency. For this purpose, dividing the scene into fragments and combining the results could be considered. The method for searching the area in which a given point is contained can also be improved. The described solution is relatively simple to implement, but there is still the opportunity for improvement (especially in the case of memory requirements).

Acknowledgements

This work was supported by the National Center for Research and Development under INNOTECH-K1/IN1/50/159658/NCBR/12 grant. We would like to express our gratitude to Katarzyna Baruch, Paulina Bugiel, Krzysztof Gabis, Ireneusz Gawlik, Bartłomiej Miga, Michał Radziszewski and Teyon for our work together during development of RAYA sound engine and for their friendship.

8. Appendix

Below is presented a pseudo-code of the algorithm for automatic portal placement. Its input is the smoothed discrete distance field (sections 3.1, 4.1).

```
function:
build_portals
input:
distance_field
output:
computed areas, portals
{
    areas = {}
    portals = {}
    for (current_dist = distance_field.maximum_distance;
    current_dist > 0; current_dist--)
    {
        expand_areas(areas, portals)
        create_new_areas(distance_field, areas, current_dist)
        // at this moment there are no unassigned voxels
        with distance current_dist
    }
}

function:
expand_areas
input:
areas, portals
output:
areas
{
    for area: areas
    {
        previous_state = area.save_state()
        for each voxel V adjacent to area:
        {
            if V.assigned_to_area = null
            {
                area.voxels += V
                V.assigned_to_area = area
            }
            else if V.assigned_to_area != area
            {
                area.restore_previous_state( previous_state )
                if failed build_portal( V.assigned_to_area,
          area, portals )
                {
                    merge_areas( V.assigned_to_area, area )
                }
            }
        }
    }
}

```

```
function:
build_portal
input:
area1, area2, portals
output:
portals, status: success or failed
{
    if portal_is_too_big
    {
        return failed
    }
    collision_point = find_collision_point(area1, area2)
    new_portal = compute_portal(collision_point)
    if failed voxelize_portal( new_portal)
    {
        return failed
    }
    portals += new_portal
    return success
}

Further explanation to build_portal function:
portal_size is equal to collision_point.distance which is
equal to current_dist, (section 4.3)
find_collision_point function is described in section 4.2
compute_portal function calculates portal shape and orientation,
as described in Sections 4.4 and 4.5
voxelize_portal fails in the following situations:
1. when the portal overlaps the border of the distance field
2. when an attempt to create a~portal from voxels already
assigned to some area is made
3. when an attempt to create a~portal from voxels already
assigned to another portal is made

function:
create_new_areas
input:
distance_field, areas, current_dist
output:
areas
{
    for each unassigned to any area voxel V with distance
    current_dist
    {
        // create new area, starting in voxel V
        new_area = create_new_area(V)
        expand_area( new_area )
        // no collisions with other areas are possible here
        areas += new_area
    }
}
```

References

[1] Alliez P., Cohen-Steiner D., Tong Y., Desbrun M.: Voronoi-based Variational Reconstruction of Unoriented Point Sets. *Proceedings of Eurographics Symposium on Geometry Processing*, 2007.

[2] Anderson B.: An Implementation of the Marching Cubes Algorithm. `http://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html`.

[3] Andersson M.: Bounding Volume Hierarchy. 2012, `http://fileadmin.cs.lth.se/cs/Education/EDAN30/lectures/S2-bvh.pdf`.

[4] Bærentzen J.A., Aanęs H.: Generating Signed Distance Fields From Triangle Meshes. 2002, IMM-Technical report-2002-21.

[5] Bellmann J., Michel F., Deines E., Hering-Bertram M., Mohring J., Hagen H.: Sound Tracing: Rendering Listener Specific Acoustic Room Properties. *EG Computer Graphics Forum*, vol. 27(3), pp. 943–950, 2008.

[6] Beucher S.: The watershed transformation applied to image segmentation. 1991.

[7] Chandak A., Lauterbach C., Taylor M.T., Ren Z., Manocha D.: AD-Frustum: Adaptive Frustum Tracing for Interactive Sound Propagation. *IEEE Trans. Vis. Comput. Graph.*, vol. 14(6), pp. 1707–1722, 2008, `http://dblp.uni-trier.de/db/journals/tvcg/tvcg14.html`.

[8] critterai.org: Study: Navigation Mesh Generation. `http://www.critterai.org/projects/nmgen_study`.

[9] Cryengine: Cryengine. `http://cryengine.com/`.

[10] Erleben K., Dohlmann H.: GPU Gems 3, Chapter 34. Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra. 2007, `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch34.html`.

[11] Fisher M.: Marching Cubes. `http://graphics.stanford.edu/~mdfisher/MarchingCubes.html`.

[12] Hassan O.: *Untersuchung akustischer Wandeigenschaften und Modellierung der Schallrückwürfe in der binauralen Raumsimulation.* Doctoral thesis RWTH Aachen University, Germany.

[13] Haumont D., Debeir O., Sillion F.: Volumetric cell-and-portal generation. *Proceedings of EUROGRAPHICS*, 2003.

[14] Haumont D., Warzée N.: Complete Polygonal Scene Voxelization. *Journal of Graphics Tools*, 2002.

[15] Intel Corporation: Software Occlusion Culling. `https://software.intel.com/en-us/articles/software-occlusion-culling`.

[16] Jachimczak T., Lentz J., Hendriks M.: Unreal engine 2: Level Optimization – BSP. `http://udn.epicgames.com/Two/LevelOptimizationBSP.html`.

[17] Jeong C.H.: Absorption and impedance boundary conditions for phased geometrical-acoustics methods. *Journal of the Acoustical Society of America*, vol. 132(4), pp. 2347–2358, 2012.

[18] Jones M., Baerentzen J.A., Sramek M.: 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 581–599, 2006.

[19] Kamisiński T.: Correction of Acoustics in Historic Opera Theatres with the Use of Schroeder Diffuser. *Archives of Acoustics*, vol. 37, pp. 349–354, 2012.

[20] Kamisiński T., Burkot M., Rubacha J., Brawata K.: Study of the Effect of the Orchestra Pit on the Acoustics of the Kraków Opera Hall. *Archives of Acoustics*, vol. 34, pp. 481–490, 2009.

[21] Kowalczyk K., van Walstijn M.: Room Acoustics Simulation Using 3-D Compact Explicit FDTD Schemes. *IEEE Transactions on Audio, Speech & Language Processing*, vol. 19(1), pp. 34–46, 2011.

[22] Kuttruff H.: *Acoustics – An Introduction.* 2006.

[23] Kuttruff H.: *Room Acoustics, 5th ed.* Spon Press, London, 2009.

[24] Lauterbach C., Chandak A., Manocha D.: Adaptive sampling for frustum-based sound propagation in complex and dynamic environments. *Proceedings of the 19th International Congress on Acoustics*, vol. 16, 2007.

[25] Lauterbach C., Chandak A., Manocha D.: Interactive sound rendering in complex and dynamic scenes using frustum tracing. *IEEE Trans. Vis. Comput. Graph.*, vol. 13(6), pp. 1672–1679, 2007, `http://dblp.uni-trier.de/db/journals/tvcg/tvcg13.html`.

[26] Lorensen W.E., Cline H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, vol. 21(3), pp. 163–169, 1987.

[27] Mahovsky J., Wyvill B.: Memory-conserving bounding volume hierarchies with coherent raytracing. *COMPUTER GRAPHICS FORUM*, vol. 25, pp. 173–182, 2006.

[28] Makinen O., Saransaari H.: Conservative cell and portal graph generation. 2014, `http://www.google.com/patents/US20140043331`, uS Patent App. 13/569,879.

[29] Meijster A., Roerdink J., Hesselin W.: A general algorithm for computing distance transforms in linear time. *Mathematical Morphology and its Applications to Image and Signal Processing Computational Imaging and Vision*, vol. 18, pp. 331–340, 2000.

[30] Miga B., Ziółko B.: Real-time acoustic phenomena modelling for computer games audio engine. *Archives of Acoustics*, vol. 40(2), 2015.

[31] Mommertz E.: *Building Acoustics and Vibration Theory and Practise.* World Scientific Publishing Co. Pte. Ltd., Singapore, 2009.

[32] Mononen M.: Navigation Mesh Generation via Voxelization and Watershed Partitioning. 2009, `http://aigamedev.com/premium/masterclass/navigation-mesh-generation/`.

[33] Neperud B., Lowther J., Shene C.K.: Visualizing and animating the winged-edge data structure. 2007.

[34] Nils L., Jansens G., Vermeir G., van der Voorden M.: Absorbing surfaces in ray-tracing programs for coupled spaces. *Applied Acoustics*, vol. 63(6), pp. 611–626, 2002.

[35] Ozgura E., Ozisb F., Alpkocaka A.: DAAD: A New Software for Architectural Acoustic Design. *Proceedings of the 33rd International Congress and Exposition on Noise Control Engineering Internoise, Prague*, 2004.

[36] Pałka S., Głut B., Ziółko B.: Visibility determination in beam tracing with application to real-time sound simulation. *Computer Science*, vol. 15, pp. 197–214, 2014.

[37] Raghuvanshi N., Lauterbach C., Chandak A., Manocha D., Lin M.: Real-time sound synthesis and propagation for games. *Communications of the ACM*, vol. 07(50), pp. 66–73, 2007.

[38] Sek A., Moore B.: Frequency discrimination as a function of frequency, measured in several ways. *Journal of the Acoustical Society of America*, vol. 97(4), pp. 2479–2486, 1995.

[39] Sekulic D.: GPU Gems, Chapter 29. Efficient Occlusion Culling. 2004, `http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html`.

[40] Shimer C.: Binary Space Partition Trees. `http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html`.

[41] Silvennoinen A., Saransaari H., Laine S., Lehtinen J.: Occluder Simplification Using Planar Sections. *Computer Graphics Forum*, vol. 33, pp. 235–245, 2014.

[42] van Toll W., Cook IV A.F., Geraerts R.: A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, vol. 23, pp. 535–546, 2012.

Affiliations

Piotr Turecki
AGH University of Science and Technology, Krakow, Poland, `pturecki@agh.edu.pl`

Tomasz Pędzimąż
AGH University of Science and Technology, Krakow, Poland, `pedzimaz@agh.edu.pl`

Szymon Pałka
AGH University of Science and Technology, Krakow, Poland, `pszymon@agh.edu.pl`

Bartosz Ziółko
AGH University of Science and Technology, Krakow, Poland, `bziolko@agh.edu.pl`

Received: 30.06.2015
Revised: 1.02.2016
Accepted: 9.02.2016