Computer Science • 15 (2) 2014 http://dx.doi.org/10.7494/csci.2014.15.2.197

Szymon Pałka
Barbara Głut
Bartosz Ziółko

VISIBILITY DETERMINATION IN BEAM TRACING WITH APPLICATION FOR REAL-TIME SOUND SIMULATION

Abstract *This work presents some aspects of beam-tracing techniques used in sound simulation. The Adaptive Frustum algorithm, which was designed for detecting obstacles using beam subdivision, was reviewed for its efficiency as well as its accuracy. Some possible improvements have been suggested; however, they do not fully solve the problems of using this algorithm in real-time applications. Improved algorithm implementation was tested in five scenes with different characteristics and varying complexity.*

Keywords beam tracing, adaptive frustum, visibility determination

1. Introduction

Current trends in game development include integration of physics simulation, simulation of sophisticated artificial intelligence, and usage of three-dimensional scenes with a high level of complexity. All of these features involve the sense of sight in the first order. On the other hand, there seems to be very little advancement in the field of sound simulation. All modern computers incorporate hardware which can create an illusion of spatial sound (using the sound card along with the proper number of speakers placed in certain points of the room), but this hardware is not fully utilized in games. Most game engines simplify sound simulation to the point of playing appropriate sound samples at well-defined moments in the game. Sound can be emitted when a particular situation takes place, a player enters a room, etc. High-end game engines use simple information about room size to alter the sound, but listener position is not accounted for in respect to obstacles.

Game engines which incorporate a sound simulation subsystem (which can alter the sound using information about current game environment state) will definitely enhance the user's experience. The main issue in spatial sound simulation is searching for sound paths between sound sources and the receiver. Well-known techniques for computing propagation paths used in computer graphics, called "ray tracing", are available to use and adapt for different situations. Unfortunately, sound is different than light in its nature, which prevents the effective use of light-simulation techniques.

Beam tracing is a promising technique for simulating sound propagation in three-dimensional scenes. It extends ray tracing by replacing infinitely-thin rays with volumetric objects called beams. The beam tracing method marks point-like sound sources, scenes built from objects represented by triangular meshes, and beams representing sound waves propagated in the scene. The sound source emits six source beams in the direction of the $+x$, $-x$, $+y$, $-y$, $+z$, $-z$ axes. Source beams generate new beams by reflecting off of object surfaces, diffracting on sharp edges, or transmitting through objects with certain materials such as glass, etc. For simplicity, we will assume that beams can only be reflected. Because a beam, unlike a ray, has volume, reflecting the whole beam from only one object is not accurate enough. Instead, it is necessary to find each object which can be reached by any part of the beam and reflect the beam only off of the part of the object's surface which has been struck. If there are no objects struck by the beam or some of its parts – information about it should also be saved.

This paper is focused on the issue of searching for objects within the range of a given beam and determining which object surfaces are not obstructed by other objects.

2. Related work

Existing solutions for determining which surfaces can be reached by a given beam are either approximate or exact. Exact object lookup is usually used with beams

with a triangular base. Heckbert and Hanrahn describe an object-lookup algorithm which does not account for intersecting triangles [6]. It uses a plane-sweeping technique (three dimensional extension of a well-known line-sweeping algorithm). The algorithm starts by positioning a plane at the beginning of a beam and advancing it to the nearest triangle. When the triangle is struck, part of a beam limited by it is removed from further processing. This step effectively subdivides the given beam into some number of smaller beams, which must be further processed.

Overbeck et al used recently-developed structures such as KD-trees and appropriate tests of intersection between beams and triangles to adapt beam tracing for soft-shadows simulation [8]. Authors use beams with a rectangular base, but simulate only primary beams. Unfortunately, an algorithm useful in the audio simulation must be able to simulate reflections, which disqualifies this algorithm in the context of our aims.

Approximate searching methods allow the generation of some errors, thus producing results that are less accurate. Usually, this kind of algorithm allows a user to limit the amount of computation (reducing quality as well). Approximate methods using beams with quadrangle bases have been introduced [7, 2]. Approximate beam tracing appears to be better suited for sound-propagation simulation because of its varying accuracy, which in turn makes it possible to adapt easily to different computational platforms.

A wide selection of algorithms have been prepared for two-dimensional cases of beam tracing. Foco et al suggest an algorithm for visibility determination by tracing beams in both scene and dual space, which allows beam tree updates after its source has been moved [3]. Sufficient execution time for real-time sound simulation has not been achieved in the two-dimensional version described in this article for one sound source, and it is unlikely that a game will use only one dynamic source at a time.

Funkhouser et al described a method of path tracing using scene voxelization [4]. Parts of the scene connected by doors or relatively small unobstructed passages are identified and represented in an adjacency graph. The authors developed a technique for precomputing possible rough sound paths from static sources in a static scene and then using this data to create exact sound paths between sources and the receiver at runtime. Proposed techniques are efficient enough for the purpose of implementing a real-time sound-tracing engine, but cannot be used in dynamic scenes where both objects and sound sources can be moved (which is the case in almost all modern game engines).

Ghazanfarpour and Hasenfratz, in their work on using a beam tracer for enhancing the quality of anti-aliasing with beam tracing [5], suggest a method for visibility determination in scenes constructed from convex polyhedra. This method uses beam subdivision for visibility determination, but it is not fast enough for real-time applications. Its primarily focus is on quality, whereas computation speed is of the highest concern for the purpose of sound-propagation simulation.

3. The aim of the project

An efficient method for obtaining an approximate set of objects obstructing a given beam is necessary for the fast beam propagation simulation. As mentioned before, sound beams do not need to be accurate with respect to the geometry of a scene; however, an equilibrium between computation time and outcome quality must be found. For example, reflecting a beam off of one scene triangle may not create a good solution if the triangle is small and there are a hundred larger triangles also obstructing this beam. On the other hand, computations necessary to process all of the triangles obstructing the beam and, further, processing all of the reflected beams, may be too computation-intensive for any hardware available at the moment. Moreover, if some triangles are coplanar, there is no need for processing a number of different beams, because merging them into one creates the solution of same quality. In the preliminary work about beam tracer capable of simulating sound propagation in real time, the Adaptive Frustum algorithm introduced in [2] was tested, and the quality of its outcome was improved. This work describes the Adaptive Frustum algorithm, problems encountered during its implementation, and integration with a beam-tracing engine. A new method of testing beam-triangle intersection which ensures better outcome quality is introduced and tested on different scenes.

4. Visible objects searching using Adaptive Frustum technique

The basic element in the algorithm is a beam. Let us assume that the beam (Fig. 1) is represented by a quadrangle lying in the plane off which the beam is reflected (initial surface or near base – determined by the points $P1$, $P2$, $P3$, $P4$), unit vectors ($V1$, $V2$, $V3$, $V4$) lying on the edges of the beam which start in points $P1$, $P2$, $P3$, $P4$ and its range (greater than zero). It is assumed that the normal vectors of planes limiting a given beam are directed towards the inside of this beam. The plane on which the beam ends can be calculated by shifting the plane of near base along its normal vector by a distance corresponding to the range of the beam (points $F1$, $F2$, $F3$, $F4$ designate a plane limiting a beam).

Adaptive-Frustum is based on adaptive subdivision of a beam into areas where a large density of triangles is encountered. We assume that after a given amount of subdivision steps, sub-beams are so small that it is possible to simplify the test whether the triangle limits the beam. We classify parts of the beam as intersecting with one of the elements of the scene, or non-intersecting with any of them. If two or more of the elements in the scene intersect with a given part of a beam, the beam portion is further divided into 4 smaller parts. In order to avoid excessive beam splitting, a subdivision limit is introduced. After the limit is reached, no further subdivision occurs, and one of the triangles obstructing a given part of the beam is arbitrarily selected. The final step is to combine those parts of the beam for which limiting triangles are the same, or lie in the same plane and, additionally, consist of

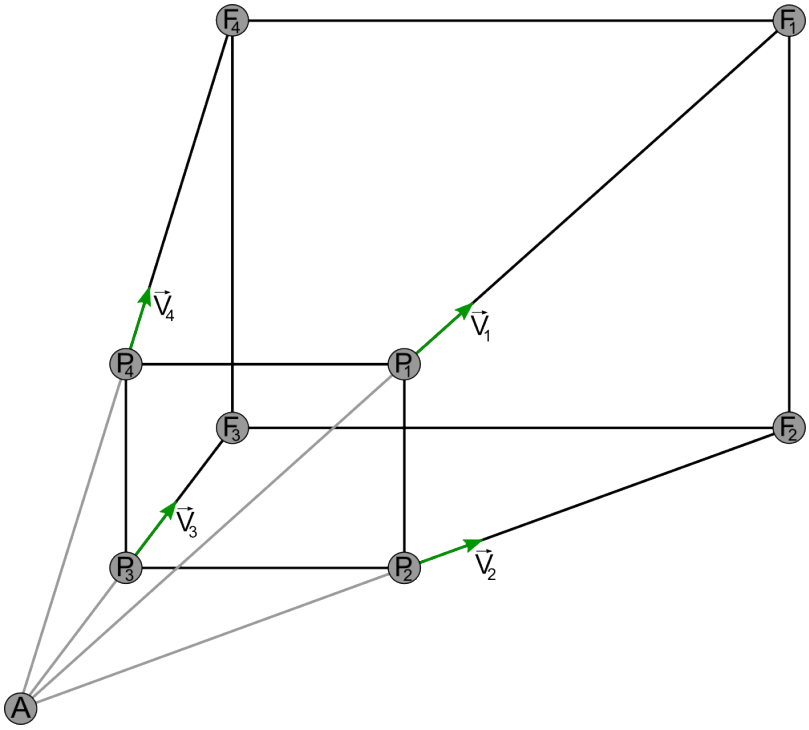

Figure 1. Example beam. Point A – an apex, $P1$, $P2$, $P3$, $P4$ are near base vertices, $F1$, $F2$, $F3$, $F4$ are far base vertices, $V1$, $V2$, $V3$, $V4$ are beam border vectors.

the same material. This step allows a significant reduction in the number of beams traced in the scene.

4.1. Structures

A quadtree structure is used to control the subdivision of a beam in the Adaptive-Frustum algorithm. A split of a beam is equivalent to the partitioning of its initial surface and appropriate interpolation of direction vectors. The structure can be applied to find the details of complex objects, modeling of edges, etc. Building a quadtree is a process of breaking a beam which intersects with a set of triangles into four smaller sub-beams (Fig. 2). Each part is then checked for collisions with triangles colliding with the beam being subdivided (e.g., with an edge or a wall of an object). If the test is positive for more than one triangle, this section is divided further into four parts. The division is executed until it reaches the desired accuracy, or until only one triangle intersects with a given sub-beam.

Depending on the level of complexity of the scene, reflections of many thousands of beams per second may need to be processed. It is impossible to browse through the complete geometry of the scene every time a beam is processed. It is necessary to

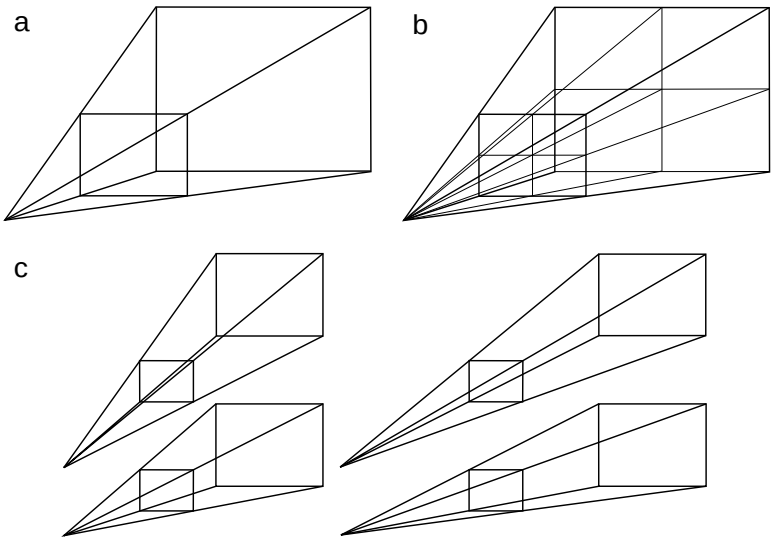

Figure 2. Beam subdivision. a – beam being subdivided. b – beam after one step of subdivision. c – four sub-beams of beam showed in a. The sub-beams are presented separately to emphasize that each of them can be processed independently. Each of sub-beams can be further subdivided.

organize objects in such a way that locating possible obstacles in a relatively small part of the scene is fast. For this purpose, tree representation of the scene is used. This structure allows for a hierarchical representation of the area, which is more accurate in areas which contain more scene objects. The octree is built from axis-aligned bounding boxes in such a way that the triangle is contained in a given cell if and only if it is completely contained by it and no part of this cell completely contains this triangle. A scene tree allows finding triangles potentially intersecting a given beam quickly by selecting the cells intersecting with it and pushing the triangles contained in these cells to the next steps of the processing algorithm, where they will be inserted into quadtree subdividing given frustum.

4.2. Searching of Obstacles

The main features of an approximate search algorithm of obstacles to the beam were already described in[2]. Listings 1 and 2 provide algorithms created using mentioned specification.

Algorithm 1: Adaptive Frustum.

```
1  adaptive frustum( beam, maximum_subdivision, scene_tree )
2  {
3    beam_subdivision = quadtree( beam )
4    possibly_intersecting_triangles = get_triangles( scene_tree, beam )
```

```
5     for triangle in possibly_intersecting_triangles:
6     {
7       insert_triangle( beam_subdivision, maximum_subdivision, triangle
8     }
9     processed_sub_beams = quadtree_leafs( beam_subdivision )
10    return processed_sub_beams
11  }
```

Algorithm 2: Inserting triangles into quadtree used in Adaptive Frustum.

```
1   insert_triangle( quadtree_cell, maximum_subdivision, triangle )
2   {
3     intersection_status = test_intersection( quadtree_cell.beam, triang
4     if intersection_status == true
5     {
6       distances = distance( quadtree_cell.beam, triangle.plane )
7       if is_empty( quadtree_cell ):
8       {
9         quadtree_cell.distances = distances
10        quadtree_cell.triangle = triangle
11      }
12      else if is_subdivided( quadtree_cell ):
13      {
14        for sub_cell : quadtree_cell.sub_cells:
15        {
16          insert_triangle( sub_cell, maximum_subdivision - 1, triangle
17        }
18      }
19      else if distances > quadtree_cell.distances:
20      {
21        return
22      }
23      else if distances < quadtree_cell.distances:
24      {
25        quadtree_cell.distances = distances
26        quadtree_cell.triangle = triangle
27      }
28      else if maximum_subdivision > 0:
29      {
30        // the cell should be subdivided
31        quadtree_cell.sub_cells = subdivide( quadtree_cell )
32        for sub_cell : quadtree_cell.sub_cells:
33        {
34          insert_triangle( sub_cell, maximum_subdivision - 1, triangle
35        }
36      }
37    }
38  }
```

4.3. Decreasing number of traced beams

In order to reduce the number of beams processed in the system, an additional step of beam merging was introduced [2].

Algorithm 3: Adaptive Frustum algorithm with additional beam merging stage.

```
adaptive frustum( beam, maximum_subdivision, scene_tree )
{
  beam_subdivision = quadtree( beam )
  possibly_intersecting_triangles = get_triangles( scene_tree, beam )
  for triangle in possibly_intersecting_triangles:
  {
    insert_triangle( beam_subdivision, maximum_subdivision, triangle
  }
  processed_sub_beams = { }
  merge_beams( beam_subdivision, processed_sub_beams, maximum_subdivi
  return processed_sub_beams
}
```

A beam-merging algorithm is based on combining beams that are sub-cells of a given quadtree cell and are limited by compatible triangles, preferably lying in the same plane or planes whose normal vectors differ by no more than a given epsilon. The triangles should also have the same material assigned. Tolerance in which beams should be combined can be described as one of the parameters defining the desired accuracy of the algorithm. Due to the possibility of different subdivision level of adjacent quadtree cells, it would be very expensive and sometimes impossible to accurately combine beams. For this reason, beams which are parts of the same larger beam are combined. Listing 4 provides the algorithm created using specifications in [2].

Algorithm 4: Beam Merging stage used in Adaptive Frustum.

```
merge_beams( quadtree_cell, merged_beams, subdivision_level, maximum_
{
  if is_not_subdivided( quadtree_cell ):
  {
    return quadtree_cell.triangle
  }
  else
  {
    to_merge = { }
    for sub_cell : quadtree_cell.sub_cells:
    {
      to_merge += merge_beams( sub_cell, merged_beams, subdivision_le
      if size( to_merge ) == 4:
      {
        if mergeable( to_merge )
        {
          if subdivision_level == maximum_subdivision:
          {
            merged_beams += merged( sub_cell, to_merge[ 0 ] )
          }
          else
          {
            return to_merge[ 0 ]
          }
        }
        for every not-merged neighboring sub_cells cell1 and cell2 :
        {
```

```
28              if mergeable( cell1, cell2 ):
29             {
30               merged_beams += merge( cell1, cell2 )
31               to_merge -= cell1
32               to_merge -= cell2
33             }
34           }
35           for sub_cell : to_merge:
36           {
37              merged_beams += sub_cell
38           }
39         }
40       }
41       return NULL
42     }
43   }
```

4.4. Existing tests of a beam – triangle intersection

Due to the complexity of the beam-type object (represented by 6 planes, 8 vertices or 4 vertices, 4 rays, and range value), a beam – triangle intersection may have a high computational complexity, which is disadvantageous for Adaptive Frustum efficiency.

4.4.1. Inaccurate intersection test using beam bounding planes

In order to verify whether a triangle intersects a given beam, triangle position relatively to the six planes limiting the beam should be tested [1]. Let us assume that all planes have a normal pointing towards the interior of the beam. A test with respect to the plane is based on the verification of whether all vertices of the triangle have a positive distance from the plane (they are at its inner side). If all vertices are outside – the triangle certainly does not intersect the beam. Otherwise, the number of planes, for which the triangle is on the inner side has to be counted (if there are 6 – the triangle is within bounds of the beam). If only one or two vertices are on the outer side of one of the planes, and the triangle is not completely on the outer side of one of the surfaces, then the triangle probably intersects the beam.

Algorithm 5: Beam-triangle intersection test using beam bounding planes.

```
1  intersects( beam, triangle )
2  {
3    inside_count = 0
4    for every plane bounding the beam:
5    {
6      if all triangle vertices are not on the plane side to which its n
7      {
8        return not_intersecting
9      }
10     if all triangle vertices are on the plane side to which its norma
11     {
12       inside_count = inside_count + 1
13     }
```

```
14    }
15    if inside_count == 6:
16    {
17      return triangle_inside
18    }
19    return probably_intersecting
20  }
```

The presented method of testing intersections is not accurate. A triangle can be classified as intersecting with a beam, even if it is not. This happens in the case of the triangles outside the beam, but intersecting at least two bounding planes of the beam. The algorithm is efficient and easy to implement, but it should be used only if inaccurate triangle classification is acceptable. Furthermore, this test does not compute distance from the frustum base (along border vectors) to the triangle plane needed in the adaptive frustum algorithm.

4.4.2. Inaccurate intersection test with plucker coordinates

It was suggested [2] to use the frustum-triangle intersection test based on Plucker coordinates. This test, like the test using bounding planes, may incorrectly classify non-intersecting triangles as intersecting with the beam. In order to accurately decide whether an intersection exists, it was suggested that one of the beam border vectors must intersect the tested triangle. Such a case arises when it is impossible to tell whether the triangle is completely inside or completely outside the beam. This solution does not solve the problem completely, because it is possible that the triangle intersects the beam but it does not intersect any of its border vectors. Although the initial test is fast, it is not accurate and needs additional distance computation step to be useful in the Adaptive Frustum algorithm. If the frustum subdivision level is high enough, the accuracy of the intersection test does not matter. However, this is not the case in the sound-rendering application, where it is desirable to find as many valid propagation paths as possible in the time designated for rendering one sound frame. Due to the high computational cost associated with tracing of a given beam in the scene, it is better to reject inaccurate propagation paths as quickly as possible.

5. Proposed Triangle-Frustum intersection test

The developed test provides not only information about the existence of intersection between beam and triangle, but it also computes additional information used in the Adaptive Frustum algorithm, and thus improves its accuracy. The test is based on projecting the beam on the plane of the triangle. If none of the distances between the near base vertex and its projection are in the interval [0, beam range], the triangle does not intersect the beam. Otherwise, the triangle $T = [A, B, C]$ intersects the beam if and only if beam projection – a quadrangle $Q = [R1, R2, R3, R4]$ – intersects triangle T (Fig. 3). In order to verify this intersection, a Separating Axis Test (SAT) is used [1] in a two-dimensional space defined by two arbitrarily-chosen vectors.

This test ensures that the result of object-visibility determination does not contain artifacts not associated with the desired precision of computation. Erroneous detection of the triangle boundary can cause a significant error – audible sound in a place where it should not be heard. It should be noted, however, that not all triangles are arranged with respect to a beam in such a way that it is possible to calculate the distance of all 4 vertices of the beam base to the plane of the triangle (Fig. 3). Therefore, such triangles must be detected and processed in suitably-small parts of the beam, for which it is possible to calculate the projection or to decide that the triangle does not intersect a given part of a beam. The solution to this problem forces a deeper division of the tree (the same problem arises when using the Plucker coordinates test).

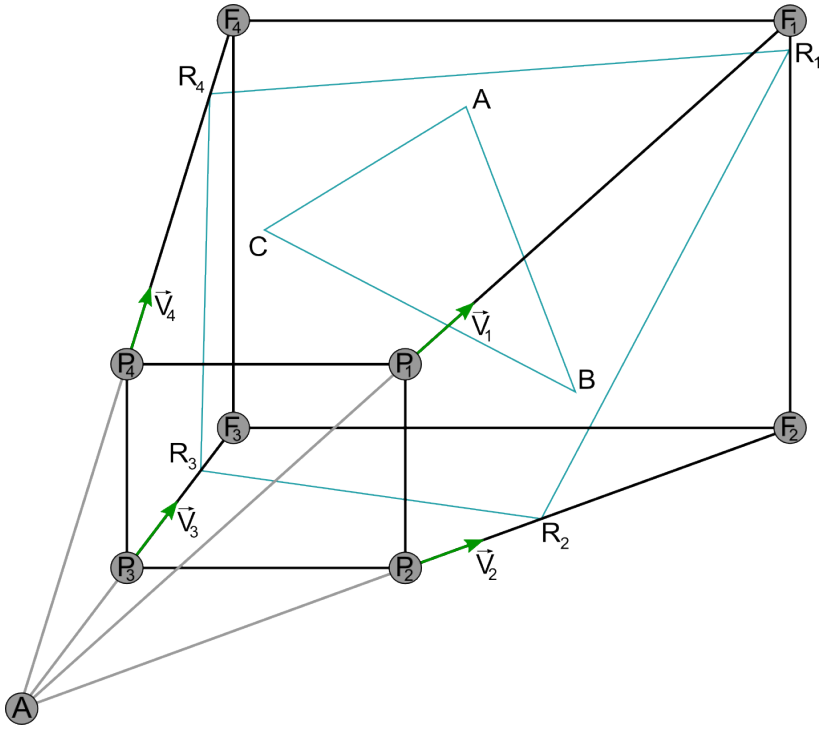

Figure 3. Beam projection on triangle $T = [A, B, C]$ plane. Projection is a quadrangle with vertices $R1, R2, R3, R4$.

6. Problems associated with using Adaptive Frustum algorithm

The main advantage of Adaptive-Frustum seems to be the possibility of adjusting the amount of computation to the required quality of an approximation of a solution

through appropriate limitation of possible beam subdivision. In theory, the possibility of an automatic adjusting beam subdivision to scene complexity should greatly decrease the amount of computation and memory usage. However, this algorithm contains a number of drawbacks.

6.1. The cost of specific cases processing

For all of the considered intersection tests, storing information about the triangle limiting particular component of a beam is based on retaining the distance between the beam base and plane described by triangle vertices. The problem occurs when the projection a beam on the triangle plane does not exist. Since the projection may not exist for the whole beam but could exist for one of its parts, it is not desirable to reject such a triangle. In practice, rejecting such a triangle is undesirable and may lead to the occurrence of large distortions. It is therefore necessary to memorize the triangles which need to be processed after sufficient beam subdivision occurs, or reject them only at the point where it is certain that the beam or its part is too large to be able to calculate the beam projection on the plane of the triangle. A solution to the problem described above requires a deep beam subdivision while encountering such triangles. Assuming that such special triangles are rare, their handling turns out not to cause an excessive increase in the calculation amount. However, in practice, most scenes are composed of a large quantity of such triangles. As mentioned above, the frequent occurrence of such special cases leads to deeper beam subdivision, which in turn leads to a greater number of performed tests and a decrease in performance.

6.2. Intersection tests cost

Testing the beam-triangle intersection not only requires a large amount of computation, but it is also performed frequently. An algorithm required in the simulation has to be efficient enough to run in real-time. Depending on scene complexity, the beam can be subdivided one or two times at most. A low subdivision level implies that most triangles are discarded immediately. Triangles which are not immediately rejected usually require an intersection test with all parts of the beam. Profiling an application by performing only beam tracing on different scenes was conducted. It revealed that about 60% of processing time is spent on performing intersection tests. The use of less-accurate testing provides an opportunity to improve efficiency, but at the cost of decreasing simulation quality. Low accuracy can be balanced by an appropriate choice of size of the scene. However, the error of incorrect classification of a triangle of large size can cause significant (and easy to detect) artifacts.

6.3. The cost of using tree structure

The high cost of tree structure usage is directly related to the speed of memory access in the computer systems that are currently in use. Tree structures are best suited for processing large amounts of data. Adaptive-Frustum uses a tree structure to manage the hierarchical division of the beam, but typically tree depth is low.

Let us assume that a triangle is inserted into the quadtree and testing the intersection for the whole beam yields a positive result. The beam has already been subdivided at least once. The next step is to determine which of the four parts of the beam intersects the triangle. Loading the current state of each part of the beam and conducting the intersection tests are necessary. Five intersection tests have been performed up to this point. The high cost of using tree structures is directly related to the process of deciding which triangles intersect with a part of the beam and the amount of data needed to be stored. Common techniques of reducing the time overhead incurred by the use of tree structures (such as the use of data pools) have not been sufficient to significantly reduce the execution time.

6.4. The cost of beam merging algorithm

A step of combining beams in the Adaptive-Frustum is simplified as much as possible. This is due to the possibility of an uneven subdivision of the beam and the difficulties in the collective determination of the cell quadtree neighbors. Even such a simple form of the algorithm in conjunction with the use of tree structures causes a large run-time cost. Practical tests have shown that the application of the algorithm allows for large-scale reduction in the number of beams. Unfortunately, the cost of applying this algorithm balances with the benefits of reducing the number of traced beams for small subdivision levels.

Another problem associated with the beam merging algorithm is its low accuracy forced by inhomogeneous distribution of the beam. In practice, it often turns out that the number of beams could undergo a further reduction. Unfortunately, increasing the accuracy of beam combination turns out to be a complex task. It is made even more difficult by the different sizes of the parts of a beam and its representation as the tree structure.

7. Tests

In order to be useful for audio rendering in games, the beam-tracing method must be as efficient as possible. Computer games often simulate a dynamic environment and, thus, the scene can change many times per second. After each such scene change, beam propagation must be performed using beam tracing and, in turn, the adaptive frustum technique. The method for determining a visible surface set must be thus as efficient as possible.

The Adaptive Frustum algorithm using the suggested frustum-triangle intersection test have been implemented and tested. C++ language with Microsoft Visual Studio 2012 was used. Performance tests were made on a computer with Intel Core I5 2430M CPU and 8GB of DDR3-1333 memory.

The beam-tracing engine was tested with scenes of different complexity. All of the scenes are briefly characterized, with emphasis on their most important features. All of the scenes are composed of a few thousand to tens of thousands of triangles. Typically, sound scenes can be greatly simplified relative to their computer graphics

versions. Objects with a size smaller than 1 meter can be omitted, and larger objects can be simplified by removing small details.

Church model (Fig. 4) is a closed scene composed of 1329 triangles. Usage of a closed scene and placement of the sound source inside it ensures a high reflection count, which in turn means that a high number of beams will be traced in the scene. A relatively-large number of small triangles in this scene ensures a high level of beam subdivision, which makes it possible to evaluate algorithm performance in the worst-case scenario.

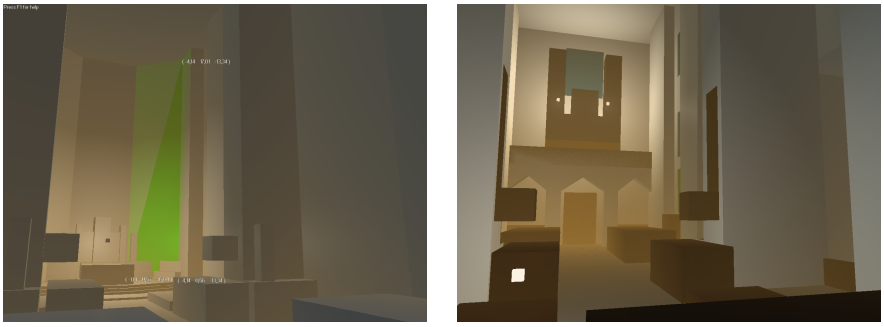

Figure 4. Church model – visualization.

Auditorium model (Fig. 5) is a closed scene composed of 2986 triangles. Characteristics of this scene are similar to the church model; however, a higher triangle count makes it possible to check algorithm scalability. The modeled room consists of low complexity walls and a highly-complicated ceiling. This property allows benchmarking of how increased triangle density in a given region affects overall performance.

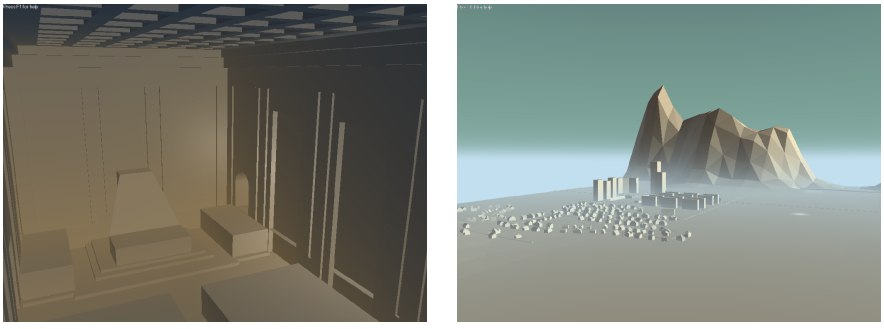

Figure 5. Auditorium model (left) and Desert scene (right) – visualizations.

Colosseum model (Fig. 6) is an open scene composed of 36 233 triangles. Independent of source position, most triangles in the scene are obstructed. Configuration with lots of invisible triangles may be considered the worst-case scenario for the adaptive frustum technique.

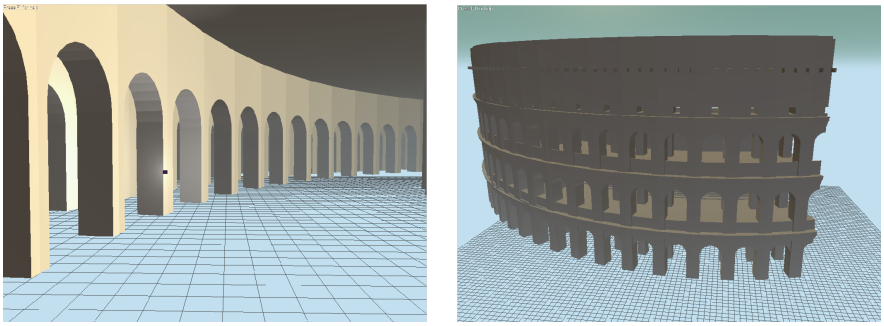

Figure 6. Colosseum model – visualization.

Desert scene (Fig. 5) is an open scene composed of 24 419 triangles. This scene is an example of a large-scale environment consisting of regions with high and low triangle density. In this scene, most beams do not strike any triangles. This in turn implies that the number of created beams is low. However, if a triangle is struck, it has a high probability of obstructing other triangles (which negatively affects performance).

Krakow Old Town model (Fig. 7) is an open scene composed of 1196 triangles. The scene is composed of various buildings of different sizes, with squares, passages between buildings, and streets meeting at different angles. Low resolution and a large-scale scene allow for very fast beam propagation.

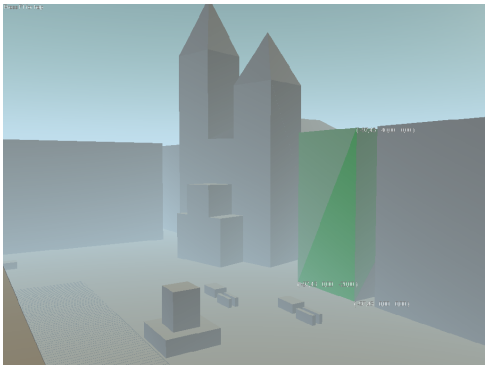

Figure 7. Krakow Old Town model – visualization.

Outcomes of performance tests on the scenes are described in Tables 1, 2, 3, and 4. *Max subdivision* parameter describes how many times a given beam can be subdivided in the Adaptive Frustum algorithm.

Performance tests show that the Adaptive Frustum algorithm can be used in real-time sound rendering in most scenes. However, even moderate quality of outcome cannot be achieved. The beam-merging technique proved useful in reducing the

Table 1

Beam propagation time in seconds with beam merging stage.

Max Subdivision	Church	Auditorium	Colloseum	Dessert	Old Town
1	0,02s	0,02s	0,21s	0,05s	0,01s
2	0,24s	0,36s	1,54s	0,38s	0,06s
3	1,94s	4,08s	10,12s	2,01s	0,42s
4	11,84s	35,58s	over 90s	12,32s	1,84s

Table 2

Beam propagation time in seconds without beam merging stage.

Max Subdivision	Church	Auditorium	Colloseum	Dessert	Old Town
1	0,03s	0,04s	0,21s	0,1s	0,02s
2	0,36s	0,6s	1,82s	0,58s	0,11s
3	5,11s	10,23s	over 90s	5,63s	1,06s
4	45,56s	over 90s	over 90s	56,68s	8,68s

Table 3

Number of beams created – with beam merging stage.

Max Subdivision	Church	Auditorium	Colloseum	Dessert	Old Town
1	252	189	273	24	126
2	6 202	6 814	7 166	2 012	2 072
3	56 054	93 178	91 695	19 683	17 797
4	319 909	646 403	not tested	142 826	81 311

Table 4

Number of beams created – without beam merging stage

Max Subdivision	Church	Auditorium	Colloseum	Dessert	Old Town
1	387	395	294	192	224
2	186 765	18 503	12 478	4 822	5 651
3	616 035	766 479	526 917	177 374	137 823
4	7 222 029	not tested	not tested	3 067 242	1 655 624

number of traced beams, but it cannot significantly reduce processing time for low-beam subdivision levels. Since each beam can be divided into no more than four parts, not many potentially-matching triangles are detected. Thus, computations associated with the beam merging algorithm are partially wasted.

8. Conclusions and future work

The Adaptive Frustum algorithm does not meet all expectations. It can be set up to provide sufficient accuracy. It can provide enough performance to be used in real-time applications. However, both of these objectives cannot be reached at the same time. All modern game engines rely heavily on power of multi-threaded processing, and after assigning processing power to rendering, artificial intelligence, and physics simulation, not much power is left. A satisfactory division of resources would be to assign 1/4 of CPU processing power to sound rendering, which equals 1 core in a 4-core processor. Taking this assumption further, if sound frames are created every 30 ms (which is the lowest frequency a typical computer game uses for rendering and, thus, for changing scenes), beam propagation could use at most half of this time, which equals 15 ms of computation time. Although using the Adaptive Frustum technique can ensure that such performance will be achieved, it cannot provide quality better than achieved with level 1 of the frustum subdivision in a scene of low complexity. This in turn means that 6 source beams can reflect at most off of 24 different surfaces, which is not a reasonable amount for any type of scene.

Tests show that most computation time is spent on performing beam-triangle intersection tests. Faster tests do not improve performance, since they do not explicitly compute distance to intersection with the triangle surface necessary for determining which triangles obstruct the others. Octree used as a scene representation structure can be easily substituted with another structure; for example, KD tree, BVH, etc. Such a structure can decrease the number of performed intersection tests.

Another topic that hasn't been fully exploited is the beam merging algorithm. Since beam subdivision is not uniform, this makes it possible to only join beams which are in the same quadtree cell. In practice, the quadtree is subdivided only a few times, which means that, in most cases, almost all of the cells will be fully subdivided and a more-accurate merging algorithm can be employed. Uniform frustum tracing is not as efficient when using high resolution but can provide better results for lower resolutions, where higher computational cost is amortized by the much-lower number of processed frusta. This solution, with proper SIMD implementation and quadtree structure removed, can possibly outperform the Adaptive Frustum algorithm and, thus, needs to be examined.

Acknowledgements

The research presented in this paper was partially supported by the National Center for Research and Development under INNOTECH-K1/IN1/50/159658/NCBR/12 grant. The research presented in this paper was partially supported by the the AGH grant 11.11.230.015.

References

[1] Akenine-Möller T., Haines E., Hoffman N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. ISBN 987-1-56881-424-7.

[2] 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*, 14(6): 1707–1722, 2008.

[3] Foco M., Polotti P., Sarti A., Tubaro S.: Sound spatialization based on fast beam tracing in the dual space. In: *6th International Conference on Digital Audio Effects, DAFx-03 Queen Mary, University of London*. 2003.

[4] Funkhouser T., Carlbom I., Elko G., Pingali G., Sondhi M., West J.: A Beam Tracing Approach to Acoustic Modeling for Interactive Virtual Environments. In: *Computer Graphics (ACM SIGGRAPH '98 Proceedings)*, pp. 21–32. 1998.

[5] Ghazanfarpour D., Hasenfratz J.: A beam tracing method with precise antialiasing for polyhedral scenes. *Computers & Graphics*, 22(1): 103–115, 1998. ISSN 0097-8493.

[6] Heckbert P. S., Hanrahan P.: Beam Tracing Polygonal Objects. *Computer Graphics*, 18-3: 119–127, 1984.

[7] Lauterbach C., Chandak A., Manocha D.: Interactive sound rendering in complex and dynamic scenes using frustum tracing. *IEEE Trans. Vis. Comput. Graph*, 13(6): 1672–1679, 2007.

[8] Overbeck R. S., Ramamoorthi R., Mark W. R.: A Real-time Beam Tracer with Application to Exact Soft Shadows. In: *Proceedings of the Eurographics Symposium on Rendering Techniques, Grenoble, France, 2007*, J. Kautz, S. N. Pattanaik, eds., pp. 85–98. Eurographics Association, 2007. ISBN 978-3-905673-52-4.

Affiliations

Szymon Pałka
AGH University of Science and Technology, Krakow, Poland, szympalka@gmail.com

Barbara Głut
AGH University of Science and Technology, Krakow, Poland, glut@agh.edu.pl

Bartosz Ziółko
AGH University of Science and Technology, Krakow, Poland, bziolko@agh.edu.pl; Techmo sp. z o.o. (http://techmo.pl), Krakow, Poland

Received: 24.10.2013
Revised: 14.11.2013
Accepted: 15.11.2013