Ray-casting point-in-polyhedron test

Denis Horvat*
Supervised by: Borut Žalik†

University of Maribor
Faculty of Electrical Engineering and Computer Science
Laboratory for Geometric Modelling and Multimedia Algorithms
Smetanova ulica 17, SI-2000 Maribor, Slovenia.

Abstract

This paper considers a ray-casting point-in polyhedron test. Although it is conceptually the simple extension of a well-known point-in-polygon ray-casting algorithm, various practical problems appear in 3D, especially, when the boundary of a geometric object is represented as a triangulated surface. When a larger number of points have to be tested regarding their positions on the considered geometric object, preprocessing may drastically reduce the testing time. This paper considers comparisons between three such methods: The first uses a k-dimensional tree (kd-tree), the second an octree, and the last is based on a three-dimensional uniform grid (3D grid). The core operation for all three methods is the ray-casting, by which the odd-even rule can be efficiently applied. Ray-casting can be susceptible to the rounding errors, which are also considered in this paper.

Keywords: 3D inclusion test, spatial subdivision, kd-tree, 3D uniform grid, octree

1 Introduction

Knowing whether a certain point lies inside a given polyhedron can be beneficial within a wide-range of computer graphic applications. This test is often called the inclusion test and is usually used as a base operation in conjunction with more complex ones, so it is essential that it satisfies certain criteria. These usually involve speed, robustness, and memory usage. The inclusion test has its well-known usages in the field of collision detection, where it helps to ensure that objects do not fall through the ground or go through walls. It can also be found in physics simulation and artificial intelligence. Many efficient methods regarding the inclusion test were presented [15, 7, 3, 1] that all have their advantages and disadvantages, depending on the tested object. They can be divided into two basic groups: methods that require a data preprocessing phase, and those that do not. The latter includes ray-crossing methods [4, 3], the angular method [7], barycentric coordinates [1], the winding number method [7], and others. The time-complexity for a single tested point in those methods without preprocessing is *O(n)* [15, 10], with *n* being the number of vertices. Whilst these methods are suitable for small a number of tested points, they become less and less appropriate when the number of tested points increases. At that stage it might be better to consider methods that perform data preprocessing before executing the actual inclusion test. During the preprocessing phase, data is systematically organized, and is later used as input for the inclusion test. Data preprocessing is usually the most intensive operation, but is only performed once. Many structures can be used for data preprocessing, such as uniform grids [15, 12]. The expected time complexity of a inclusion test for methods that use data preprocessing is *O(log(n))* or even *O(1)*. In this paper, the advantages and drawbacks of three data structures that can be used for the inclusion test are analized, described, and compared with each other. The problem of inclusion is solved for the boundary representation (B-rep) of a polyhedra that consist of triangular meshes.

The paper is organized in 7 sections. Section 2 briefly describes the related works that have already successfully solved the problem of inclusion. Section 3 describes how to find an intersection between a point and triangular plane. Section 4 describes how to subdivide space using a kd-tree and an octree. Solutions for how to traverse the mentioned tree structures are also given. Section 5 explains how to voxelise a scene and use it for the inclusion test. Section 6 tackles those problems that may arise from rounding errors when using rays. Section 7 describes and compares the results of experiments conducted on a single workstation using the preprocessing methods described in sections 4 and 5. Section 8 summarises the paper.

2 Related work

Very few original methods seem to have been developed for the inclusion test in three-dimensional space. Most of them are just extensions of their two-dimensional counterparts. Feito and Torres [2] solved the problem of inclusion

*denis.horvat@uni-mb.si
†zalik@uni-mb.si

without the usage of trigonometric functions, and without solving any equations. In [16], a layer-based structure was used for scene preprocessing, where the occlusion relation between faces and edges were calculated, and the faces were then projected on sequentially arranged layers. Later, a binary search algorithm was used on the preprocessing data. The storage-space used during preprocessing was greatly reduced because much of the information for the polyhedrons is represented implicitly. Other various methods can be used by algorithms that are popular in computer graphics. The inclusion test using rays is closely related to ray-tracing, as each ray must be processed by a ray tracing-algorithm in order to determine the intersection points (if they exist) with the tested object. In his PhD thesis [6], V. Havran conducted a comprehensive comparison between twelve commonly used ray-tracing algorithms for a set of thirty test scenes. His findings were, that ray-tracing algorithms based on a kd-tree achieved the best results in comparison with other tested algorithms. Octrees were placed second.

3 Inclusion test using ray-casting

Methods that involve ray-casting follow a simple principle when it comes to the inclusion tests. The basic primitive in ray-casting is a ray $\overrightarrow{R}$, which is defined by its origin O and a normalized direction vector $\overrightarrow{D}$. The ray is cast from a tested point p, that serves as its origin, in any direction. Next, the number of intersections is determined between the polyhedron and the ray. If this is an odd number, p lies inside the polyhedron, otherwise it is outside (odd-even rule). Each individual triangle that is a part of triangular mesh is tested with the casted-ray in order to count the number of intersections for a single point p. This is done by first finding the intersection of a ray with a triangular plane. Any point P lying on $\overrightarrow{R}$ can be defined by a parametric representation (1) of that ray, where t is the signed distance.

$$P = O + t\overrightarrow{D} \tag{1}$$

The signed distance t can be calculated from (2), where d is the distance of the plane from the origin and $\overrightarrow{N}$ the plane normal vector.

$$t = \frac{-(O\overrightarrow{N} + d)}{\overrightarrow{D}\overrightarrow{N}} \tag{2}$$

If $t < 0$, then the triangular plane lies behind the vector origin any tests for that particular triangle can be aborted (figure 1a). This also happens when the ray and triangular plane are parallel to each other (figure 1b), or in geometry terms, when a dot product between the triangular plane and $\overrightarrow{D}$ equals zero. In the case of a positive t, intersection P with a triangular plane is calculated (figure 1c) using (1). This intersection point is then tested with one of the point in polygon tests without preprocessing mentioned in section 1. As already stated, this part can be susceptible to rounding errors, which are addressed in section 6.

Thus, the basic inclusion test is already possible, but every triangle is tested for each point, which leads to undesirable results regarding speed. Consequently, data preprocessing is introduced to ensure that the minimal number of triangles is tested. There are many structures that can be used for spatial subdivision that all have their advantages and disadvantages depending on the given scene [6, 1]. The next two sections explore three of these structures: kd-tree, octree, and uniform grid.

It is also important to mention, that in order for the described methods to provide valid results, the polyhedra should not contain missing triangles or cracks, as the ray could go through that hole and consequentially the point would be classified incorrectly. This problem can be tackled by testing each point using more rays, and selecting the result with the majority.

4 Using tree structures for spatial subdivision

One of the ways to minimise the number of tested triangles for each individual ray, is to recursively subdivide the space by constructing a tree structure. The divided space volume is presented in the form of axis-aligned bounding boxes (AABB). Each node is associated with his AABB, whiles the bounding box of the root node covers the whole of scene S. Nodes that have no child nodes are called leaves. Leaves that contain at least one object of S are called full leaves, otherwise they are empty leaves.

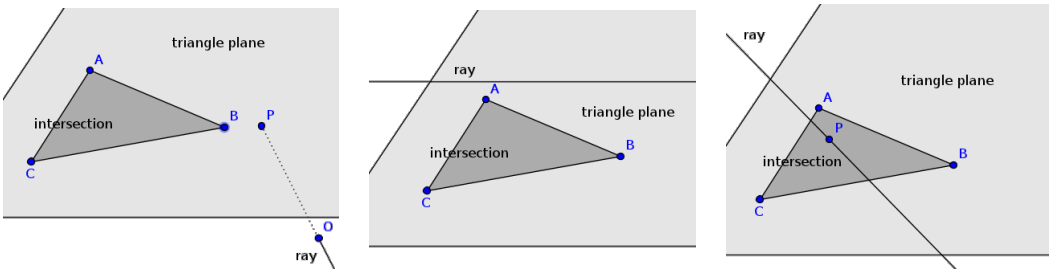
4.1 Kd-trees

Space can be subdivided by a k dimensional tree (kd-tree). Here, the number of dimensions k is limited to three. A kd-tree is a binary tree, which recursively divides space into two new AABB. The division stops after a given criteria is reached. What makes the kd-tree unique is that AABB is divided by a splitting plane, which can be positioned anywhere as long it is perpendicular to its dividing axis. One of the ways to choose the dividing axis is to change it in the cyclic order x,y,z one axis per each depth, usually starting with x. The method of always splitting the longest axis can also be considered. An example of a simple subdivision within a two-dimensional space using a kd-tree, is shown in Figure 2.

At this stage, two important questions regarding the tree construction need to be answered [6]:

- Where to position the splitting plane?
- When to stop dividing?

The answer regarding the first question is very important as it can improve the overall speed of the tree-traversing step. Several methods are known for the positioning of the splitting plane:



(a) Plane lies behind ray origin

(b) Ray and plane are parallel to eachother

(c) Ray intersects plane in P

Figure 1: Different cases of ray-plane intersection

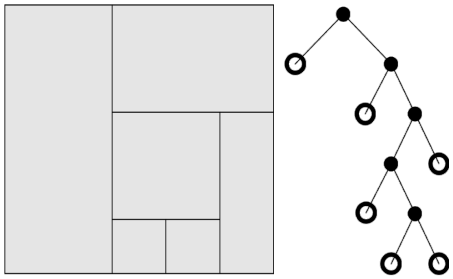


Figure 2: Example of a non-balanced kd tree

- **Spatial Median:** Existing AABB is always divided into two halves by its splitting plane.
- **Object Median:** The splitting plane is positioned in such a way, that the number of objects on each side is roughly the same. This is generally a bad idea [6], because no leaf in the kd-tree will be an empty leaf, meaning that when the tree is traversed, all the objects would need to be checked for each leaf the ray passes through.
- **Cost Model:** This method determines the optimal splitting plane position by calculating the splitting cost using a heuristic for each considered plane. The heuristics used is called *the surface area heuristic or SAH*, as described in [6, 11]. *SAH* uses the idea that the chance of a ray hitting an AABB is proportional with its surface area. This means that it is best to isolate the bigger empty nodes, so that the ray has the highest chance of passing through them unhindered. The cost is calculated using (3), where C_t is the cost of traversal, p_l, p_r the probability of a ray intersecting the left or right node, and C_l, C_r the estimated cost of the left and right sub-node.

$$C_n = C_t + p_l \cdot C_l + p_r \cdot C_r \tag{3}$$

Next, a termination criteria is determined for when to stop dividing and classifying the current node as a leaf. The criteria used is called *ad hoc* termination criteria, where the current node C_n becomes a leaf when a tree depth reaches a certain threshold T_{max}, or the number of objects in C_n is less than the constant O_{min}. Both constants are determined by the user.

After space has been partitioned using a kd-tree, the data obtained can be used when an inclusion test is performed. Only those leaves that the ray intersects are examined instead of the whole scene. This is done with the tree traversal. The recursive ray traversal algorithm TA^A_{rec} [6, 5] was used to traverse the kd-tree. Intersections with a polyhedra are determined by testing all the objects from intersected leaves using the method described in Section 3. The algorithm was published by [8] and uses near-far node classification based on the ray's origin. When a node is traversed, it is calculated which one of his child nodes must be traversed, and which can be skipped. Three cases of traversal are possible: traverse near, traverse far, traverse near and then far. The algorithm uses a stack structure to keep track of the nodes that need to be traversed. The result of the traversal can be seen in Figure 3.

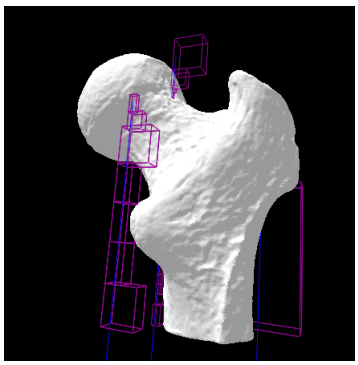


Figure 3: Result after kd-tree traversal using $T_{max} = 21$ and $O_{min} = 4$ and SAH split division criteria.

4.2 Octrees

Octrees are usually used to partition three dimensional space by dividing it into eight octants. Another difference

compared with the kd-tree is that all three planes are divided per each depth, instead of just one. Each node has its own AABB and the leaf nodes have a list of objects that are contained within that AABB. Triangle-cube intersection algorithm [9] was used to classify objects to their individual AABB. The space is subdivided until the *ad-hoc* termination criteria described in subsection 4.1 is reached. Simple subdivision of the three-dimensional space using octree, can be seen in Figure 4.

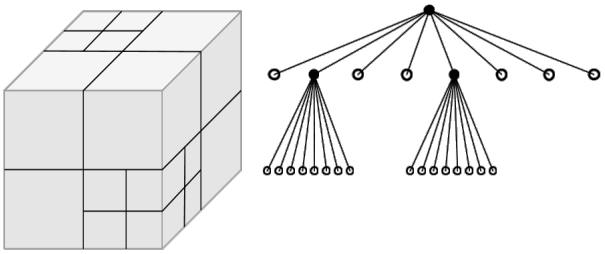


Figure 4: Example of an non-balanced octree.

The traversal step is performed by using an *efficient parametric algorithm for octree traversal* as published in [13]. Similarly to section 3, this algorithm leans on the fact that any point lying on ray can be defined by a parametric representation (1) of that ray $\overrightarrow{R}$, where t is the signed distance. Let b_{max} be the AABB maximum boundary point and b_{min} its minimum. If at least one positive distance t exists on $\overrightarrow{R}$ that is between the AABB boundaries, then $\overrightarrow{R}$ intersects that AABB. The algorithm works with distances from the ray origin O to the nodes limits b_{max} and b_{min}. These distances t_{max_i} and t_{min_i} are calculated using the equations (4), where $\overrightarrow{D}$ is the ray orientation and i one of the coordinate.

$$t_{max_i} = \frac{b_{max_i} - O_i}{\overrightarrow{D_i}} \qquad t_{min_i} = \frac{b_{min_i} - O_i}{\overrightarrow{D_i}} \tag{4}$$

This calculation is only done for the root node. Distances for child nodes are incrementally calculated from parent nodes using three additions and three shifts as the algorithm recursively progresses. For each voxel, the first crossed node is determined (if it exists). Based on the first node, the next nodes can be found until ray exists the current voxel. The algorithm recursively progresses until the leaf nodes are reached. The result can be seen in Figure 5.

5 Uniform grids

For the inclusion test using uniform grids, the algorithm from [12] was implemented. The mentioned algorithm is a three-dimensional extension of the *two-dimensional cell-based containment algorithm (CBCA)* [15]. During

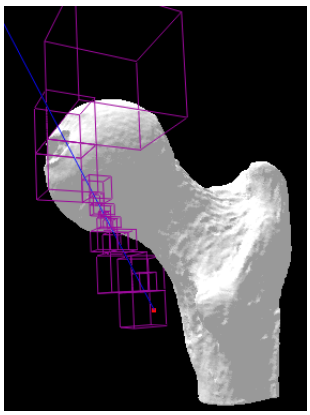


Figure 5: Result of octree traversal for one ray using T_{max} = 20 and O_{min} = 50

the preprocessing phase, CBCA constructs a raster and places it onto a given scene. Each cell is marked as: inside, outside, or as a border cell. Flood fill algorithm is used in order to determine whether a cell is located inside of a polygon. When the actual inclusion test is performed, the algorithm calculates in which cell the tested point is located, and then checks the cell's status. In the case of a cell being marked as inside, the point is declared to be inside, otherwise it is outside. When it is marked as a border cell, additional testing is performed, depending on whether a detailed inclusion test is requested.

In a three-dimensional space, a grid of uniform voxels is used instead of a raster, and ray casting is performed instead of the flood fill algorithm. The object is voxelised, but similarly to CBCA, not all the cells (here voxels) lie completely inside or outside. If the approximation test suffices, then the voxels are marked as inside if 50% of their volume lies inside the tested object, and vice versa. When a detailed test is required, then those voxels that contain the object surface are marked with one more additional bit. Additional tests are performed for those points that lie inside such voxels.

5.1 Voxelisation

Voxelisation is done by an algorithm described in [14], but the idea for uniform voxelisation of three-dimensional polygonal objects or polyhedra comes from [17]. So-called optimised ray-casting is used to determine whether a voxel lies inside or outside of a given object. Voxelisation is performed in two steps:

- In the first step, the initial AABB of the scene is calculated and its xy plane is partitioned using a quadtree. Partitioning stops when the *ad hoc* termination criteria described in section 4.1 is reached. Each leaf in the quadtree corresponds to an AABB with its deph equal to the initial AABB. Each leaf keeps track of all objects that are contained within its AABB. A

Cohen-Sutherland line clipping algorithm [3] is used for determining the containment of an object to the specific AABB. Once the space is partitioned and the tree is built, the next step can commence.

- The second step begins by covering the scene with a uniform grid, containing $m \times n \times p$ voxels. The question regarding the grid resolution must be answered, as it influences the preprocessing speed, memory usage, and accuracy. In 2D, an equation can be used to determine the grid's size [15]. Here, three methods are considered [12]: The first method has a fixed voxel size that is applied to all scenes. The voxel size can also be determined by the user, which gives him/her more control over each individual scene. The third method calculates the voxel size in relation to the scene's extent. After the grid has been constructed, $m \times n$ rays are cast parallel to the positive z axis. One of the $m \times n$ voxel's center then serves as the origin for each ray. The z value of each individual origin is equal to the minimal z value of the initial AABB. For each individual ray, an AABB is found by recursively looking into the quadtree, obtained during the previous step, and finding the leaf that corresponds to the x, y coordinate of the ray's origin. All objects in this AABB are checked for intersections and for each intersection found, its z coordinate value is stored inside a linked list. A new list is always generated for each ray. This list is then sorted in ascending order. For each voxel that is traversed by a ray, the number of values in the list are counted that are smaller than the current voxel center z value. The odd-even rule is applied, which means that if the number of values is odd, then the voxel lies inside of the object, or outside if it is even. The result can be seen in Figure 6.

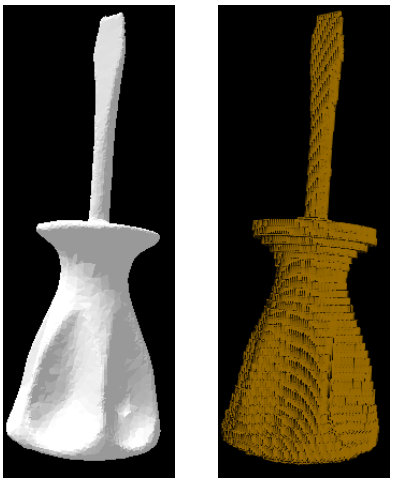


Figure 6: Voxelisation using 128^3 voxels.

5.2 Inclusion test

After the object has been fully voxelised and state of each voxel is known, inclusion test is as simple as reading the voxel's state in which the tested point is contained. The coordinates of a voxel v(m_v, n_v, p_v) for a tested point p(x_p, y_p, z_p) can simply be calculated [12] using the following equations (5):

$$m_v = \frac{x_p - x_{min}}{size_m} \quad n_v = \frac{y_p - y_{min}}{size_n} \quad p_v = \frac{z_p - z_{min}}{size_p} \tag{5}$$

Problems occurs when the objects' boundaries are passing through the voxel. Part of the voxel may be located inside and the other part outside the object, but the voxel can only have one of the two states. This can cause that the result of the inclusion test is incorrect. When the voxel is marked as a boundary, a ray parallel to its z axis is cast from the tested point. The ray stops after a non-boundary voxel is encountered. Intersections are determined (as described in subsection 5.1) and if their number is odd, the status of the tested point is opposite to that of the voxel status in which it is contained.

6 Numeric stability

Sadly computer arithmetic is finite. This causes rounding errors to occur and consequently, inclusion tests can return false results.

As shown in section 3, the intersection between a ray and triangular plane is calculated. This intersection point is then tested using one of the basic point-in-triangle tests and if it is located inside, then intersection occurs. The biggest occurs when this intersection point is located just on the triangle's edge (or very near). Rounding errors can cause the calculated point to be slightly shifted and can now be falsely located in one of the neighbouring triangles. This means, no intersection with that triangle will be found and the result from the inclusion test will be incorrect because the methods used rely on the fact that number of intersections is calculated correctly, so that the odd-even rule can be applied. Robustness was achieved through shared calculations [1] between triangles that share a common edge, using the triple scalar product. All the floating point numbers were compared using relative tolerance comparison. It is important to say, that although the mentioned test improves the robustness when checking for intersections, it is still not 100% accurate.

7 Results

The solutions were tested and compared on a desktop workstation running on Windows 7 using the following

hardware: an Intel i5 processor with a clock speed of 3300 MHz and 4 GB (DDR3) of RAM. All the algorithms were implemented in C++ using the Qt framework. The OpenGL graphical library was used for visual presentation. The structure for storing the model data consisted of a list of triangles and vertices. Normal vectors for each of the triangles were precalculated, when the model was loaded into the memory. Data about the models used for testing, are shown in Table 1 and their thumbnails can be seen in Figure 10. For each result, the average from three measurements was used.

Table 1: Model data

model:	num. vertices	num. triangles
pumpkin	5002	10000
cat	7335	14634
owl	19884	39764
bust	47516	95028
gren1	122227	134016
isis	93823	187642
tiger	309403	618786
gren2	957507	1072128

Total of three methods were implemented for the point-in-triangle test: the angle sum method, the winding number method and a method that uses barycentric coordinates. In the presented tests the method using barycentric coordinates came out on top as it was about 155% quicker compared to the winding number method and more than 223% than the angle sum method. Consequently, the method using barycentric coordinates was used in all the future tests.

Table 2 shows the results for each method implemented. The inclusion test was executed on 300,000 points that were randomly placed on the scene. For the kd-tree (Kd HS represents a tree built using the object median criteria, whiles kd SAH uses the surface area heuristic) and the octree, *ad-hoc* termination criteria was used where $T_{max} = 16$ and $O_{min} = 6$. The object was voxelised using a grid size of 256^3 voxels for approximate testing, where the results were about 99.3% accurate. An accurate grid test was not used for comparison because classification of the boundary voxels took too long. If antialiasing is used during voxelisation, as described in [14], some voxels can still be missed and are not classified as boundaries. Consequently, each voxel must be tested using the *triangle-cube intersection algorithm* [9] for all triangles that are contained within the quadtree node that correspond to the current ray. As soon as only one triangle is found, the boundary test for that voxel can be aborted and the voxel classified as boundary. For example, the preprocessing for the model *cat* took more than 2.5 minutes, which is not even remotely comparable with other methods.

The preprocessing phase T_p and the executing phase T_e were measured for each method. The results for preprocessing T_p are shown in Figure 7. Time required for preprocessing rises with number of triangles on the scene. Preprocessing for kd-trees executes faster than for octrees. This happens because the algorithm for octree must classify each individual triangle to its octant by using the *triangle-cube intersection* test, for each triangle. Classification for a kd-tree only consists of a test that simply determines which side of the splitting plane the triangle is located.

Table 2: CPU(s) for times used for preprocessing and executing for all the tested models, using the implemented methods.

model	Kd HS		Kd SAH		Octree		Grid AP	
	T_p	T_e	T_p	T_e	T_p	T_e	T_p	T_e
pumpkin	0.02	2.19	0.12	2.01	0.12	3.18	2.29	0.05
cat	0.05	1.76	0.12	1.51	0.19	2.2	2.28	0.03
owl	0.06	2.90	0.28	2.70	0.45	4.01	4.07	0.06
gren1	0.08	5.26	0.70	1.95	2.72	2.26	8.85	0.05
bust	0.08	4.32	0.53	3.33	1.12	4.15	7.27	0.03
isis	0.13	7.98	1.33	5.28	2.46	5.51	14.6	0.05
tiger	0.20	11.1	2.95	4.92	10.9	4.31	36.7	0.05
gren2	0.29	21.7	4.49	11.85	20.3	2.84	62.3	0.05

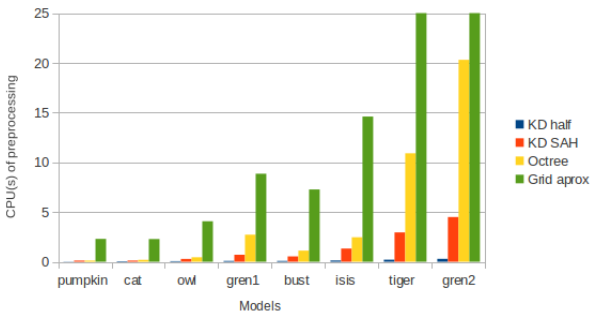


Figure 7: T_p for each model and using all the implemented methods

The results featured in Figure 8 show the execution times T_e after the preprocessing has already been done. The expected execution time for three-dimensional grids is *O(1)* so it is constant and executed the fastest, but the result is only a approximation. The kd-tree build with the SAH heuristic performed best in almost all tested cases when the results were expected to be accurate. It was outperformed by octree in *gren2* where the scene was heavily divided.

Figure 9 shows the results, where each column represents the sum of T_p and T_e for each method implemented on the tested scenes. The kd-tree using the SAH heuristic *KD SAH* perfomed the best in all scenes. The kd-tree build with the half split criteria *KD HS* performed better than the octree in almost all cases except on the *gren1* scene where it was outperformed by a small margin. Even if octree traversal is sometimes done even faster than the traversal of kd-trees, the preprocessing phase is always slower and

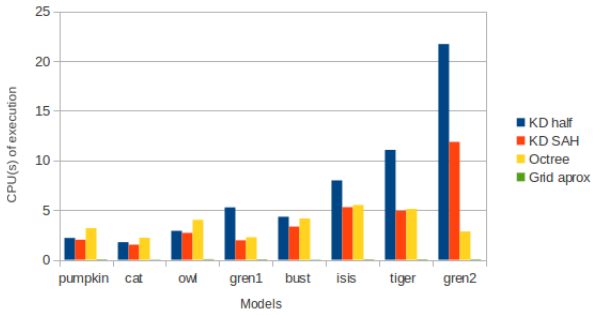


Figure 8: T_e for each model and using all the implemented methods

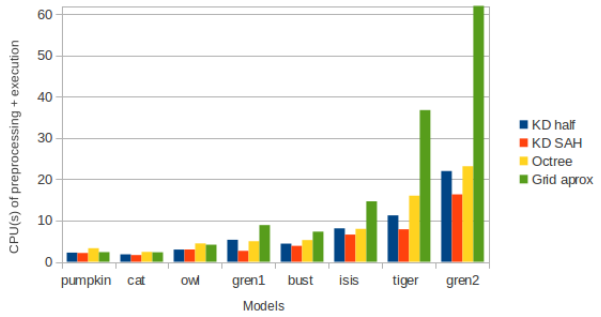


Figure 9: $T_p + T_e$ for each model using all the implemented methods

that is why the total performance of kd-trees is usually better.

8 Conclusion

In this paper, three different methods were used for solving the problem of inclusion using spatial subdivision as preprocessing. Our conclusions support the ones of [6], namely that for a small number of rays cast, data preprocessing does not pay off. After testing the mentioned scenes, the kd-tree using the SAH heuristic gave the best results, except for densely occupied scenes where SAH could not be fully utilized. There is still much that can be done regarding preprocessing, such as cutting off empty space in kd-trees or the usage of an octree special variant called octree-R. Therefore no final answer can yet be given regarding the fastest method for the inclusion test within three-dimensional spaces.

References

[1] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufman Publishers, 2005.

[2] F. Feito and J. Torres. Inclusion test for general polyhedra. *Computers and Graphics, Volume 21 Issue 1*, pages 23–30, 1997.

[3] J. Foley, A. van Dam, S. Feiner, and J. Hudgens. *Computer graphics: principles and practice in C (2nd ed.)*. Addison-Wesley Professional, 1996.

[4] M. Gombosi and B. Žalik. Point-in-polygon tests for geometric buffers. *Computers and Geosciences, Volume 31 Issue 10*, pages 1201 – 1212, 2005.

[5] M. Hapala and V. Havran. Review: Kd-tree traversal algorithms for ray tracing. *Computer Graphics Forum, Volume 30 Issue 1*, pages 199–213, 2011.

[6] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, 2000.

[7] P. Heckbert. *Graphics Gems IV*. Morgan Kaufman Publishers, 1994.

[8] E. Jansen. Data structures for ray tracing. *Data Structures for Raster Graphics*, pages 57–73, 1986.

[9] D. Kirk. *Graphics Gems III*. Morgan Kaufman Publishers, 1994.

[10] J. Li, W. Wang, and E. Wu. Point-in-polygon tests by convex decomposition. *Computers and Graphics, Volume 31, Issue 4*, pages 636–648, 2007.

[11] J. MacDonald and K. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer: International Journal of Computer Graphics, Volume 6 Issue 3*, pages 153–166, 1990.

[12] K. Ooms, P. De Maeyer, and T. Neutens. A 3d inclusion test on large dataset. *Developments in 3D Geo-Information Sciences*, pages 181–199, 2010.

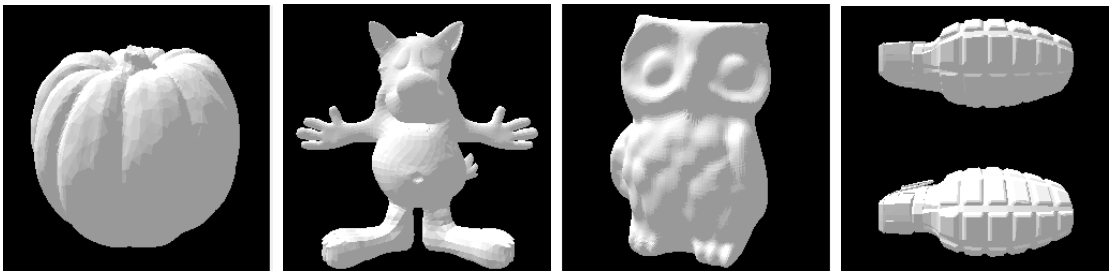
[13] J. Revelles, C. Urea, and M. Lastra. An efficient parametric algorithm for octree traversal. In *WSCG'00*, pages –1–1, 2000.

[14] S. Thon, G. Gesquire, and R. Raffin. A low cost antialiased space filled voxelization of polygonal objects. *GraphiCon 2004 proceedings, Moscou*, pages 71–78, 2004.

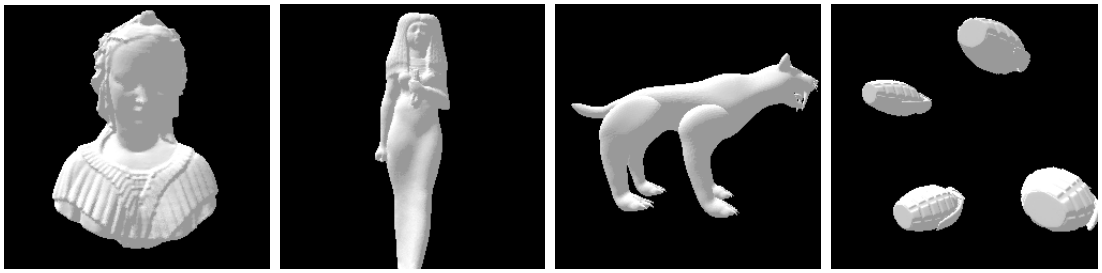
[15] B. Žalik and I. Kolingerova. A cell-based point-in-polygon algorithm suitable for large sets of points. *Computers and Geosciences, Volume 27 Issue 10*, pages 1135 – 1145, 2001.

[16] W. Wang, J.Li, H. Sun, and Enhua Wu. Layer-based representation of polyhedrons for point containment tests. *Ieee transactions on visualization and computer graphics, Volume 14, Issue 1*, pages 73 – 83, 2008.

[17] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12, 1992.



(a) pumpkin (b) cat (c) owl (d) gren1



(e) bust (f) isis (g) tiger (h) gren2

Figure 10: Thumbnails of the used models