Soft Shadow Volumes for Ray Tracing with Frustum Shooting

Thorsten Harter*
Chalmers University of Technology
University Karlsruhe

Markus Osswald†
Chalmers University of Technology
University Karlsruhe

Ulf Assarsson‡
Supervisor
Chalmers University of Technology

Abstract

We present a new variant of the algorithm of [Laine et al. 2005] for rendering physically-based soft shadows in ray tracing-based renderers. Our method replaces the internal acceleration data structure, a variant of the hemicube [Cohen and Greenberg 1985], to store potential silhouette edges. Instead we use a kd-tree [Havran 2000] and different building, storing and accessing techniques. Compared to the original algorithm, these changes lower the memory consumption of the soft shadow calculation, but increase the running time. We tested all our modifications for running time, memory consumption and efficiency.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Shadowing; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

Keywords: shadow algorithms, visibility determination

1 Introduction

In this paper we describe our attempt to replace the hemicube in the soft shadow volume algorithm of [Laine et al. 2005] with frustum shooting from the point to be shaded to the rectangular area light source. We tested this different approach to find the silhouette edges that affect visibility regarding computing time and memory consumption.

The basic idea is to use a space partitioning of the scene not only for increasing ray tracing efficiency, but also for generating soft shadows. Space is divided recursively by planes into non-overlapping regions, so that any point in space can be identified to lie in exactly one of the regions. Those regions form a hierarchy, a space-partitioning tree. We decided to use a 3-dimensional kd-tree as data structure, which uses only splitting planes perpendicular to the coordinate axes to divide space. The advantage of this is that the resulting regions form axis-aligned bounding boxes. Those are very easy to handle, e.g. the intersection with each other, with a ray or with the frustum planes. Each node of the tree is assigned the position and the extensions of an axis-aligned bounding box and a list of all edges that are possible silhouette edges from the light source and that are completely or partly contained in this box. This replaces building penumbra wedges for all possible silhouette edges and storing them into a hemicube-like data structure.

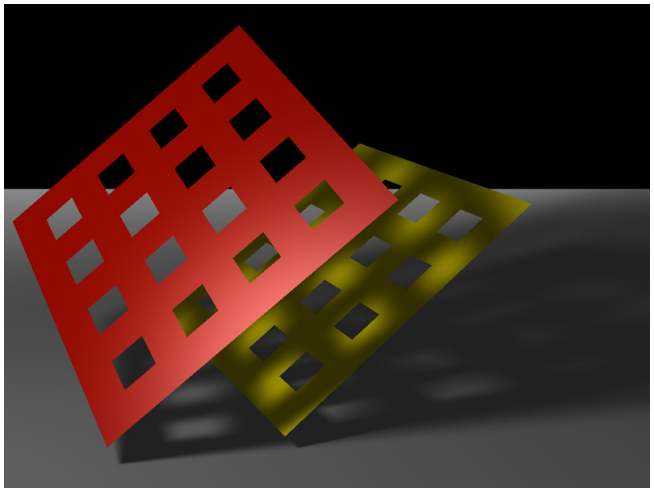


Figure 1: This image was rendered using our variant of the soft shadow volume algorithm with frustum shooting.

In Section 3.1 we describe how we build the kd-tree, in particular how we decide where to place the splitting planes. We tried two different methods and analysed their practicability.

Section 3.2 explains how we shoot a frustum during shading to get a list of possible silhouette edges. The data structure of an edge contains the plane equations of the two adjacent triangles. This allows to test if it is an silhouette edge from the point to be shaded. The reconstruction of the visibility function is done exactly as described in [Laine et al. 2005].

We compare five different versions of our new and the original algorithm and analyse the differences in Section 4. Only one version (0.6) uses less memory then the original algorithm while still achieving acceptable rendering times.

2 Previous Work

This section concentrates mainly on the explanation of the soft shadow volume algorithm presented by [Laine et al. 2005]. The method replaces the hundreds of shadow rays commonly used in stochastic ray tracers with a single shadow ray and a local reconstruction of the visibility function. Compared to tracing shadow rays, this approach gives a clear

*e-mail: thorsten.harter@web.de
†e-mail: markus_osswald@web.de
‡e-mail: uffe@ce.chalmers.se

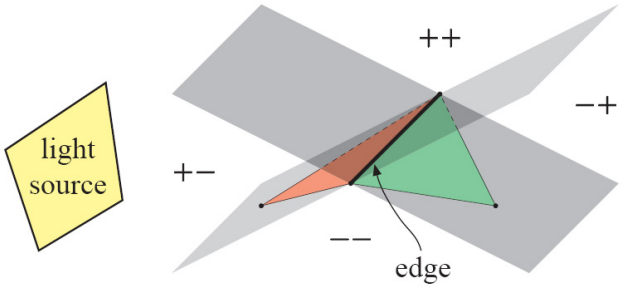


Figure 2: Determining silhouette edges from the light source. The planes of two triangles connected to an edge define four subspaces. If the light source lies entirely inside the – or ++ subspace, the edge cannot be a silhouette edge. Otherwise, the edge is a potential silhouette edge, as in the case depicted in the figure. Courtesy of [Laine et al. 2005]

speedup.

The algorithm can be divided into two different stages. In the pre-processing stage, silhouette edge information is extracted from the whole scene and stored into a static acceleration structure. This structure is a variant of the hemicube [Cohen and Greenberg 1985]. The second stage describes the shadow query for a point **p** in the scene. In this stage, first a list of penumbra wedges of all potential silhouette edges is created by projecting the point **p** on the surface of the hemicube. A penumbra wedge is the bounding volume of the penumbra region defined by a silhouette edge. After that, the edges in the list are worked off by checking each for being a silhouette edge from **p**. The ones that pass the test are projected onto the area light source. Now only one shadow ray from the area on the light source with the lowest depth complexity is checked for occlusion to decide how much of the light source is visible from point **p**.

2.1 Acceleration structure

To understand the differences between our new method of saving the potential silhouette edges in the pre-processing stage and the method of using a variant of the hemicube [Cohen and Greenberg 1985] needs a detailed view on the old data structure. The relevant silhouette edges to store in the hemicube must satisfy the criterion to be a silhouette edge from at least one point on the light source. If this criterion does not apply, then the edge can not be silhouette edge from any point of the scene. For this, the area light vertices have to be checked against the triangle planes of the triangles connected to the edge as illustrated in Figure 2. If there is only one triangle connected to the edge it must always be considered as a potential silhouette edge.

After constructing penumbra wedges for the potential silhouette edges using a robust algorithm [Assarsson and Akenine-Möller 2003], the hemicube footprint is conservatively rasterised into the hemicube as illustrated in Figure 3. After rasterisation, each cell of the hemicube contains a list of wedges whose footprint intersects the cell.

In practice, the memory consumption of the hemicube may grow to an unacceptable level. To conserve memory each face of the hemicube is represented by a multiresolution grid.

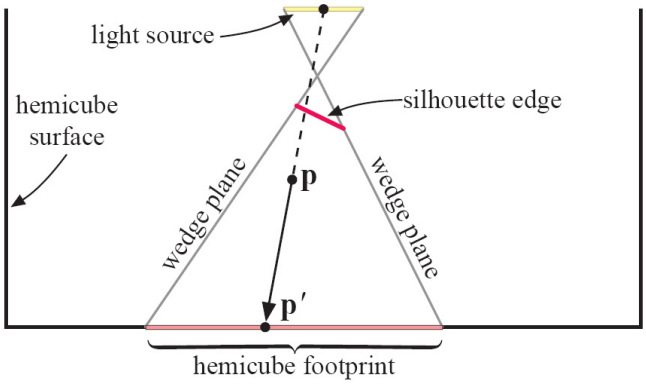


Figure 3: 2D illustration of how the hemicube footprints of penumbra wedges can be used for deciding whether the corresponding silhouette edge may overlap the light source from a given point. The intersection of the wedge and the surface of the hemicube is the hemicube footprint of the wedge. To determine if point **p** may be inside the wedge, the point is projected onto the surface of the hemicube from the center of the light source. If the projected point **p'** is inside the hemicube footprint of the wedge, point **p** may be inside the wedge. Otherwise point p is guaranteed to be outside the wedge, and consequently the silhouette edge does not overlap the light source from **p**. Courtesy of [Laine et al. 2005]

2.2 Finding silhouette edges from a point

In this part, the second stage including the execution of the shadow query is explained in detail. The shadow query is executed for a point **p** and it returns as result which light samples are visible from that point. At first, **p** is projected onto the surface of the hemicube to find out in which wedges **p** is situated. A list of edges is created by collecting the wedges from all levels of the multiresolution grid. After that, it has to be decided which ones are silhouette edges from **p** and overlap the light source. The first test is to check if the edge is indeed inside the wedge, which corresponds to testing that the edge E overlaps the light source from **p**. After that it has to be checked if edge E is a silhouette edge from **p** by testing the point against the triangle planes of the triangles connected to edge E similar as illustrated in Figure 2. If there is only one triangle connected to the edge, it is always a silhouette edge.

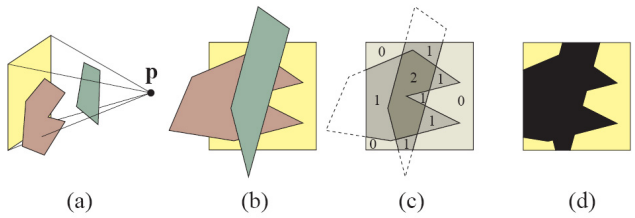


Figure 4: Illustration of the depth complexity function. (a) Two occluders are located between point p and the light source. (b) The occluders projected on the light source from p. (c) The depth complexity function tells the number of surfaces that overlap a point on the light source. (d) The visibility function is reconstructed from the depth complexity function. Courtesy of [Laine et al. 2005]

2.3 Local reconstruction of the visibility function

The second part of the shadow query is to decide which samples are visible from point **p**. Therefore, the silhouette edges from **p** are projected onto the area light source. These projected silhouette edges now represent relative changes of the depth complexity function. The depth complexity function of a point **s** on the light source is defined as the number of surfaces a ray from **p** to **s** intersects (Figure 4). Point **s** is only visible if the depth complexity of **s** is 0. It is now possible to calculate the depth complexity for each light sample on the light source. After that, only one reference ray from a light sample with the lowest depth complexity to **p** is needed. If it is occluded, all other light samples are also occluded. Otherwise the samples with the lowest depth complexity are visible.

3 Frustum Shooting

In this section we will give a detailed description of our algorithm and where it differs from the hemicube algorithm. The pseudocode in Figure 5 gives a short overview.

```
    CONSTRUCT KD-TREE
 1  build an AABB with the extents of the scene
 2  add all triangles to the triangle list of the root node
 3  for each edge E
 4    if E is a potential silh. edge from the light source
 5      add E to the edge list of the root node
 6    end if
 7  end for
 8  while termination criteria not met
 9    subdivide root node recursively (split boxes, triangle
        lists and edge lists)

    SHADOW-QUERY(point p)
10  clear depth complexity counters of light samples
11  intersect frustum from p to the area light with the
      kd-tree
12  L_E ← list of edges in the frustum
13  for each edge E in L_E
14    if E is a silhouette edge from p
15      project E onto the surface of the light source
16      update depth complexity counters of light samples
17    end if
18  end for
19  cast a shadow ray to a light sample with lowest depth
      complexity
20  if ray is blocked
21    return all light samples are hidden
22  else
23    return light samples with lowest depth
        complexity are visible
24  end if
```

Figure 5: A short description of the changed soft shadow volume algorithm. CONSTRUCT KD-TREE builds a space-partitioning data structure which stores the edges in the scene. SHADOW-QUERY finds relevant edges by shooting a frustum which intersects the kd-tree.

Instead of storing penumbra wedges in the hemicube in the preprocessing stage, we construct a kd-tree and store for each node a list of edges which are contained in its axis-aligned bounding box. Edges that can not be silhouette edges from any of the area lights in the scene are discarded. When the shadow query is executed, instead of projecting the point **p** to be shaded on the hemicube, we shoot a frustum from **p** to the four corners of the area light source. By intersecting this frustum with the boxes of the kd-tree we get a list L_E of all the edges in the frustum. This list is then processed as described in Section 2.3. In Section 3.1 the construction of the kd-tree is described and Section 3.2 explains how the frustum shooting is done.

3.1 Construction of the kd-tree

A kd-tree (short for k-dimensional-tree) is a space partitioning data structure. kd-trees are a special case of BSP trees. While arbitrary splitting planes can be used in a BSP tree, a kd-tree uses only splitting planes perpendicular to the coordinate axes.

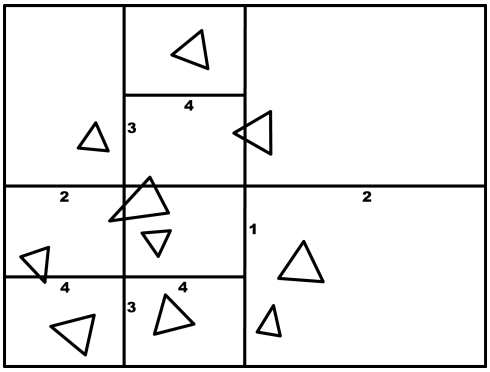


Figure 6: Visualisation of the simple subdivision algorithm in 2 dimensions: Each box is split exactly in half until there are no more than two triangles in a box. If a triangle is intersected by a splitting line it is assigned to the lists of both resulting boxes. The numbers at the lines indicate the level of recursion. In this example eight subdivisions occur. The resulting tree has a depth of five and nine leaves.

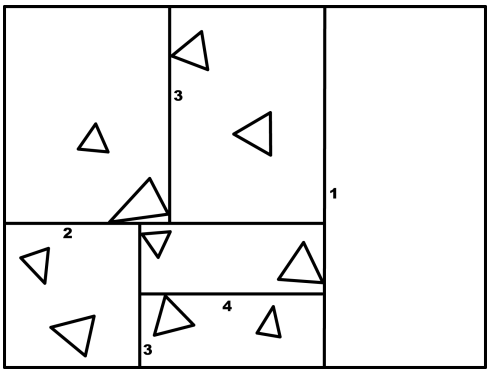


Figure 7: The surface area heuristics is supposed to isolate geometry from empty space. Each box is split at the splitting position with minimum cost. Since this method always chooses a splitting position at a triangle vertex, it decreases the risk of intersecting a triangle and thus storing it in more than one list. In this example five subdivisions occur. The resulting tree has a depth of five and six leaves.

The root node of our kd-tree is assigned an axis-aligned bounding box with the extents of the scene, a list of all triangles and a list of all possible silhouette edges. For deciding if an edge is a possible silhouette edge from the light source we use the method explained in Figure 2. Depending on the scene this allows us to discard a high percentage of the edges and thus save memory and computing time. The root node invokes the recursive subdivide method which creates two child nodes. The axis-aligned bounding box is split in two and each of the two child nodes is assigned to one of the resulting boxes. Each triangle and edge in the list of the parent node is tested against the splitting plane and then assigned to the list of the proper node. If a triangle or a edge intersects the splitting plane it will be assigned to the lists of both nodes. Then, the two child nodes are processed recursively until one of the termination criteria is met.

We used the following two criteria:

1. number of triangles in the box < given number
2. size of the box < given number

By trying different values for these two termination criteria, depth and size of the kd-tree can be changed. This allows to find a good compromise between construction time of the tree and rendering time.

There are different approaches to choose the splitting axis and the position of the splitting plane on the axis. First we used a very simple strategy: The box is split in half on the axis on which it has its largest extension. A visualisation of this is shown in Figure 6.

Then we tried something more sophisticated to determine the position of the splitting plane. The surface area heuristic as described in [Havran 2000] should help to isolate geometry from empty space and thereby decrease intersection tests. Figure 7 shows its application on a 2-dimensional example and can be compared to the simple method shown in Figure 6.

For defining the splitting cost we first calculate the surface area SA of a box, which determines the chance that this box is hit by a ray:

$$SA = 2 * (width * height + height * depth + depth * width)$$

The cost of splitting a box at a particular position is calculated by adding the costs of the two resulting boxes. The following formula calculates the cost for splitting a box:

$$splitCost = traversalCost + (SA(lchild) * prims(lchild) + SA(rchild) * prims(rchild))/intersectionCost$$

where:
$traversalCost = 0.3$
$SA(node) = surface\ area\ of\ this\ node$
$prims(node) = number\ of\ primitives\ of\ this\ node$
$intersectionCost = SA(parent\ node)$

To find the optimal split plane position, we determine the minimum of splitCost for all possible splitting positions. We are searching for the optimal position at the boundary of a primitive, since the number of primitives does not change between the boundaries of two primitives. Therefore we use a list of the positions of all the vertices in the box on the splitting axis. Despite this restriction it is an immense effort to determine the surface area and number of primitives of the resulting boxes for each splitting position candidate.

Besides the two previously mentioned termination criteria there is now a third one.

3. splitCost < cost for leaving

If the cost for splitting a box is higher than leaving it as it is, it will not be split. We used the number of primitives in the box as cost for not splitting.

In Chapter 4 we will discuss the results we got from using the two different approaches for building the kd-tree.

3.2 Intersection of Frustum and kd-tree

For every point **p** to be shaded and for every area light source, we have to determine the edges which can be possible silhouette edges.

Therefore, we intersect the frustum from **p** to the four corners of the area light source with the boxes of the kd-tree, as shown in Figure 8. We begin with the box of the root node and then walk through the tree. If the box of an inner node is intersected by the frustum, one child node is put on a stack and the other child node is tested next. If the box of a leave node is intersected or if a box is completely contained in the frustum, the edge list of this node is added to the list of possible silhouette edges L_E and the next node is taken from the stack. If the box is completely outside of the frustum, we continue with the next node from the stack. The pseudocode for this method is shown in Figure 9.

The point **p** is inside one of the boxes. Thus, there will be at least one box which is intersected by or contained in the frustum every time, but since not all boxes contain possible silhouette edges the returned edge list L_E may be empty.

For the box-frustum intersection test we used the algorithm described by [Assarsson and Möller 2000]. For each of

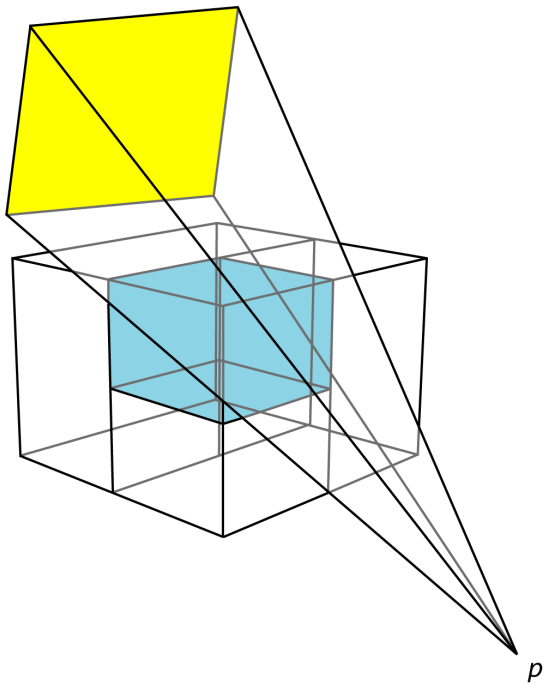


Figure 8: The frustum from the point **p** to the rectangular light source intersects the marked box. All the edges in this box will be added to L_E, although not all of them are inside the frustum.

```
    GET EDGES IN FRUSTUM(point p)
 1  current node ← root node
 2  while nodes left
 3    while current node is inner node
 4      get box of current node
 5      execute box frustum intersection test
 6      if box inside frustum
 7        add edgelist of current node to L_E
 8        current node ← pop from stack
 9      else
10        if box intersects frustum
11          current node ← get left child node
12          put right child node on stack
13        else //box outside frustum
14          if stack empty
15            return
16          else
17            current node ← pop from stack
18          end if
19        end if
20      end if
21    end while
22    //current node is leaf
23    execute box frustum intersection test
24    if box intersects frustum or box inside frustum
25      add edge list of current node to L_E
26    end if
27    current node ← pop from stack
28  end while
```

Figure 9: This method walks through the kd-tree and adds the edges of all the boxes that are inside the frustum or intersected by the frustum to L_E.

the five frustum planes we do a box-plane intersection test, which can be done by testing the two end points of the diagonal most closely aligned with the plane's normal, the so called n- and p-vertices.

Finding the n- and p-vertices can be done in 3 comparisons respectively:

```
for each axis
  if planeNormal[axis] > 0
    n-vertex[axis] ← min[axis]
  else
    n-vertex[axis] ← max[axis]

for each axis
  if planeNormal[axis] > 0
    p-vertex[axis] ← max[axis]
  else
    p-vertex[axis] ← min[axis]
```

where:
min[axis] = minimum coordinate of the box on axis
max[axis] = maximum coordinate of the box on axis

First the n-vertex is inserted in the plane equation and if it is outside, the box is outside the plane. If it is inside, the p-vertex is tested. If it is inside, the box is inside the plane, otherwise it intersects the plane.

With this method we test the box against the for side planes of the frustum and the plane of the area light. If the box is outside one of the five planes it is outside the frustum. If it is inside all five planes, it is inside the frustum, otherwise it intersects the frustum.

After all edges in the frustum have been found, every edge is tested if it is a possible silhouette edge from the point to be shaded. The resulting edge list is then processed exactly as described for the original algorithm in Chapter 2.

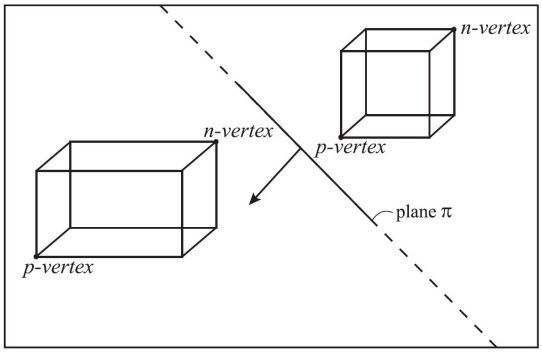


Figure 10: The negative far point (n-vertex) and positive far point (p-vertex) of a bounding box corresponding to plane π and its normal. Courtesy of [Assarsson and Möller 2000]

4 Results

Comparison method We compared our implementation against the original one of the soft shadow volumes algorithm using a variant of the hemicube [Cohen and Greenberg 1985] based on our own implementation of a raytracer using a kd-tree to store and fast access the scene primitives. We tested different versions and methods against the original implementation:

Version 0.4 gets the potential triangles in the frustum from the kd-tree leaves which contain all triangles of the scene and then constructs edges out of the triangles.

Version 0.5 first constructs all edges of the scene before building the kd-tree and stores them in the leaves of the kd-tree. It gets the edges in the frustum from the leaves.

Version 0.51 stores the edges also in non-leaves which accelerates the tree traversal, but increases the memory consumption.

Version 0.6 stores only possible silhouette edges in the kd-tree and uses a faster box-frustum intersection. The edges found in the frustum (pseudocode in Figure 9) are no longer stored in an array. Instead we use a set with hash table to accelerate insert and find operations.

Version 0.61 stores the potential edges also in non-leaves which accelerates the tree traversal, but increases the memory consumption.

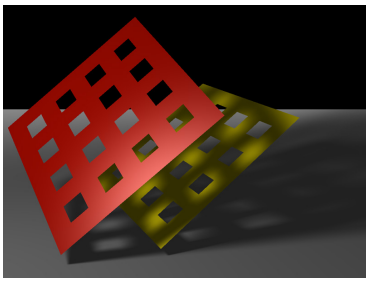
Scenes and setup We tested our variant of the soft shadow volumes algorithm in five scenes, shown in Figure 11. The Grids scene is a geometrically simple scene (262 triangles), but it has big penumbra regions. Another characteristic of the scene is that all silhouette edges are single edges, which

Scene	Resolution	# Triangles	kd-Tree Depth	# Leaves
Grids	800x600	262	17	163
Objects	800x600	4.520	23	4.738
Temple	800x600	39.599	23	16.323
Cubes	800x600	108.000	23	90.452
Ball	800x600	229.088	22	137.750

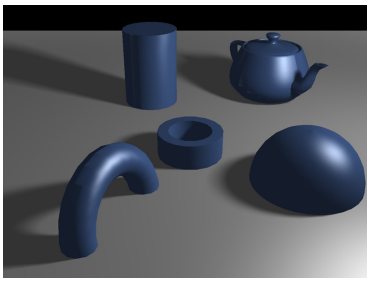
Table 1: Details of test scenes

	No Shadow	Hemi.	Frust. 0.4	Frust. 0.5	Frust. 0.51	Frust. 0.6	Frust. 0.61
Grids							
Build Tree	0 s	0 s	0 s	0 s	0 s	0 s	0 s
Area Lights	0 s	0 s	0 s	0 s	0 s	0 s	0 s
Rendering	2 s	7 s	28 s	22 s	22 s	23 s	23 s
Sum	2 s	7 s	28 s	22 s	22 s	23 s	23 s
Comparison	**0.3**	**1.0**	**4.0**	**3.1**	**3.1**	**3.3**	**3.3**
Memory (MB)	-	1.2	0.7	0.7	0.9	0.7	1.0
Non-Silh.	-	0%	-	-	100%	0%	0%
Objects							
Build Tree	0 s	0 s	0 s	0 s	0 s	0 s	0 s
Area Lights	0 s	1 s	0 s	0 s	0 s	0 s	0 s
Rendering	2 s	6 s	28 s	22 s	22 s	23 s	23 s
Sum	2 s	7 s	28 s	22 s	22 s	23 s	23 s
Comparison	**0.3**	**1.0**	**4.0**	**3.1**	**3.1**	**3.3**	**3.3**
Memory (MB)	-	1.6	0.8	2.2	5.2	1.0	1.4
Non-Silh.	-	60%	-	-	93%	76%	76%
Temple							
Build Tree	1 s	1 s	0 s	2 s	1 s	1 s	1 s
Area Lights	0 s	1 s	0 s	0 s	0 s	0 s	0 s
Rendering	2 s	9 s	334 s	468 s	396 s	58 s	72 s
Sum	3 s	11 s	334 s	470 s	397 s	59 s	73 s
Comparison	**0.3**	**1.0**	**30.4**	**42.7**	**36.1**	**5.4**	**6.6**
Memory (MB)	-	6	0	11	38	2	7
Non-Silh.	-	51%	-	-	91%	50%	50%
Cubes							
Build Tree	3 s	4 s	3 s	7 s	5 s	4 s	4 s
Area Lights	0 s	0 s	0 s	0 s	0 s	0 s	0 s
Rendering	7 s	13 s	243 s	169 s	156 s	57 s	57 s
Sum	10 s	17 s	246 s	176 s	161 s	61 s	61 s
Comparison	**0.6**	**1.0**	**14.5**	**10.4**	**9.5**	**3.6**	**3.6**
Memory (MB)	-	7	0	30	95	4	10
Non-Silh.	-	0%	-	-	87%	7%	7%
Ball							
Build Tree	5 s	5 s	4 s	17 s	7 s	8 s	6 s
Area Lights	0 s	3 s	0 s	0 s	0 s	0 s	0 s
Rendering	15 s	51 s	563 s	232 s	235 s	176 s	421 s
Sum	20 s	59 s	567 s	249 s	242 s	184 s	427 s
Comparison	**0.3**	**1.0**	**9.6**	**4.2**	**4.1**	**3.1**	**7.2**
Memory (MB)	-	34	0	65	208	19	61
Non-Silh.	-	26%	-	-	74%	29%	29%

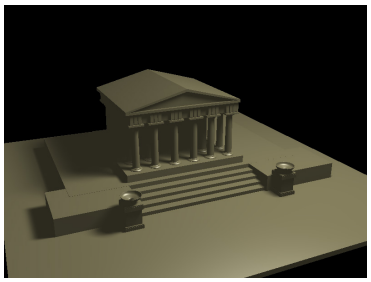
Table 2: This table shows the test results for the five different test scenes. The measured times are the time to build the kd-tree, to build the area lights including building the hemicube and the time for rendering the image. The Memory row specifies how much memory is needed for the soft shadow calculation. The non-silhouette ratio is an indicator for the number of edges which are found in the hemicube/frustum as potential silhouette edges but are discarded. This happens when the result of the silhouette test similar to the test in Figure 2 of the point to be shaded against the planes of the adjacent triangles of this edges is negative.



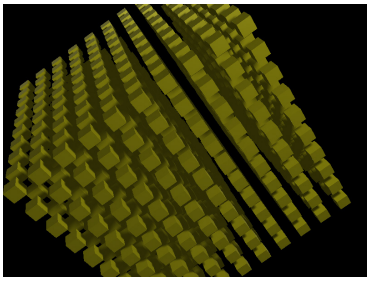
(a) Grids



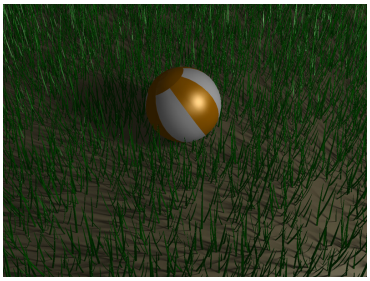
(b) Objects



(c) Temple



(d) Cubes



(e) Ball

Figure 11: Test scenes

Scene	Method	Frust 0.51	Frust. 0.6	Frust 0.61
Grids	shadow query	19 s	20 s	19 s
	edge validation	0.5 s	2.2 s	2.2 s
	cast reference ray	2.6 s	2.6 s	2.5 s
	box/frustum intersection	9.7 s	6.0 s	5.9 s
	add edges	3.1 s	5.7 s	5.8 s
	# added edges (10^6)	-	14.2	14.2
Objects	shadow query	34 s	21 s	20 s
	edge validation	0.8 s	1.0 s	1.0 s
	cast reference ray	2.5 s	2.5 s	2.5 s
	box/frustum intersection	21.0 s	12.4 s	12.0 s
	add edges	6.6 s	2.5 s	2.5 s
	# added edges (10^6)	-	7.08	7.03
Temple	shadow query	393 s	56 s	69 s
	edge validation	1.7 s	5.0 s	7.2 s
	cast reference ray	1.6 s	1.6 s	1.5 s
	box/frustum intersection	25 s	14 s	14 s
	add edges	358 s	25 s	43 s
	# added edges (10^6)	-	23.3	20.5
Cubes	shadow query	149 s	51 s	50 s
	edge validation	2.0 s	2.3 s	2.2 s
	cast reference ray	4.3 s	4.3 s	4.2 s
	box/frustum intersection	45 s	33 s	32 s
	add edges	86 s	6.4 s	6.3 s
	# added edges (10^6)	-	13.5	13.3
Ball	shadow query	219 s	159 s	405 s
	edge validation	5.6 s	16 s	18 s
	cast reference ray	18 s	19 s	18 s
	box/frustum intersection	93 s	56 s	55 s
	add edges	80 s	53 s	299 s
	# added edges (10^6)	-	95.4	95.2

Table 3: Running time of the most important shadow query methods in detail

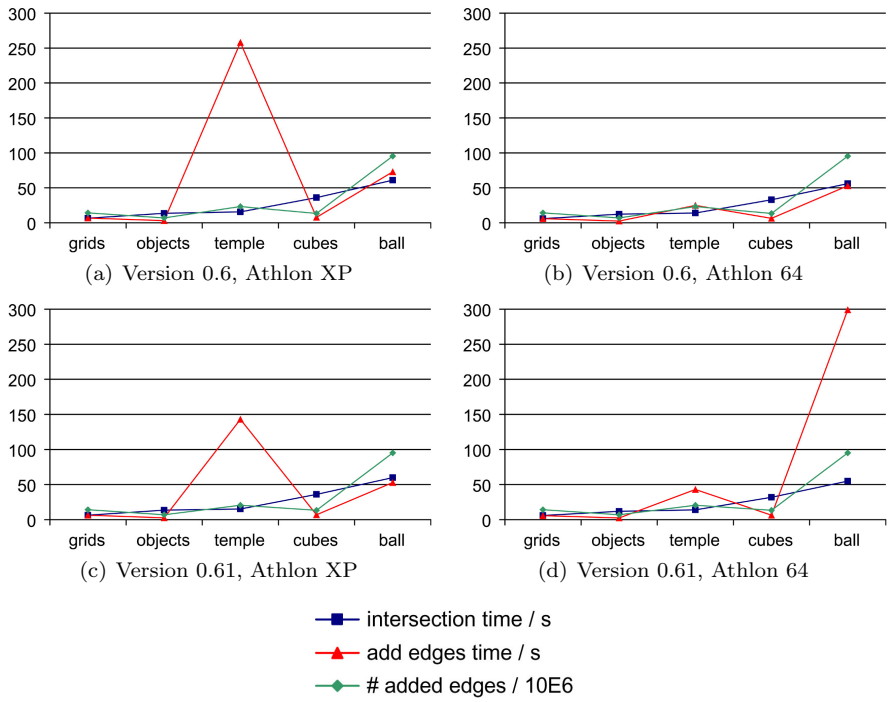


Figure 12: Comparison of the computing times for the box-frustum intersection method and the add edges method on two different processors. The time for the box-frustum intersection method corresponds approximately to the number of triangles in the scene. The time for the add edges method corresponds in most cases to the total number of edges that have been added but depending on the used processor it differs greatly from the expected result.

means that they are only connected to one triangle. The Objects scene is a little bit more complex (4.5k triangles) and contains single edges as well as double edges. Temple (40k triangles) and Cubes (108k triangles) slowly increase the number of triangles to store in the acceleration data structure. The most complex scene is Ball (230k triangles). It consists of a huge number of objects which all throw a soft shadow.

In all scenes, we deactivated the ray casting recursion for specular materials to concentrate on the calculation of the soft shadows. The scenes are all of the same size so that the penumbra regions of similar area light sources are sized equal.

The performance measurements were run on a 2.0GHz Athlon64 3200+ in 32bit mode with 1 GB of memory. As output resolution we choose 800x600 to compromise between having useful results and keeping the execution time manageable.

As characteristics for the kd-tree we chose as termination criteria for the subdivision algorithm less than ten triangles in a box or size of a box less than 0.5% of the size of the biggest, the scene box. We tried several other settings and came to the conclusion that this settings are the best compromise between speed and memory consumption. For determining the splitting position, we had to use the simple split-in-half strategy. The surface area heuristics speed up rendering time by 5 - 10%, but the time for constructing the kd-tree grew exponentially and the method was therefore not applicable for bigger scenes. Table 1 gives details about each scene and its kd-tree.

Performance analysis Table 2 lists the performance measurements from our test scenes. Generally, we had to realise that we could not achieve the performance of the original implementation. The best result was obtained with Version 0.6 in the Ball scene where our modification was only 3.1 times slower than the original algorithm. Positive was that we lowered the memory consumption for the soft shadow calculation of this scene by 44%.

Version 0.4: Our first approach yielded a very bad result. In the simple scenes, the runtime is acceptable. However, if the scenes become more complex, meaning that the kd-tree contains more leaves, the fact that the edges always have to be extracted from the triangles leads to a huge penalty. The only positive aspect of this version is that we do not use any extra memory for the edges because the triangles are already stored in the kd-tree for the ray intersection.

Version 0.5: Because of this bad result, we decided to shift the extraction of the edges and generation of the planes of the connected triangles to the kd-tree buildup. This has the consequence that we now also have to save the edges in the nodes of the kd-tree which leads to a memory consumption that is higher than that of the hemicube. As expected, the runtime decreased: e.g. for the Cubes scene from 14.5 times to 10.4 times the runtime of the hemicube implementation. The only thing left to do when finding a node whose box intersects the frustum is to add the edges of this node to an array and to check for the array for double added edges. A surprising result is that in the Temple scene the runtime increases by about 35%.

Version 0.51: To speed up the collecting of the edges in the frustum we decided not to delete the edge lists in the non-leaf nodes of the kd-tree. This has the effect that we do not have to collect the edge lists of the child leaves of a node that is completely inside the frustum. Another positive consequence is that we now have less double added edges

because it is not unusual that in the subdivision method an edge is added to both child nodes. A negative effect is that we increased the memory consumption by up to 3.5 times. But the results of the runtime measurements could not verify our assumption. If there was a speedup it was so small that it could not justify the increased memory consumption. We also activated the statistics of the soft shadow volumes library and added some new outputs to analyse the efficiency of our algorithm. The non-silhouette ratio is an indicator for the number of edges which are found in the hemicube/frustum as potential silhouette edges but are discarded. This happens when the result of the silhouette test in Figure 2 is negative (i.e., testing the point to be shaded against the edge's adjacent triangle planes). The non-silhouette ratio for version 0.51 is with about 70% to 90% much higher than the ratio of the hemicube implementation. For the Grids scene the non-silhouette ratio is 100%, because each edge pair in the scene has two adjacent triangles that lie in one plane. Therefore, it will definitely fail the silhouette test. All silhouette edges of this scene are single edges. It is obvious that the reason for this inefficient behaviour of our algorithm is the missing pre-selection of potential silhouette edges of the scene.

Version 0.6 and Version 0.61: To enhance our implementation we added a pre-selection of potential silhouette edges to the extraction of the edges from the triangles. That means that for every edge the vertices of the rectangular area light source are checked against the planes of the adjacent triangles as illustrated in Figure 2. Now the kd-tree only contains edges which can be a silhouette edge from some point in the scene. As a result the memory consumption sank below that of the original implementation using the hemicube. The non-silhouette ratio is now close to that of the hemicube implementation. The running time is the lowest we could achieve with our approaches.

The modification from version 0.6 to 0.61 is completely analog to that of the version 0.5 to 0.51. Except for the Temple and the Ball scene the running times hardly change. The increasing memory consumption listed in Table 2 is out of all proportion to the number of added edges listed in Table 3. Hence, we regard version 0.6 as our best result.

Table 3 and Figure 12 show anomalies in the computing time of the add edges method, which transforms edges from world to light coordinates and inserts them into a set. In some cases the measured time was ten times higher than expected. Those results were reproducible. The only possible explanation we can think of is that cache thrashing occurs in those special cases.

Generally, we noticed that with this test scenes and this setup we got the best test results for the comparison with the hemicube implementation. If we enlarge the penumbra area by changing the distance to the area light source or the size of the light source the test results become worse. In some cases while the hemicube implementation had a runtime of a few seconds we waited in vain for any progress while running our implementation for several minutes.

5 Discussion and Future Work

Our alternative approach for the generation of soft shadows does not achieve the performance of the original algorithm. Even in the best suited test scenes the computing time is 3-times as long. The positive aspect is that the memory consumption has been reduced somehow.

The use of a kd-tree as the data structure for storing the edges is problematic since an unfavourable subdivision of the scene can result in longer computing times. Maybe another subdivision algorithm than the unflexible divide-in-half and the impracticable surface area heuristic algorithm could lead to better results. Since both ray tracing and soft shadow generation depend on the same kd-tree and therefore on the same spatial subdivision, it can be hard to find the optimal parameters. A kd-tree with small boxes that leads to short rendering times for the unshadowed scene might not be optimal for the scene with soft shadows, because many edges are stored in more than one box and will thus be processed several times.

Depending on the scene and on the processor on which the algorithm was running, the resulting computing times were in some cases surprising. In particular the method that transforms the edges that where found in the frustum from world coordinates to light coordinates and inserts them into the set showed a behaviour that we cannot explain. We can only assume that an optimisation of memory usage could lead to a better cache behaviour and further improve the performance.

Acknowledgements

Temple scene by Bruno Lomio, http://www.3dgrafix.net.

References

Akenine-Möller, T., and Assarsson, U. 2002. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 297–306.

Assarsson, U., and Akenine-Möller, T. 2003. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Trans. Graph. 22*, 3, 511–520.

Assarsson, U., and Möller, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools 5*, 1, 9–22.

Assarsson, U., Dougherty, M., Mounier, M., and Akenine-Möller, T. 2003. An optimized soft shadow volume algorithm with real-time performance. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 33–40.

Cohen, M. F., and Greenberg, D. P. 1985. The hemicube: a radiosity solution for complex environments. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 31–40.

Havran, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Laine, S., and Aila, T. 2005. Hierarchical penumbra casting. *Computer Graphics Forum 24*, 3, 313–322.

Laine, S., Aila, T., Assarsson, U., Lehtinen, J., and Akenine-Möller, T. 2005. Soft shadow volumes for ray tracing. *ACM Trans. Graph. 24*, 3, 1156–1165.