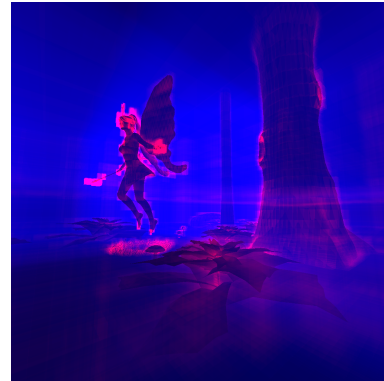
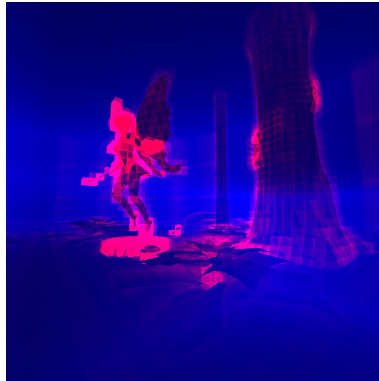


Compact Rectilinear Grids for the Ray Tracing of Irregular Scenes

Vasco Costa
INESC-ID/IST
Lisbon, Portugal
vasc@vimmi.inesc-id.pt

João Madeiras Pereira
INESC-ID/IST
Lisbon, Portugal
jap@vimmi.inesc-id.pt



Fairy Forest irregular scene rendering complexity

The regular grid spatial subdivision on the left provides a less well balanced triangle distribution per cell than the rectilinear grid structure on the right. Both spatial partitioning grids feature a similar resolution.

ABSTRACT

Regular grid spatial subdivision is frequently used for fast ray tracing of scanned models. Scanned models feature regular sized primitives, with a regular spatial distribution. Grids have worse performance, than other subdivision techniques, for irregular models without these characteristics. We propose a method to improve the performance of grids for rendering irregular scenes by allowing the individual placement of grid split planes: the rectilinear grid. We describe how to construct and traverse a rectilinear grid. To exploit cache memory in modern processors compression is used for the split plane and grid cell data. We demonstrate in a series of tests that the method has faster ray tracing rendering performance than a compressed regular grid of similar dimensions.

Keywords

Ray tracing, grids.

1. INTRODUCTION

Whitted ray tracing [Whi80] is a technique which is experiencing a renaissance in the graphics research community, since it is a simple, elegant algorithm which can accurately render not just local illumination, but also shadows, reflections, and refractions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The algorithm is also amenable for implementation into parallel architectures, because of its inherent per ray parallelism. In addition it can accurately render higher order object primitives such as spheres, cylinders, cones, or other quadrics. Not just triangles.

The recent improvements in graphics hardware and GPGPU programming languages have been empowering software developers everywhere to replace the rendering pipeline partially or even in its entirety. In order to be able to compete with rasterization, ray tracing must have good enough performance. It should be able to render the scenes which users expect to visualize today. It is the

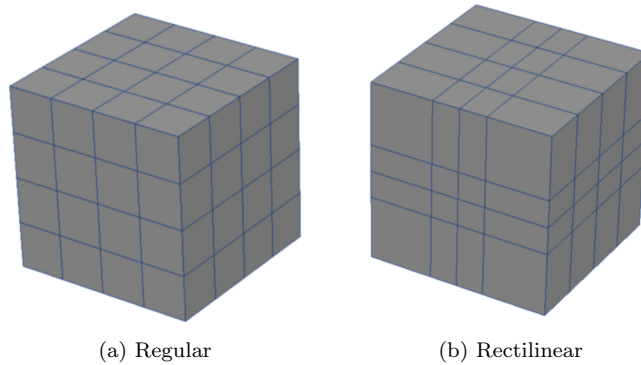


Figure 1. Grid spatial subdivision types

author’s opinion that is desirable to use ray tracing across the whole rendering pipeline as a replacement for rasterization. Barring the rendering performance reasons, which this work aims to address, using ray tracing across the whole pipeline would enable a more seamless experience for artists and application developers. One example is the application of shadows on a scene, where ray tracing does not suffer from the artifacts and difficult parametrization issues of current rasterization algorithms. The implementation issues of these algorithms are described by Kuehl et al. in [KBB07].

2. BACKGROUND

In order to have real-time ray tracing performance, for complex scenes, it is necessary to employ so called acceleration techniques. These techniques employ a divide-and-conquer strategy to solve the ray tracing problem. The most popular acceleration techniques are spatial subdivision techniques: bounding volume hierarchies (BVHs), kd-trees, and grids [WMG⁺07].

Grids are a 3D space subdivision method, introduced by Fujimoto in [FTI86], where space is subdivided into same sized cubically shaped cells. These cells also known in the literature as *voxels*. Grids have several interesting traits. They feature linear $O(N)$ construction time, constant $O(1)$ update time, and have fast traversal times for rendering. Best case traversal complexity is $O(1)$ and worse case traversal complexity is $O(\sqrt[3]{N})$. N is the number of objects in a scene.

The main issues with regular grids have been high memory consumption and poor adaptation to irregular scenes. Irregular scenes can feature different sized, irregularly distributed, geometry. One example of such troublesome geometry would be a highly detailed object inside a low detail box. This is known in the literature as the teapot in a

stadium problem. In contrast grids are very efficient at rendering scanned scenes.

Both of these issues are due to the way the splitting planes are positioned in a regular grid. All the cells must have the same size, so regular grids are a poor fit for irregular scenes. The heuristic used to compute the number of split planes for each axis of the 3D grid is usually some variation of:

$$\sqrt[3]{\rho \times N} \quad (1)$$

Where ρ is a user defined parameter which describes the density of the scene. Most implementations use a default ρ value of 4. This heuristic ensures the memory consumption is proportional to the number of primitives in the scene.

One approach to solve the issue is Jevans and Wyvill’s recursive grid [JW89]. In this approach grids are recursively applied to subdivide the scene in a shallow grid hierarchy. This technique places grids inside grids to enable variable subdivision cell sizes across the scene. This technique improves rendering performance substantially. The main issues with this approach are that it further increases memory consumption, makes it impossible to do $O(1)$ partial scene updates, and the method still has some issues adapting to irregular scenes compared to other techniques such as kd-trees and BVHs which employ surface area heuristics (SAH) to more accurately place the split planes [WH06, Wal07]. Much of the traversal time in the grids is still spent skipping empty cells with no geometry in them. Memory and time efficient methods to construct and traverse such recursive grids were described by Costa et al. [CPJ10].

Several researchers have tried in the past to improve regular grid heuristics with a limited degree of success. One approach, followed by Klimaszewski in [KS97], was to build a grid hierarchy overlaid on a previously constructed SAH bounding volume hierarchy. Cazals and Puech in

[CP97] analyzed the scene geometry by clustering geometry into groups and attempting to place the regions of regular geometry in the scene into separate grids. These techniques have not been very popular because of their implementation complexity, and a performance which is not globally better than that of the simpler to implement recursive grid. Havran did an interesting performance analysis in [HS99] of these grid spatial subdivision techniques. Ize provides in [ISP07] an in depth analysis of grid heuristics. There he describes how to build a good performing grid for several kinds of scenes. Ize also described an heuristic for recursive grids which maintains total space consumption proportional to the number of primitives in the scene. This recursive grid heuristic is included in the Manta interactive raytracer [BSP06].

3. RECTILINEAR GRIDS

We propose to relax the grid split plane positioning using a rectilinear grid (see Figure 1). This should enable a better balancing of the scene geometry among the grid cells resulting in faster ray tracing performance. Rectilinear grids have previously been used in the field of volume ray tracing [PPL⁺05]. In this work we describe how to efficiently construct and traverse a rectilinear grid for general ray tracing purposes.

Construction

First we compute the scene's bounding box. Next we finely sample the scene along each major axis x, y, z . The number of samples taken per axis is described by the following equation:

$$sample_i = 100 \times S_i \times \sqrt[3]{\frac{4 \times N}{V}} \quad i \in x, y, z \quad (2)$$

Where N is the number of primitives, V is the volume of the scene, and S_i contains the bounding box dimensions in that axis. Each sample contains a count of the number of primitives in that particular subvolume.

Then we compute the sum of these primitive counts for each axis:

$$sum_i = \sum_{j=0}^{sample_i} count_j \quad i \in x, y, z \quad (3)$$

The scene is partitioned into $ncells_x \times ncells_y \times ncells_z$ voxels. The split planes for each cell are placed in order to divide the scene into regions with a similar number of primitive counts per axis where:

$$ncells_i = S_i \times \sqrt[3]{\frac{4 \times N}{V}} \quad i \in x, y, z \quad (4)$$

Algorithm 1 : Rectilinear grid traversal

```

function NEAREST-AXIS(int)
  if intx < inty then
    if intx < intz then
      return x
    else
      return z
    end if
  if inty < intz then
    return y
  else
    return z
  end if
end if
end function

function TRAVERSE(ori, dir, planes, cells, ncells)
  test if the ray hits the bounding box of the scene
  find the nearest ray/box intersection point p
  id ← FIND-FIRST-CELL(p, planes)
  for all i ∈ x, y, z do
    if diri < 0 then
      stepi, incri, stopi ← 0, -1, -1
    else
      stepi, incri, stopi ← 1, +1, ncellsi
    end if
    inti ← (planesi[idi + stepi] - orii) / diri
  end for
  t ← +∞
  loop
    if cells[idx, idy, idz] ≠ ∅ then
      traverse cell
      find the nearest ray/object intersection t
    end if
    repeat ▷ skip empty cells
      i ← NEAREST-AXIS(int)
      if inti ≥ t then
        return t
      end if
      idi ← idi + incri
      if idi = stopi then
        return t
      end if
      inti ← (planesi[idi + stepi] - orii) / diri
    until cells[idx, idy, idz] ≠ ∅
  end loop
end function

```

This rectilinear grid has a similar number of cells compared to a regular grid because we use the same heuristic to compute the number of split points. Contrary to a regular grid, the split planes are not equidistant; but divide regions with similar amounts of geometry. Non empty cells in a rectilinear grid will thus contain a smaller amount of primitives on average than a regular grid of the same dimensions. This will be explored in greater detail in Section 4. This construction method has a computation time complexity of $O(N)$. Space complexity is also $O(N)$.

Traversal

We employ a generalization of the voxel traversal method described by Cleary and Wyvill [CW88].

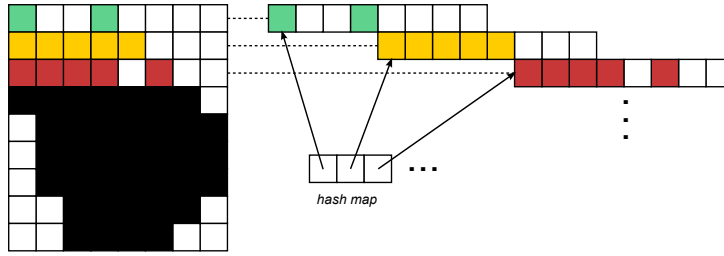


Figure 2. Row displacement compression

This method is described in Algorithm 1. Since we have variable positions for the split planes, it is not so worthwhile to precompute many of the traversal variables. If we cached these values we would spend many computing cycles performing slow memory loads just to save a couple of fast arithmetic operations. Profiling results showed that one of the main bottlenecks of this rectilinear grid traversal method consists in loading the split plane data from cache. Another large bottleneck is the branch mispredictions which occur while searching the next non empty cell. This rectilinear grid traversal method spends more time doing cell traversals than a regular grid traversal method because of these loads and the extra arithmetic operations. However, since each cell contains less primitives, less time is spent performing ray/primitive intersections. The end result is a net increase in rendering performance for many scenes.

Finding the First Cell

We have the split plane data for each axis in the rectilinear grid. The most expedient way to im-

plement an algorithm to find a point in this grid, without by using any extra memory, is by doing a binary search in the ordered list of the split points. This approach has $O(\log N)$ time complexity. This is clearly worse performing than the method employed for finding the first point in a regular grid which has $O(1)$ time complexity.

We speeded up this part of the algorithm by employing a lookup table which maps the quantized fine regular grid coordinates used in the sampling step during construction to the actual rectilinear grid coordinates. This lookup table, if uncompressed, would not easily fit in the caches compromising the rendering performance of the algorithm.

To fit the lookup table into the processor cache we compressed the data using arithmetic encoding (see Algorithm 2). The unpacking function, which loads a bit array from memory into an integer register, can be further optimized using assembly instructions. The prediction function we employed assumes most split planes will be regularly separated at constant intervals. If the distance between all the split planes is the same, which is common for scanned scenes, we have a perfect prediction. This means for such scenes no additional memory would be required to store the table. This prediction function also behaves well for other more irregular scenes. This can be seen in the results for the Fairy Forest scene. Time complexity with the lookup table is $O(1)$.

Algorithm 2 : Finding the first cell in a rectilinear grid in $O(1)$ time with a stored compressed bit array

```

function UNPACK(axis, i)
    diff ← 0
    for all j such that  $0 \leq j < msb[axis]$  do
        diff ← diff × 2 + storage[axis][i × msb[axis] + j]
    end for
    return diff
end function

function LOOKUP(axis, i)
    predict ←  $i \times n_{cells}_{axis} / n_{samples}_{axis}$ 
    diff ← UNPACK(axis, i)
    return  $min_{axis} + predict + diff$ 
end function

function FIND-FIRST-CELL(p, planes)
    for all axis ∈ x, y, z do
        i ← CLAMP( $\frac{p_{axis}}{sample_{size}_{axis}}$ , 0,  $n_{samples}_{axis} - 1$ )
        idaxis ← LOOKUP(i)
    end for
    return id
end function

```

Cell Compression

We minimized the memory required to store the grid by employing the row displacement compression algorithm [LD08] described by Lagae and Dutré. This algorithm works by compressing empty grid cells via row hashing (see Figure 2). Frequently grid acceleration structures feature 90% or more empty cells. This compression scheme reduces the memory required to store a grid up to 20 : 1.

The main issue with this scheme is that it makes it harder to perform partial grid updates. How-

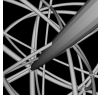
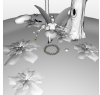


				
	AEK-24-cell	Fairy Forest	Buddha	Thai Statue
scene				
num triangles	122.88 K	174.11 K	1.09 M	10.00 M
memory	2.14 MB	4.22 MB	18.67 MB	171.66 MB
regular grid				
grid res	79x78x79	140x36x140	121x295x121	302x508x261
% empty cells	93.58%	79.12%	94.86%	98.44%
avg objects / n-empt cell	14.06	6.70	13.02	29.25
avg cells / object	3.58	5.67	2.66	1.83
mem cells	249.39 KB	800.85 KB	1.88 MB	8.97 MB
mem object lists	1.68 MB	3.76 MB	11.03 MB	69.78 MB
memory	1.92 MB	4.55 MB	12.91 MB	78.75 MB
build time	0.03 s	0.06 s	0.29 s	2.38 s
render time	2.54 s	4.95 s	0.70 s	1.63 s
rectilinear grid				
sample res	7944,7829,7903	13968,3570,13968	12130,29537,12144	30187,50824,26072
grid res	79x78x79	139x36x139	121x294x121	300x506x259
% empty cells	91.02%	78.40%	92.82%	96.52%
avg objects / n-empt cell	1.13	1.59	0.77	0.61
avg cells / object	4.49	6.37	3.06	2.41
mem cells	319.09 KB	929.48 KB	2.42 MB	13.64 MB
mem object lists	2.11 MB	4.23 MB	12.69 MB	92.08 MB
mem planes	960 B	1.24 KB	2.11 KB	4.18 KB
memory	2.42 MB	5.14 MB	15.12 MB	105.72 MB
build time	0.09 s	0.14 s	0.85 s	7.35 s
render time	2.23 s	2.76 s	0.71 s	1.38 s

Table 1. Performance results for several scenes. All scenes were rendered at 1024×1024 resolution with one ray sample per pixel. Only one thread was employed.

ever it is quick to rebuild such a grid, most of the steps performed in the construction algorithm can be parallelized, interactive frame rates can be achieved for many scenes.

4. RESULTS

A prototype implementation was written in order to test the viability of this algorithm for ray tracing complex scenes. The implementation was coded in ANSI C++ with use of the Boost libraries. This implementation does not use intrinsics or assembly instructions. The implementation was run on an Intel Core 2 CPU at 3.0 GHz with 2 GB of RAM under the Linux operating system.

All test scenes were rendered at 1024×1024 resolution with one ray sample per pixel and diffuse shading. Only one rendering thread was employed. Ray tracing performance scales linearly with the number of processor cores in the system.

Triangles are stored in memory using indexed vertex arrays. Ray/triangle intersection is done using

the Möller-Trumbore algorithm [MT05].

The performance comparison baseline is the compressed regular grid algorithm described in [LD08]. Our rectilinear grid algorithm uses a similar grid resolution as can be seen at Table 1. Both of these algorithms were implemented by us on our ray tracing system.

We selected four scenes for testing purposes: AEK-24-cell, Fairy Forest, Buddha, Thai Statue. These scenes are representative for many kinds of applications. AEK-24-cell contains a scene with data similar to that used for scientific visualization, Fairy Forest is an irregularly distributed scene similar to what we could find in a game application, the Buddha and Thai Statue are scanned scenes with heavy geometry.

Several things can be noted by examining the test results. As expected our rectilinear grid implementation has much improved render time performance (79% faster) for the irregular Fairy Forest scene. It also provides a performance speedup

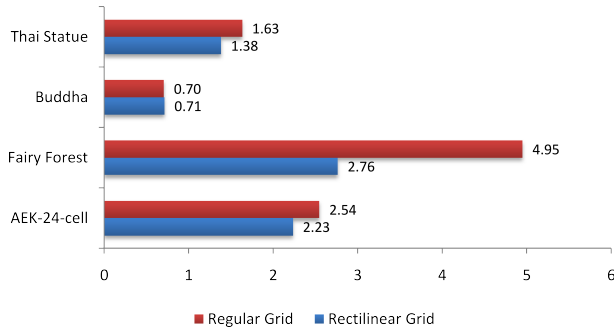


Figure 3. Render time in seconds. Lower values are better.

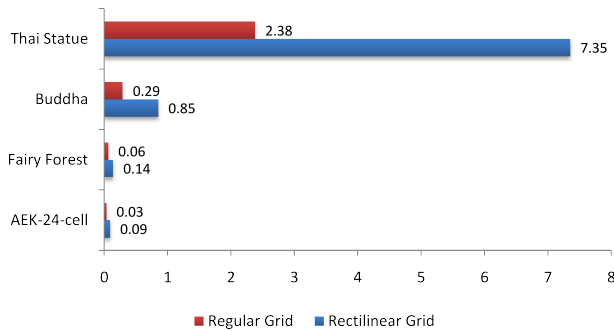


Figure 4. Build time in seconds. Lower values are better.

for the AEK-24-cell (14%) and Thai Statue (18%) scenes. In contrast the rectilinear grid has slightly worse rendering performance (-1%) for the Buddha scene.

We decided to examine these results in more depth. The rectilinear grids contain less triangles in each cell than a regular grid would. This was expectable due to the way we compute the split plane positions. However each triangle in the rectilinear grid overlaps more cells. This may make it worthwhile to employ mailboxing in our rendering system. Our implementation does not have this feature.

The split plane data easily fits into the L1 cache minimizing the amount of memory fetches required. Both grid implementations feature a 3D bitmap. Each bit in the bitmap states if a grid cell is empty or not. This allows us to reduce the amount of memory bandwidth required to traverse empty cells.

The build times for the rectilinear grid (see Figure 4) are 2-3x slower. This is mostly due to the sampling step during preprocessing. The sampling is done at a 100x higher resolution, per axis, than that of the grid itself.

Increasing the sampling resolution further does not improve the rendering performance for the

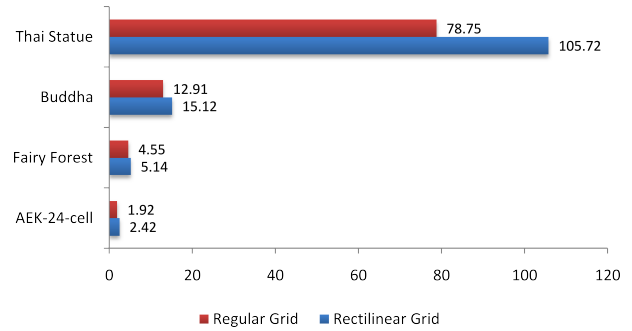


Figure 5. Memory consumption in megabytes. Lower values are better.

tested scenes. It would be interesting to further investigate different heuristics for selecting the amount of samples. In particular it would be possible to have quite different heuristics for selecting the sample size, and the grid size, rather than the simple multiple we use now.

Memory consumption (see Figure 5) is also higher for the rectilinear grid. This is due to the greater number of occupied cells, and large cell triangle lists.

Figures 6, 7, 8, 9 display the number of traversal steps required to render a given test scene. Pixels with a lighter tone of blue have more cell traversals. Pixels with a lighter tone of red have more triangle intersections. In this way it is easier to visualize the pros and cons of each acceleration structure.

5. CONCLUSIONS

We have proposed and implemented a rectilinear grid ray tracing algorithm. This algorithm has up to 79% better performance compared to a regular grid on the tested scenes using a similar grid resolution. It is especially well suited to render irregular scenes, which have typically been an issue with grid ray tracing.

Now that we can individually control the split plane positions, it should be easier to devise more sophisticated surface area grid heuristics based on previous work on BVHs and kd-trees.

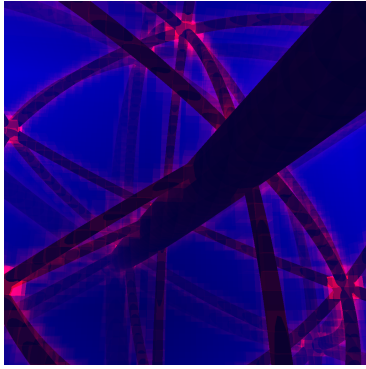
It should also be interesting to apply the scene sampling information generated during the construction stage to further construct a hierarchical data structure to speed up empty cell traversal. There are many other techniques to speed up empty cell traversal such as macro-regions [Dev89] and proximity clouds [CS94] which could prove useful for this purpose.

6. ACKNOWLEDGEMENTS

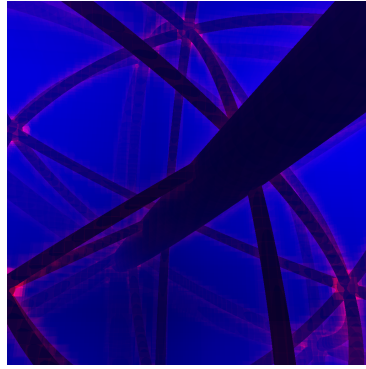
We would like to thank the Utah 3D Animation Repository for the AEK-24-cell and Fairy Forest scenes, the Stanford 3D Scanning Repository for the Buddha and Thai Statue scenes.

7. REFERENCES

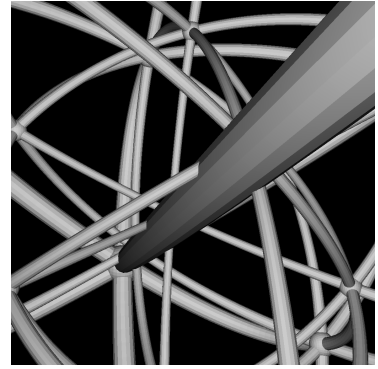
- [BSP06] J. Bigler, A. Stephens, and S.G. Parker. Design for parallel interactive ray tracing systems. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 187–195, 2006.
- [CP97] F. Cazals and C. Puech. Bucket-like space partitioning data structures with applications to ray-tracing. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 11–20. ACM New York, NY, USA, 1997.
- [CPJ10] V. Costa, J. Pereira, and J. Jorge. Multi-Level Hashed Grids for Ray Tracing. In *WSCG'2010*, 2010.
- [CS94] D. Cohen and Z. Sheffer. Proximity clouds - an acceleration technique for 3d grid traversal. *The Visual Computer*, 11(1):27–38, 1994.
- [CW88] J.G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, 1988.
- [Dev89] O. Devillers. The macro-regions: an efficient space subdivision structure for ray tracing. In *Eurographics*, volume 89, pages 27–38, 1989.
- [FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *Computer Graphics and Applications, IEEE*, 6(4):16–26, 1986.
- [HS99] V. Havran and F. Sixta. Comparison of hierarchical grids. *Ray Tracing News*, 12(1):1–4, 1999.
- [ISP07] T. Ize, P. Shirley, and S. Parker. Grid creation strategies for efficient ray tracing. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 27–32, 2007.
- [JW89] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Graphics Interface '89*, pages 164–172, June 1989.
- [KBB07] B. Kuehl, K.J. Blom, and S. Beckhaus. Generation of Shadows in Scene Graph based VR. In *WSCG'2007*, 2007.
- [KS97] K. Klimaszewski and TW Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, 1997.
- [LD08] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(8), 2008.
- [MT05] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2005.
- [PPL+05] S. Parker, M. Parker, Y. Livnat, P.P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. In *ACM SIGGRAPH 2005 Courses*, page 15. ACM, 2005.
- [Wal07] I. Wald. On fast construction of sah-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 33–40, 2007.
- [WH06] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, 2006.
- [Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.
- [WMG+07] I. Wald, W.R. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S.G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. *Eurographics 2007 State of the Art Reports*, 2007.



(a) regular grid

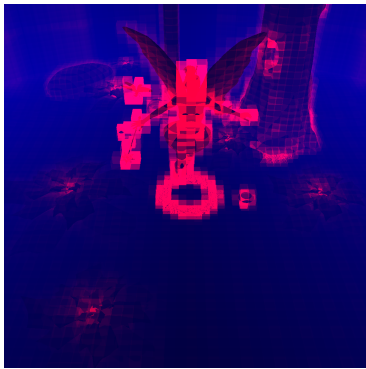


(b) rectilinear grid

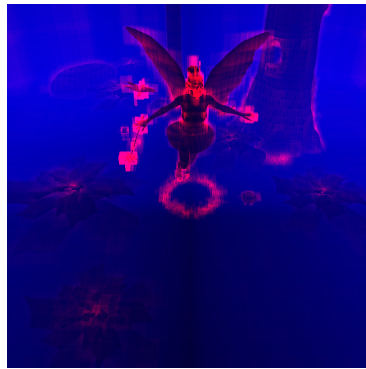


(c) rendered image

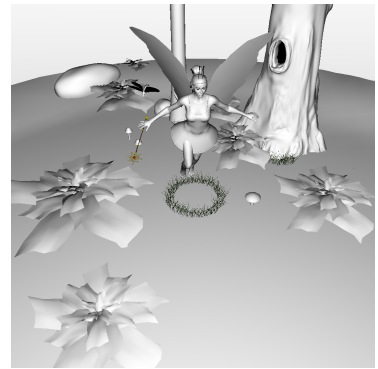
Figure 6. AEK-24-cell (122.88 ktri)



(a) regular grid

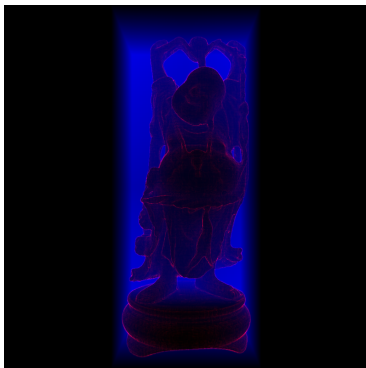


(b) rectilinear grid

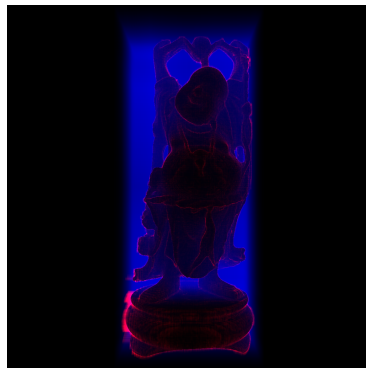


(c) rendered image

Figure 7. Fairy Forest (174.11 ktri)



(a) regular grid

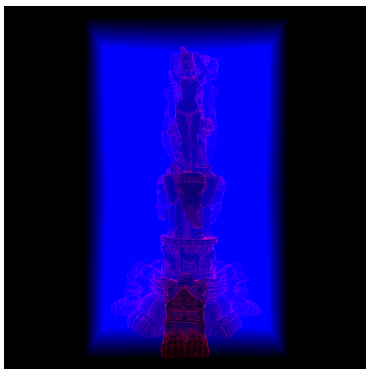


(b) rectilinear grid

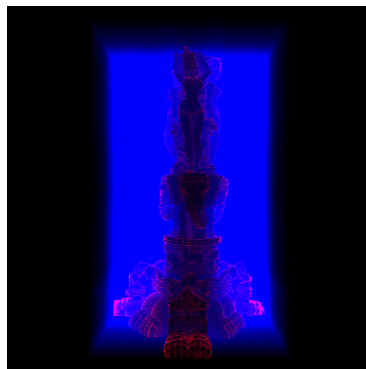


(c) rendered image

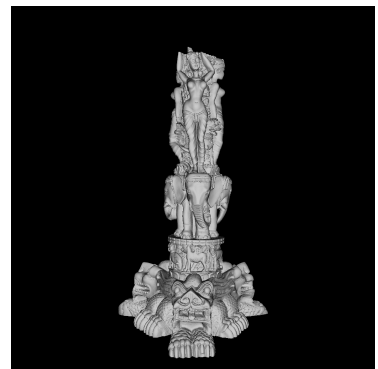
Figure 8. Buddha (1.09 Mtri)



(a) regular grid



(b) rectilinear grid



(c) rendered image

Figure 9. Thai Statue (10.00 Mtri)