Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 1201–1217
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor

Alphan Es[a], Veysi İşler[b,*]

[a]TÜBİTAK UZAY, METU, Turkey
[b]Department of Computer Engineering, METU, Turkey

Received 17 November 2005; received in revised form 28 February 2007; accepted 22 June 2007
Available online 19 July 2007

Abstract

Modern graphics processing units (GPUs) are an implementation of parallel stream processors. In recent years, there have been a few studies on mapping ray tracing to the GPU. Since graphics processors are not designed to process complex data structures, it is crucial to explore data structures and algorithms for efficient stream processing. In particular ray traversal is one of the major bottlenecks in ray tracing and direct volume rendering methods. In this work we focus on the efficient regular grid based ray traversals on GPU. A new empty space skipping traversal method is introduced. Our method extends the anisotropic chessboard distance structure and employs a GPU friendly traversal algorithm with minimal dynamic branching. Additionally, several previous techniques have been redesigned and adapted to the stream processing model. We experimentally show that our traversal method is considerably faster and better suited to the parallel stream processing than the other grid based techniques.

Keywords: Parallel stream processing; Distance fields; Ray tracing; Parallel rendering; Ray traversal

1. Introduction

Ray tracing is a well known photo realistic image synthesis method [3,43]. It can accurately simulate reflections, refractions, shadows and other various light phenomena by tracing ray trajectories, thus can generate high quality images of virtual environments. Fast synthesis of realistic images has broad range of application areas including virtual and mixed reality, visual simulations, entertainment, etc. On the other hand, ray tracing is computationally very expensive, and thus considered as an off-line rendering method till recently. Lately, with the advent of more capable hardware, interactive ray tracing research gained more popularity. Moreover, ray tracing is expected to be a viable alternative to raster based graphics rendering [18]. Some of the recent works focused on accelerating ray tracing algorithmically [32,40], while some others are centered upon specialized ray tracing processors [33,34,44]. Utilizing graphics processors of the commodity graphic cards for ray tracing is another research area drawing increasing amount of attention. Originally, graphics processors are meant to rasterize and shade simple primitives such as triangles or lines. However, over the last decade graphics processors improved not only in terms of speed but also in terms of flexibility and programmability. Today's graphics processors are considered as an implementation of streaming parallel processors. Huge processing power and steep acceleration rate in speed led many researchers to develop graphics processing unit (GPU) specific solutions to known problems. Many graphics and non-graphics related problems were successfully mapped to the GPU programming model [13]. Among these, GPU based ray tracing acceleration is relatively new. Carr et al. [5], Purcell [30] and Purcell et al. [31] show how to use GPU for ray tracing computations. Karlsson et al. [20] have implemented a ray tracer that runs fully on GPU utilizing empty space skipping data structures, while Weiskopf et al. [42] have developed a GPU based non-linear ray tracer. Foley et al. [10] implemented kD tree acceleration structure on GPU for ray tracing. More recently bounding

* Corresponding author.
E-mail addresses: alphan.es@uzay.tubitak.gov.tr (A. Es), isler@ceng.metu.edu.tr (V. İşler).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.06.011

volume hierarchy (BVH) based methods were successfully used for ray tracing on GPU [6,38].

Ray traversal is one of the most time consuming parts in ray tracing [19] and direct volume rendering methods [21]. Many acceleration structures have been proposed in the past. Havran explains and compares many of these traversal techniques in detail [16]. Because GPUs are parallel stream processors they favor simple localized data access, exploiting instruction level parallelism and arithmetically intensive kernels for maximum efficiency [28,29]. GPU friendly data structures and algorithms should conform to the stream processing model for efficient processing. Three-dimensional regular grid based methods are ideal for hardware stream processing, as they can be represented and accessed efficiently using texturing facilities of the GPU. Consequently, in this work we have chosen regular 3D grid based methods because they can be represented naturally using 3D textures. Moreover traversal algorithms running on them are relatively simple and can be made fit into the GPU programming model with some modifications. Some of the fastest known grid based traversal algorithms use distance transformations to accelerate ray traversals [7,36,45]. A number of these techniques are developed for ray casting based direct volume rendering. Essentially, distance based methods utilize distance fields calculated during preprocessing stage. Distances to the nearest objects are stored in the distance field. Distances are calculated by using a metric such as Euclidian, city block or chessboard distance. Distance based algorithms accelerate traversals by skipping empty macro regions with the information encoded in distance fields. It is worth mentioning here that there are some works not relying on distance fields yet can skip macro regions [8].

In this work we have focused on accelerating ray traversals using regular grids and distance based techniques. The system completely runs on the GPU and uses GPU friendly data structures and algorithms.

There are three main contributions of this paper. The first contribution is the introduction of a new chessboard distance metric based traversal algorithm. We have extended Sramek et al.'s data structure [36] and devised a GPU based minimum branching traversal algorithm, which we call as extended anisotropic chessboard distance (EACD) traversal. The second contribution is the redesign of the previous grid based traversal algorithms. We show how the traversals can be mapped to GPU programming model efficiently. The methods presented also suit well to the streaming SIMD model of the modern CPUs. As the third contribution; this work is the first attempt to make efficient implementations and comparisons between the GPU specific versions of the regular grid based traversal techniques.

Three of the previously known traversal methods have been adapted to GPU. These methods include Amanatides and Woo's [2] digital differential analyzer (DDA) based ray traverser, Cohen and Sheffer's [7] proximity clouds (PC) and Sramek and Kaufman's [36] anisotropic chessboard distance (ACD) based ray traverser. We choose these techniques because firstly all of them work on grids. Secondly, they cover both empty space skipping and non-empty space skipping traversals. In DDA based traversal methods, grid is traversed in a face connected cell incremental fashion. That is one of the neighboring cells is chosen for the next traversal step. Distance field methods (PC, ACD) on the other hand, skip range of empty cells in big steps. This way, a more inclusive comparison between our method and the previous works is made possible.

In particular Amanatides and Woo's traversal algorithm [2] is a variant of 3D-DDA. It is simpler and requires fewer operations than Fujimoto et al.'s original 3D-DDA traverser [11]. It does not perform empty space skipping. On the other hand we show that it can be implemented on GPU very efficiently using SIMD operations and without data dependent branching inside the traversal loop.

Cohen and Sheffer's [7] proximity clouds utilize 3D distance fields, which can be represented with 3D textures. The traversal algorithm is not complex. The simplicity of data structure and the traversal algorithm make it a good candidate for stream processing. Non-synthetically generated scenes generally contain empty spaces between the objects. Proximity clouds can skip empty regions in big steps to accelerate traversals. In order not to miss any possible intersections, it switches to face connected cell incremental stepping mode, when a ray is in close proximity to an object. The switching requirement is one of the biggest drawbacks of this method when adapted to GPU since it implies data dependent branching. It is possible to use different distance metrics in proximity clouds such as chessboard, city-block or Euclidian. We have used city-block metric since the distance fields can be created quickly and it results in longer traversal steps. Refer to Cohen and Sheffer [7] for a discussion of using different distance metrics.

Sramek et al.'s [36] method, anisotropic chessboard distance traversal, is based on chessboard distance metric. Their algorithm is better than proximity clouds in some aspects, such that switching to face connected incremental cell stepping mode is not required and a ray can skip the whole empty region which it resides in. The original implementation can handle not only regular and Cartesian grids but also rectilinear grids. In our work, we have developed a different and GPU friendly traversal algorithm utilizing ACD grids.

The rest of the paper is organized as follows. In the following section, implementation of parallel stream processing on GPUs and the overview of our GPU based ray tracer is given. In the third section, you can find efficient GPU based redesign and implementation of the previous algorithms. Last part of the third section and the fourth section gives the details of our GPU based minimal branching chessboard distance traversal algorithm and the underlying data structures. The test results are presented and discussed in the fifth section. Finally, conclusion and future work is given in the last section.

2. GPU ray tracer

Before giving details of our ray tracer, it is useful to explain how parallel stream processing is mapped to the GPU rendering pipeline. Stream can be described as an ordered set of data. Streams are processed by functions called as kernels, generally by executing a series of instructions for each element in sequence. Kernels accept a number of input streams and generate

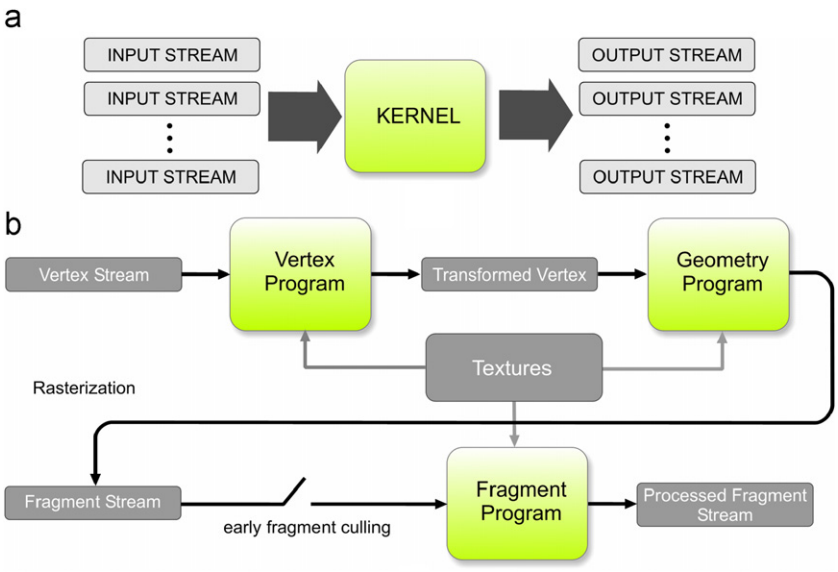

Fig. 1. (a) Stream processing structures (b) stream processing in the graphics pipeline. Vertex, fragment and geometry programs are the kernels run by the programmable processors on the GPU.

one or more output streams (Fig. 1a). A number of kernels can be chained to accomplish complex operations. Stream processing favors arithmetic intensity, high compute-to-bandwidth ratio, data locality, little global data access and parallelism [25]. Especially media and graphics applications are good candidates for efficient stream processing.

Fig. 1b depicts the simplified rendering pipeline of the modern graphics processors. In case of GPU, input stream elements may consist of vertex attributes (position, normal, color, etc.), texture elements (i.e. texels), fragments (pixel candidates) and the output is the stream of fragments. Most modern GPUs have two programmable sections along the pipeline, called vertex and fragment processors. As of today, the latest generation of GPUs has also geometry processors located after the vertex processors in the pipeline. These processors execute vertex, geometry and fragment programs (kernels). Vertex programs typically transform vertex information and send the processed vertices thru the pipeline, while geometry processors operate on geometric primitives such as lines or triangles. Along the pipeline the geometry is converted to fragment stream by rasterization. Majority of modern GPUs allow early fragment culling that is based on the outcome of stencil or depth tests. Utilizing early fragment culling is advantageous because it facilitates killing fragments before reaching the fragment processor and thus saves processing power. On the other hand there are some GPU specific rules to make early fragment culling work. Later on, fragment programs operate on the rasterized fragment stream and send the processed fragments to the raster operations units for final composition. All programmable processors within the pipeline support instruction level parallelism. Moreover there are numerous vertex, geometry and fragment processors running in parallel. The latest generation of GPUs such as NVidia's GeForce 8 series, have many unified processing units which are allocated for vertex, geometry or pixel processing on demand [12].

Note that programmability of vertex, geometry and fragment processors facilitates employing GPUs not only for graphics operations but also for non-graphics related problems. Therefore, GPUs are considered as general purpose parallel stream processors. A graphics API such as OpenGL or DirectX is required to program GPUs [23,35]. Since these APIs are tailored for graphics programming, they incur some performance penalty on implementations related to general purpose computation. Moreover, these APIs make general purpose GPU programming unintuitive and difficult. In order to alleviate such problems new technologies are being developed providing more direct access to GPU [1,26].

2.1. System overview

We implemented a Whitted style [43] ray tracer with full shadow, reflection and refraction effects in order to compare the traversal methods studied in this work. In Whitted style ray tracing, eye rays are generated and traced. Upon a surface hit, shadow rays for each light and reflection/refraction rays are generated depending on material properties of the surface. Basic constructs of our ray tracer are similar to Purcell et al.'s work [31]. Differently, we utilize depth buffer for early fragment culling and use a secondary grid as the acceleration structure. Data layout and buffer semantics are designed in such a way that memory I/O is minimized and more work is done per byte read or written. Where required, data packing is used to save bandwidth. The implementation of the system is done with OpenGL and Cg [22].

Triangle meshes are used for scene description. The scene database is stored in 2D textures. The database consists of vertex coordinates and normals, connectivity information and shader parameters. Two indices (2D texture space coordinates) are used for indexing triangles in the database.

Two-dimensional indexing makes sense and allows us to address data directly without index to texture coordinate conversions.

The grids are stored as 3D textures. We refer to the grid that keeps the scene partitioning information as the partition grid. Partition grid texture has 2-component 16-bit color format which points to a triangle list. If a voxel of the grid is empty (no triangles inside), it's value which is used for indexing to the triangle list is set to $(-1, -1)$. Triangle lists further point to triangle indices. Finally, triangle vertices are accessed using the triangle indices. Aside from the partition grid, another grid called as the acceleration grid is utilized for the traversals. Acceleration grids have lower component resolution (maximum 8 bits/color channel) to reduce the memory and bandwidth requirements.

An off-screen rendering context (PBuffer) with six render buffers and a depth buffer is used during execution of the kernels. Four-component 32-bit floating point color format is used for the render buffers. The system relies on early z-culling by means of depth bounds testing for computational masking. In the course of tracing, rays are given states and specific kernels operate only on the rays of a given specific state. Possible states for the rays are *creating*, *traversing*, *intersecting*, *intersected* or *shaded* and *out*. Ray state values are kept in the depth buffer for efficient masking by means of early depth testing (early fragment culling). In the GPU we used, it is required to create and modify depth buffer with `LESS`, or `GREATER` depth test function in order to benefit from early masking. Test direction should not be changed for the subsequent rendering passes. Otherwise, depth buffer optimization breaks down and early fragment culling does not work.

As depicted in Fig. 2, the system consists of five main kernels with a couple of smaller kernels for ray counting, intersection position/normal calculation, and depth mask modification. Additionally, there is a kernel to fire shadow rays which is very similar to the eye ray generator kernel. A single run of the traversal kernel followed by the intersection kernel is called as a *trace step*.

Prior to the rendering, ray states are initialized by setting depth buffer to 1 (*creating* state) and depth test function is set to `LESS`. Therefore, to be able to write new ray states, monotonically decreasing values are assigned to the *traversing* and *intersecting* states in each trace step. Only the rays with the greatest state value are processed by the next kernel. For the trace step n, where $n \geqslant 0$, the state value of ray R is calculated as below

$$statevalue\ (n) = \begin{cases} 1 & \text{if } R \text{ is being created}, \\ 0.9 - 2\delta n & \text{if } R \text{ is traversing}, \\ 0.9 - 2\delta n - \delta & \text{if } R \text{ is intersecting}, \\ 0.1 & \text{if } R \text{ is intersected or shaded}, \\ 0 & \text{if } R \text{ is out}. \end{cases}$$

For δ, we use 0.0001 which is sufficiently small to generate enough unique decreasing values for the *traversing* and *intersecting* states. After a main kernel, another kernel is run to write the new state values to the depth buffer, according to the output of the previous kernel.

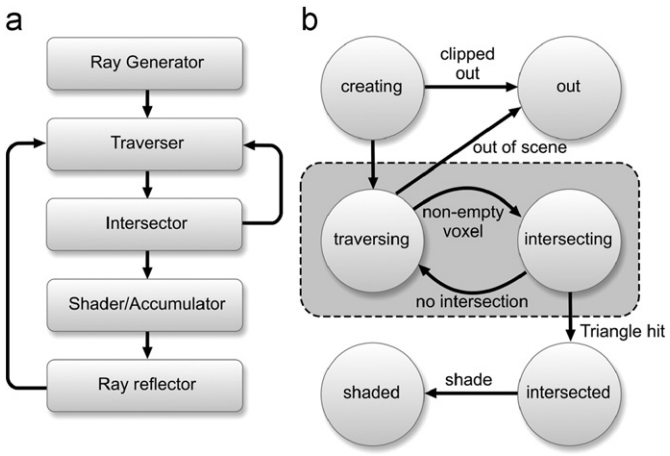

Fig. 2. (a) Flow of main kernels and (b) ray state transition diagram. Ray states are written to depth buffer between the main kernels for computation masking. Dashed region in (b) constitutes the trace step.

Fig. 3 shows detailed input/output streams of the main kernels. As seen in the figure, ray generator kernel creates eye ray origins and directions as two output streams. Eye rays are clipped against the bounding box of the scene. Since our traverser kernels require voxel indices for some computations, rays hitting the bounding box are transformed from the world space to the grid space (with unit voxel dimensions). This way, the integer part of the coordinates directly gives the current voxel index. Transformed rays are stored in a different buffer. Grid space rays are used by the traverser kernels. On the other hand, the intersector kernel uses the world space rays since the scene database is stored in world coordinates. After eye rays are generated, depth is set to the *out* state for out-of-scene rays and to the *traversing* state for remaining rays.

The second kernel, which is the main focus of this work, is the traverser. Traversal algorithms are implemented by the traverser kernels. Dynamic loop statements are used inside the intersector and traversal kernels. Traversal loops are repeated until a non-empty voxel is found or the ray gets out of the scene. There are two variants of traverser kernels. One of them is executed in the first trace step, while the other one is called in the subsequent trace steps. The only difference between the two is that the first one begins with checking if the ray starts in an empty voxel and loops until a non-empty voxel is found. The other kernel steps a single voxel first and then loops until a non-empty voxel is found. The output of the traverser kernel is used by the intersector kernel or again by the traverser kernel in the next trace step. Output values are slightly different for each traversal method. As is common to all kernels, the index of the current voxel and parametric distance value to the voxel boundary (used during intersection tests) is written to the output stream. When the traverser kernel is done, depth buffer is modified so that the rays in non-empty voxels are set to the *intersecting* state, while the remaining rays are set to the *out* state. Then the rays in the *intersecting* state are counted. If the count is zero the shader kernel is called, otherwise the execution continues with the intersector kernel.

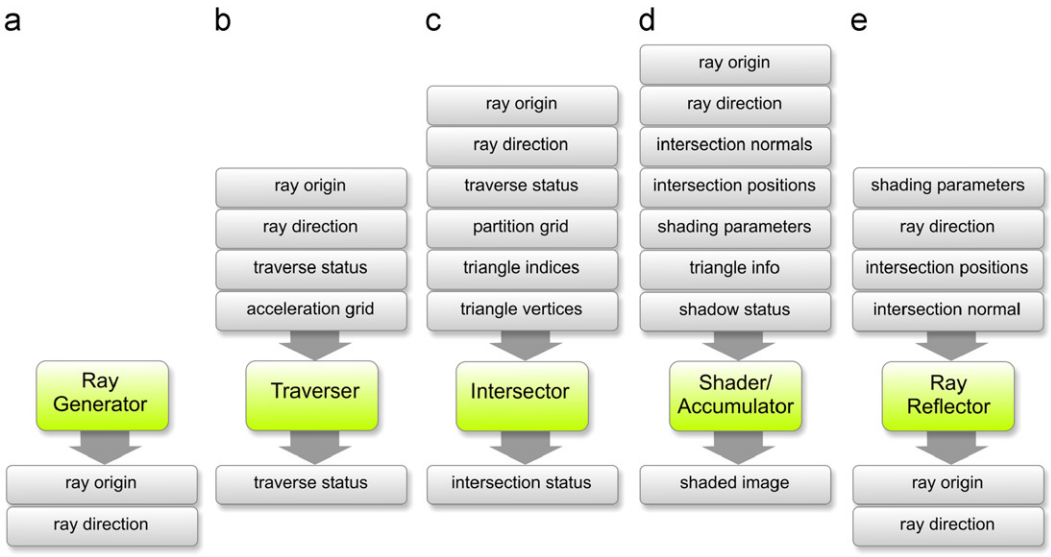

Fig. 3. Input/output streams of the main kernels: (a) ray generator, (b) traverser, (c) intersector, (d) shader/accumulator and (e) ray reflector.

The third kernel is the intersector kernel, which is employed to perform ray-triangle intersection tests for the rays in *intersecting* state. The intersector reads traverser outputs and accesses the voxel triangles from the scene database. Möller and Trumbore's ray-triangle intersection algorithm is used for intersection testing [24]. The output of the kernel is the barycentric coordinates of the intersection point if there is a hit. Otherwise, no output is created. The depth value is set to *intersected* state for intersected rays, or *traversing* state for the other rays. Note that the state value of the *traversing* state is calculated by incrementing the trace step count by one, so that the *traverser* kernel in the next step can process them. The execution continues with the *traverser* kernel if there are still some non-intersected rays.

The fourth main kernel is the shader/accumulator kernel, which performs shading calculations for the intersected rays. Shader kernel is executed once for each light source and the resulting color values are combined with the intermediate shading results of previous passes. Prior to the shader kernel, intersection positions and normals are calculated by using the results of the intersector kernel. A simple Phong shader is used for the rendering, although it is possible to implement more complex shaders. If shadowing is enabled, some extra steps are necessary before executing the shader kernel in order to determine shadowed pixels. These extra steps constitute what we call as the shadow pass. In shadow pass, secondary rays are fired (shadow rays) from the intersection positions towards the light position. The shadow rays are traversed to test if there is an intersection along the path. If an intersection is found, the point in consideration is not visible to the light and thus marked as shadowed (tagged as *intersected*). Shadow pass is nearly identical to casting of eye rays. Almost the same traversal and intersector kernels with minimal modifications aimed for performance optimizations are used for shadow pass. Ultimately, before executing the shader kernel all buffers keeping information about the rays and intersection results are saved to offline buffers. Afterwards shadow pass is realized for each light source. At the end of the shadow pass, the pixels tagged as *intersected* state are said to be in shadow with respect to the current light source. Shader kernel gets the shadow results along with other parameters and illuminates pixels accordingly.

The kernels described up to this point are enough to realize a basic ray caster with shadowing. A basic ray caster considers eye rays only. On the other hand, it is possible to extend the system to perform full ray tracing. In full ray tracing, new rays for reflection and refraction are generated at the intersection points and traced recursively. However, GPU hardware does not support recursion. In order to facilitate ray tracing, recursion is simulated by a buffer stack class. The buffer stack operates on a whole draw buffer instead of simple data types and has a typical stack interface. It stores the buffer contents in textures. Buffers keeping the current ray states and the intersection results are pushed into stacks before starting a new reflection or refraction pass. When the pass is finished, old buffers are popped from the stacks. Thus, the recursion is simulated in buffer level instead of ray (fragment) level. Ray reflector kernel is used to fire reflection or refraction rays. The kernel calculates the reflection or refraction directions and generates new rays from the current intersection points towards the calculated directions. Similar to the ray generator kernel, new rays are tagged as *traversing* state. As stated before, prior to the reflector kernel, current contents of the buffers are pushed into buffer stacks. Afterwards, reflection (or refraction) pass takes place. At this point, as shown in Fig. 2a, tracing is continued from the traverser kernel using the reflection (or refraction) rays instead of the eye rays. The recursion continues until a user defined recursion depth is reached.

3. Traverser kernels

During the implementation of the traversal algorithms we tried to improve instruction level parallelism, minimize required intermediate states, minimize data dependent branching

and provide balance between the memory and arithmetic operations. Firstly, efficient GPU implementations of the previous DDA and PC traversal algorithms will be presented. Next, GPU based minimal branching ACD and EACD traversal algorithms and related data structures will be introduced.

3.1. DDA traversal

DDA performs face connected cell incremental stepping and does not benefit from empty space skipping. The original algorithm requires two floating point comparisons and one floating point addition at each step. Although it is possible to port the algorithm directly to the GPU, this results in under utilization of GPU. Operations are scalar and data dependent flow control is used heavily to avoid floating point arithmetic. CPUs are fairly optimized for efficient flow control and branching, on the other hand they are relatively weak on intense floating point arithmetic compared to GPUs. We have redesigned the algorithm to use vector operations as much as possible. We also removed the data dependent flow control from the loop body.

In the original algorithm there are several comparison instructions to determine the step direction. Consequently, there are four main code paths in the loop body. Moreover in each path, ray is tested if it is inside the grid boundaries or not. In order to eliminate flow control, vectorized conditional set instructions are used. Because most GPUs are based on SIMD architecture, making the comparisons in a single vector instruction instead of several scalar instructions separated with different control paths will result in more efficient operation. Moreover, the ray-box containment tests are postponed and unified with the loop control statement. To achieve this, border color of the acceleration grid texture is set to a specific *out* value and texture wrap mode is set to CLAMP_TO_BORDER. Hence, if an out-of-grid voxel is sampled, the sampler returns with the *out* value. Mono 8-bit color format is sufficient for the acceleration grid. We assign 1 for non-empty voxels and 0 for empty voxels. Border color (e.g., *out* value) is set to 0.5. As a result, box containment tests are eliminated and traversal loop continues until the voxel value is non-zero.

Parametric ray equations are used during the traversals. Throughout the context, vectors are shown in uppercase, while vector components and scalars are given in lowercase. A point on a ray R can be calculated as: $R(t) = O + Dt$, where O is the origin, D is the direction vector and $t \geqslant 0$ is the parameter of the ray. The Cg code of the traversal loop is given in Fig. 4. Some variables should be initialized prior to the loop in the setup phase. Among these, `voxelIdx` is the current voxel the ray is in. `cellStep` is a 3-vector denoting the increment to the next voxel in major axis directions, which is calculated as $sgn(D)$, where *sgn* is the signum function. `tStep` is the vector of signed parametric distances required to cross a whole voxel, which is calculated as $sgn(D)/D$. `invGridSize` is the reciprocal vector of the grid dimensions. It is used to convert integer voxel indices into [0,1] ranged texture coordinates. Finally, `texAccelGrid` is the acceleration grid texture. Note that $1/D$ may produce floating point specials ($\pm$infinity), which can cause NAN (not-a-number) results in the subsequent operations [25]. To prevent errors $1/D$ is clamped to $[-\Gamma, +\Gamma]$ range, where Γ is a sufficiently big number.

```
// while voxel is empty & inside the grid
while ( voxel == 0 )
{
  // find minimum parameter distance
  tmin = min(t.x, min(t.y,t.z));

  // determine the step direction
  incr.xyz = (t.xyz == tmin.xxx);

  // advance
  t.xyz += tStep*incr;
  voxelIdx.xyz += cellStep *incr;

  // read next voxel
  voxel = tex3D(texAccelGrid, voxelIdx*invGridSize).a;
}
```

Fig. 4. DDA traversal loop.

3.2. Proximity clouds traversal

In the original algorithm distance information is stored in the background (e.g., empty) voxels. Instead, we keep distance values in the acceleration grid. Acceleration grid has 2-component (luminance-alpha) 8-bit texture format, capable of representing the maximum distance value of 255. Distances are measured using city block metric. Luminance component is set to the distance value, while alpha keeps the voxel status (empty or non-empty). Similarly to the DDA traversal, alpha component is set to 0 for empty and 1 for non-empty voxels, while alpha of the border color is set to 0.5. Because distances are calculated from the voxel centers, rays may miss non-empty voxels if they advance as much as the distance value. Distance values are subtracted by 1 and stored as subtracted to prevent this situation.

PC traversal requires switching between DDA and space skipping. In the original algorithm traversal in skip mode continues in a loop until a ray is close to a non-empty voxel. Then face connected traversal continues until the ray is no longer in vicinity of a non-empty voxel. In this case two inner loops within a bigger loop are required. We found that replacing inner loops with an *if* statement is more efficient as it requires less flow control. In each step the distance value is checked: If it is 0, DDA steps are taken otherwise PC stepping is performed. You can see our GPU implementation of PC traversal in Fig. 5. In the figure, `invD` is reciprocal of ray direction D. `invGridSize`, `cellStep` and `tStep` are the same as in DDA. `tStart` is the initial parametric distance to voxel borders from the origin. Calculation of `tStart` is explained in the next section. Lastly `C` is the distance function constant as stated in Cohen and Sheffer [7]. Note that since the texture sampler normalizes the color values into [0, 1] range, it is required to scale and round samples to recover the integer values, which unfortunately results in a performance hit.

3.3. Anisotropic chessboard distance traversal

Sramek and Kaufmann [36] use chessboard metric for the distance computations. Their chessboard distance traversal method has a couple of advantages over the proximity clouds:

```
// while voxel is empty & inside the grid
while ( voxel.a == 0 )
{
  // If close to non-empty voxels, then DDA stepping
  if ( voxel.r == 0 )
  {
    // same as DDA
    tmin = min(t.x, min(t.y,t.z));
    incr.xyz = (t.xyz == tmin.xxx);
    t.xyz += tStep*incr;
    voxelIdx.xyz += cellStep*incr;

    // update ray position
    pos.xyz = O+t*D;
  }
  else { // otherwise PC stepping

    // convert distance to integer
    dist = round( voxel.r * 255);

    // advance
    pos.xyz += C*dist;
    voxelIdx.xyz = floor(pos);

    // update ray parameter
    t.xyz = tStart + cell*invD;
  }

  // read next voxel
  voxel = tex3D( texAccelGrid, voxelIdx*invGridSize );
}
```

Fig. 5. PC traversal loop.

Firstly, switching between the stepping modes is not necessary. Secondly, empty space in a region can be skipped to the full extent, thus it is not necessary to subtract distance values by one. Their algorithm can also work on rectilinear grids with some additional costs. They observed that in distance based traversals, ray steps get shorter as the ray gets closer to the objects, and many small steps are taken until the ray gets far away from the close vicinity. To alleviate this problem, they propose using anisotropic empty regions depending on the ray directions. Rays are classified by the component sign of their directions ($\pm x$, $\pm y$, $\pm z$) giving 8 *direction octants*. Thus in ACD, instead of a single symmetric distance, 8 empty region distances are computed and the appropriate value to be used is determined by the component signs of the ray direction. Anisotropic distance calculation requires applying small masks over the grid in 8 passes (one pass for each direction octant). Note that although these 8 macro regions around a voxel form an anisotropic shape, each octant defines a cubic region individually in the grid space (assuming unit voxel dimensions).

The original algorithm divides a traversal step into slave steps and a master step, and uses data dependent branching to choose appropriate steps to minimize arithmetic operations. Although this approach may be good for CPUs it decreases the intensity of arithmetic operations, increases flow controls and ultimately results in poor GPU utilization.

3.3.1. Minimum branching ACD traversal algorithm

In this part, we will introduce our GPU based traversal algorithm for anisotropic chessboard distance transformation grids. In essence, the traversal algorithm is similar to the DDA. For the sake of simplicity and conciseness, the algorithm will be explained in 2D without loss of generality. It is straightforward to extend the operations into 3D.

The acceleration grid G has dimensions $w \times h \in \mathbf{Z}^+$. A voxel $V \in G$ is identified by the indices i, j where $i, j \in \mathbf{Z}$ and $0 \leqslant i < w$, $0 \leqslant j < h$. V stores the chessboard distance to the nearest full voxel. So $V(i, j) = \Delta = (\delta_x, \delta_y)$. If $\delta_x = \delta_y$ distances along x and y directions are equal. The parametric equation of a ray R can be decomposed into x and y components as

$$r_x(t) = o_x + d_x t,$$

and

$$r_y(t) = o_y + d_y t.$$

We traverse macro regions in a face connected fashion. Therefore it is required to determine the set of lines (planes) for the region that the ray is possibly intersecting. Intersection lines depend on the ray direction as depicted in Fig. 6. The corner voxel inside the region which is adjacent to both intersection lines is called the *apex* voxel. Given a voxel $V_{i,j}$ and a direction D, the intersection line set L is defined as

$$L(i, j) = \{x = L_x(i), y = L_y(j)\}, \tag{1}$$

where

$$L_x(i) = i + sat(sgn(d_x)),$$

$$L_y(j) = j + sat(sgn(d_y)),$$

where *sat* (saturate) function clamps values to [0,1] range. Note that since *sat* can be applied as an instruction modifier on the GPU, it can be executed without performance penalty. If the ray is currently in voxel $V_{i,j}$ and the distance vector is Δ, the indices of the apex voxel i_a, j_a for the macro region are calculated as

$$(i_a, j_a) = sgn(D)(\Delta - (1, 1)) + (i, j). \tag{2}$$

Substituting Eq. (2) into Eq. (1), we get the intersection line equations for the region

$$x = L_x(i_a) = i + sgn(d_x)(\delta_x - 1) + sat(sgn(d_x)),$$

$$y = L_y(j_a) = j + sgn(d_y)(\delta_y - 1) + sat(sgn(d_y)).$$

The parametric distances to the intersection points on the intersection lines are

$$(t_x, t_y) = \frac{L(i, j) - O}{D},$$

$$t_x = \frac{i + sgn(d_x)(\delta_x - 1) + sat(sgn(d_x)) - o_x}{d_x}, \tag{3}$$

$$t_y = \frac{j + sgn(d_y)(\delta_y - 1) + sat(sgn(d_y)) - o_y}{d_y}.$$

The parametric distance $t_{\min}$ for ray's next position

$$t_{\min} = \min(t_x, t_y).$$

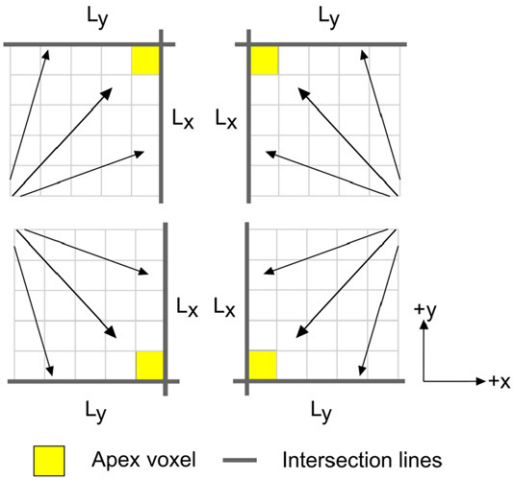

Fig. 6. Apex voxels and the intersection lines of a rectangular region depend on the ray direction.

```
// while voxel is empty & inside the grid
while( voxel.a == 0 )
{
  // compute signed distance
  dist.xyz = round(sgnScaled*voxel.r);

  //find apex voxel
  apexVoxel.xyz = (voxelIdx + dist);

  // compute tmin
  t.xyz = tStart + apexVoxel*invD;
  tmin = min(t.x, min(t.y,t.z));

  // advance
  pos.xyz = oEps+tmin*D;

  // find voxel indices
  voxelIdx.xyz = floor(pos);

  // read next voxel
  voxel = tex3D( texGrid, voxelIdx*invGridSize+offs);
}
```

Fig. 7. ACD traversal loop.

Then next position of the ray can be calculated as

$$P = R(t_{\min}) = \varepsilon + O + Dt_{\min},$$
$$\varepsilon = sgn(D) * 0.0001. \tag{4}$$

And the voxel indices for the new position are

$$(i, j) = (\lfloor p_x \rfloor, \lfloor p_y \rfloor).$$

Unfortunately, current GPUs have questionable arithmetic precision [17]. Therefore we found that adding ε to position is required to avoid floating point round-off problems. Otherwise ray may get stuck and never advance to the next voxel due to the precision errors. On the other hand using ε may result in skipping some non-empty voxels when a ray traverses very close to voxel corners. In order to ensure that no triangles are skipped in the intersection tests, we use slightly expanded ($\pm 10^{-6}$) voxel borders for triangle-box containment tests during scene partitioning.

Some of the operations in above equations can be taken out of the loop body and done in the traversal setup. In order to avoid decrementing δ_x, δ_y by 1 each time during the traversal, we store them pre-subtracted in the acceleration grid. The Eq. (3) can be rewritten as:

$$T = T_{\text{start}} + T_{\Delta},$$
$$T_{\text{start}} = \left(\frac{sat(sgn(d_x)) - o_x}{d_x}, \frac{sat(sgn(d_y)) - o_y}{d_y} \right),$$
$$T_{\Delta} = \left(\frac{i_a}{d_x}, \frac{j_a}{d_y} \right), \tag{5}$$

where the apex voxel indices (i_a, j_a) are (assuming distance values are pre-subtracted)

$$(i_a, j_a) = (i + sgn(d_x)(\delta_x), j + sgn(d_y)(\delta_y)) \tag{6}$$

T_{start} is computed once in the traversal setup, whereas T_{Δ} is computed inside the loop. The Cg code for the traversal loop is given in Fig. 7. Texture format for the acceleration grid is 2-component 8-bit (luminance-alpha). The semantics of the components are similar to the PC traversal. To be able to store 8 distance values per voxel, the size of the acceleration grid is twice as big as the partition grid. It can be considered as voxels are divided into 8 sub-voxels and each sub-voxel keeps the distance information of a specific direction octant. A factor and an offset is used to access to the appropriate sub-voxel. `invGridSize` is the mentioned factor and `offs` is the mentioned offset value. `offs` is calculated as (`invGridSize`/2) $*$ ($sat(-sgn(D))$. `dist` is the signed distance value and `apexVoxel` is the indices of the apex voxel as in Eq. (6). `sgnScaled` is calculated in traversal setup as $255 * sgn(D)$. `oEps` is pre-calculated as $\varepsilon + O$ of Eq. (4). T_{start} is denoted by `tStart`.

4. Extended anisotropic chessboard distance traversal

Even though the ACD traversal reduces the number of ray steps considerably, it is possible to improve the structure further for faster traversals. As expressed in the previous section ACD uses a single distance value for each direction octant. Instead we allow different distance values along x, y and z axes. That is three values are defined for each octant. As illustrated in Fig. 8 this facilitates non-cubic macro regions. As a result, rays can traverse with greater steps especially in long thin or narrow empty spaces or in close object vicinities. The traversal is almost the same as the explained ACD traversal algorithm. The only difference is instead of a single distance value for all axes, three distance values are used representing major axial distances. A 4-component texture format (red-green-blue-alpha) is used for the data, where red-green-blue components store distance values and alpha component keeps the voxel type information as before. EACD traversal code is given in Fig. 9.

The memory cost of the acceleration grid for EACD is three times as much as the ACD. In order to reduce this cost a packed

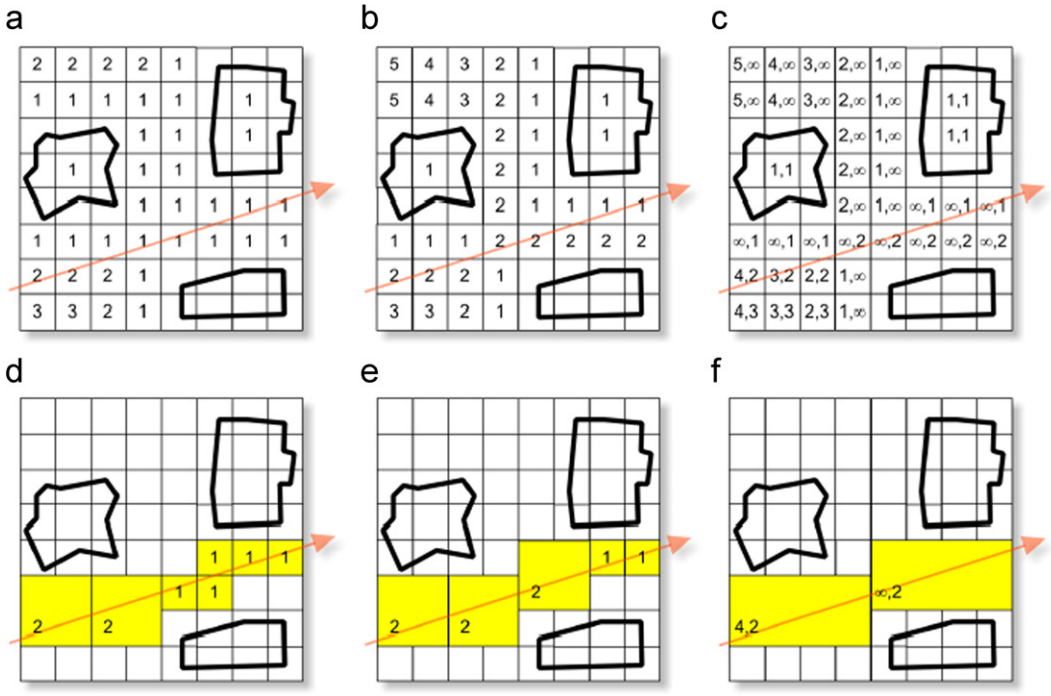

Fig. 8. (a) CD acceleration grid has single isotropic distance value per voxel (b) ACD acceleration grid stores a distance value for each direction quadrant. The distance field of the $(+x, +y)$ quadrant is shown here. (c) EACD acceleration grid stores different distance values for each primary axis. In the figure only the $(+x, +y)$ quadrant values are shown (first and second values are the macro distances along $+x$ and $+y$ axes, respectively). (d), (e) and (f) depict example traversals based on the shown acceleration grids for CD, ACD and EACD, respectively. EACD traversal significantly reduces the number of traversal steps in this situation.

```
// while voxel is empty & inside the grid
while( voxel.a == 0 )
{
  // compute signed distance
  dist.xyz = round(sgnScaled*voxel.rgb);

  //find apex voxel
  apexVoxel.xyz = (voxelIdx + dist);

  // compute tmin
  t.xyz = tStart + apexVoxel*invD;
  tmin = min(t.x, min(t.y,t.z));

  // advance
  pos.xyz = oEps+tmin*D;

  // find voxel indices
  voxelIdx.xyz = floor(pos);

  // read next voxel
  voxel = tex3D( texGrid, voxelIdx*invGridSize+offs);
}
```

Fig. 9. EACD traversal loop.

color format (RGB5A1) is used for the grid texture. This format can represent the distance range of up to 32 voxels along each direction.

4.1. Construction of the acceleration grid

Construction of EACD grid can be carried out in different ways. We use a heuristic with a simple greedy search. The heuristic strives to find the largest empty region by extending the ACD regions. Finding the largest non-cubic empty regions can be a very time consuming process. Therefore, instead of searching the largest empty spaces from scratch, we make use of ACD acceleration grid and extend regions along the main axes. Consequently, building the EACD grid involves two phases.

The first phase is exactly the same as creating ACD grid. The strategy to create distance transformation is based on the idea of propagating local distances over the grid cells [4]. Firstly the cell contents of the distance maps are initialized to infinity for empty voxels, and to zero for non-empty voxels. Then a mask is overlaid onto each cell of the map in a specific direction (such as beginning from bottom-left to top-right). Each element of the mask is summed with the value of the corresponding cell of the distance map. The resulting value of the cell is the minimum of these sums and the initial value of the cell. Generation of anisotropic chessboard distance maps involves applying eight different masks to the grid data beginning from one of the corners towards the opposite diagonal corner (Fig. 10). Consequently, eight masks are applied to the volumetric grid, and one (anisotropic) distance map is created for each direction octant. These eight maps are interleaved into a single big grid with eight times the size. The computational complexity of ACD transformation is $O(n)$, where n is the number of voxels.

Before the second phase we create six axial distance fields representing distances to the nearest full voxels along the $\pm x$, $\pm y$ and $\pm z$ axes directions. Similar to the ACD grid, the auxiliary grid distances are computed in linear time.

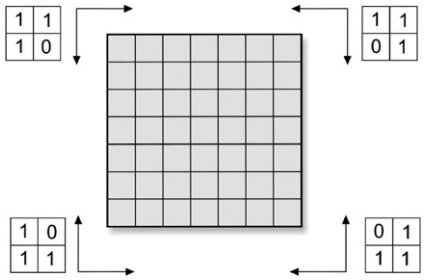

Fig. 10. ACD grid creation in 2D. Four masks are applied in the shown directions. Four (anisotropic) distance maps are generated as the result.

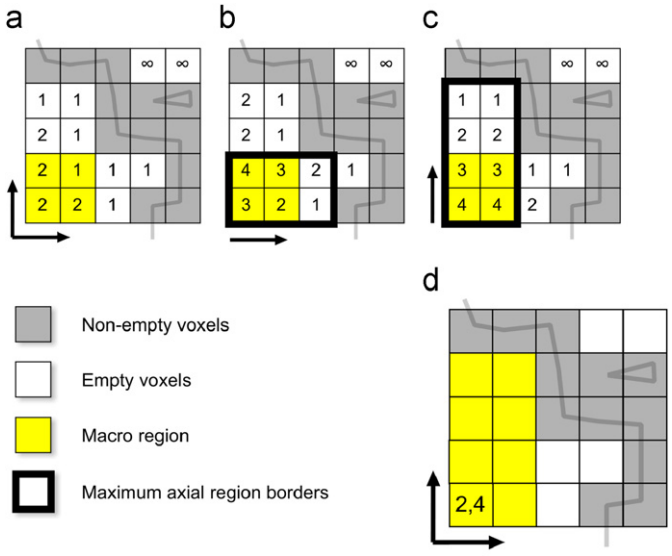

Fig. 11. Finding the EACD macro region for the lower left cell. Only $(+x, +y)$ direction quadrant is shown (octant for 3D). Arrows denote the orientation of the distance values (a) is the base ACD grid. (b) and (c) are the axial distance grids along $+x$ and $+y$ directions, respectively. (d) is the resulting EACD distances.

Using the auxiliary axial distance grids we determine, in a greedy manner, how much a cubic empty region can be extended. This operation is done for all empty voxels of ACD grid. The extension is carried out as follows: Border voxels of the empty region are walked and maximum possible extension along the main axes is computed. The region is then extended along the axis giving the maximum volume. This step is repeated once more, for one of the remaining two axes which results in a larger volume. Fig. 11 depicts computing EACD macro region for an empty grid cell (corresponds to an empty voxel) in 2D.

5. Results and discussion

In order to test the traversal algorithms, some of SPD and the 70K Stanford bunny models were used [15,37]. Ground plane in some SPD models was reduced in size to make better use of the grid space. The rendered images are shown in Fig. 15. Tests were performed on a 580 MHz Nvidia GeForce 7800GTX graphics board with 512 MB of memory. Release 93.71 graphics drivers and Cg toolkit 1.4 were used. The resolution is 512×512 for all of the images.

Table 1
Instruction counts of the traverser kernels

	EACD	ACD	PC	DDA
Setup/write	32/4	32/4	34/4	20/2
Loop	16/1	16/1	25/1	12/1

In the cells, first number is the fragment program instruction count (including texture lookups). Second number is the texture lookfup count.

In order to collect the rendering statistics, we implemented additional versions of the traverser kernels. The additional versions count the number of loops and the number of texture lookups performed. The collected values are written to a secondary color buffer (by using multiple render targets), without affecting the main rendering operation. After a fragment program is executed, its counter values are read from the buffer. This way, we are able to determine loop counts, the number of texture accesses and approximate bandwidth requirements of each kernel, per pass. Execution times, on the other hand, are taken by the original versions of the kernels.

Among the results, the traverser kernel performances are focused in particular since we propose a new traversal algorithm. In a ray tracer, the total rendering time also greatly depends on the intersection testing times. On the other hand, in direct volume rendering [21] there are no intersection tests and the main determining factor of rendering performance is the traversal part. The traversal methods explained here can be applied to volume rendering easily as shown in our previous work [9].

5.1. Branching vs. non-branching kernel implementation

As expressed in Section 2, data dependent branching is used in the traverser kernels. The other way of implementing the kernels might be to use multi-pass (non-branching) rendering to simulate data dependent loops. Both approaches have some advantages and disadvantages. Note that, multi-pass approach may be the only option if the GPU does not support dynamic branching. To compare the performances of branching and multi-pass approaches, we eliminated the dynamic loop instructions and built non-branching versions of the traverser kernels.

Traverser kernels include a setup phase and a write phase, in addition to the loop body. Setup phase consists of a number of texture lookups to reload last ray position and direction possibly with some data unpacking. Additionally, initial values of some variables should be computed in the setup and write phases. As shown in Table 1 the majority of kernels consist of setup/write phases. The number of instructions and texture fetches are given in the table. Note that the instruction count is not the only factor determining the performance. In a multi-pass implementation, setup/write phases consume more bandwidth and

computational power than the looping implementation since the kernels are called more times. This is especially true for DDA, which has to carry out possibly many small steps, and thus needs many rendering passes. Space skipping techniques suffer less from this condition, as they require less number of traversal passes. Another problem with the multi-pass implementation is that depending on the number of passes, the overhead of API calls (fragment program switching, state settings, etc.), modification of depth buffer and depth queries negatively impacts the overall performance. Especially for the latest and future generation GPUs, API overhead may be a bottleneck in a multi-pass implementation.

However, data dependent branching is not cheap and can easily be the bottleneck in GPUs due to deep pipelining and SIMD style parallelism. Fragment processing units of modern GPUs only support SIMD style branching: If some of the fragments in a group take a branch while remaining fragments take another branch, both branches are executed. Therefore all running fragments in the group should follow the same execution path for the highest performance. The group in this context is a rectangular block of fragments. As the block size in which all fragments follow the same execution path grows, the branching performance increases. There is an optimal block size for the best branching performance. The block size depends on the GPU model. Refer to GPU Bench [14] for dynamic branching benchmarks of different GPUs with respect to varying block sizes. Unfortunately it is hard to make all fragments of a block to follow the same execution path, since the neighboring rays gradually loose coherence and tend to choose different paths during rendering. One way to overcome this problem is to reduce the frequency of data dependent branching. Table 2 shows the average number of branching performed by scene rays. Clearly, empty space skipping greatly helps to reduce the branching frequency. The table can also be interpreted as how quickly rays completed the traversal. It is observed that EACD has lower branching frequency and thus requires less traversal steps compared to the other methods.

In non-branching approach, it is very costly to wait for all rays to reach non-empty voxels before the intersection test which results in large number of traversal passes. Instead, we employed a simple heuristic similar to Purcell et al. [31]; if 20% of the traversing rays require intersection, traversal is interrupted and the intersector kernel is run. This cuts down the number of traversal steps and the traversal times greatly, although increases the total intersection times. For best results, this heuristic should be fine tuned for each scene and even for different camera setups. DDA benefits largely from the fine-tuned multi-pass approach due to high branching frequency. EACD, on the other hand, issued slightly worse and sometimes slightly better rendering times. In order to evaluate non-branching kernels, ray casting traversal times for all test scenes are given in Table 3. These figures can be compared to the results of the branching kernels given in Table 4(c). Test results demonstrate that empty space skipping techniques are better than DDA both in multi-pass and branching approaches, and EACD gives the best performance compared to other space skipping techniques.

Table 2
Average data dependent branching per ray for the traversal kernels

		EACD	ACD	PC	DDA
Bunny	$32 \times 32 \times 32$	2.45	3.6	7.12	16.58
	$64 \times 64 \times 64$	2.98	4.32	7.9	33.39
	$128 \times 128 \times 128$	3.75	4.96	8.54	67.01
Tree	$32 \times 32 \times 32$	2.4	4.15	11.73	34.62
	$64 \times 64 \times 64$	2.47	4.47	12.64	69.41
	$128 \times 128 \times 128$	3.35	4.74	13.3	139.1
Jacks	$32 \times 32 \times 32$	3.91	4.77	12.36	21.99
	$64 \times 64 \times 64$	4.09	5.64	14.58	44.2
	$128 \times 128 \times 128$	4.62	6.19	14.57	88.63
Lattice	$32 \times 32 \times 32$	1.96	2.91	7.36	7.31
	$64 \times 64 \times 64$	3.38	4.91	13.01	15.06
	$128 \times 128 \times 128$	3.91	5.8	17.08	30.52
Sphereflake	$32 \times 32 \times 32$	2.87	4.41	11.08	21.21
	$64 \times 64 \times 64$	3.2	5.08	13.15	42.84
	$128 \times 128 \times 128$	3.56	5.51	14.69	86.12

Table 3
Time results (in milliseconds) of non-branching kernel implementations (grid size: $128 \times 128 \times 128$)

		EACD	ACD	PC	DDA
Bunny	Traverse	22,15	23,05	28,13	100,58
	Frame	71,08	73,43	80,04	152,69
Tree	Traverse	18,25	19,2	32,01	181,7
	Frame	202,59	209,09	292,68	413,85
Jacks	Traverse	17,89	20,77	32,56	132,02
	Frame	70,69	83,89	118,07	197,25
Lattice	Traverse	25,17	29,04	39,31	53,27
	Frame	94,51	106,42	154,79	198,37
Sphereflake	Traverse	18,32	20,12	29,09	88,69
	Frame	77,98	83,44	111,95	187,69

5.2. Fragment processor utilization

In order to benefit from the computational power of GPU, fragment processors should be utilized as much as possible. Crudely, utilization can be expressed as the ratio of the number of processed fragments over the number of rasterized fragments. Throughout the execution, the number of active rays to be processed decreases in each trace step, which means that the utilization drops in each subsequent step. Although there is a decline in the utilization, early rejection of the fragments before reaching the shader unit greatly helps to keep the fragment processors busy with useful fragments. Consequently the performance drop is not as sharp as expected for the most of the rendering. We check this situation by

Table 4
Ray casting time results in milliseconds

		EACD	ACD	PC	DDA
(a)					
Bunny	Traverse	8,43	11,79	18,38	17,86
	Frame	100,31	103,76	112,73	112,22
Tree	Traverse	9,02	11,84	23,43	26,01
	Frame	172,29	174,37	195,45	196,81
Jacks	Traverse	15,6	20,72	35,45	29,58
	Frame	162,85	167,79	189,93	184,2
Lattice	Traverse	14,93	19,56	24,36	17,37
	Frame	258,66	262,97	276,8	269,12
Sphereflake	Traverse	11,03	14,18	26,41	28,97
	Frame	208,24	210,58	231,43	244,65
(b)					
Bunny	Traverse	12,06	18,43	30,04	40,38
	Frame	62,7	70,73	84,42	93,79
Tree	Traverse	10,4	21,78	38,83	60,84
	Frame	182,35	186,38	215,52	233,99
Jacks	Traverse	18,49	28,97	59,02	67,1
	Frame	102,4	113,41	147,14	155,03
Lattice	Traverse	22,36	29,05	57,62	47,28
	Frame	126,36	136,62	171,01	159,12
Sphereflake	Traverse	15,13	20,85	43,24	52,79
	Frame	128,72	135	160,8	175,37
(c)					
Bunny	Traverse	21,64	36,73	62,38	114,25
	Frame	70,37	86,46	149	165,2
Tree	Traverse	14,96	19,89	46,8	130,38
	Frame	196,61	204,68	235,35	317,18
Jacks	Traverse	24,48	41,77	84,56	165,3
	Frame	76,45	94,2	139,22	235,69
Lattice	Traverse	29,7	40,08	106,56	126,65
	Frame	95,05	105,74	173,86	193,79
Sphereflake	Traverse	22,73	32,99	73,5	136
	Frame	86	96,52	138,82	207,43

Frame is the total rendering time. Grid sizes are (a) $32 \times 32 \times 32$ (b) $64 \times 64 \times 64$ and (c) $128 \times 128 \times 128$.

measuring the average traversal time per ray in each pass. Fig. 12 depicts the per-pass results for the *bunny* scene. In the graphs, as the number of active rays decreases in each step, the processing time decreases in a similar shape. Average time per ray is calculated for each step by dividing the total step time to the number of active rays in that step. The "time per ray" graph shows that for the majority of the rays ($> 98\%$) time spent is well below 100 nanoseconds. Especially, after around step 20 efficiency declines quickly. In fact, this is expected because the timings include not only the kernel processing time but also API and CPU overheads. Moreover, even if there is just a single active ray left, a screen sized quad is rasterized. Therefore, when the number of active rays is low enough, the mentioned overheads devastate the fragment processing time. However, only a small fraction of the rays suffer from this condition.

In an animation sequence, one possible way to reduce the overheads incurred in the later trace steps might be to cut rendering when the percentage of active rays is lower than a defined threshold. For example, in *bunny* scene, 99.5% of the rays are already terminated in step 16. Our other test scenes behave similarly. Fig. 13 shows the image rendered immediately after step 16. For this image, the partial render time is 65% of the full rendering time (using EACD). When the camera stops moving, the rest of the image can be rendered progressively. It is also possible to approximate the unfinished pixels of the partially rendered image by applying simple image reconstruction filters. On the other hand, this kind of rendering is reasonable for fine grid resolutions. In low resolutions, partial rendering saves less time and exhibits more artifacts. This is because of the fact that the number of trace steps is already low and the artifacts due to the unfinished parts become more apparent as the voxel sizes are bigger.

5.3. Timing results

The traversal algorithms were tested using several grid resolutions. In order to rule out external factors affecting the measurements as much as possible (such as file access, background processes etc.); each test was run many times and the minimum of the timing results is taken. Fig. 14 illustrates the number of traversal steps taken by each algorithm for the *bunny* scene. In the shown images, brighter pixels represent greater traversal step counts. As clearly seen in Fig. 14, DDA requires the greatest number of steps among all, since it steps only one voxel at a time regardless of the empty regions. PC on the other hand can skip isotropic macro regions in larger steps, while it behaves similarly to DDA in the vicinity of non-empty voxels. ACD performs better than PC in the close proximity to the non-empty voxels. This is because of the anisotropic macro regions and the fact that rays can take bigger steps according to their directions. EACD performs the best among all the other methods. The average traversal loop count (data dependent branching performed) per fragment is 34.6, 4.43, 2.56 and 1.94 for DDA, PC, ACD and EACD, respectively, for this particular scene. Thus, although DDA has the least expensive kernel it should loop many more times than space skipping techniques, resulting in much longer execution.

Table 4 shows that the ray casting speedup due to EACD is as much as 870%, 358% and 170% compared to DDA, PC and ACD, respectively. The speedup increases especially for the scenes where rays have to pass through non-cubic empty regions. Since both ACD and EACD use the same algorithm essentially, their performances should be similar in the worst case. On the other hand, maximum distance limit imposed by the low precision texture format may prevent full speedup possible with EACD. This is especially apparent in *tree* scene with $128 \times 128 \times 128$ grid dimensions, where there are large empty regions around the object. It is possible to use a higher precision texture format to alleviate this problem, if GPU has enough memory space. In *sphereflake*, model fills the grid space more fully and there are many narrow, non-cubic empty spaces where rays can pass thru. Therefore the EACD performance is better compared to *tree*. As an empty space skipping technique, PC traversal does not perform as fast as EACD and ACD. This is

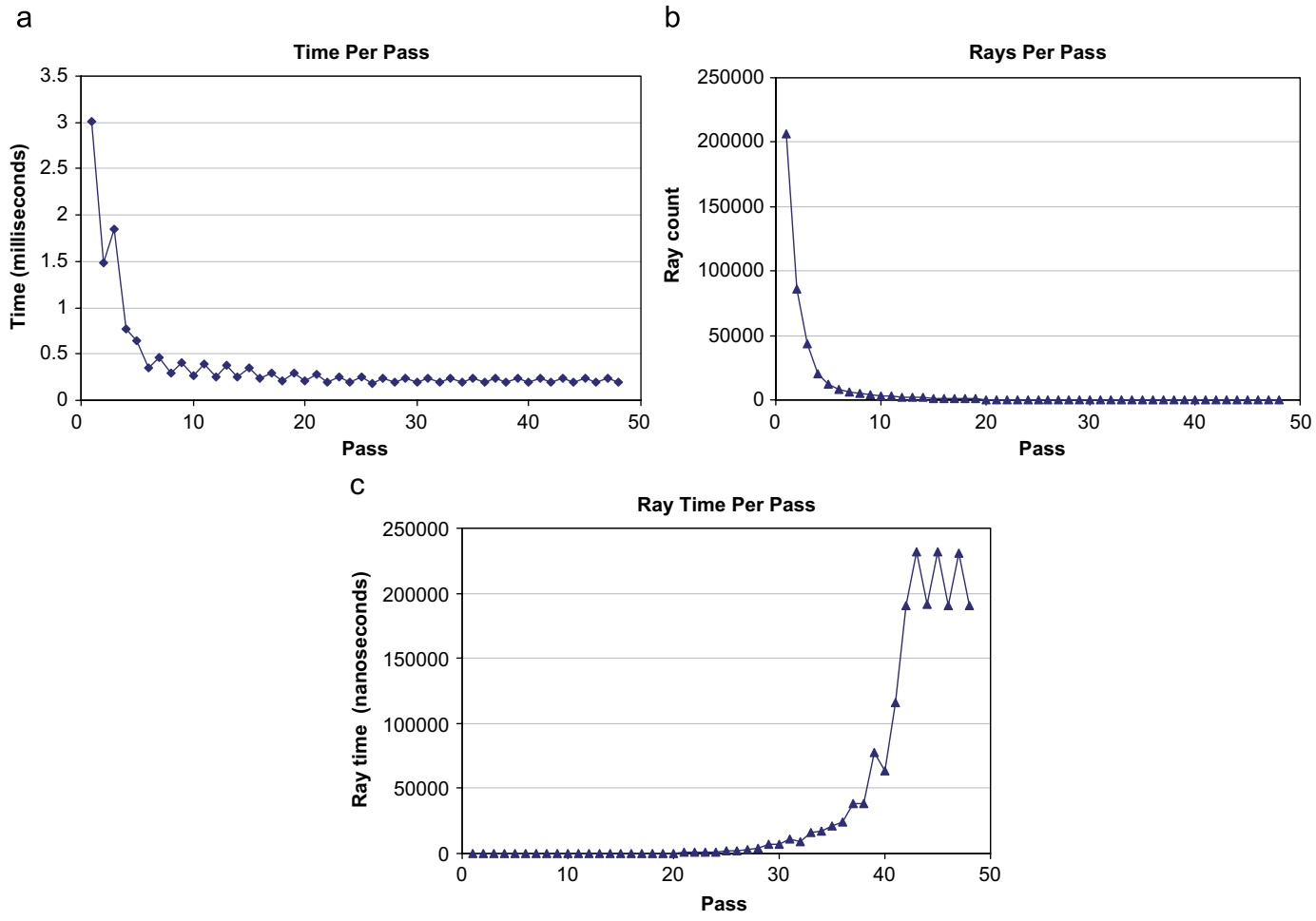

Fig. 12. (a) Number of active rays per pass, (b) traversal time per pass, (c) average ray traversal time per pass.

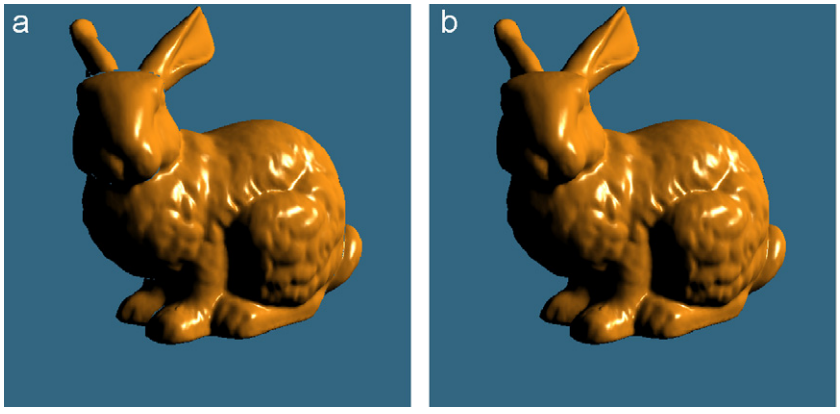

Fig. 13. Ray casted *bunny* image (a) after 16 trace steps (b) after all trace steps completed. Note that 99.5% of the rays are already terminated in (a) and most of the image is rendered. Grid resolution is 128 × 128 × 128.

largely due to the longer loop body with relatively higher data dependent branching frequency and shorter ray steps. Despite the fairly efficient loop body, DDA has almost always the worst performance. Especially in finer grid resolutions DDA cannot match the traversal speed of EACD and ACD.

We also compared ray tracing performances of the kernels with varying number of lights and trace depths. *Lattice* is used for the ray tracing tests since majority of the reflected rays stay inside the scene and keeps bouncing. Additionally, small surfaces inside the scene reflect rays to different directions lowering the ray coherence rapidly. Test results are given in Table 5. As seen in the table, reflected rays cause greater performance hit than shadow rays. This is because of the fact that reflected rays tend to loose coherence rapidly, while shadow

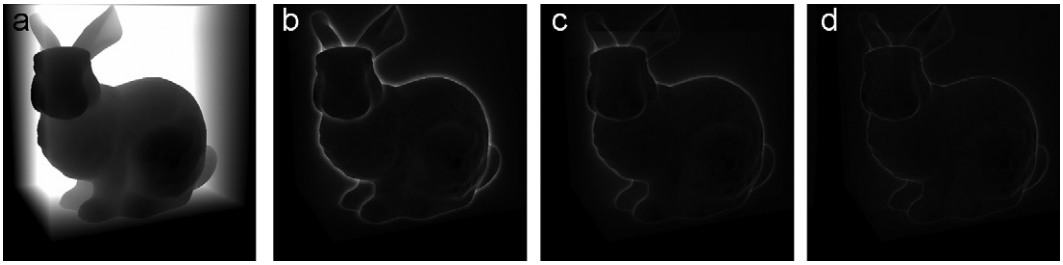

Fig. 14. Illustration of the traversal step counts for (a) DDA, (b) PC, (c) ACD and (d) EACD. Brightness of the image is set to 170% and contrast is set to 155% for better visual clarity.

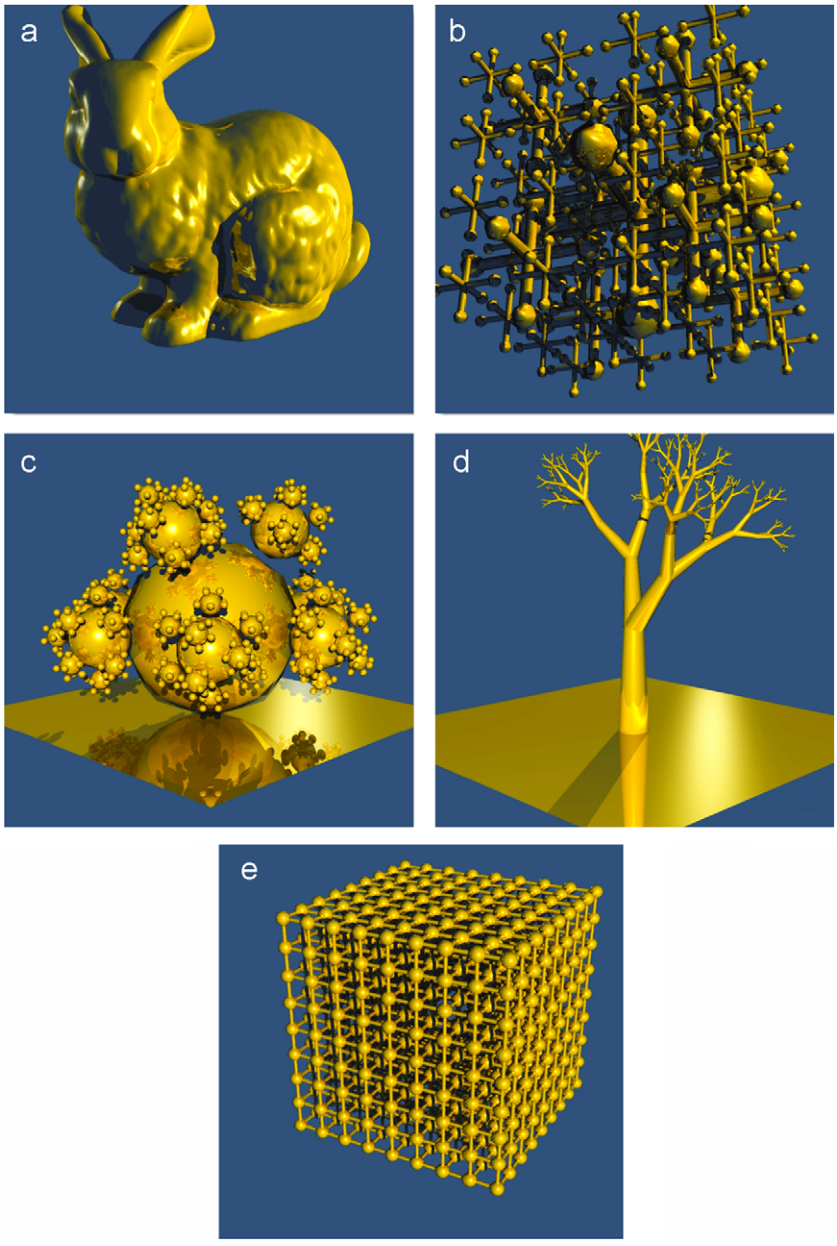

Fig. 15. (a) Bunny (69451 tris), (b) Jacks (24528 tris), (c) Sphereflake (88562 tris), (d) Tree (67454 tris) and (e) Lattice (125388 tris).

rays are more coherent since they are directed to a single point in space (position of the light source). Table reveals that the performance gap between EACD and other techniques broadens as the coherency is lost. This is expected since EACD can reach to non-empty voxels with less number of steps and thus requires less data dependent branching operations.

Table 5
Ray tracing time results of *Lattice* scene in milliseconds (grid size: 128 × 128 × 128)

		EACD	ACD	PC	DDA
Ray cast	Traverse	29,7	40,08	106,56	126,65
	Frame	95,05	105,74	173,86	193,79
Ray cast 1SH	Traverse	66,89	92,35	258,15	626,84
	Frame	205	232,76	435,91	802,52
Ray cast 2SH	Traverse	87,46	125,26	304,67	905,74
	Frame	270,92	308,56	493,49	1096,98
Ray trace 1R	Traverse	105,32	170,4	408	970,69
	Frame	345,6	409,65	655,17	1219,13
Ray trace 2R	Traverse	192,26	330,05	760,25	2108,27
	Frame	649,02	787,97	1227,54	2590,06
Ray trace 3R	Traverse	244,39	421,22	961,26	2788,76
	Frame	833,21	1009,53	1562,62	3409,25
Ray trace 1R/2SH	Traverse	248,41	387,05	915,503	2873,38
	Frame	801,2	936,01	1474,3	3444,19
Ray trace 2R/2SH	Traverse	424,17	684,23	1592,51	5141
	Frame	1397,14	1651,37	2582,13	6151
Ray trace 3R/2SH	Traverse	545,32	888,03	2049	6741
	Frame	1816,15	2149,33	3327,42	8051

Frame is the total rendering time. The ray tracing setups are; *Ray Cast* (eye rays without shadows), *Ray Cast 1-2SH* (eye rays + one and two lights with shadows), *Ray Trace* 1-3*R* (eye rays + reflections of depth 1 to 3, without shadows), *Ray Trace 1-3R/2SH* (eye rays + reflections of depth 1 to 3 + two lights with shadows).

We measured the traversal times with frame sizes of 256×256 and 1024 × 1024 in addition to 512 × 512. When the frame resolution is increased 4 times (doubled along each dimension), traversal is slowed down by around 2 times for all methods. This is because of the fact that coherence increases as the frame size grows.

Additionally, the stall rate of fragment processors due to waiting for texture units is measured using NVPerfKit [27]. NVPerfKit is a tool giving access to some low level GPU performance counters. Stalls due to texture sampling may occur if too many incoherent texture lookups are performed. For ray traced *lattice*, average stall rate is 1.77%, 1.20%, 0.89% and 0.24% for EACD, ACD, PC and DDA kernels, respectively. Ultimately, fragment processor stalls due to texture access do not cause a bottleneck on the system.

Ray-triangle intersection testing is the most time consuming part for EACD and ACD kernels. Increasing the grid resolution is a way to reduce the number of intersection tests and thus the total rendering time. However for DDA, in most cases traversal is already the most time consuming part. Therefore, increasing the grid resolution will not improve the overall performance in most of the test scenes since it will largely increase the traversal times. Similar situation holds for the PC traversal. The only exception to this observation is the *tree* scene. This scene has highly non-uniform triangle distribution; most of the triangles are grouped inside a small number of voxels around the branch tips. This is a typical weakness of the grid based scene partitioning structures. In case of ACD and EACD traversals, increasing the grid resolution will help to improve the performance for all of the test scenes. The problem for these techniques is that, the acceleration structure is eight times as big as it is for PC and DDA. The limit for the maximum grid resolution for ACD and EACD is 256 × 256 × 256 in our hardware. As the graphics memories enlarge, this will be less of a concern, but a better solution to this problem may be to use hybrid partitioning structures.

5.4. Comparison to other GPU ray tracers

Since we focused on grid based acceleration techniques in this work, non-grid based GPU ray tracing techniques have not been implemented. However, we rendered ray casted images of 70 K *bunny* with one light source on a GPU similar to the other works for a rough comparison. Note that all methods compared below uses the same *bunny* model. As reported in Carr et al. [6], Thrane and Simonsen's BVH implementation [38] obtained 257 ms (GeForce6800 Ultra), while Carr et al.'s geometry images based BVH technique [6] has an estimated frame time of ∼ 360 ms (X800 XT PE). Two kD tree based methods as described in Foley and Sugerman [10] rendered the scene in 690 ms using the backtrack algorithm, and 701 ms using the restart algorithm (X800 XT PE). As for the comparison, we measured 141 ms without shadows and 216 ms with shadows on the average, using EACD (GeForce 6800 Ultra, grid size 128 × 128 × 96). From the results we conclude that grid based empty space skipping methods, especially EACD and ACD, is very competitive or better than other techniques for *bunny* like scenes. Additionally, from the test results EACD and ACD are also competitive at scenes with large empty spaces and moderately even triangle density in non-empty voxels. Purcell's grid based ray tracer is similar to our non-branching DDA implementation [31]. Our implementation has dynamic loops for the intersection tests and use depth buffer for early fragment culling; otherwise the two are almost identical. Therefore non-branching DDA results given in Table 3 may be used for a rough comparison to EACD.

EACD and ACD seem to be suitable for static scenes since they require a relatively time consuming pre-processing stage. Other techniques also suffer from this situation at varying degrees. Among the GPU based ray tracers, only Carr et al. [6] focused on animated scenes. It is still possible to use grid based methods hierarchically for non-deforming or articulated model animations, where each model has its own grid. Rays entering the bounding volume of a model is then transformed into the grid space of the object and traced locally. For deformable animations, the acceleration structure should be reconstructed. Wald et al. describe a ray tracer using similar approach [39]. Additionally, in a recent work Wald et al. showed that uniform grids can be used for ray tracing of dynamic scenes [41]. We think that it may be an interesting future work to study efficient ways to create the ACD and EACD acceleration grids, either fully or partially, for animated scenes.

6. Conclusion

It is crucial to explore efficient data structures and algorithms conforming to the parallel stream processing model to make best possible use of the graphics hardware. In this work we have

studied regular grid based traversal techniques and introduced a GPU based traversal algorithm using extended anisotropic chessboard distance transformations. In order to compare the performance, efficient GPU implementations of some of the previously known traversal techniques have been given. It is shown that the introduced traversal algorithm is several times faster than DDA, and considerably faster than other regular grid based empty space skipping methods. In addition, our algorithm suits well to the modern pipelined superscalar CPU architectures which support streaming parallel instructions. Therefore, presented methods can be ported to SIMD capable CPUs with some minor modifications.

As demonstrated by the results the traversal part is not the bottleneck in most cases when empty space skipping is used. In general, especially for EACD and ACD, the main determining factor of the performance is the time spent for intersection tests. The time required for intersection tests can be reduced by using finer grid subdivisions. In contrast, size of the graphics memory defines a limit on the maximum grid dimensions. Hence, huge memory requirement of the acceleration grids is currently the major drawback. As a future work we consider exploring GPU based hybrid or hierarchical acceleration techniques which can compromise between memory and speed. EACD is also suitable for direct volume rendering, since there are no time consuming intersection tests and the traversal speed is the major factor of the performance.

There are some open research areas to accelerate GPU based ray tracing further. Coherence is one of the key elements for high performance rendering. We think that it is worthwhile to perform research on increasing the level of coherency in GPU ray tracers. Rendering a scene as small tiles instead of a whole buffer, or reordering rays for coherency may help to increase performance. Finding faster, more coherent and memory efficient ways for ray recursions may also be a valuable future work. With each new generation, GPUs are becoming more parallel, powerful and flexible processors. Therefore, we believe that GPUs have great potential for achieving the goal of real time photo-realistic ray tracing.

References

[1] Advanced Micro Devices, ATI CTM Guide, Online document: ⟨http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf⟩, 2007.

[2] J. Amanatides, A. Woo, A fast voxel traversal algorithm for ray tracing, in: Proceedings of Eurographics, 1987, pp. 3–10.

[3] A. Appel, Some techniques for shading machine renderings of solids, in: SJCC, 1968, pp. 27–45.

[4] G. Borgefors, Distance transformations in digital images, Comput. Vision Graph Image Processing 34 (3) (1986) 344–371.

[5] N.A. Carr, J.D. Hall, J.C. Hart, The ray engine, in: Proceedings of Graphics Hardware, 2002, pp. 1–10.

[6] N.A. Carr, J. Hoberock, K. Crane, J.C. Hart, Fast GPU ray tracing of dynamic meshes using geometry images. in: Proceedings of the 2006 Conference on Graphics Interface, June 07–09, 2006, pp. 203–209.

[7] D. Cohen, Z. Sheffer, Proximity clouds: an acceleration technique for 3D grid traversal, Visual Comput. 10 (11) (1994) 27–38.

[8] O. Devilliers, The macro-regions: an efficient space subdivision structure for ray tracing, in: Proceedings of Eurographics'89, 1989, pp. 27–38.

[9] A. Es, H. Yalım, V. İşler, Accelerated volume rendering with homogeneous region encoding using EACD on GPU, in: Proceedings of EGPGV'06, 2006, pp. 67–73.

[10] T. Foley, J. Sugerman, KD-tree acceleration structures for a GPU raytracer, in: Proceedings of Graphics Hardware, 2005, pp. 15–22.

[11] A. Fujimoto, T. Tanaka, K. Iwata, ARTS: accelerated ray tracing system, IEEE Comput. Graphics Appl. 6 (4) (1986) 16–26.

[12] GeForce 8 Series, Web site: ⟨http://www.nvidia.com/page/geforce8.html⟩, 2007.

[13] GpGPU, General purpose GPU, Web site: ⟨http://www.gpgpu.org⟩, 2007.

[14] GPU Bench, Web site: ⟨http://graphics.stanford.edu/projects/gpubench/⟩, 2007.

[15] E. Haines, The standard procedural database (SPD), Version 3.14, 2007. Web site: ⟨http://www.acm.org/tog/resources/SPD/overview.html⟩.

[16] V. Havran, Heuristic ray shooting algorithms, Ph.D. Thesis, The Faculty of Electrical Engineering, Czech Technical University, Prague, 2000.

[17] K. Hillesland, A. Lastra, GPU floating-point paranoia, in: Proceedings of GP2 Workshop, 2004.

[18] J. Hurley, Ray tracing goes mainstream, Intel Technol. J. 9 (2) (2005).

[19] H.W. Jensen, Realistic Image Synthesis Using Photon Mapping, AK Peters, July 2001.

[20] F. Karlsson, C.J. Ljungstedt, Ray tracing fully implemented on programmable graphics hardware, Master's Thesis, Chalmers University of Technology, Department of Computer Engineering, Göteborg, 2005.

[21] M. Levoy, Display of surfaces from volume data, IEEE Comput. Graphics 9 (3) (1990) 245–261.

[22] W.R. Mark, R.S. Glanville, K. Akeley, M.J. Kilgard, Cg: a system for programming graphics hardware in a C-like language, ACM Trans. Graphics (Proc. ACM SIGGRAPH) 22 (3) (2003) 896–907.

[23] Microsoft DirectX: Home Page, Web site: ⟨http://www.microsoft.com/windows/directx/default.mspx⟩, 2007.

[24] T. Möller, B. Trumbore, Fast, minimum storage ray-triangle intersection, J. Graphics Tools (1997) 21–28.

[25] nVIDIA, Floating point specials, Web site: ⟨http://developer.nvidia.com/object/floating_point_specials.html⟩, 2007.

[26] NVIDIA CUDA Homepage, Web site: ⟨http://developer.nvidia.com/object/cuda.html⟩, 2007.

[27] NVPerfKit, Web site: ⟨http://developer.nvidia.com/object/nvperfkit_home.html⟩, 2007.

[28] J. Owens, Streaming architectures and technology trends, in: M. Pharr, R. Fernando (Eds.), GPU Gems 2, Addison Wesley Professional, 2005, pp. 457–470.

[29] J.D. Owens, Computer graphics on a stream architecture, Ph.D. Thesis, Stanford University, November 2002.

[30] T.J. Purcell, Ray tracing on a stream processor, Ph.D. Dissertation, Stanford University Department of Computer Science, March 2004.

[31] T.J. Purcell, I. Buck, W.R. Mark, P. Hanrahan, Ray tracing on programmable graphics hardware, ACM Trans. Graphics (Proc. ACM SIGGRAPH) 21 (3) (2002) 703–712.

[32] A. Reshetov, A. Soupikov, J. Hurley, Multi-level ray tracing algorithm, ACM Trans. Graph. 24 (3) (2005) 1176–1185.

[33] J. Schmitter, I. Wald, P. Slusallek, SaarCOR—a hardware architecture for ray tracing, in: Proceedings of EUROGRAPHICS Graphics Hardware, 2002, pp. 27–36.

[34] J. Schmittler, S. Woop, D. Wagner, W.J. Paul, P. Slusallek, Realtime ray tracing of dynamic scenes on an FPGA chip, in: Proceedings of Graphics Hardware 2004, August 2004, pp. 95–106.

[35] SGI—OpenGL: Home Page, Web page available at: ⟨http://www.sgi.com/products/software/opengl/⟩, 2007.

[36] M. Sramek, A. Kaufman, Fast ray-tracing of rectilinear volume data using distance transforms, IEEE Trans. Visualization Comput. Graphics 6 (3) (2000) 236–252.

[37] The Stanford 3D Scanning Repository, Web site: ⟨http://graphics.stanford.edu/data/3Dscanrep/⟩, 2007.

[38] N. Thrane, L.O. Simonsen, A comparison of acceleration structures for GPU assisted ray tracing, Master's Thesis, University of Aarhus, Denmark, 2005.

[39] I. Wald, C. Benthin, P. Slusallek, A simple and practical method for interactive ray tracing of dynamic scenes, Technical Report, Computer Graphics Group, Saarland University, 2002.
[40] I. Wald, S. Boulos, P. Shirley, Ray tracing deformable scenes using dynamic bounding volume hierarchies, ACM Trans. Graphics 26 (1, Art. 6) (2007).
[41] I. Wald, T. Ize, A. Kensler, A. Knoll, S.G. Parker, Ray tracing animated scenes using coherent grid traversal, ACM Trans. Graphics (Proceedings of ACM SIGGRAPH 2006) (2006) 485–493.
[42] D. Weiskopf, T. Schafhitzel, T. Ertl, GPU-based nonlinear ray tracing, in: Proceedings of EUROGRAPHICS'04, vol. 23(3), 2004.
[43] T. Whitted, An improved illumination model for shaded display, CACM 23 (6) (1980) 343–349.
[44] S. Woop, J. Schmittler, P. Slusallek, RPU: a programmable ray processing unit for realtime ray tracing, in: Proceedings of SIGGRAPH 2005, 2005.
[45] K.J. Zuiderveld, A.H.J. Koning, M.A. Viergever, Acceleration of ray-casting using 3D distance transforms, in: Visualization in Biomedical Computing II, Proceedings of the SPIE 1808, 1992, pp. 324–335.

Alphan Es is a Ph.D. student at the Middle East Technical University (METU) in Turkey. He received a B.S. in 1996 from the Ege University and M.S. in 2000 from METU. He has been working in the Scientific and Technological Research Council in Turkey since 1997. His research interests include real-time rendering, photo-realistic rendering, game programming and GPU based algorithms. He is a member of ACM since 1998.

Veysi İşler received the B.S. degree in computer engineering from the Middle East Technical University (METU), Ankara, Turkey, and the M.S. degree in computer engineering and information science from the Bilkent University, Ankara, Turkey, in 1987 and 1989, respectively. He has received his Ph.D. degree in the Department of Computer Engineering and Information Science at the Bilkent University in the area of parallel rendering in 1995. After he received his Ph.D. degree, he worked as a research associate at the Computer Graphics and Multimedia Laboratory, Department of Computing, The Hong Kong Polytechnic University. In 1996 he joined the Department of Computer Engineering of METU. Between 2000–2005 he worked as an R&D Director for industry. Since July 2005 he has been with the Department of Computer Engineering of METU, where he is currently an associate professor. He is also the director of Modeling and Simulation Research and Development Center of METU. His research interests include rendering, visualization, virtual reality, game technology, driving simulators and parallel graphics algorithms.