Interactive Ray Tracing

Gabriel Mistelbauer

June 1, 2007

Abstract

In this paper several ways of achieving interactive ray tracing are described. First from the software point of view common acceleration structures, such as grids and bounding volume hierarchies, are introduced. Then multi-level ray tracing and distributed ray tracing on clusters of computers are described. The second part of this article focuses on the hardware requirements for interactive ray tracing, describing cards, such as common video cards, which are specially designed and optimized for ray tracing.

1 Introduction

Interactive rendering is mostly the domain of rasterization-based algorithms that have been put into highly integrated and optimized graphics chips. In the last few years their performance increased rapidly and they are provided at low cost such that today's PCs already come equipped with such cards. Although the performance increased so much some applications require more realism and much more complex scenes to render. Some examples are architectural and engineering design projects such as entire power plants or airplanes.

Current graphics hardware is limited due to the linear cost in the number of polygons. To achieve interactive frame rates a sophisticated preprocessing is required to reduce the number of rendered polygons per frame. This can be done by techniques such as level-of-detail, occlusion culling, portal-rendering and many others. Another problem rises when it comes to scaling the graphics pipeline to parallel processing. The main issues were communication requirements between parallel units and to avoid redundant processing and data storage (e.g. for textures).

Another problem is achieving high realism. Although current graphics devices have several features for shading, multi-texturing, vertex and fragment shader programming and many more, it is sometimes difficult for applications to express even simple shading effects.

Another rendering algorithm, which is called ray tracing, has been used for many years and solves some of the problems mentioned above. Ray tracing is known for generating high-quality images but is also well known for its long rendering times due to the high computational costs. This cost results from traversing many rays through the scene, intersecting each ray with the geometric objects, shading the visible surface and finally sending the resulting pixel to the screen. Thus ray tracing is often used as an offline rendering technique where rendering time is less important than image quality. [WS]

In ray tracing the most expensive task is the tracing of rays into the scene and the computation of the intersections of the objects contained in the scene with those rays. Many rays do not even hit an object and the intersection calculation is therefore not necessary. To avoid this, a structuring of the space is needed to eliminate these needless intersection computations which results in a performance gain. Some structures used are grids, kD-trees and other spatial subdivision methods.

Another approach to achieve real-time performance in ray tracing is connecting several computers over a network and split the ray tracing computations along them. Even in ray tracing this method is really applicable because tracing rays is an inde-

pendent operation and can be easily parallelized and distributed.

Although ray tracing takes so much time for rendering is offers some advantages over rasterization-based techniques.

Flexibility In ray tracing either groups of rays or single rays can be traced, which offers efficient computation of the just required information (e.g. sampling narrow glossy highlights). [WS]

Logarithmic Complexity and Occlusion Culling Complex scenes can be efficient rendered with ray tracing because of its logarithmic complexity in the number of scene primitives, in comparison with the linear complexity for rasterization-based algorithms. Another advantage is the built in occlusion culling of ray tracing and using a simple search structure can quickly locate the relevant geometry in a scene and stops its front to back processing if visibility has been determined. [WS]

Efficient and Simpler Shading In ray tracing shading is only applied to samples after visibility checking which avoids redundant computations for invisible geometry. Even programming shaders for special lighting effects is much simpler because it is at the core of realistic rendering. [WS]

Correctness Ray tracing computes by default physically correct reflections, refractions and shading. If the correctness is not necessary or too costly, ray tracing can use the same approximations used by rasterization-based approaches to achieve these effects. Contrary rasterization-based techniques can only use approximation and it is difficult to achieve realistic effects. [WS]

Parallel Scalability Ray tracing can be easily parallelized and given the exponential growth of hardware resources, ray tracing should be able to better utilize it than rasterization-based algorithms, which have been difficult to scale efficiently. [WS]

Coherent Ray Tracing Adjacent primary rays often operate on the same data during ray traversal, intersection and shading. The same is true with a lesser degree for secondary rays, such as shadow rays for the same light source, or reflection rays off the spatially close intersection points from coherent primary or event secondary rays. This leads to group coherent and correlated rays into packets of rays and traversing, intersection and shading them in parallel. Furthermore this can be improved by using data parallel operations in SIMD (Single Instruction Multiple Data), such as Intel's SSE or AND's 3DNow! instructions, fashion and resulting in interactive ray tracing on standard PCs. [WS]

Optimizing Code and Memory Access Ray tracing is not bound to CPU speed as the general opinion is, but is usually bandwidth-limited by access to the main memory. Specifically if shooting incoherent rays, as done in some global illumination algorithms, this will result in more or less random accesses and bad cache performance. On current PCs, bandwidth to main memory is 10 times less than to primary caches so using an optimized memory layout for the most common used data structures will result in better cache performance. Given the big latency of accessing main memory it will be reasonable to load data into the cache before it is used and do not fetch it on demand. So with carefully designed data structures bandwidth, cache misses and false sharing of cache lines are reduced. [WS]

Ray Triangle Intersection The core part of ray tracing is the computation of ray triangle intersections. Here Arenberg's algorithm is taken to optimize the usage of the transformation unit as it computes the intersection and the dot product of the ray direction and triangle normal vector for free. The method used is called the unit triangle intersection method as it first transforms the triangle into a coordinate system, where it is the unit triangle. Next a simplified intersection test is performed with the transformed ray.

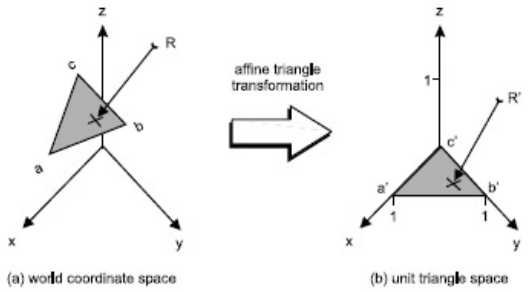

Figure 1: *Ray triangle intersection*

Since the triangle is transformed into the unit triangle, the dot product of the transformed ray direction and the unit triangle's normal, which is leading along the positive z-axis, is simply the z-coordinate of the transformed ray direction. [SWW+]

2 Acceleration structures

When tracing a huge number of rays through a scene it becomes very inefficient to intersect every object with every ray, because a lot of intersection calculations can be omitted if the objects are stored in a special way to determine quickly if a ray needs to be intersected with an object. This approach leads to the following structures that are called acceleration structures, because they greatly increase performance. Hence such a structure can quickly determine if a ray hits an object or not and with which objects the ray must be intersected. Some acceleration structures are based on a tree like structure to benefit from the logarithmic traversal costs.

Most acceleration structures have been designed and optimized for static scenes, where objects are not moving. Recently with achieving interactive ray tracing, dynamic scenes rose up and new requirements for the acceleration structures have come. For example the update operation must be very fast to ensure the performance gain, which is the goal of such structure, because if it takes too long the benefit of the acceleration structure will vanish. This allows interactive animated and modified scenes as it is common from scan-line algorithms.

To create such an acceleration structure either the space containing the object can be partitioned or the list of objects can be partitioned.

2.1 Partitioning Space

If the space is partitioned in disjoint volume elements a ray intersects only those objects that are referenced in the volumes the ray intersects itself. Although this greatly increases performance, the main drawback of this approach is that objects are often referenced multiple times, because they may overlap with more than one volume element. This results in a larger memory footprint and a mailbox mechanism must be used to avoid multiple intersections of one ray with the same object. Grids and structures based on binary space partition such as BSP-trees or kD-trees fall into this category. A problem with such spatial subdivision structures is if an object moves outside the extent of the spatial subdivision, some structures require a complete rebuild.

2.1.1 Grids

Without modifications, grid spatial subdivisions for static scenes are already useful for animated scenes because insertion and deletion operations are practically simple and even the ray traversal costs are low. However, if an object moves outside the extent of the spatial subdivision, the acceleration structure has to be rebuilt. To avoid this expensive operation the grid is logically replicated over space. If an object now crosses the boundaries of the grid, it wraps around before reinsertion. Ray traversal also wraps around the grid if a boundary is reached. To offer a stopping criterion for ray traversal, a logical box is maintained which contains all objects, including the ones that have crossed the original perimeter. As this scheme does not require grid re-computation every time an object moves far away, the cost will be significantly lower while maintaining the spatial subdivision. The grid is built as usual in a pre-processing step and the initial bounding box enclosing the whole scene is called the physical bounding box. If an object moves outside the physical bounding box the logical bound-

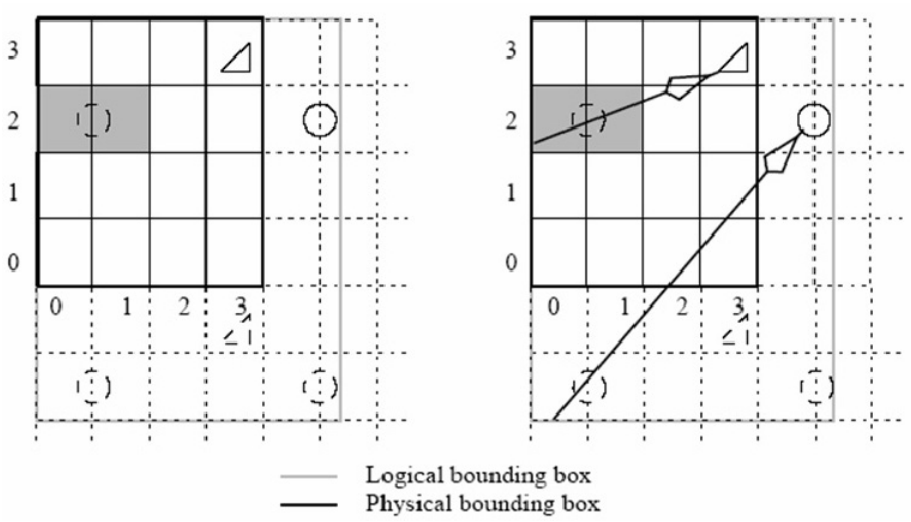

Figure 2: *Grid insertion (left). The sphere has moved outside the physical grid, now overlapping with voxels (4, 2) and (5, 2). Therefore, the object is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: ray traversal through extended grid. The solid lines are the actual objects whereas the dashed lines indicate voxels which contain objects whose actual extents are not contained in that voxel.*

ing box is extended to include all objects. Initially the logical bounding box equals the physical bounding box one. When the logical bounding box becomes much larger than the physical one, there is a trade-off between traversal speed and rebuilding cost of the grid. [RSH]

2.1.2 Hierarchical Grids

In order to make the insertion and deletion of objects independent of their sizes, hierarchical spatial subdivisions are used, such as octrees, binary trees and hierarchical grids. An extension of hierarchical grids is that objects can also be stored in intermediary nodes or even in the root depending on the size of the object (normally objects are only stored in leaf nodes). If an object needs to be inserted, the grid level needs to be computed first which is done by comparing the size of the object with respect to the size of the scene. A straightforward heuristic is to determine the grid level from the lengths of the diagonals of the two bounding boxes. This ensures constant time insertion but results in a little more complicated traversal algorithm. Traversal starts at a leaf node which may first be mapped to a physical leaf node as mentioned in the preceding paragraph. The ray is intersected with this voxel and all its parents until the root node is reached. This must be performed because objects at all levels of the hierarchy occupy the same space as the currently traversed leaf node. Traversal finishes if an intersection is found within the space of the leaf node and if not, the next leaf node is selected and traversal is repeated. This results in visiting many parent nodes a lot of times for the traversal of the same ray. To avoid this problem the common ancestor of the current leaf node and the previously intersected one need not be traversed again. Another optimization is that the highest levels of the grid may not contain any objects and thus ascending to the highest level in the grid is not always necessary and it can be stopped when the largest voxel containing objects is visited. [RSH]

2.2 Partitioning Object

When a list of objects is partitioned, each object is referenced at most once. Another advantage is that an object is intersected at most once with a ray and thus the use of a mailbox algorithm becomes obsolete. One disadvantage is that groups of objects often cannot be disjoint. Bounding volume hierarchies for example are part of this category. [CW06]

2.2.1 Bounding Volume Hierarchies (BVH)

A BVH is a tree of bounding volumes, where each inner node corresponds to bounding volume (BV) containing its children. Each leaf node contains at least one primitive. Common bounding volumes are axis aligned bounding boxes (AABBs), oriented bounding boxes (OBBs) or spheres. AABBs make a good choice because they are well-balanced between tightness of fitting the primitives, computational cost and they provide efficient algorithms for ray-box intersection calculations which are really essential in ray tracing.

In static scenes kD-trees outperform AABBs because in the worst case only one plane intersection calculation has to be made for the split plane in kD-trees and even six plane intersections need to be performed in AABBs. Another drawback of AABBs is that they do not provide a real front-to-back ordering during ray traversal which results in if a ray intersects a primitive, the algorithm cannot terminate (as it is possible for kD-trees) and needs to continue in order to find all the other intersections. Even the storage is better in kD-trees because a node requires much less space than one for AABBs but on the other hand AABBs often could do with fewer nodes representing the whole scene than kD-trees. This result from kD-trees often references primitives multiple times in the hierarchy because no better split plane could be found and AABBs reference them only once.

Although AABBs have all these disadvantages against kD-trees the former are much better updated in linear time with incremental techniques whereas there do not exist any equivalent algorithms for kD-trees. Another advantage which is really useful is that AABBs provide a tighter fit to the primitives with fewer levels in the tree than kD-trees because the latter need multiple subdivisions in order to discard empty space.

AABBs hierarchies are constructed in top-down fashion by recursively dividing the set of initial primitives into two subsets until each subset has the desired number of primitives. Which number to choose fits best in regard to performance varies but one is a good choice although it leads to a deeper tree, but node intersection is much cheaper than primitive intersection. The most important operation during construction is to find a divider of the two subsets which is optimized for ray traversal. Regardless which heuristic is used for finding the split the construction time is $O(n \cdot log_k(n))$ where k is the number of children in each node and n the number of primitives.

For dynamic scenes the AABBs need to be updated every time an object is moved or deformed. To avoid reconstructing the entire tree, the hierarchy is updated recursively in postorder traversal starting at the root node. If a leaf node is reached, a new BV which fits tightest to the deformed geometry is computed. The traversal continues from the leaf node bottom-up and the BV of an intermediate node is initialized with the BV of the leftmost node and expanded with the BVs of the rest of its sibling nodes. This gives a complexity of $O(n)$ which is much better than the construction method.

Although the update method is much faster, the BVH needs to be rebuilt some time, because the grouping of the primitives and the hierarchy does not change on update and this can lead to overlapping BVs which degrade the ray traversal because more intersections between AABBs and the rays are needed. Here it is necessary to have a criterion which determines when a rebuild has to take place or when an update operation is satisfying. One heuristic used is to determine the ratio between a parent node's surface area to the sum of the area of its two children. The larger the sum becomes, the more imbalance exists in sizes. Now the difference is computed in every inner node and summed up until the root is reached. Then this sum is normalized by the number of inner node (n-1). This relative value describes the overhead of updating the BVH instead of rebuilding it. Finally if this relative value exceeds a chosen threshold value

the tree must be rebuilt. [LYTM]

3 Multi-Level Ray Tracing

Contrary to the common approach of finding objects that are intersected by a ray, the approach of eliminating all objects that are not intersected by a ray is followed here. To achieve this a frustum culling algorithm is used.

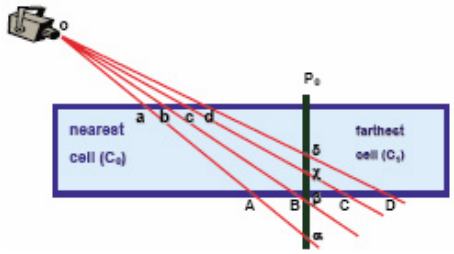

Figure 3: *Tracing a group of rays lead to different rays go through differen cells in the tree*

Initially rays are packed together into groups but then there must be dealt with the situation that rays can follow different paths in the acceleration structure (kD-tree). Now there are two possibilites to handle such situations: Work with a variable number of rays and regroup them every time or mask the inactive rays in the group. Some problems of these approaches are that the benefit of processing multiple rays simultaneously cannot be realized and that the overhead of tracking the inactive rays may result in a peformance loss. So grouping rays should be done if all rays have the same direction and intersect a split plane of the acceleration structure in the same direction.

The frustum is formed by 4 planes that intersect in one point. Each plane is defined by this intersection point and a normal vector which points outward the frustum. For each plane of the frustum, there are two vertices important belonging to the axis-aligned box: the one laying farthest in positive direction of the plane's normal (p-vertex) and the one laying farthest in the negative direction of the normal (n-vertex). By inserting the n-vertex into the plane equation the decision can be made whether the n-vertex and thus the whole box is laying outside the plane. This can be done for each plane resulting in an easy rejection algorithm. The p-vertex can be used to determine whether or not the entire box is completely inside the frustum.

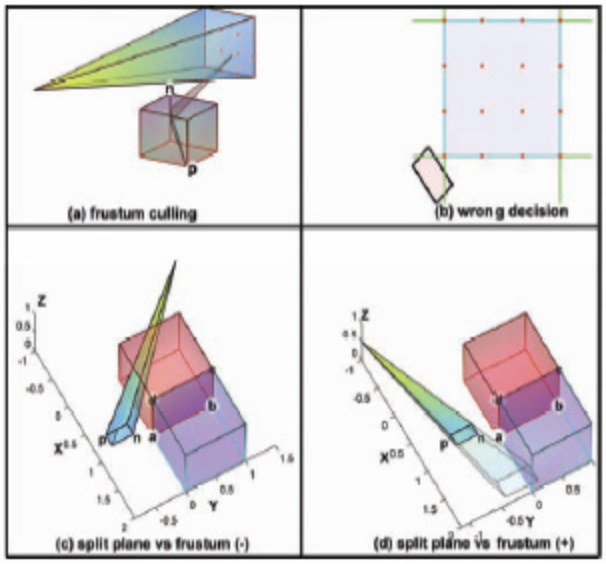

Figure 4: *Direct (a, b) and inverse (c, d) frustum culling*

Although this rejection algorithm works quite well it fails the n-vertex outside test for each of the frustum's plane assuming it may intersects if the whole box is completely outside the frustum. To avoid this, the roles of the frustum and the AABB is reversed. The planes of the AABB are taken to separate it from the frustum instead of the other way around. Here the normals of the frustum's planes are not used anymore. Instead the rectangular bounds for each axis-aligned plane which encloses frustum/plane intersection are taken. This approach makes it possible to use groups of rays, that do not share the same origin.

Another key feature of the inverse frustum culling algorithm with a kD-tree as acceleration structure is that it uses values only for one axis at a time and during traversal different axis will processed at each level which results in effective culling of the current

cell. This avoids unnecessary intersection tests by using frustum/plane intersection data and peforms additional clipping tests at the leaf nodes. At every leaf node clipping is done for all possible pairs of axis (xy, yz, xz) against the 6 AABB box faces.

Some times when using kD-trees the entire tree has to be traversed, because the split planes are not chosen optimal. Therefore it would be better to have an alternative way of finding the entry points in the tree and start traversing at these nodes. One way is to use the information of the frustum culling algorithm and find the optimal entry point for all rays that hold the following conditions: All rays go in the same direction and for any given plane a rectangle inside the plane which contains all possible ray/plane intersection points is computed (the rectangle does not have to be tight). This entry point search finds the optimal place to start the intersection search. That is way it is called multi-level ray tracing, because the tracing of rays may start at several levels. The algorithm is a depth-first traversal of the visible nodes in the tree allowing for early escape of the traversal of branches that do not contribute to the final result. For a detailed description of the algorithm see [RSH05].

4 Parallel Ray Tracing

Another approach to speed up ray tracing is parallelization for which ray tracing fits very well because each pixel in the frame buffer is independent from all others. This leads to parallelize the computations for every pixel or blocks of pixels.

A way of implementing a parallel ray tracer is using parallel shared memory and the Master/Slave demand driven scheduling. Although this implementation is simple it achieves reasonable performance. It basically consists of two different tasks, the master task which is responsible for pushing a primary ray for every pixel which needs to be traced in a shared queue on which the slave tasks a listening. If the master task has pushed all rays in the queue the slave tasks pop one and trace it into the scene computing its color. If all slave tasks have finished, the shared queue is empty and the final frame can be presented. [PWS+]

The pseudo code of the master task is:

Algorithm 1 Master Task

```
initialize model
initialize ray tracing slaves on each free CPU
loop
update viewing information
lock queue
place all primary rays in queue
unlock queue
when the queue is empty redraw screen and handle
user input
end loop
```

The ray tracing slaves are simple programs that grab primary rays from the shared queue and compute the color values of the pixel:

Algorithm 2 Slave Task

```
initialize memory
loop
if queue is not empty then
lock queue
pop ray request
unlock queue
compute RGB for pixel
write RGB into frame buffer pixel
end if
end loop
```

Although this implementation will work it has a great synchronization penalty because every pixel is independent and needs to be synchronized. A better solution is to work on larger basic task sizes assigning groups of rays to each processor. The larger these groups get, the less synchronization has to be done but if the groups get too large the performance gain is lost due to poor load balancing because all processors have to wait for the last job to finish the frame before they can start over to the next frame. [PWS+]

This problem can be addressed by decreasing the size of the jobs linearly starting from a high number of pixels. All pixels have to be traced before the screen swaps the buffers. This is called the conventional

mode. This mode is limited due to the synchronization of the processor which takes for example in Irix and average of 5 msecs to synchronize 64 processors. This limits the scaling at high frame rates. [PWS+]

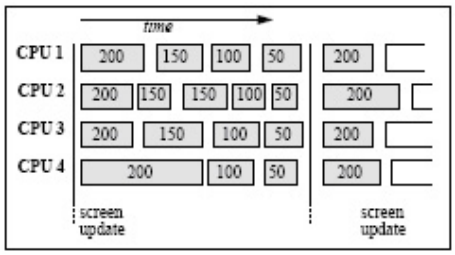

Figure 5: *Operations of a ray tracer in synchronous mode. Numbers in boxes are pixels of one tile being processed by one CPU. All pixels have to be traced before the screen swaps the buffers.*

Another approach is using the frameless mode. Here the screen and viewpoint are updated synchronously, but the pixels are updated according to an asynchronous quasi random pattern. [PWS+]

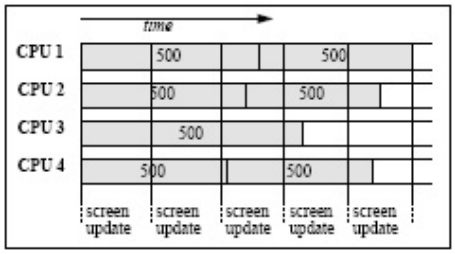

Figure 6: *Operations of a ray tracer in asynchronous (frameless) mode. Screen constantly swaps buffers and each CPU repeatedly processes its tile.*

Here a static pixel distribution is assigned to the rendering threads and every processor has a list of the pixels it will update which requires minimal inter-thread synchronization. Each rendering thread draws asynchronously the buffer to the screen at regular intervals. The display thread updates the screen at some user-defined frame rate. When updating the image pixels can be blended into the image which results in a smoother image because some samples have an exponential decay. [PWS+]

Grouping rays into packets are not always perfectly coherent because some rays may hit one object while others hit a second object and in the worst case each ray does something different. To avoid this ray packets can be split into sub packets that after intersection or shading. Splitting the packet into sub packets is done by finding a run of rays which share some common characteristics, for example the hitting the same material shader. [BSP]

5 Distributed Ray Tracing

Although distributed ray tracers are designed to run distributed on a cluster of PCs the client/server approach is used for ray tracers as described with the master and slave tasks. Here the master centrally manages a number of slave machines by assigning a number of tasks to each client. The clients then perform the ray tracing computations and send the results back to server in form of color values for each pixel.

For realtime ray tracing there are two methods of splitting the task of ray tracing into independent subtasks: object space and screen space subdivisions.

Object space approaches require the entire scene data to be distributed across a number of machines, based on the initial spatial partitioning scheme. Rays are then forwarded between clients based on the next spatial partition pierced by the ray. The resulting network bandwidth used can be too large and tracing individual rays can be traced much faster than they can be transferred across the network.

In screen-based approach clients compute disjunctive regions of the same image. The main disadvantage of this parallelization method is that it requires a local copy of the whole scene to reside at each client.

For communication in parallel systems standardized libraries such as MPI or others are used although they are very powerful tools they often do not meet the requirements which are needed for an interactive ray tracing environment. A better approach is to use the TCP/IP protocol which allows extracting the maximum performance out of the network. Another

optimization is grouping small messages into bigger packets which results in a better network throughput when a lot of small packets are sent. On the other hand this technique may lead to more latency that is sometimes not required, for example, when sending tiles to the clients because minimum latency is necessary here. The update propagation used is to send only those settings that have changed from each frame to the next because network bandwidth is not high enough for sending the entire scene to each client every frame.

Asynchronous communication is used between the clients and the server to minimize the latency. While clients are just rendering the current frame the server prepares the new settings of the scene for the next frame asynchronous because this may take several computations. When finished preparing the server waits for the clients to complete rendering the current frame, displays the frame and then starts over specifying the next frame. This results in what is called asynchronous rendering. At the client side while rendering the current tile one thread already receives the scene updates from the server immediately to further reduce communication latencies. Once the client's rendering threads have finished most of the data for the next frame has already arrived.

Instead of assigning every client only one tile to render at a time the client request several tiles in order to prefetch them. This results in better client utilization because clients do not have to wait for the server sending the next tile. In order to further improve coherence the same image tiles are assigned to the same clients in subsequent frames whenever possible. [WBDS]

6 Ray Processing Unit (RPU)

Hardware support for ray tracing was held back for a long time because of three main issues: a very large amount of floating point computation has to be performed, flexible control flow (such as recursion and branching) should be supported and at last the handling of the memory access patterns for very large scene data is very difficult.

Although realtime ray tracing has been achieved on a single high performance processor for larger and even more complex scenes a cluster of CPUs is still needed to fulfill the realtime task. This is the main drawback for software solutions, because the price needed to be paid to achieve realtime performance with a cluster of CPUs is very high. Even if more cores were placed on a single motherboard this would take some years to reach the level of performance needed.

Today in nearly every PC there is a GPU available which is a graphics processing unit used for speeding up 3D applications such as games. One drawback is that the current GPUs are only used for scanline rendering algorithms and cannot be easily adapted to ray tracing. Therefore an extra chip is needed with the requirement of flexibility as the GPUs. Thus a programmable interface like the shader languages used on GPUs would be of great efficiency.

Ray tracing is compute intensive, recursive and highly parallel algorithm with complex control flow requirements. The large number of mostly unstructured memory access can be reduced by analyzing the coherence between rays. Most operations in ray tracing are floating point vector operations.

These properties lead to the following architecture design:

Vector Operations Four components single precision floating point vectors are the base type in the Shader Processing Unit (SPU). Four vectors are used which takes the advantage of the available instruction level parallelism. [WSS]

Threads Here the advantage of data parallelism in ray tracing is taken for the multi-threaded design. For every primary ray a new thread is created. The state of these threads is maintained in hardware and the SPU can switch between the threads. [WSS]

Chunks Ray tracing has a huge bandwidth requirement which can be reduced by exploiting the high coherence between adjacent rays. Therefore M threads are created and executed synchronously in SMID mode in parallel by multiple SPU. Because all threads are executed on

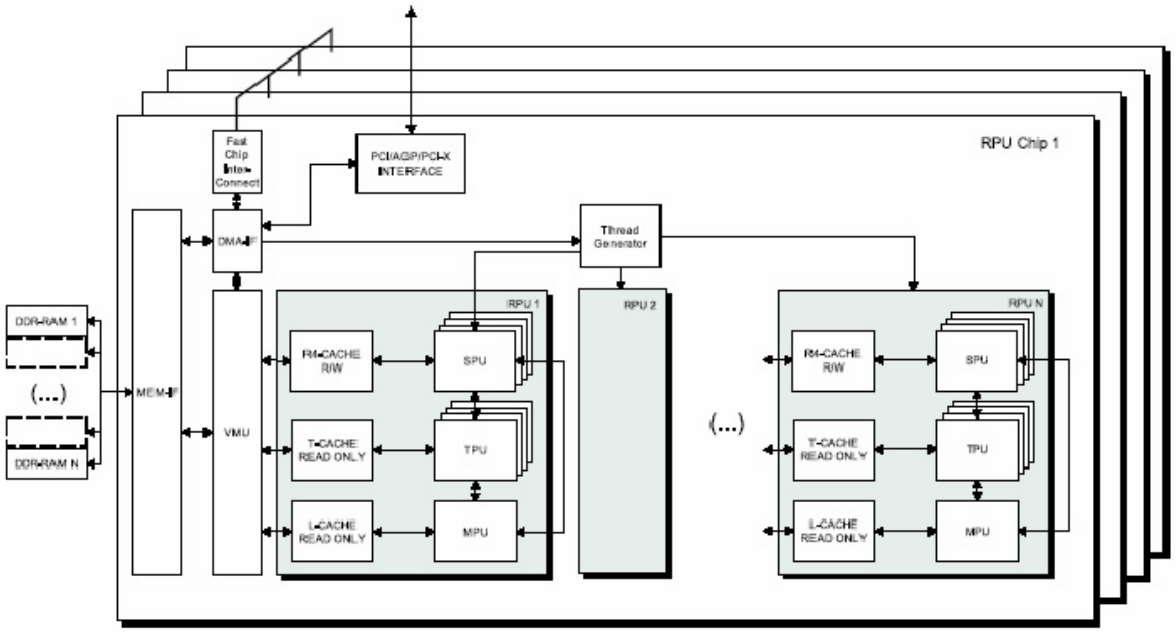

Figure 7: *RPU Architecture*

the same instruction memory requests are highly likely for coherence rays. [WSS]

Control Flow and Recursion The architecture supports conditional branch execution, full recursion, which is necessary for ray traversal, using masked execution and a hardware maintained register stack. [WSS]

Dedicated Traversal Units Here a k-D tree is used as index structure to speed up ray traversal. This unit is not fully programmable to gain the maximum benefit of the hardware solution in performance and efficiency. [WSS]

Memory Access All memory accesses go through small dedicated caches to reduce external bandwidth. [WSS]

General Purpose Computing The design has been optimized for the properties of ray casting: high data coherence, high instruction and data parallelism and a large number of short vector operations. However it also supports integer arithmetic and memory read write operations.[WSS]

Scalable Design The RPU architecture is a scalable multi-core design supporting many independent RPUs working together in parallel. Each RPU contains multiple SPUs and TPUs working on coherent chunks of rays. Another advantage like in software solutions is that multiple RPUs can be combined within a chip and across multiple chips, boards, or even clusters of PCs.[WSS]

Architecture The thread generator forks a new thread each time a hardware thread is available. Then chunks of M threads are scheduled on demand to the SPUs. When the shaders execute a trace instructions control is transferred to the TPUs, which synchronously traverse chunks of rays through a spatial index, a k-D tree. While the spawning thread is suspended the SPUs continue executing instructions asyn-

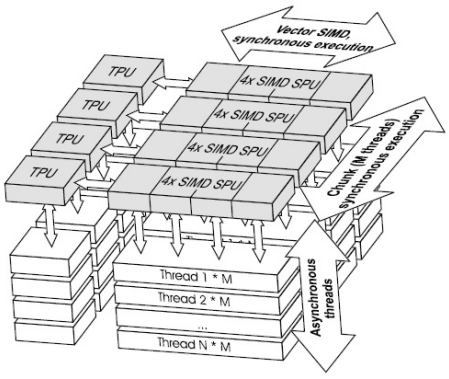

Figure 8: *RPU: Multiple Shader Processing Units (SPUs) spawn M threads. On the Traversal Processing Units (TPUs) chunks of rays are traversed through a kD-tree asynchronously to the SPUs.*

chronously from other threads. When traversal reaches a non.empty leaf node the MPU is called to iterate through the entries (objects or primitives) stored there. For each remaining entry an implicit function call is performed on all corresponding threads and the threads are scheduled onto the SPUs again. Each unit, the SPUs, TPUs and the MPU, have separate first level caches which are fed through the Virtual Memory Management Unit (VMU). [WSS]

Shader Programming Model One major difference between RPU and GPU programming is that on GPUs only local shading can be performed while on RPUs global shading effects can be done by tracing secondary rays through the scene. Another advantage is that the geometry is fully separated from the shading and multiple shaders are independent too. [WSS]

7 SaarCOR

7.1 Implementation Issues

To accelerate ray triangle intersections, a spatial index is used that subdivides space and quickly locates objects in space. The creation of the index, here a kD-tree, is linear in the number of scene primitives. This seems to remove the advantage of the index in dynamic scenes, but large parts of the scene remain static over a long time and this amortizes the building of the index over many frames. Primitives are split into separate objects which move under a single transformation with respect to the scene. These objects have an index structure on their own, which is embedded in the top-level spatial index under their specific transformation. Now only the top-level index must be rebuilt if an object is moving.

When a ray encounters an object in the top-level index it is transformed to the object's local coordinate space and can now traverse through the already computed index structure of the object. Multiple instances can be made from one object by simply storing the transformation and only the reference to the object. This approach maintains the logarithmic cost in scene complexity even if the scene is dynamic with multiple instances.

This approach is applied to the hardware too where a functional unit performs the transformations of the rays. As this unit is idle most of the time another approach of the usage of this unit in ray tracing is taken.

The transformation process consists of one 3x3 matrix multiplication for the triangle and a vector addition for the ray's origin and another matrix multiplication for the ray's direction. This must be performed by the transformation unit, but since matrix multiplications are expensive in hardware terms only one matrix multiplication unit together with the vector addition unit is implemented. The remaining transformation of the direction is done by feeding zeros to the correct coefficients. Thus there are two steps to be done by the transformation unit to transform a ray. This can be reduced because very often many rays with the same origin (primary rays or shadow rays) need to be transformed. [SWW+]

7.2 Architecture

The Dynamic SaarCOR architecture extends the SaarCOR architecture to support dynamic objects and multiple object instantiation. A virtual memory architecture for ray tracing is used to remove the

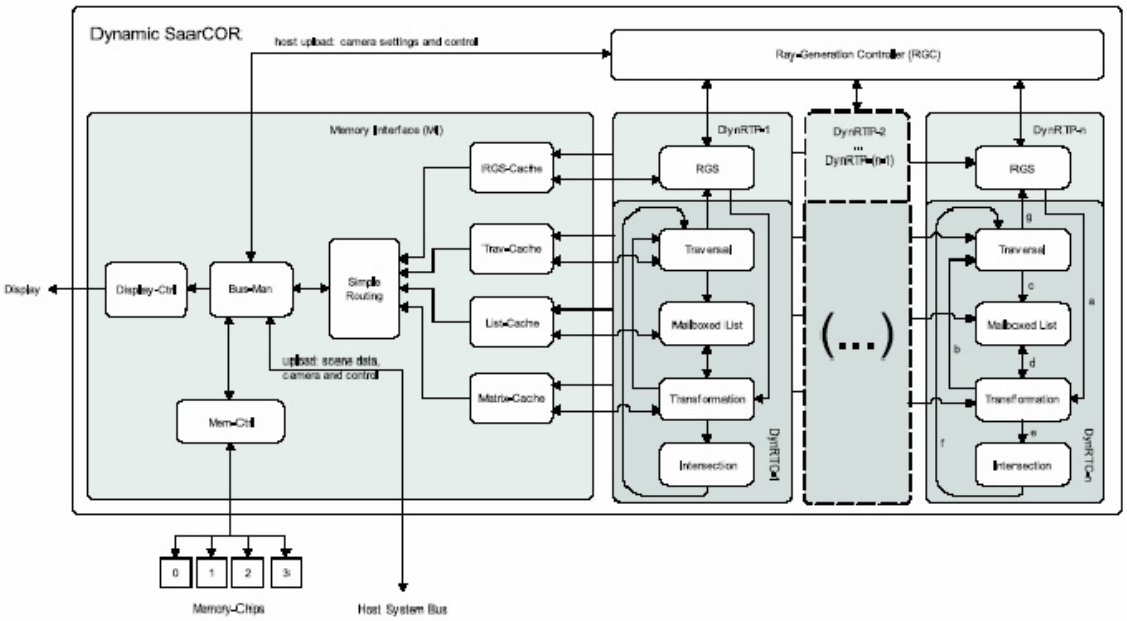

Figure 9: *SaarCOR Architecture*

need to have the entire scene in the graphics memory which is only used as a cache and to load the required scene data from the host memory on demand.

The memory chips store the scene data as well as the frame buffer. The rendered image is presented by the Display controller. The core of the ray tracing algorithm is included in the dynamic ray tracing pipeline (DynRTP). A ray is started at the ray generation and shading unit (RGS) which generates an initial ray and transformation which are sent via path a to the dynamic ray tracing core (DynRTC). Then the transformation is applied to the initial ray by the transformation unit and the transformed ray is sent to the traversal unit via path b. The traversal unit traverses the transformed ray through the top-level kD-tree until a leaf node is found. Then the ray with the list of objects is forwarded to the mailboxed list unit via path c, which sequentially fetches objects referenced in the list while ignoring objects already visited by the same ray (for which the mailboxes are used). On path d the ray is sent together with a transformation to the transformation unit to map the ray into object space. Then the ray is sent to the traversal unit again via path b to start the bottom-level traversal. The same process starts over except that leaf nodes now contain lists of triangle transformations used for transforming the ray into triangle space for intersection. Then the transformed ray is forwarded on path e to the intersection unit whose results are returned to the traversal unit via path f. The operation continues at the list until all elements have been processed. Finally the results are shipped back to the RGS unit for shading via path g. A multi threaded approach is used for processing rays simultaneously.

All memory accesses of the DynRTP are handled by the memory interface (MI) which contains individual caches that share the same connection to the memory via the Simple Routing unit. This unit acts like a multiplexer for memory requests and uses a labeled broadcast for memory responses. The memory controller (Mem-Ctrl) manages the accesses to the external memory chips. For scalability several DynRTPs are supported and the work done by each Dyn-

RTP is controlled by the ray generation controller (RGC).

The floating point units of SaarCOR implement single operations, auch as additions, comparisons and multiplications where 24 bit floating point numbers with 16 bit mantissa are a good choice between accuracy and hardware cost. Six banks of SRAM exist of which two are used to implement the double buffer with a resolution up to 1024 x 768 pixels. Another bank is used to store the shading data and the others are used for the kD-tree and the object and triangle transformations. Packets of four rays are used which are traversed in parallel and intersected sequentially because a single FPGA can only hold one intersection and typically more traversal operations than intersections are performed. [SWW+]

8 Conclusion

Summing software aspects up there are some ways to accelerate ray traversal even in dynamic scenes without rebuilding the whole acceleration structure every frame. Although this sounds quite good letting the acceleration structure be degenerated until a threshold is reached, results in some performance loss. Another way of speeding up ray tracing is to parallelize it which can be done quite easily, because the rays can be traced independently of each other. Further this leads to spread the computations for the ray traversal to several computers which results in distributed ray tracing. Here great effort needs to be done to minimize the network latency and maximize the utilization of every computer. Although ray tracing in software was used as offline rendering in the past, recently interactive frame rates have been achieved even on single core processors.

Concluding hardware we have seen that interactive frame rates can be optained with ray processing units (RPUs) that are like the common graphic processing units (GPUs) but optimized for ray tracing. Although they have reasonably little performance in regard to the GPUs they perform quite well.

Finally all these approaches achieve interactive frame rates but still need some special efforts. Either very sophisticated optimization techniques in software or implemenations in hardware.

References

[BSP] James Bigler, Abe Stephens, and Steven G. Parker, *Design for parallel interactive ray tracing systems.*

[CW06] Alexander Keller Carsten Wächter, *Instant ray tracing.*

[LYTM] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha, *Rt-deform: Interactive ray tracing of dynamic scenes using bvhs.*

[PWS+] Steven Parker, WilliamMartin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen, *Interactive ray tracing.*

[RSH] Erik Reinhard, Brian Smits, and Charles Hansen, *Dynamic acceleration structures for interactive ray tracing.*

[RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley, *Multi-level ray tracing algorithm.*

[SWW+] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek, *Realtime ray tracing of dynamic scenes on an fpga chip.*

[WBDS] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek, *Interactive ray tracing on commodity pc clusters. state of the art and practical applications.*

[WS] Ingo Wald and Philipp Slusallek, *State of the art in interactive ray tracing.*

[WSS] Sven Woop, Jörg Schmittler, and Philipp Slusallek, *Rpu: A programmable ray processing unit for realtime ray tracing.*