# Hardware design for ray tracing

Jae-sung Yoon

## Introduction

Realtime ray tracing performance has recently been achieved even on single CPU. [Wald et al. 2001, 2002, 2004] However, higher resolutions, complex scenes, and advanced rendering effects still require more performance.

Ray tracing is highly parallel algorithm because the contribution of each ray to the final image can be computed independently from the other rays. To exploit this massive parallelism, multi-core, multi-threaded hardware support is essential and efficient.

For increased performance, ray tracing require spatial index structures for quickly finding the respective subset of rays. One drawback of spatial indices is that dynamic changes in the scene require re-computation of the index. But, in recent hardware architectures[Woop 2005],[Schmittler 2004],[Purcell 2002], only static scenes or software-supported dynamic scenes are used.

Therefore, my goal is to propose an overall ray tracing hardware architecture including update of acceleration structure.

## Related Work

[Schmittler 2004], [Purcell 2002], [Woop 2005] architectures are basically based on the property of ray-tracing's parallelism. For high performance, they use multi-core and multi-threading approaches.

SaarCOR[Schmittler 2004] used dedicated hardware for ray-tracing. And RPU[Woop 2005] upgrades this to have more programmable hardware. These two works used the dynamic scene management scheme in [Wald 2003]. This scheme is based on the fact that large parts of a scene often remain static over long periods of time. So, they separate the scene into independent three classes of objects. One is the static objects. Second is the object undergoing affine transformation. Third is the object with unstructured motion. Static objects and affine transforming objects can totally removes the reconstruction cost for hierarchically animated objects. So, this method can reduce the cost of structure reconstruction. But, in the SaarCOR and RPU architecture provides

no special support for building spatial index structures. This task has to be performed by the host CPU.

Figure 1 shows the SaarCOR hardware architecture. The core ray tracing algorithm is contained in the dynamic ray tracing pipeline (DynRTP). RGS generates an initial ray R and transformation T. T is applied to the R in the transformation unit, and transformed ray is sent to the traversal unit. Then the traversal unit starts traversing the ray through the KD-tree until a leaf node is found. The ray with the list of objects is then forwarded to the mailbox unit. Mailbox unit can ommit objects that have already been isited by the same ray. Then the ray is sent to the transformation unit which maps the ray into object space. The ray is sent to the traversal unit to start the bottom-level traversal. From this method, recursive traversal can be executed. Finally the results are handed back to the RGS for shading.

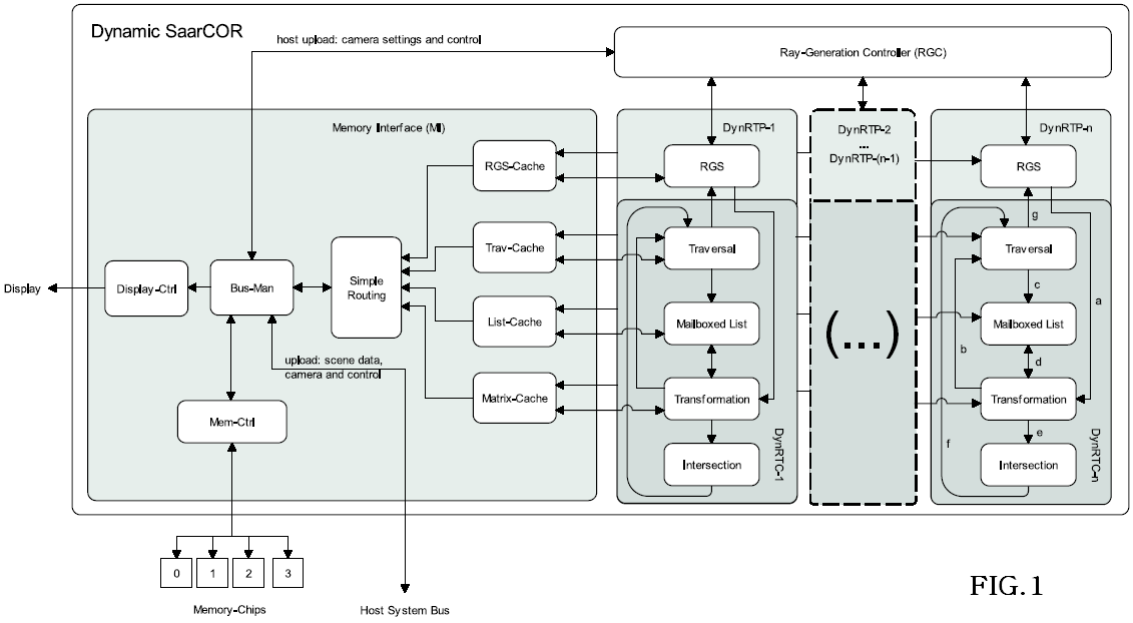



FIG. 1

The RPU is the next version of the SaarCOR. The major difference of this architecture is that it used programmable units for some operations. For example, the intersection test can be performed in shader as shown in figure 2. Ray transformation and intersection tests in ray tracing can be performed nicely by shader instructions.

Figure 3 shows the RPU architecture. SPU is the Shader Processing Unit, the TPU is the Traversal Processing Units, and the MPU is Mailboxed List Processing Unit. SPU is based on current GPUs, and used for above example, global lighting or vertex shading. TPU is dedicated hardware because traversal of a ray through a K-D tree typically requires 50 to 100 steps with scalar floating point operations. Using a fully

programmable vector unit for these operations wastes precious cycles. The TPU share a dedicated MPU. After traversal the SPU can perform intersection test.

```
 1 load4x A.y,0          ; load triangle
                         ; transformation
 2 dp3_rcp R7.z,I2,R3    ; transform ray dir to
 3 dp3 R7.y,I1,R3        ; unit triangle space
 4 dp3 R7.x,I0,R3
 5 dph3 R6.x,I0,R2       ; transform ray origin to
 6 dph3 R6.y,I1,R2       ; unit triangle space
 7 dph3 R6.z,I2,R2
 8 mul R8.z,-R6.z,S.z    ; compute hit distance d
   + if z <0 return      ; and exit if negative
 9 mad R8.xy,R8.z,R7,R6  ; compute barycentric
                         ; coordinates u and v
   + if or xy            ; and return if
       (<0 or >=1)       ; hit is outside
         return          ; the bounding square
10 add R8.w,R8.x,R8.y    ; compute u+v and test
   + if w >=1 return     ; against triangle diagonal
11 add R8.w,R8.z,-R4.z   ; terminate if last hit
   + if w >=0 return     ; distance in R4.z is
                         ; closer than the new one
12 mov SID,I3.x          ; set shader ID
   + mov MAX,R8.z        ; and update MAX value
13 mov R4.xyz,R8         ; overwrite old hit data
   + return              ; and return
```

FIG.2

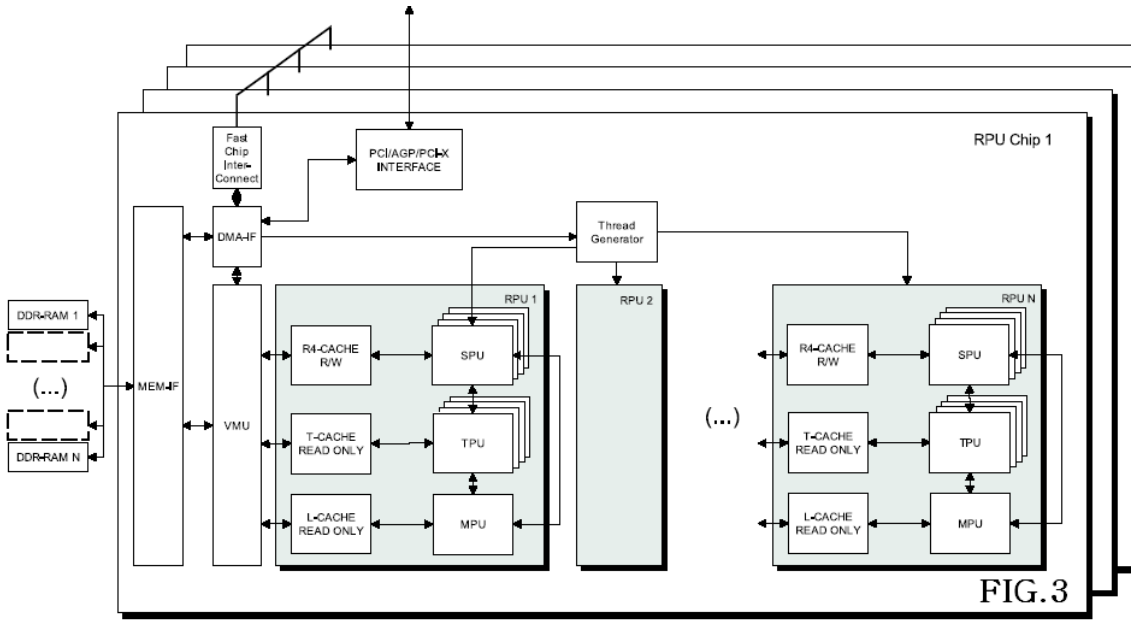



FIG.3

During the transformation, 4-D vector operation is needed. So, Each SPU operates on 4-component vectors as its basic data type. Also, to exploit the data parallelism, SPU and TPU is designed to be multi-threaded. Multi-threading allows to increase hardware utilization by filling instruction slots that would otherwise not be used due to instruction dependencies or memory latency.

[Purcell 2002] is based on the GPU hardware, but it is not well known current GPU but the 'stream processor'. Stream Processor is more programmable and more general purpose hardware than GPU. It is proposed in [Khailnay et al. 2000]. Stream processor has two new concepts which are the kernel and stream. Basically, stream is the input or output data of operation units, and kernel is program to process streams. Stream ray tracer can be split into four kernels: eye ray generation, traversal, ray-triangle intersection, and shading as shown in the figure 4.

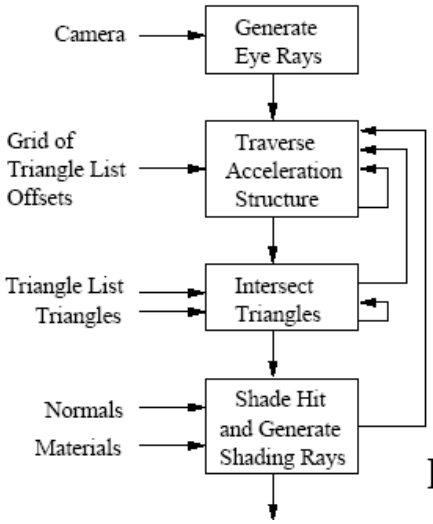


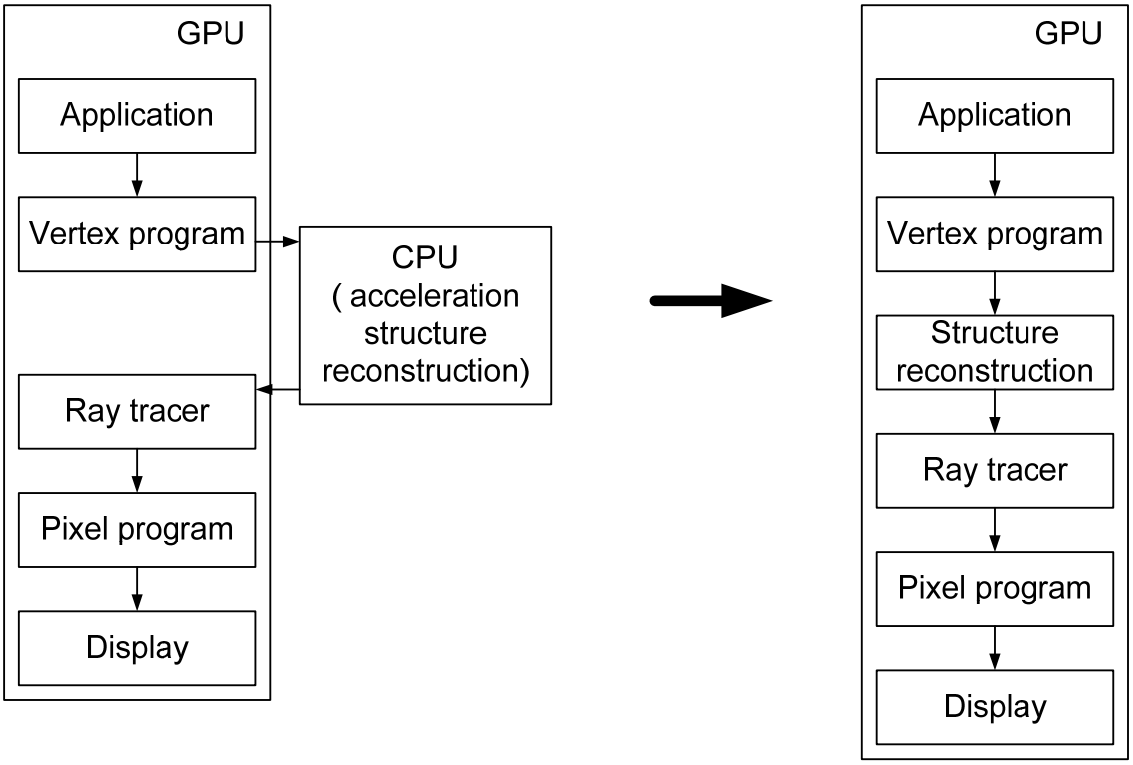

FIG. 4

The eye ray generator kernel produces a stream of viewing rays. The traversal kernel reads the stream of rays produced by the eye ray generator. The traversal kernel steps rays through the grid until the ray encounters a voxel containing triangles. The ray and voxel address are output and passed to the intersection kernel. The intersection kernel is responsible for testing ray intersections with all the triangles contained in the voxel. The intersector has two types of output. If ray-triangle intersection (hit) occurs in that voxel, the ray and the triangle that is hit is output for shading. If no hit occurs, the ray is passed back to the traversal kernel and the search for voxels containing triangles continues. The shading kernel computes a color. If a ray terminates at this hit, then the color is written to the accumulated image. Additionally, the shading kernel may generate shadow or secondary rays; in this case, these new rays are passed back to the traversal stage.

But, the [Purcell 2002] did not consider the cost of building data structure, so it is not be efficient for dynamic scenes.

Overview

There are two main issues in real-time ray tracing. One is the recursive traversal of acceleration structures. And the other is re-computation of acceleration structures for dynamic scene. The hardware-driven recursive traversal method is proposed in the recent works[Woop 2005],[Schmittler 2004],[Purcell 2002]. But the hardware support for building spatial index structures is not proposed yet. In the conventional graphics pipeline, ray-tracing is located after vertex shading. Without hardware support for structure reconstruction, CPU should receive the vertex data after vertex shading, and make data structure, and then give this to ray-tracing hardware. It is very inefficient




Therefore, my goal is to propose an overall ray tracing hardware architecture including update of acceleration structure.