Hardware-Driven Ray Tracing

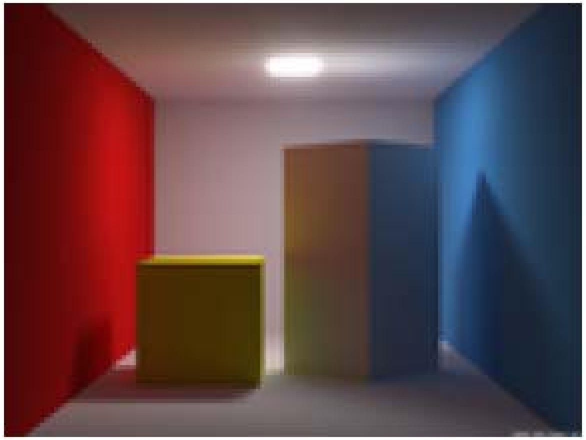
Jae-Sung Yoon

KAIST (Korea Advanced Institute of Science and Technology)

Contents

Introduction Ray Tracing

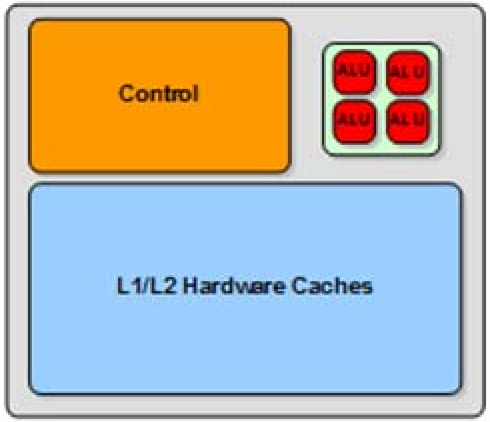
- **One of the most popular and simple way of doing global illumination.**



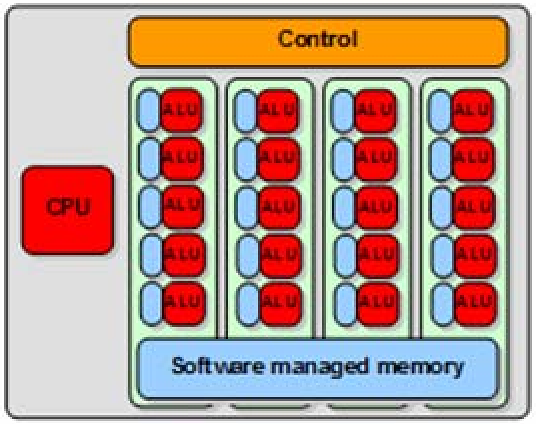
[Purcell 2004]

Why Ray Tracing Hardware?

- CPU can accelerate only single-threaded application
- CPU Cannot optimally exploit the inherent **parallelism in ray tracing**.

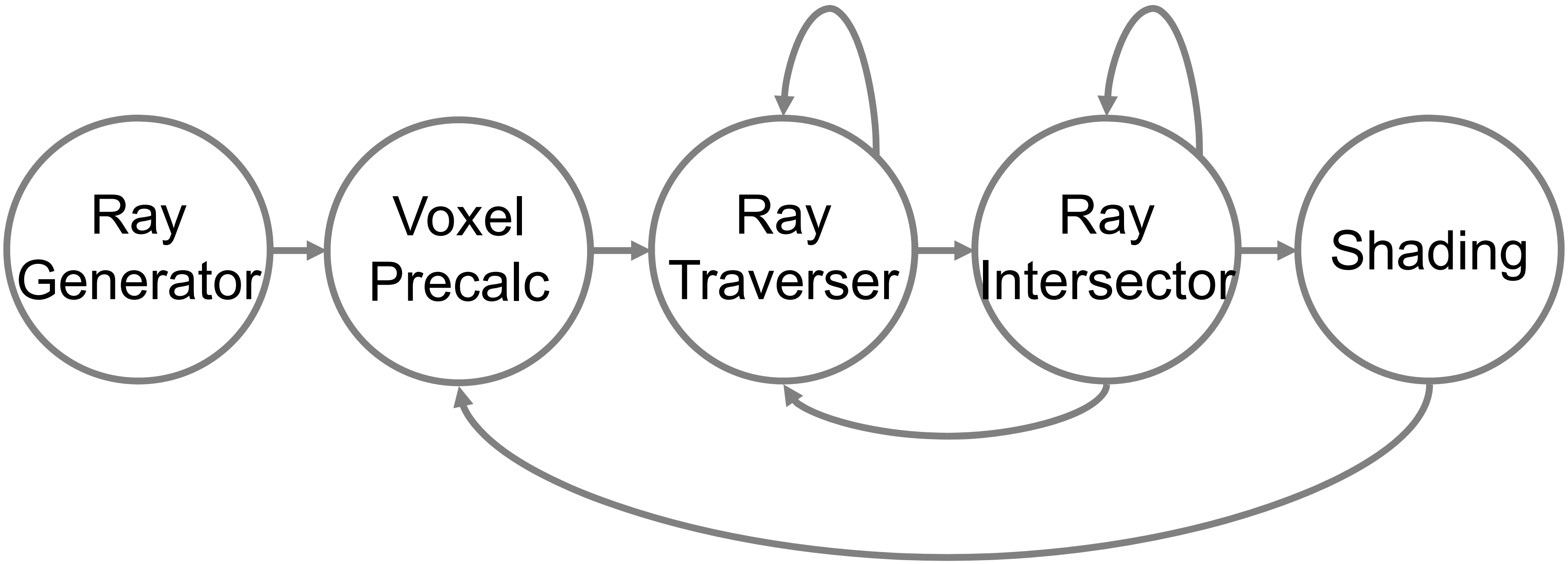


CPU or DSP



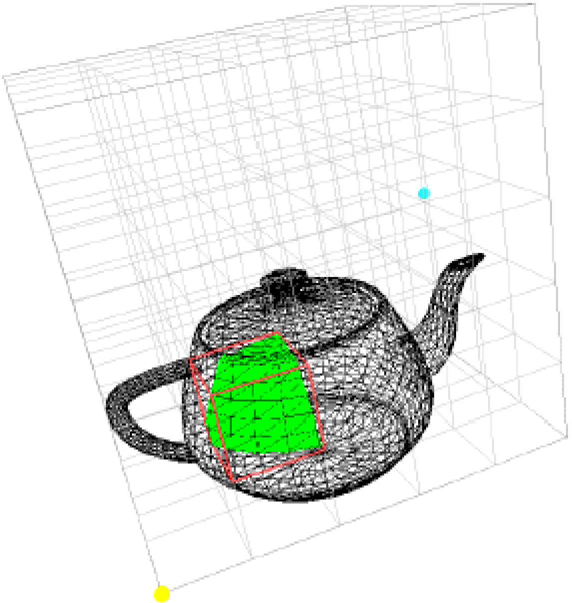
GPU or
Stream Processor

Basic Algorithm



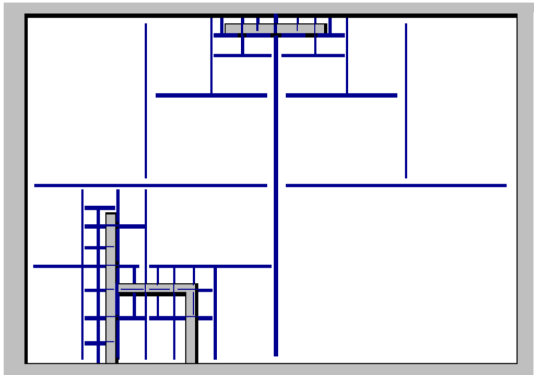
For BSP structure, Traversal is typically 2-3 times as costly as intersection test [Wald 2001]

Acceleration Structure (AS)

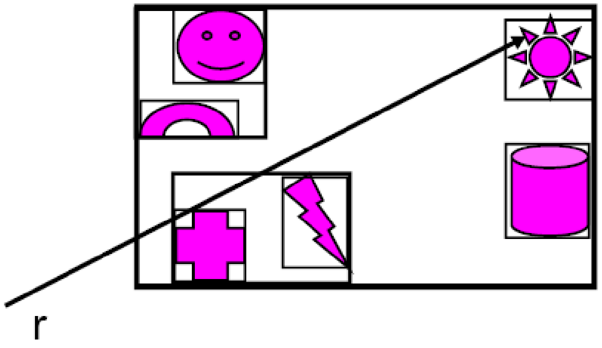


Uniform Grid

[Martin 2005]



KD-tree

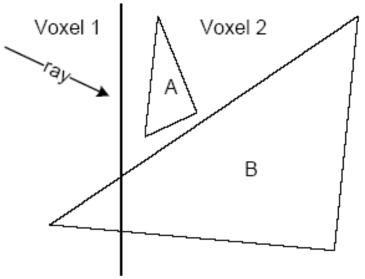


BVH

(bounding volume hierarchy)

Spatial Subdivision Schemes

- Uniform Grid , KD-tree
- Commonalities
  - 1. Multiple Reference
    - triangles can straddles a voxel boundary
  - 2. Can stop traversal at first intersection
    - visit the voxels in order of increasing distance from the ray origin
  - 3. Stack



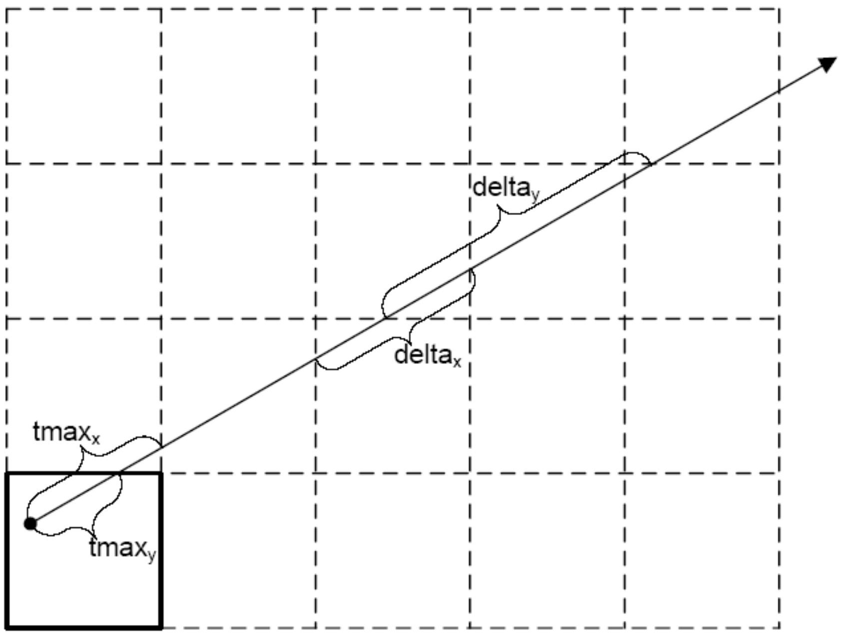
KAIST

1. Uniform Grid (Construction)

- **1. Decide the resolution of the grid**
  - **High resolution : fewer triangle intersection**
  - **Low resolution : fewer voxel traversal**
  - **Triangle / voxel = 20 works well [Woo]**
  - $3\sqrt[3]{N}$ **voxels alog the longest axis [Pharr]**
- **2. Add a reference to the triangle in each voxel**

1. Uniform Grid (Traversal)

- **3D DDA(digital differential analyzer) algorithm [Amanatides & Woo]**



if $tmax_x < tmax_y$ then
  $X \leftarrow X + step_x$
  $tmax_x \leftarrow tmax_x + delta_x$
else
  $Y \leftarrow Y + step_y$
  $tmax_y \leftarrow tmax_y + delta_y$

2. KD-tree (Construction)

- Top-down in a recursive fashion
- Choose an axis-perpendicular splitting plane
- If a primitive straddles the split plane then both child get a reference

2. KD-tree (Construction)

- **Where to place the splitting plane?**
- **Surface area heuristic [MacDonald & Booth 1990]**

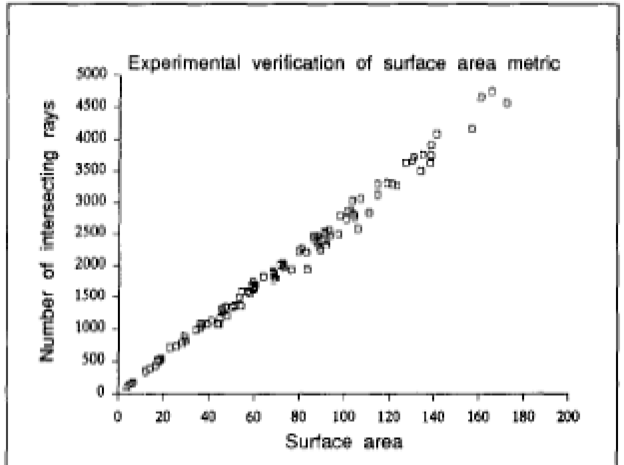
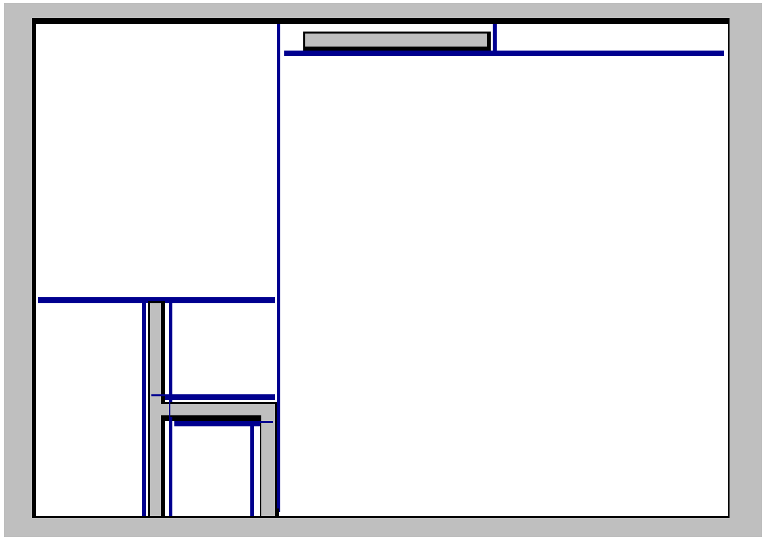


Fig. 1. Surface area heuristic data

[MacDonald 1990]



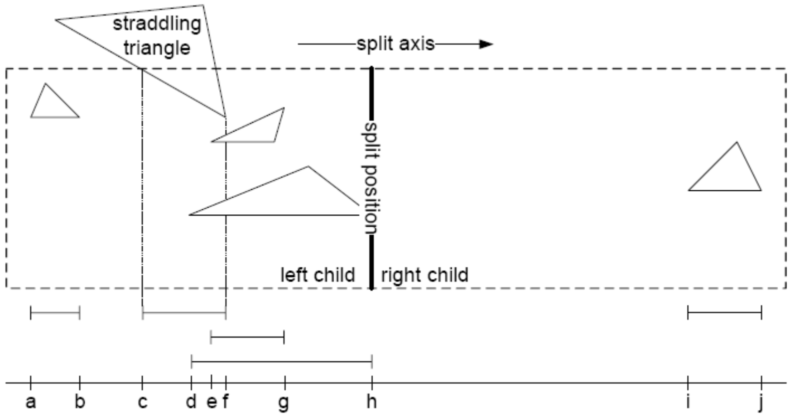
[Wald 04]

2. KD-tree (Construction)

$$= \frac{C_i \cdot \sum_{i=1}^{N_i} SA(i) + C_l \cdot \sum_{l=1}^{N_l} SA(l) + C_o \cdot \sum_{l=1}^{N_l} SA(l) \cdot N(l)}{SA(root)}$$

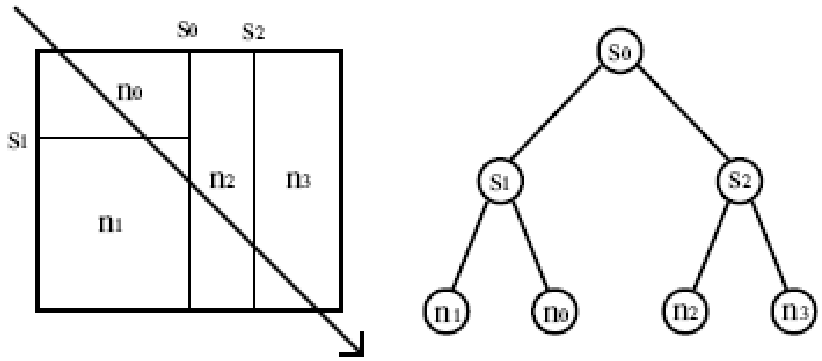
C_i = cost of traversing an interior node
C_l = cost of traversing a leaf
C_o = cost of testing an object for intersection

N_i = no. of interior nodes
N_l = no. of leaves
$N(l)$ = no. of objects stored in leaf l
$SA(i)$ = surface area of interior node i
$SA(l)$ = surface area of leaf node l



2. KD-tree (Traversal)

- **Given node N**



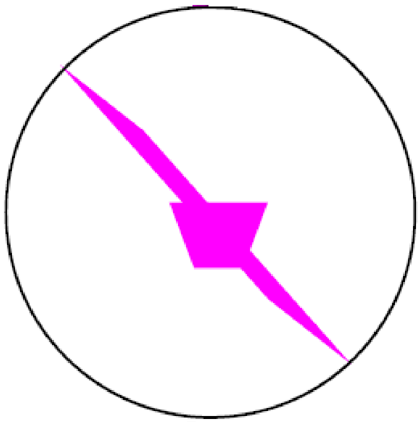
- **If N is leaf, intersection test for triangles**
- **If N is internal, determine which child node is first hit by ray**
- **Stack is needed (stack per ray)**

But, GPU don't have Stack

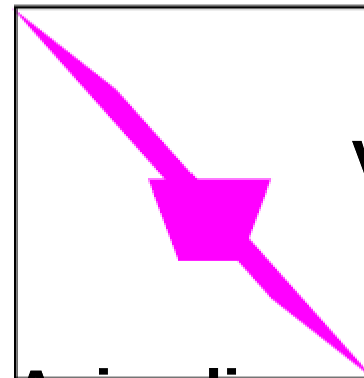
3. BVH (Construction)

- There is no known algorithm for constructing optimal BVH
- Cost function - surface area heuristic [Goldsmith & Salmon]
- AABB is most widely used.
- There are two major paper for BVH construction
  - [Kay & Kajiya] top-down
  - [Goldsmith & Salmon] bottom-up

3. BVH (Construction)

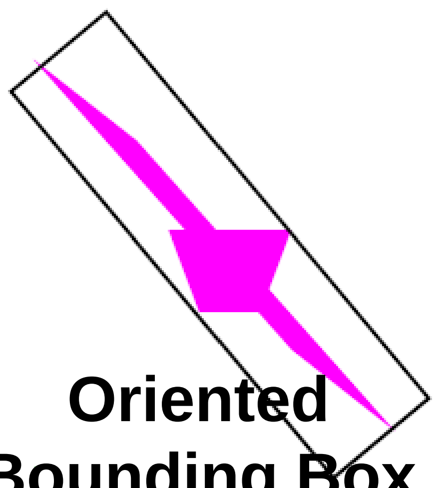


Sphere

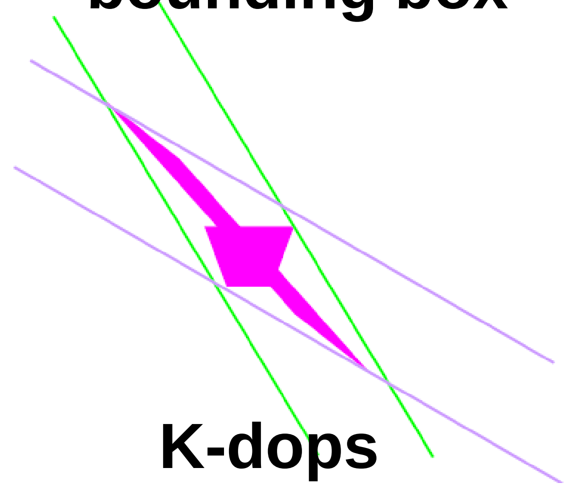


Axis-aligned bounding box

Very cheap to compute



Oriented Bounding Box



K-dops

3. BVH (Construction) Kay/Kajiya

```
BVNODE BuildTree(triangles)

if we were passed just one triangle then
  return leaf holding the triangle
else
  Calculate best splitting axis and where to split it
  BVNODE result
  result.leftChild ← BuildTree(triangles left of split)
  result.rightChild ← BuildTree(triangles right of split)
  result.boundingBox ← bounding box of all given triangles
  return result
end if
```

3. BVH (Construction) Goldsmith/Salmon

- Assign first triangle as the root
- For other objects, best position in the tree is found by cost function

3. BVH (Traversal)

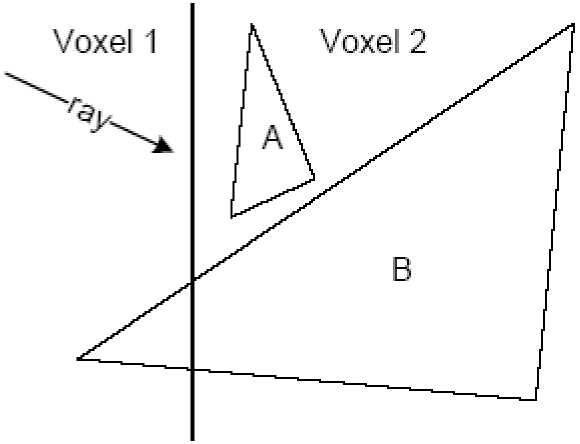
- Unlike the kd-tree, all the children have to be visited
- Stack is needed
- Main issue : order in which a node's children are traversed

Techniques for Acceleration & Hardware

- 1. Mailbox [Amanatides & Woo]
- 2. Stackless KD-tree traversal [Foley 2005]
- 3. Distributed Interactive Ray Tracing of Dynamic Scenes [Wald 2003]

1.Mailbox (spatial structure)

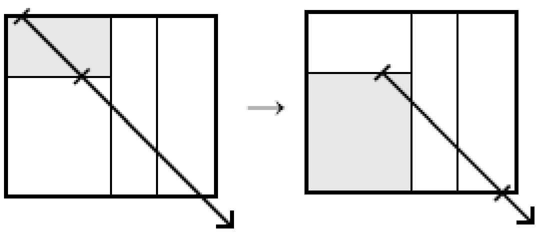
- **Problem**
  - **If triangle is split into two child voxel, each voxel has reference to the original triangle**
  - **We could test intersection against the same ray more than once.**



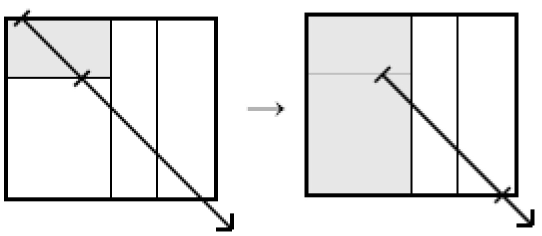
- **Solution**
  - **store the ID of ray with the triangle to skip the test the second time**

2. Stackless KD-tree Traversal

- KD-Restart
  - If leaf intersection fail, update (tmin, tmax)
  - Then, restart the search at the root



- KD-Backtrack
  - Store the parent link in the nodes
  - If leaf intersection fail, update (tmin, tmax)
  - Then, move up the tree to find parent

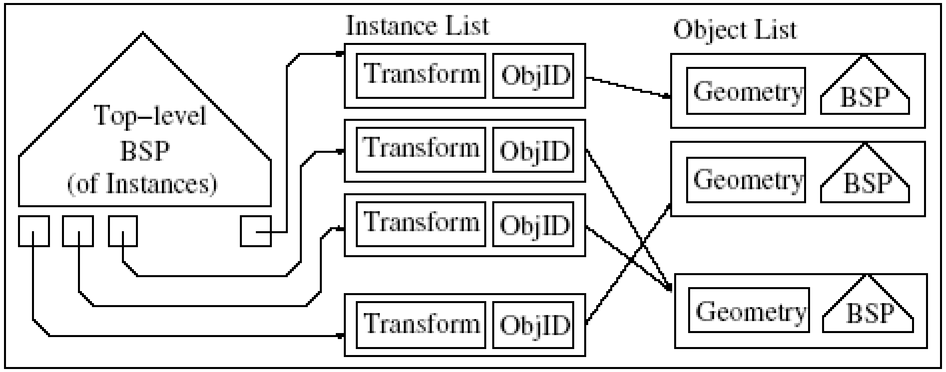


3. Distributed Interactive Ray Tracing of Dynamic Scenes

- **Motivation : In dynamic scene, large parts of a scene is static.**
- **Objects can be separated into three classes**
  - **Static objects**
  - **Objects undergoing affine transformation**
    - **$x \rightarrow Ax + B$**
  - **Objects with unstructured motion**

KAIST

- **For affine transformation,**
  - **(Ray – transformed Object) intersection == (inverse transformed Ray – Object) intersection**
- **Can remove the reconstruction cost for transforming objects**
- **Only top-level structure has to be rebuilt**



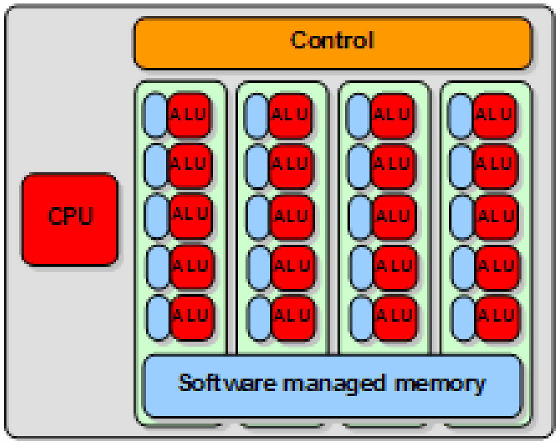
Two-level BSP

Hardware Implementations

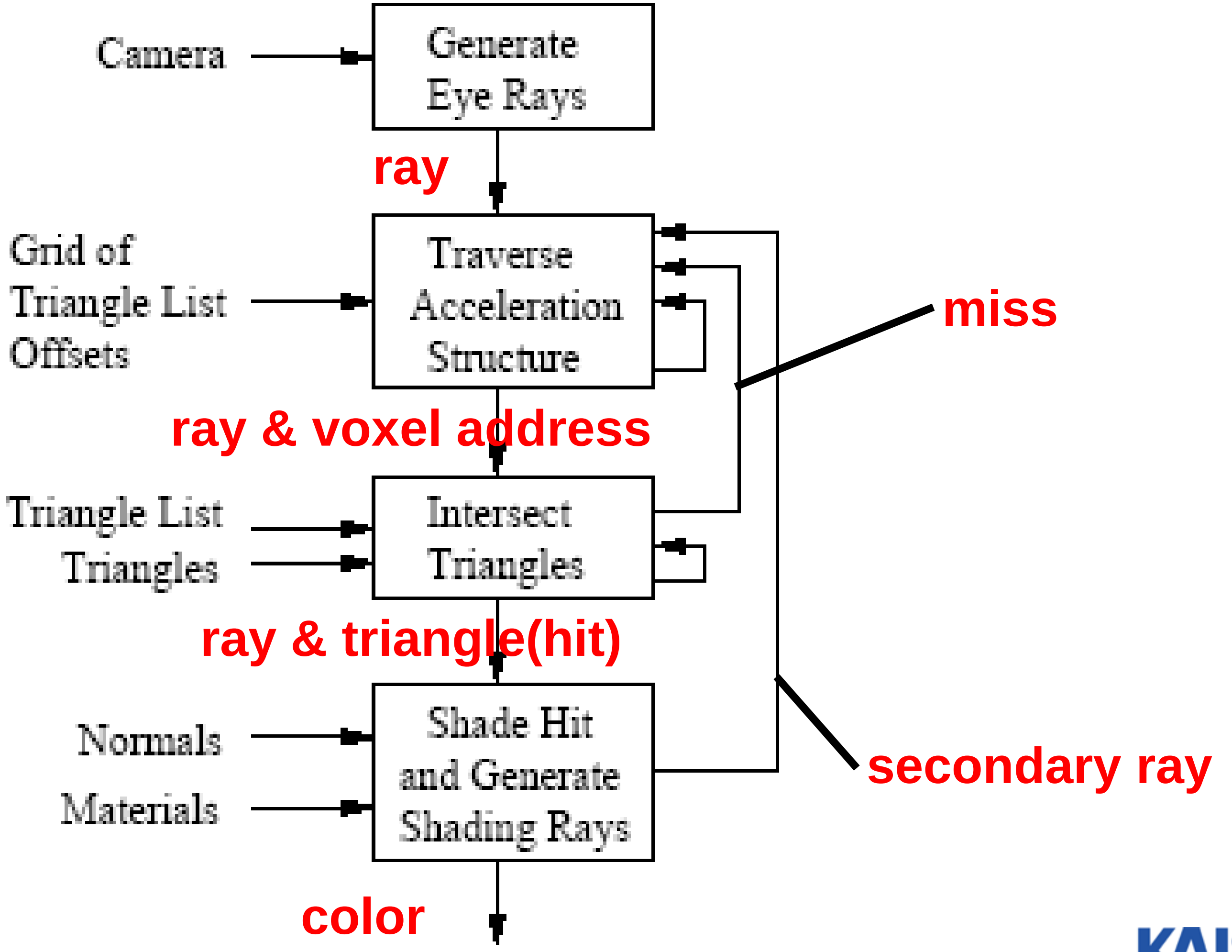
- 1. Streaming Ray Tracing [Purcell 2002]
- 2. Static SaarCOR [Schmittler 2002]
- 2. Dynamic SaarCOR [Schmittler 2004]
  - Upgrade version of Static SaarCOR
- 3. RPU [Woop 2005]
  - Programmable version of SaarCOR
- 4. Ray Tracing in FPGA [Lee 2007]

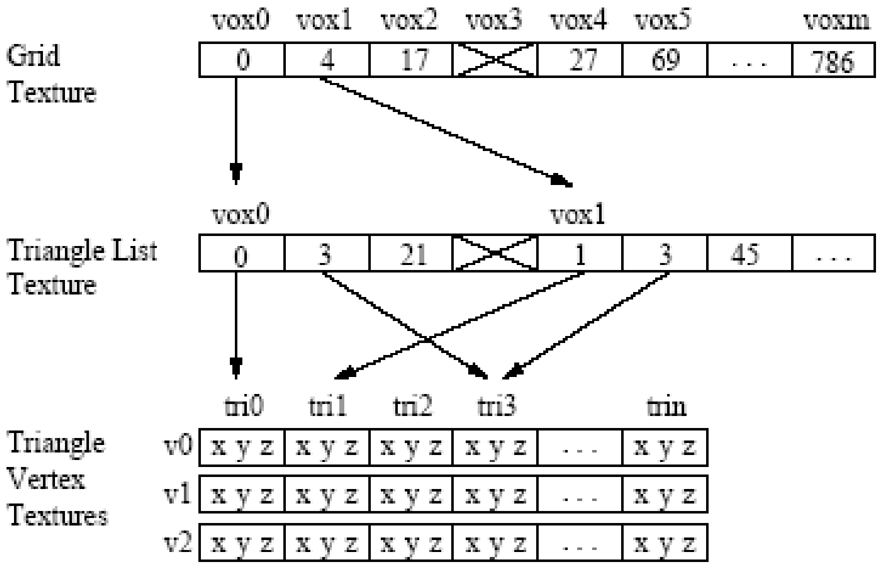
1. Streaming Ray Tracing [Purcell 2002]

- Used a streaming processor
- All geometry is stored in texture memory before rendering
- Used a uniform grid acceleration structure
- Used static scene

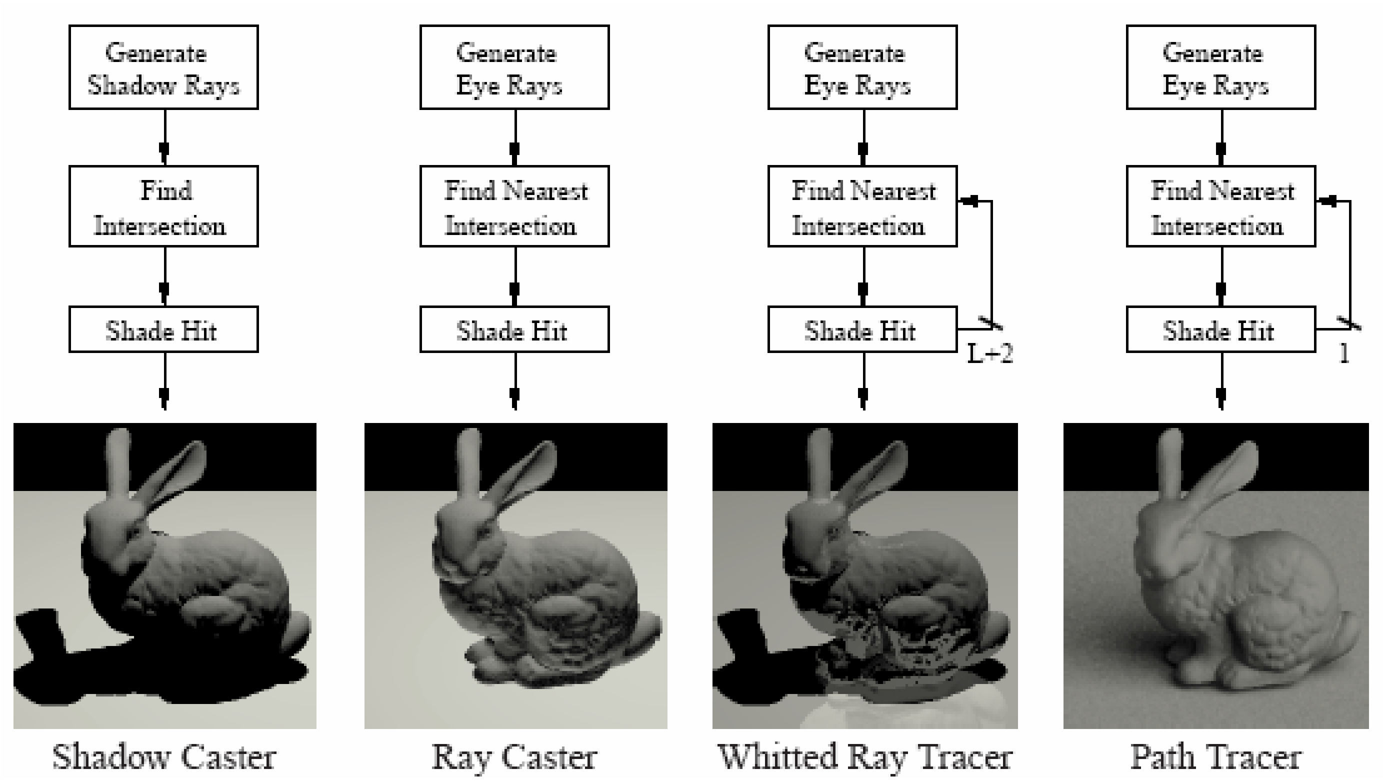


Stream Processor


Camera
Generate Eye Rays
ray
Grid of Triangle List Offsets
Traverse Acceleration Structure
miss
ray & voxel address
Triangle List
Triangles
Intersect Triangles
ray & triangle(hit)
Normals
Materials
Shade Hit and Generate Shading Rays
secondary ray
color



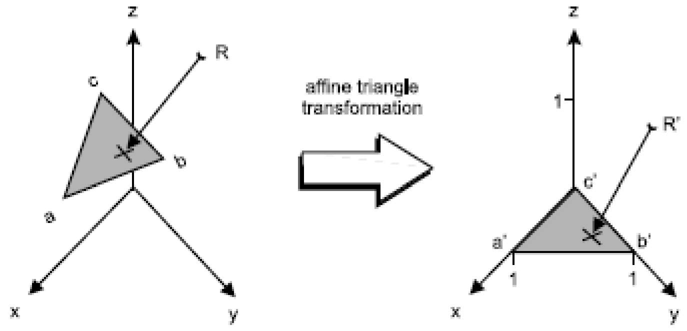
- **Grid list contains a pointer to a list of triangles**
- **Triangle list contains a pointers to vertex data**



Generate Shadow Rays
Find Intersection
Shade Hit
Generate Eye Rays
Find Nearest Intersection
Shade Hit
Generate Eye Rays
Find Nearest Intersection
Shade Hit
L+2
Generate Eye Rays
Find Nearest Intersection
Shade Hit
1
Shadow Caster
Ray Caster
Whitted Ray Tracer
Path Tracer

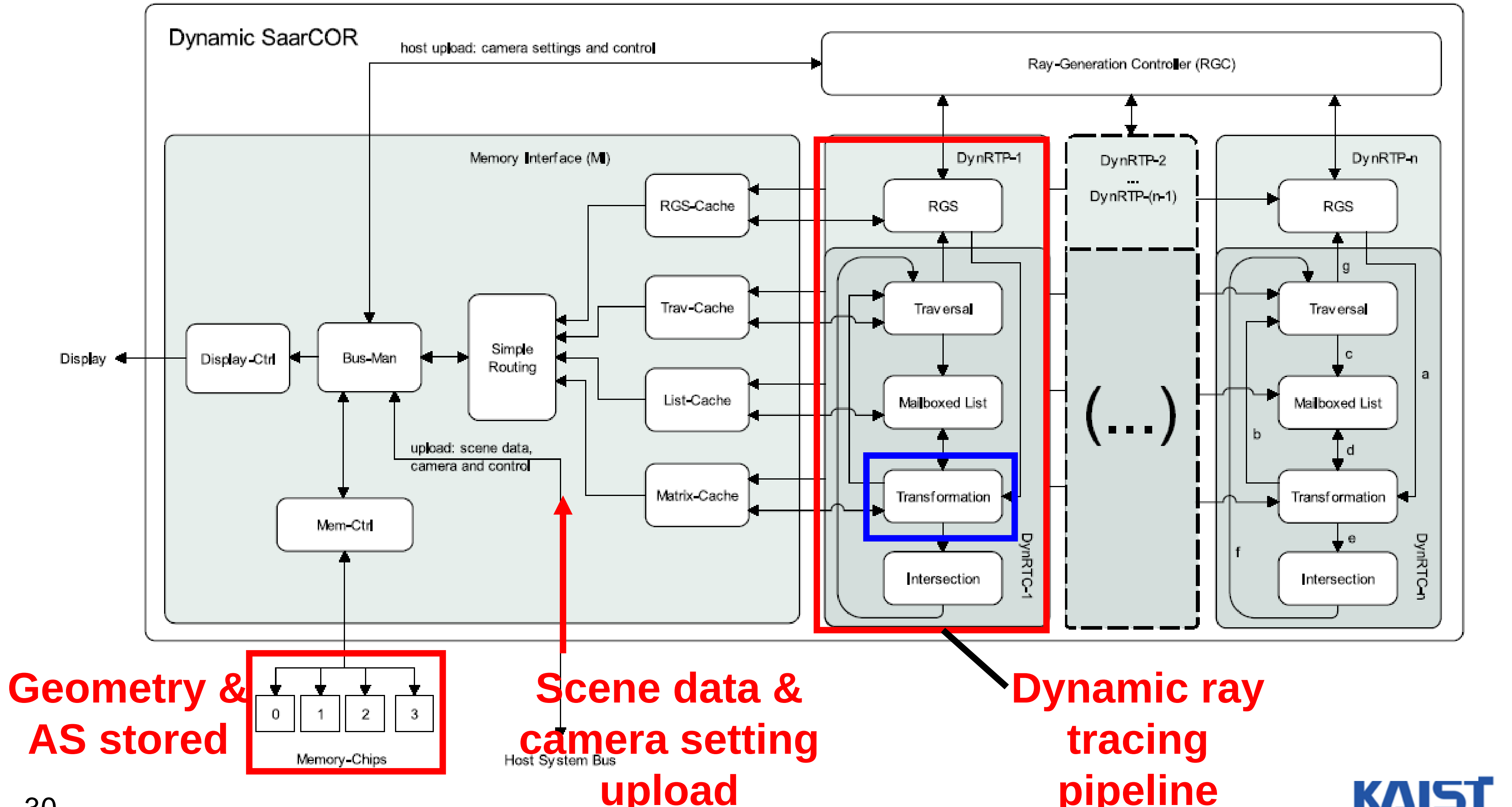
2. Dynamic SaarCOR[Schmittler 2004]

- Special purpose hardware for ray tracing
- Concerned about AS reconstruction using [Wald 2003] method
- Using cache[Schmittler 2003], load scene data from host
- Used a KD-tree
- Used unit triangle intersection method

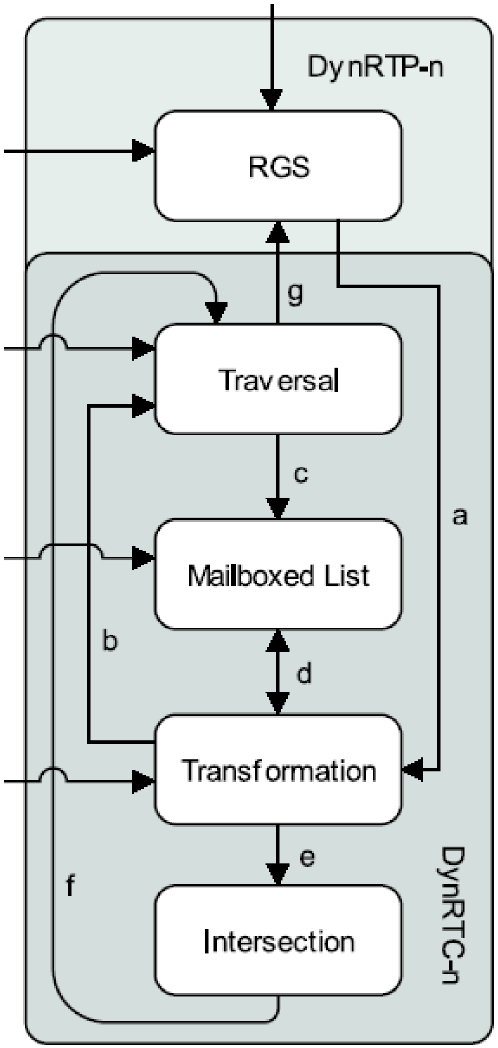


(a) world coordinate space (b) unit triangle space

Architecture

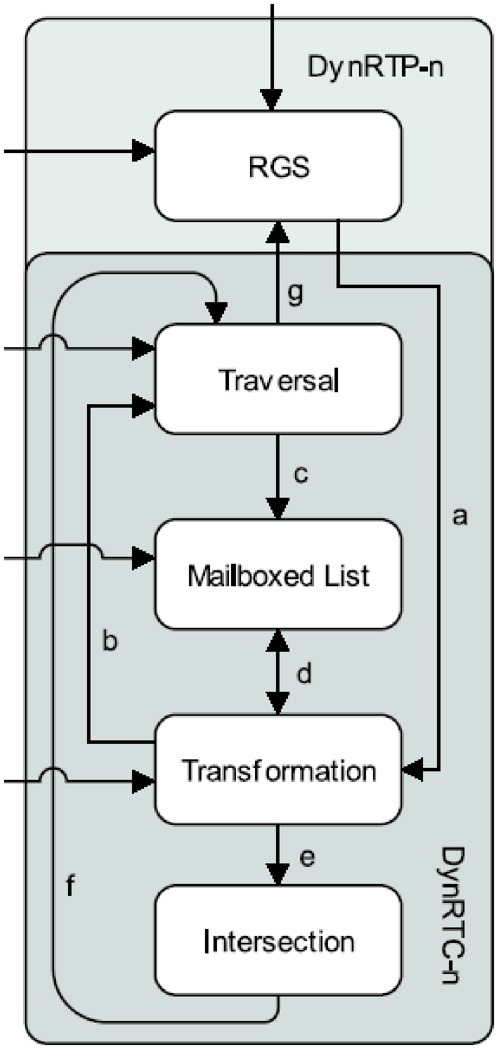


Operation Details



1. **RGS** : ray generation
2. **a**: ray & T(matrix) are sent
3. **Transformation**
4. **b & Traversal** : top-level traversal
5. **c**: ray & objects are sent
6. **Mailbox**
7. **d & Transformation** : ray transform into object space
8. recursive traversal

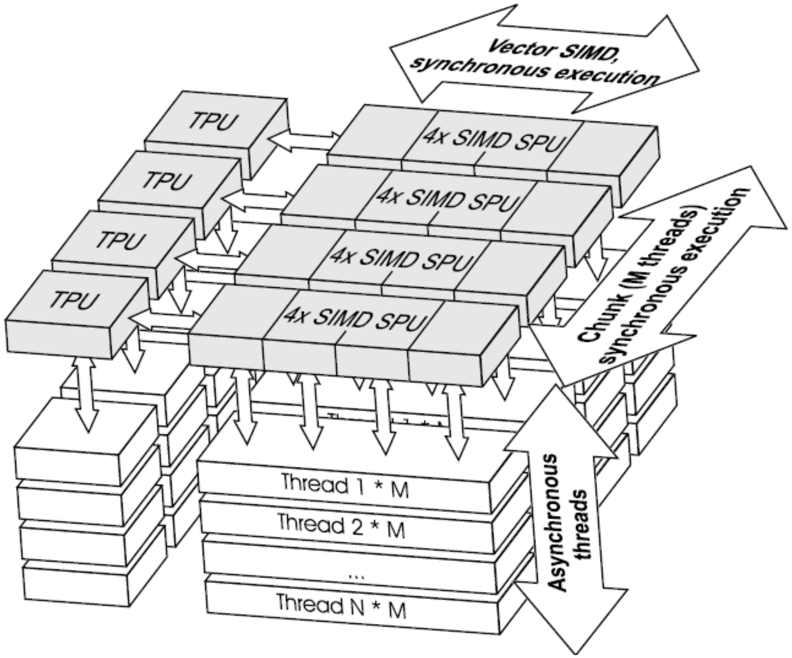
Operation Details



9. **Transformation : ray transform to unit triangle space**
10. **e & Intersection**
11. **f & recursive traversal**
12. **g: final result sent**
13. **RGS : shading**

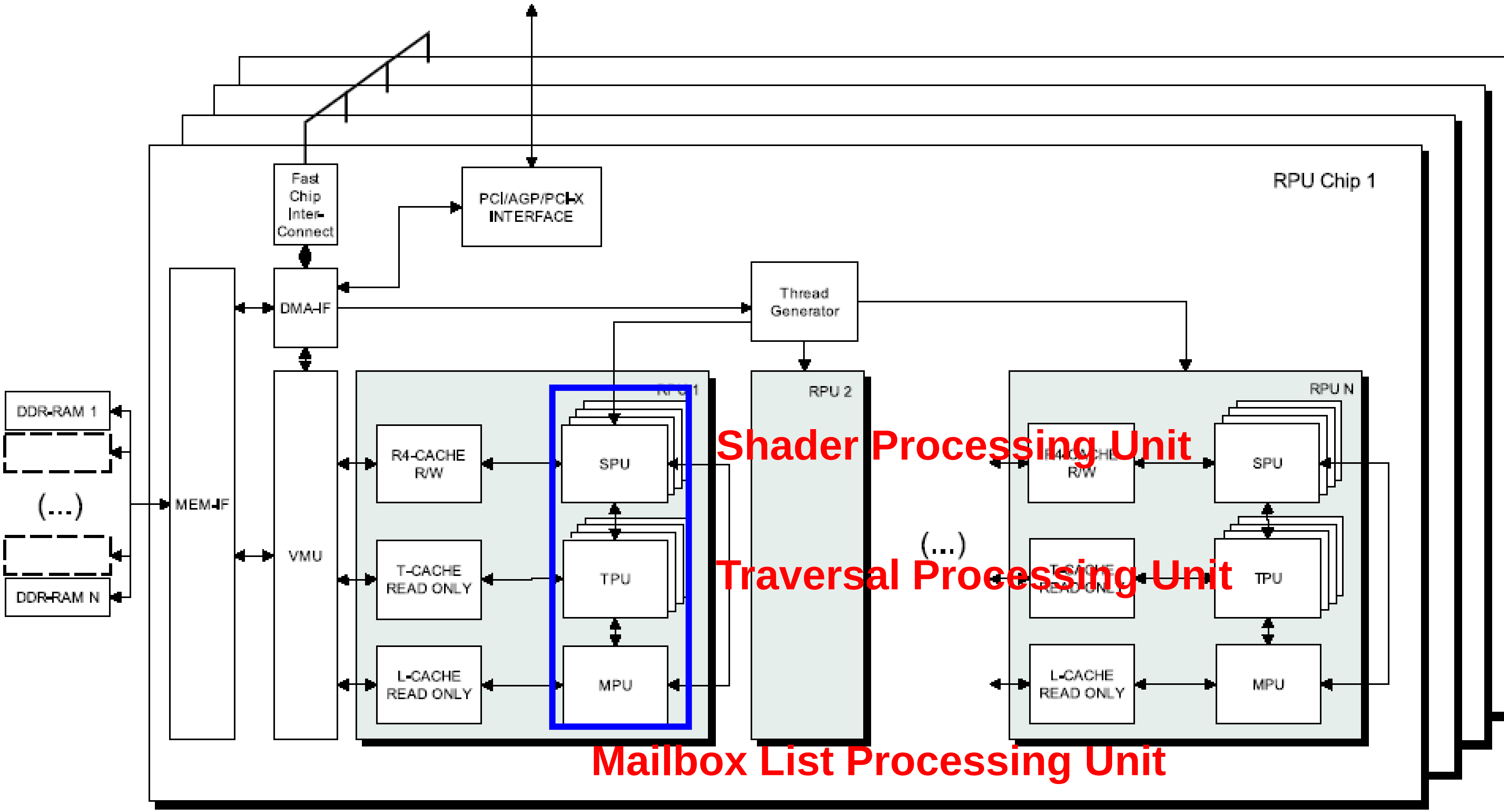
3. RPU [Woop 2005]

- Programmable hardware for ray tracing
- Traversal needs dedicated unit
- Same algorithm with Dynamic SaarCOR
- Upgrade version of Dynamic SaarCOR



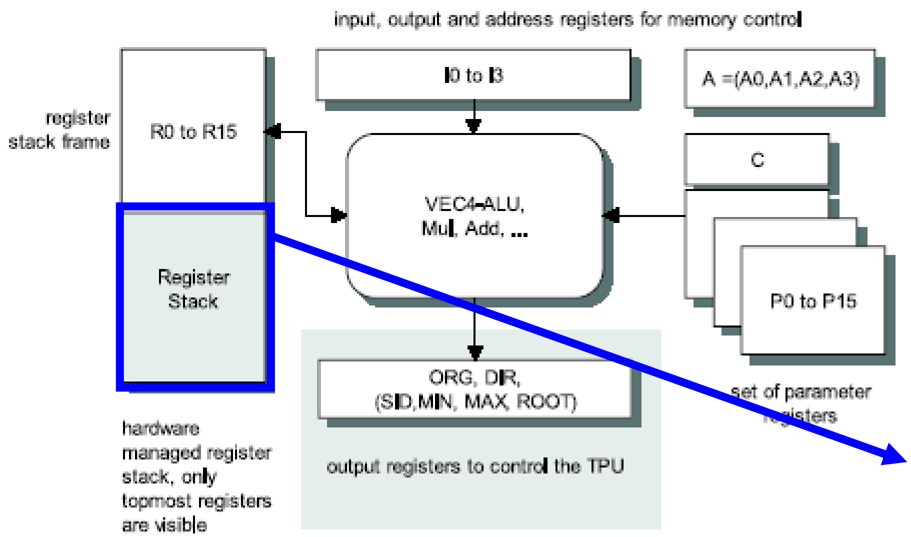
4D-vector dual-issue system

Architecture



Dedicated Traversal Unit

- **K-D tree traversal is very expensive scalar floating point operation.**
- **So, programmable hardware is inefficient**
- **Stack is in the SPU register**



Registers in SPU

→ **SPU is quite similar with current GPU**

Communicate with main memory because on-chip stack can be not sufficient

Intersection Test Program

Used unit triangle intersection method

Program quite fit into GPU

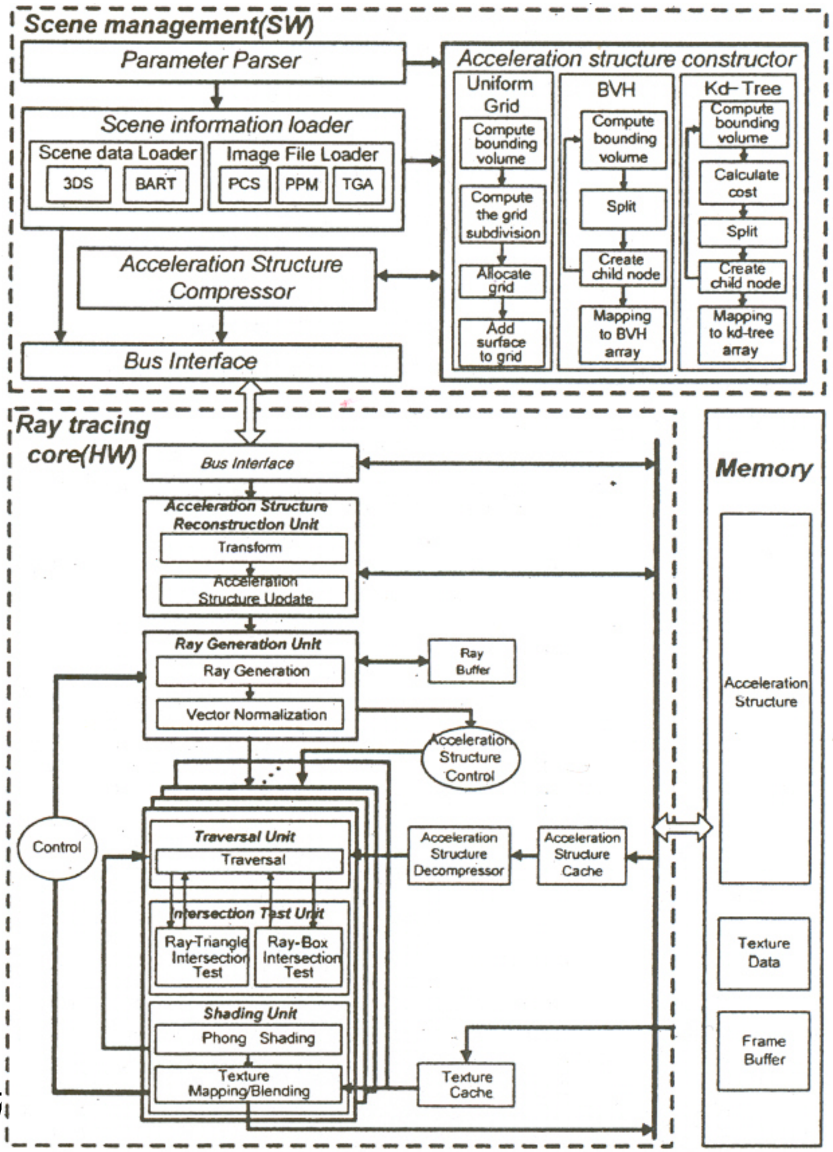
```
 1 load4x A.y,0          ; load triangle
                         ; transformation
 2 dp3_rcp R7.z,I2,R3    ; transform ray dir to
 3 dp3 R7.y,I1,R3        ; unit triangle space
 4 dp3 R7.x,I0,R3
 5 dph3 R6.x,I0,R2       ; transform ray origin to
 6 dph3 R6.y,I1,R2       ; unit triangle space
 7 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
```

Vector transformation

Scalar operations

Dual issue feature is efficient

4. Ray Tracing in FPGA [Lee 2007]



Software

Hardware

1. Dedicated Hardware
2. Used KD-backtrack