RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing

A Paper accepted for SIGGRAPH’05

by

Sven Woop
Jörg Schmittler
Phillip Slusallek

Presentation by Basil Fierz

Outline

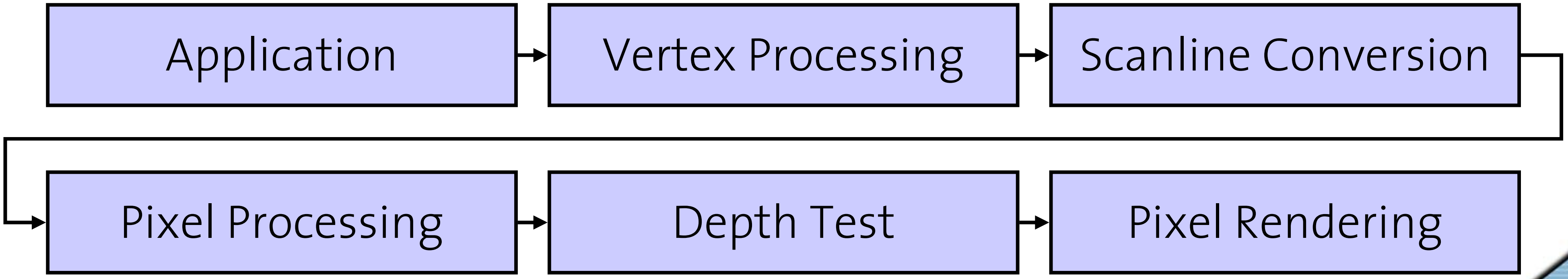
- Rasterizing & Raytracing
- Architecture
- Instruction set
- Implementation
- Results

Rasterizing vs. Raycasting

- Rasterizer Problem:
  - Given a set of rays and a primitive, efficiently compute the subset of rays that hit the primitive.
- Raycasting Problem:
  - Given a ray and a set of primitives, efficiently compute the subset of primitives that are hit by the ray

The Rasterizing Problem

- Set of rays defined by the pixels
- Primitives are triangles
- Triangles are processed independently
- Only local shading possible



Application

Vertex Processing

Scanline Conversion

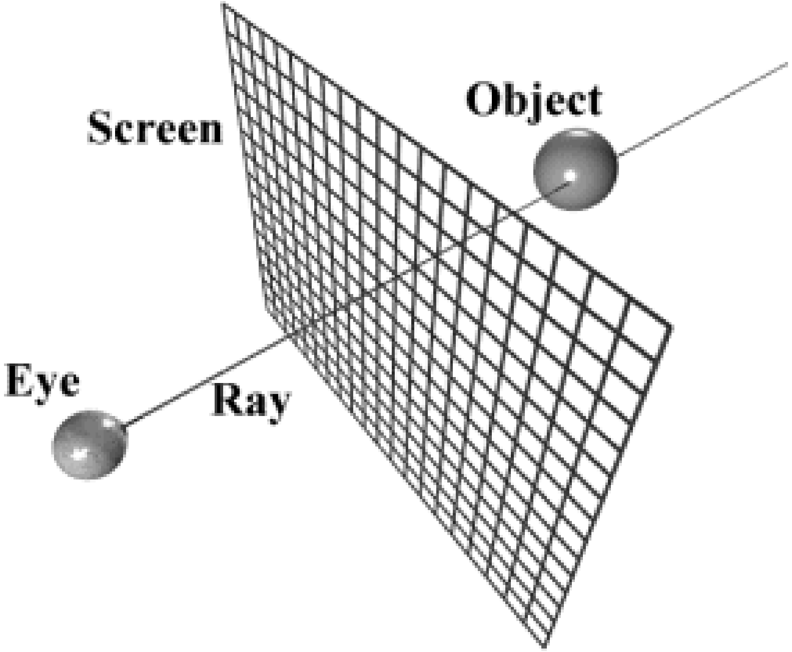
Pixel Processing

Depth Test

Pixel Rendering

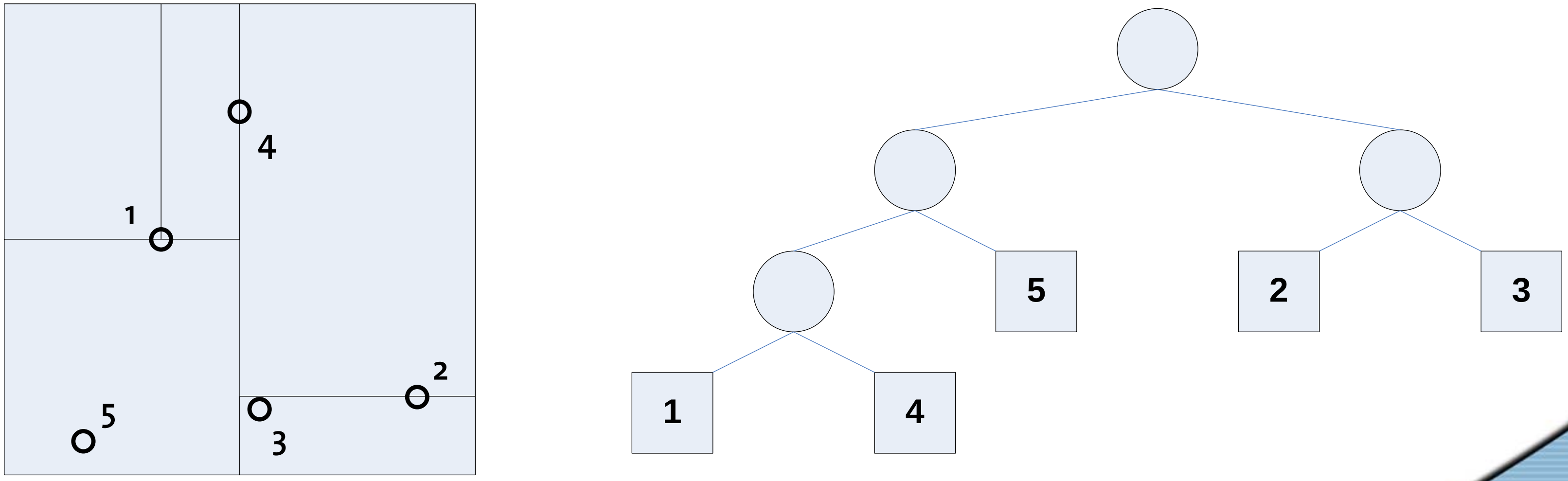
The Raycasting Problem

- Per screen pixel a ray is sent through the scene
- For each ray the intersection with a primitive of the scene is calculated
- At each intersection
  - Shading can occur
  - New rays can be spawned



Hierarchical Index Structure

- Organize primitives to speed up intersection calculation
- E.g. kD-Tree. With k as dimension
  - Iteratively divides the list of primitives by the median of one coordinate in two sub lists connected by a node.



Raytracing Issues

- Many floating point computations
- Flexible control flow (recursion & branching)
- High memory bandwidth
- Unstructured memory accesses

Realtime Raytracing

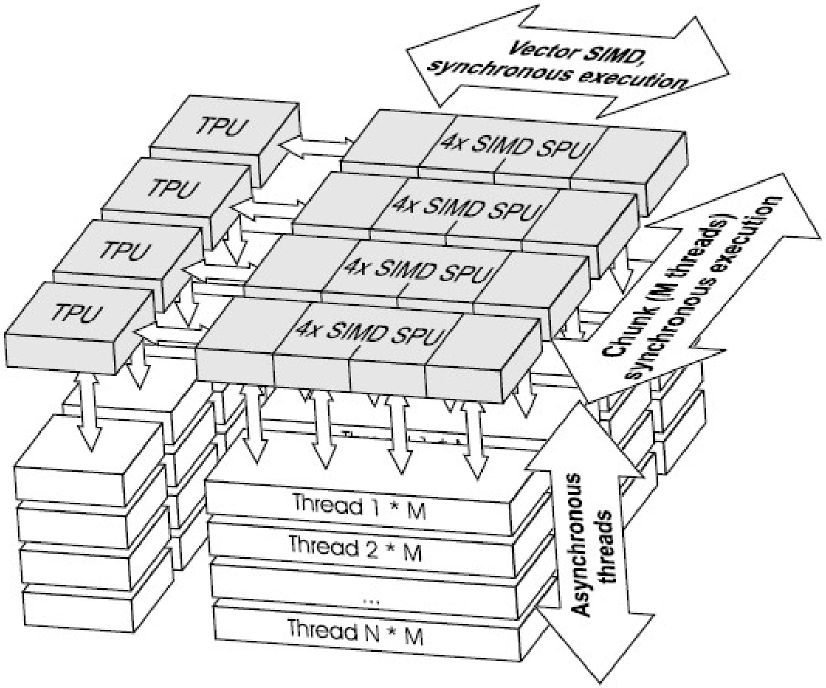
- Software implementation
  - OpenRT
- Fixed function ray tracing processor
  - SaarCOR [Schmittler 2004]

RPU: Design Decisions (1)

- Vector operations for instruction level parallelism
- Threads and Chunks for data level parallelism
- Tree Traversal Unit
- Scalability

RPU: Design Decisions (2)

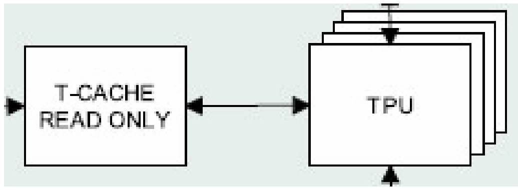
- Threads
  - New thread for every primary ray
  - Increase hardware utilisation



- Chunks
  - Exploit coherence between rays
  - Synchronously executed

RPU: Architecture Details (1)

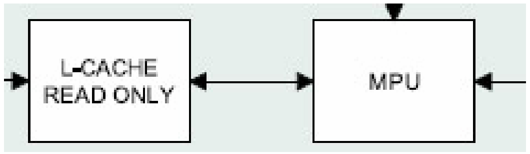
- Traversal Processing Unit (TPU)



  - Synchronously traverse the entire chunk through the kD-tree
  - At each node chunks can be split into sub-chunks

RPU: Architecture Details (2)

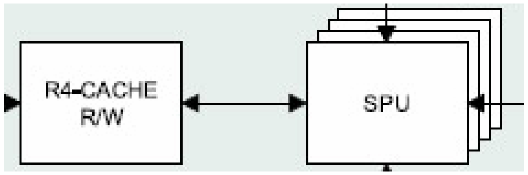
- Mailboxed List Processing Unit (MPU)



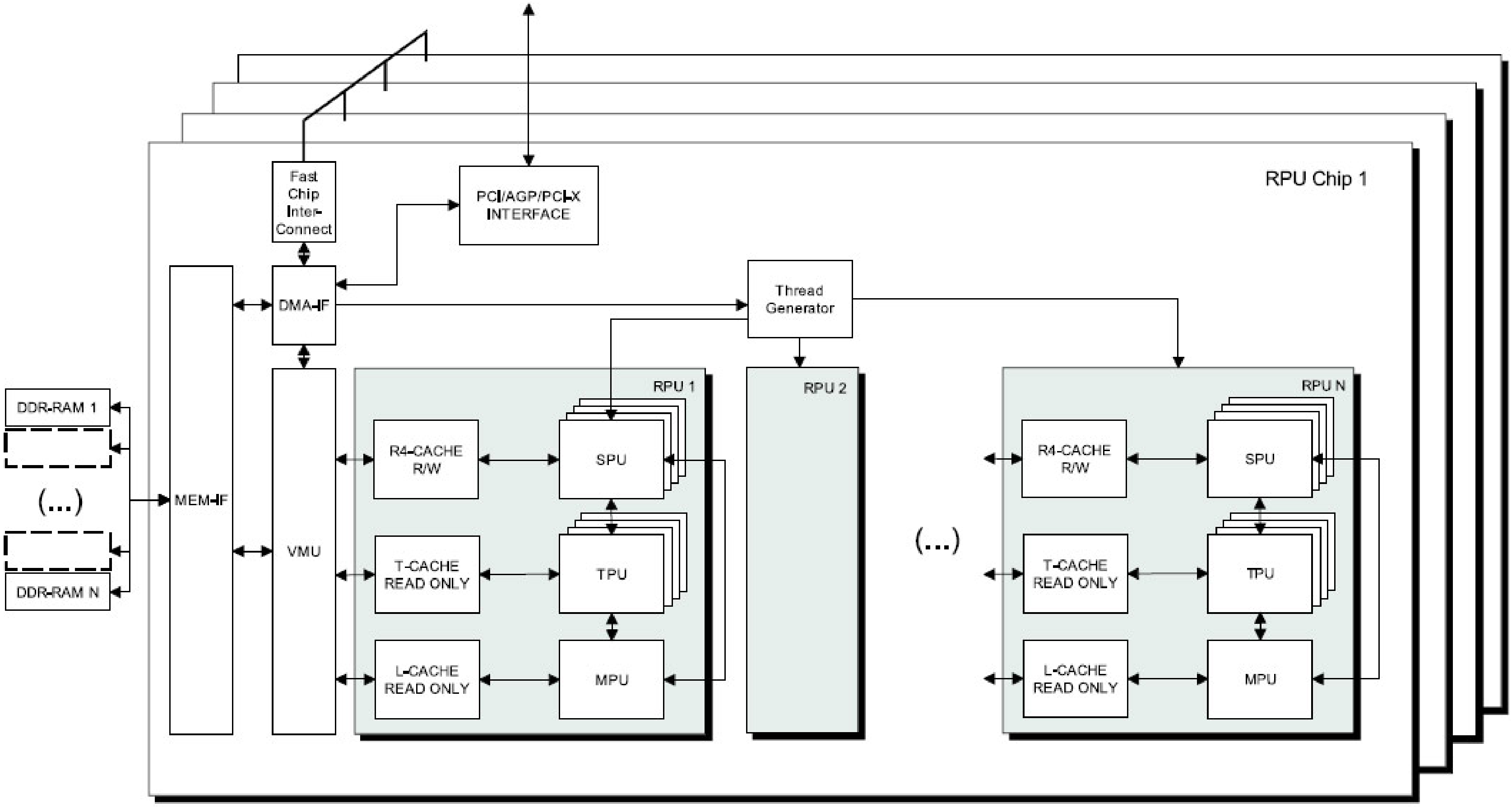
  - Called by the TPU at each non-empty leaf
  - For each list entry the corresponding threads are scheduled for the SPUs

RPU: Architecture Details (3)

- Shader Processing Unit (SPU)
  - SIMD Unit
  - Execute shaders
  - Vector splitting: 2/2, 3/1
  - Stack to support function calls



The Entire Architecture



Programming Model

- Ray centric
- Global shading (secondary rays, shadow rays)
- Procedural Geometry
  - Geometry shaders are called for each kD-tree entry encountered during the traversal of a ray
  - Allows direct ray tracing of e.g. bi-cubic splines
- Programmable Materials
  - Classical material shaders
  - Reflection and lighting can take global parameters into account

SPU Instruction Set (1)

- Inspired by the instruction set of current GPUs
  - Per component addition, multiplication
  - Dot products, integer computation
  - Memory reads and writes
  - 2D texture read and writes
  - Swizzling
  - read/write masking
  - clamping results to [0,1]
  - Function call and branching instructions

SPU Instruction Set (2)

- A special instruction ‘trace’ to spawn new rays (recursive ray tracing)
- Instruction pairing
  - No dynamic scheduling available
  - Two slots per instruction for static scheduling

Example Shader

```
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
```

Prototype Implementation (1)

- Xilinx Virtex-II FPGA
- 66 MHz clockspeed
- Four 16bit wide memory chips used as 64bit memory interface
- PCI interface for host communication

Prototype Implementation (2)

- Support for 32 hardware threads with four SPUs (Chunk size M = 4)
- Due to the size limitation
  - Integer operations were omitted
  - Shaders have a maximum length of 512 instructions
  - 24 bit floats

Results: Test settings

- Fully programmable **RPU** running at 66 MHz
- Fixed function **SaarCOR** scaled down to match the RPU
- 2.66 GHz Intel Pentium 4 running (**OpenRT**)
- 512x384 pixels image resolution
- Only primary rays

Results: Scene6

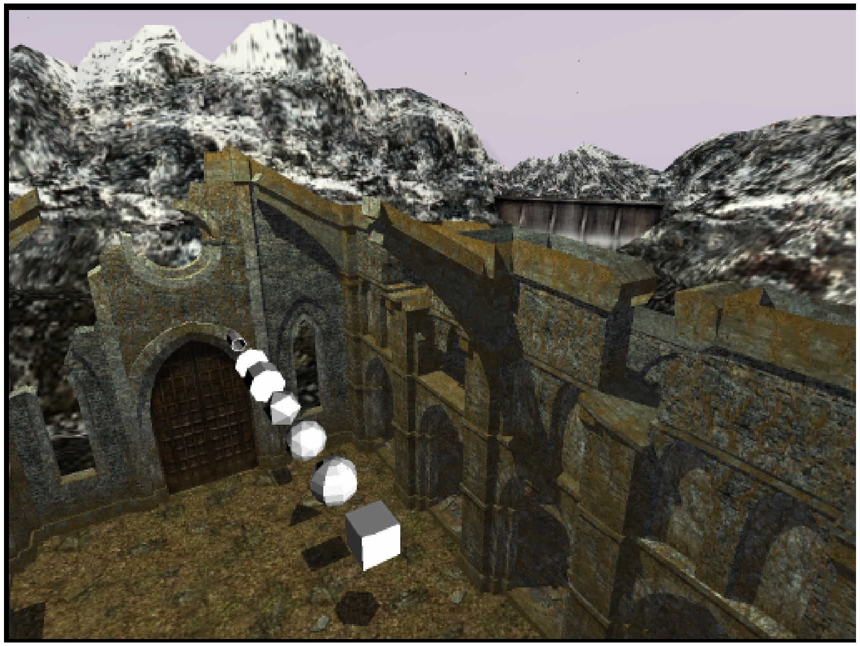
- Simple Scene
- Triangles: 806
- OpenRT: 12.9 fps
- SaarCOR: 44.6 fps
- RPU: 20.8 fps

Results: Quake 3

- Complex Scene
- Triangles: 52’790
- OpenRT: 7.9 fps
- SaarCOR: 19.6 fps
- RPU: 9.7 fps

Results: Castle

- Complex Scene
- Triangles: 20’891
- OpenRT: 9.2 fps
- SaarCOR: 17.5 fps
- RPU: 2.8 fps



Conclusions

- First programmable realtime raytracing unit
- Realtime ray tracing as a vision
- As powerful as software