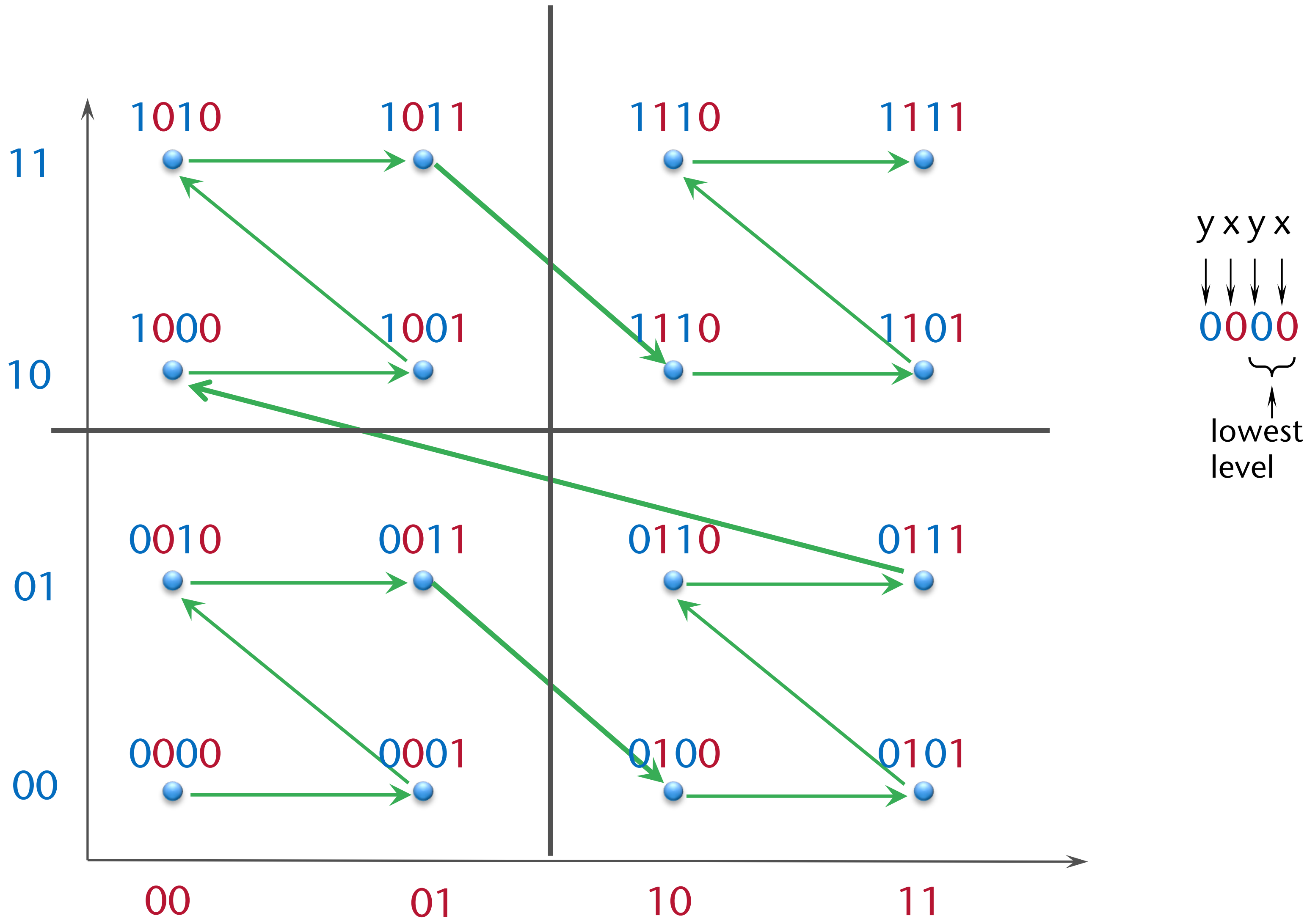
Construction of the Z-Order Curve in 3D

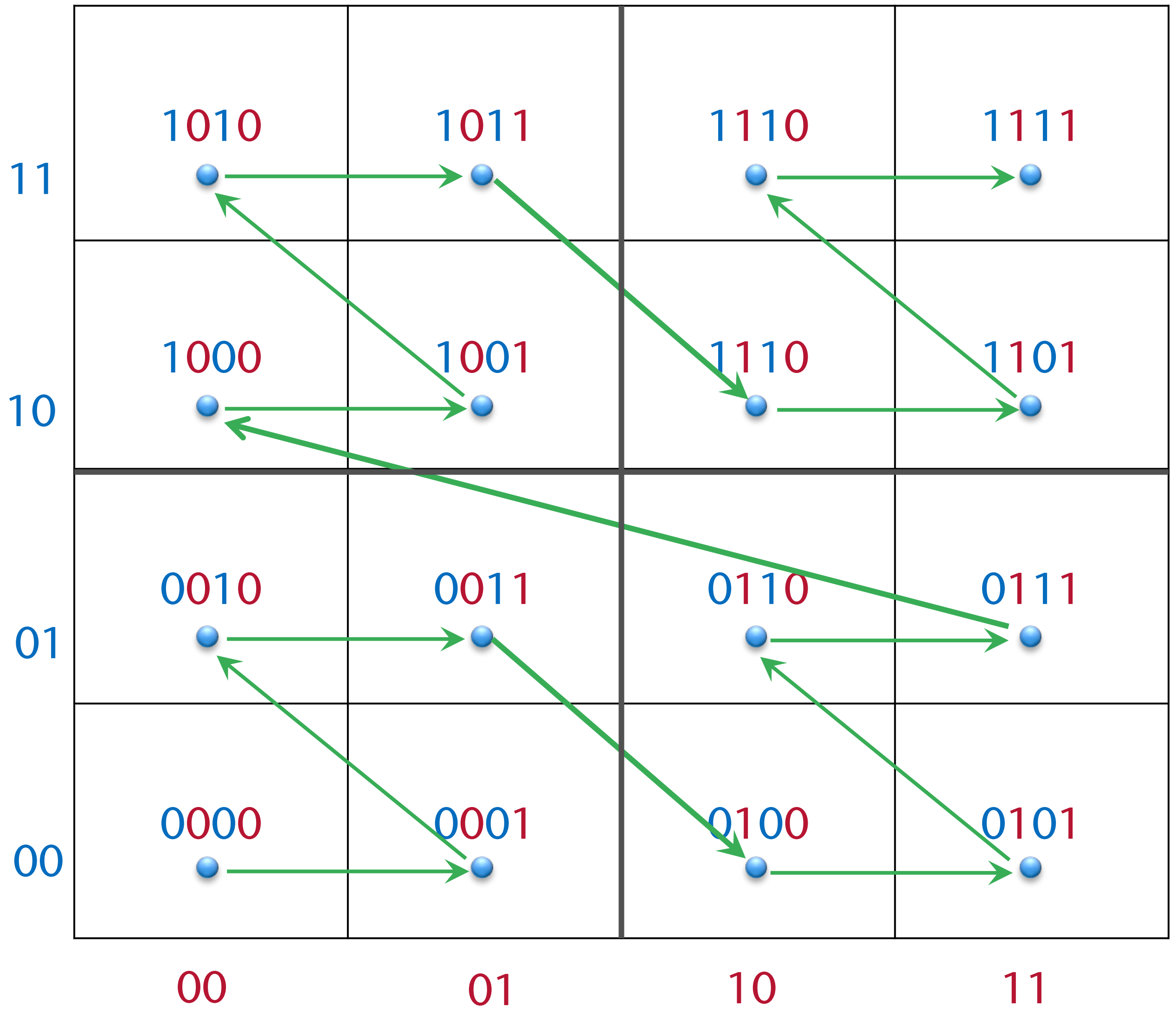
1. Choose a level k
2. Construct a regular lattice of points in the unit cube, 2^k points along each dimension
3. Represent the coordinates of a lattice point p by integer/binary number, i.e., k bits for each coordinate, $p_x = \mathrm{b}_{x,k} \ldots \mathrm{b}_{x,1}$
4. Define the Morton code of p as the interleaved bits of the coordinates, i.e., $m(p) = \mathrm{b}_{z,k}\mathrm{b}_{y,k}\mathrm{b}_{x,k} \ldots \mathrm{b}_{z,1}\mathrm{b}_{y,1}\mathrm{b}_{x,1}$
5. Connect the points in the order of their Morton codes $\longrightarrow$ z-order curve at level k

Example



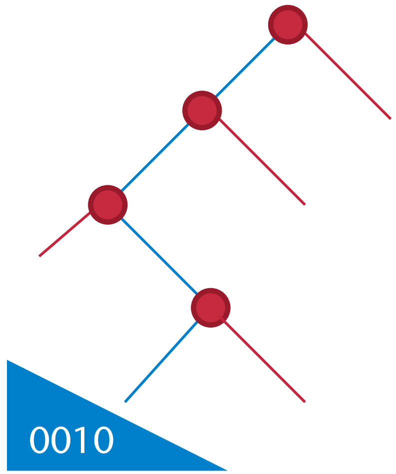
1010
1011
1110
1111
11
1000
1001
1110
1101
10
0010
0011
0110
0111
01
0000
0001
0100
0101
00
00
01
10
11
y x y x
0000
lowest level

Note: the Z-curve induces a grid (actually, a multi-grid)



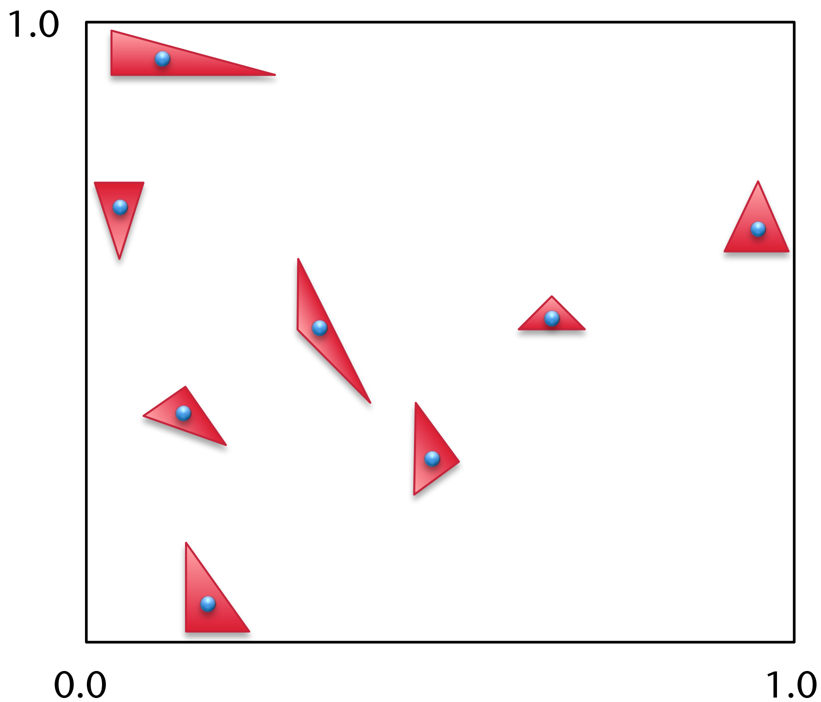
Properties of Morton Codes

- The Morton code of each point is 3*k* bits long
- All points *p* with Morton code *m*(*p*) = 0xxx lie below the plane z=1/2
- All points with *m*(*p*) = 111xxx lie in the upper right quadrant of the cube
- If we build a binary tree/quadtree/octree on top of the grid, then the Morton code encodes the *path* of a point, from the root to the leaf that contains the point ("0" = left, "1" = right)
- The Morton codes of two points differ for the first time – when read from left to right – at bit position *h* ⇔ the paths in the binary tree over the grid split at level *h*

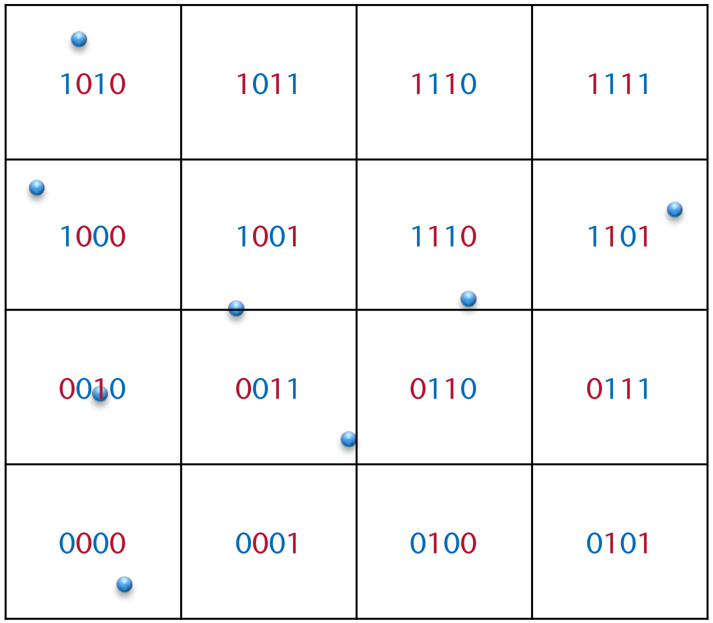


Construction of Linear BVHs

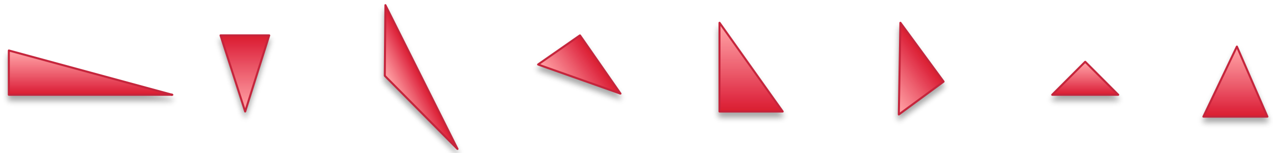
- Scale all polygons such that bbox = unit cube
- Replace polygons by their "center point"
  - E.g., center point = barycenter (Schwerpunkt), or center point = center of bbox of polygon



- Assign Morton codes to points according to enclosing grid cell
- Assign those Morton codes to the original polygons, too



- Now, we've got a list of pairs of ⟨polygon ID, Morton code⟩
- Example:

Pgon ID ⟶


Morton code ⟶	1010	1000	1001	0010	0000	0011	1110	1101

- Sort list according to Morton code, i.e., along z-curve
  ⟶ linearization

Pgon ID ⟶


Morton code ⟶	0000	0010	0011	1000	1001	1010	1110	1101
Array index *i* ⟶	0	1	2	3	4	5	6	7

- Next: find index intervals representing BVH nodes at different levels

- Now, root of BVH = polygons in index range 0,…,N-1
  - All polygons with first bit of Morton code = 0/1 are below/above the plane z = 1/2
  - Find index i in sorted array where first bit (MSB) changes from "0" to "1"
  - Left child of root = polygons in index range 0,…,i-1
  - Right child of root = polygons in index range i,…,N-1
- In general (recursive formulation):
  - Given: level h, and index range i,…,j in sorted array, such that Morton codes are identical for all polygons in that range up to bit h
  - Find index k in [i,j] where the bit at position h' ($h' > h$) in Morton codes changes from "0" to "1"
- Can be achieved quickly by binary search and CUDA's **`__clz()`** function (= "count number of leading zeros")

- Consider polygon i and i+1 in the array
- Condition for "same node":
  Polygons i and i+1 are in the same node of the BVH at level h ⇔
  Morton codes are the same up to bit h
- Define a split marker := ⟨index i, level h⟩
- Parallel computation of all split markers ⟶ "split list":
  - Each thread i checks polygons i and i+1
  - Loop over their Morton codes, let h be left-most bit position where the two Morton codes differ
  - Output split markers $\langle i,h \rangle$, …, $\langle i,3k \rangle$ (seems like a bit of overkill)
  - Can be at most 3k split markers per thread ⟶ static memory allocations works

- Example:

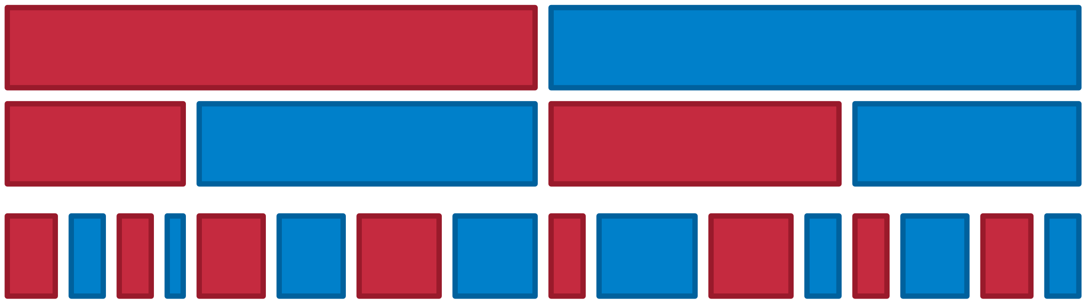
Pgon ID ⟶

Morton code ⟶	0000	0010	0011	1000	1001	1010	1110	1101
Array index i ⟶	0	1	2	3	4	5	6	7

↓	↓	↓	↓	↓	↓	↓
(0,3)	(1,4)	(2,1)	(3,4)	(4,3)	(5,2)	(6,4)
(0,4)		(2,2)		(4,4)	(5,3)	
		(2,3)			(5,4)	
		(2,4)				

Split pair = (i,h) , $i \in [0,N\text{-}2]$, $h \in [1,3k]$

- Last step:
  - Compact split list
  - Sort split list by level *h*
  - Must be stable sort!
- For each level *h*, we now have ranges of indices in the resulting list; all primitives within a range are in the same node on that level *h*



- Example:

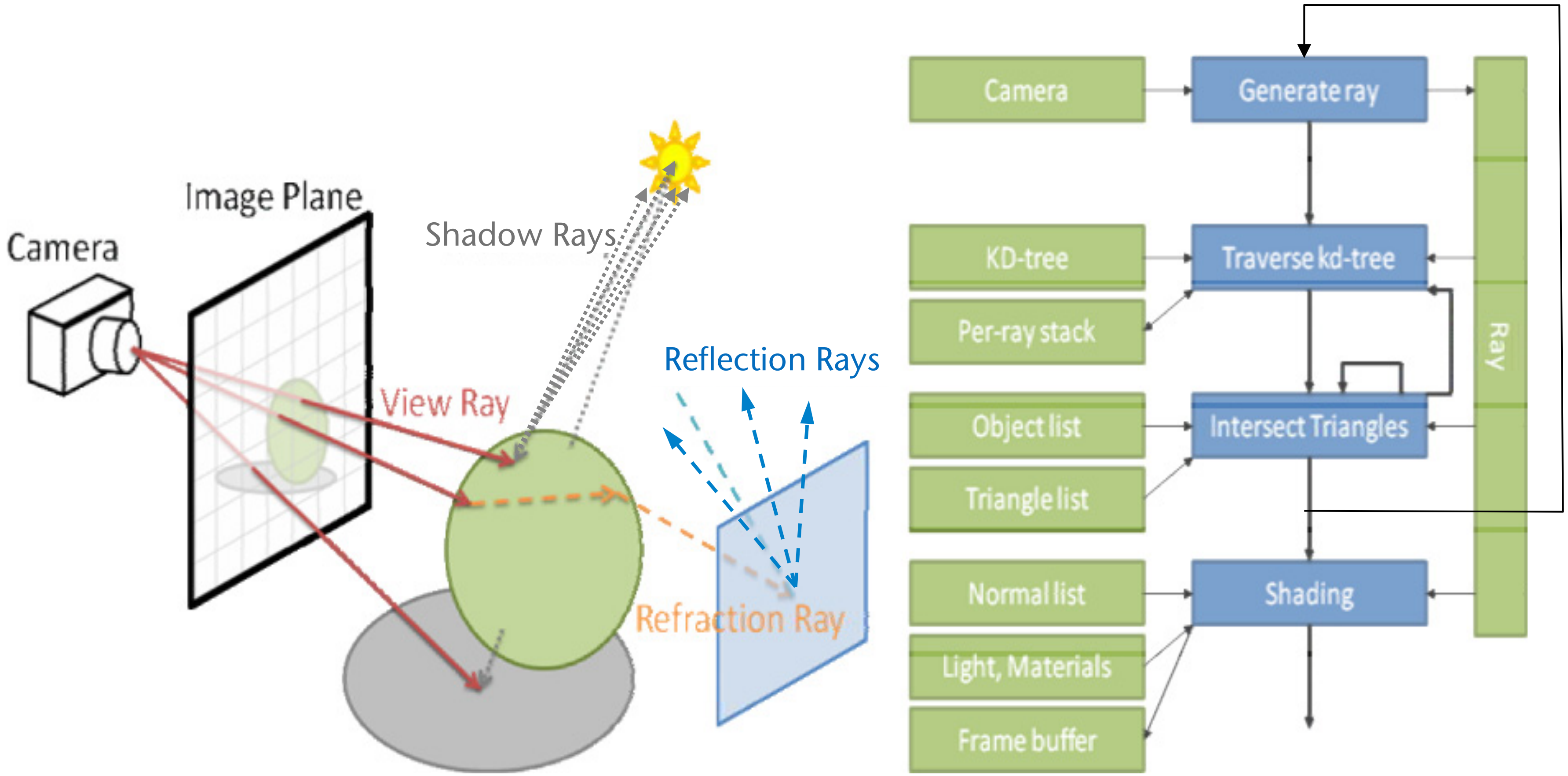
- Final steps:
  - Remove singleton BVH nodes
  - Compute bounding boxes for each node/interval
  - Convert to "regular" BVH with pointers

- Limitations:
  - Not optimized for ray tracing
  - Morton code only *approximates* locality

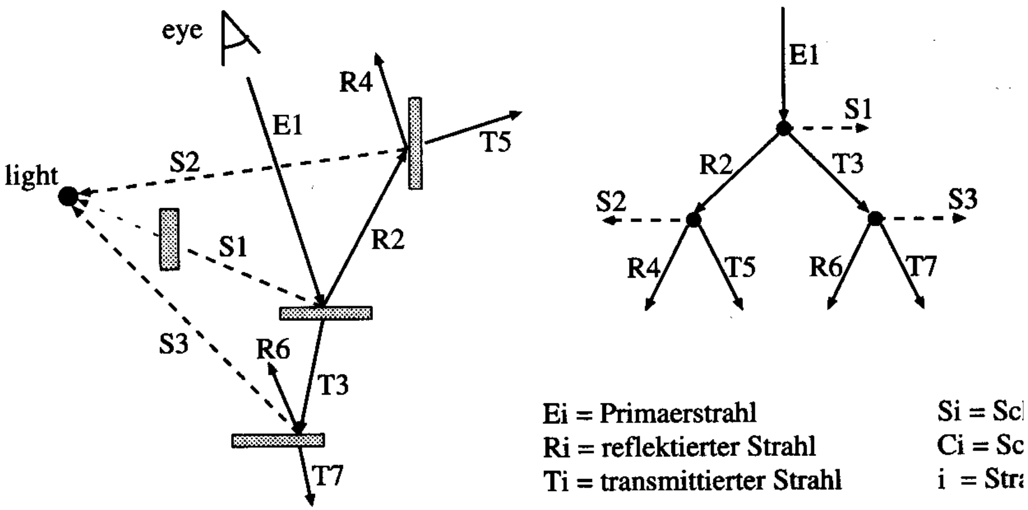
Faster Ray-Tracing by Sorting

- Recap: the principle of ray-tracing
  - Shoot one (or many) primary rays per pixel into the scene
  - Find first intersection (accelerate by, e.g., 3D grid)
  - Generate secondary rays (in order to collect light from all different directions)
  - Recursion ⟶ ray tree
- Ray-Tracing is "embarrassingly parallel":
  - Just start one thread per primary ray
  - Or, is it that simple?

- Visualization of the principle and the work flow:



- The ray tree for one primary ray:

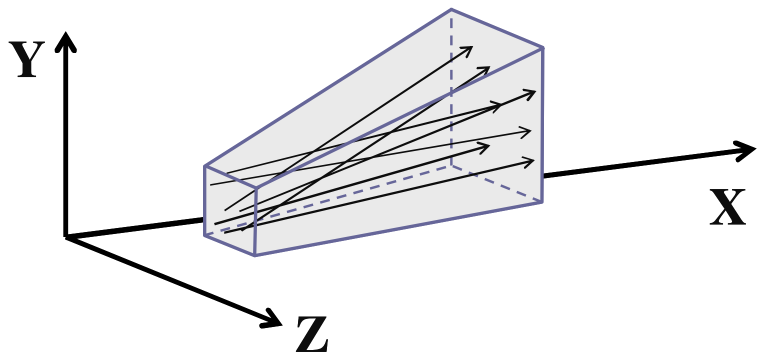


- Problem for massive parallelization:
  - Each thread traverses their own ray tree
  - The rays each thread currently follows go in all kinds of different directions
  - Consequence: **thread divergence**!
  - Another problem: each thread needs their own stack!

- Definition coherent rays:

  Two rays that have "approximately" the same origin and the same direction are said to be coherent rays.

  A set of coherent rays is sometimes called a coherent ray packet.



- Observations:
  - Coherent rays are likely to hit the *same object* in the scene
  - Coherent rays will likely hit the *same cells* in an acceleration data structure (e.g., grid or kd-tree)

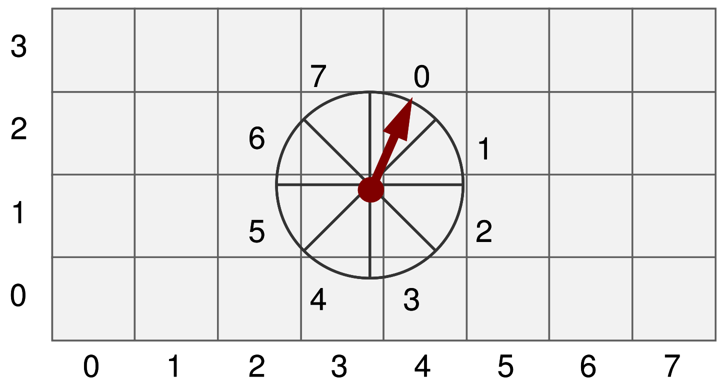
Approach to Solve the Divergence Problem

- Take a stream of rays as input
  - Can be arbitrary mix of primary, secondary, tertiary, shadow rays, ...
- Arrange them into packets of coherent rays

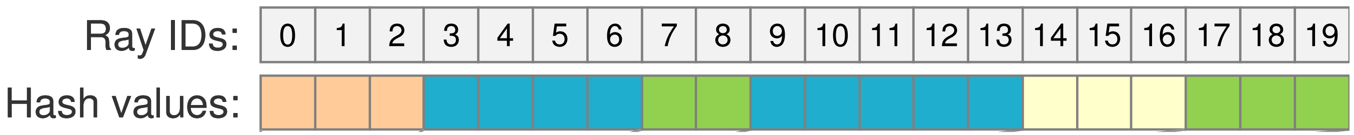
  In the following, we will look at this step
- Compute ray-scene intersections
  - One thread per ray
  - Each block of threads processes one coherent ray packet
  - Each thread traverses the acceleration data structure
  - At the end of this procedure, each thread generates a number of new rays

Identifying Coherent Rays

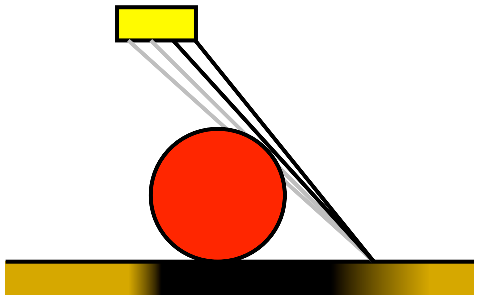
- General approach: classification by discretization
- Here: compute a (trivial) *hash value* per ray
  - Discretize the ray origin by a 3D grid ⟶ first part of hash value
  - Discretize ray direction by direction cube ⟶ second part
  - Concatenate the two hash parts ⟶ complete hash value

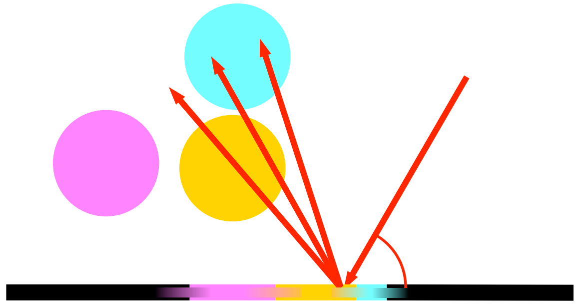


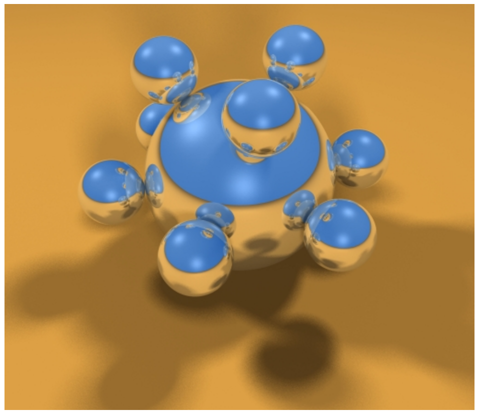
- Can be done in parallel for each ray:

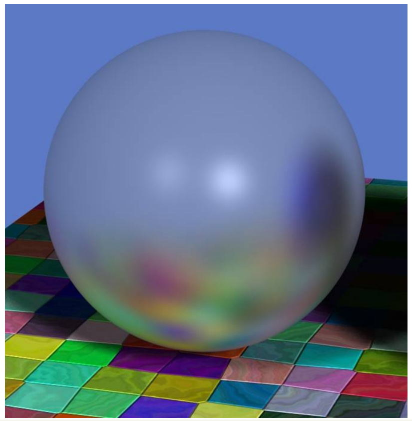


- Note: often, there are many consecutive rays (in the input array) that are coherent, i.e., will map to the same ray hash value
  - For instance, shadow rays
  - Multiple secondary rays from glossy surfaces, etc.

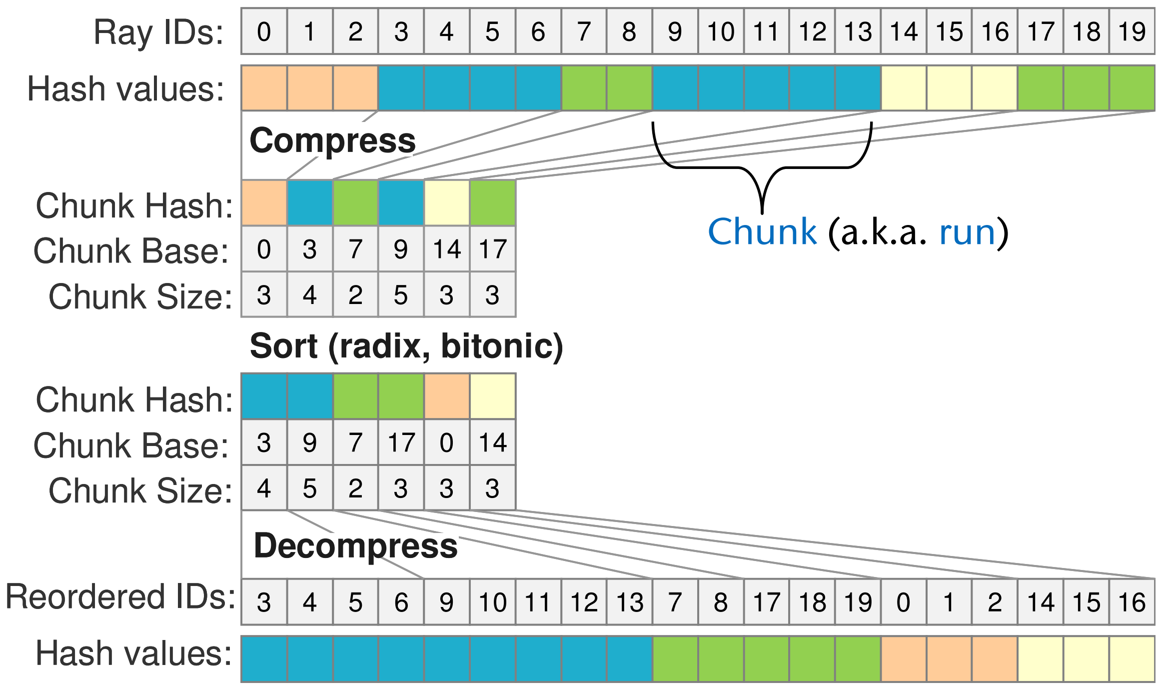






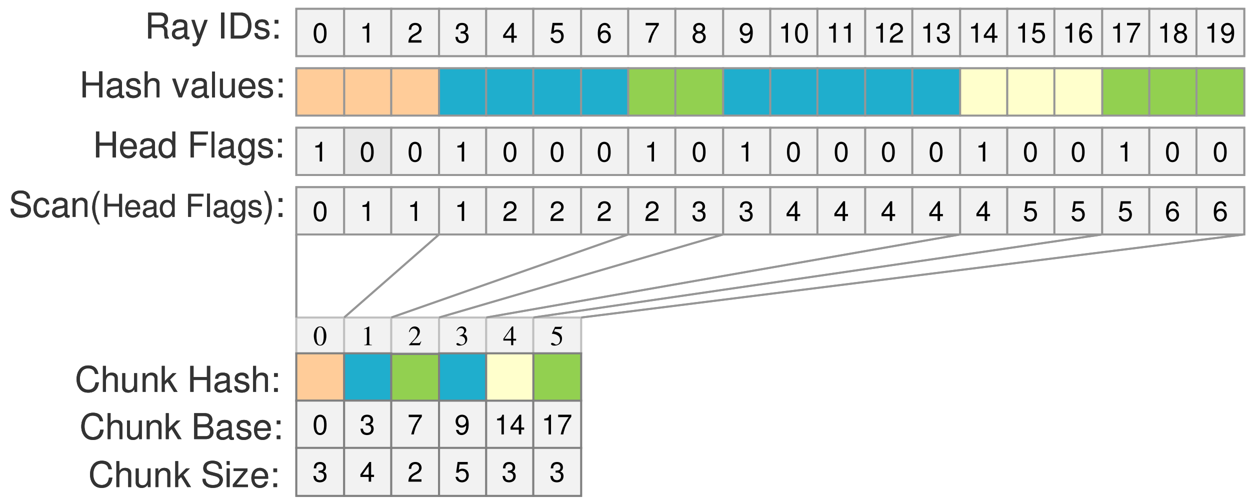


- Can we sort the array of rays yet?
- We could, but we'd perform way too much work!
- Idea:
  1. Compact the array
     - Similar to run length compression/coding
  2. *Sort*
  3. Unpack



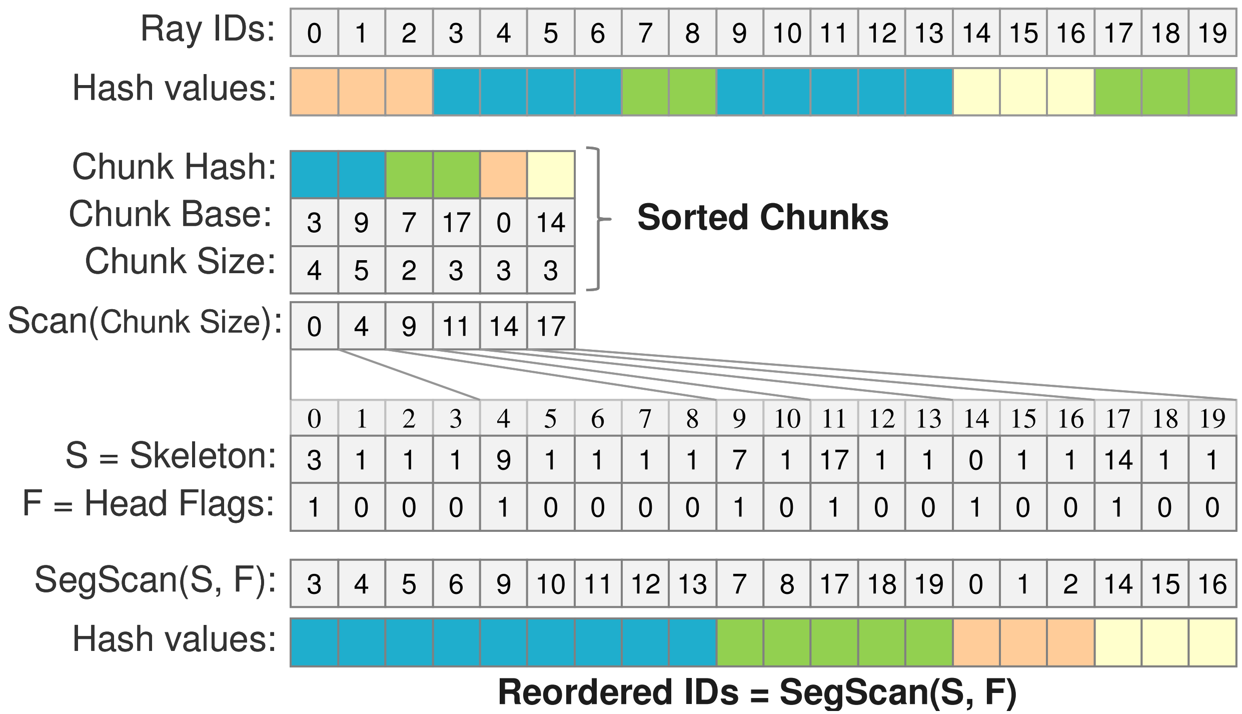
Ray Array Compaction

1. Set all `HeadFlags[i] = 1`, where `HashValue[i-1] ≠ HashValue[i]`,
   else set `HeadFlag[i] = 0`
2. Apply exclusive prefix sum to `HeadFlags` array ⟶ `ScanHeadFlags`
   - Now, `ScanHeadFlags[i]` contains new position in the Chunk arrays
3. For all *i*, where `HeadFlags[i]==1`:
   ```
   ChunkBase[ ScanHeadFlags[i] ] = i
   ChunkHash[ ScanHeadFlags[i] ] = HashValue[i]
   ```
4. Set all
   ```
   ChunkSize[i] =
   ChunkBase[i+1]
   – ChunkBase[i]
   ```



Unpacking the Chunk Array

- Compute exclusive prefix-sum on `ChunkSize` ⟶ `ScanChunkSize`
  - `ScanChunkSize` contains first index in output array for range of ray IDs the chunk represents
- Init array `S` with `1`'s, init array `HeadFlags` with `0`'s



- For all i = 0, …, #chunks-1: set
  `S[ ScanChunkSize[i] ] = ChunkBase[i]`
  `HeadFlags[ ScanChunkSize[i] ] = 1`
- Perform *inclusive segmented prefix-sum* on S with bounds specified by `HeadFlags` ⟶ `SegScan` array

Ray IDs:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Hash values:																				

Sorted Chunks

Chunk Hash:						
Chunk Base:	3	9	7	17	0	14
Chunk Size:	4	5	2	3	3	3
Scan(Chunk Size):	0	4	9	11	14	17

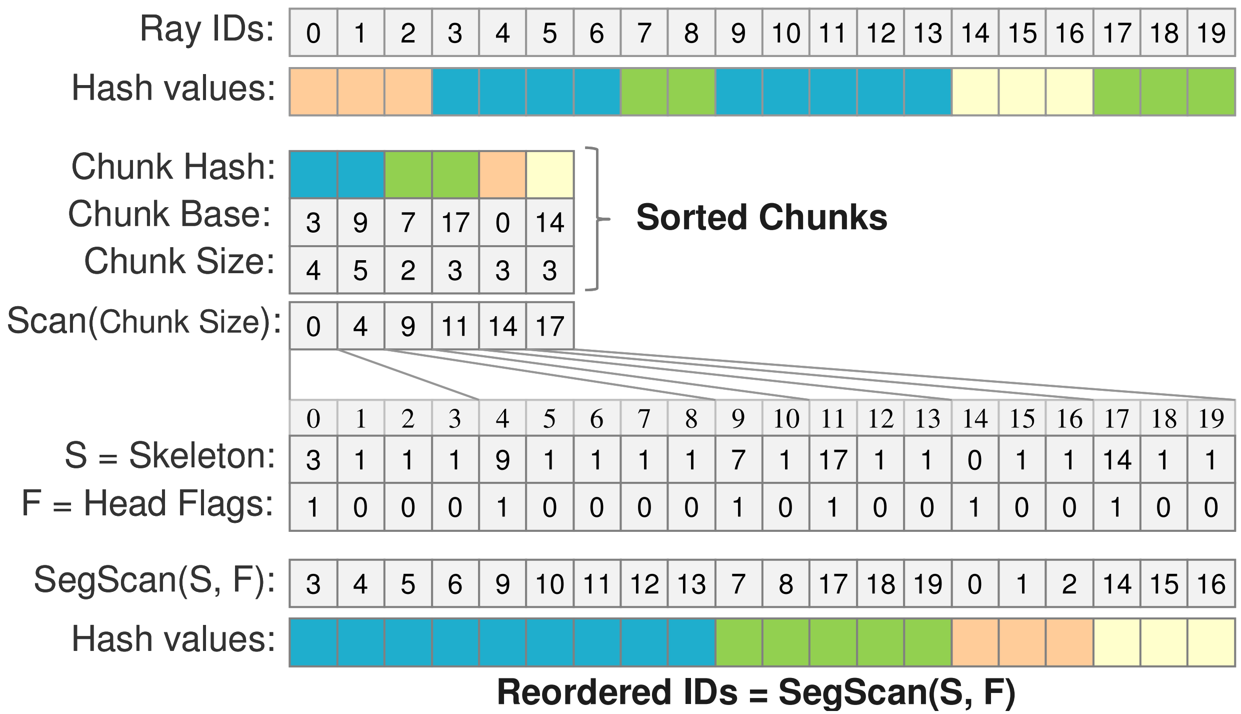
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
S = Skeleton:	3	1	1	1	9	1	1	1	1	7	1	17	1	1	0	1	1	14	1	1
F = Head Flags:	1	0	0	0	1	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0
SegScan(S, F):	3	4	5	6	9	10	11	12	13	7	8	17	18	19	0	1	2	14	15	16
Hash values:																				

Reordered IDs = SegScan(S, F)

- For all *i* in [0,#rays-1]:

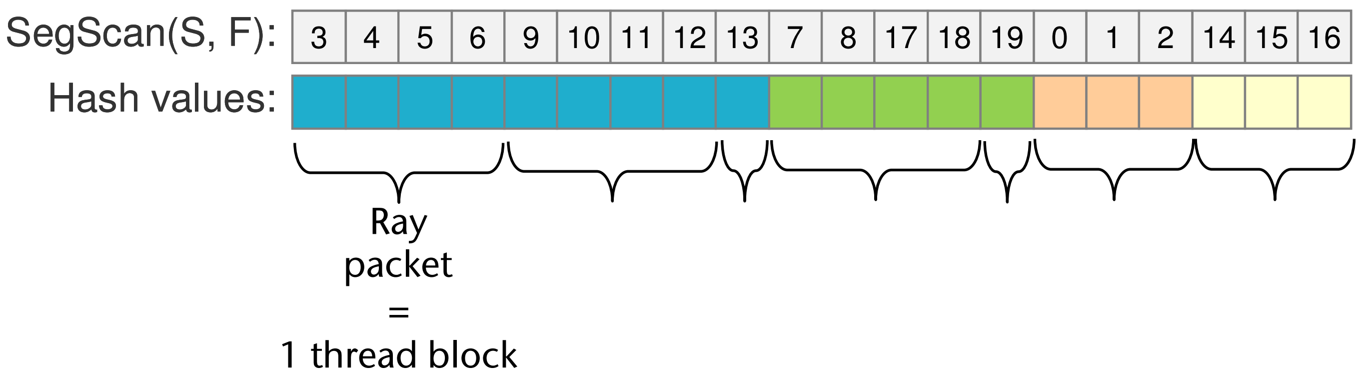
  set `Output[i] = RayID[ SegScan[i] ]`

- Result = array of re-ordered ray IDs, ordered by their hash value (= "coherence hash value")



Partition Into Ray Packets

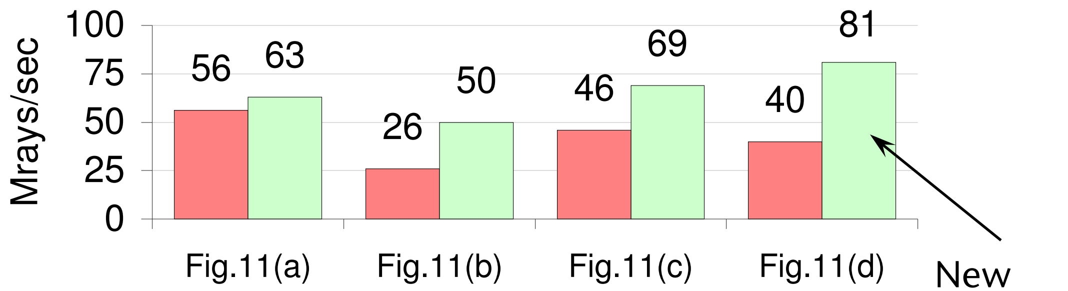
- Remaining problem: the sets of rays with same (coherence) hash value can have very different lengths
- Solution: partition into ray packets
- Definition of ray packet:
  Ray packet = index range (in array of re-ordered rays) such that
  1. all rays have same coherence hash value, and
  2. number of rays in range < maximum packet size.



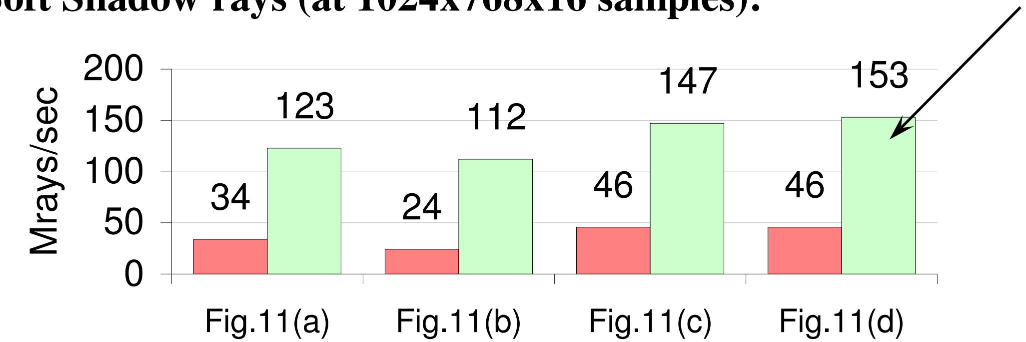
Results

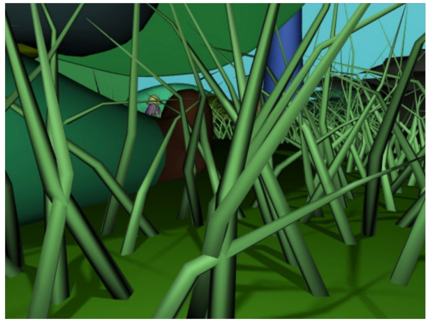
- Comparison (only!) for primary and shadow rays ("New method" contains some further tricks not described here):

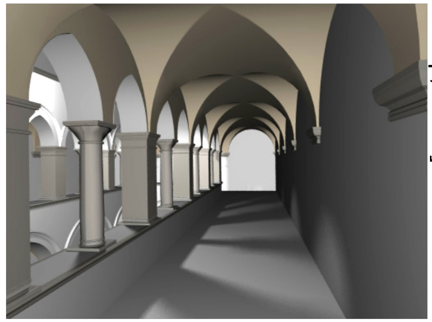
Primary rays (at 1024x768):



Soft Shadow rays (at 1024x768x16 samples):







[Garanzha & Loop, 2010]

Anyone up for a real & thorough comparison?