matfyz

Computer Graphics Charles University

Acceleration Techniques for Ray-Tracing

CGG MFF UK Praha

pepca@cgg.mff.cuni.cz

https://cgg.mff.cuni.cz/~pepca/

Ray-scene intersection

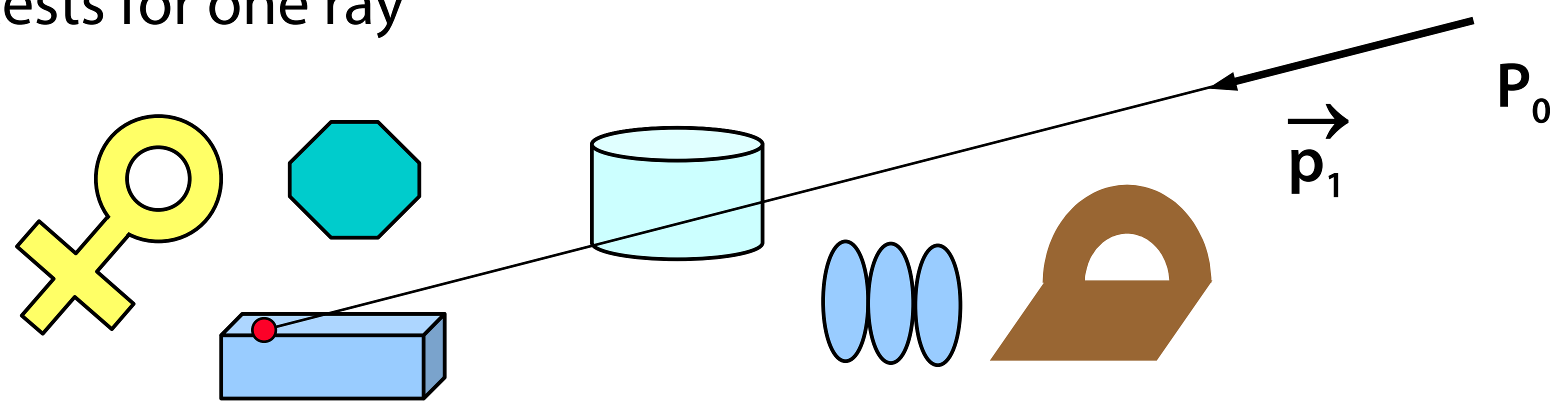
Takes **most of the CPU time** (Whitted: up to 95%)

Scene composed of **objects** (CSG: elementary solids)

- CSG: sphere, box, cylinder, cone, triangle, polyhedron…
- number of objects … **N**

Naïve algorithm tests **every ray** (up to the proper recursion depth **D**) against **every object**

- **O(N)** tests for one ray



Classification

❶ **Faster "ray × scene"**

- **faster "ray × solid"** test
  - » bounding volumes with more efficient intersection algorithms
- **less "ray × solid"** tests
  - » bounding volume hierarchy, space subdivision (spatial data structures), directional techniques (+2D data structures)

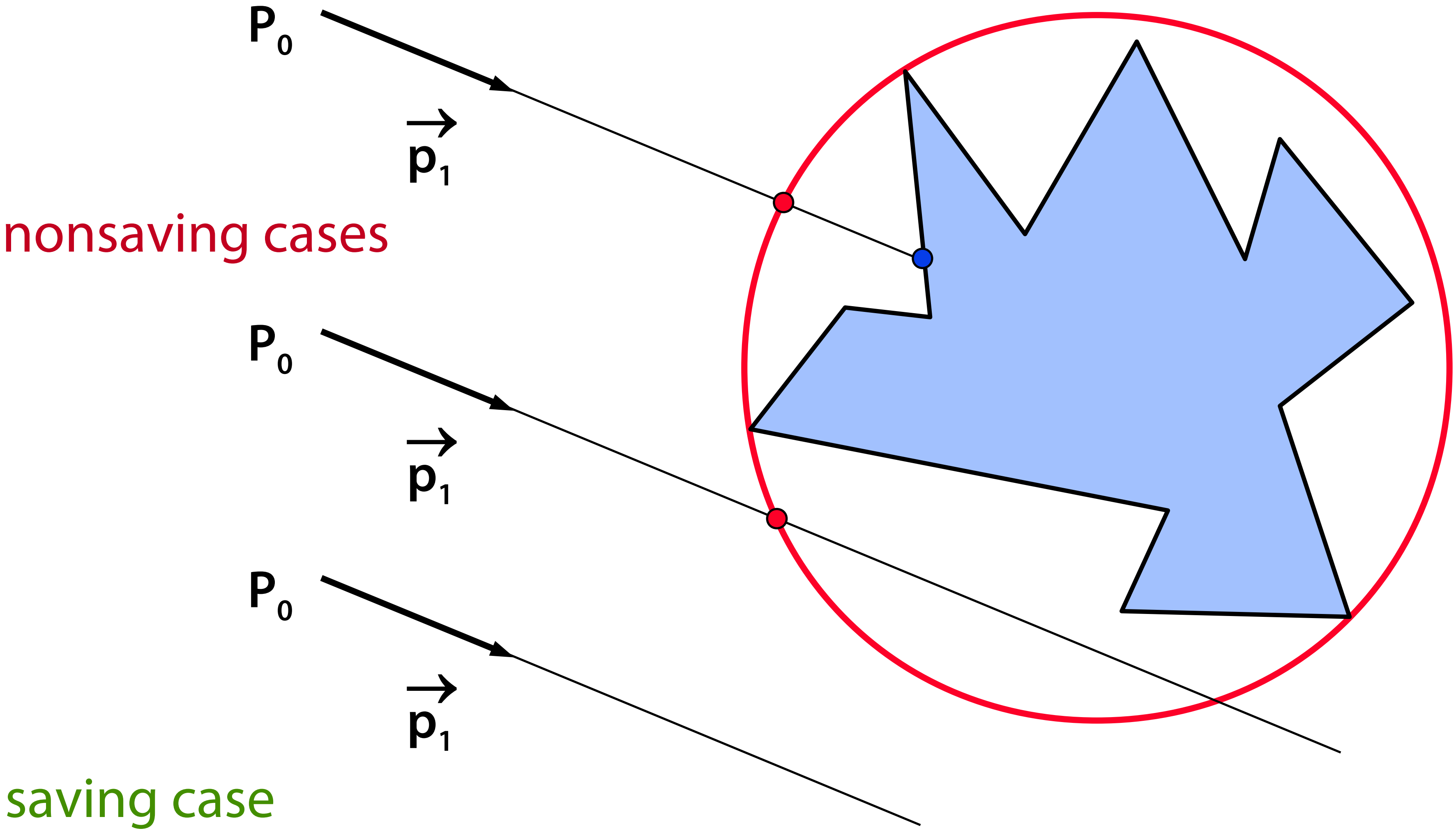
❷ **Less rays**

- » dynamic recursion control, adaptive anti-aliasing

❸ **Generalized rays** (carrying more information)

- » polygonal ray bundle, ray cone…

Bounding volume



Bounding volume

❶ **Intersection with bounding volume is** [much] **faster**

- sphere, **box** (axis-aligned "AABB" or "OBB" with arbitrary orientation), intersection of strips…

❷ A bounding volume should enclose an original object **as tight as possible**

Eficiency of a bounding volume … middle ground between ❶ and ❷

- total asymptotic complexity is still **O(N)**

Bounding volume efficiency

Expected **intersection time** ray vs. object

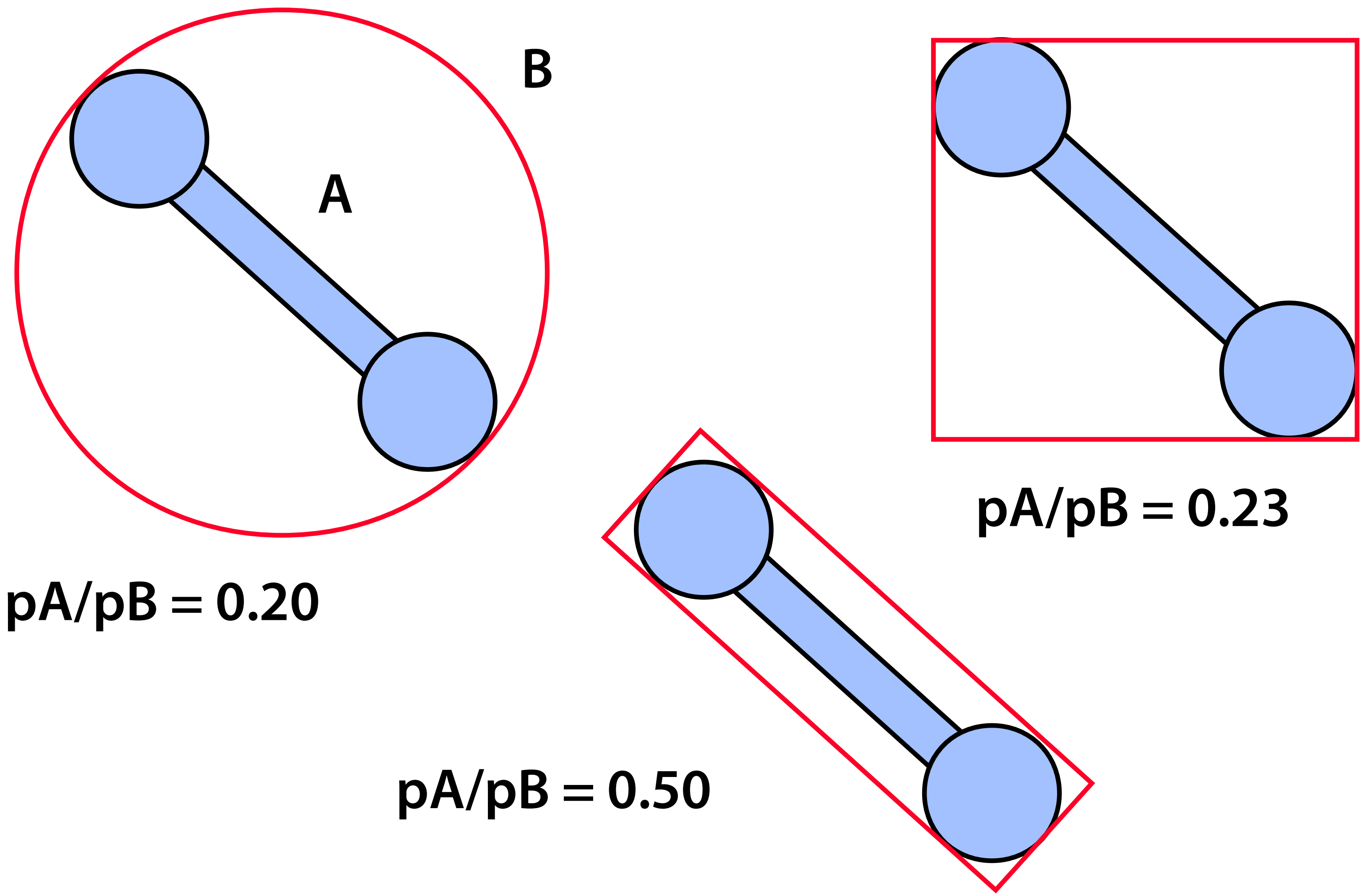
$$\mathbf{B} + \mathbf{p} \cdot \mathbf{I} \overset{?}{<} \mathbf{I}$$

I … intersection time with an **original object**

B … intersection time with a **bounding volume**

p … probability of **hitting a bounding volume** (percentage of ray-hits in total)

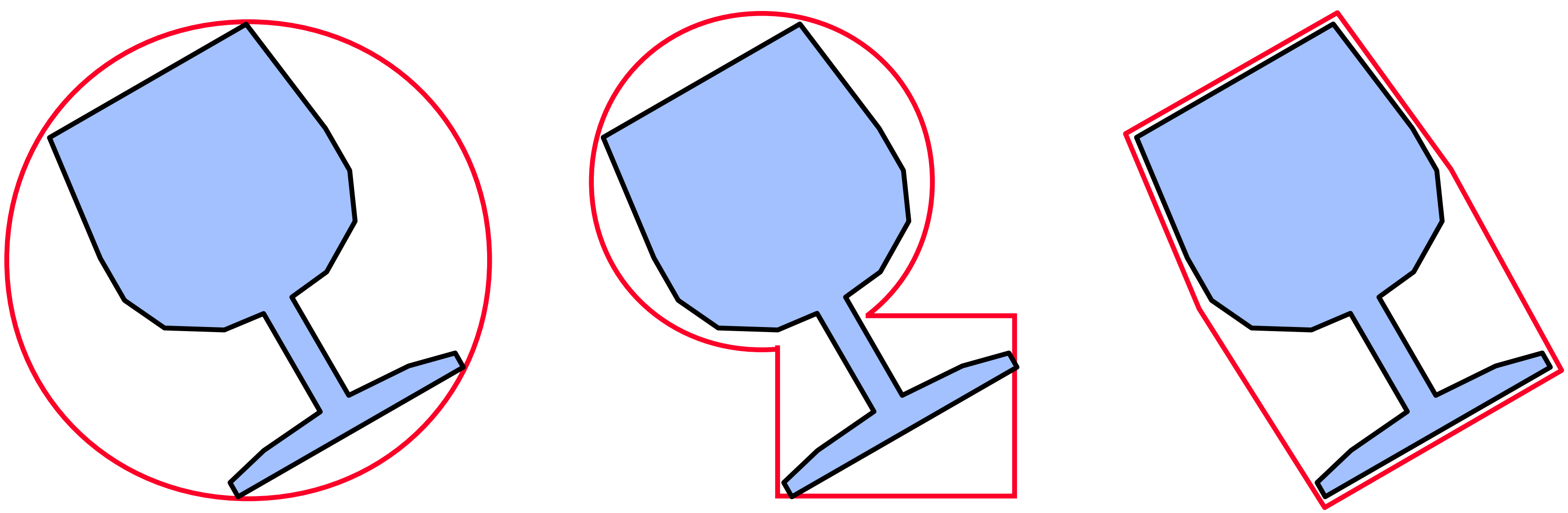
Bounding solid efficiency



Combined bounding solids

Better approximation of an original shape

Unions and intersections of simple bounding shapes



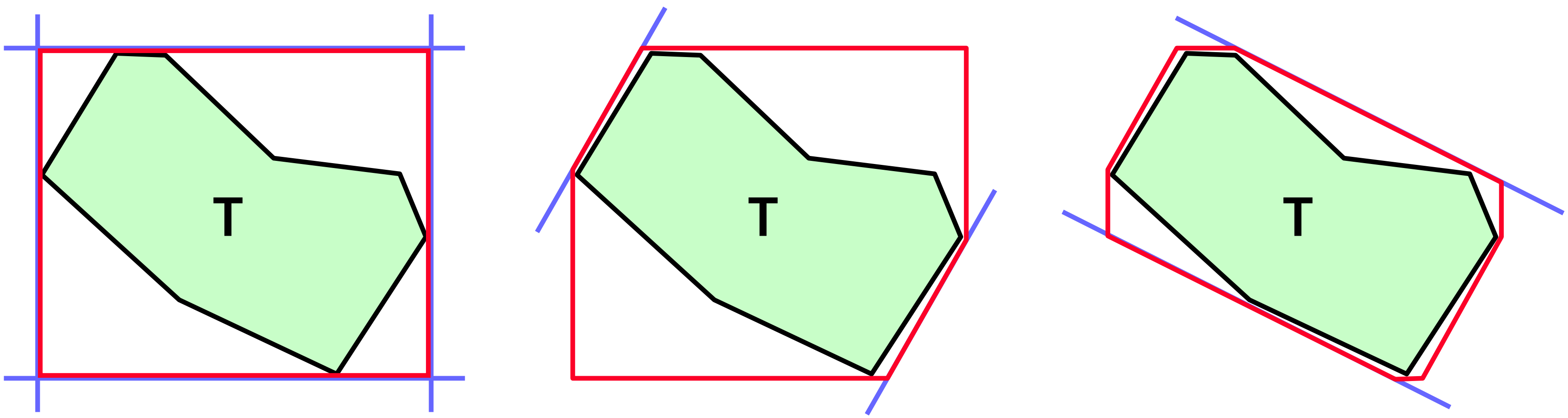
Convex shapes

Bounding solid for **convex shapes**

Intersection of strips (**“k-DOP” system**)

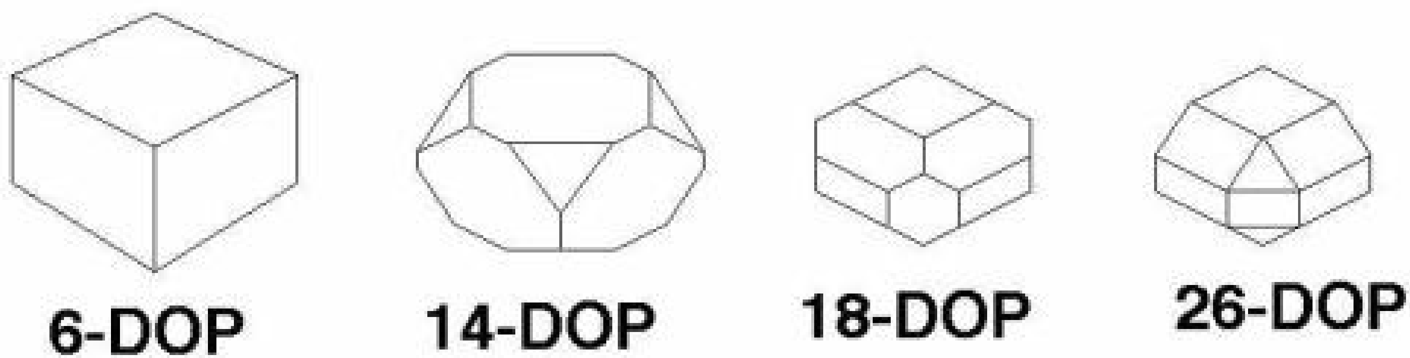
- strip = space between two parallel planes
- efficient computation of **d** and **D** constants is necessary

$$\mathbf{d} = \min_{[x,y,z] \in T} \{ ax + by + cz \}, \quad \mathbf{D} = \max_{[x,y,z] \in T} \{ ax + by + cz \}$$



Most popular k-DOPs

6-DOP (= AABB)
14-DOP (corners)
18-DOP (sides)
26-DOP (sides and corners)

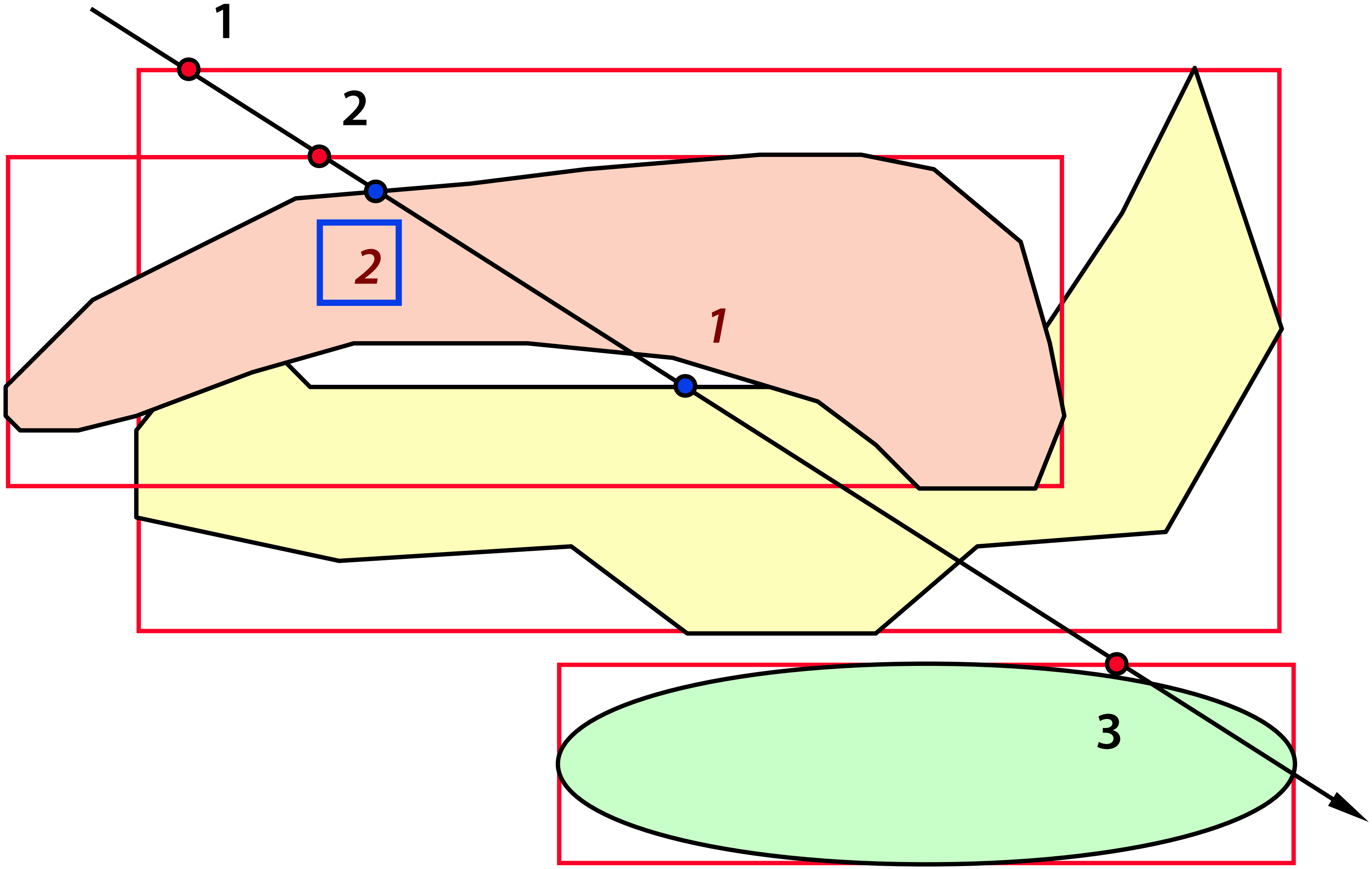


Intersection using bounding volumes

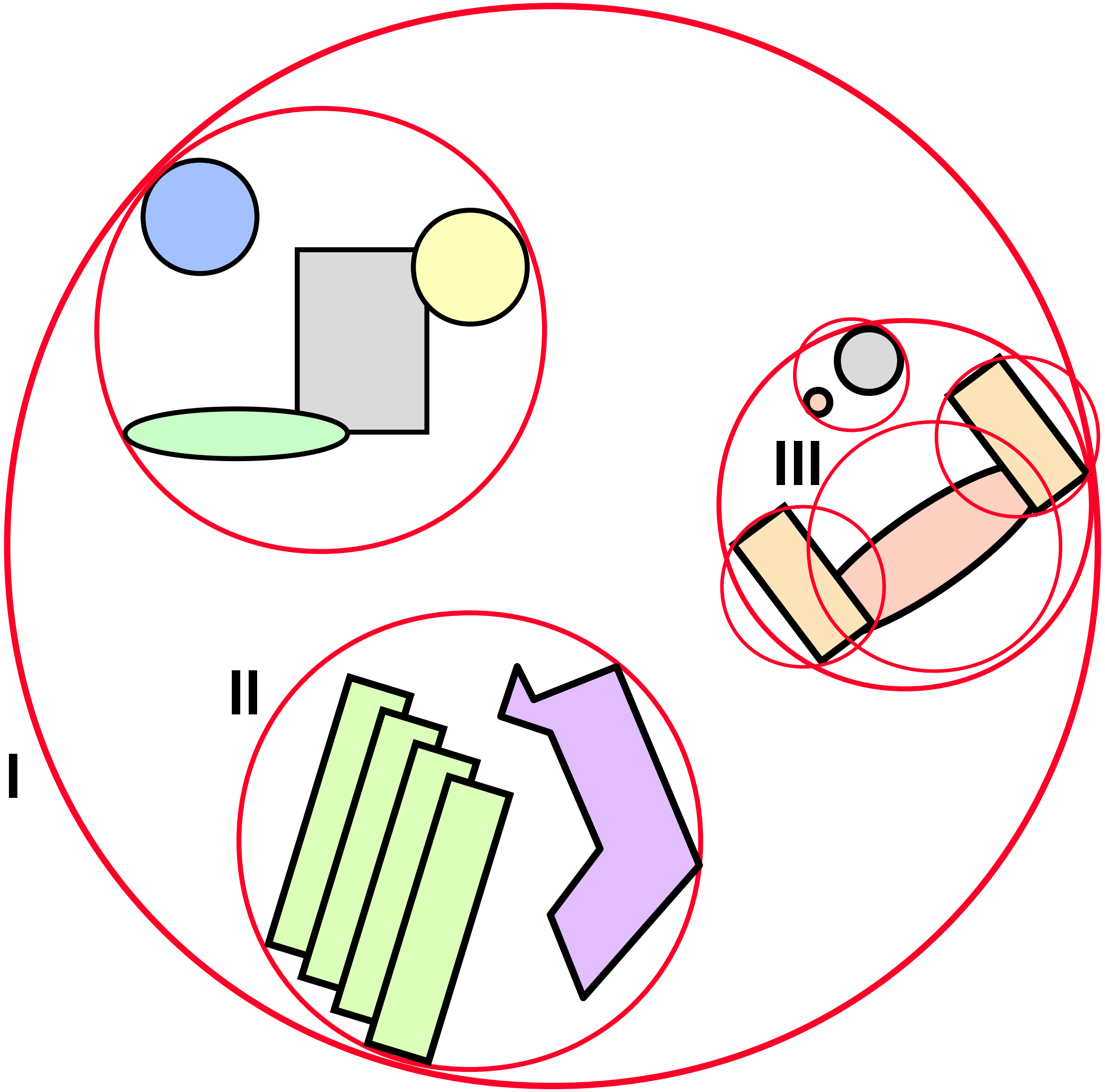
1. Intersections with all **bounding volumes**
2. Intersected **bounding volumes** are sorted in ascending order from the ray origin
3. **Original objects** will be checked (intersected with the ray) in the same order

   If there is an intersection and all **bounding volumes with closer intersection** were already tested, the intersection is the closest one

An efficient algorithm



Bounding Volume Hierarchy (BVH)



Hierarchy

Ideal asymptotic complexity is **O(log N)**

Efficient for **well structured scenes**
- many well separated small objects/clusters
- natural in CSG representation (cutting a CSG tree)

Automatic construction is possible
- building optimal tree would be very complex
- suboptimal algorithms – many different principles

In case of “AABB“ it is called **R-tree** (Guttman, 1984)
- see: database spatial query technology

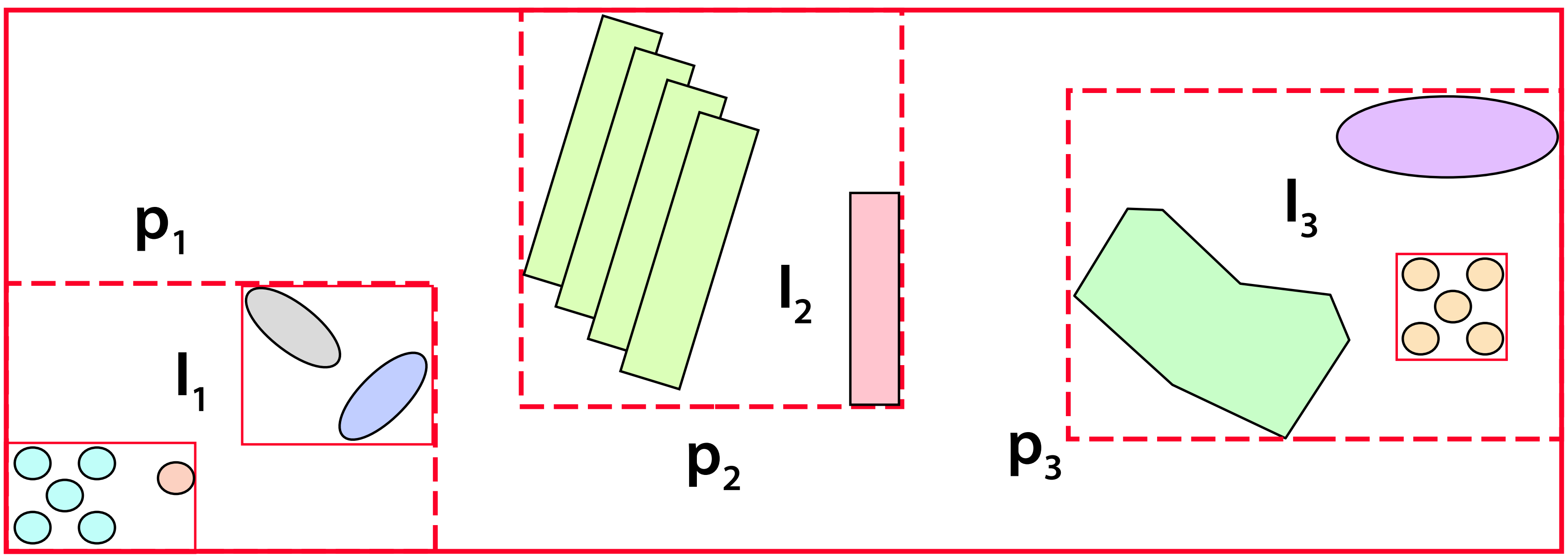
Hierarchy efficiency

$$K \cdot B + \sum_{i=1}^{K} p_i\, I_i \overset{?}{<} \sum_{i=1}^{K} I_i$$

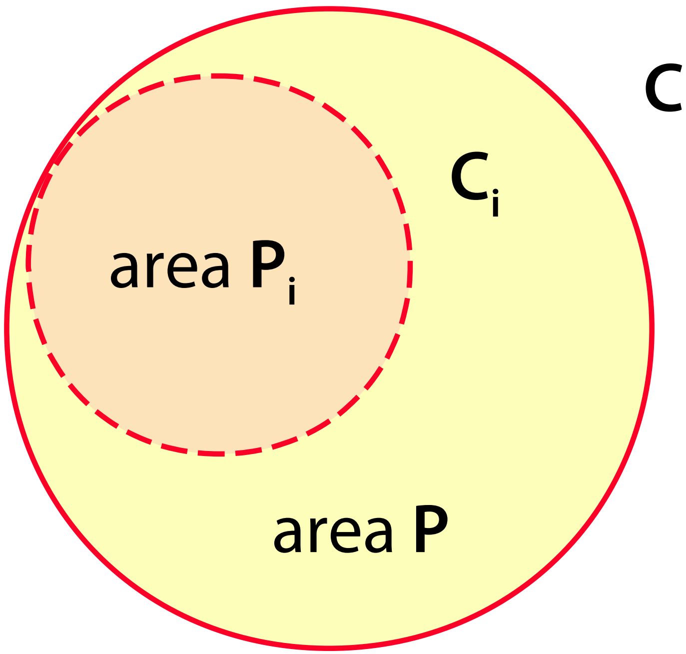
B … intersection time with the bounding volume

p_i … probability of hitting the i-th bounding volume

I_i … time for objects inside of the i-th bounding volume



Sub-volumes (basis for SAH)



P(d), P$_i$(d) … area projected from the direction **d**

S, S$_i$ … surface area of a shape

For a single direction **d**

$$p_i = Pr(\text{ hit } C_i \mid \text{ hit } C) = \frac{P_i(d)}{P(d)}$$

For every direction and **convex objects**

$$p_i = \frac{\int P_i(d)\, dd}{\int P(d)\, dd} = \frac{S_i}{S}$$

Bounding volume hierarchies

"**Sphere tree**" (Palmer, Grimsdale, 1995)

– simple test and transformation, worse approximation

"**AABB tree**", "**R-tree**" (Held, Klosowski, Mitchell, 1995)

– simple test, complex transformation

"**OBB tree**" (Gottschalk, Lin, Manocha, 1996)

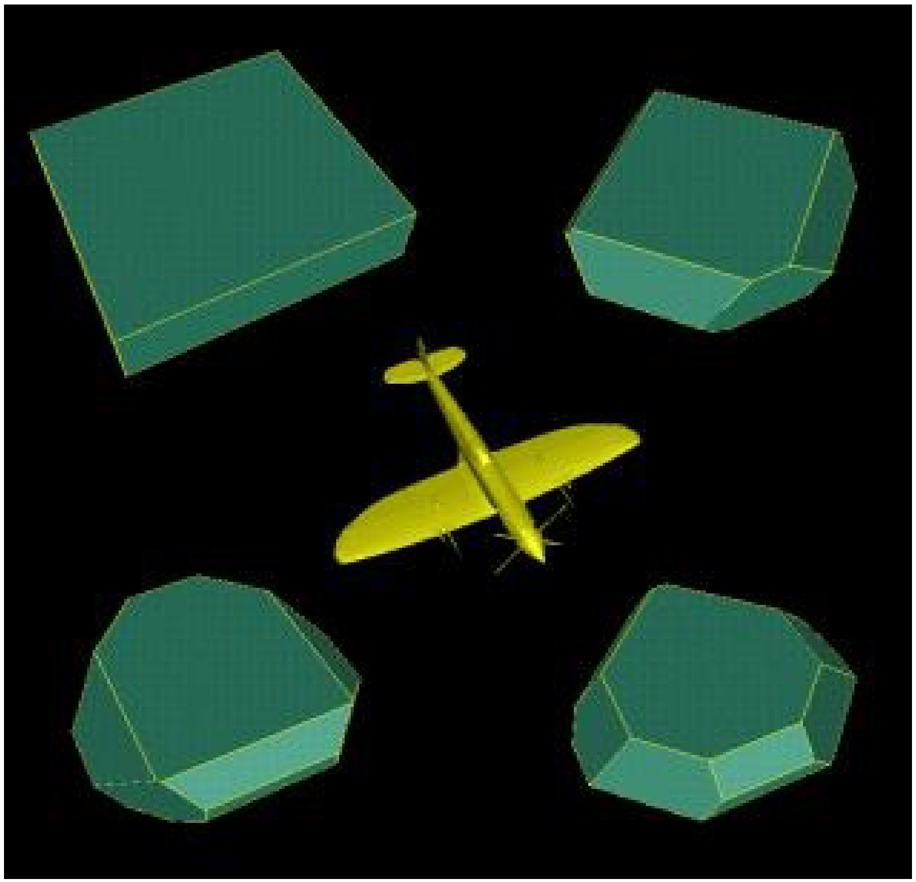
– simple transformation, more complex test, good approximation

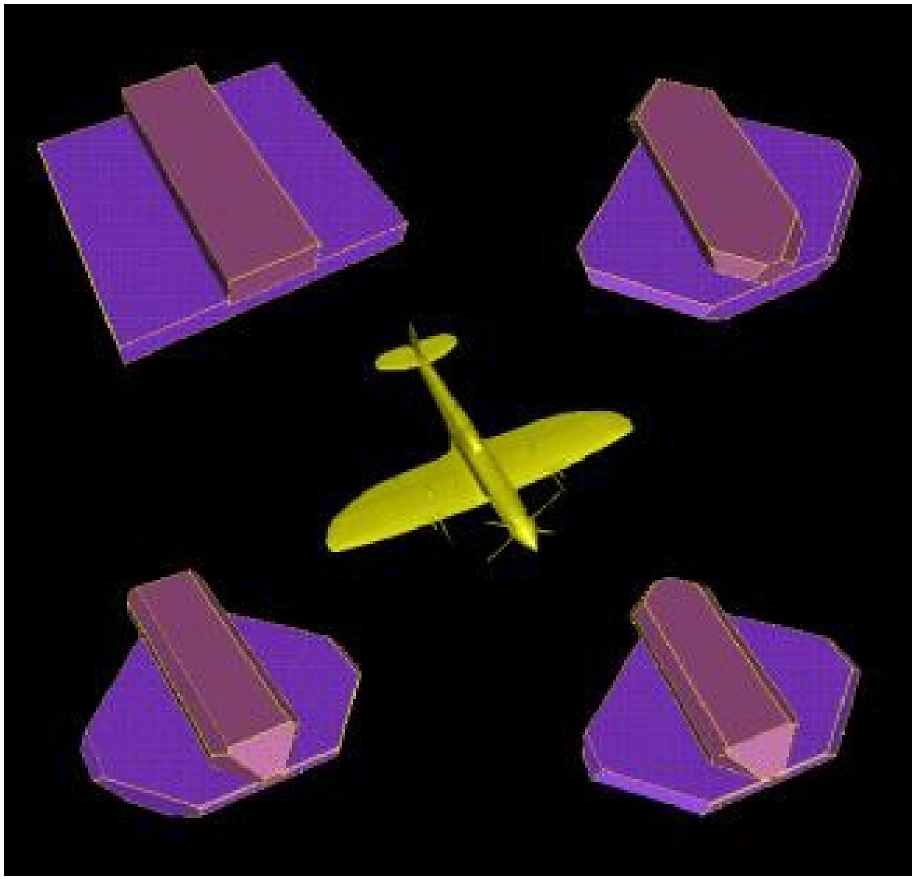
"**K-DOP tree**" (Klosowski, Held, Mitchell, 1998)

– more complex transformation and test, excellent approximation

K-DOP hierarchy – levels 0 & 1

6-DOP 14-DOP 6-DOP 14-DOP



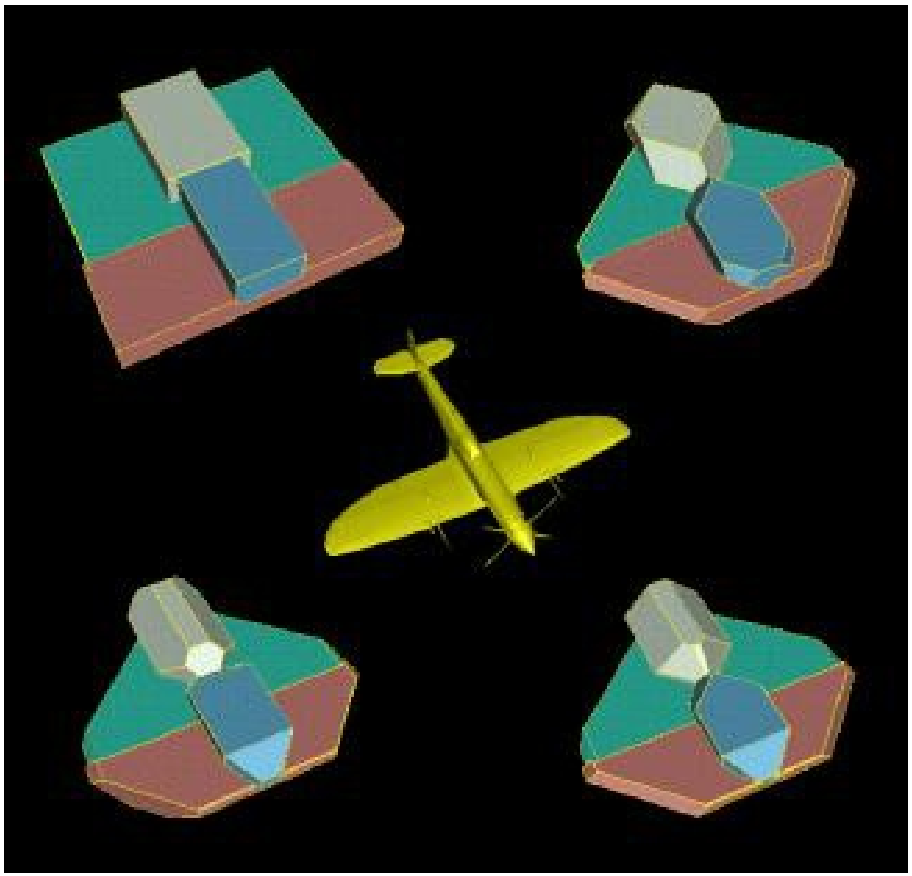


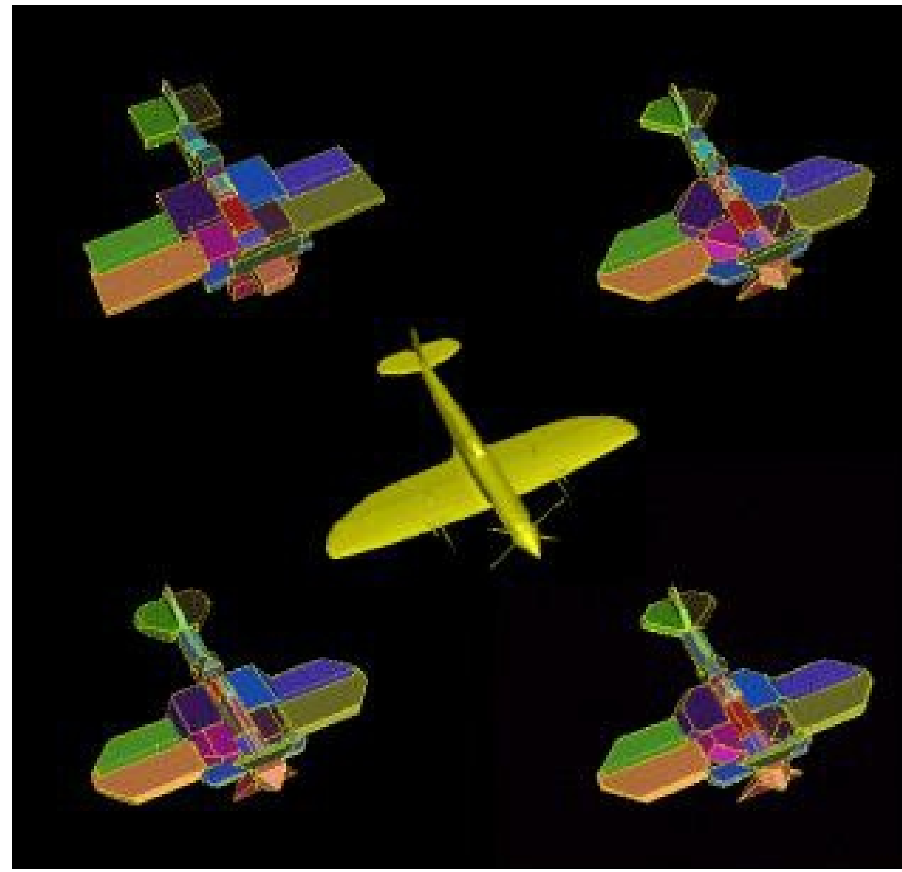
18-DOP 26-DOP 18-DOP 26-DOP

© 2007, dinglijl

K-DOP hierarchy – levels 2 & 3

6-DOP 14-DOP 6-DOP 14-DOP





18-DOP 26-DOP 18-DOP 26-DOP

BVH construction ideas

Top-down construction

- classical time-efficient construction (“divide and conquer”)
- sub-optimal (“local greedy”) rules like **SAH**

Bottom-up construction

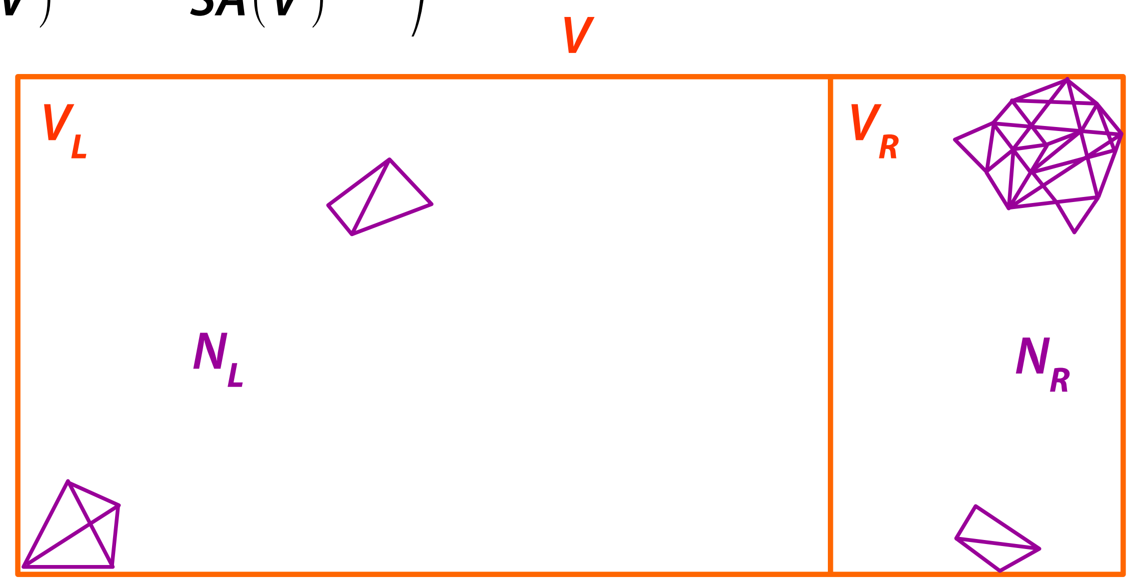
- theoretically better result efficiency but slower construction
- clustering (starting with single triangles)

Parallel construction

- GPU tree construction (Morton code based)

Surface Area Heuristics (SAH)

$$Cost(V \rightarrow \{L, R\}) = t_{tra} + t_{tri}\left(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R\right)$$



V … parent volume

L*, *R … two children (V_L … volume, N_L … number of objects)

t_{tra} … traversal cost (bounding volume tests, recursion)

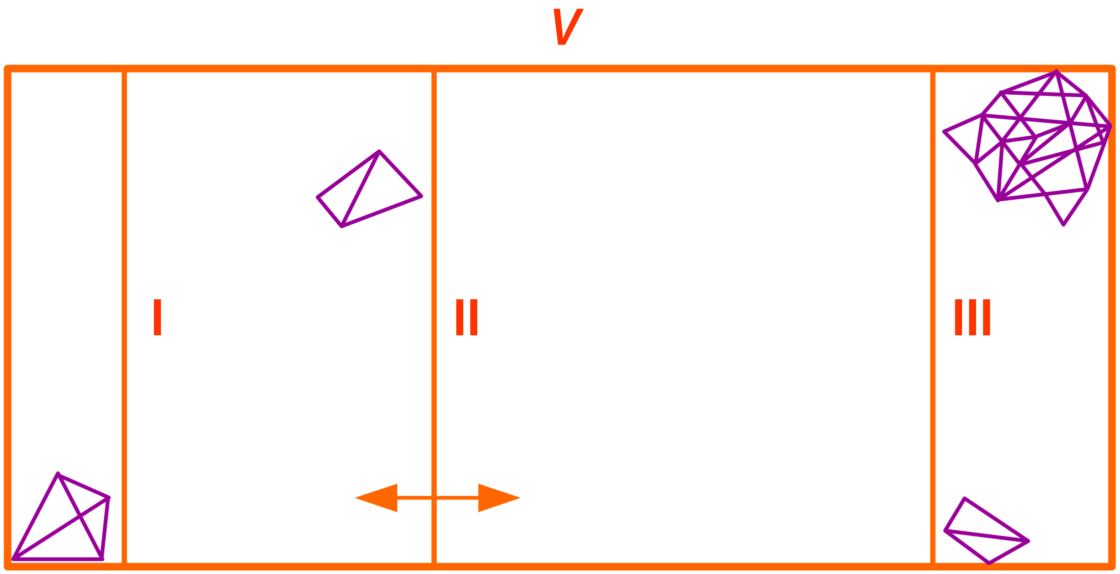
t_{tri} … object (triangle) intersection cost

$SA(V)$ … surface area of the volume V

Top-down SAH construction

In a volume ***V*** we have ***N*** triangles

- a dividing plane divides ***V*** into ***L*** and ***R***
- a plane leading to minimum expected intersection **Cost** has to be found
- plane orientation **{x|y|z}** is deterministic ("round-robin") or optimized as well



SAH decision

$$Cost\left(V \to \{L, R\}\right) = t_{tra} + t_{tri}\left(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R\right)$$

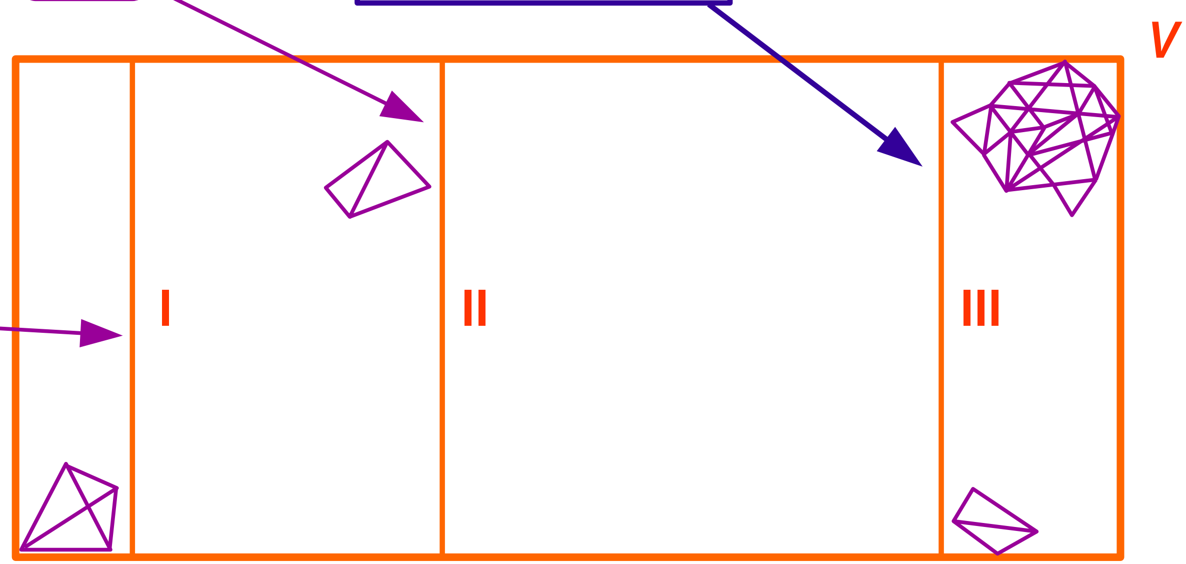
$t_{tra} = 3.0$
$t_{tri} = 1.0$

$N = 76$
$Cost(V) = \boxed{76}$

$N_L = 12$
$N_R = 64$
$p_L = 0.90$
$p_R = 0.29$
$Cost(\mathrm{III}) = \boxed{32.4}$

$N_L = 12$
$N_R = 64$
$p_L = 0.52$
$p_R = 0.67$
$Cost(\mathrm{II}) = \boxed{52.1}$

$N_L = 8$
$N_R = 68$
$p_L = 0.28$
$p_R = 0.91$
$Cost(\mathrm{I}) = \boxed{67.1}$



One more level

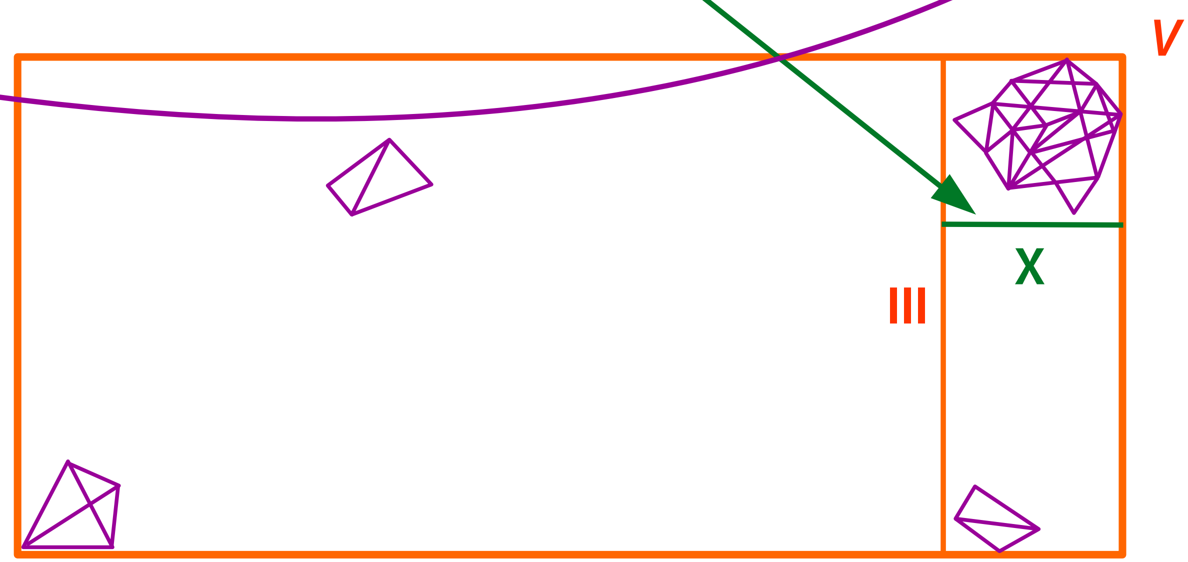
$$Cost\left(V \to \{L,R\}\right) = t_{tra} + t_{tri}\left(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R\right)$$

$N_L = 4$
$N_R = 60$
$p_L = 0.78$
$p_R = 0.40$
$Cost(\mathrm{X}) = \boxed{30.0}$

$t_{tra} = 3.0$
$t_{tri} = 1.0$

$N = 76$
$Cost(V) = \boxed{76}$

$N = 76$
$Cost(\mathrm{III+X}) = \boxed{22.5}$



CSG tree “cutting”

Efficient for **subtractive set operations** (intersection, difference)

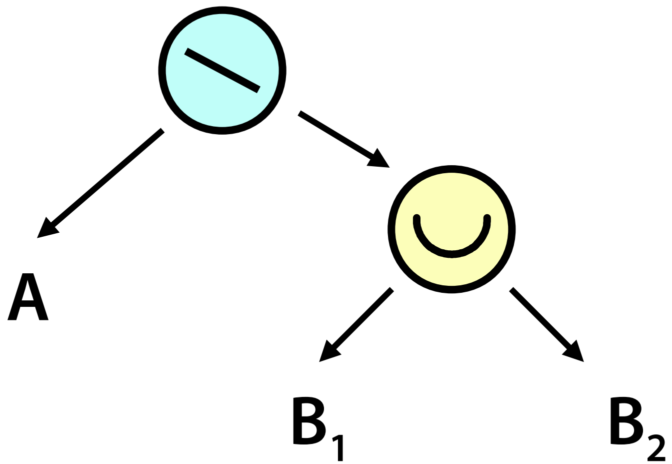
Primary bounding solids are assigned to (finite) **elementary solids**

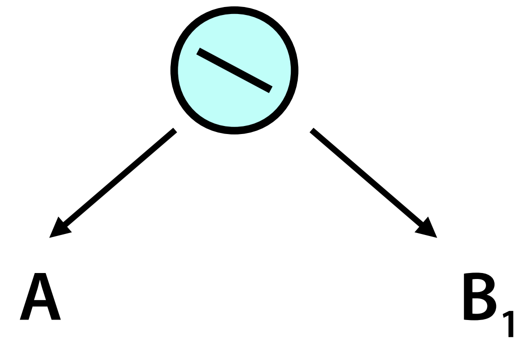
- analytic computation

Bounding solids are propagated **from leaves to the root node**

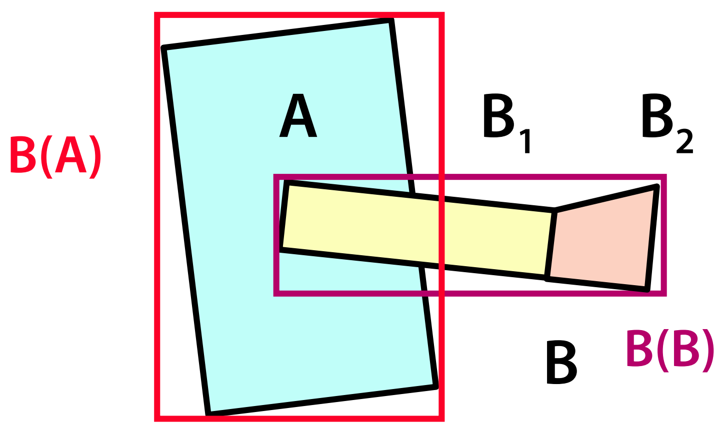
Subtractive operations can reduce bounding solids in ancestors (arguments)

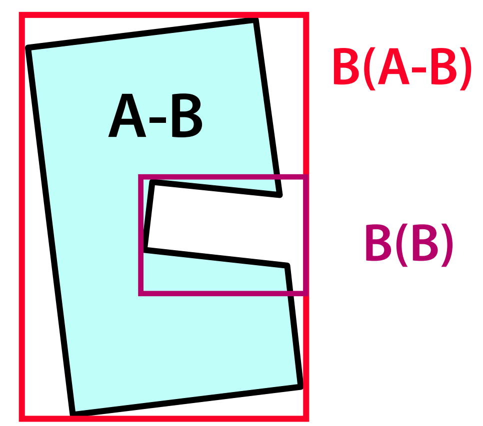
CSG tree “cutting”





$$B(A\text{-}B) \cap B(B_2) = 0$$





Space subdivision (spatial directories)

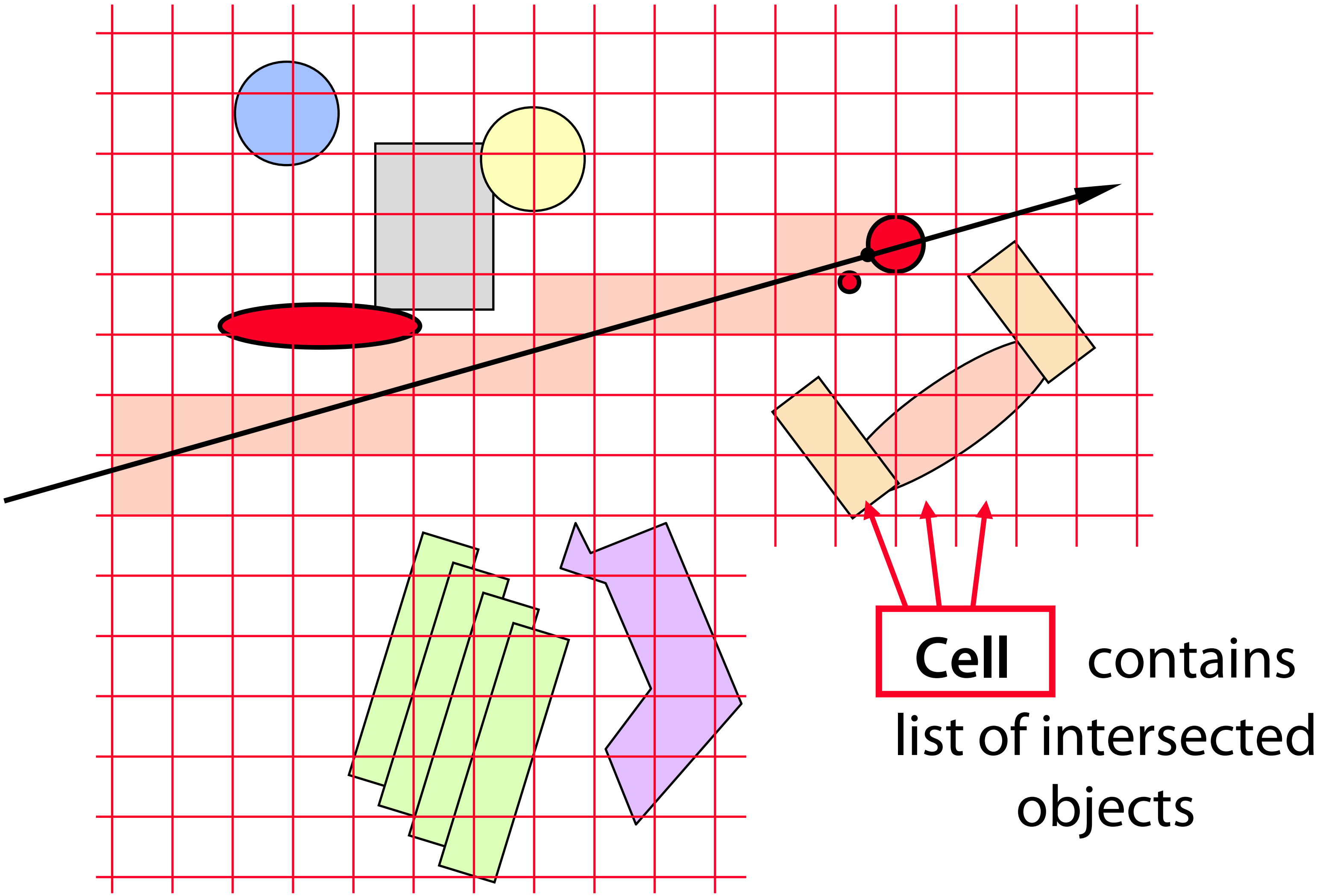
Uniform subdivision (equal cells)

+ simple traversal & addressing
– many traversal steps
– big data volume

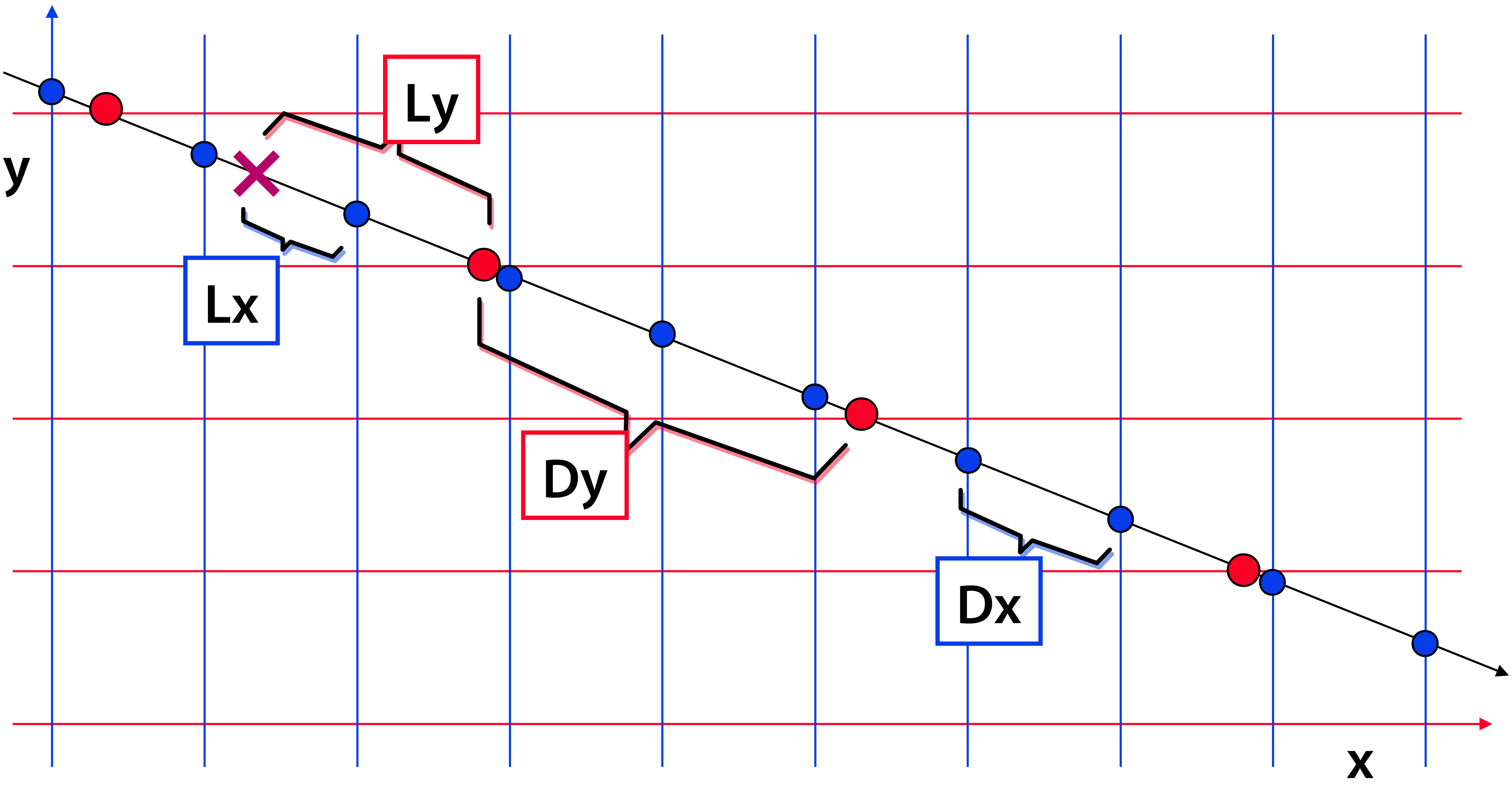
Nonuniform subdivision (mostly adaptive)

+ less traversal steps
+ less data
– more complex implementation (data struture & traversal)

Uniform subdivision (grid)



Grid traversal (3D DDA)



Grid traversal (3D DDA)

Ray $\mathbf{P_0 + t \cdot \vec{p}_1}$ for $\mathbf{t > 0}$

For the given direction $\mathbf{\vec{p}_1}$ there are precomputed **constants Dx, Dy, Dz**

– distance between subsequent intersections of the ray and the parallel wall system (perpendicular to **x, y, z**)

For the $\mathbf{P_0}$ there is an **initial cell [i, j, k]** and quantities **t, Lx, Ly, Lz**

– ray parameter **t**, distances to the closest walls in the **x, y, z** system

Grid traversal (3D DDA)

Processing in the **cell [i, j, k]** (intersections)

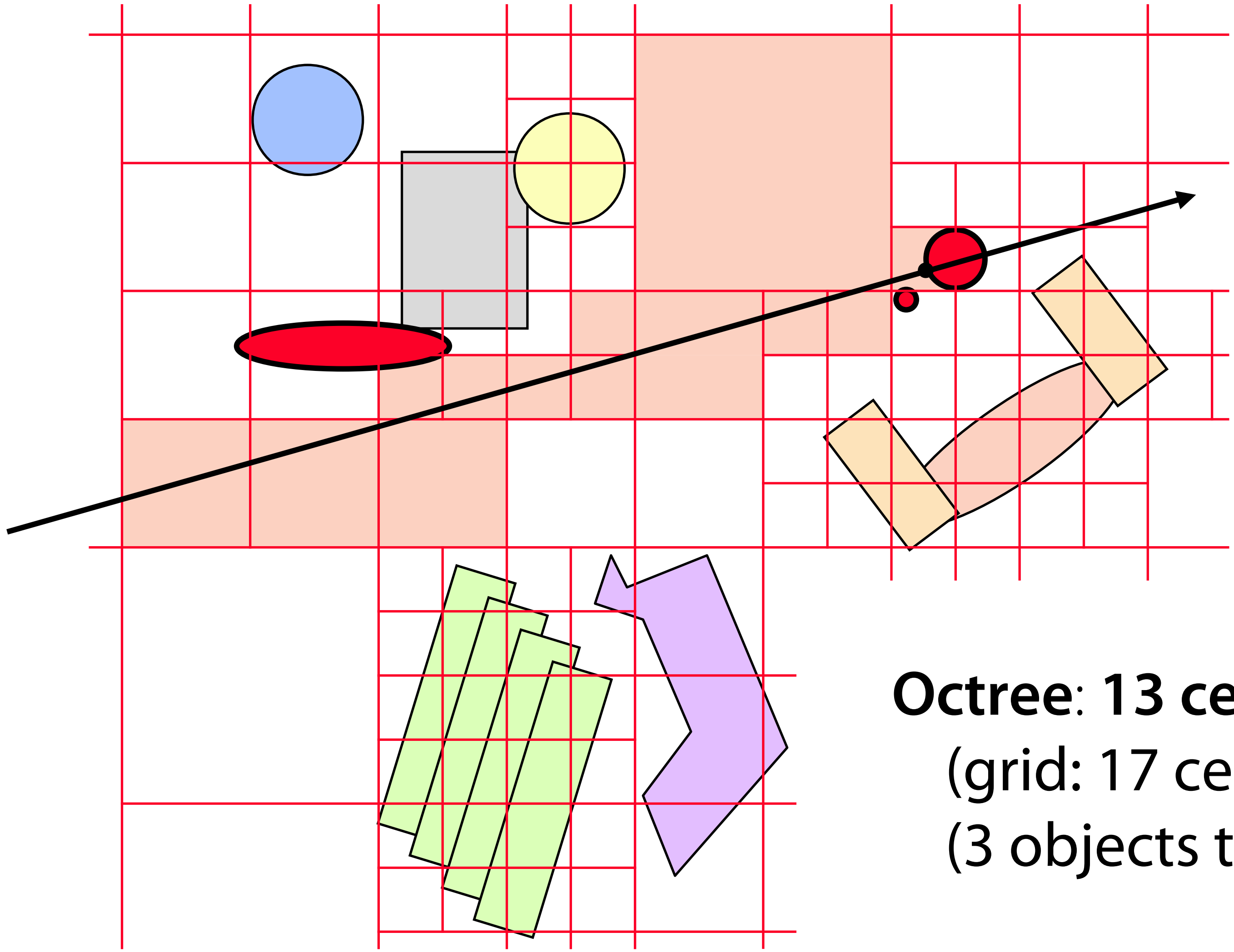
Stepping to the **next cell**

- **D = min { Lx, Ly, Lz };** /* assumption: **D = Lx** */
- **Lx = Dx; Ly = Ly - D; Lz = Lz - D;**
- **i = i ± 1;** /* according to the sign of $\mathbf{P_{1x}}$ */

End conditions

- an actual (the closest) intersection was found
  - » the intersection is <u>in the current cell</u>
- no intersection was found and the next cell is outside of the grid domain

Nonuniform subdivision of space



Octree: **13 cells**
(grid: 17 cells)
(3 objects tested)

Adaptive subdivision systems

Octree (division in the middle)

- representation – pointers, implicit representation or hash table (Glassner)

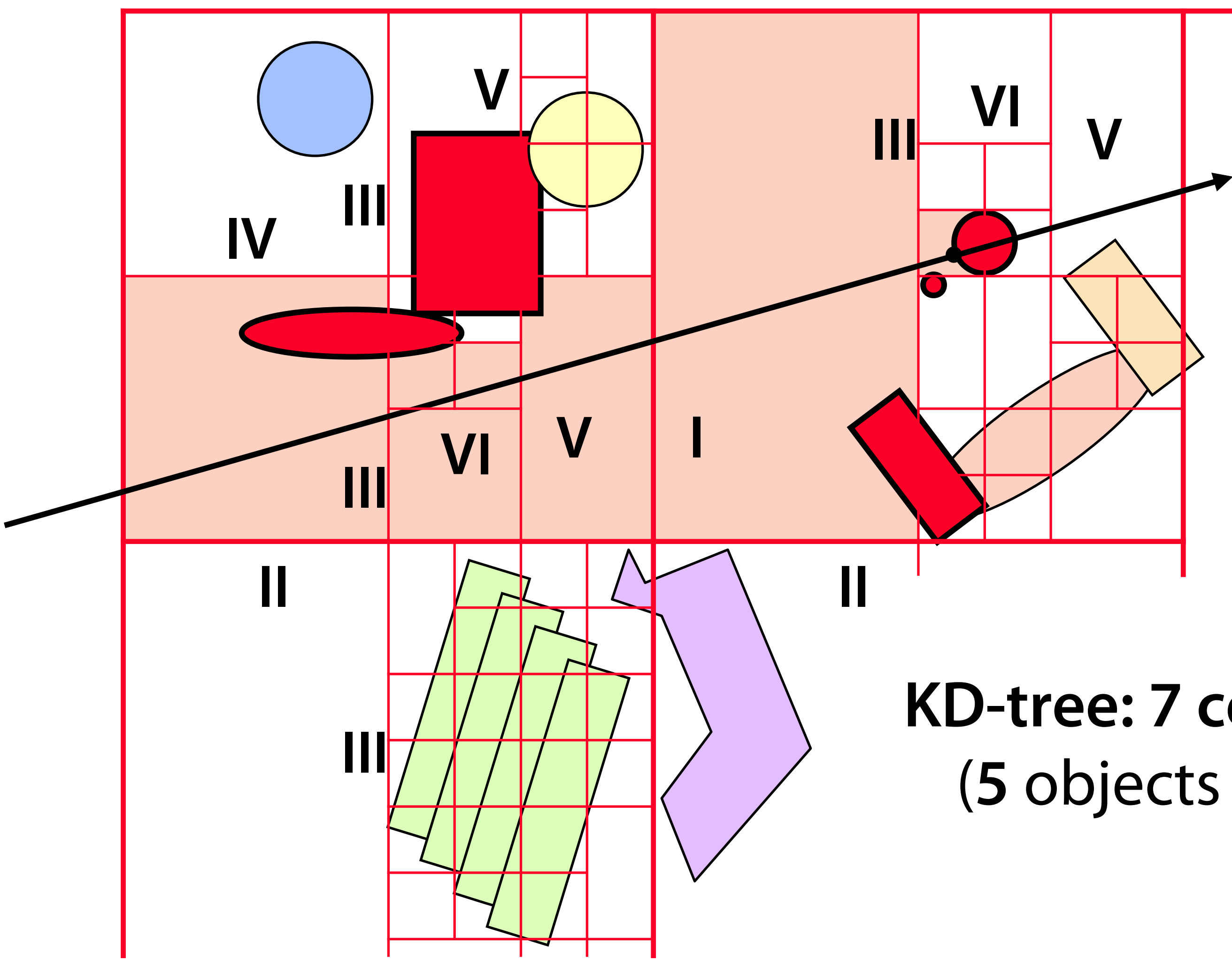
KD-tree (Bentley, 1975)

- static division: in the middle, cyclic coordinate component
- adaptive: both components and bounds are dynamic
- SAH heuristics can be used

[General **BSP-tree**]

- dividing planes have arbitrary orientation

KD-tree (static variant)



KD-tree: 7 cells
(**5** objects tested)

Adaptive subdivision criteria

Limited **number of objects** and **subdivision depth**

- if a cell is intersected by more than **M objects** (e.g. **M = 4…32**), subdivide it
- **maximal subdivision level** is **K** (e.g. **K = 5…25**)

Limited **number of cells** or **memory occupation** instead of subdivision depth limit

- subdivision is finished after filling the whole **reserved memory**
- subdivision controlled by a **breadth-first traversal** (FIFO data structure holding candidate cells)

Traversing adaptive subdivision

Marching the ray – finding the next cell from the root

Preprocessing – tree traversal used for dividing the ray into individual segments (intersections with cells)

- **t** parameter segments for individual cells

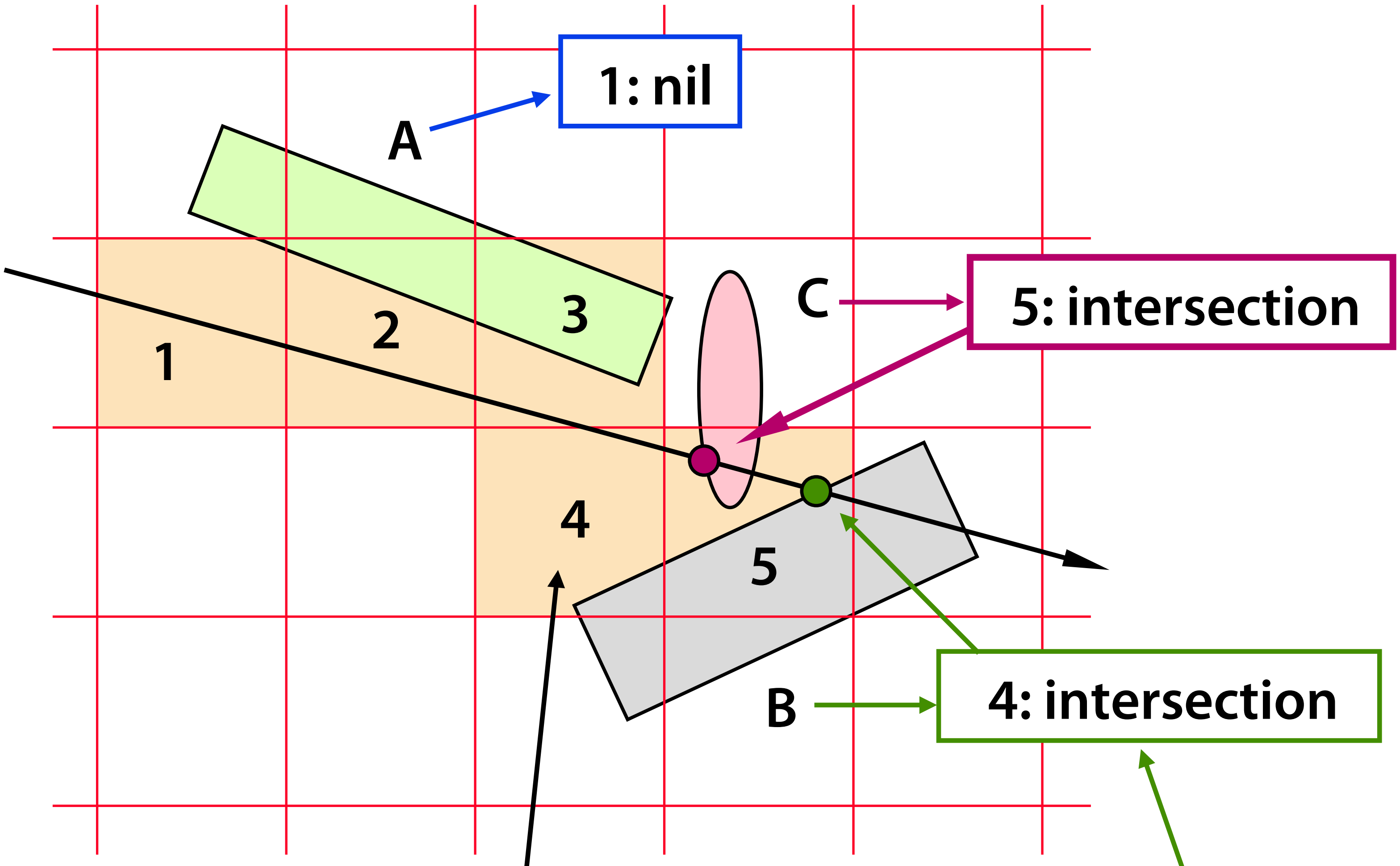
Additional support data (à la “finger tree”)

- pointer to the neighbour cell (on the same tree level)

Recursive **depth-first traversal** with heap

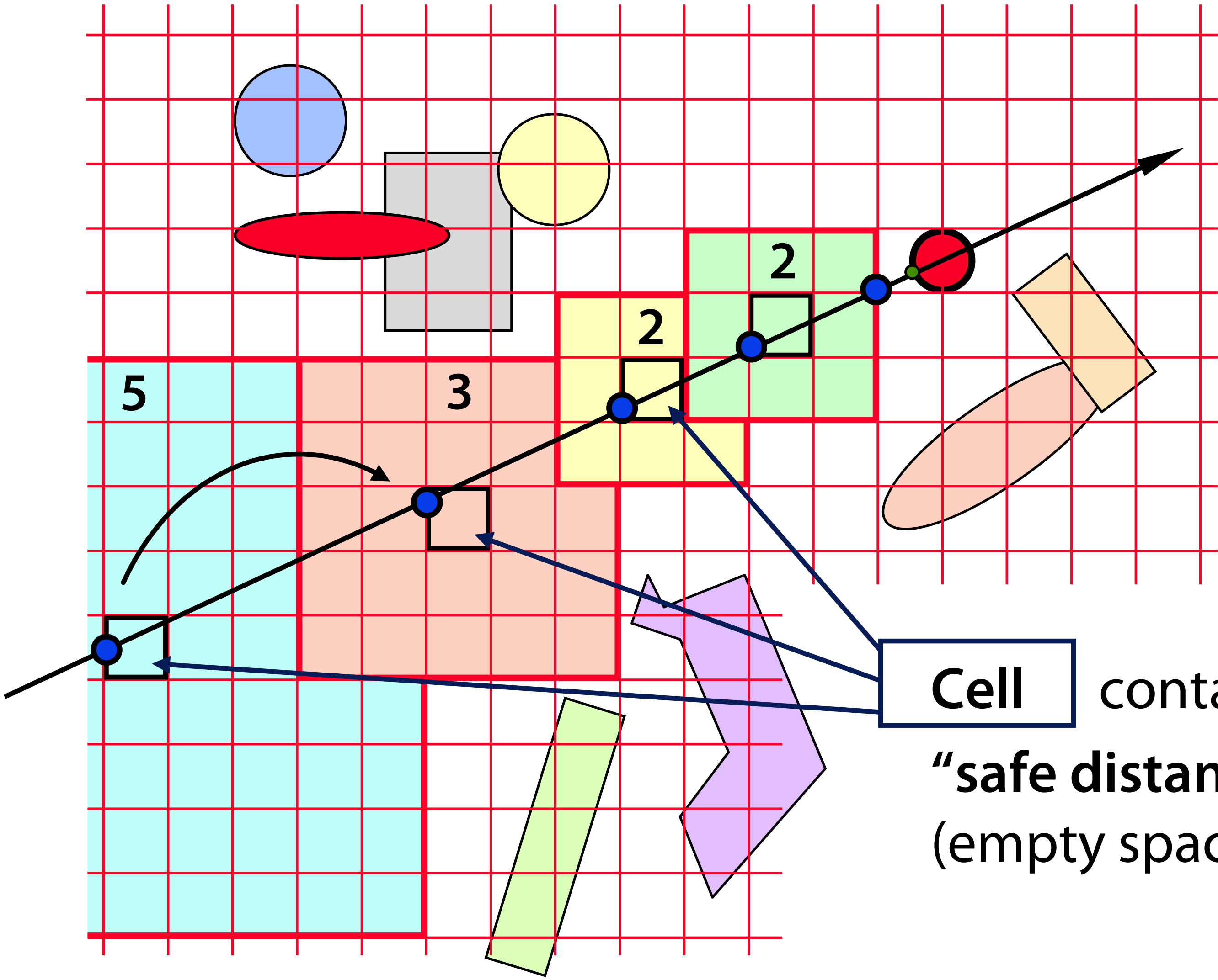
- heap – list of potentially intersected cells (ordered from the most promising ones)

“Mailbox” technique



The intersection must be in the current cell (otherwise it is postponed)

Macro-cells (Miloš Šrámek)



Cell contains **"safe distance"** (empty space)

Directional acceleration techniques

Utilizing **directional cube**:

Light buffer

– speeding up shadow rays to point light sources

Ray coherence

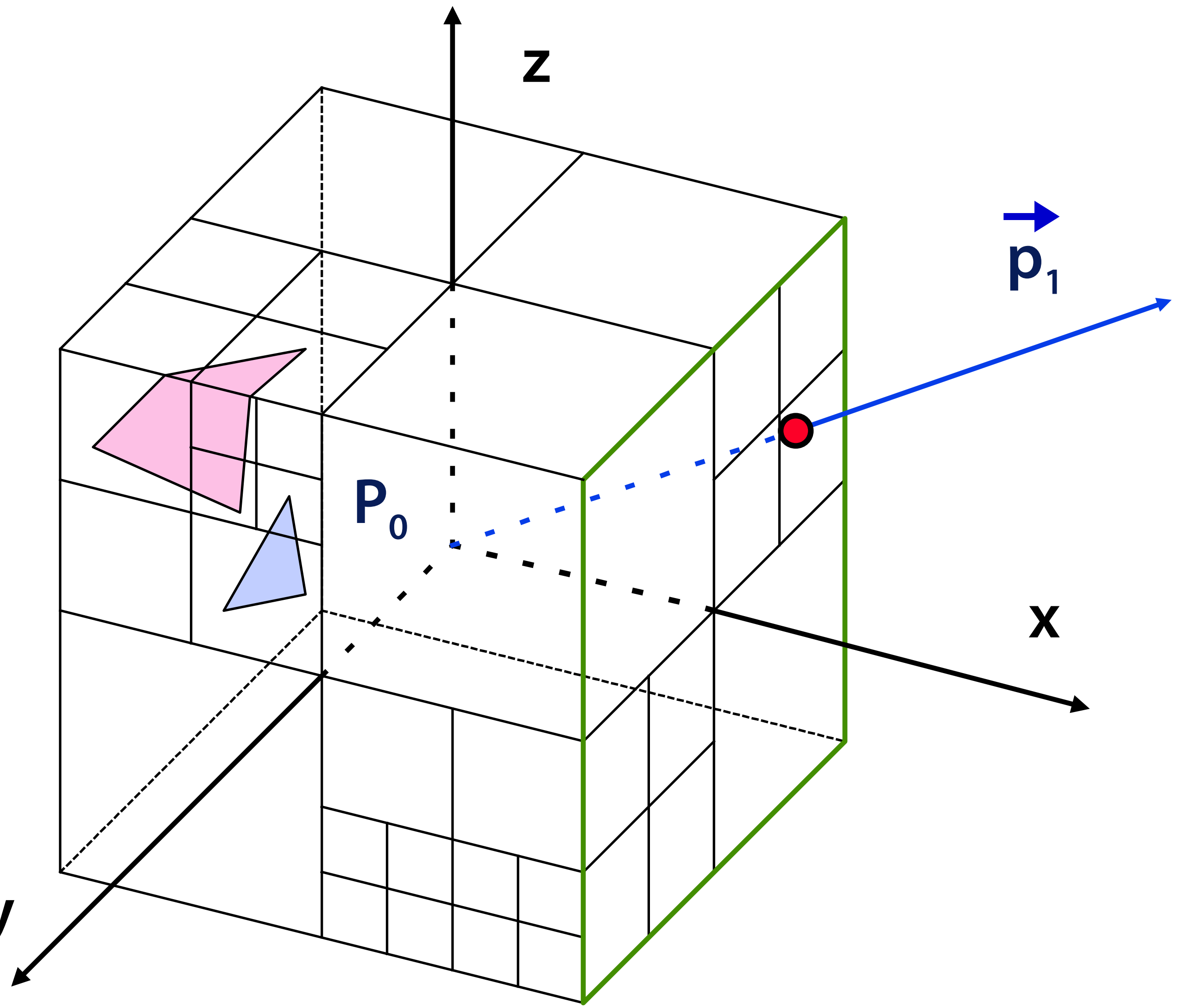
– for all secondary rays

5D **ray classification**

Image plane directory (visibility precomputation)

– only for primary rays

Directional cube (adaptive)



Directional cube

Axis-oriented

Cube faces divided into **cells**

- uniform or adaptive division
- every cell stores list of relevant objects (can be ordered by the distance from the cube)

HW rasterization and visibility (depth-buffer) can be used for uniform division

Light buffer

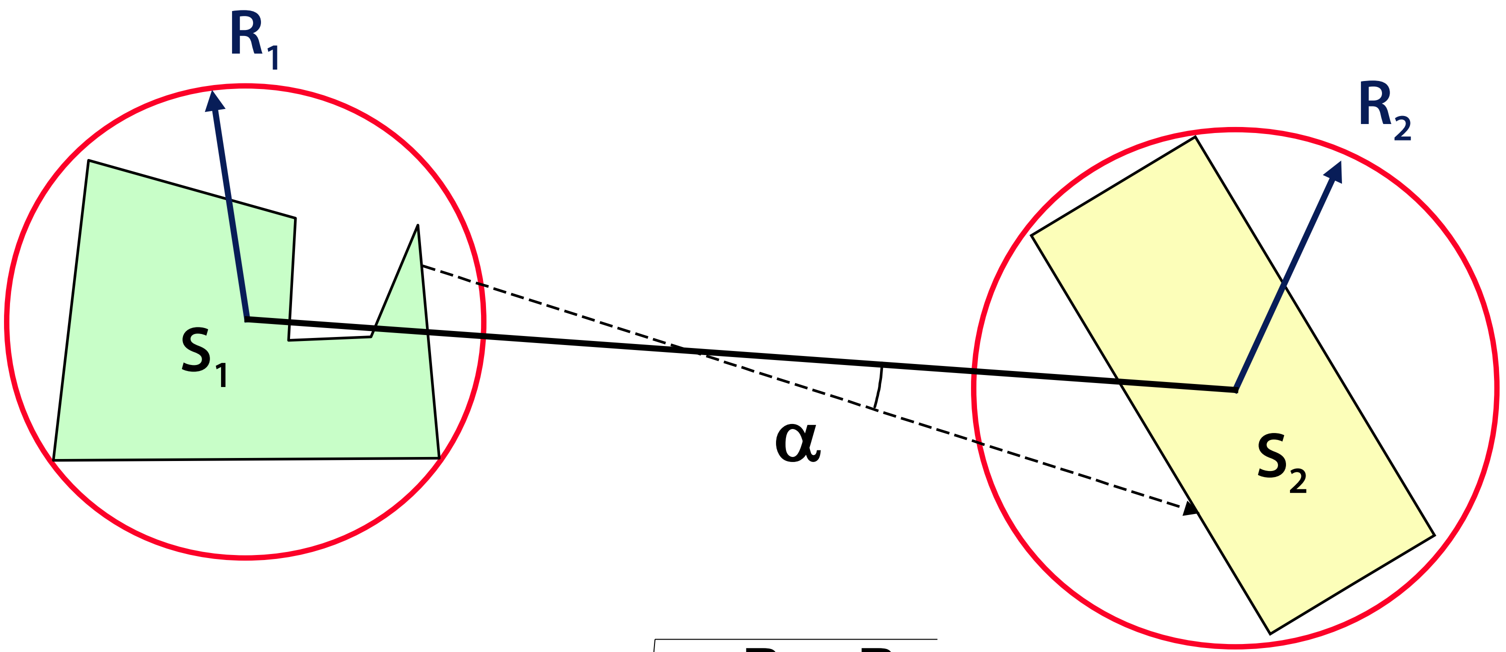
Speeding up **shadow rays**

Directional cube in every point light source

- possible visibility of objects from the light-source point
- some cells might be covered completely by one object (everything else is in shadow)

For a **shadow ray** only objects projected in the relevant cell are considered

Ray coherence



$$\cos\alpha \geq \sqrt{1 - \frac{R_1 + R_2}{\|S_1 - S_2\|}}$$

Using coherence

For every **secondary rays**

- reflected, refracted, shadow

Assumed bounding solid: **sphere**

Directional cube placed in every **center of bounding sphere**

- list of projected objects/light sources in every cell
  - coherence condition is used
  - lazy evaluation!
- lists can be ordered by distance from the cube

5D ray space

Rays in 3D scene

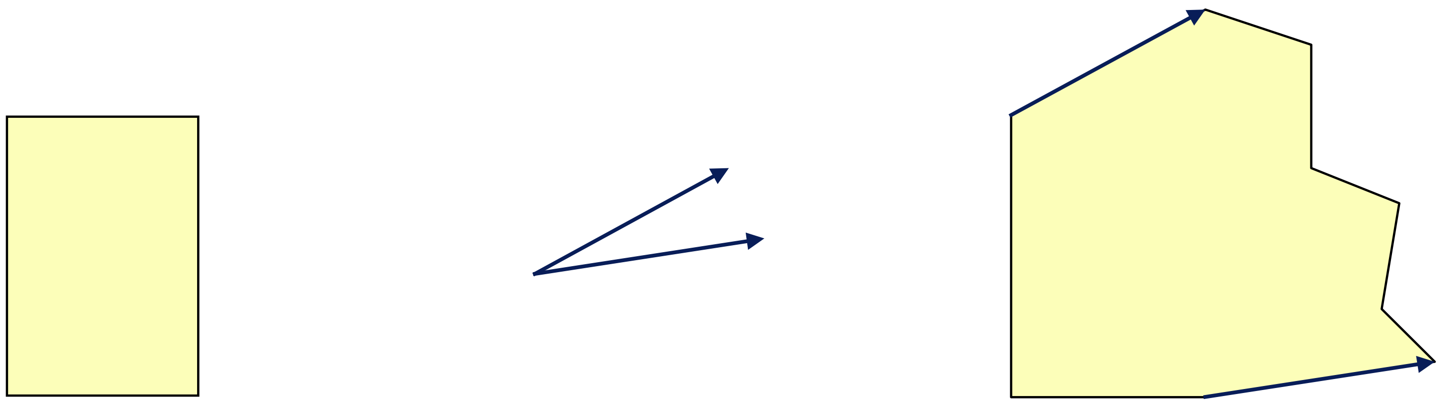
- **origin** P_0 **[x, y, z]**
- **direction** [φ, θ]

5D hypercube divided into **cells**

- every cell contains list of possible intersections for the associated ray pencil ("beam")
- adaptive subdivision (merging neighbour cells with equal or similar lists)

6D variant – one more quantity (time) for animations

Ray classification



origin (2-3D) + direction (1D, 2D) = bundle / pencil

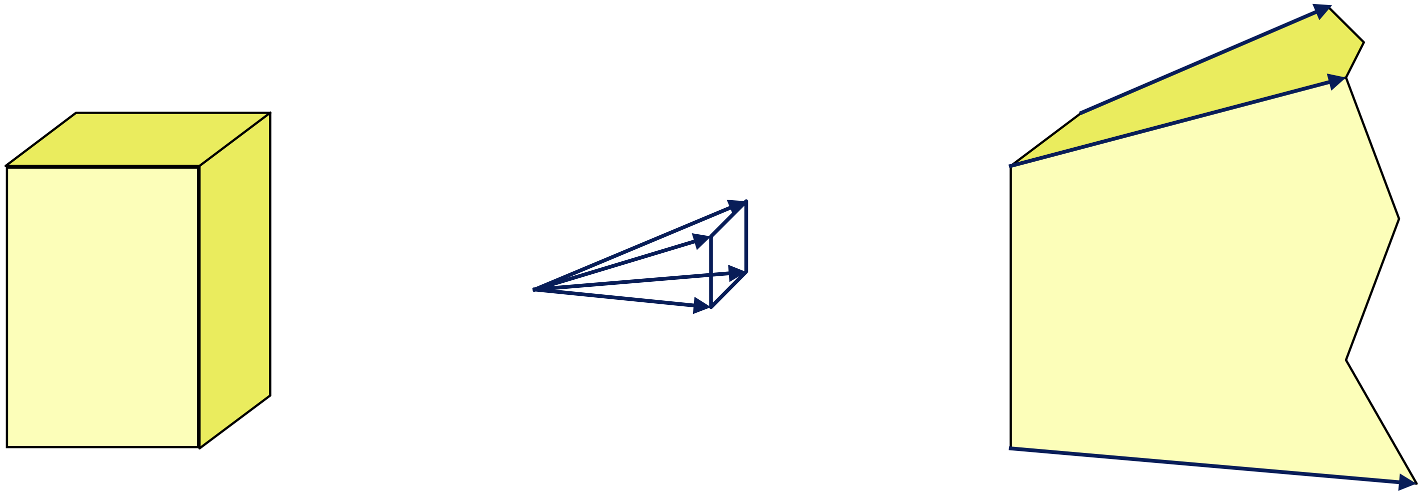


Image plane directory

For **primary rays**

Projection plane is (adaptively) divided into **cells**

- possible visibility of individual objects in a cell (together with order)
- complete coverage by one cell by one object is possible (hard to test)

Robust variant **of used visibility method**

- in most pixels it can be done with complete certainty

Generalized rays

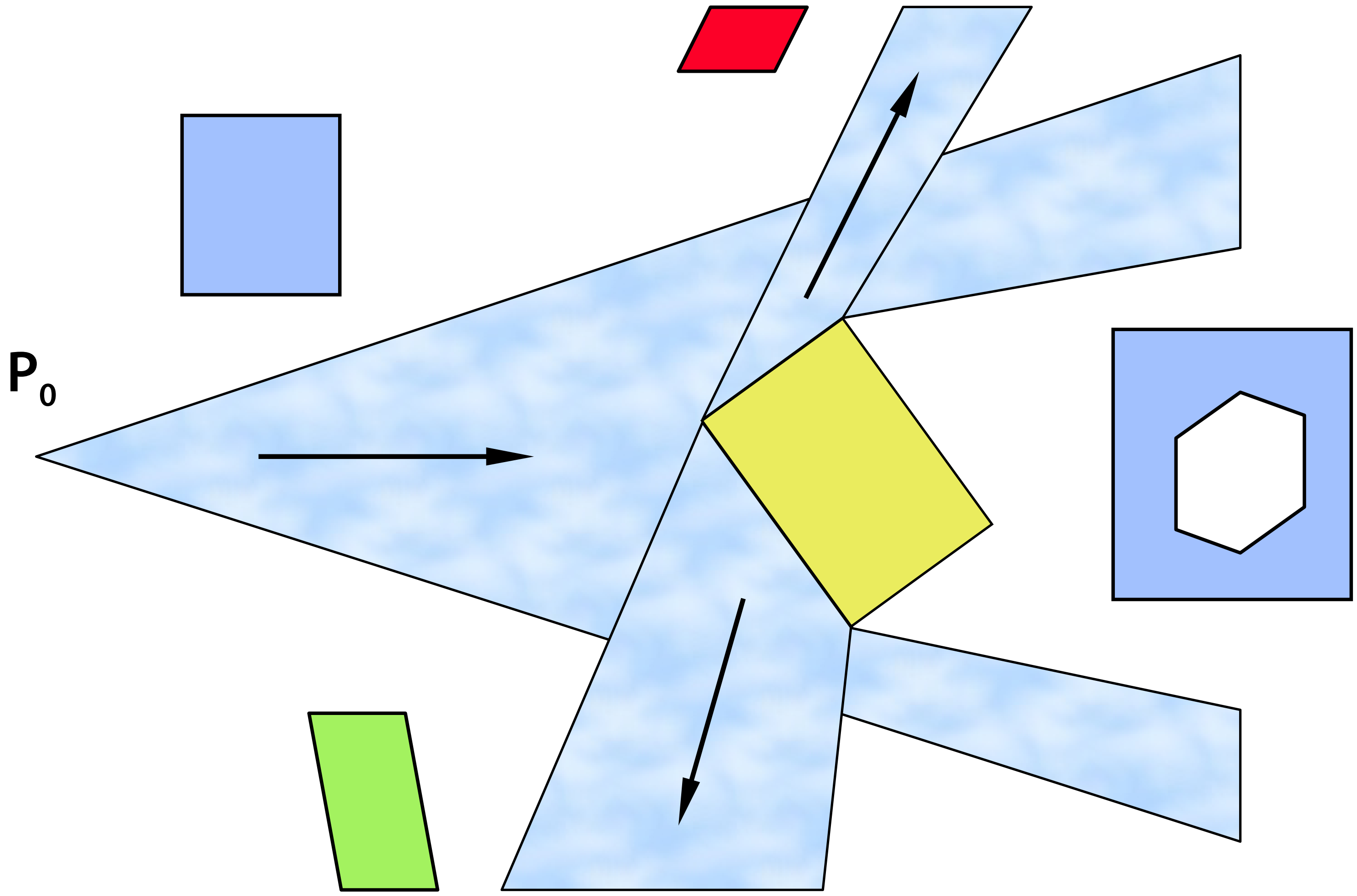
Computing more information about **f(x,y)**

- for anti-aliasing (average color estimation) or soft shadows (shadow ratio)
- some restrictions to a scene are necessary

Forms of **generalized rays**

- rotational or elliptical cone, regular pyramid
- pyramid with polygonal cross section (polygonal scene, see the next slide)

Polygonal scene



RT acceleration on GPU

BVH hierarchy used in **rendering**

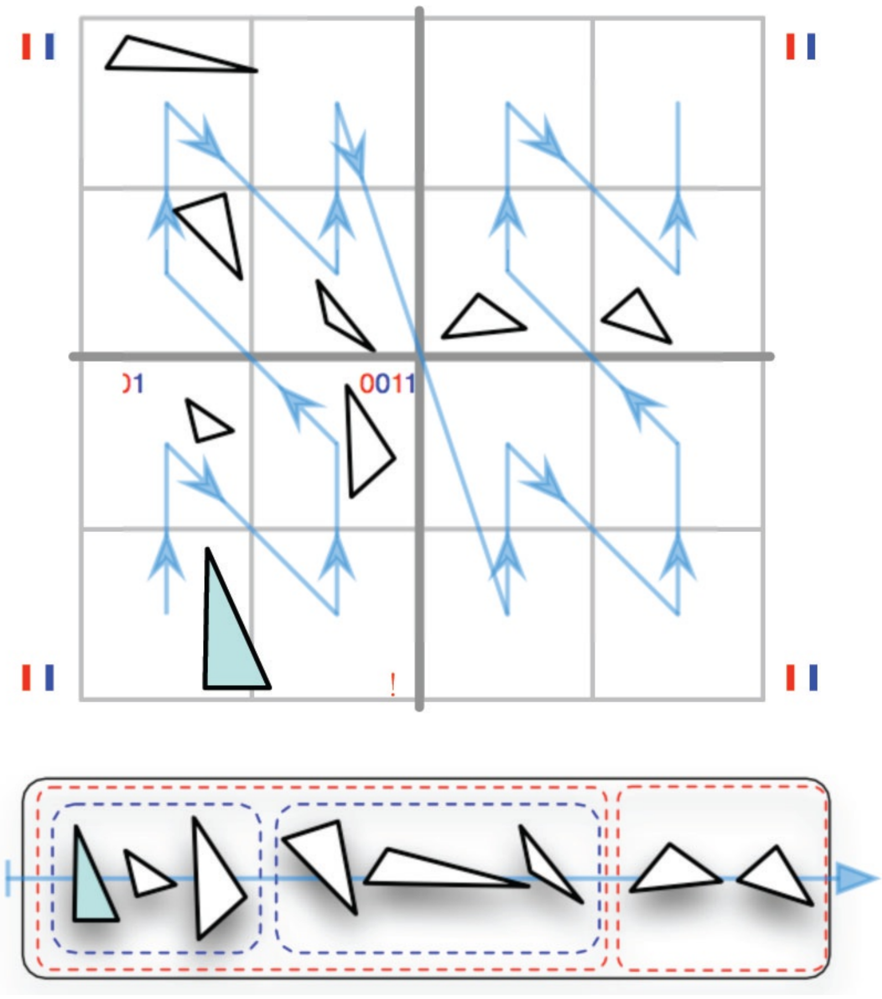
- RTX cores (NVIDIA)

BVH **construction** can be done on GPU as well

- CUDA cores (NVIDIA)

Morton Codes or Extended Morton Codes (linearization)

- Lauterbach et al (2009) or Vinkler et al (2017)
- LBVH construction – parallel!



Morton code based BVH tree

Grid $2^k \times 2^k \times 2^k$

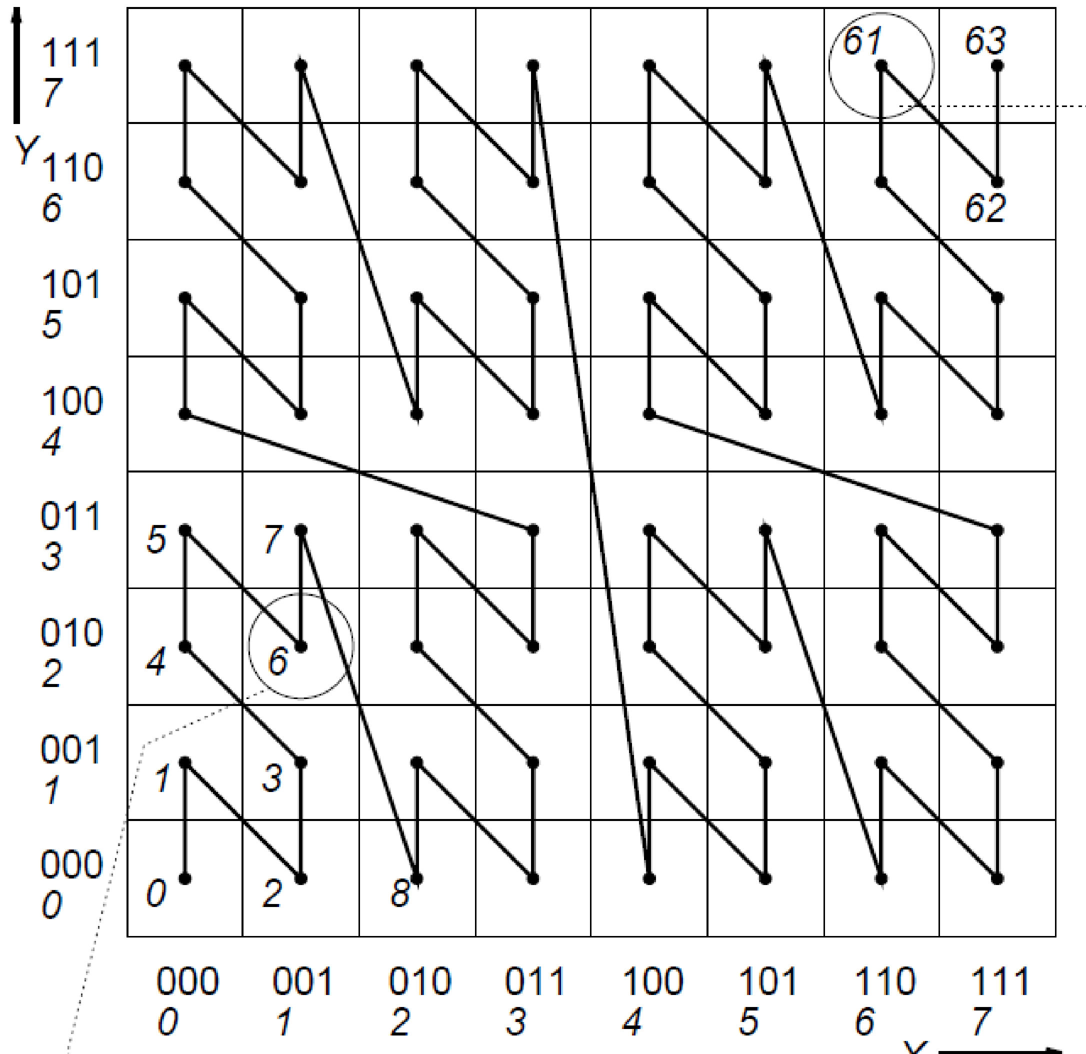
- 3k bits [x:k, y:k, z:k]
- direct point localization

Interleaving **x, y** and **z** into one **3k bit code**

- 2D example

Sorting triangle codes

- parallel radix-2 sort
- tree node = interval of indices $[\mathbf{l}_i, \mathbf{r}_i)$



Two examples of Morton-codes:

x=001, y=010 -> xy=000110 (decimal 6)

x=110, y=111 -> xy=111101 (decimal 61)

Parallel radix sort on GPU (CUDA)

Split and **Compaction** kernels

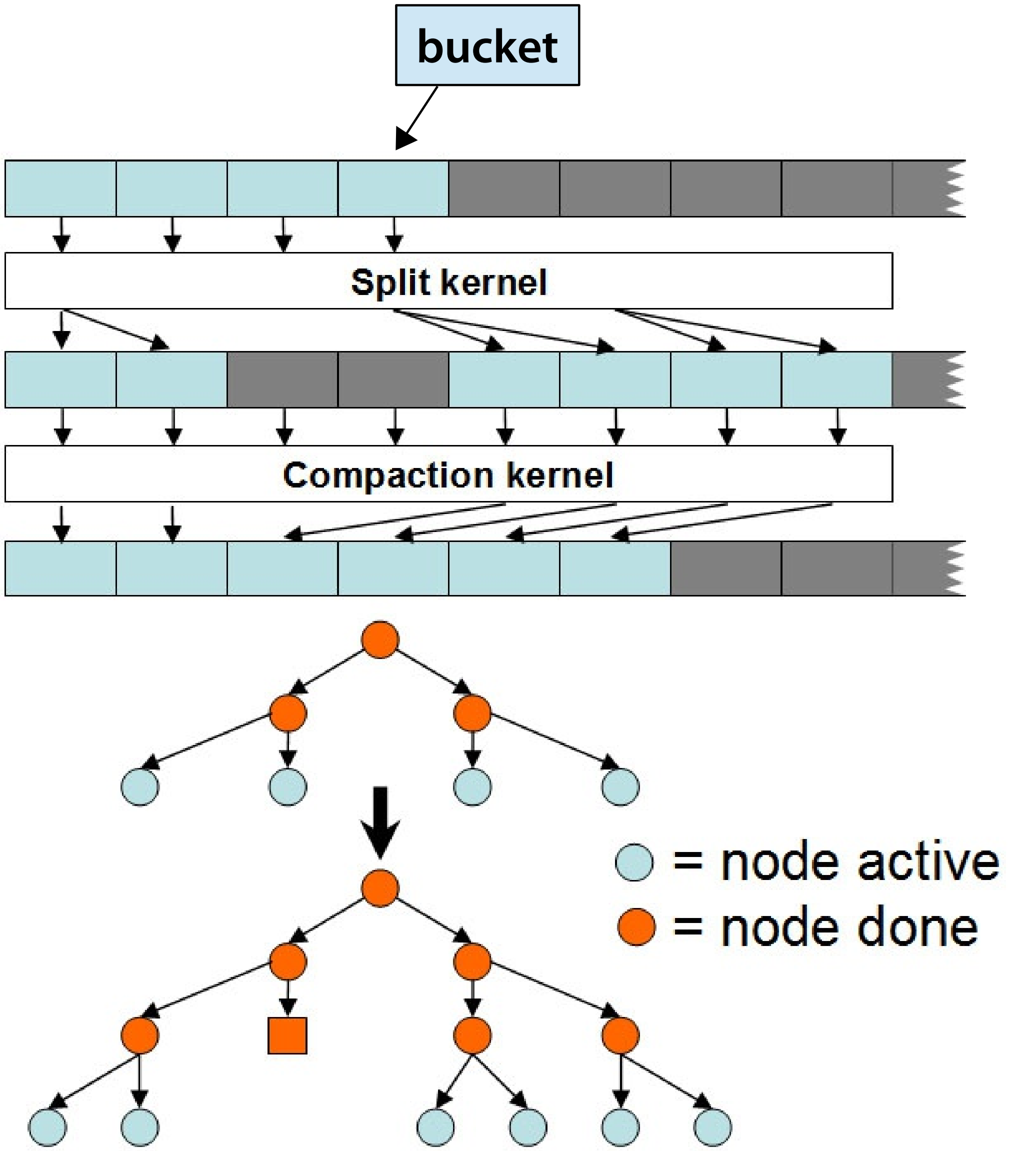
Split: two work queues

- input [m] → output [2m]
- all Splits **in parallel**
  - » read(i) → write(2i)
  - » read(i) → write(2i + 1)

Compaction:

- the **output** queue is compacted back to the **input** queue

Overall **O(n log n)**



Data-Parallel SAH split

1. Determine the **best split position**

- **3k** threads … **k** uniform split candidates for each axis
- if number of triangles is not greater than **k**, triangle positions are used (= full SAH evaluation)
  - » testing triangle by triangle, updating AABBs
  - » at the end, the thread computes its cost
  - » parallel_reduction(min) determines the best option
    - **$\log_2 k$** steps (binary reduction using `syncthreads()`)

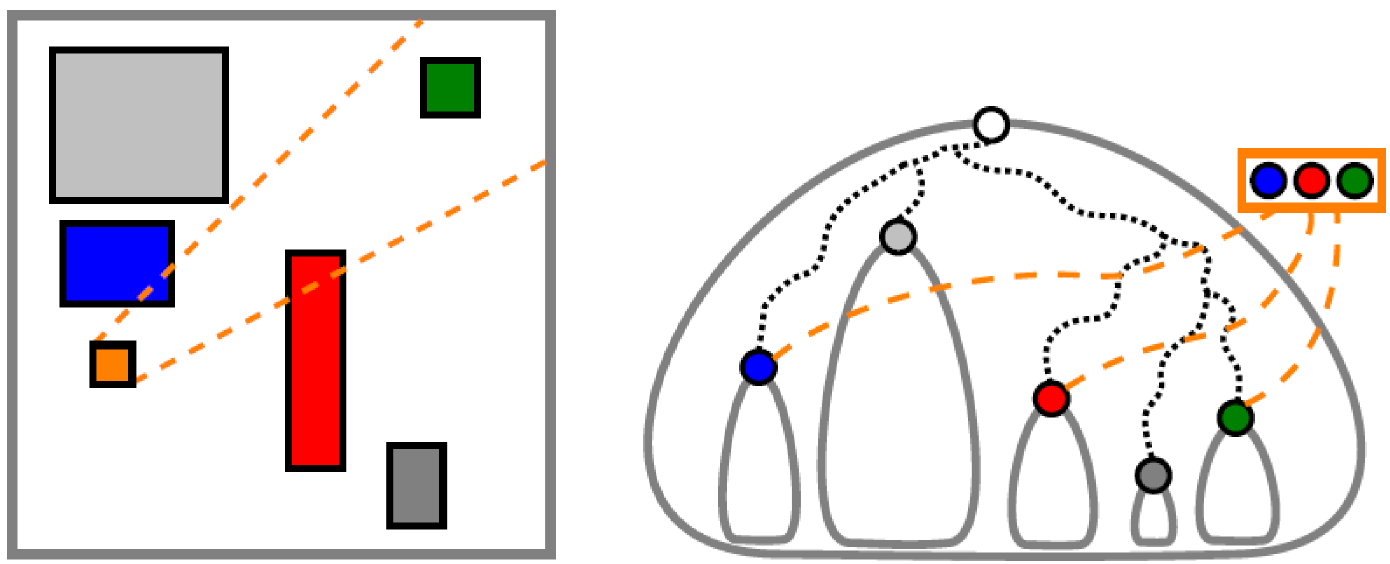
2. Reordering the primitives

- no splitting of triangles → in place operation, using **indices** only
  - » 1/0 classif. of every triangle in the block, parallel sum → node size
  - » creating the nodes and their AABBs

Contemporary methods

J. Hendrich: ***Adaptive Acceleration Techniques for Ray Tracing***, PhD thesis, ČVUT, 2023

- rearranging the BVH
- 5D shafts

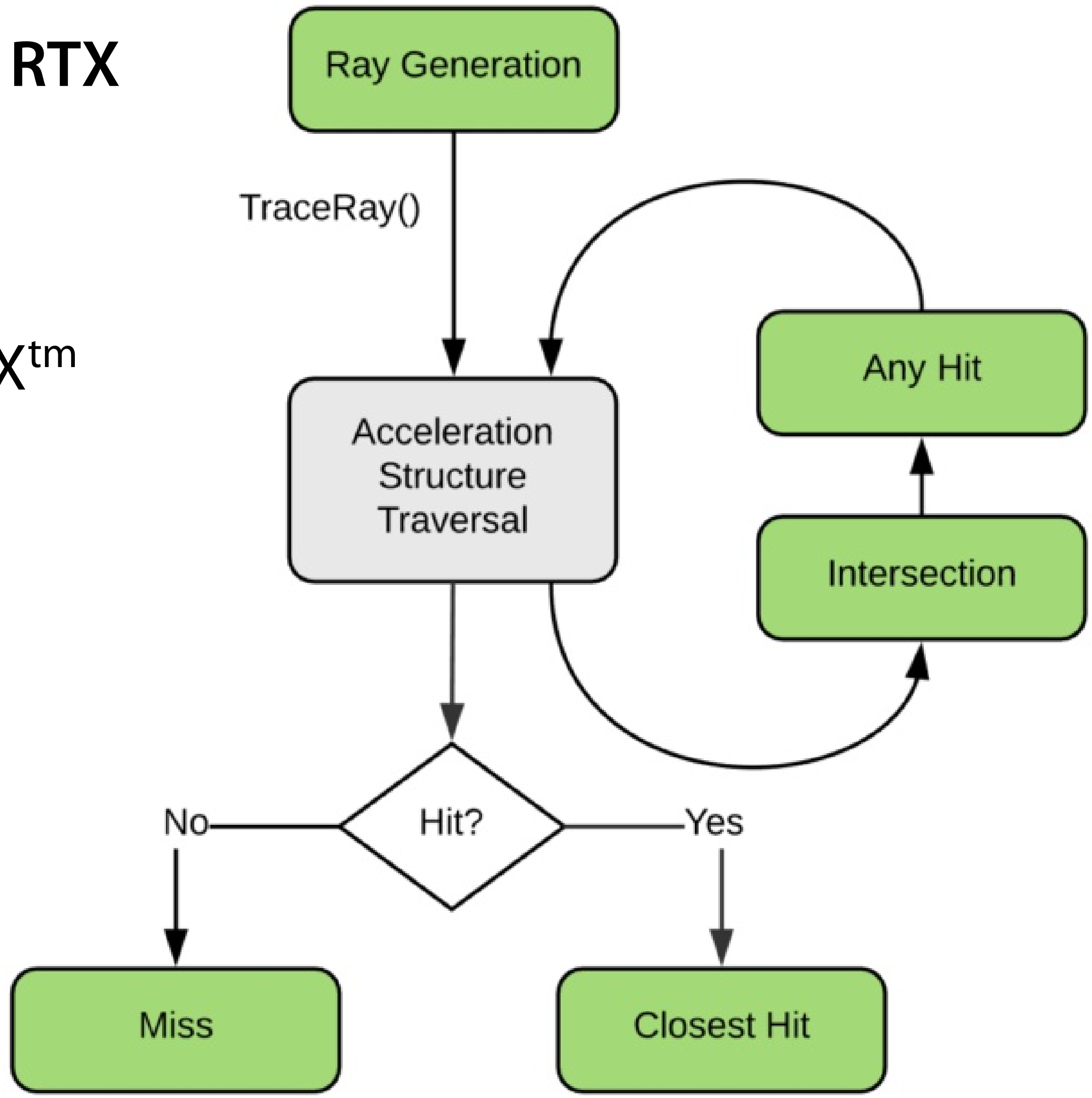


DirectX Raytracing (DXR)

Built on top of **NVIDIA RTX**

- Volta and later GPUs
- DirectX 12

Similar to NVIDIA OptiX^{tm}



© 2018, NVIDIA

DirectX Raytracing (DXR)

Ray Tracing Pipeline with new shader types

- **Ray Generation** (à la Compute)
  - 2D grid scheme, no thread groups, no barriers
- **Intersection**
  - custom shape, default = triangle. No payload access (geometry only)
- **Any Hit**
  - **no guaranteed traversal order**!
  - can terminate ray, can modify payload, for transparency…
- **Closest Hit / Miss**
  - called **after** all „any hits“
  - can read/write payload
  - may call **TraceRay()** … recursion (**ray tracing**)

Ray Generation shader example

```
// An example payload struct. We can define and use as many different ones as we like.
struct Payload
{
    float4 color;
    float  hitDistance;
};

// The acceleration structure we'll trace against. This represents the geometry of our scene.
RaytracingAccelerationStructure scene : register(t5);

[shader("raygeneration")]
void RayGenMain()
{
    // Get the location within the dispatched 2D grid of work items
    // (often maps to pixels, so this could represent a pixel coordinate).
    uint2 launchIndex = DispatchRaysIndex();

    // Define a ray, consisting of origin, direction, and the t-interval we're interested in.
    RayDesc ray;
    ray.Origin = SceneConstants.cameraPosition.
    ray.Direction = computeRayDirection(launchIndex); // assume this function exists
    ray.TMin = 0;
    ray.TMax = 100000;

    Payload payload;

    // Trace the ray using the payload type we've defined.
    // Shaders that are triggered by this must operate on the same payload type.
    TraceRay(scene, 0 /*flags*/, 0xFF /*mask*/, 0 /*hit group offset*/,
             1 /*hit group index multiplier*/, 0 /*miss shader index*/, ray, payload);

    outputTexture[launchIndex.xy] = payload.color;
}
```

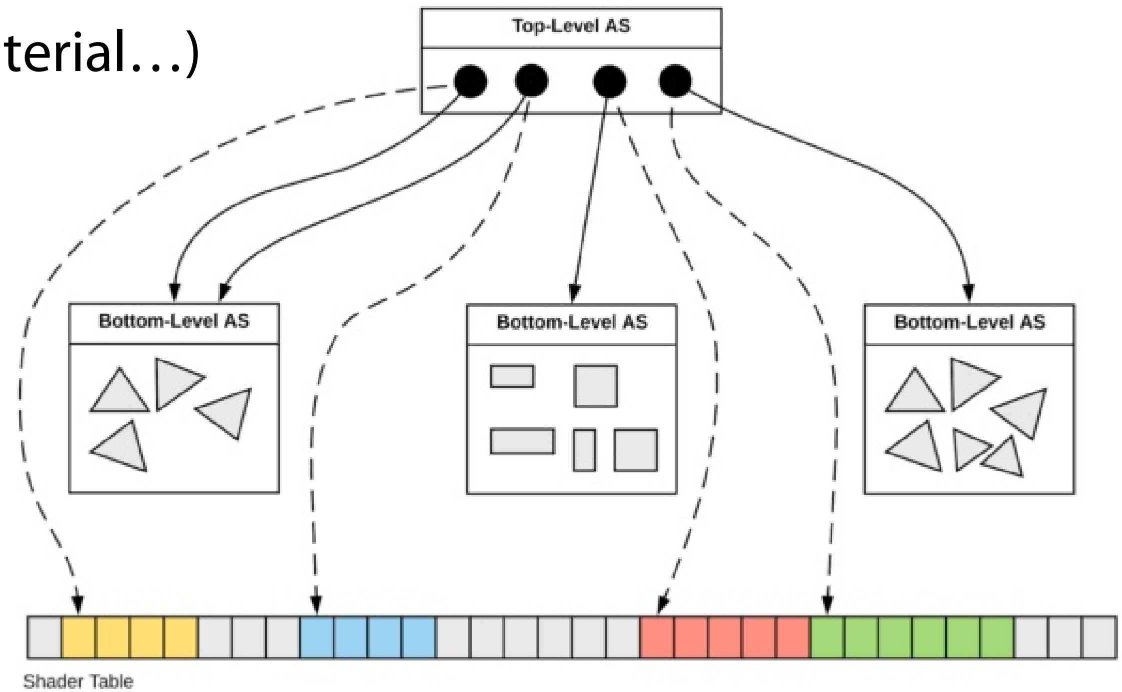
Closest Hit shader (color ←barycentric)

```
// Attributes contain hit information and are filled in by the intersection shader.
// For the built-in triangle intersection shader, the attributes always consist of
// the barycentric coordinates of the hit point.
struct Attributes
{
    float2 barys;
};

[shader("closesthit")]
void ClosestHitMain(inout Payload payload, in Attributes attr)
{
    // Read the intersection attributes and write a result into the payload.
    payload.color = float4(attr.barys.x, attr.barys.y,
                           1 - attr.barys.x - attr.barys.y, 1);

    // Demonstrate one of the new HLSL intrinsics: query distance along current ray
    payload.hitDistance = RayTCurrent();
}
```

DXR acceleration

Two level structure

- **Top-level AS** are for animation (easy rebuild)
- **Bottom-level AS** represent scene geometry (trees over triangles)
- **Instance descriptor** connects a BLAS to TLAS
  - » transformation matrix
  - » shader table offset (material…)



„Reflections“ demo (2018)

UE4, Epic, ILMxLAB, and NVIDIA

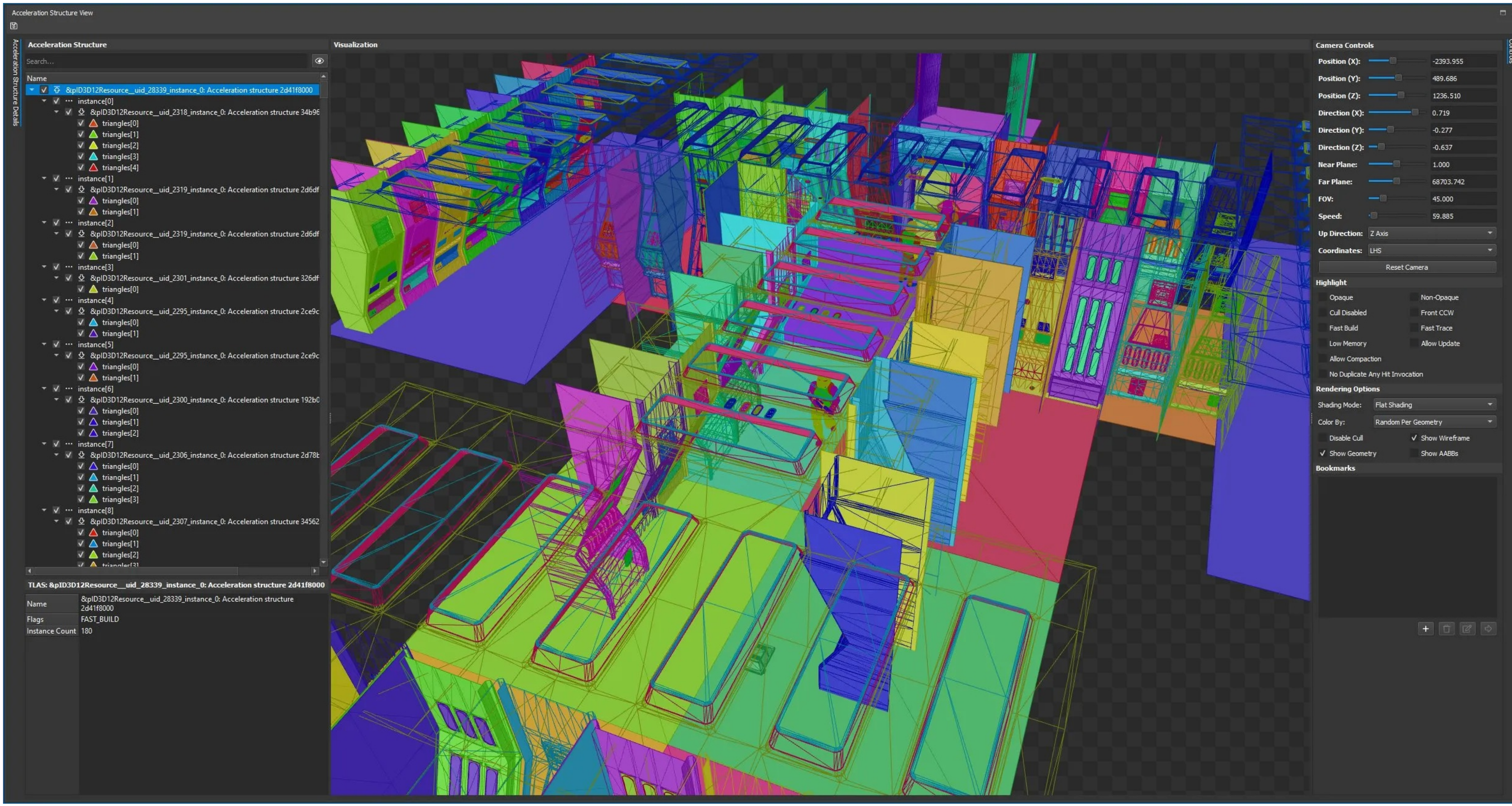
- https://www.youtube.com/watch?v=lMSuGoYcT3s
- the 1st demo of real-time ray tracing in Unreal engine
- „Cinematic Lighting in Unreal Engine“, talk at GDC 2018

Hardware

NVIDIA DGX Station (to achieve 24fps@1080px)

- 4x watercooled **Tesla V100**
- 4-way NVLINK GPU interconnect

Scene

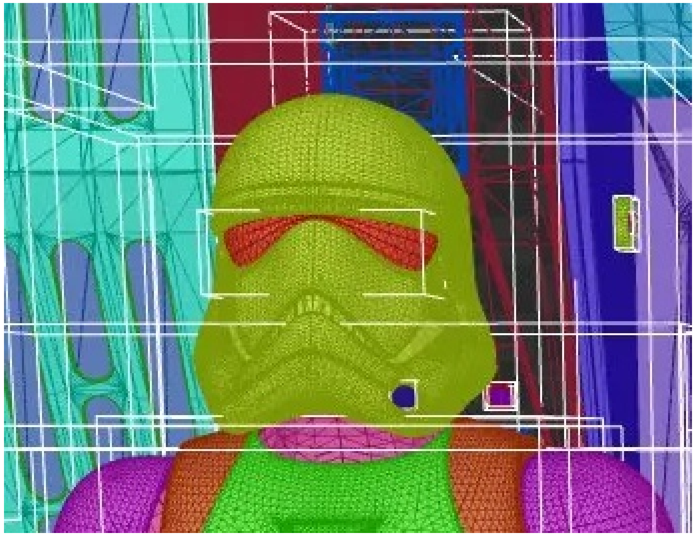


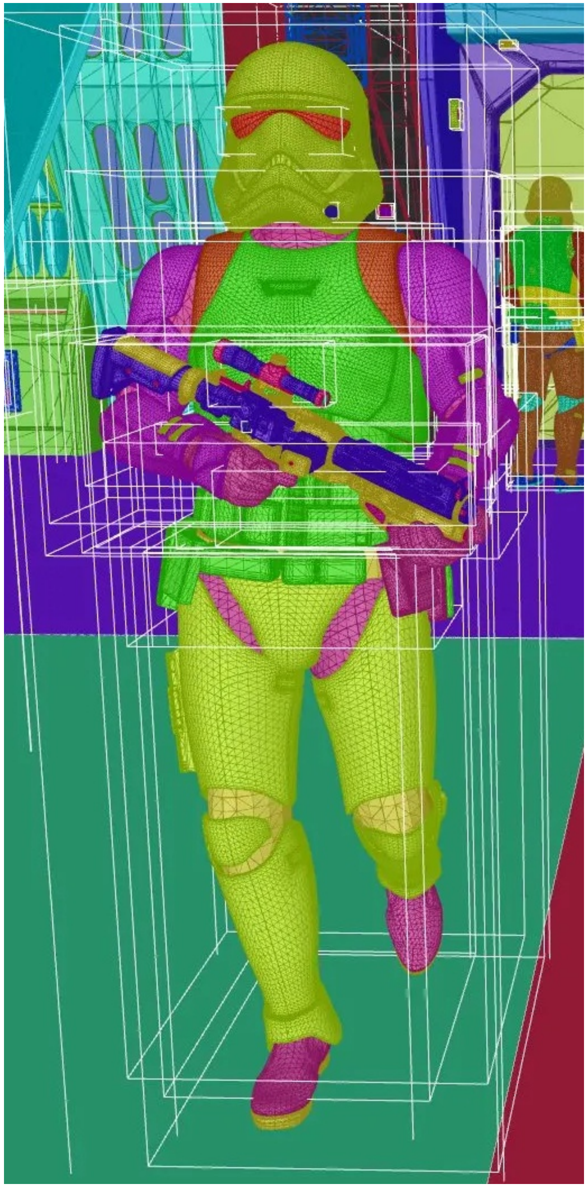
© clamchowder, Chips and Cheese

Scene and AABBs

Total: 3,447,421 triangles

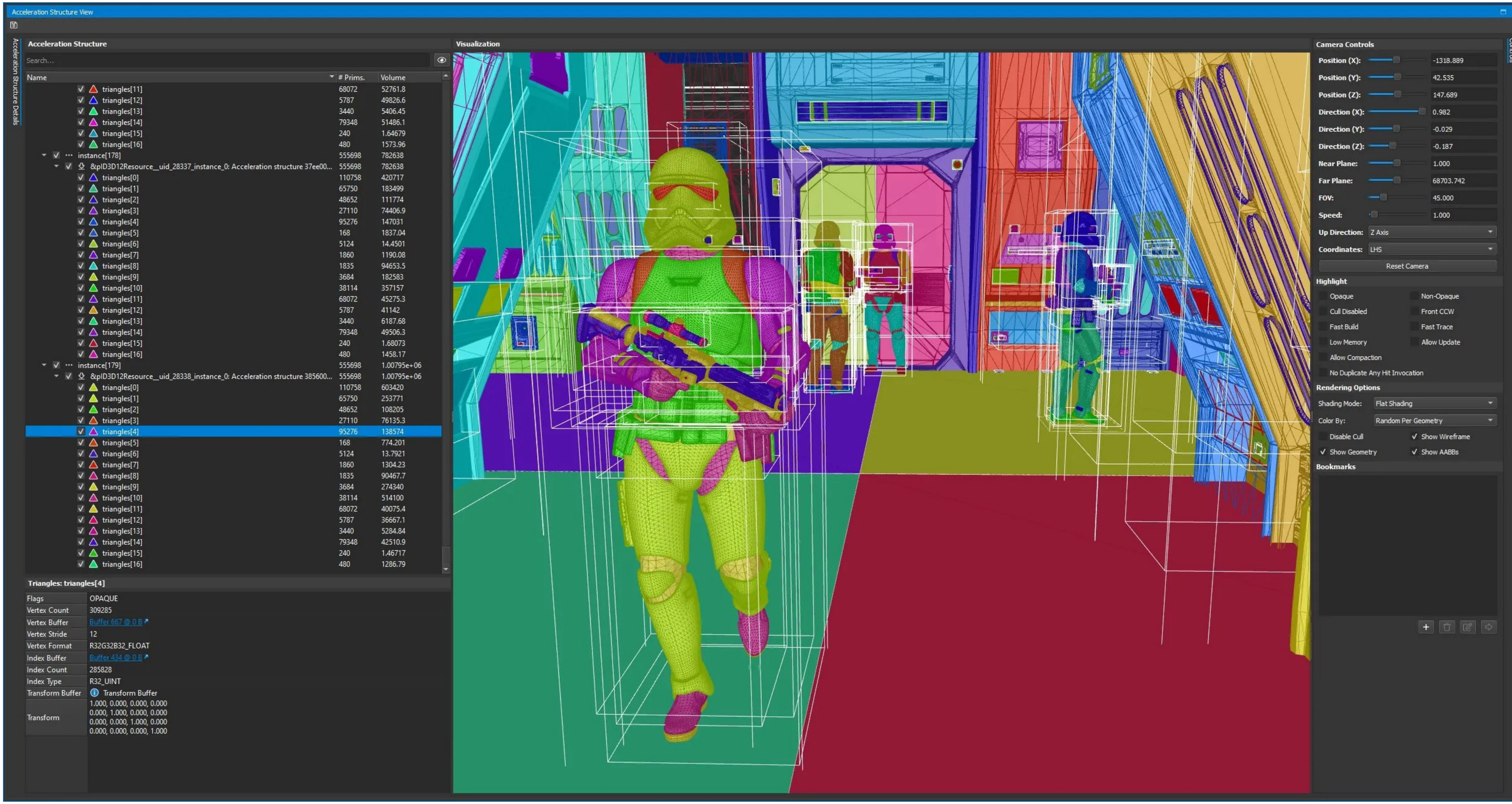
- one stormtrooper: 555k triangles in 17 sub-boxes
- the helmet alone: 110k triangles





© clamchowder, Chips and Cheese

BVH in action



© clamchowder, Chips and Cheese

Literature I

A. Glassner: ***An Introduction to Ray Tracing***, Academic Press, London 1989, 201-262

A. Watt, M. Watt: ***Advanced Animation and Rendering Techniques***, Addison-Wesley, Wokingham 1992, 233-248

V. Havran: ***Heuristic Ray Shooting Algorithms***, PhD thesis, FEL ČVUT Praha, 2001

I. Wald, V. Havran: ***On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)***, IEEE Symposium on Interactive Ray Tracing, 2006

I. Wald: ***On fast Construction of SAH-based Bounding Volume Hierarchies***, IEEE Symp. on Inter. Ray Tracing, 2007

Literature II

A. Benthin, S. Woop, I. Wald, A. Afra: *Improved Two-Level BVHs using Partial Re-Braiding*, HPG'17, Los Angeles, 2017

M. Vinkler, J. Bittner, V. Havran: *Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction*, HPG'17, Los Angeles, 2017

D. Meister, J. Bittner: *Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing*, JCGT, vol. 11, No. 3, 2022

- https://github.com/meistdan/hippie

H. Samet: *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, 2006