Computational Geometry

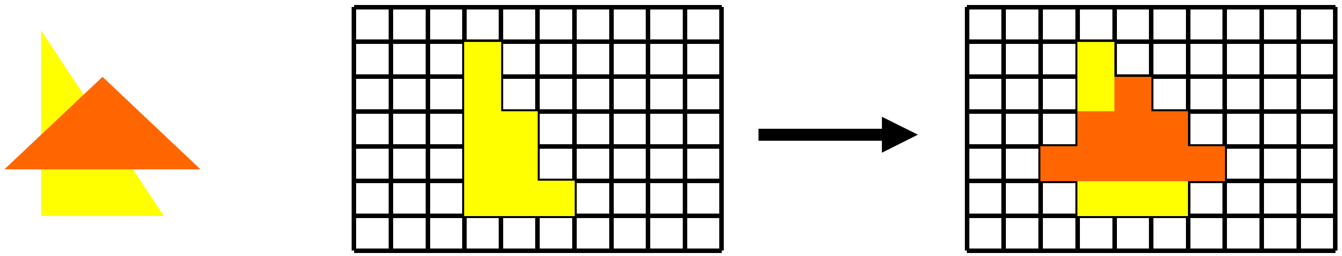
3. Binary Space Partitions (BSP-Trees)

3.1 Painter's Algorithm

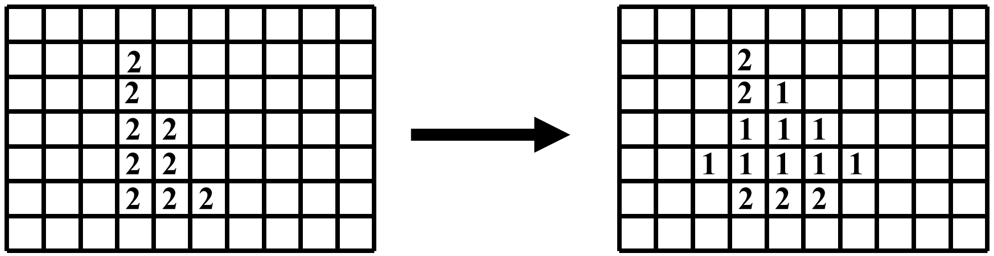
- Real time rendering of complex scenes is important for computer games but also for technical applications like flight simulators.
- One important aspect is here the visibility of objects.
- For the rendering complex scenes are partitioned into polygons (usually triangles), that are processed in the rendering pipeline of the graphical processing unit (GPU).

3.1 Painter's Algorithm

- The visibility problem is often solved with a z-buffer in image space, i.e. for every pixel.



Frame-Buffer (RGB color)



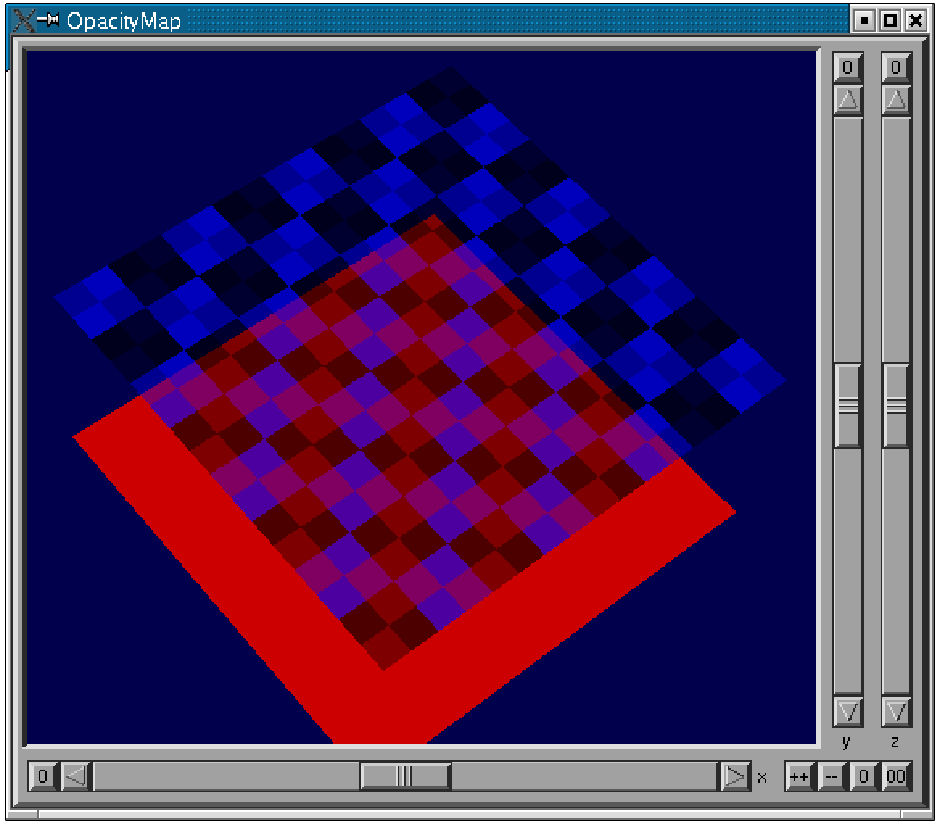
z-buffer (z-coordinate = depth)

3.1 Painter's Algorithm

- Advantages of the z-buffer:
  - No sorting of the triangles necessary.
  - Independent of the complexity of the scene.
  - Implemented in hardware in today's GPUs.
- Disadvantages of the z-buffer:
  - It uses additional memory, e.g. 1600 x 1400 x 16 Bit = $4{,}48$ MByte.
  - For every pixel the z-coordinate of every overlapping polygon must be compared.
  - For transparent objects the image must be composed front-to-back or back-to-front requiring sorting.

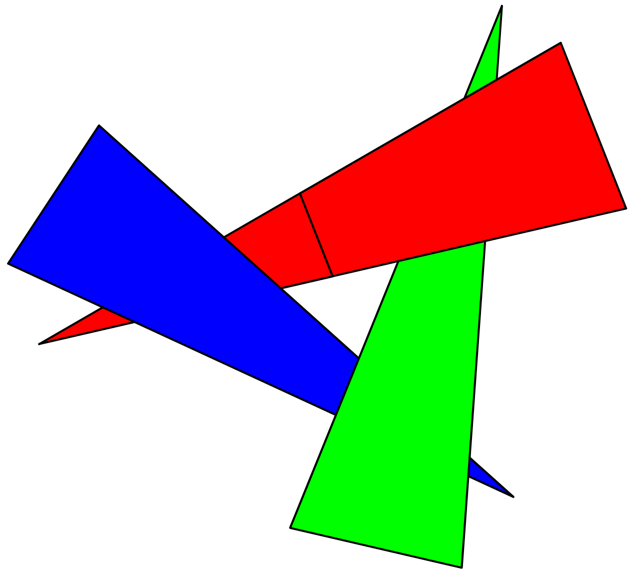
3.1 Painter's Algorithm

- Using textures and opacity-maps, that control the transparency of individual objects, the objects must be sorted by z-coordinates.



3.1 Painter's Algorithm

- Painter's Algorithm [3] constructs an image from back to front, like painting first the background and then adding objects sorted from back to front to the image.
- But:
  - The sorting depends on the view point and must be recomputed for every view point.
  - Cyclic overlaps of objects can only be sorted correctly, if at least one object is split into fragments.
- What can be done to speed this process up?

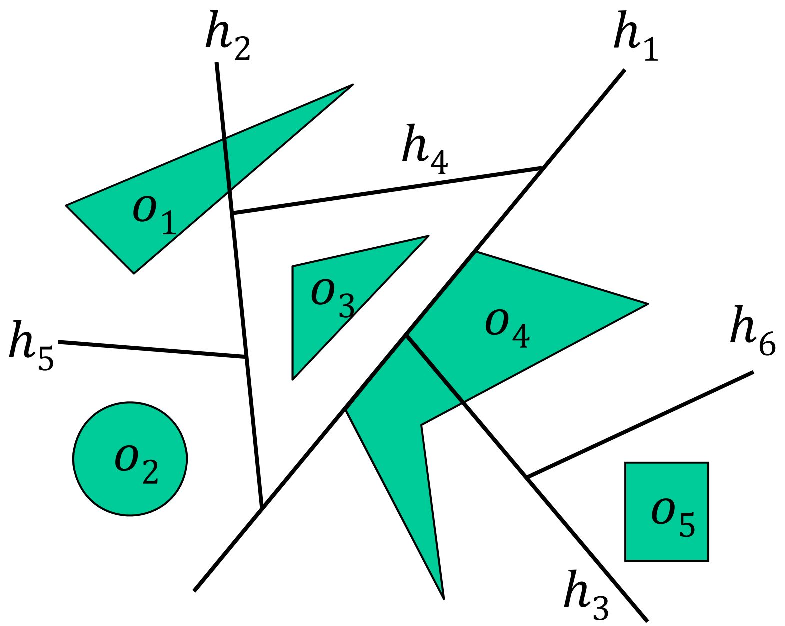


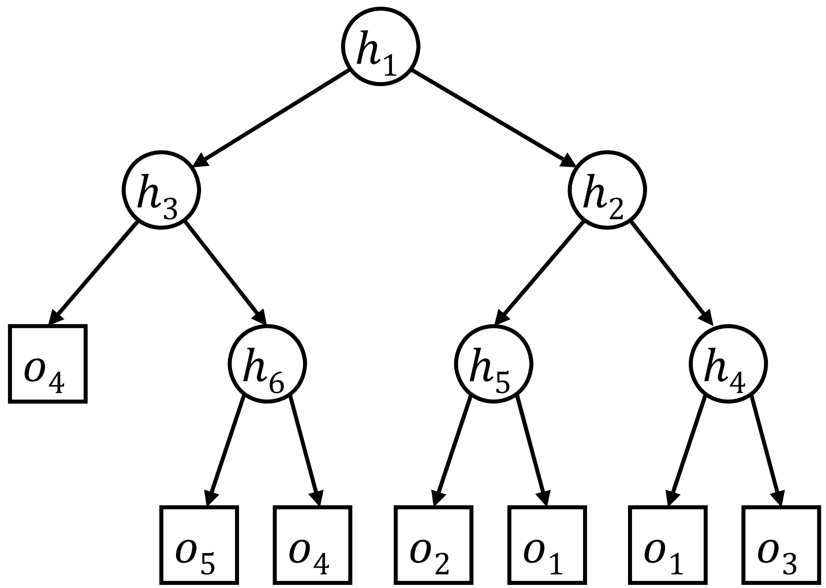
3.2 BSP-Trees

- A data structure to sort and split planar polygons is a *Binary Space-Partitioning Tree* (*bsp-tree*).
- A BSP-tree partitions $\mathbb{R}^d$ recursively along arbitrary hyper-planes (($d-1$)-dimensional affine sub-spaces).
  - Often these hyper-planes are defined by edges or faces of individual objects, yielding the so-called *auto-partitioning*.
    - In 2D only edges (line segments) of objects need to be partitioned.
  - Every inner node has two children: in- or outside the hyper-plane.
  - Every inner node contains a splitting line and possibly the ($d-1$)-dimensional objects contained in that line.
  - The leaves partition $\mathbb{R}^d$ into the so-called *bsp-partition*.
    - The leaves contain the faces of the bsp-partition and the object fragments within this face.

3.2 BSP-Trees

Example:





3.2 BSP-Trees

- The hyper-plane h of an inner node is determined by a point $p_h \in \mathbb{R}^d$ and a normal vector $n_h \in \mathbb{R}^d$ partitioning the space into two half-spaces:

$$h^+ \coloneqq \{q \mid (q - p_h) \cdot n_h > 0\},$$
$$h^- \coloneqq \{q \mid (q - p_h) \cdot n_h < 0\}.$$

- During the recursive construction of a bsp-tree the objects are partitioned into the two half-spaces corresponding to the two child nodes.
  - Objects belonging to both half-spaces are split into fragments.
  - Objects completely contained in h are stored in a list corresponding to the node.

3.2 BSP-Trees

Proposition 1

With a bsp-tree of size m the correct sorting of polygons along a given (view) direction z can be computed in $O(m)$.

Proof

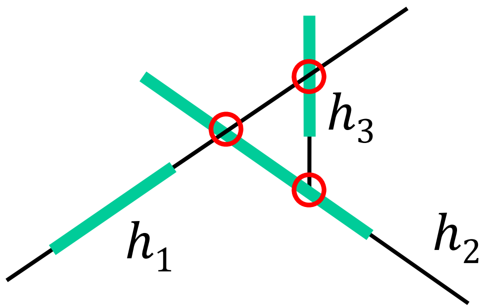
- The sorting is generated traversing the sub-trees in the sequence

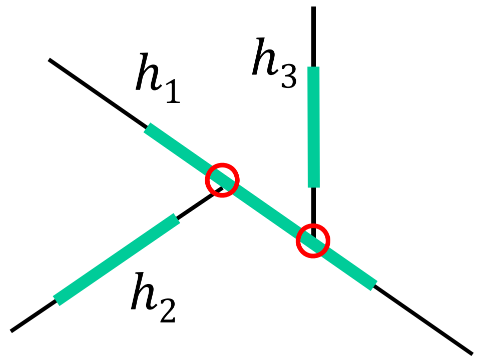
$$\begin{cases} h^+, h, h^-, & \text{if } z \cdot n_h > 0 \\ h^-, h, h^+, & \text{otherwise} \end{cases}.$$

- Here, every node is visited only once.
- Because of the splits there are no cyclic overlaps. $\square$

3.2 BSP-Trees

- Every inner node corresponds to a region of $\mathbb{R}^d$, defined by the intersection of all half-spaces of nodes above.
- The size of a bsp-tree, i.e. the number of its nodes, depends linearly on
  - the number of objects plus
  - the number of fragmentations of objects, which depends on the sequence of splits.
- Example:





3.3 Construction in 2D

- How to construct a bsp-tree with minimal (or small) size?
- Using auto-partitioning the hyper-planes are given by the objects and the size depends only on the sequence of splits.
- First only the 2D case:
  - For a given set S of non-intersecting line segments in the plane, construct the BSP-tree with as few as possible inner nodes.
- We take a randomized approach:

3.3 Construction in 2D

Algorithm 1: BSP2D(S)

Input: A set $S = \{s_1, \dots, s_n\}$ of **uniformly distributed**, non-intersection line segments in the plane.

Output: Root of a bsp-tree representing S.

v = new node;
if ($|S| \leq 1$) **then** $v.S = S$;
else {
  h = line through s_1;
  $v.S = \{s \in S : s \subset h\}$;
  $v.left$ = BSP2D($\{s \in S : s \cap h^- \neq \emptyset\}$);
  $v.right$ = BSP2D($\{s \in S : s \cap h^+ \neq \emptyset\}$);
}
return v;

3.3 Construction in 2D

Proposition 2

The expected size of a bsp-tree for n non-intersecting line segments is $O(n \log n)$ and the expected time for the construction is $O(n^2 \log n)$.

Proof

- Expected number of fragmentations $O(n \log n)$ ➡ size.
- Every split takes $O(n)$.

➡ Expected runtime $O(n^2 \log n)$.

3.3 Construction in 2D

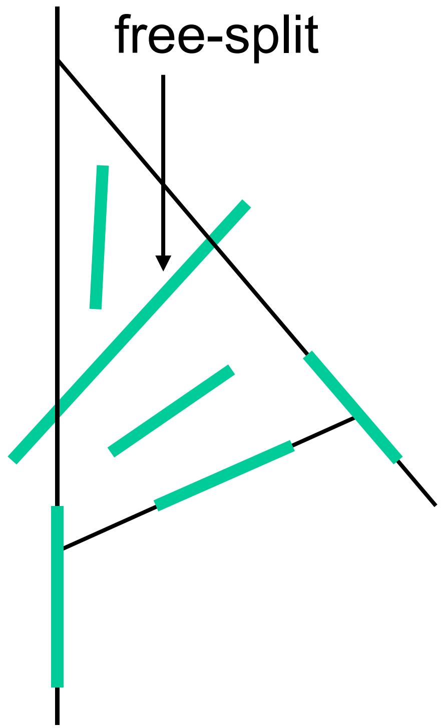
- In the proof of Proposition 2 we made no assumptions about the positions of the line segments, causing the bad runtime bound.
  - In practice the number of fragmentations is below the expected value, making the algorithm applicable, because the bsp-tree is computed off-line.
- Without auto-partitioning, there are deterministic algorithms to construct a bsp-tree in $O(n \log n)$ [1].
  - This is the lower bound for the runtime of Algorithm 1.
- Open questions: What is a lower bound for the memory?

3.3 Construction in 2D

- Runtime and memory can be reduced choosing better segments for the splits, i.e. *free-splits* that fragment no other objects.
- Because for every inner node all objects must be assigned to a half-space, the search for a free-split among them does not increase the runtime,
  - provided we can decide in $O(1)$ if a segment is a free-split or not.

3.3 Construction in 2D

- A subset of the free-split-segments, are segments that split a face of the bsp-partition into two faces.
  - These are usually long segments.
  - To determine these segments two flags are used for the two end-points of the segments initialized with 0.
  - If a segments is fragmented the flag of the split point is set to 1.
  - If the flags for both end-points are 1, the respective segments is a free-split.

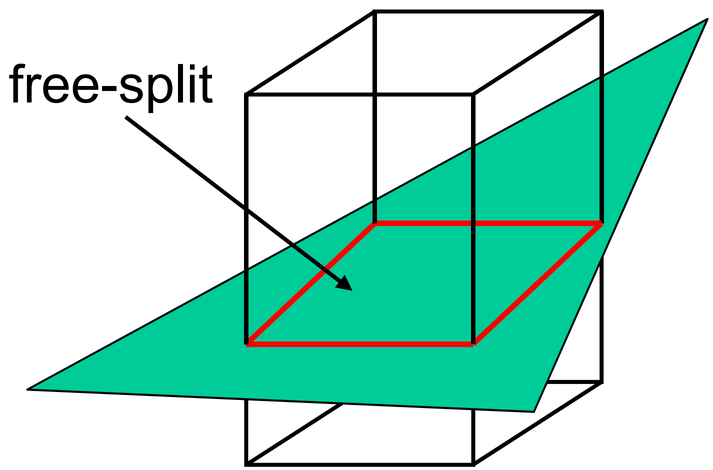


3.4 Extension to 3D

- Algorithm 1 can directly be extended to spaces of dimension $d > 2$.
- Objects are *simplexes* of dimension $d - 1$, i.e. convex hulls of d affine independent points.
  - Such a simplex spans an affine subspace h of dimension $d - 1$ (hyper-plane) that is used for the splitting.
- Analog to the 2D algorithm, in 3D also favorable positions of segments can be used to reduce the runtime and memory using free-splits.

3.4 Extension to 3D

- In 3D free-splits are identified via the edges:
  - The triangles are fragmented into convex, planar polygons.
  - For every fragmentation the new edges are marked.
  - A free-split is a polygon without unmarked edges (if there are no intersections of the initial triangles).
- If there is no free-split, the algorithms takes the first polygon from the random list (or a polygon with a small number of marked edges).



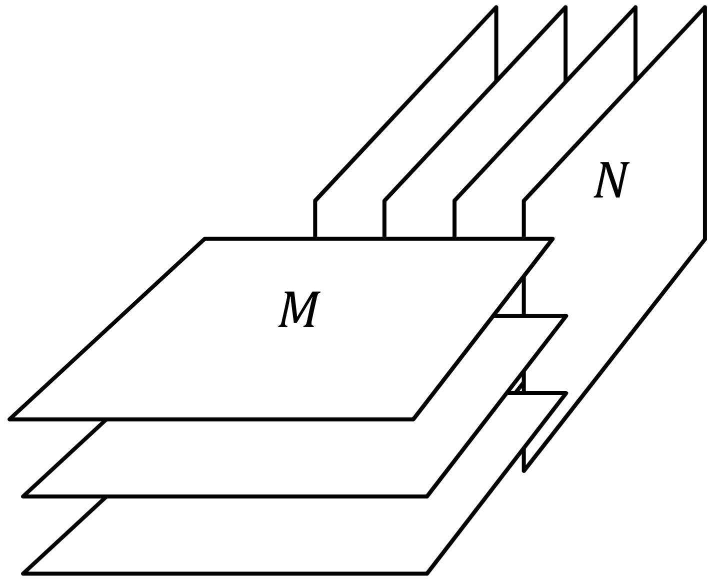
3.4 Extension to 3D

Proposition 3

A lower bound for the size of a bsp-tree in 3D using auto-partitioning is $\Omega(n^2)$.

Proof

- Worst case scenario.



3.4 Extension to 3D

- Also in 3D in practice most input data sets are mostly well-behaved, such that the algorithm takes less runtime and memory than in the worst case.
- Without auto-partitioning, for orthogonal rectangles, analog to Proposition 3, the bsp-tree has size $O(n\sqrt{n})$ [4].
- With certain assumptions, e.g. an upper bound on the ratio of maximal to minimal length of objects, the size of the bsp-tree reduces to $O(n)$ [2].

3.5 Literature

[1] Marc de Berg et al., *Computational Geometry: Algorithms and Applications*, 2nd Edition, Springer, 2000, Chapter 12.

[2] M. de Berg, *Linear size binary space partitions for fat objects*, 3rd European Symposium on Algorithms (ESA), 252-263, 1993.

[3] H, Fuchs, Z.M. Kedem, and B. Naylor, *On visible surface generation by a priori tree structures*, SIGGRAPH 1980, 124-133.

[4] M.S. Paterson and F.F. Yao, *Optimal binary space partitions for orthogonal objects*, Journal on Algorithms, 13: 99-113, 1992.