Spatial Data Structures

Acknowledgement: Marco Tarini

University of Victoria | Computer Science

Types of Queries

- Graphic applications often require spatial queries
  - Find the k points closer to a specific point p (k-Nearest Neighbours, knn)
  - Is object X intersection with object Y? (Intersection)
  - What is the volume of the intersection between two objects?
- Brute force search is expensive. Instead, you can solve these queries with an initial preprocessing that creates a data structure which supports efficient queries
- The data structure to use is application-specific

Two Main Ideas

1. You can explicitly index the space itself (Spatial Index)

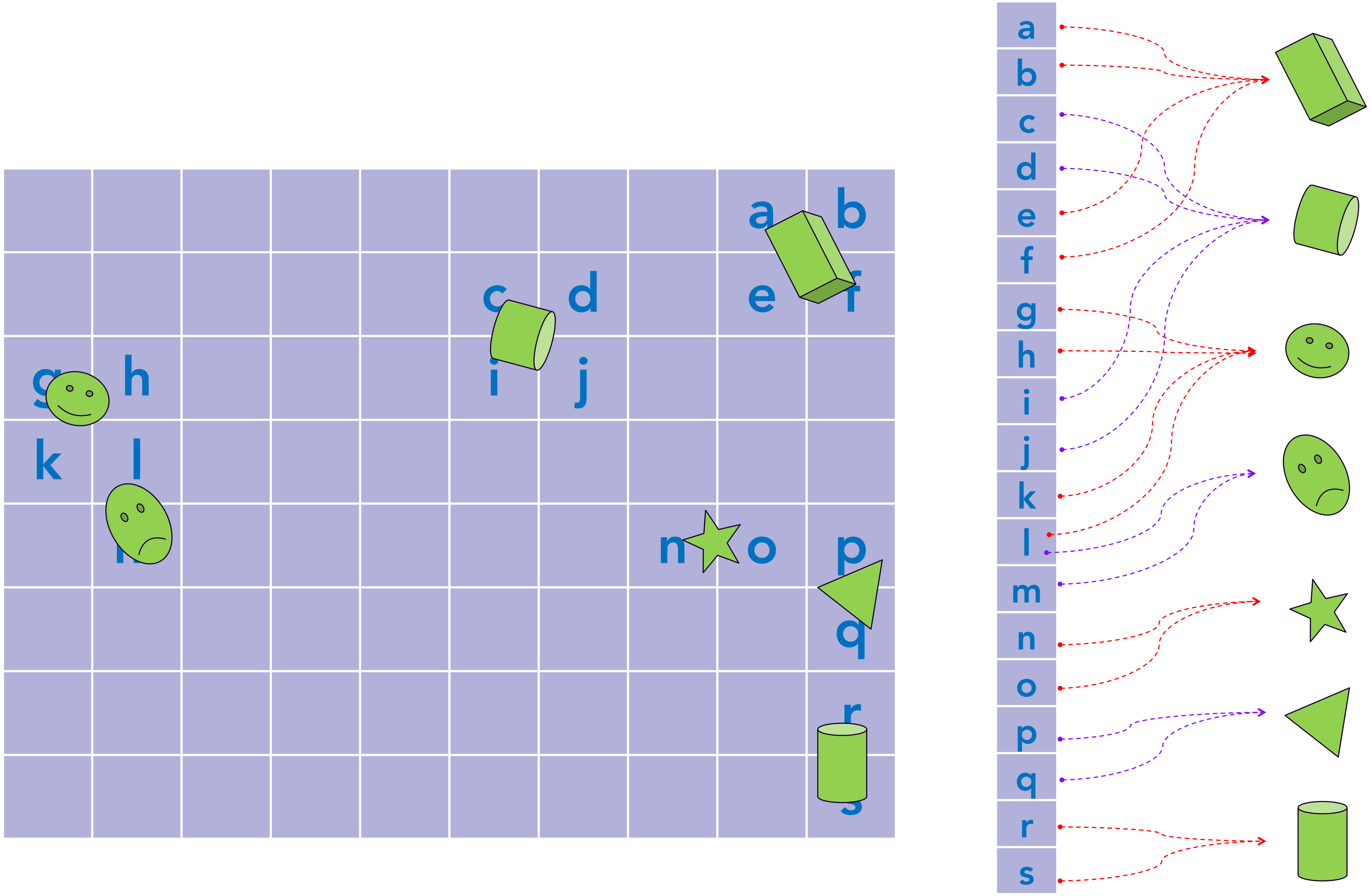
2. You can "sort" the primitives in the scene, which implicitly induces a partition of the space (Bounding Volume Hierarchies)

Spatial Indexing Structures

Spatial Indexing Structures

- Data structures to accelerate queries of the kind:
  "I'm here. Which object is around me?"
- Tasks:
  - (1) construction / update
    - for **static** parts of the scene, a preprocessing.
    - for **moving** parts of the scene, an update.
  - (2) access / usage
    - as fast as possible
- The most common structures are:
  - Regular Grid
  - kD-Tree
  - Oct-Tree/Quad-Tree
  - BSP Tree

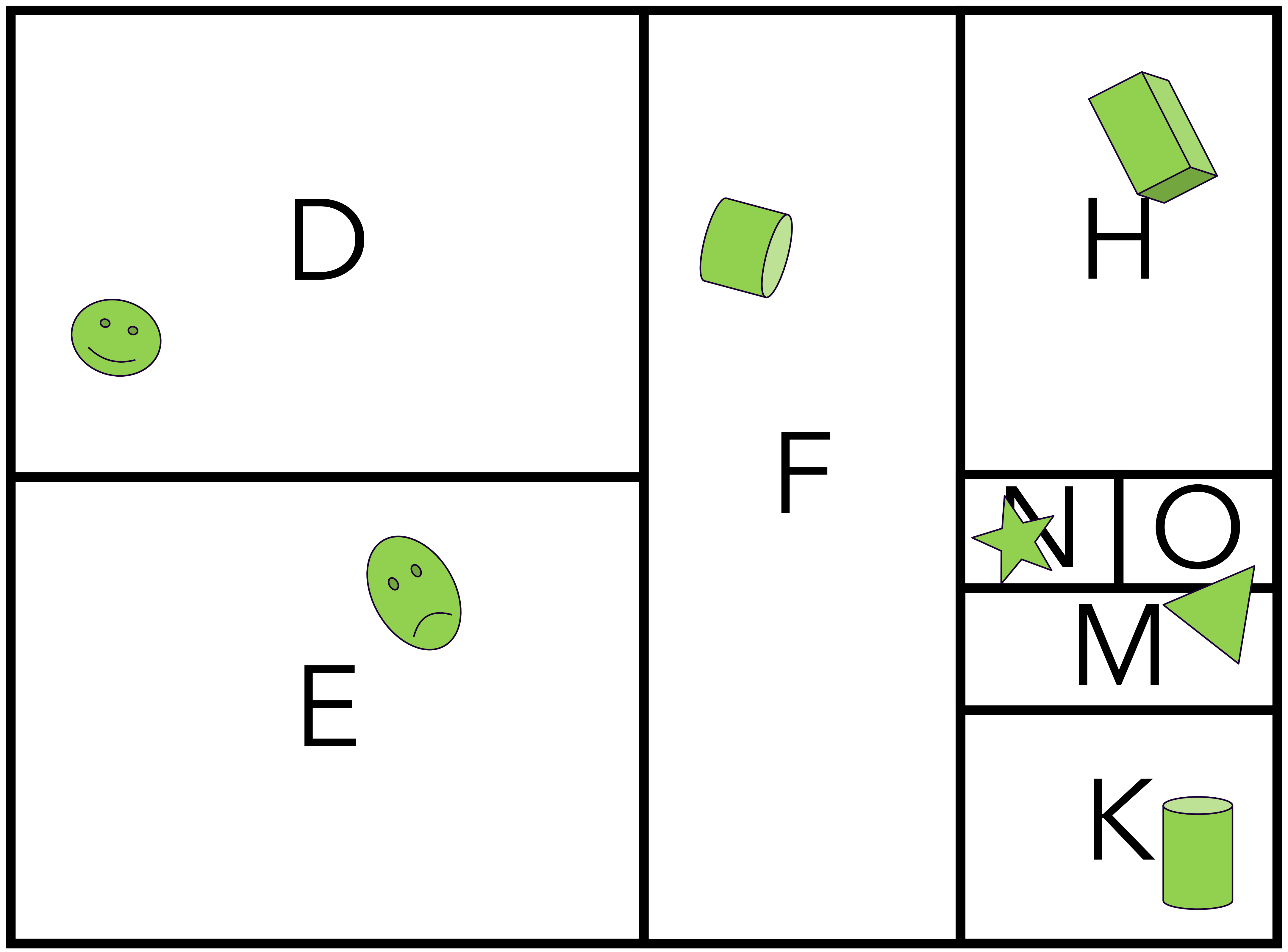
Regular Grid (aka lattice)



Regular Grid (or: lattice)

- Array 3D of cells (same size)
  - each cell: a list of pointers to colliding objects
- Indexing function:
  - Point3D ∩ cell index, (constant time!)
- Construction: ("scatter" approach)
  - for each object B[i]
    - find the cells C[j] which it touches
    - add a pointer in C[j] to B[i]
- Queries: ("gather" approach)
  - given a point to test *p*,
    find cell C[j], test all objects linked to it
- Problem: cell size
  - too small: memory occupancy too large
    quadratic with inverse of cell size!
  - too big: too many objects in one cell
  - sometimes, no cell size is good enough

kD-tree



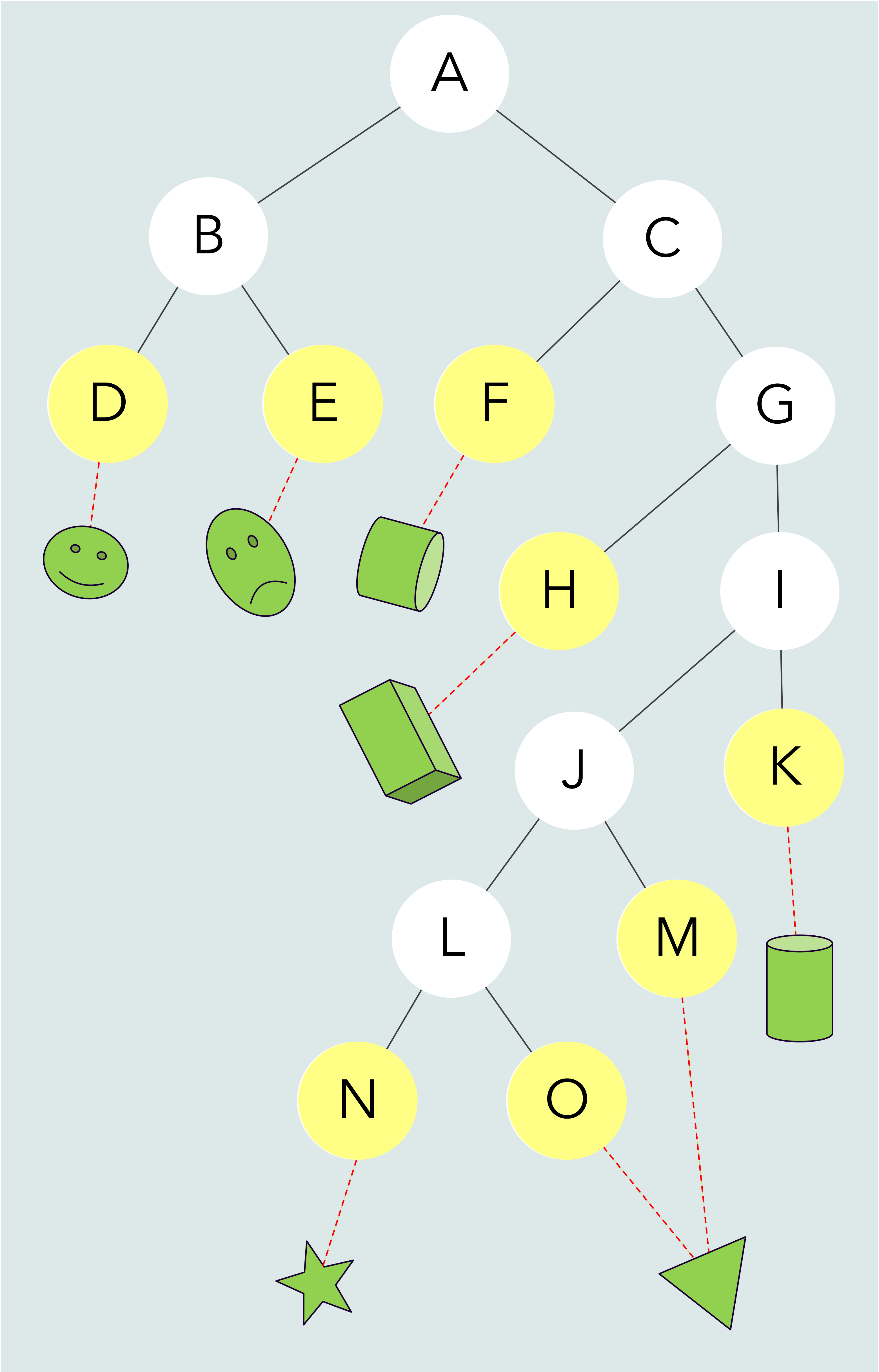


Image Copyright: Marco Tarini

kD-trees

- Hierarchical structure: a tree
  - each node: a subpart of the 3D space
  - root: all the world
  - child nodes: partitions of the father
  - objects linked to leaves
- kD-tree:
  - binary tree
  - each node: split over one dimension (in 3D: X,Y,Z)
  - variant:
    - each node optimizes (and stores) which dimension, or
    - always same order: e.g. X then Y then Z
  - variant:
    - each node optimizes the split point, or
    - always in the middle

Quad-Tree (2D)

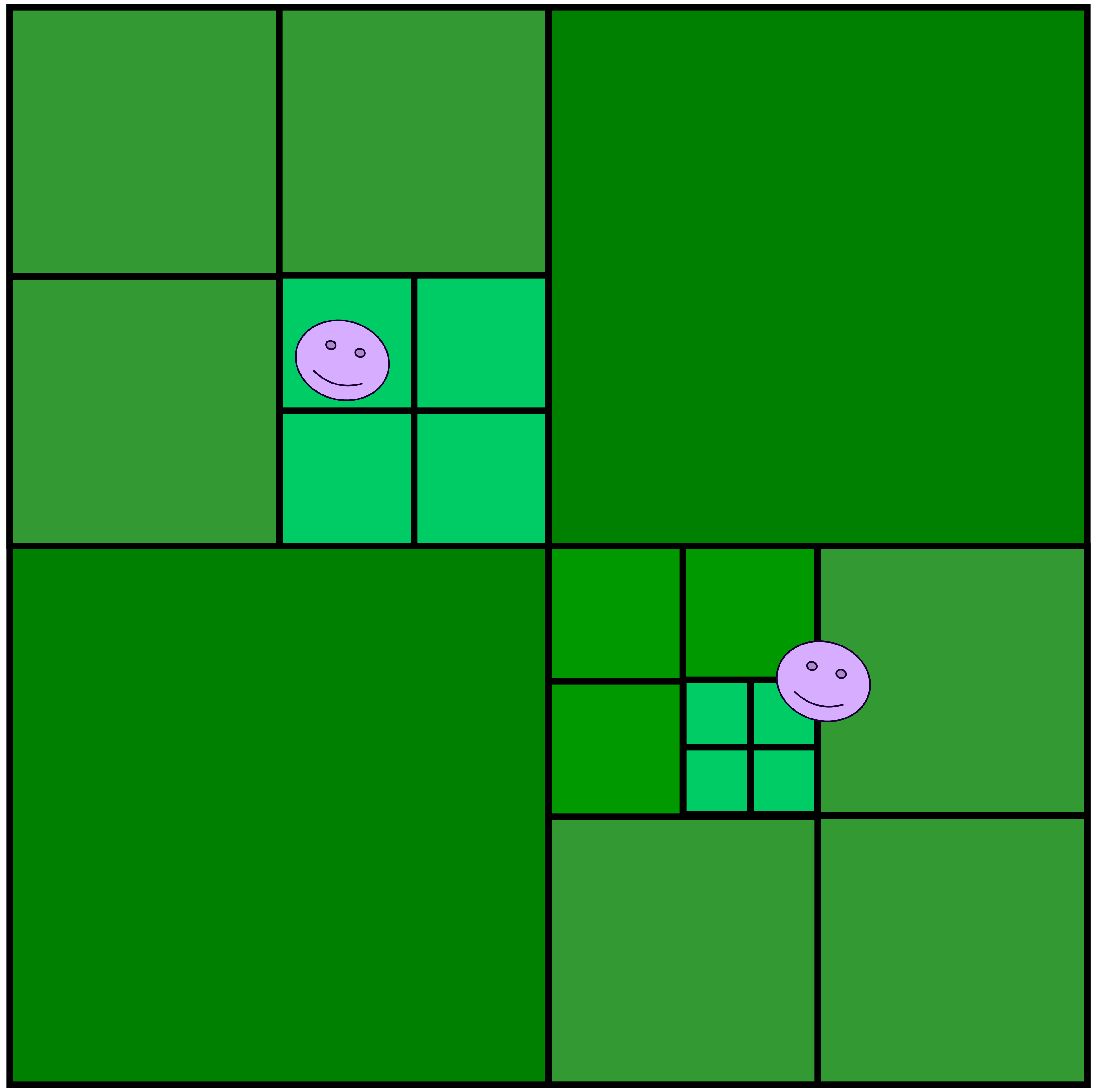
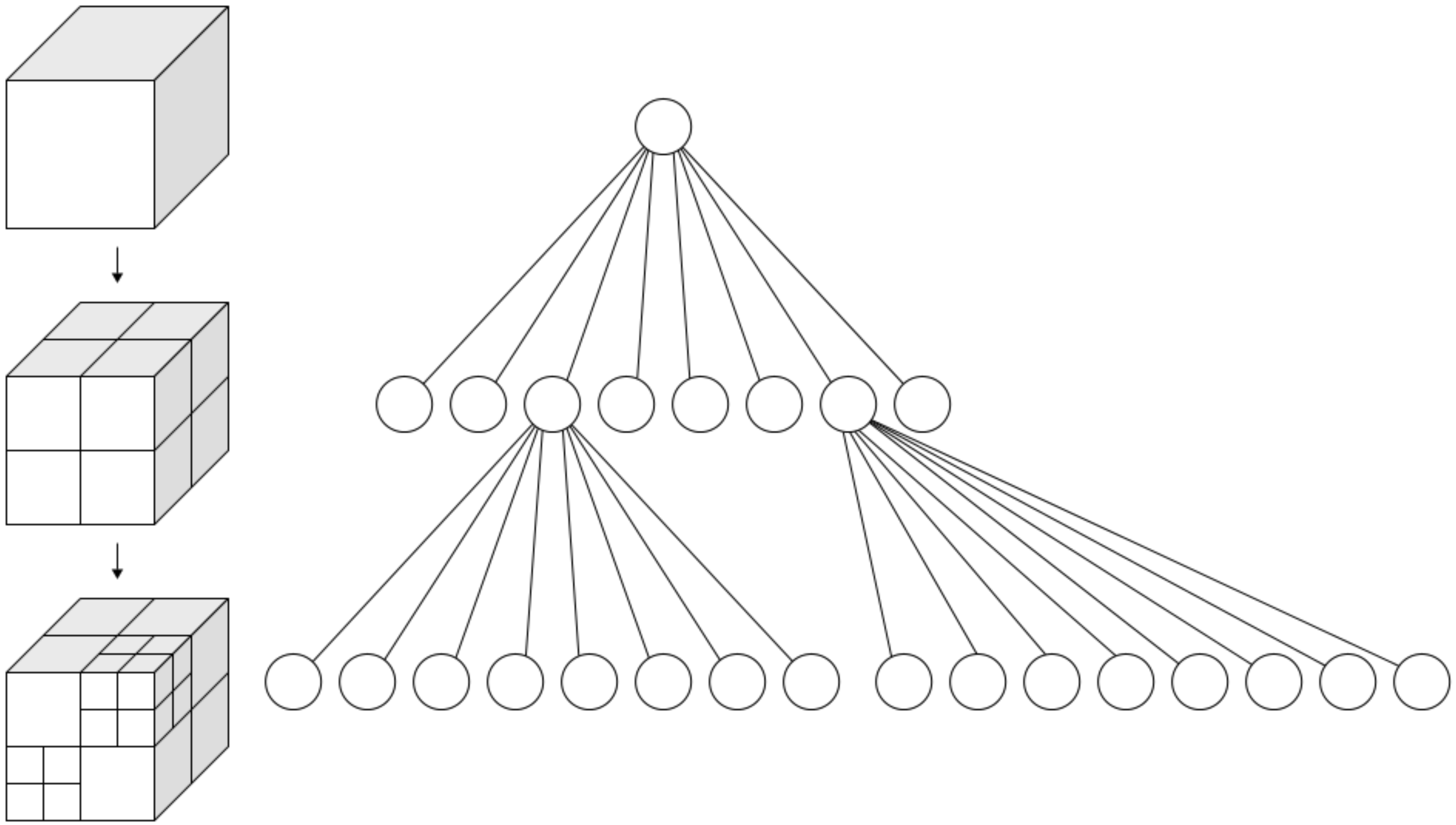


Image Copyright: Marco Tarini

Oc-Tree (3D)



Quad trees (in 2D)
Oct trees (in 3D)

- Similar to kD-trees, but:
  - tree: branching factor: 4 (2D) or 8 (3D)
  - each node: splits into all dimensions at once,
    (in the middle)

- Construction (just as kD-trees):
  - continue splitting until a end nodes has few enough objects
    (or limit level reached)

BSP-tree

Binary Spatial Partitioning tree

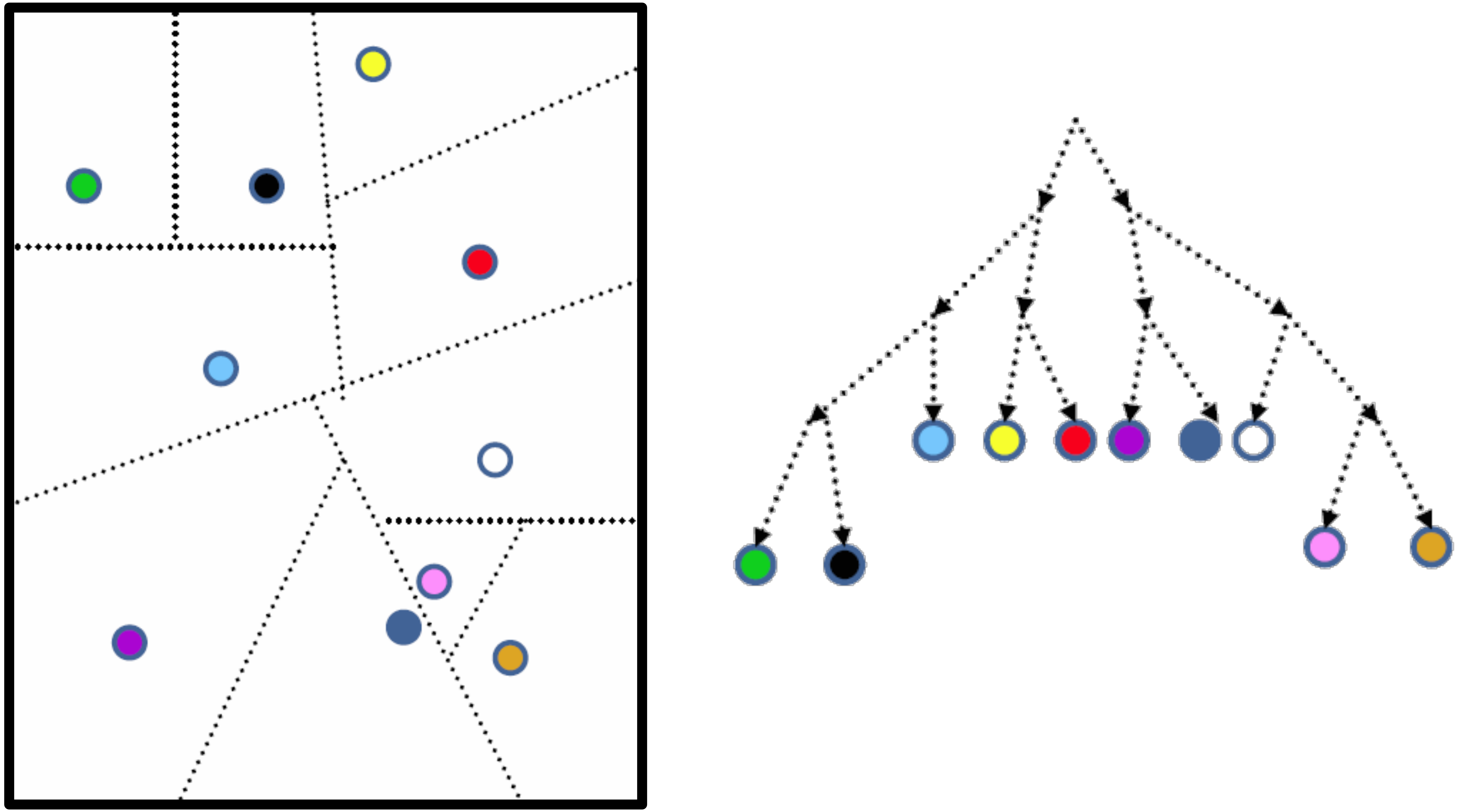


Image Copyright: Marco Tarini

BSP-trees for the Concave Polyhedron proxy

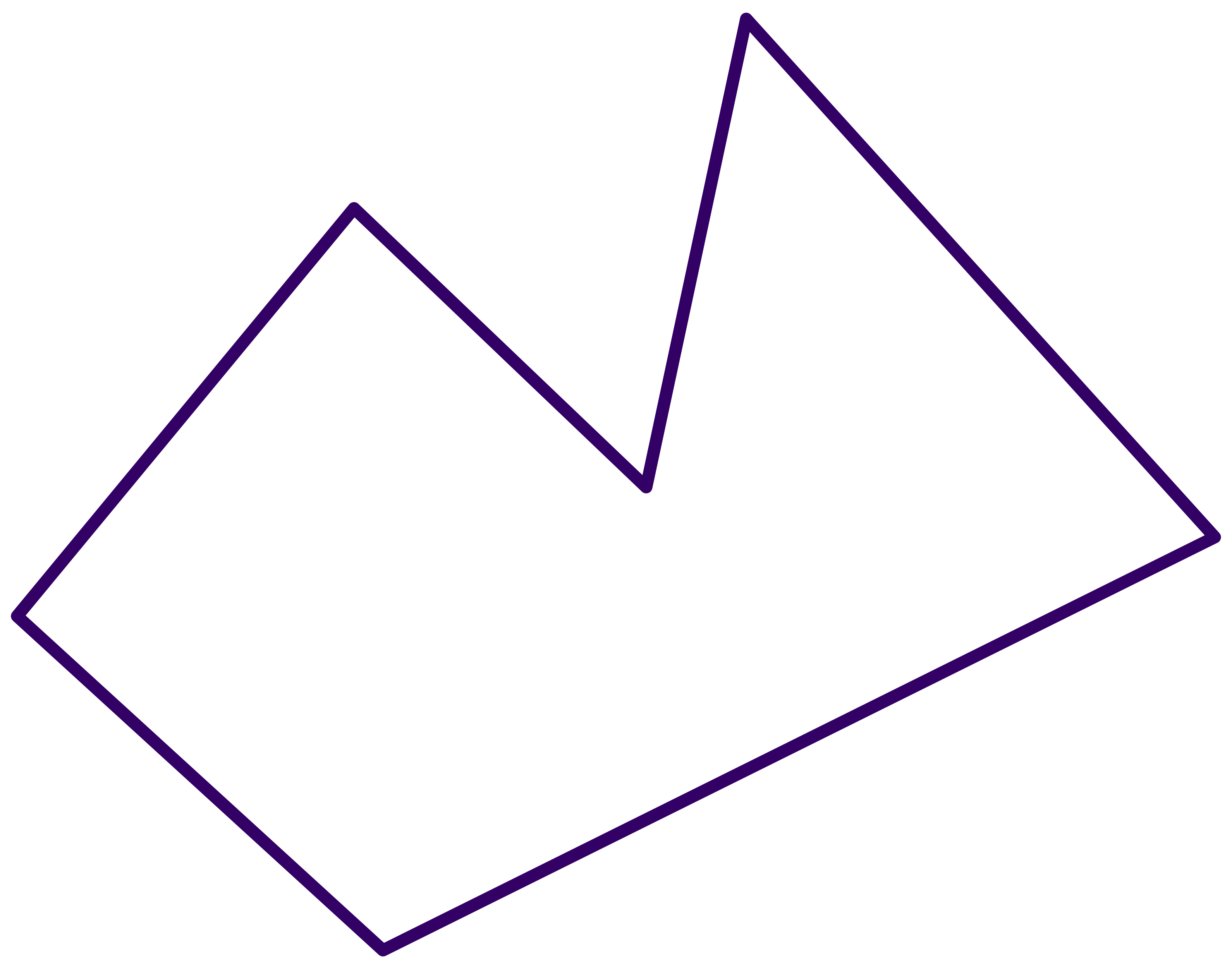


Image Copyright: Marco Tarini

BSP-trees for Inside-Outside Test

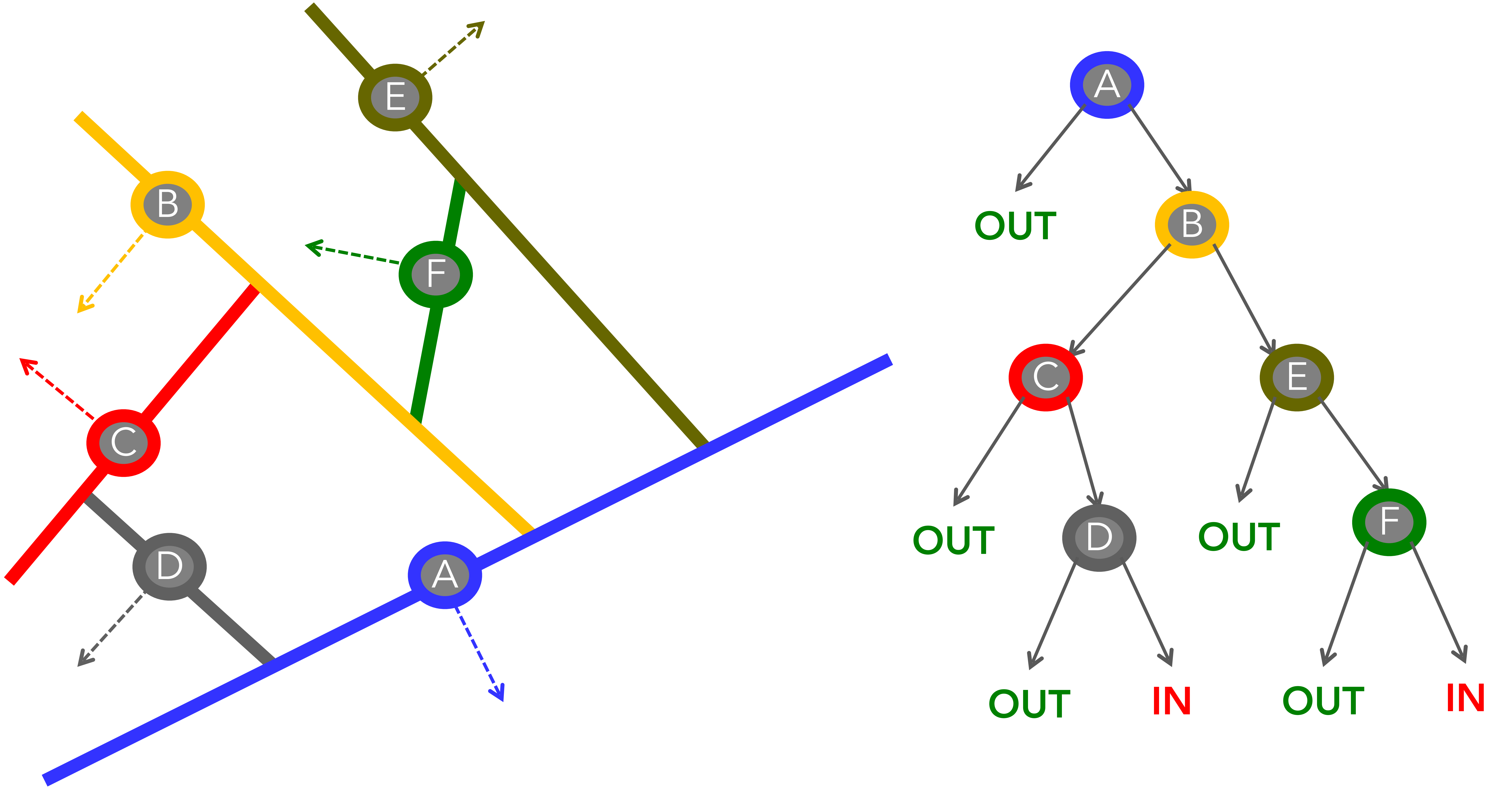


Image Copyright: Marco Tarini

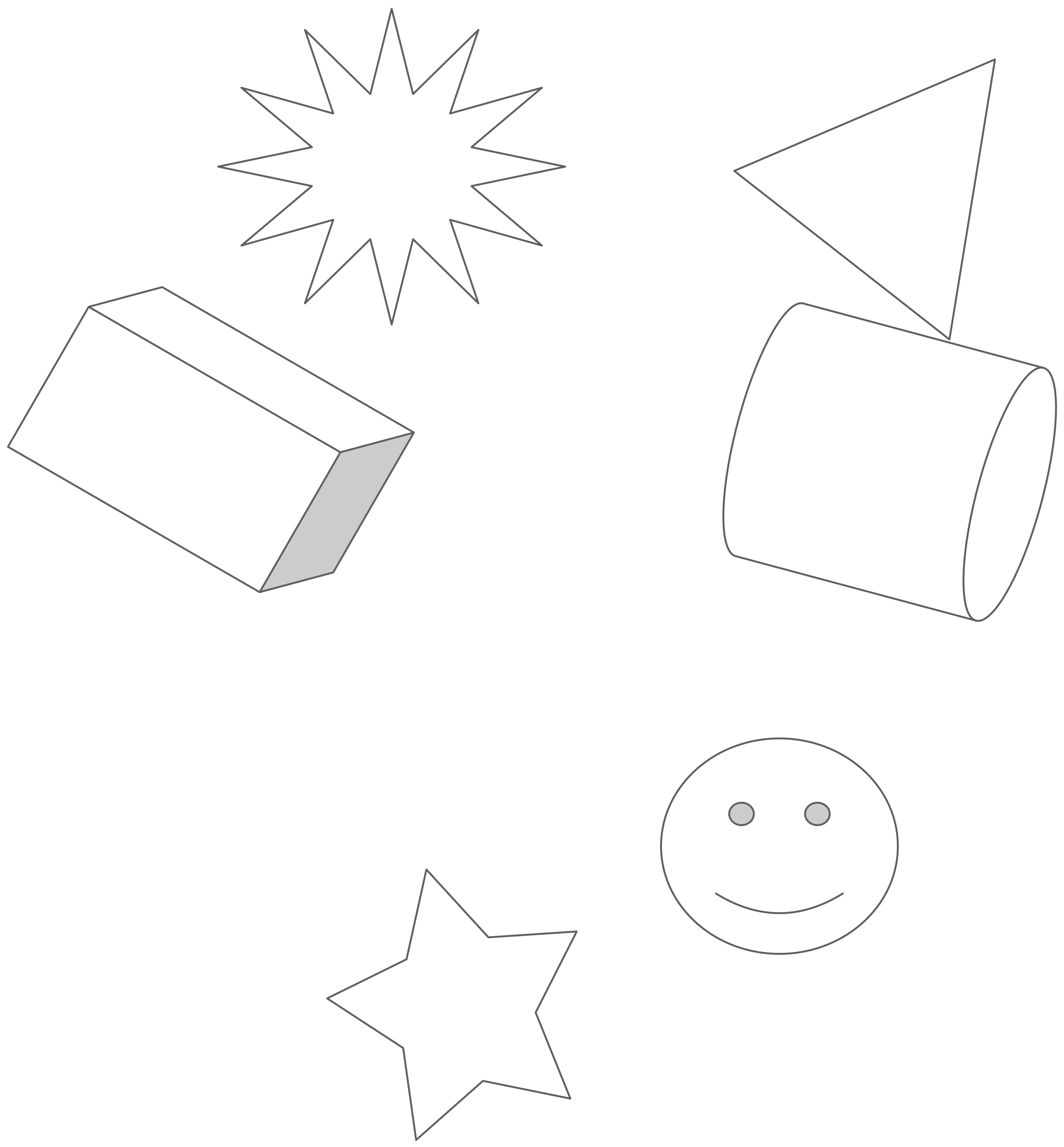
University of Victoria | Computer Science

BSP-tree
Binary Spatial Partitioning tree

- Another variant
  - a binary tree (like the kD-tree)
    - root = all scene (like kD-tree)
  - but, each node is split by an **arbitrary** plane
    - (or a **line**, in 2D)
    - plane is stored at node, as (nx, ny, nz, k)
  - planes can be optimized for a given scene
    - e.g. to go for a 50%-50% object split at each node
- Another use: to test (Generic) Polyhedron proxy:
  - note: with planes defined in its **object space**
  - each leaf: inside or outside
    - (no need to store them: left-child = in, right-child = out)
  - tree precomputed for a given object

Primitive Sorting Structures

Bounding Volume Hierarchies (BVH)



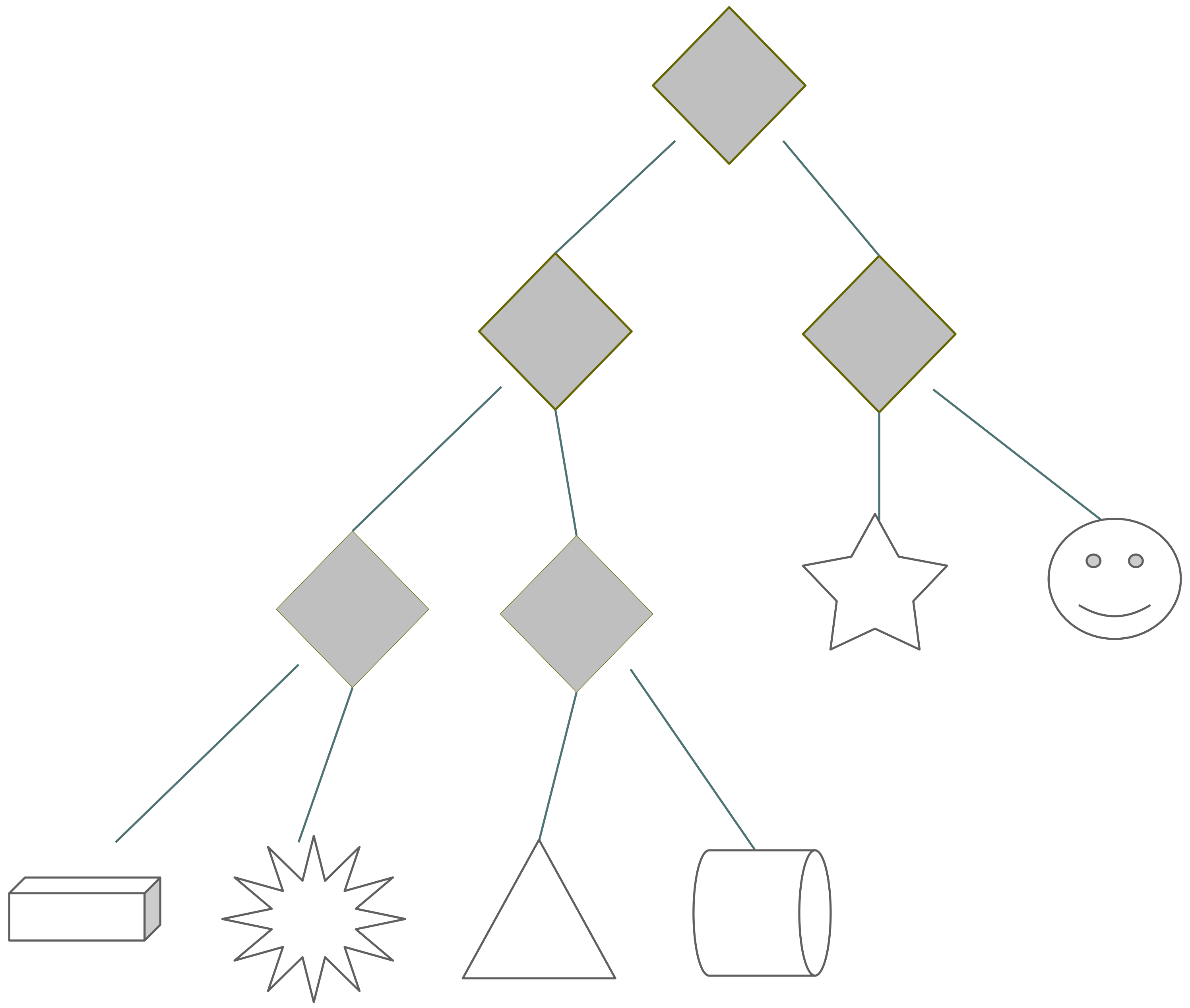
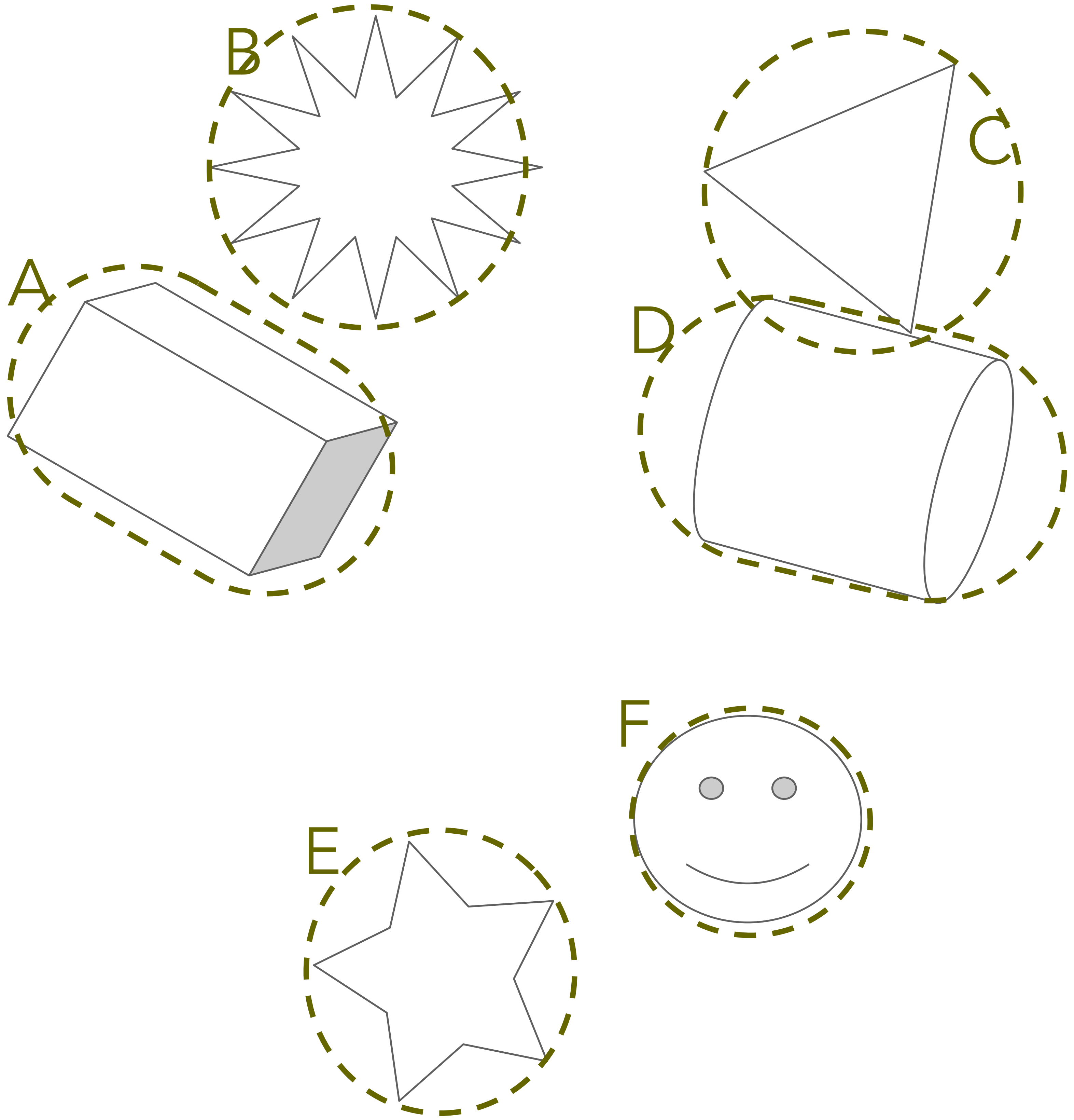


Image Copyright: Marco Tarini

Bounding Volume Hierarchies (BVH)



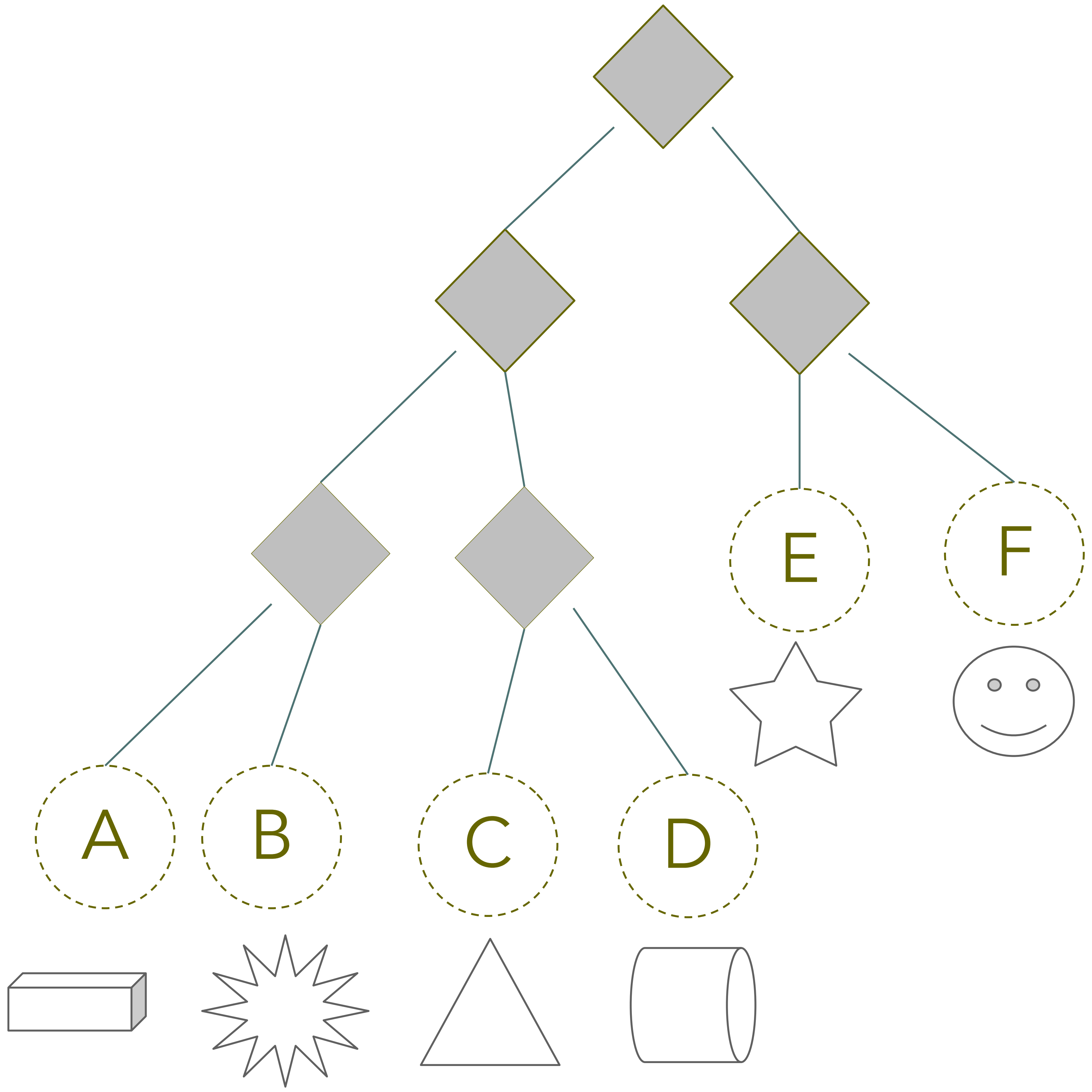
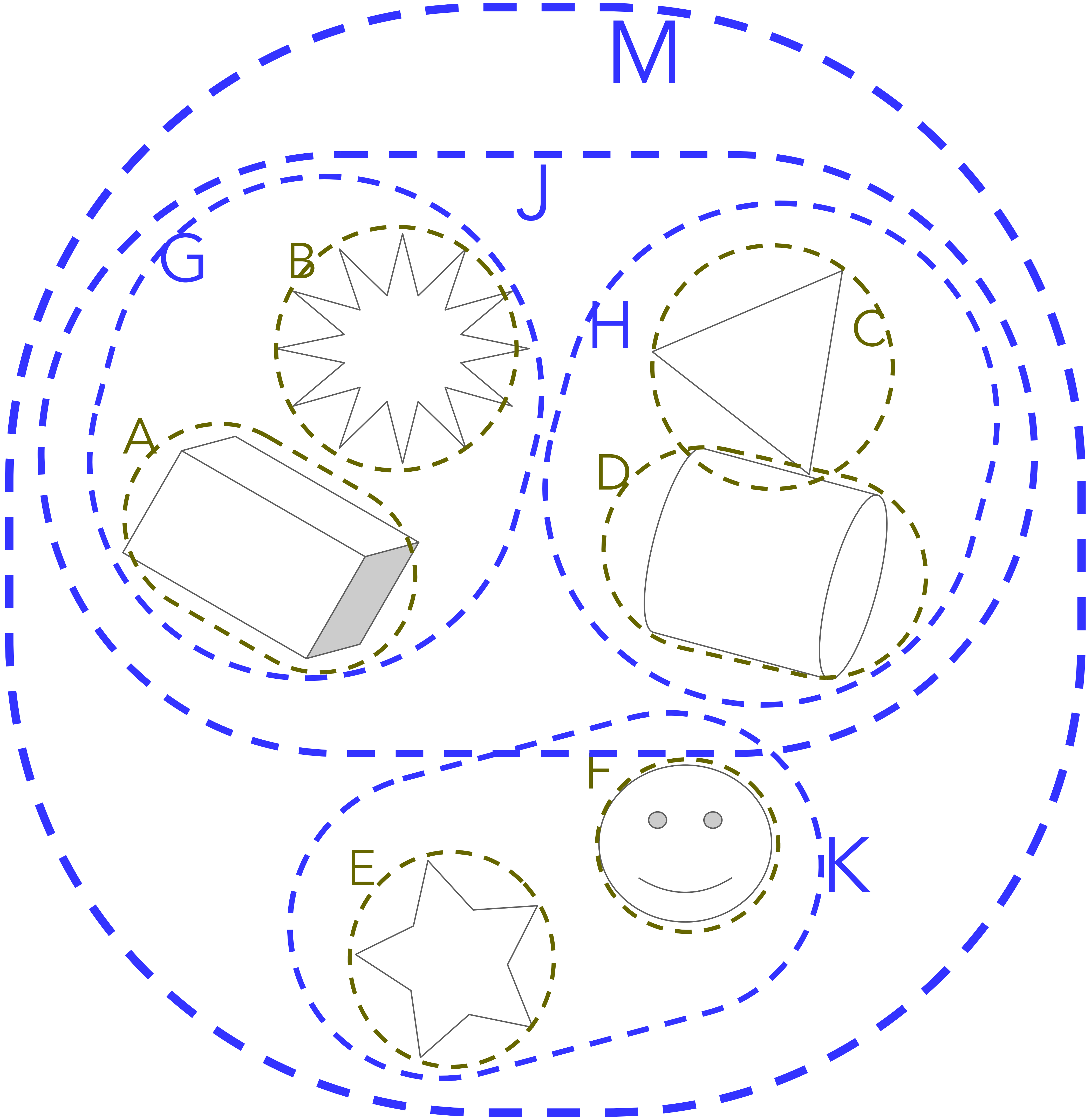
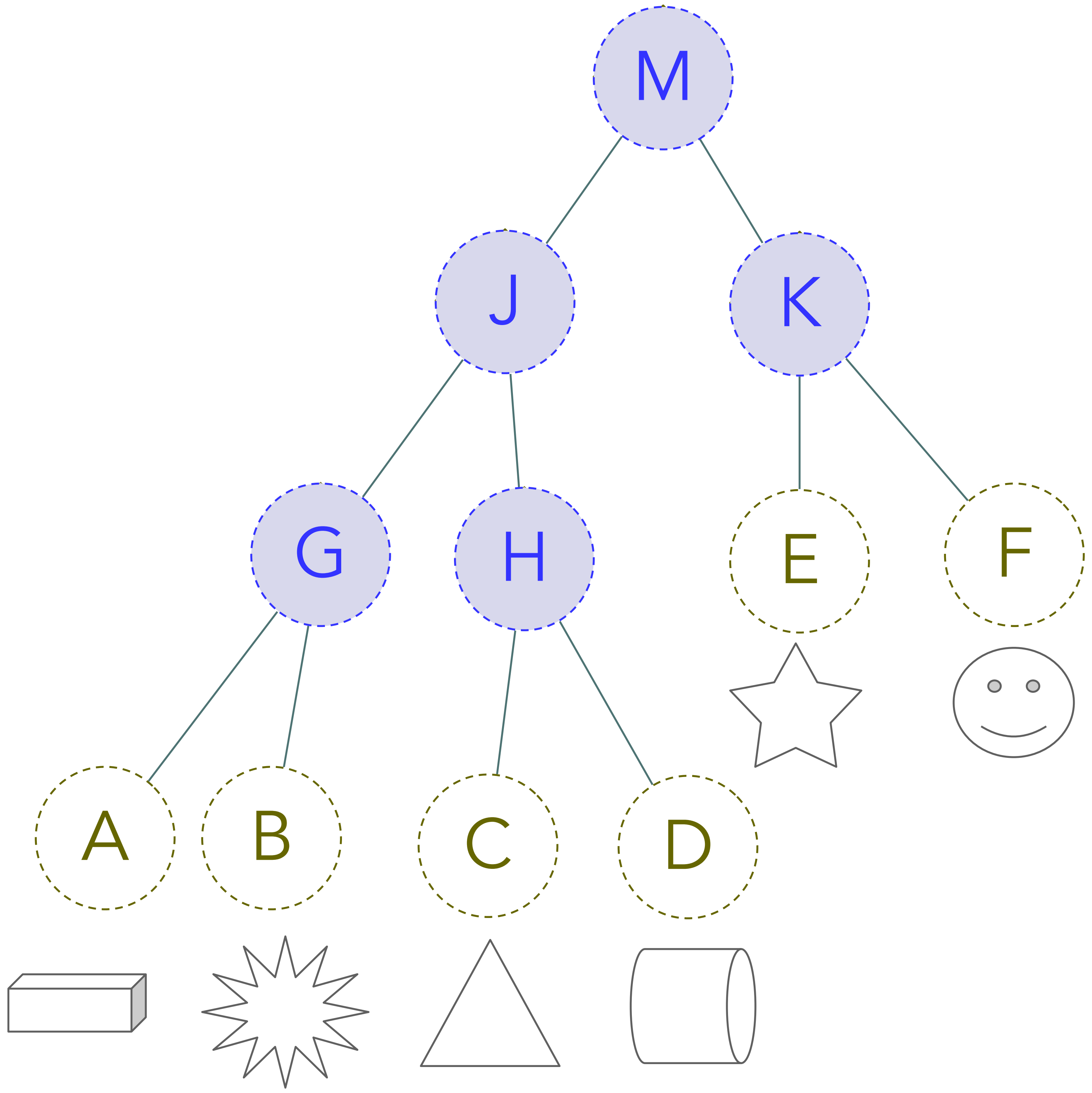


Image Copyright: Marco Tarini

Bounding Volume Hierarchies (BVH)





BVH
Bounding Volume Hierarchy

- Idea: use the scene hierarchy given by the scene graph
  - (instead of a spatial derived one)
- associate a Bounding Volume to each node
  - rule: a BV of a node bounds all objects in the subtree
- construction / update is fast
  - bottom-up: recursive
- using it:
  - top-down: visit
  - *note:* ***not*** a single root to leaf path
    - may need to follow *multiple* children of a node
      (in a BSP-tree: only one)

Spatial Indexing Structures

- Regular Grid
  - the most parallelizable (to update / construct / use)
  - constant time access (best!)
  - quadratic / cubic space (2D, 3D)
- kD-tree, Oct-tree, Quad-tree
  - compact
  - simple
  - non constant accessing time (still logarithmic on average)
- BSP-tree
  - optimized splits! best performance when accessed
  - optimized splits! more complex construction / update
  - ideal for static parts of the scene
  - (also, used for generic polyhedron inside/outside test)
- BVH
  - simplest construction
  - non necessarily very efficient to access
    - may need to traverse multiple children
    - if you do not have a scene-graph you need to create one
  - ideal for dynamic parts of the scene

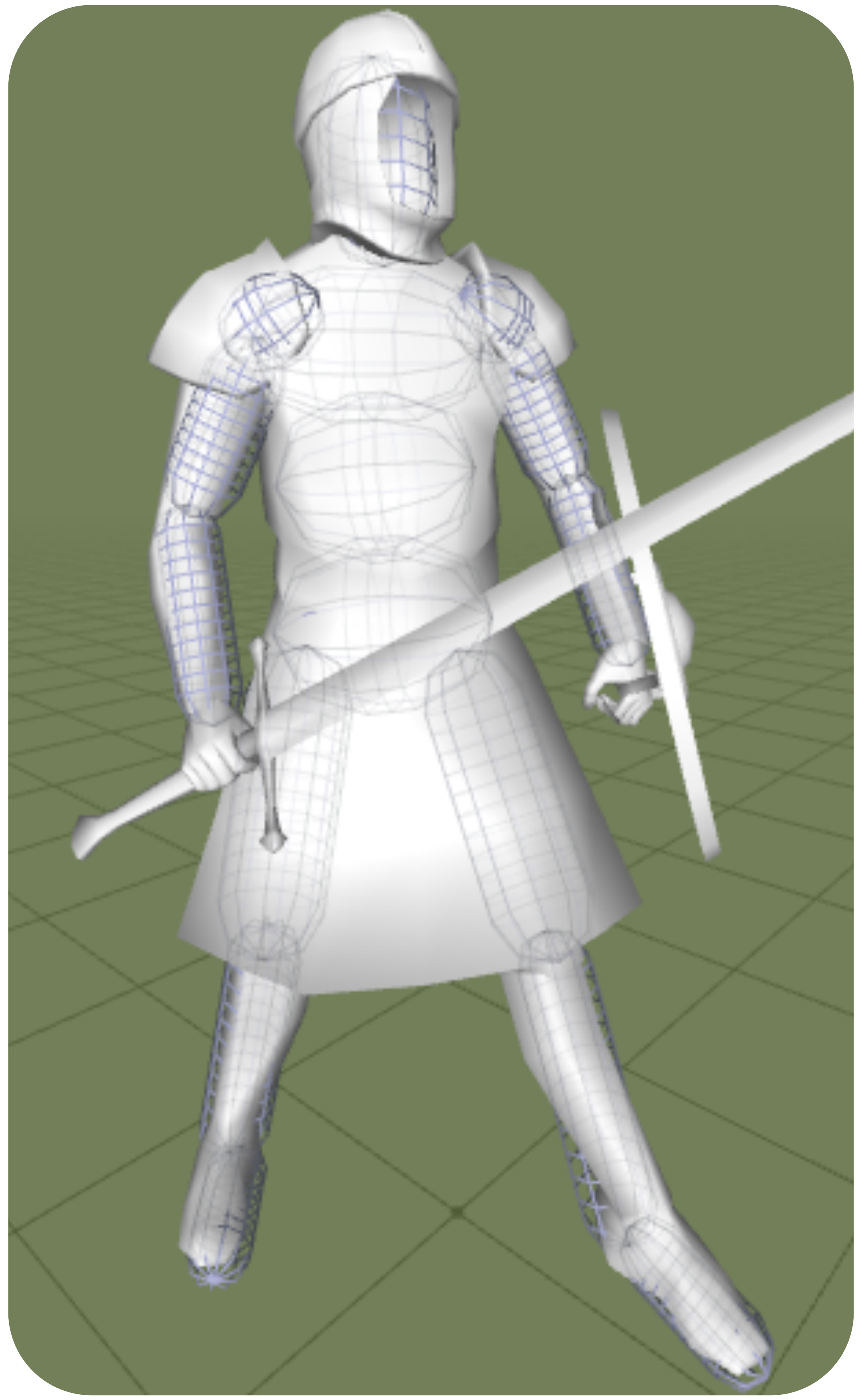
Intersection Acceleration Data Structures

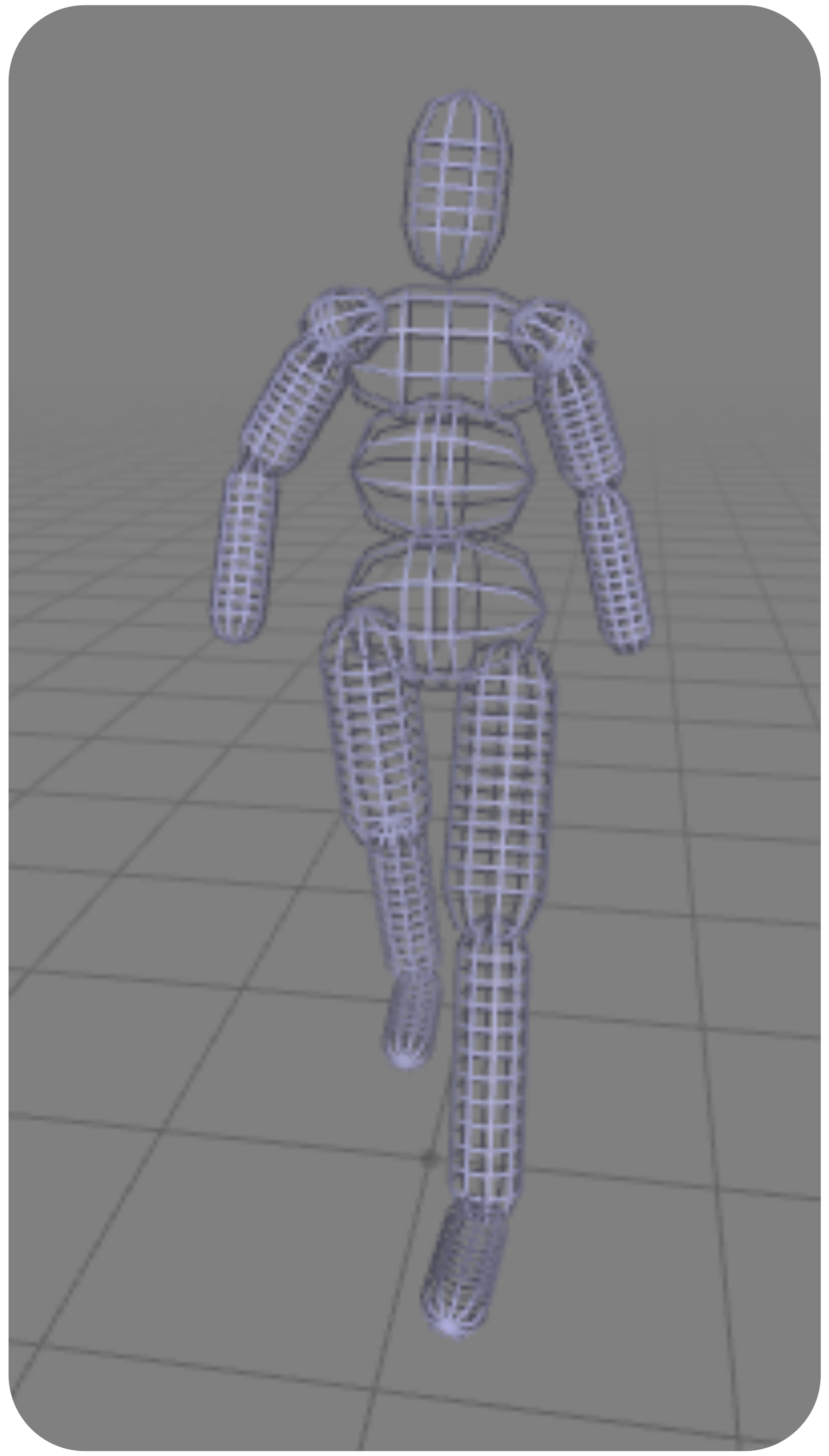
Collision Detection

- It is easy to do, the challenge is to do it efficiently
- An observation:
  - most pair of objects do not intersect each other in a scene, **collisions are rare**
  - optimizing the intersections directly is important but not sufficient, we need to optimize the detection of non intersecting pairs ("**early rejects**")

Geometric Proxies

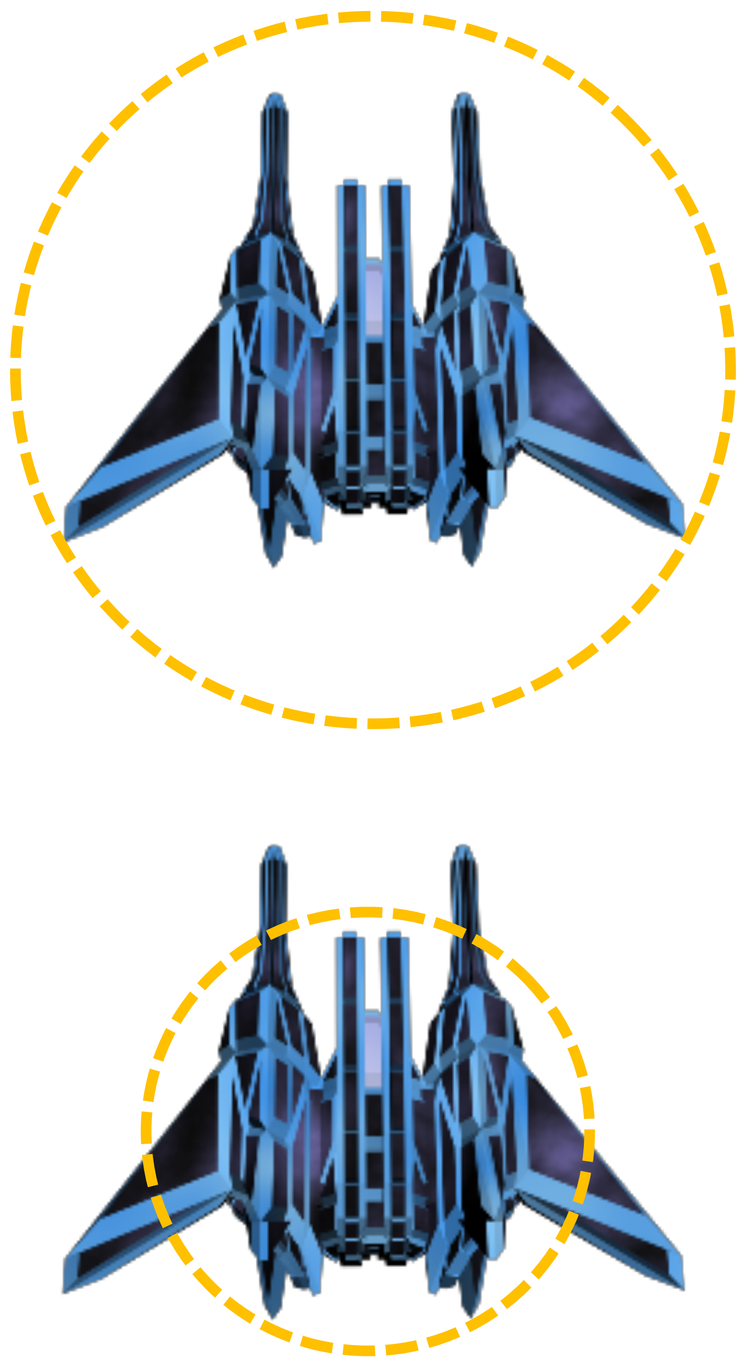
- Idea: use a geometric proxy to approximate the objects in the scene



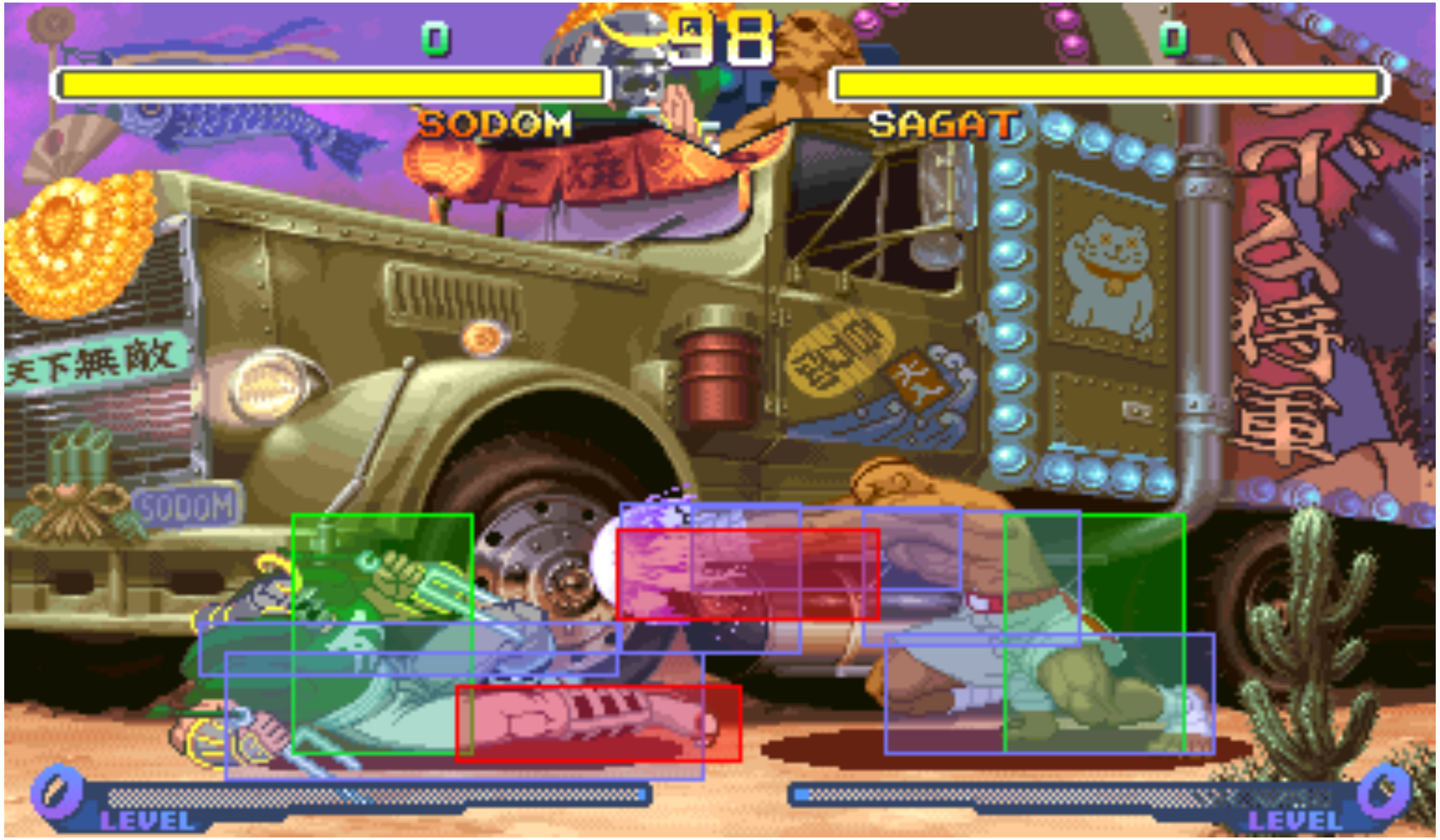


Geometric Proxy

- Extremely coarse approximation
- Used as a:
    - Bounding Volume
        - the entire object must be contained inside
        - exact result, you need to do more work if you detect a collision
    - Collision Object (or "hit-box")
        - approximation of the object
        - no need to do anything else if an approximation is ok for your use case



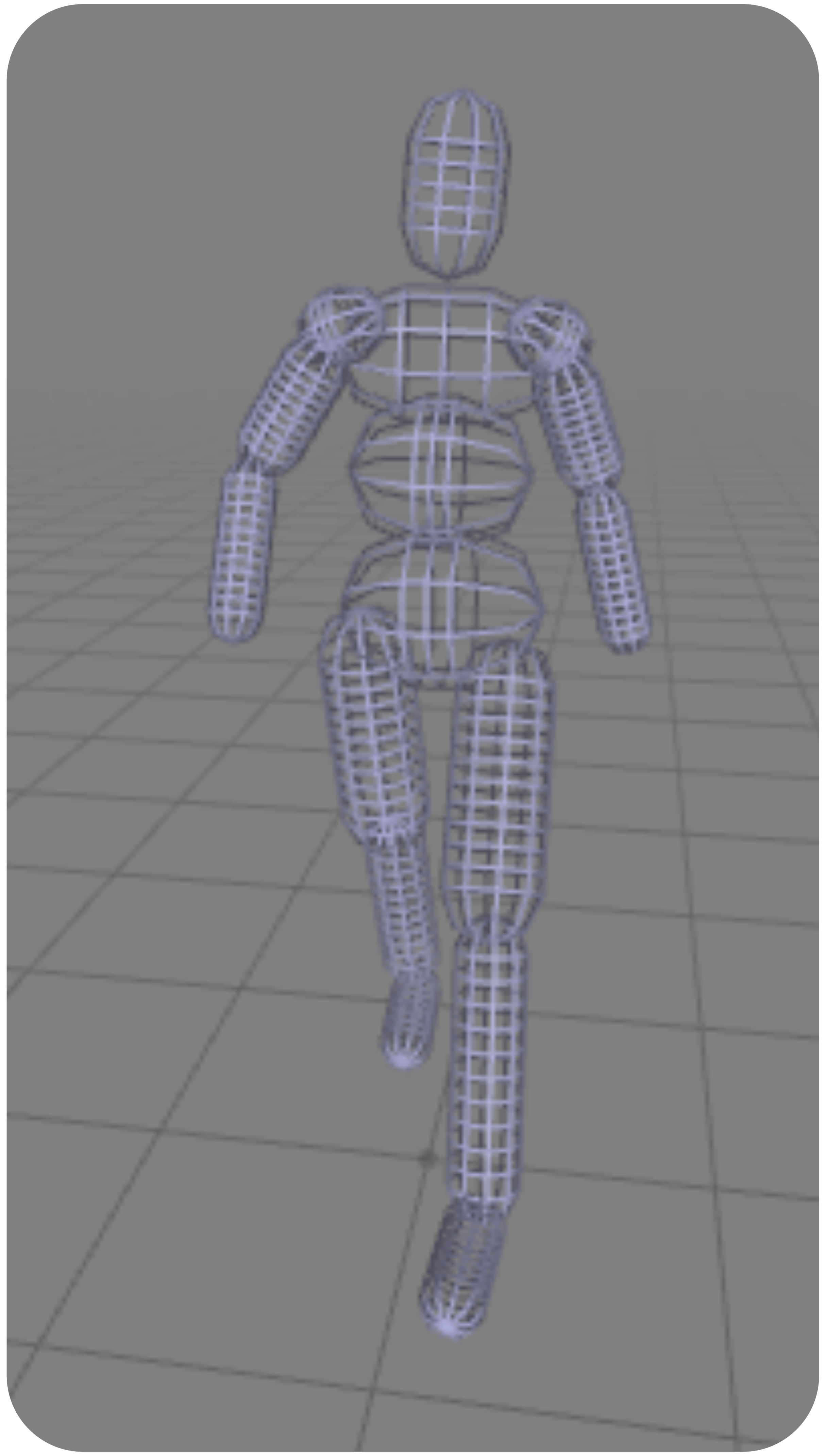
Example: Fighting Games



Street Fighter Alpha, CAPCOM 1995

Extremely Common

- Physic engine
  - collision detection
  - collision response
- Rendering
  - view frustum culling
  - occlusion culling
- AI
  - visibility test
- GUI
  - picking

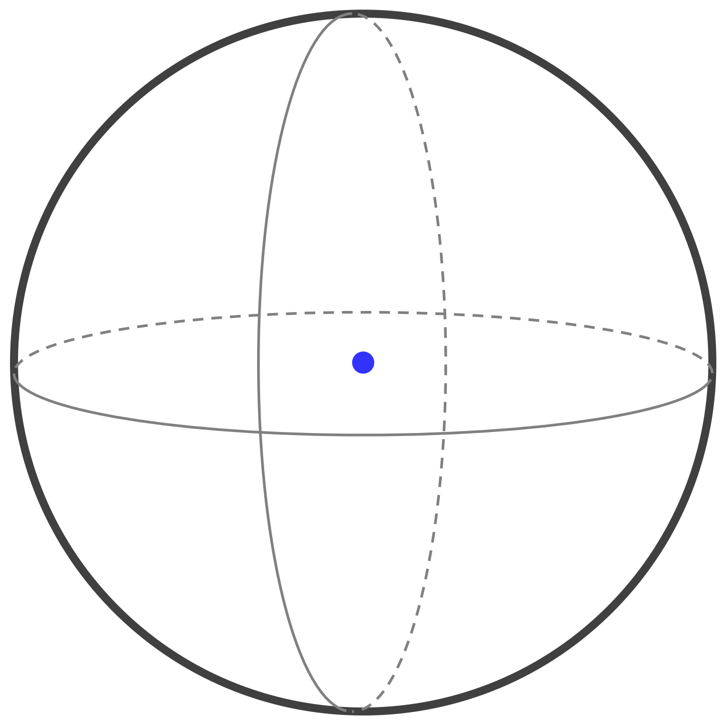


Properties of Geometric Proxies

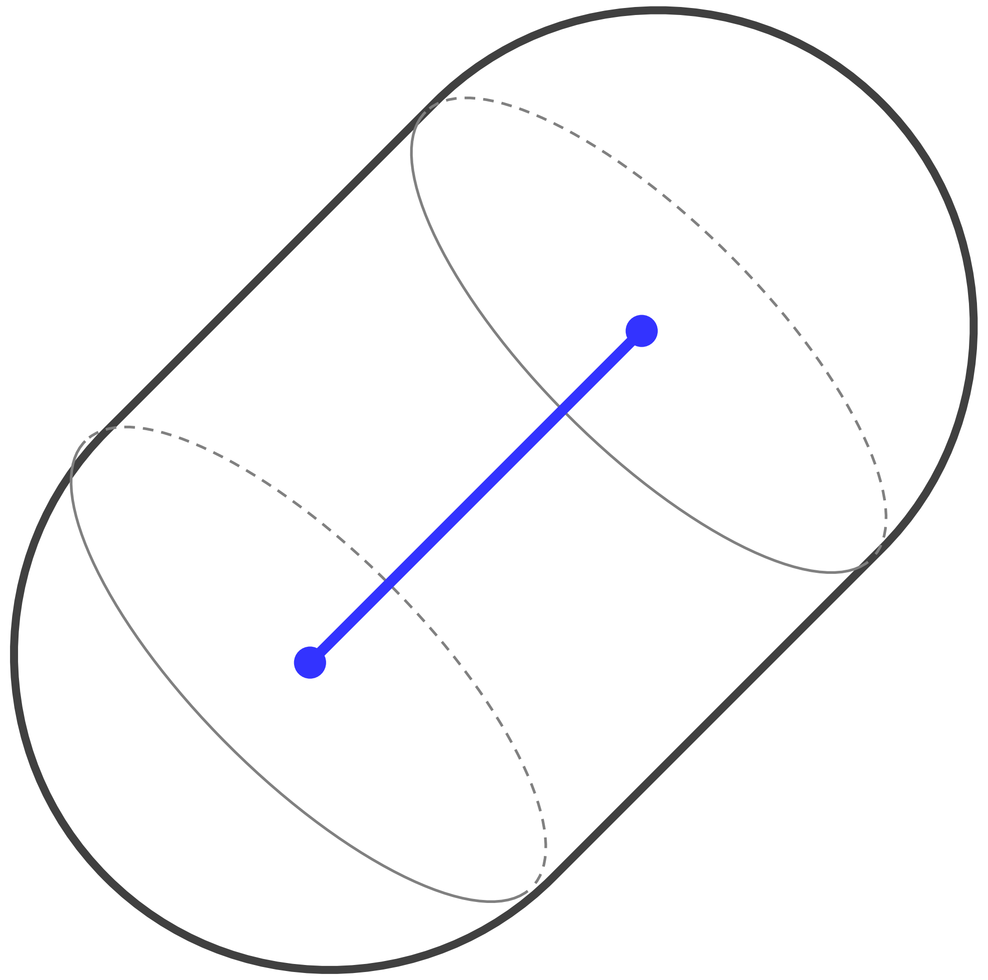
1. How expensive are they to compute/update?
2. How much space do you need?
3. Are they invariant to the transformations applied on the object?
4. How good is the approximation?
5. How expensive are the collision queries with the other objects in the scene?

Geometry Proxies: Sphere

- Easy to compute and update
- Compact (center, radius)
- Very efficient collision tests
- Can only be transformed rigidly
- The quality of the approximation is low



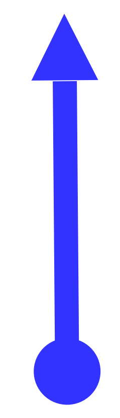
Geometry Proxies: Capsule



- Def:
  - Sphere ==
    set of all points with dist from a point < radius
  - Capsule ==
    set of all points with dist from a segment < radius
    - i.e. a cylinder ended with two half-spheres (all same radius)
- Stored with:
  - a segment (two end-points)
  - a radius (a scalar)
- Popular option, compact to store, easy to construct, easy to detect intersections, good approximation

Geometry Proxies: Half Space

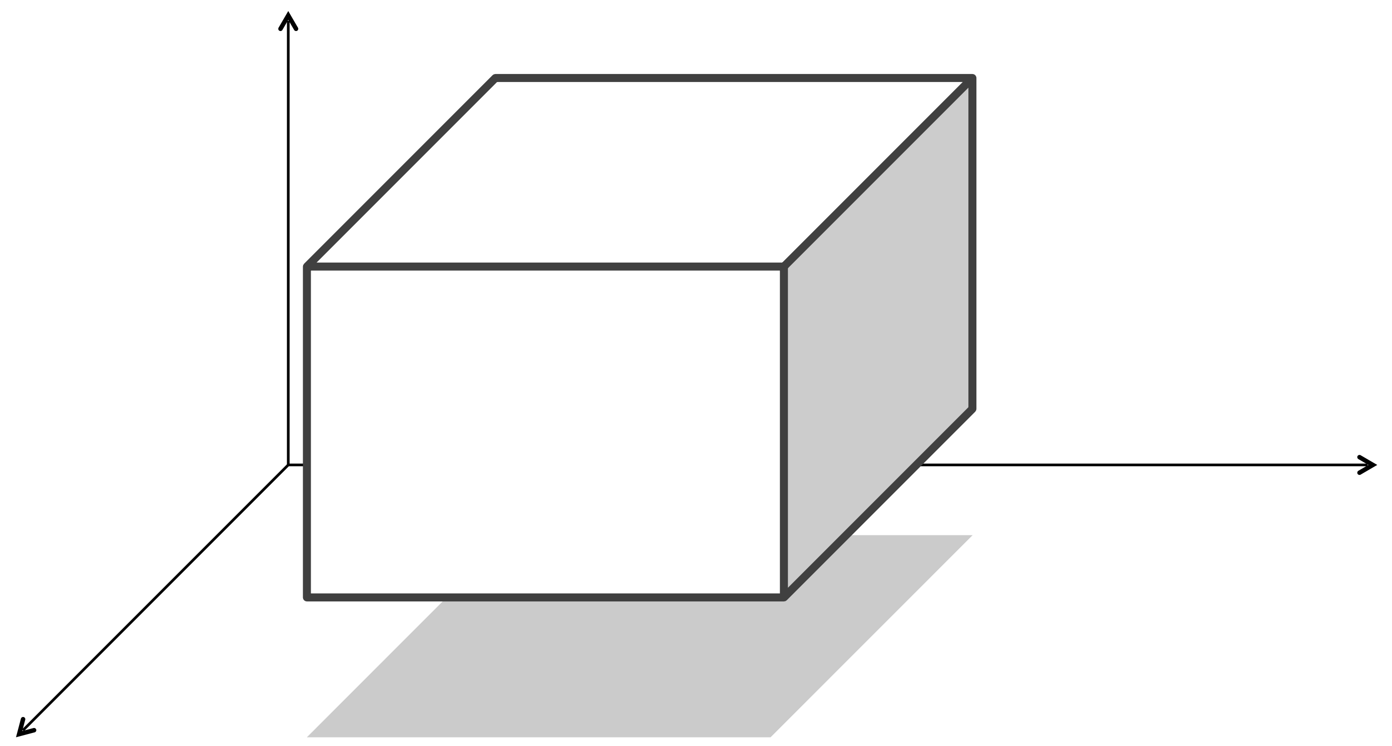
- Trivial, but useful
  - e.g. for a flat terrain, or a wall



- Storage:
  - (nx, ny, nz, k)
  - a normal, a distance from the origin
- Tests are trivial

Geometry Proxies: Axis-Aligned Bounding Box (AABB)

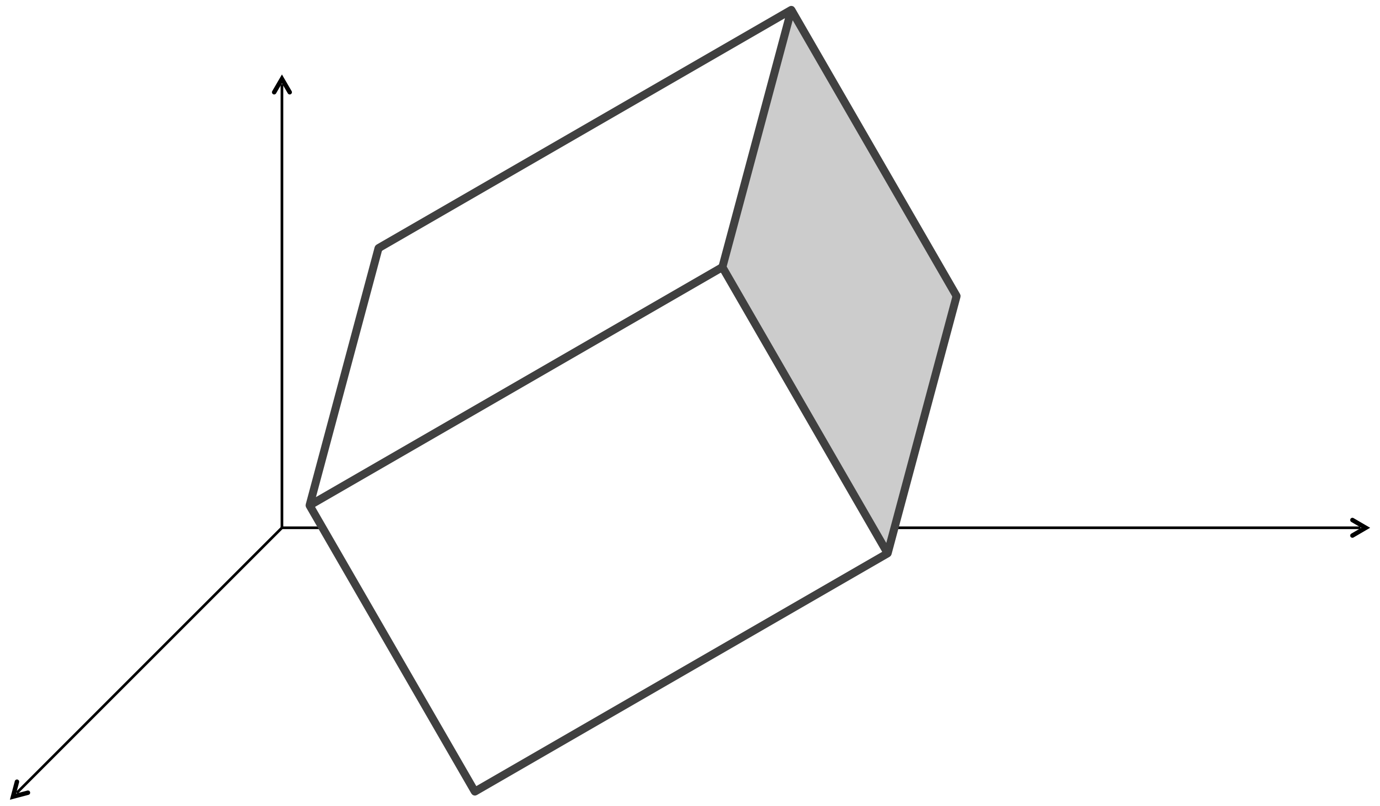
- Easy to update
- Compact (three intervals)
- Trivial to test



- It can only be translated or scaled, rotations are not supported

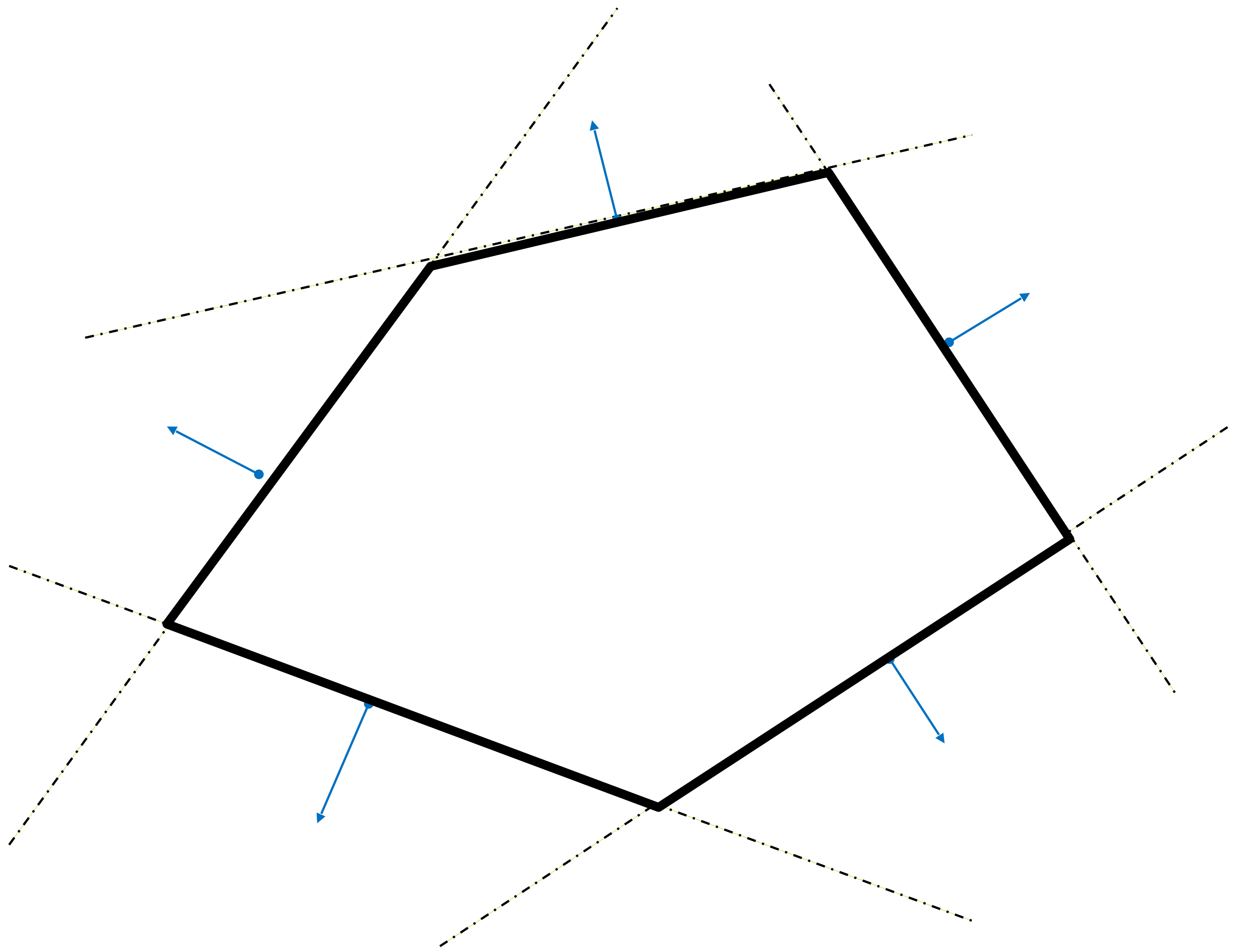
Geometry Proxies: Box

- Similar to AABB, but not axis-aligned
- More expensive to compute and store
  - You need intervals and a rotation
- Still not a great approximation, but it is invariant to rotations and it is fast to compute and use



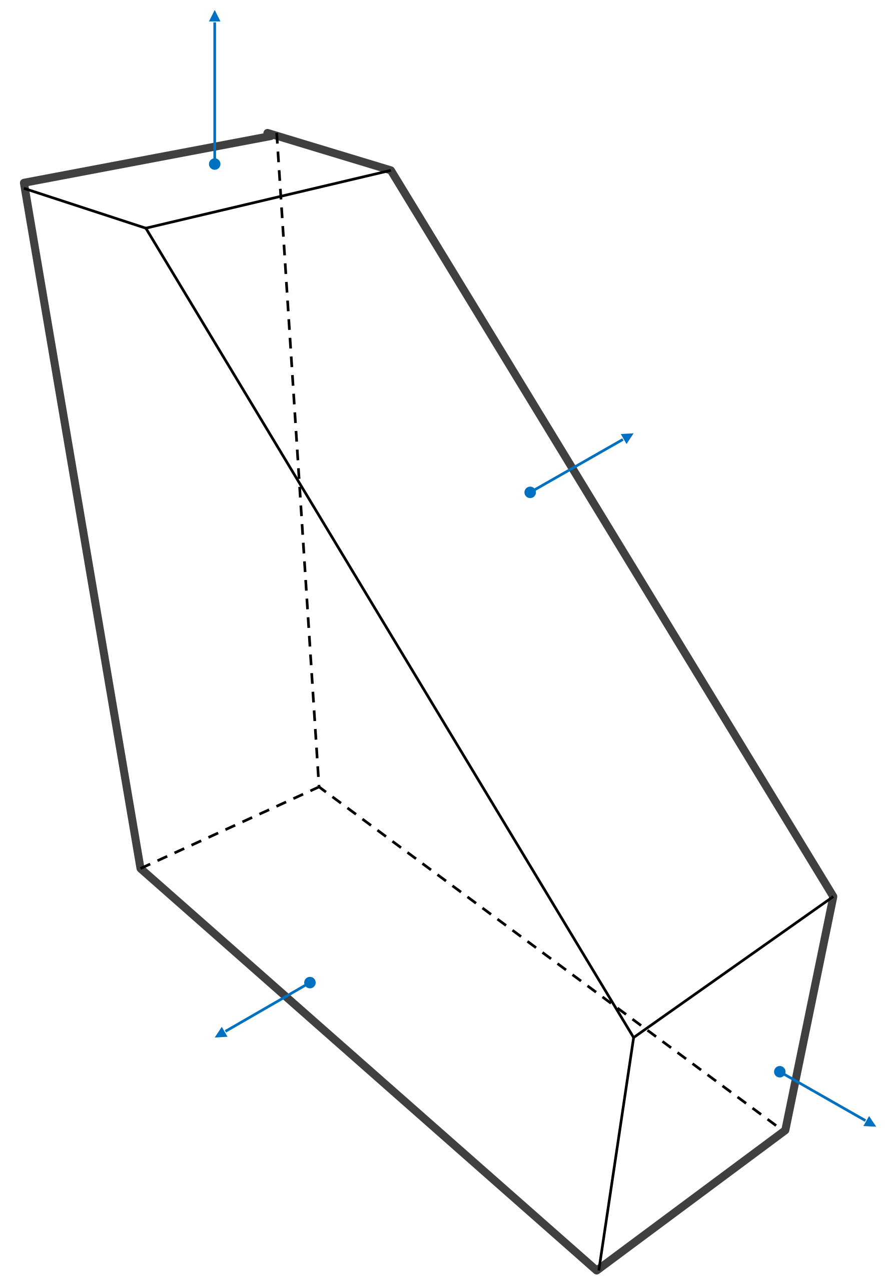
Geometry Proxies (in 2D): Convex Polygon

- Intersection of half-planes
  - each delimited by a line
- Stored as:
  - a collection of (oriented) lines
- Test:
  - a point is inside iff
    it is in each half-plane
- Good approximation
- Moderate complexity



Geometry Proxies (in 3D): Convex Polyhedron

- Intersection of half-spaces
- Similar as previous,
  but in 3D
  - Stored as a collection
    of planes
  - Each plane is a normal + distance from origin
  - Test: inside proxy iff
    inside each half-space



University of Victoria | Computer Science

Geometry Proxies (in 3D): (General) Polyhedron

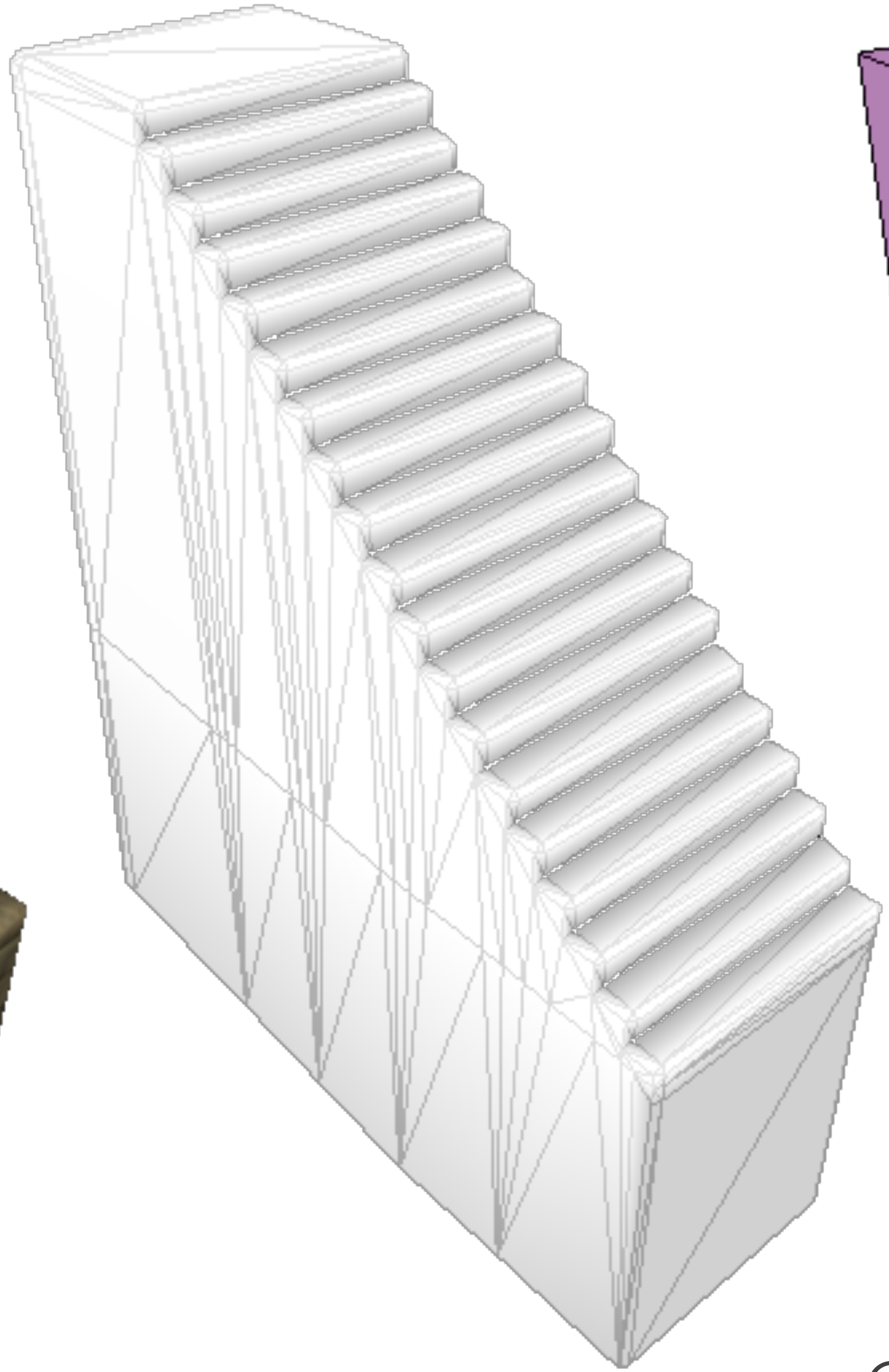
- Luxury Hit-Boxes :)
  - The most accurate approximations
  - The most expensive tests / storage
- Specific algorithms to test for collisions
  - requiring some preprocessing
  - and data structures (BSP-trees)
- Creation (as meshes):
  - sometimes, with automatic simplification
  - often, hand made (low poly modelling)

3D Meshes as Hit-Boxes

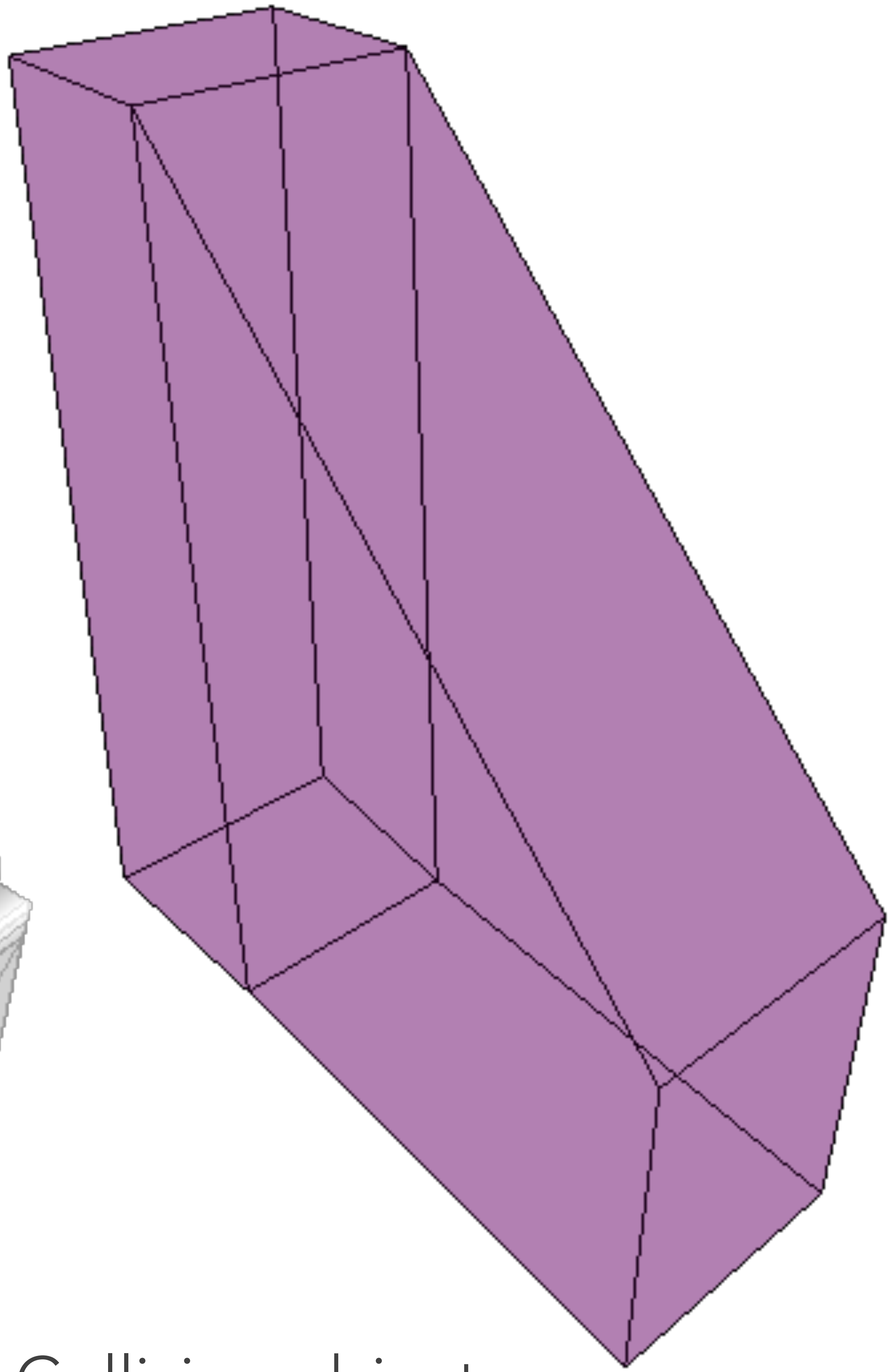
- These are often NOT the meshes that you use for rendering
  - much lower resolution (~ $O(10^2)$)
  - no attributes (no uv-mapping, no col, etc)
  - closed, water-tight (inside != outside)
  - often convex only
  - can be polygonal (as long as the faces are flat)

3D Meshes as Hit-Boxes

mesh for rendering
(~600 tri faces)



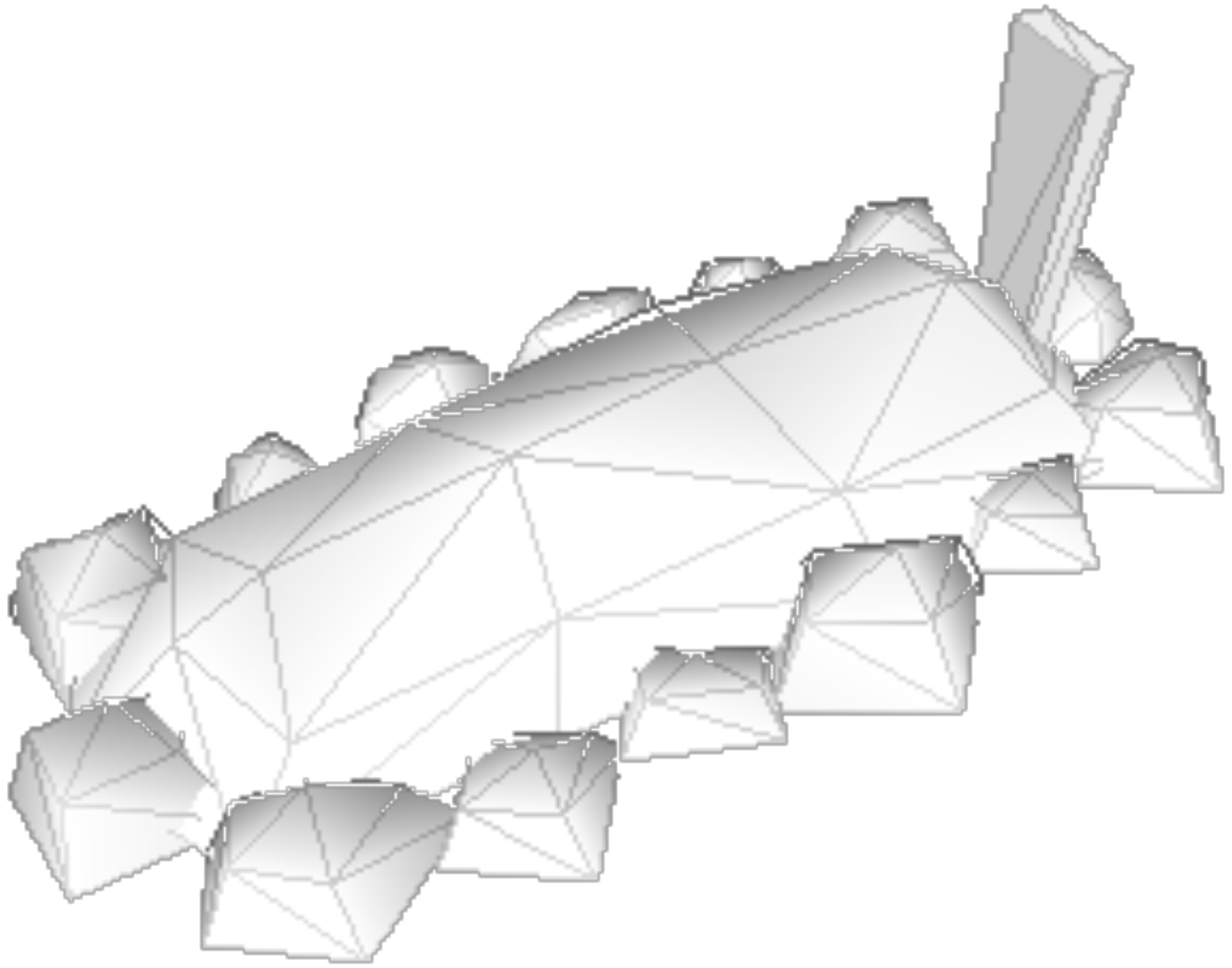
(in wireframe)



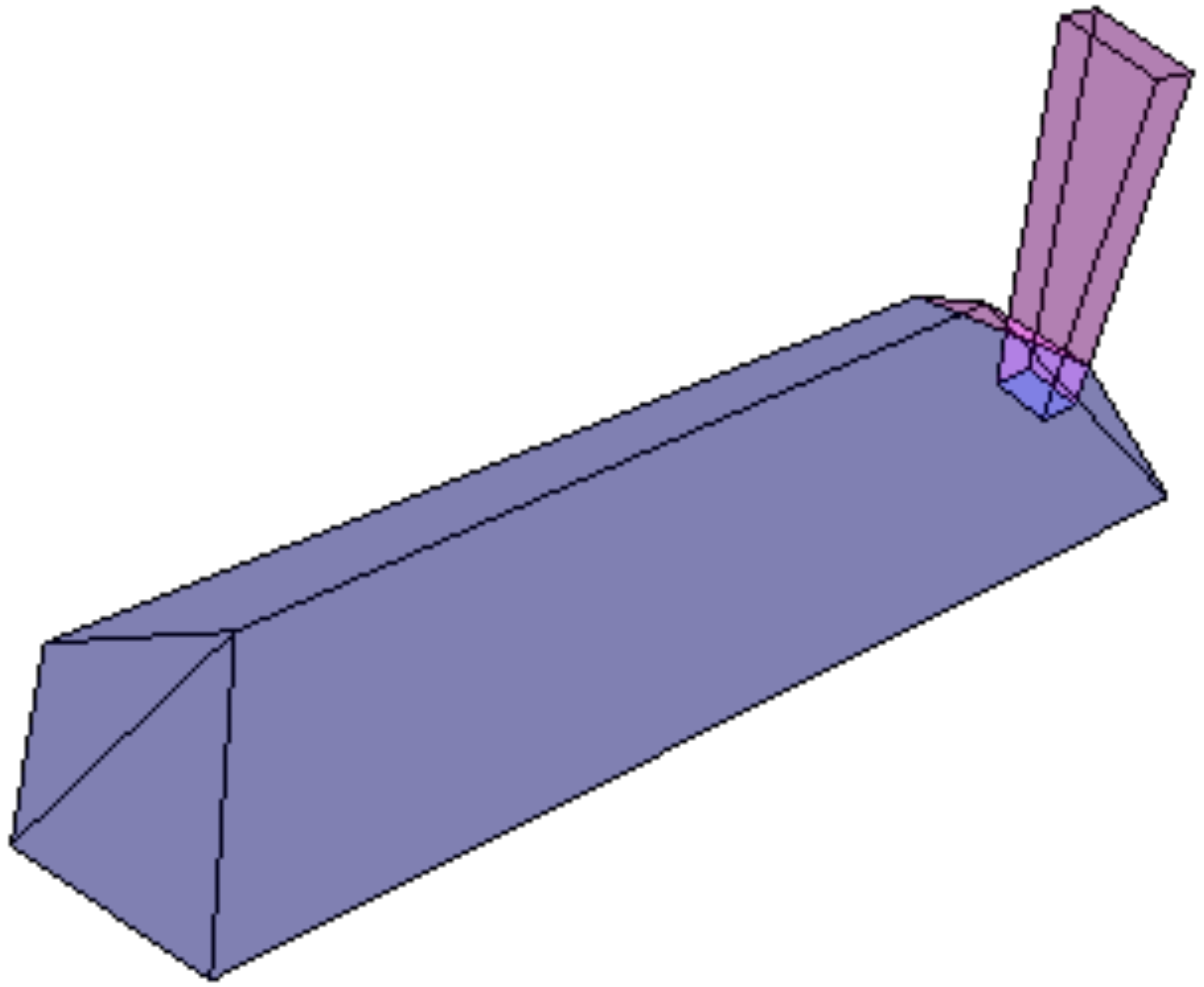
Collision object:
10 (polygonal) faces

3D Meshes as Hit-Boxes

mesh for rendering
(~300 tri faces)



(in wireframe)



Collision object:
12 (polygonal) faces

Geometry Proxies: Composite Hit-Boxes

- Union of Hit-Boxes
  - inside *iff* inside of *any* sub Hit-Box

- Flexible

  - union of **convex** Hit-Boxes ==> **concave** Hit-Box
  - shape partially defined by a sphere, partially by a box ==> better approximation

- Creation: typically by hand
  - (remember: hit-boxes are usually **assets**)

How To Choose The Proxy?

- Application dependent
- Note: # of intersection tests to be implemented quadratic wrt # of types supported

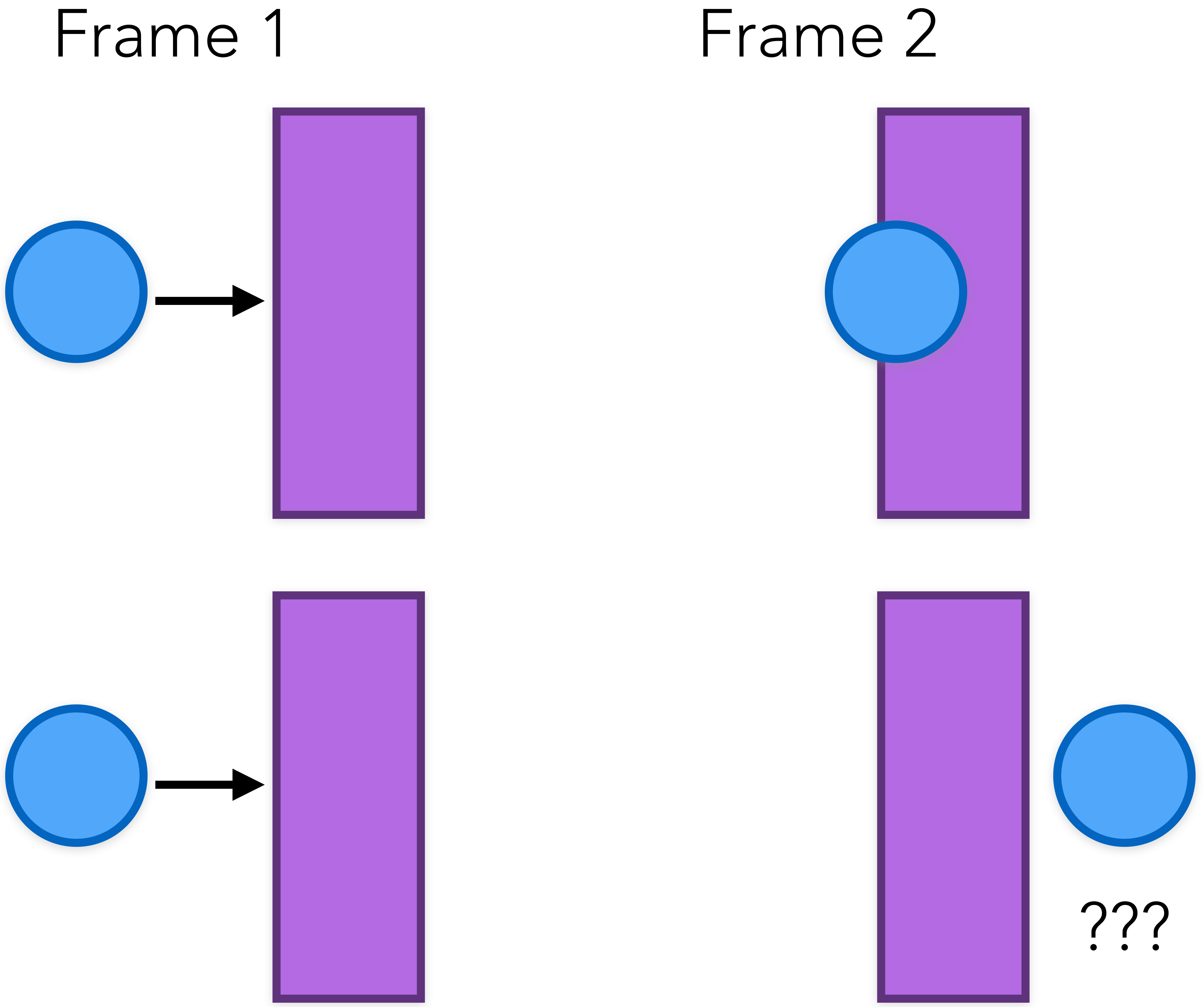
vs	Type A	Type B	Type C	Point	Ray
Type A	*algorithm*	*algorithm*	*algorithm*	*algorithm*	*algorithm*
Type B		*algorithm*	*algorithm*	*algorithm*	*algorithm*
Type C			*algorithm*	*algorithm*	*algorithm*

useful,
e.g.
for visibility

Collision Detection Strategies

- **Static** Collision detection
  - ("a posteriori", "discrete")
  - approximated
  - simple + quick

- **Dynamic** Collision detection
  - ("a priori", "continuous")
  - accurate
  - demanding



Existing Implementations

- Intel Embree - BVH Tree - https://embree.github.io
- Nori - BVH - https://github.com/wjakob/nori
- Approximate knn - https://www.cs.umd.edu/~mount/ANN/
- Intersections - http://www.realtimerendering.com/intersections.html

References

Foundations of Multidimensional and Metric Data Structures
Hanan Samet

http://www.realtimerendering.com/books.html

http://www.realtimerendering.com/intersections.html

Polygon Mesh Processing
Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, Bruno Levy

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**
Chapter 12