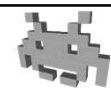



Course Plan



lec. 1: **Introduction** ●

lec. 2: **Mathematics** for 3D Games ●●●●●●

lec. 3: **Scene Graph** ●●

lec. 4: **Game 3D Physics** ●●●● + ●●

lec. 5: **Game Particle Systems** ●

lec. 6: **Game 3D Models** ●●

lec. 7: **Game Textures** ●●

lec. 8: **Game 3D Animations** ●●●

lec. 9: **Game 3D Audio** ●

lec. 10: **Networking** for 3D Games ●


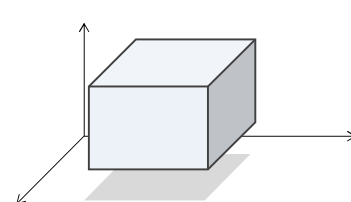
lec. 11: **Artificial Intelligence** for 3D Games ●

lec. 12: **Game 3D Rendering Techniques** ●●

Let's continue the discussion on **geometry proxies** for **collision detection**

44

Geometry proxies: «AABB»



Misnomer: not necessarily a "bounding" volume: could be used as a collider too

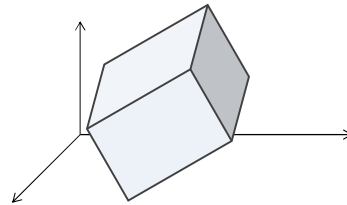
Axis Aligned (Bounding) Box

- Easy to compute / update
- Concise to store
 - Hint: it's three interval: on X, on Y, on Z
- Easy to test for collision VS a point, or another AABB, etc
- Transforms:
 - ☹️ ☹️ ☹️ cannot be rotated
 - can be easily scaled / translated

45

Geometry proxies: Box

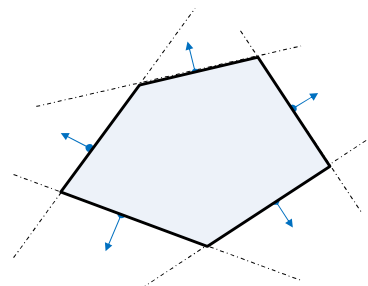
- “Parallelepiped”
 - non axis aligned
 - generalized version of AABB
 - storage:
 - a rotation +
 - an AABB
 - Can be freely transformed
 - note: only if scaling is uniform
 - Tests: a more computations needed



46

Geometry proxies (in 2D): a Convex Polygon

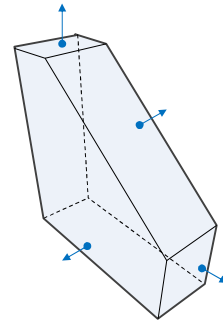
- Intersection of half-planes
 - each delimited by a line
- Stored as:
 - a collection of (oriented) lines
- Test:
 - a point is inside the proxy iff it is in each half-plane
- Flexible (good approximations)... and still moderate complexity



47

Geometry proxies (in 3D): a Convex Polyhedron

- Intersection of half-space
- Similar as previous, but in 3D
 - stored as a collection of planes
 - each plane = a vec4 (normal, distance from origin)
 - tests: inside the proxy iff inside each half-space



48

Geometry proxies (in 3D): a (general) Polyhedron

- potentially **concave**
- Luxury **Colliders** :) ← not worth it for a **Bounding Volume** !
 - The most **accurate** approximations
 - The most **expensive** tests / storage
- Specific algorithms to test for collisions
 - requiring some preprocessing
 - and data structures (**BSP-trees**, see later)
- Creation (as meshes):
 - sometimes, with automatic simplification
 - often, hand-designed by artists (low poly modelling)
 - collision proxies are **assets**!
- Similar to a 3D mesh used for rendering?
 - Many differences (compare with mesh, lecture 6)

49

3D meshes for geometry proxies vs 3D meshes for rendering



see lecture on 3D models later

- Proxy meshes are
 - much **lower res** (e.g. $< 10^2$ faces)
 - no **attributes** (no uv-mapping, no color, etc)
 - based **generic polygons**, not just **tris** (as long as they are *flat*)
 - **closed**, **water-tight** (inside \neq outside)
 - sometimes: **convex** only
 - completely different internal data structures (e.g. set of bounding planes)

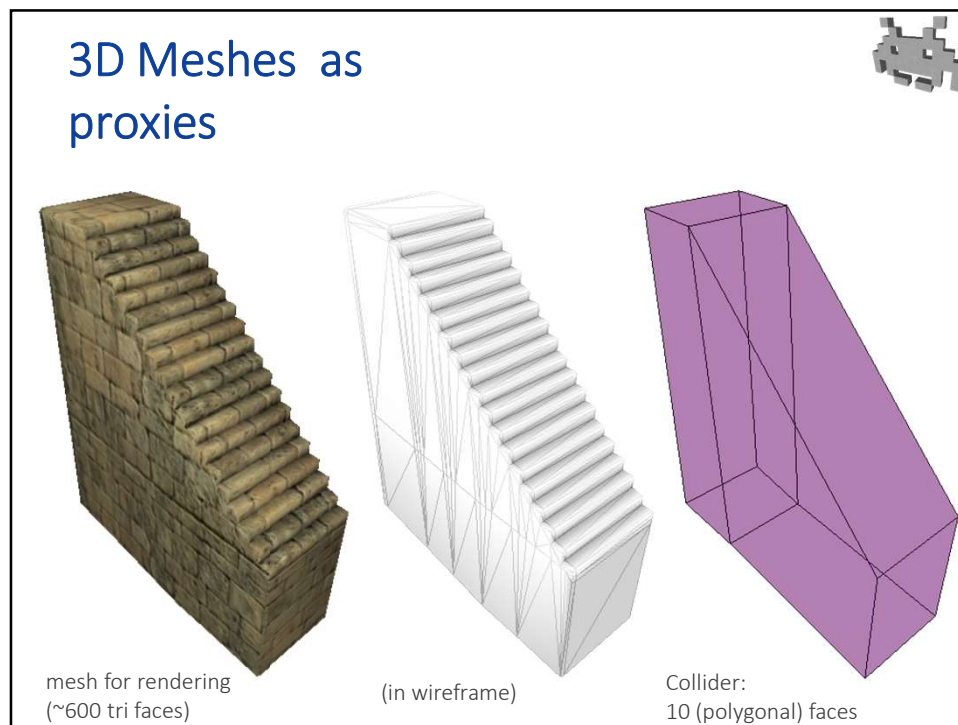
50

Geometry proxies: composition of many proxies

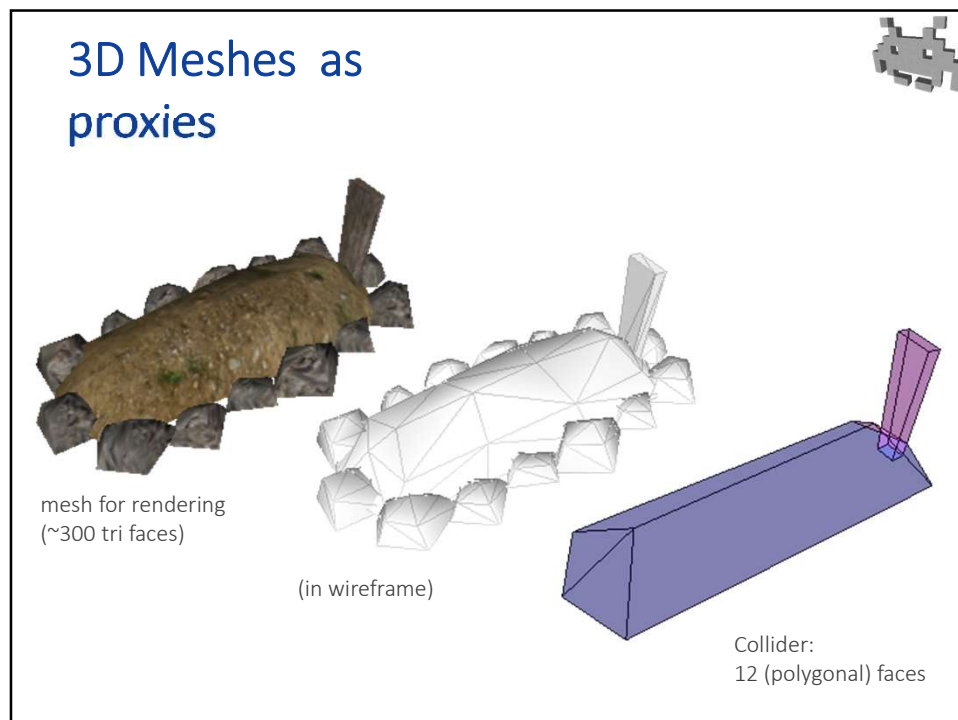


- A proxy can be a union of sub-proxies
 - inside the proxy *iff* inside of *any* sub proxy
- Very expressive!
 - better approximation for many objects, even with very few proxies
 - note: union of **convex** proxies can be **concave** !
- Still quite efficient to store / test
- Very difficult to construct automatically
 - Open problem!

51



52

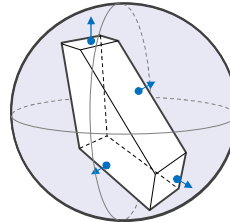


53

Bounding Volume + Collision Object

```
if (!intersect( boundingVol, X ) )
{
    // nothing to do: early reject!
}
else {
    CollisionData d;
    if (collide( hitBox, X , &d ))
    {
        collision_response( d );
    }
}
```

note: **intersect** and **collide**
aren't the same function here



a simpler
Bounding Volume
around
a more complex
Collision Object
approximating
the same object

54

How to construct geometry proxies?

- “Given an object representation M , build an appropriate proxy for it”
 - a M = 3D model of e.g. a dragon, a castle, a character...
- It's a difficult task to automatize
 - especially for **colliders**
 - it's a bit easier for **bounding volumes**
 - especially if we want to pick simpler (more efficient) proxies
 - such as collection of a few spheres, capsules, boxes
 - especially if we want good approximations
- It's often done manually by digital artists

Geometry proxies are **assets** !

55

Collision detection: Static

- Check for collision only after each step
- Problem: non-penetration is temporarily violated
 - patching it in **collision response**
not always easy
- Problem: «tunneling»
 - Can happen if:
 - dt too large,
 - or, speed too large
 - or, objects too thin

aka { «static» (because objects are tested as if they are still)
«a posteriori» (because coll. are detected after they happen)
«discrete» (because we check at discrete time intervals)

Diagram illustrating static collision detection. At time t , a blue circle is moving towards a green rectangle. At time $t + dt$, the circle has moved past the rectangle. Both states are labeled "NO COLLISION".

56

Collision detection: strategies

- **Static** Collision detection
 - ("a posteriori", "discrete")
 - approximated
 - simple + quick
- **Dynamic** Collision detection
 - ("a priori", "continuous")
 - accurate
 - demanding

Diagram illustrating collision detection strategies. The top part shows static collision detection: at time t , a blue circle is moving towards a green rectangle; at time $t + dt$, the circle has moved past the rectangle. The bottom part shows dynamic collision detection: at time t , a blue circle is moving towards a green rectangle; at time $t + dt$, the circle is shown at two positions, with a red arrow indicating a "COLLISION" with the rectangle.

57

Collision detection: Dynamic

- Much more accurate detection
- Bonus:
 - no need to «teleport the object in the safe position».
 - it never left a safe position!
 - preventing penetrations easier than curing them.
- Much more difficult to do, too
 - for one-way collision: check the penetration between the static object and the volume **swept** (ita: *spazzato*) by the moving object *during the entire duration of the frame*
 - easy for: points (swept volume = segment)
 - easy for: spheres (swept volume = capsule – which one?)
- Basically, practical to apply only in these cases
 - and when required

aka {
«dynamic»
(because moving objects
are tested)
«a priori»
(because coll. are detected
before they happen)
«continuous»
(because it is checked
over a time interval)

58

Dirgression: collision detection in traditional 2D games

- A much easier problem
- We can leverage **collision detection for 2D sprites**
 - it's accurate: «**pixel perfect**»
 - it's efficient: **HW supported**
(hard-wired support like sprite rendering)
 - little need for **proxy** approximations for colliders
 - good proxy for bounding volumes: sprite rectangle

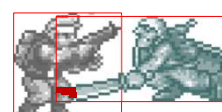
in screen space



NO COLLISION



NO COLLISION



COLLISION

59

Collision detection



- Efficiency issues:

a) test between object pairs:

- Must be efficient

b) avoid quadratic explosions of needed tests

- N objects $\rightarrow N^2$ tests ?

60

How to avoid a quadratic explosions of needed tests



- Classes of solutions:

1) **spatial indexing** structures

2) BVH – Bounding Volume Hierarchies

61

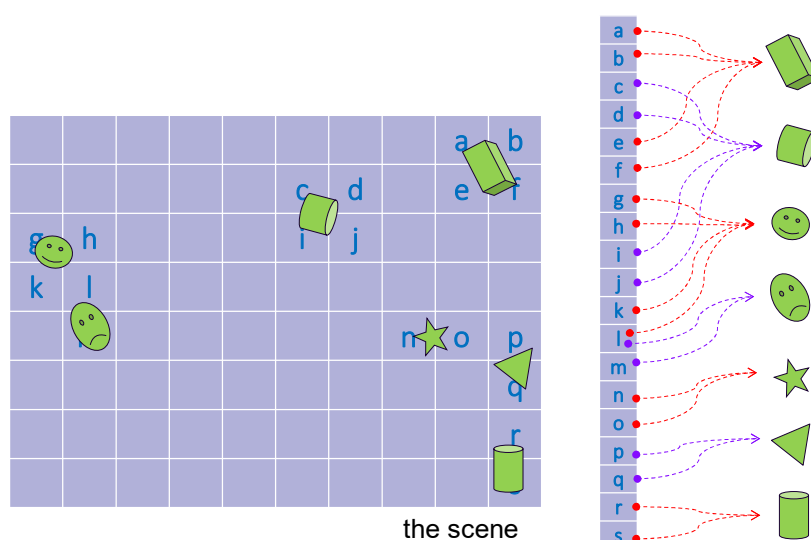
Spatial indexing structures



- 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. Cheap! 😊
 - for **moving** parts of the scene, an update! Consuming! ☹
 - (another good reason to tag them)
 - (2) access / usage
 - as fast as possible
- Commonest structures (in games):
 - **Regular Grid**
 - **kD-Tree**
 - **Oct-Tree**
 - and it’s 2D equivalent: the **Quad-Tree**
 - **BSP Tree**

62

Regular Grid (or: lattice)



63

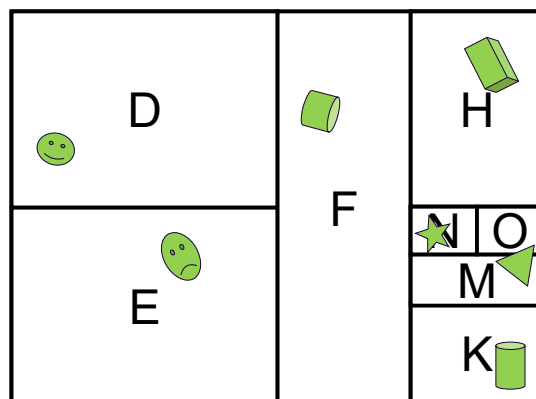
Regular Grid (or: lattice)



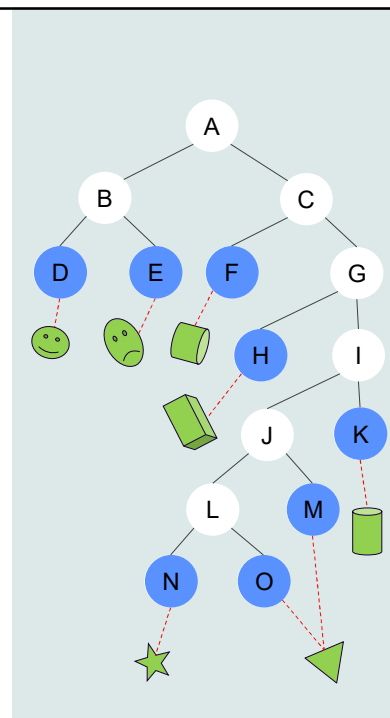
- Array 3D of cells (all the same size)
 - each cell = a list of pointers to collision objects
- Indexing function:
 - Point3D \rightarrow cell index, (constant time!)
- Construction: ("scatter" approach)
 - for each object B, find all the cells it touches, add a pointer to B to them
- Queries: ("gather" approach)
 - given query point p ,
return all object in corresponding cell and adjacent ones
- Difficult choice: cell size
 - too small: memory occupancy explodes
 - too big: too many objects in one cell (not efficient)
- Problem: RAM size
 - Cubic with resolution!
 - Most cells are empty: hash tables can be used to balance efficiency / storage-update cost

64

kD-trees



the scene



65

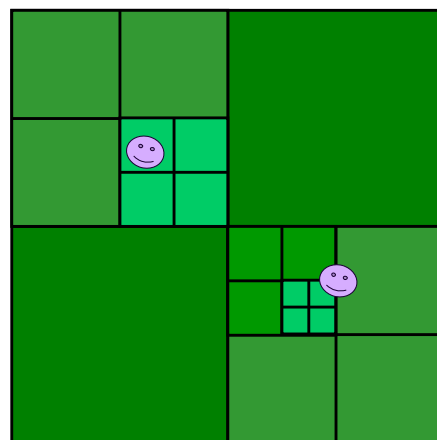
kD-trees



- 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

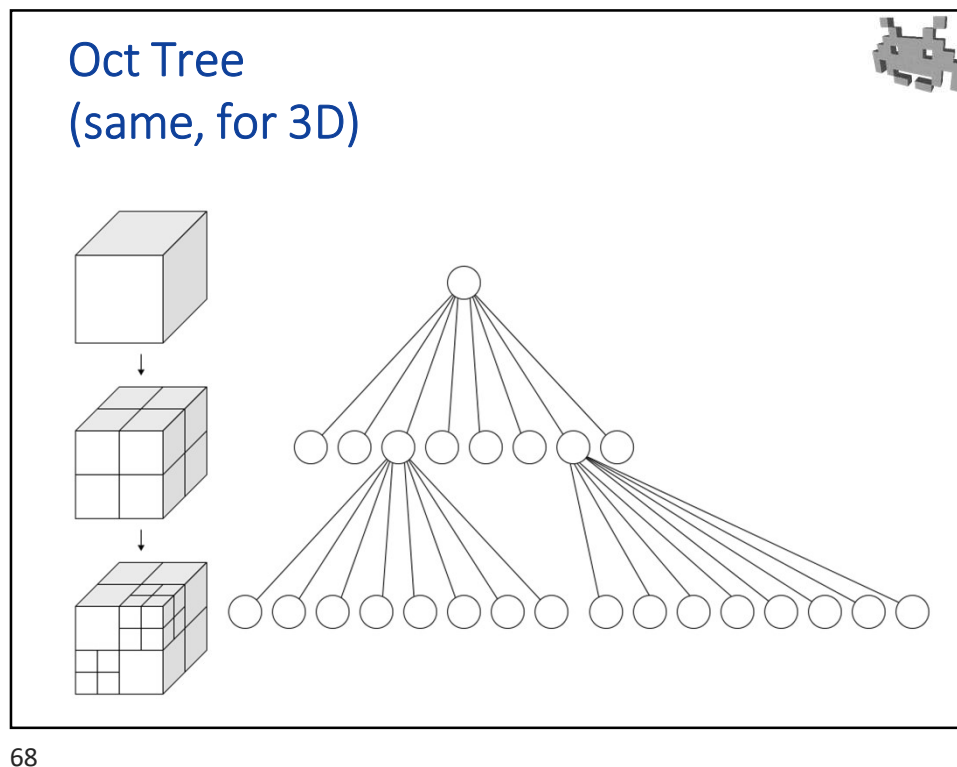
66

Quad-Tree (in 2D)



the (2D) world

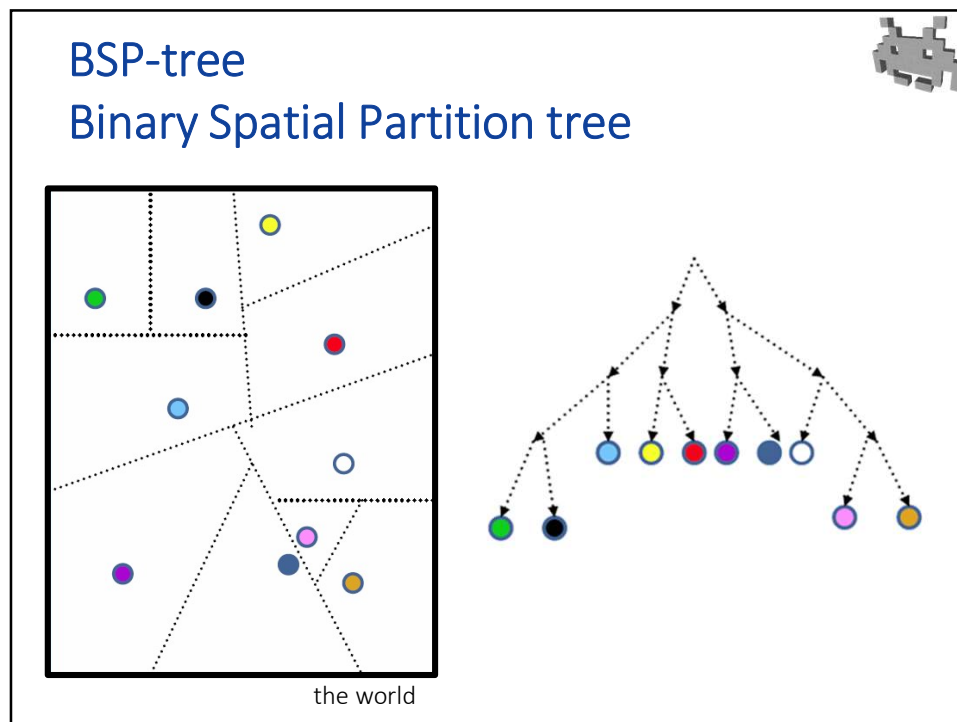
67



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

- Similar to kD-trees, but:
 - tree: branching factor: 4 (in 2D) or 8 (in 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)

69



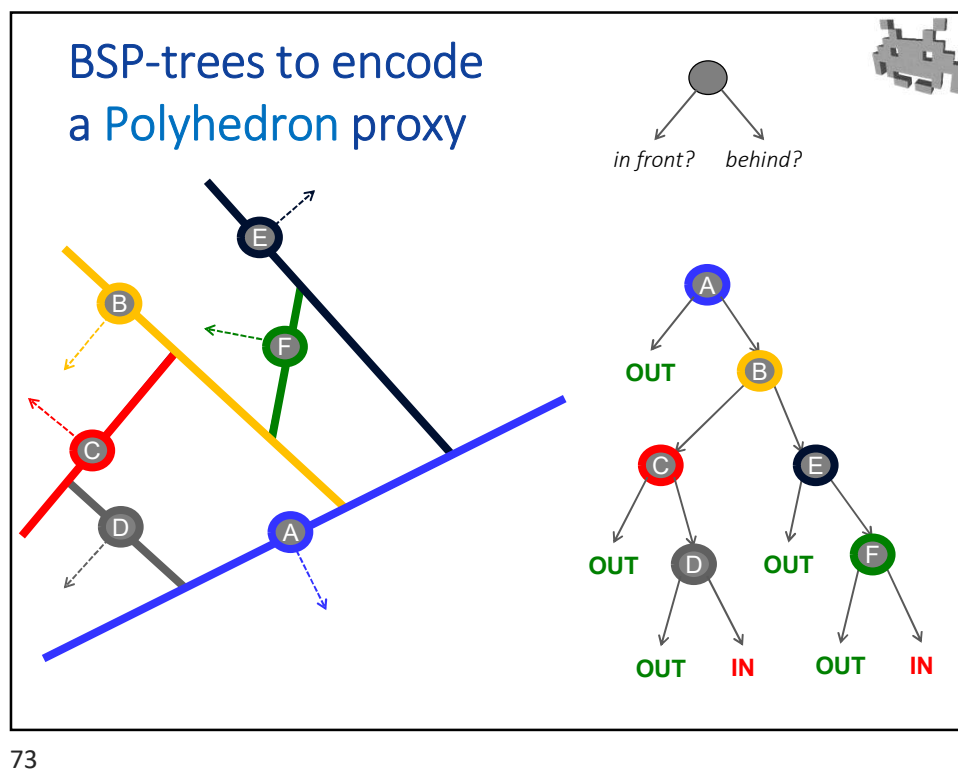
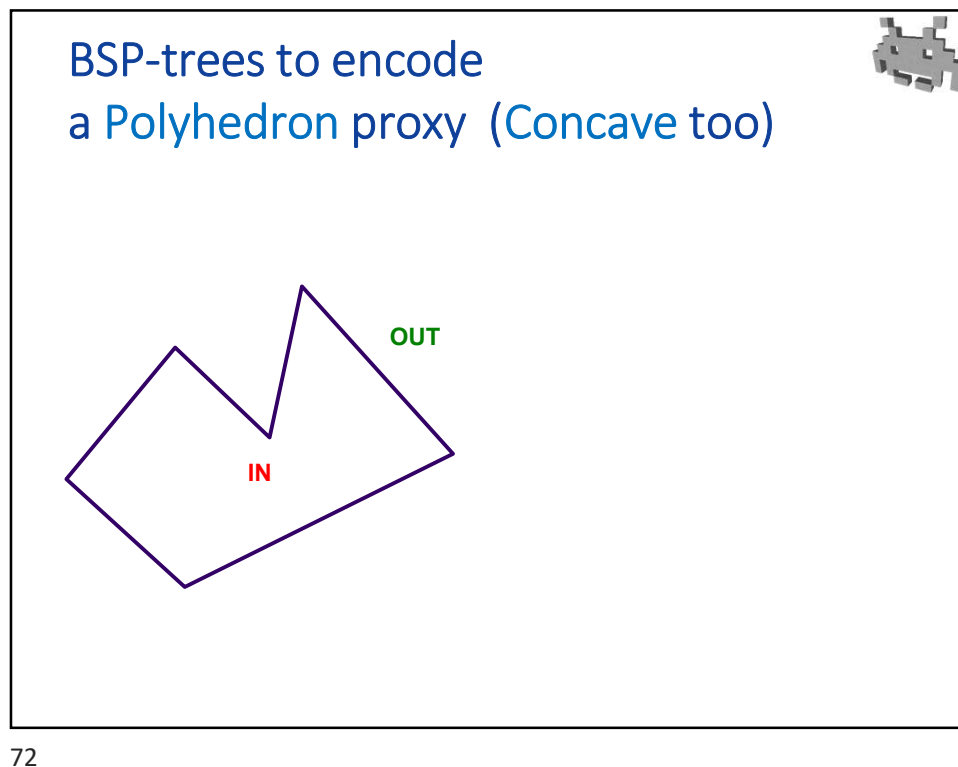
70

BSP-tree

Binary Spatial Partitioning tree

- Another hierarchical spatial structure
 - root = all scene, child-nodes = partition of parent (as usual)
 - spatial query = traverse the tree from the top down (as usual)
 - a binary tree (as *kD*-trees)
 - BUT: each node is split by an *arbitrary* plane ← in 2D: a *line*
 - plane is stored at node, as (n_x, n_y, n_z, k)
 - planes can be optimized for a given scene
 - e.g., to go for a 50%-50% object split at each node
 - e.g., to leave exactly *one* object at leaves ← assuming it is always possible to split any two apart – a reasonable assumption
 - Pro: they can be optimized for optimal queries: better query time!
 - Con: must be optimized during construction: worse construction time!
- Another use: to store/test (General) Polyhedron proxy:
 - note: planes are stored in its *object space*
 - each leaf: a bit: inside or outside
 - tree is precomputed for a given Collision Polyhedron

71



How to avoid a quadratic explosions of needed tests



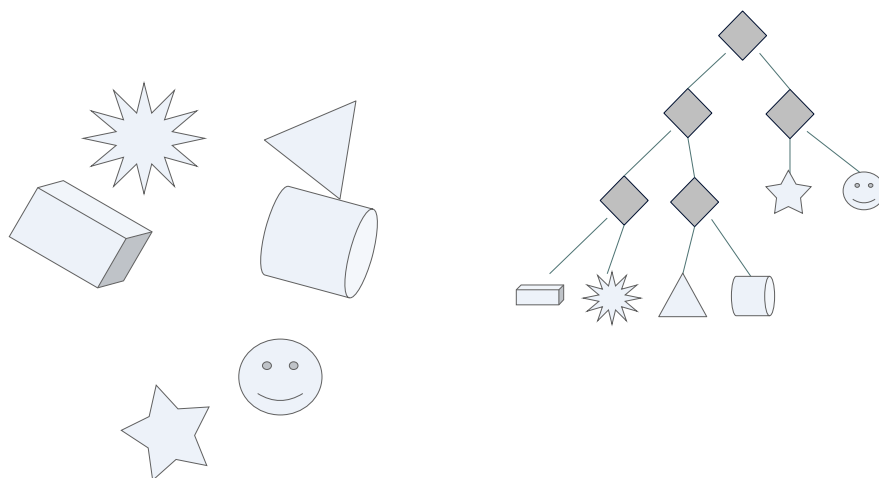
- Classes of solutions:

1) *spatial indexing* structures

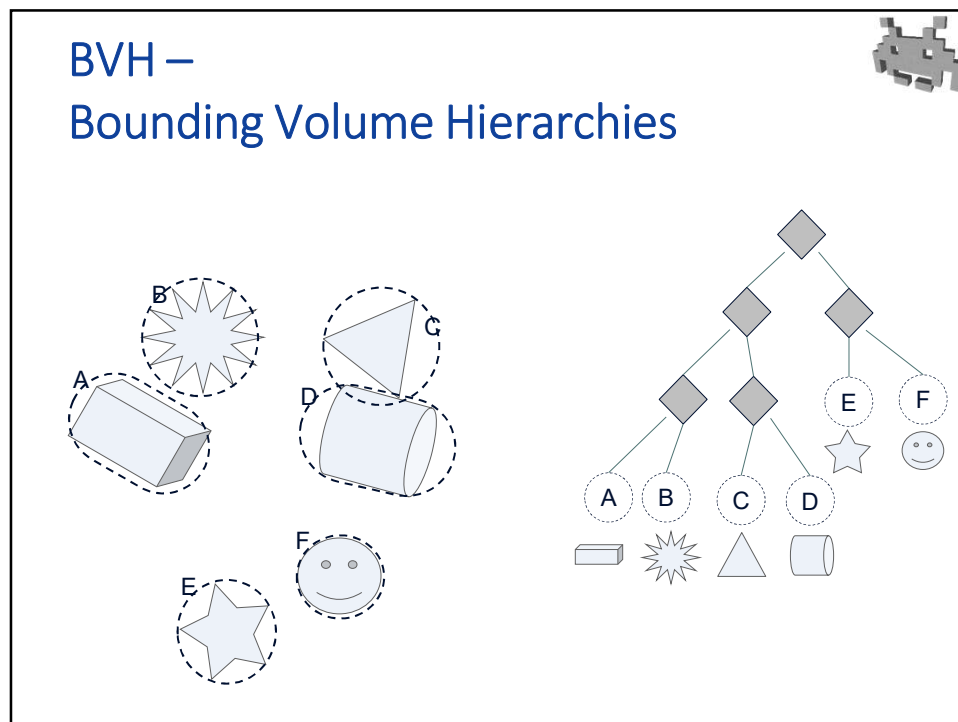
2) BVH – Bounding Volume Hierarchies

74

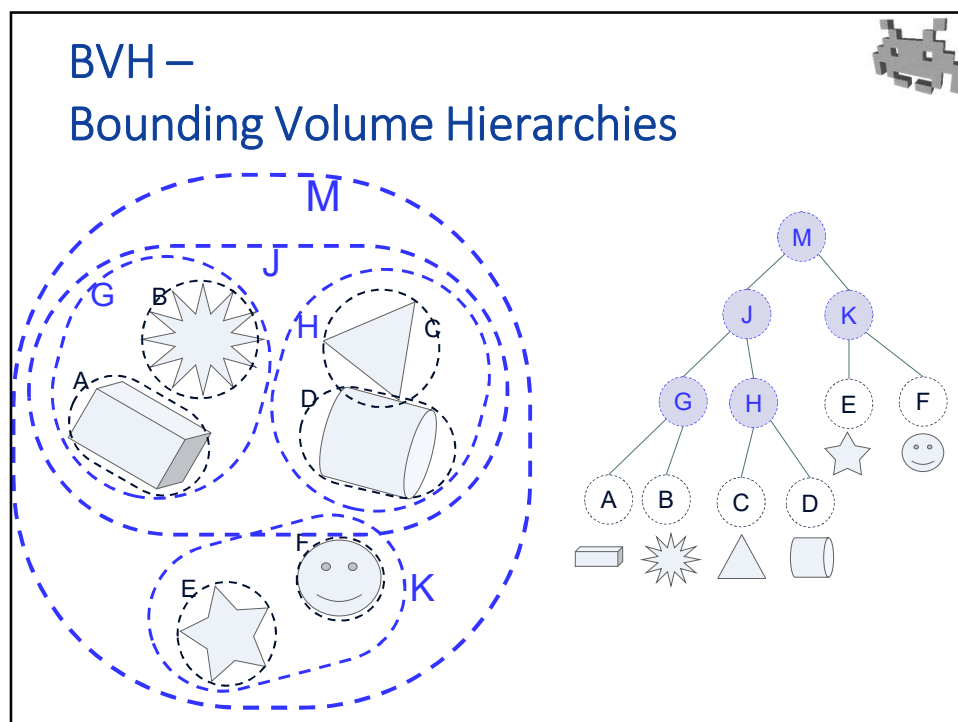
BVH Bounding Volume Hierarchy



75



76



77

BVH

Bounding Volume Hierarchy



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

78

Spatial indexing structures

Recap



- **Regular Grid**
 - 😊 the most parallelizable (to update / construct / use)
 - 😊 constant time access (best!)
 - ☹ quadratic / cubic RAM space (2D, 3D) – unless hashing
- **kD-tree, Oct-tree, Quad-tree**
 - 😊 compact
 - 😊 simple
- **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 Polyhedral Collider)
- alternative: **BVH**
 - 😊 simplest construction
 - ☹ non necessarily super efficient to access
 - may need to traverse multiple children
 - if uses same hierarchy of the scene-graph: not always the best
 - **ideal for dynamic parts of the scene?**

79

Physics Engine: an implementation problem



- Task: **Dynamics**:
 - (forces, speed and position updates...)
 - simple structures, fixed workflow
 - highly parallelizable: **GPU** possible
- Task: **Constraints Enforcement**:
 - still moderately simple structures, fixed workflow
 - problem: collision constraints not known a-priori
 - still highly parallelizable: hopefully, **GPU** possible
- Task: **Collisions Detection**:
 - non-trivial data structures, hierarchies, recursive algorithms...
 - hugely variable workflow
 - (e.g.: quick on no-collision, more work to do when the rare collisions occur)
 - difficult to parallelize: **CPU**
 - but outcome affects the other two tasks (e.g. creates constraints):
 - ==> **CPU-GPU** communication, and ==> **GPU** structures updates (problematic on many architectures)

80

Physics: that's all folks. To gather more info...



- Erwin Coumans
SIGGRAPH 2015 course
<http://bulletphysics.org/wordpress/?p=432>
- Müller-Fischer et al.
Real-time physics
(Siggraph course notes, 2008)
<http://www.matthiasmueller.info/realtimephysics/>

81