


## Course Plan



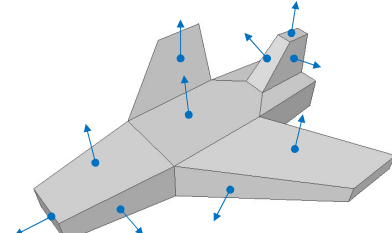
- lec. 1: Introduction ●
- lec. 2: **Mathematics** for 3D Games ●●●●●●
- lec. 3: **Scene Graph** ●
- lec. 4: Game **3D Physics** ●●●●● + ●📍
- lec. 5: Game **Particle Systems** ▸
- lec. 6: Game **3D Models** ●▸
- lec. 7: Game **Textures** ▸●
- lec. 9: Game **Materials** ▸
- lec. 8: Game **3D Animations** ▸●●
- lec. 10: **Networking** for 3D Games ●
- lec. 11: **3D Audio** for 3D Games ●
- lec. 12: **Rendering Techniques** for 3D Games ●
- lec. 13: **Artificial Intelligence** for 3D Games ●

69

## Geometry proxies a (general) Polyhedron

potentially *concave*

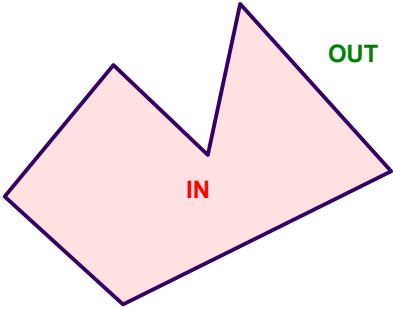
not worth it for a *Bounding Volume* !



- A... luxury **Collider**
  - The most **accurate** approximations
  - But, the most **expensive** tests / storage
- Specific algorithms to test for collisions
  - requiring some preprocessing
  - and data structures (**BSP-trees**, see next lecture)
- Creation (treat them as meshes):
  - sometimes, with automatic simplification
  - often, hand-designed by artists (low poly modelling)
- Similar to a 3D mesh used for rendering?
  - Many differences (compare with mesh, see “3D Models” lecture)

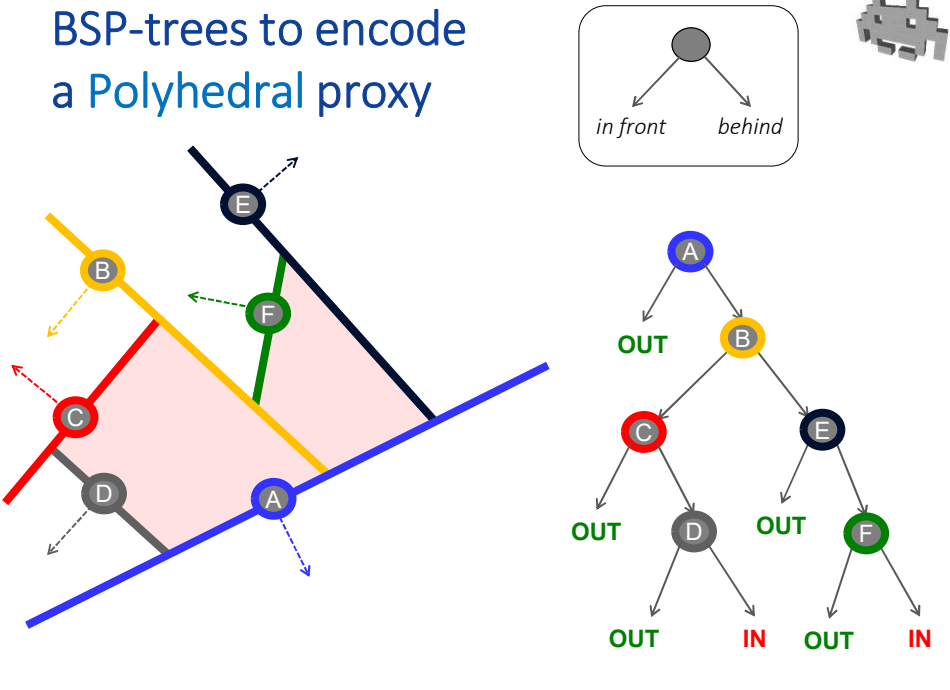
70

### BSP-trees to encode a Polyhedral proxy (Concave too)



72

### BSP-trees to encode a Polyhedral proxy



73

```
graph TD; A((A)) --> C((C)); A --> B((B)); B --> E((E)); B --> OUT1[OUT]; C --> OUT2[OUT]; C --> D((D)); D --> OUT3[OUT]; D --> IN1[IN]; E --> OUT4[OUT]; E --> F((F)); F --> OUT5[OUT]; F --> IN2[IN];
```

## BSP-tree (Binary Spatial Partitioning tree)



- A way to store a (convex, or concave) polyhedron
- A hierarchical structure
  - a binary tree
  - root = all space, child-nodes = partition of parent
  - each internal node is split by an *arbitrary* plane in 2D: a line
  - plane stored as  $(n_x, n_y, n_z, k)$
  - each leaf: one bit: "inside" or "outside" the proxy
  - tree is precomputed (and optimized) for a given polyhedron
  - to test a point = traverse the tree from the top down

74

## Collision detection on Polyhedral proxies: examples



- Point VS Polyhedron:  
just follow the tree, end in an IN or OUT leaf
- Sphere VS Polyhedron: more complex (think about it)
- Segment / Ray VS Polyhedron: also complex (think about it)
- Polyhedron VS Polyhedron: much more complex.  
A trace of an algorithm is:
  - Preprocessing: find and store all edges (segments) of all Polyhedra (each edge: two endpoints)
  - At testing time: test all edges of polyhedron A vs polyhedron B (segment VS polyhedron), and viceversa

75

## 3D meshes for geometry proxies vs 3D meshes for rendering (notes)

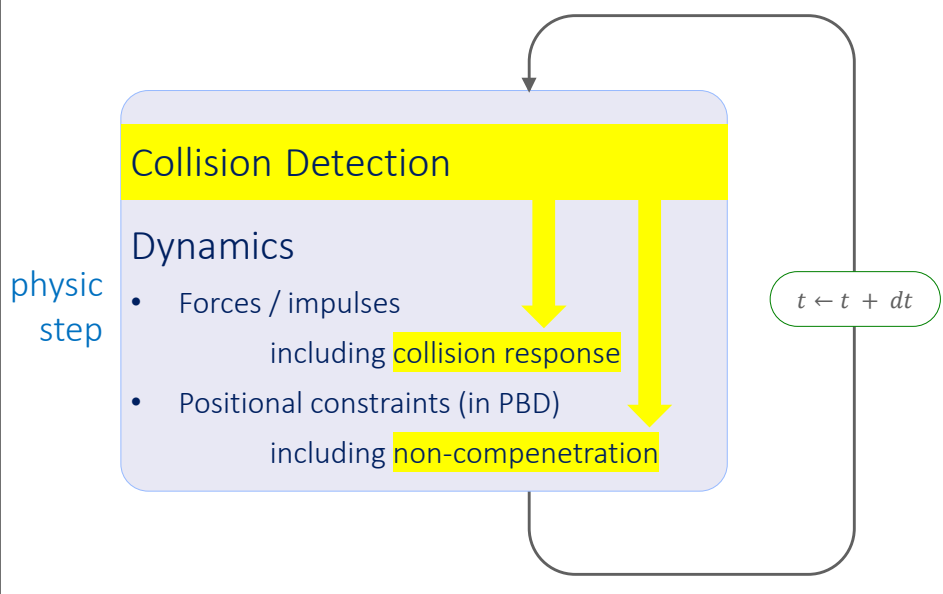


see lecture on 3D models later

- 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)
  - different internal representation:
    - if **convex** : a set of bounding planes
    - if **convex** : a BSP tree

76

## Collision detection: When?



77

## Collision detection: strategies

- Static** Collision detection
  - (“a posteriori”, “discrete”)
  - approximated
  - simple + quick
- Dynamic** Collision detection
  - (“a priori”, “continuous”)
  - accurate
  - resource consuming

78

## Collision detection: Static

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)

- 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

79

## Collision detection: Dynamic

- Much more accurate detection
- Bonus:
  - no need to «teleport the object in the safe position».
  - it never left a safe position!
  - it's easier to prevent penetrations than to heal them
- Much more difficult to do
  - 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, not practical to do in any other these
  - and even then, only use 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 temporal interval)

80

## 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 ?

81

## Collision detection: the broad phase

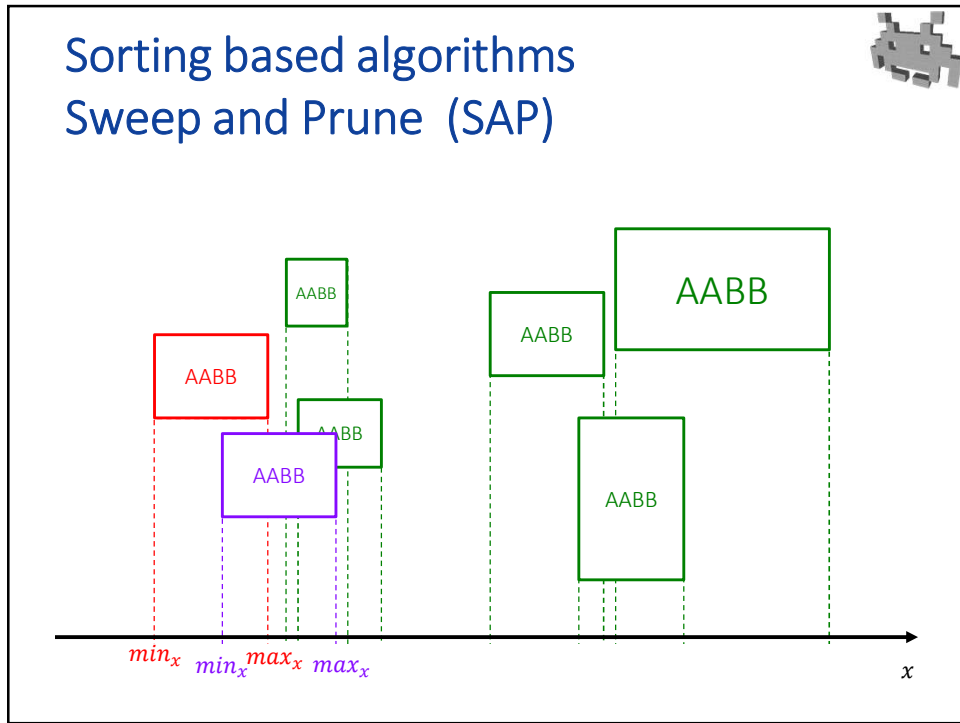
- So far, we have seen how to detect a collision between one given pair of objects
- Problem: we don't want to test every pair of objects!
- Idea: in a «**broad phase**», we quickly identify pairs of objects that need testing
  - Objects that are safely far from each other are never even tested
  - Only objects that are... “suspiciously close” must be tested
- Note: the broad phase must be *strictly conservative*
  - **not ok**: discard object pairs that actually collided,
  - **ok**: test objects that *didn't* actually collide
- Let's see strategies to do so

82

## The «broad-phase» of coll. detection (avoiding quadratic explosion of # of tests)

- Classes of solutions:
  - 1) Sorting-based algorithms
  - 2) **spatial indexing** structures
  - 3) BVH – Bounding Volume Hierarchies

83



84

### Sweep And Prune (SAP) strategy (or "Sort and Sweep")

1. **Bound:**
  - Quickly find the AABB for each collider (in its current rotation + translation)
  - E.g.: use the AABB encapsulating the transformed Bounding Sphere
2. **Sort**  $min_x$  and  $max_x$  of all AABB together
  - Just adjust the sorting used in the previous frame
  - It will be already *almost* sorted! To exploit this...
  - use an *incremental* sorting algorithm, such as quicksort
3. **Sweep** the sorted intersections, from smaller to larger
  - Quickly detect intersecting intervals in  $x$  (how?)
4. **Prune:** among AABB intervals, ignore the ones that don't *also* intersect in both  $y$  and  $z$ 
  - Test the other pairs for collision

only  $O(n \log n)$

Even faster!  $O(n)$

85



## The «broad-phase» of coll. detection (avoiding quadratic explosion of # of tests)



- Classes of solutions:
  - 1) Sorting-based algorithms
  - 2) spatial indexing structures
  - 3) BVH – Bounding Volume Hierarchies

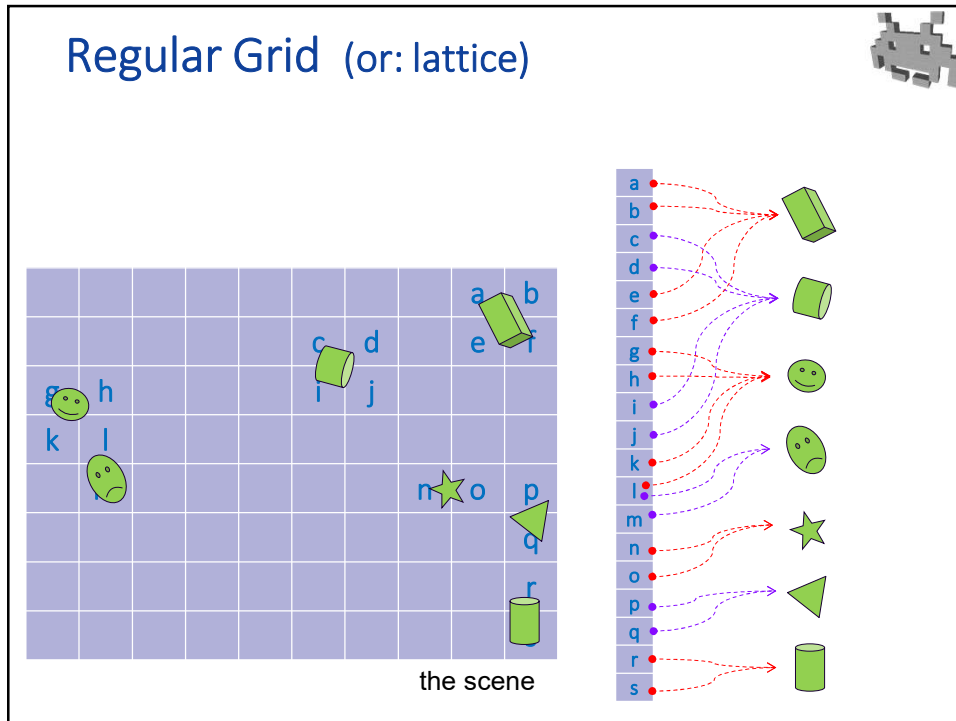
86

## Spatial indexing structures



- Data structures to accelerate queries of the kind:  
“I’m in this 3D pos. Which object(s) are around me, if any?”
- 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:
  - **Regular Grid**
  - **kD-Tree**
  - **Oct-Tree**
    - and its 2D equivalent: the **Quad-Tree**
  - **BSP Tree**

87

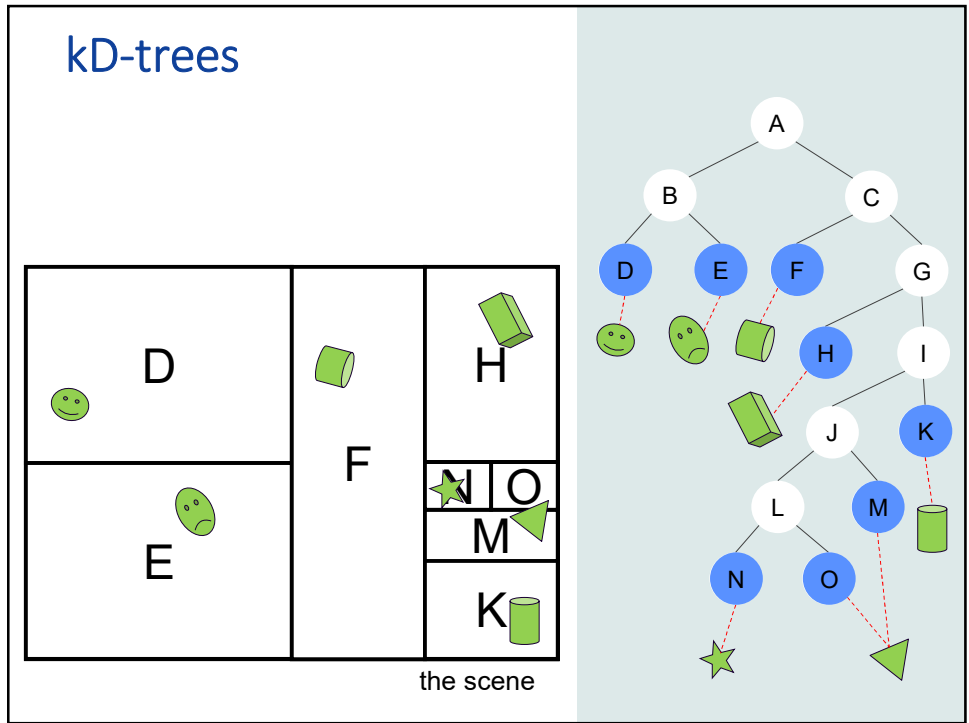


88

### 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

89

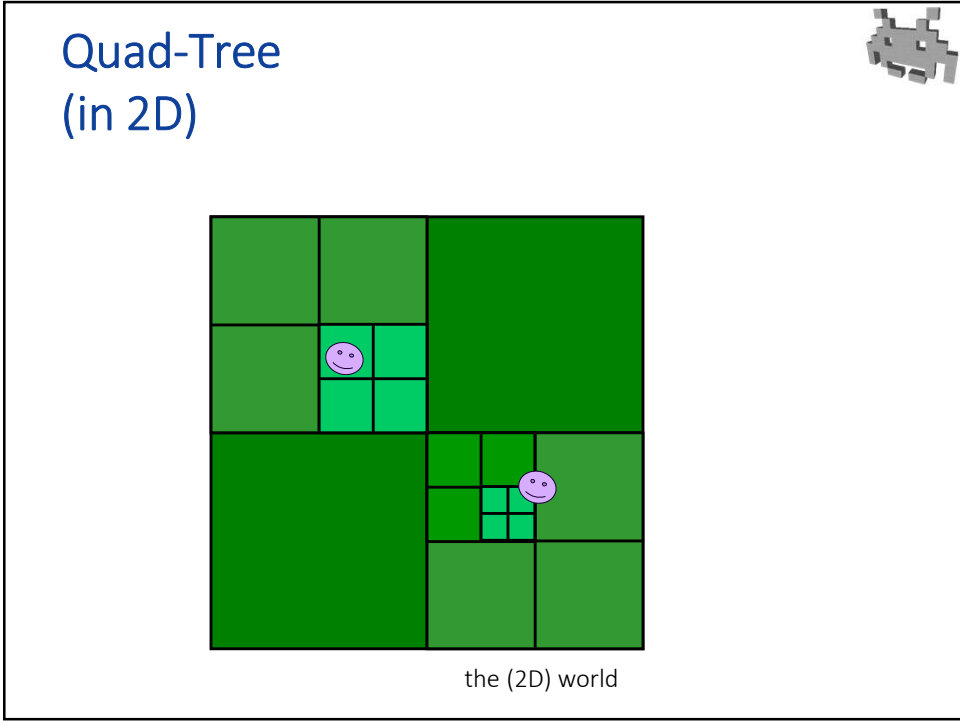


90

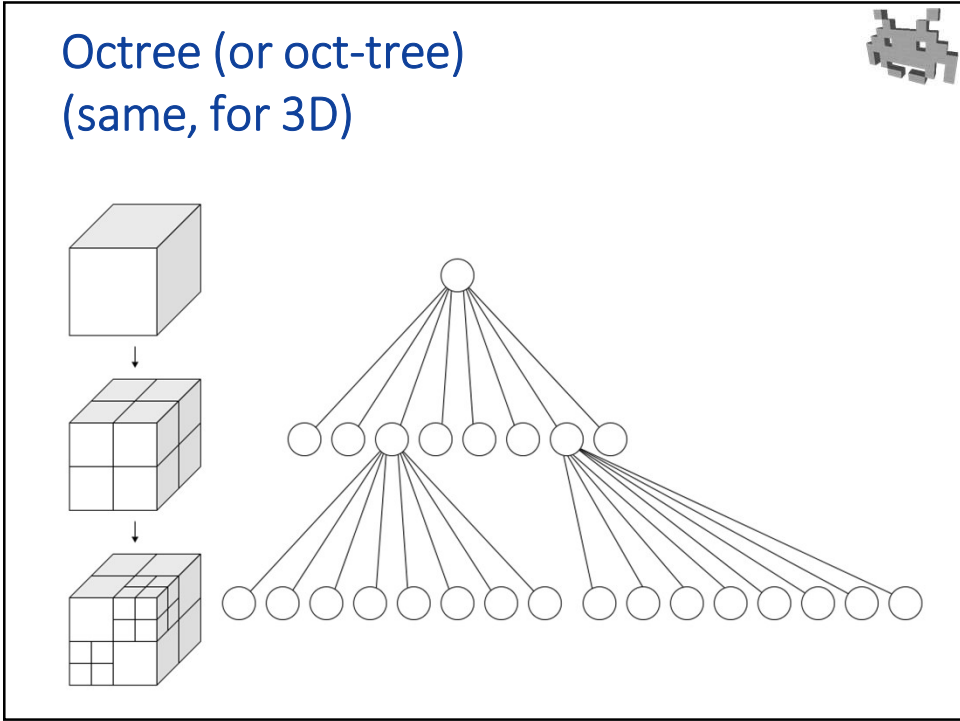
### kD-trees

- 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

91



92



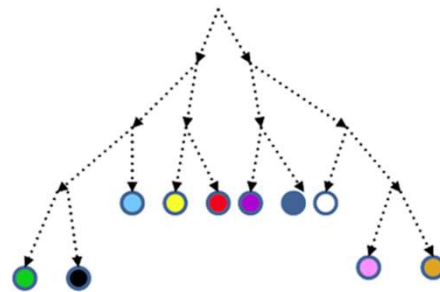
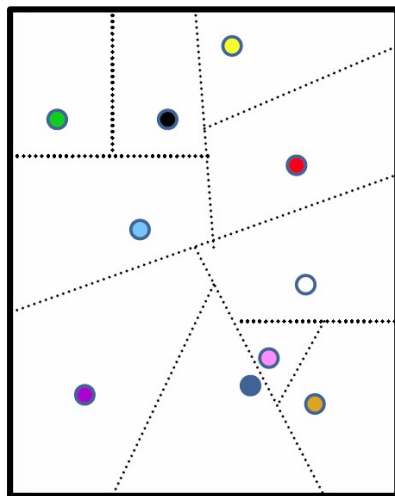
## Quad-trees (in 2D) Octrees (in 3D)



- Similar to kD-trees, but:
  - tree: branching factor: 4 (in 2D) or 8 (in 3D)
  - each node: splits halfway across all dimensions at once  
X and Y in 2D  
X and Y and Z in 3D
- Construction (just as kD-trees):
  - continue splitting until end nodes have few enough objects  
(or limit depth reached)

94

## BSP-tree Binary Spatial Partition tree



95

## BSP-tree, this time as a spatial indexing 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 (so far, same as as *kD*-trees)
- 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
  - Pro:  
they can be optimized for optimal queries: better query time!
  - Con:  
must be optimized during construction: worse construction time!

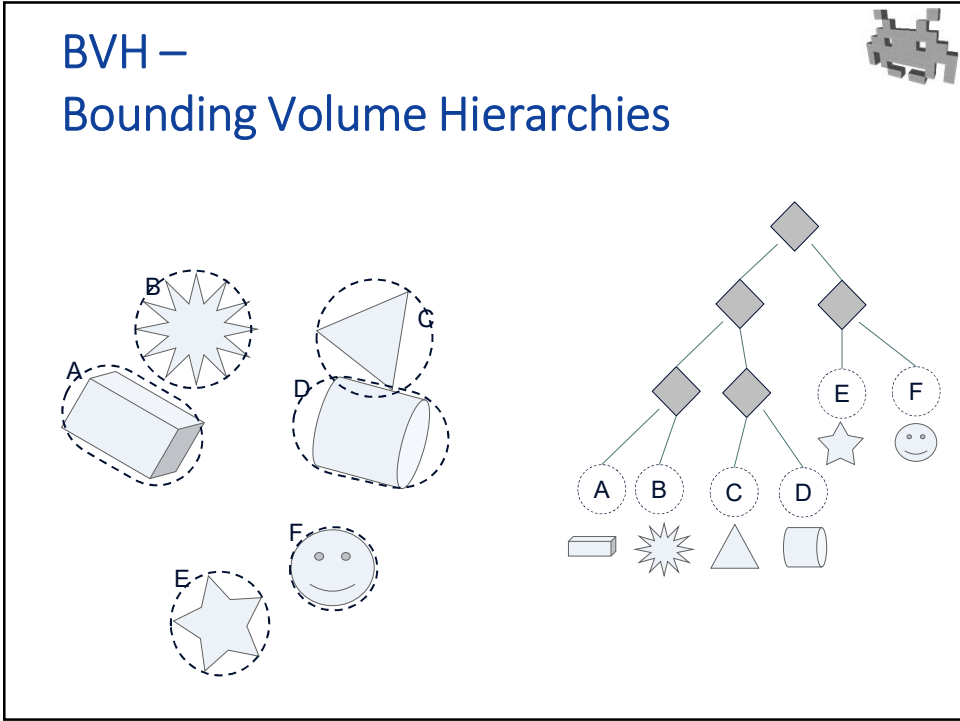
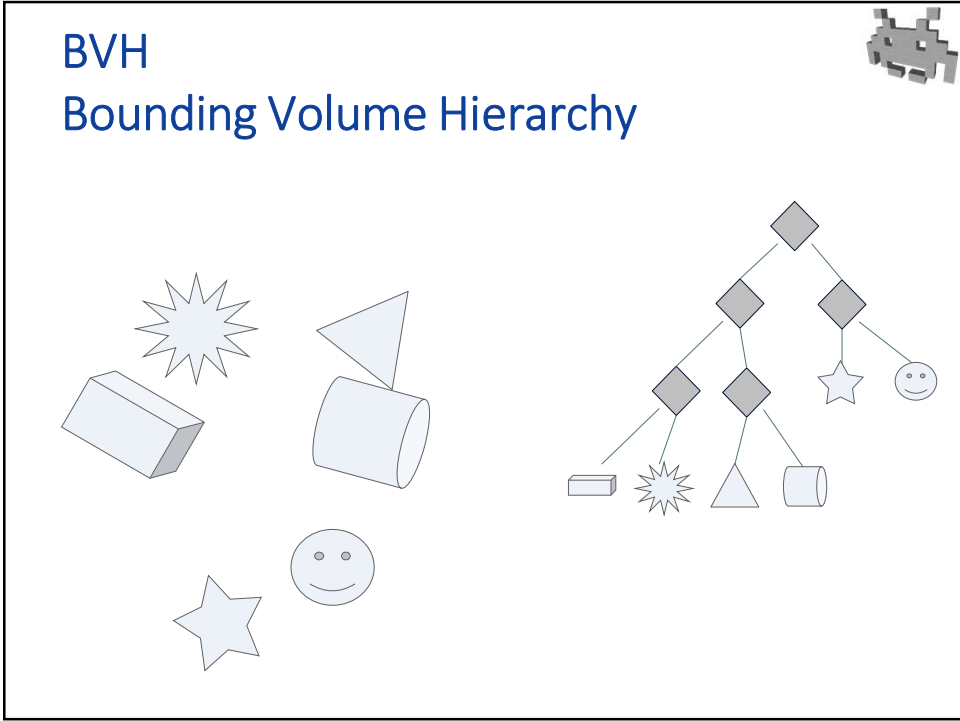
96

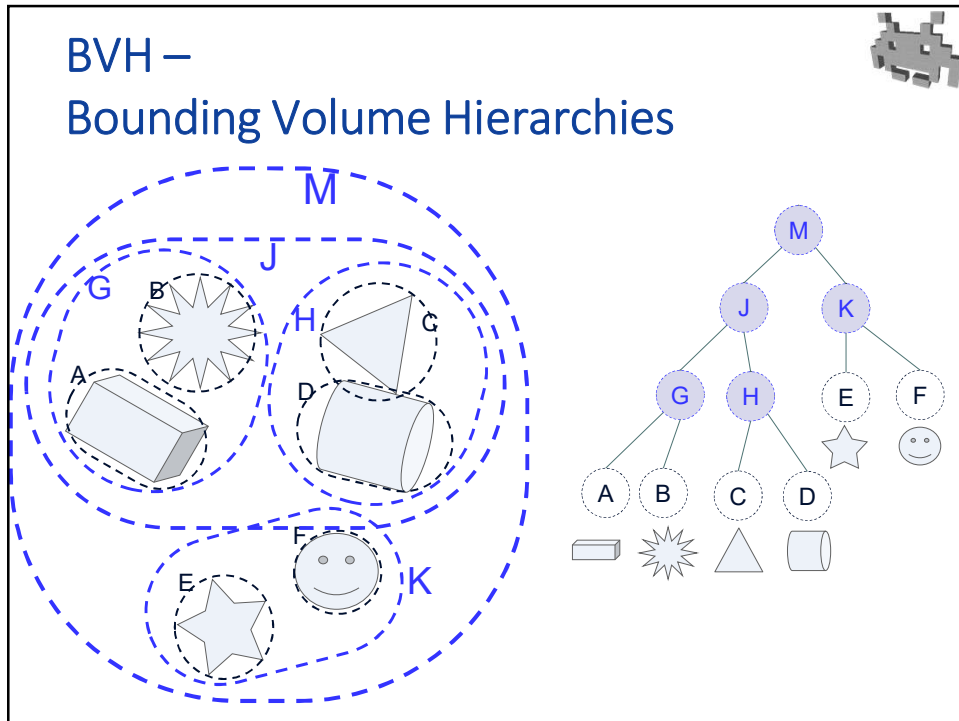
## The «broad-phase» of coll. detection (avoiding quadratic explosion of # of tests)



- Classes of solutions:
  - 1) Sorting-based algorithms
  - 2) *spatial indexing* structures
  - 3) BVH – Bounding Volume Hierarchies

97





### BVH Bounding Volume Hierarchy

- We can use the hierarchy already defined by the scene graph
  - instead of a spatially 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
- using it:
  - top-down: visit (how?)
  - *note*: it's **not** a single root to leaf path
    - may need to follow *multiple* children of a node (in a BSP-tree: only one)

101



## Broad phase strategies: Recap



- **Regular Grid**
  - ☺ parallelizable construction
  - ☺ constant time access (best!)
  - ☹ huge in RAM space – OR hashing (extra cost)
  - ☹ *requisite*: volume of playfield must be known in advance, cannot be too large
- **kD-tree, Oct-tree, Quad-tree** : as above but...
  - ☺ more compact in RAM / can deal with larger playfields
  - ☹ more complex, not as parallelizable construction
- **BSP-tree**
  - ☺ optimized splits! → best performance when accessed
  - ☹ optimized splits! → more complex construction / update
  - **good candidate for broad-phase of static parts of the scene?**
  - (also, the perfect structure to model (general) *Polyhedral Geometric Proxies*)
- **BVH**
  - ☺ can exploit existing scene hierarchy (scene graph)
  - ☹ non necessarily very efficient to access (excessive tree depth)
  - **good candidate for intermediate phase of dynamic parts of the scene?**
- **SAP**
  - ☹/☺  $N \log N$  to construct, but faster to update
  - Requisite: objects cannot be too large (e.g. 3D model of a room / a cave / etc)
  - **good candidate for broad phase of dynamic parts?**

102

## Collision Detection: to learn more...



Christer Ericson (ACTIVISION):  
**Real-Time Collision Detection**  
The Morgan Kaufmann Series in  
Interactive 3-D Technology  
HAR/CDR Edition  
Elsevier

103

## Physics Engine: an implementation issue for GPU



- 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, sorting...
  - hugely variable workflow
    - e.g.: quick on no-collision, more work to do when the rare collisions occur
  - difficult to parallelize: **CPU**
  - but the outcome affects the other two tasks (e.g., creates constraints)
    - ==> **CPU-GPU** communication, and ==> **GPU** structures updates (problematic on many architectures)

104

## End of Game Physics. 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/realtimetypephysics/>
- David H. Eberly:  
Game Physics (2nd Edition)  
MK Press
- Ian Millington:  
Game Physics Engine Development (2nd Edition)  
MK Press

105