

## 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: **3D Audio** for 3D Games ●
- lec. 11: **Networking** for 3D Games ●
- lec. 12: **Artificial Intelligence** for 3D Games ●
- lec. 13: **Rendering Techniques** for 3D Games ●

34

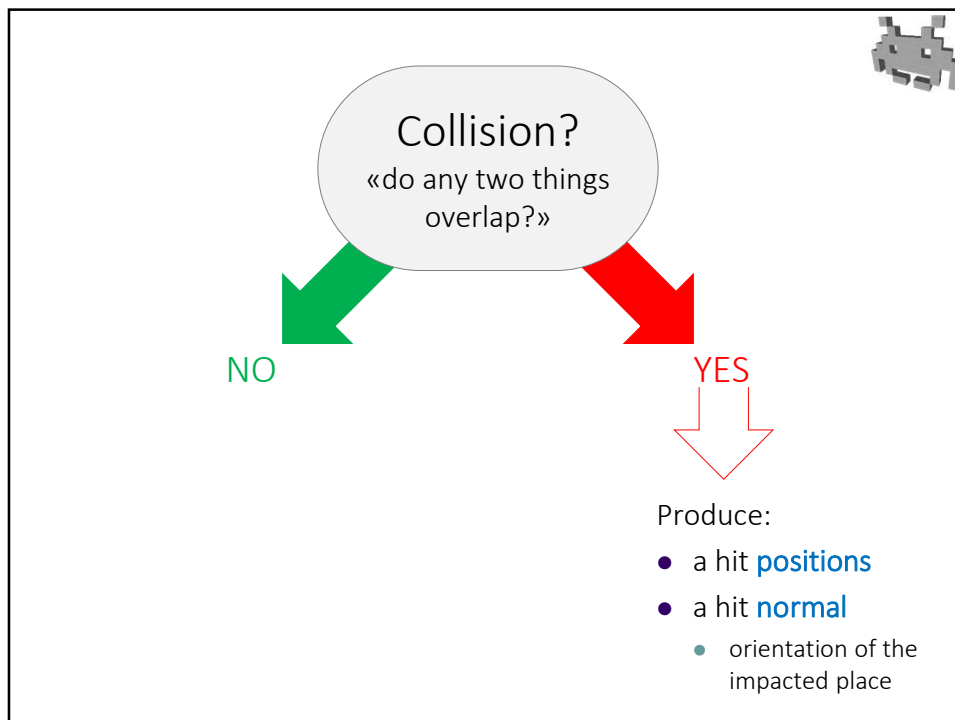
## Collision Handling: two tasks



- **Collision detection**
  - find out when they occur
  - if so, produce **collision data** for the response
- **Collision response**
  - compute their effects



35



36

## From detection to response

The collision detection needs to tell us, for any pair of objects:

- Collision? **Yes / No**
  - «do these two things overlap?»

And, when it's a **Yes**...

- a hit **positions**
- **normal** of one collision plane
  - ~orientation of the impacted part
  - needed to: resolve the impact (except for purely inelastic)
  - needed to: apply frictions

**«collision data»**  
output of detection,  
input of response

37

## Collision detection: a preliminary observation



- The usual concern: *efficiency*
- Key observation:
  - almost 100% of the object pairs, almost 100% of the times, **do NOT collide.**
  - for efficiency, the «no-collision» case needs to be optimized
  - «early reject» of the collision test

38

## Example: this billiard shot



- A very “collidey” situation, right?
- Let’s do the math
  - Balls: 16 (=15+1)
  - Ball pairs: 136 (=16 x 17 / 2 )
  - Shot duration: ~10 seconds = ~600 physics frames
  - Assume ~2 collisions for each ball (a lot!) during the shot: ~16 collision events (each involving two balls)
  - Total: 16 collisions over 136 x 600 tests.
  - **only < 0.02% of the potential collisions will collide!**

40

## Collision detection



- Efficiency issues:

a) how to test between object pairs:

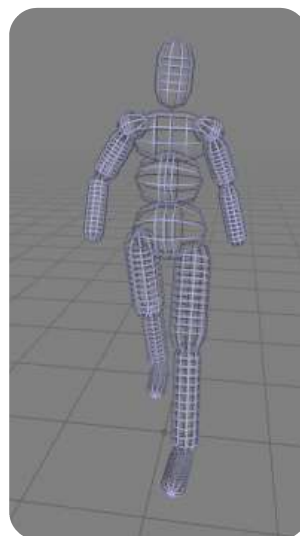
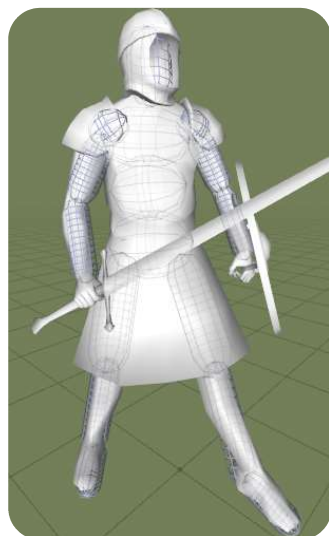
- In an efficient way

b) how to avoid quadratic explosions of needed tests

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

41

## Geometric proxies



42

## Geometric proxies



A simplified representation of the shape (the geometry) of the object, to be used in its place

- Note: it can be a *much* cruder approx. than the 3D model used for rendering

Two possible uses:

- as **Bounding Volume**
  - **upper bound** of the object spatial extension; object is *all inside* the proxy
  - for *conservative* tests
- as **Collider** (or **hit-box**, or **collision proxy**)
  - **approximation** of the object spatial extension
  - for *approximate* tests



("hit-box" is a misnomer: it's not necessarily a "box")

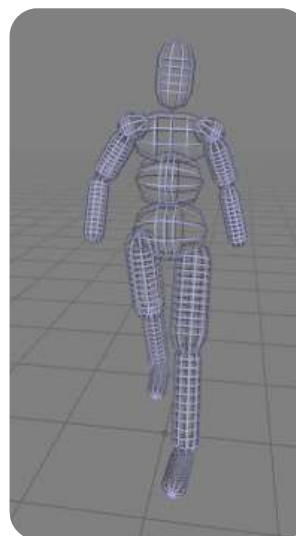
43

## Geometric proxies: not only for collision detection, but also:



- **physic engine**
  - extract data for collision response
  - extract *barycenter* position & *moment-of-inertia* matrix of rigid bodies assuming uniform density (*Ita.: peso specifico*)
- **rendering optimizations**
  - "view frustum culling" (*bounding volumes*)
  - "occlusion culling" (*bounding volumes*)
- **AI**
  - visibility tests
  - in general, simulation of NPC senses
- **GUI**
  - picking (one of the ways to do that)
- **3D sounds**
  - sound absorption in 3D sound propagation

Basically, for any other task except rendering: internally, objects *are* their proxies.



44

## Semantic of a geometric proxy

Another proxy, a point, a ray...

$\text{intersection}(\text{proxy\_A}, \langle \text{something} \rangle) \neq \emptyset ?$

- if **proxy\_A** serves as **Bounding Volume** :
  - if NO: no collision
  - if YES: we don't know yet

An «early reject» optimization
- if **proxy\_A** serves as **Collider** :
  - if NO: no collision
  - if YES: **collision detected** !
    - Must compute **collision data** from proxy\_A

An approximation of the collision detection

Despite this semantic difference, the data types used to represent proxies are the same.



45

## Geometric proxies: shapes

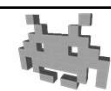
- Spheres
- Capsules
- Half-spaces
- Axis Aligned (Bounding) Boxes
  - aka AABB
- Generic Boxes
- Discrete Oriented Polytopes
  - aka DOP
- Ellipsoids
  - axis aligned or not
- Cylinders
- Convex polyhedrons
- Non-convex polyhedrons
  - Meshes
- ...

46

🙄 **choosing Geometric Proxies:**  
things to consider

by algorithms   assets!  
by artists

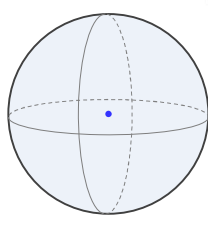
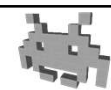
- Workload needed to **compute** / **create** them
- RAM space needed to **store** them
- Behavior under **transformations**
  - the ones we plan to use, e.g., roto-translations
- How good is the geometric **approximation**
  - for the objects we will use in the game
  - for bounding volumes ==> how *small* / *tight* is it?
  - for colliders ==> how *accurate* is the approximation?
- Workload for an **intersection test**
  - with other proxies, points, rays
  - also, how { easy to compute | good } is the collision data?



47

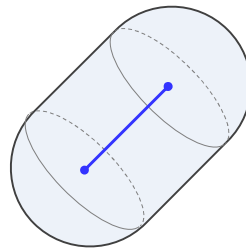
**Geometric proxies:**  
**A sphere**

- 😊 easy to compute as a boundary
  - only the approximately optimal one
- 😊 tiny to store
  - center (a point) + radius (a scalar) – or, a vec4  $(c_x, c_y, c_z, r)$
- 😊 collision test: trivial (against spheres or other things)
  - how? exercise – including collision data computation
- 😊 can easily undergo translation/rotation/scaling
  - how? exercise – note: scaling must be uniform
- 😞 approximation quality:
  - it depends on the object (as usual)
  - often, quite poor:
    - e.g.: a head? A character? A house? A sword?

49

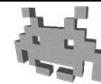
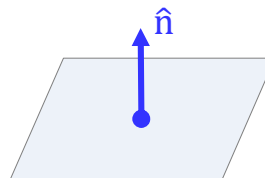
## Geometry proxies: «Capsule»



- Generalizes the sphere:
  - Sphere  $\triangleq$  the set of points having dist. from a **point**  $\leq$  radius
  - Capsule  $\triangleq$  the set of points having dist. from a **segment**  $\leq$  radius
    - i.e. 1 cylinder ended with 2 half-spheres (all 3 with same radius)
- Stored as:
  - a segment (its two end-points)
  - a radius (a scalar)
- Exercise :
  - Q: how does it «score» w.r.t. the above measures?
  - (A: quite well  $\rightarrow$  a very popular proxy in games!)

51

## Geometry proxies: a half-space



- Trivial, but useful!
  - e.g. for a flat terrain,
  - or a wall
  - or an invisible “force field” to limit the game level (hated by players :-)
- Storage:
  - a point on the plane + its normal
  - better: a normal + a distance from the origin
  - which is a vec4  $(n_x, n_y, n_z, k)$
- how to test , transform, etc:
  - easy and efficient algorithms (check me)

53



### Mini-exercise: Plane VS Point test

- Input: a point  $\mathbf{q}$  and a plane given by:
  - its normal:  $\hat{\mathbf{n}}$
  - a point on it at random:  $\mathbf{p}$
- Q: on which side of the plane is  $\mathbf{q}$  ?
- A: it's the sign of

$$\hat{\mathbf{n}} \cdot (\mathbf{q} - \mathbf{p}) =$$

$$\hat{\mathbf{n}} \cdot \mathbf{q} - \hat{\mathbf{n}} \cdot \mathbf{p} =$$

$$\hat{\mathbf{n}} \cdot \mathbf{q} + k =$$

$k = -\hat{\mathbf{n}} \cdot \mathbf{p}$   
 (minus distance of plane from origin)

$$(\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z, k) \cdot (\mathbf{q}_x, \mathbf{q}_y, \mathbf{q}_z, 1)$$

a 4D vector representing the plane

54

### Which geometric proxy types to support in a game (-engine)?

- an implementation choice of the **Physics Engine**
- # of intersection-test algorithms to be *implemented* : **quadratic with** # of supported types

VS	Type A	Type B	Type C	a Point	a Ray
Type A	algorithm 1	algorithm 2	algorithm 3	algorithm 4	algorithm 5
Type B		algorithm 6	algorithm 7	algorithm 8	algorithm 9
Type C			algorithm 10	algorithm 11	algorithm 12

useful, e.g. for visibility

55

## Example: algorithm to testing capsule VS capsule



Input:

- Capsule 0: point  $\mathbf{a}_0$   $\mathbf{b}_0$  radius  $r_0$
- Capsule 1: point  $\mathbf{a}_1$   $\mathbf{b}_1$  radius  $r_1$

Output:

- Do they intersect?
- If so, intersection point  $\mathbf{p}$  and normal  $\hat{\mathbf{n}}$ ?

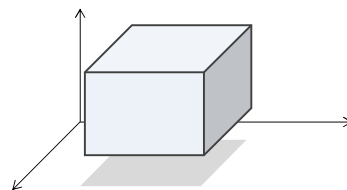
Solution (trace):

1. Find  $\mathbf{c}_0$  and  $\mathbf{c}_1$ , the two points on the two segments closest to each other (see exercises on points and vectors)
2. Test:  $\|\mathbf{c}_0 - \mathbf{c}_1\| < r_0 + r_1$  ?
3. Is so, collision detected with  $\hat{\mathbf{n}} = \frac{\mathbf{c}_0 - \mathbf{c}_1}{\|\mathbf{c}_0 - \mathbf{c}_1\|}$

56

## Geometry proxies: «AABB»

As the name implies, typically used as BOUNDING volume, not a collider



### Axis Aligned Bounding Box

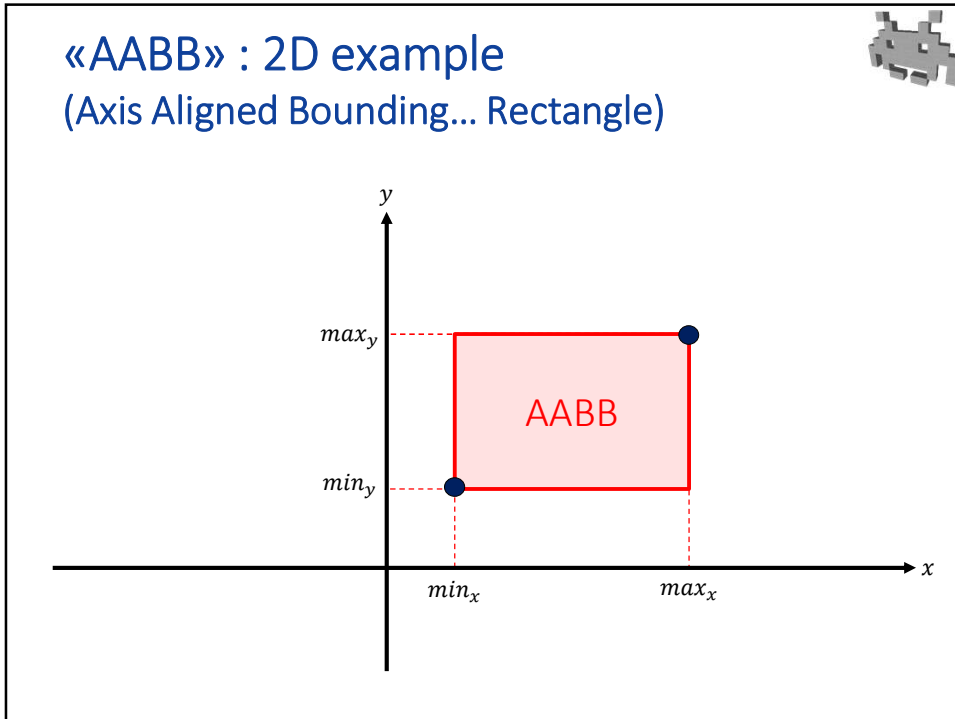
- Consists of three interval  

$$[min_x, max_x] \times [min_y, max_y] \times [min_z, max_z]$$
- Concise to store
  - Two 3D points:  $(min_x, min_y, min_z)$  &  $(max_x, max_y, max_z)$
- Easy to find the minimal AABB encapsulating a given set of points
- Easy to test for collision VS a point, or another AABB
  - Exercise: how?
- Under transforms:
  - ⊗ ⊗ ⊗ if rotated, an AABB expands
  - (but can be easily scaled / translated)

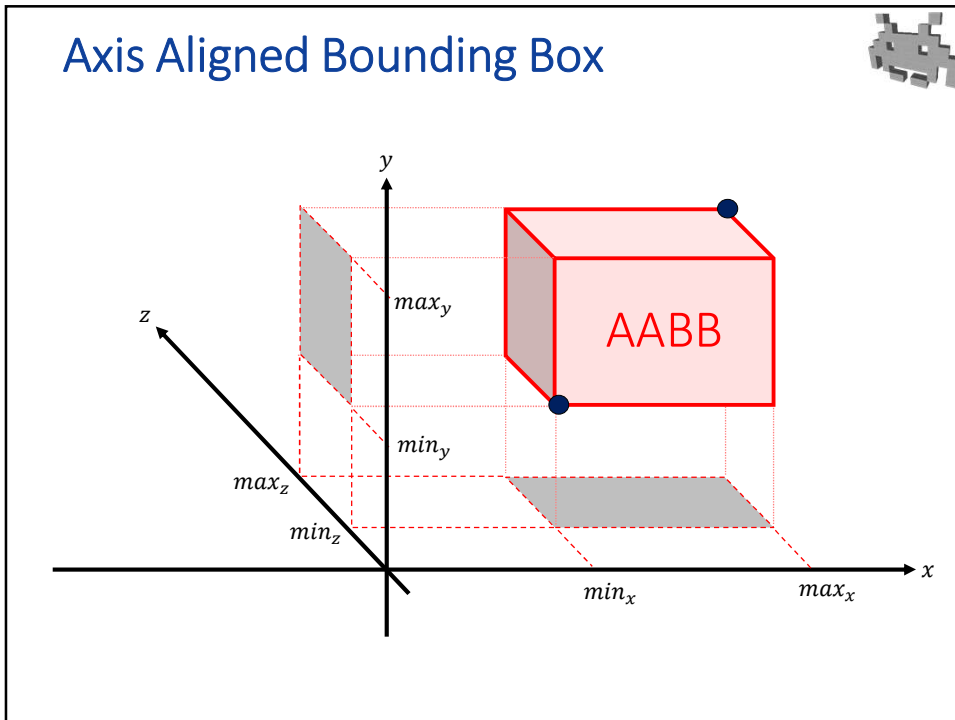
Cartesian product



58



59

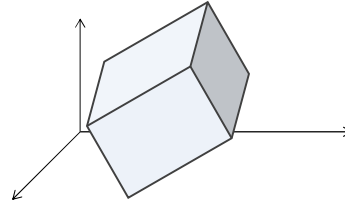


60

## Geometry proxies: Oriented Bounding Box (OBB)



- A “parallelepiped”
  - generalized version of AABB: it’s not axis-aligned
  - storage:
    - a rotation (e.g. a quaternion) +
    - an AABB
  - Can be freely transformed
    - note: but only if scaling is uniform
  - Tests: still relatively easy (exercise: how to test points?)

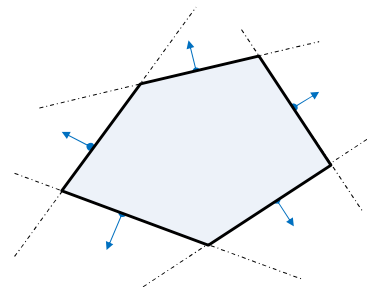


62

## Geometry proxies (in 2D): a Convex Polygon



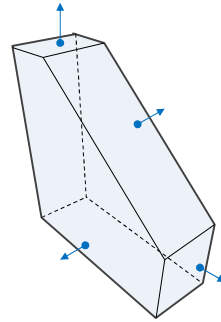
- Intersection of half-planes
  - each delimited by a line
- Stored as:
  - a collection of (oriented) lines
- Test:
  - a point is inside the proxy iff it is in each half-plane
- Flexible (good approximations)... and still moderate complexity



64

## Geometry proxies (in 3D): a Convex Polyhedron

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



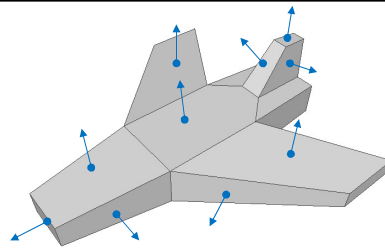
66

## Geometry proxies a (general) Polyhedron

↖ potentially **concave**

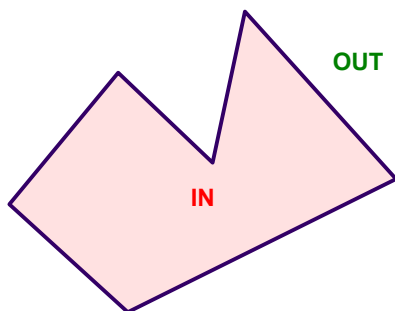
↖ not worth it for a **Bounding Volume** !

- 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)
- Creation (treat them as meshes):
  - sometimes, with automatic simplification
  - often, hand-designed by artists (low poly modelling)
- Wait, is this the same as a 3D mesh used for rendering?
  - Many differences (compare with mesh later, lecture 6)



68

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



69

## 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

71

## 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** of course (no uv-mapping, no color, etc)
  - made of **generic polygons**, not just **tris** (as long as they are *flat*)
  - always **closed, water-tight** (inside != outside)
  - very different internal representation:  
a set of bounding planes (in a BSP tree probably)  
in addition to collection of vertices (3D points)

72

## 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

73

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



see lecture on 3D models later

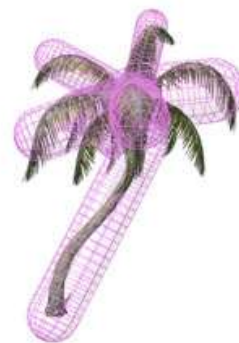
- Proxy meshes are
  - much **lower res** (e.g.  $< 10^2$  faces )
  - no **attributes** (no uv-mapping, no color, etc)
  - based **generic polygons**, not just **tris** (as long as they are *flat*)
  - **closed, water-tight** (inside != outside)
  - different internal representation:
    - if **convex** : a set of bounding planes
    - if **convex** : a BSP tree

74

## Geometry Proxies: Composite

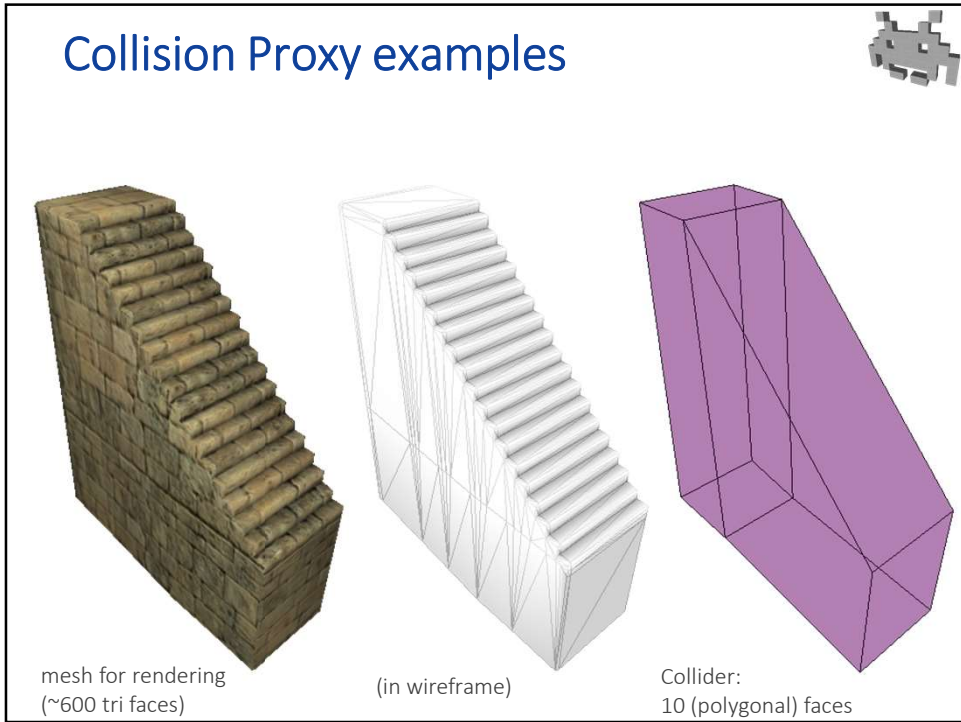


- A proxy can be a union of sub-proxies
  - inside the proxy *iff* inside of *any* sub proxy
- Very expressive
  - better approximation for many objects, even with few proxies
  - note: union of **convex** proxies can be **concave** !
- Efficient to test / store
  - Compared to alternatives
- Difficult to construct automatically

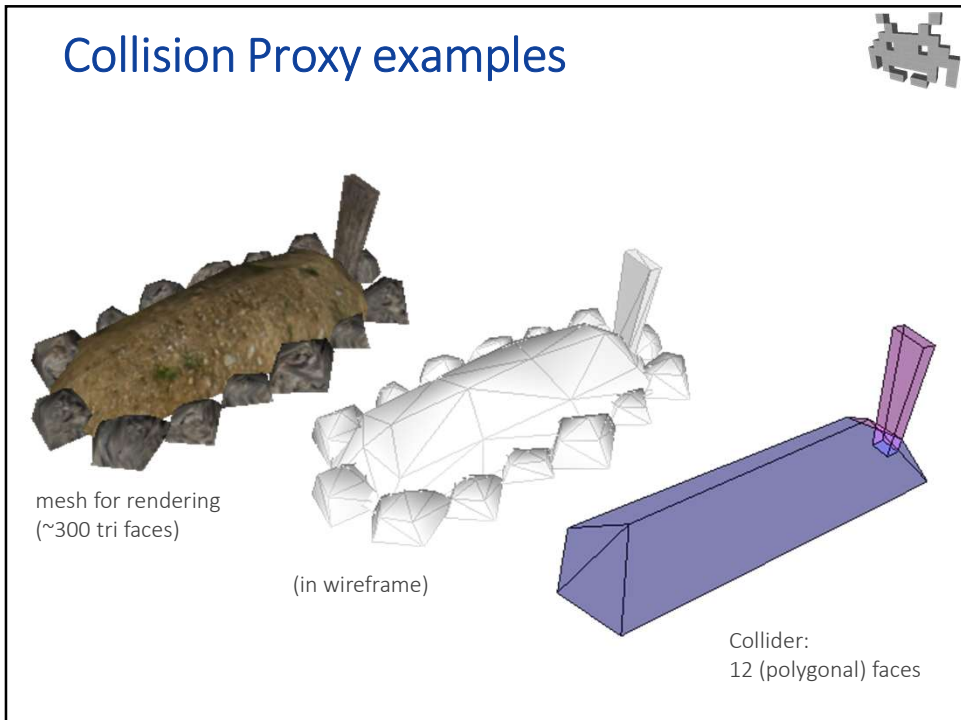


76

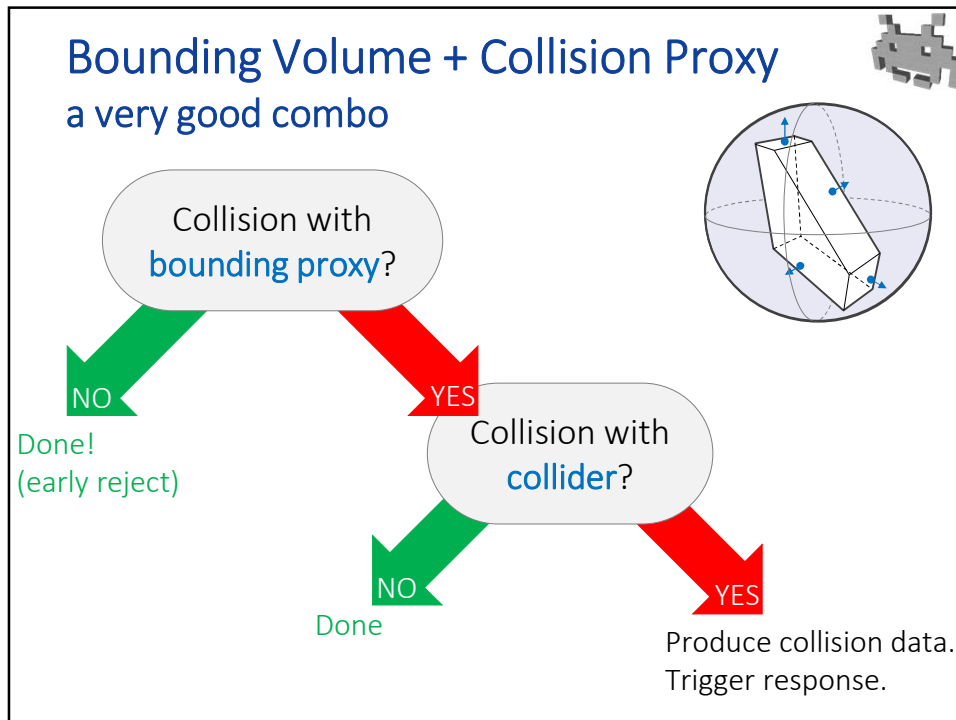




77



78



79

### Bounding Volume + Collision Proxy a very good combo

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

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

a simpler **Bounding Volume** with, inside, a more complex **Collision Object** approximating the object

80

## How to construct a geometry proxy to be used as a collider?



“Given an object representation  $M$ , build a good **collision proxy** for it”

- $M$  = 3D model of e.g. a dragon, a castle, a character...



difficult task to automatize (by algorithms)

- especially if we want to pick simpler (more efficient) proxies (such as compound of a few spheres, capsules, boxes)
- especially if we need good approximations



often done manually (by digital artists)

**Geometry proxies for colliders are assets !**

81

## How to construct a geometry proxy to be used as a bounding volume?



“Given an object representation  $M$ , build a thight **bounding volume** for it”

- a  $M$  = 3D model of e.g. a dragon, a castle, a character...



This task can be (and is) automatized

- note: finding the optimal (smallest possible) bounding volume: computationally difficult (**can be NP complete**)
- find a “good enough” bounding volume: a lot easier (**heuristics**)
- can be done on the fly during game execution
- For example, think about algorithms to find a bounding volumes of type...
  - AABB (trivial!)
  - Sphere – i.e. a “bounding sphere” (less trivial)
  - Capsule, OBB (more difficult!)

82

## Digression: collision detection in traditional 2D sprite-based games



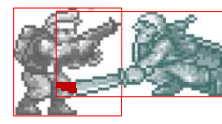
- An easier problem
- We can leverage **collision detection for 2D sprites** ← in screen space
  - *it's accurate*: «**pixel perfect**»
  - *it's efficient*: **HW supported**  
(hard-wired support, as part of sprite rendering)
  - little need for **proxy** approximations for colliders  
(same structure – the sprite – both for collision and for rendering)
  - easy bounding “volume”: axis-aligned bounding-rectangle of the sprite



NO COLLISION



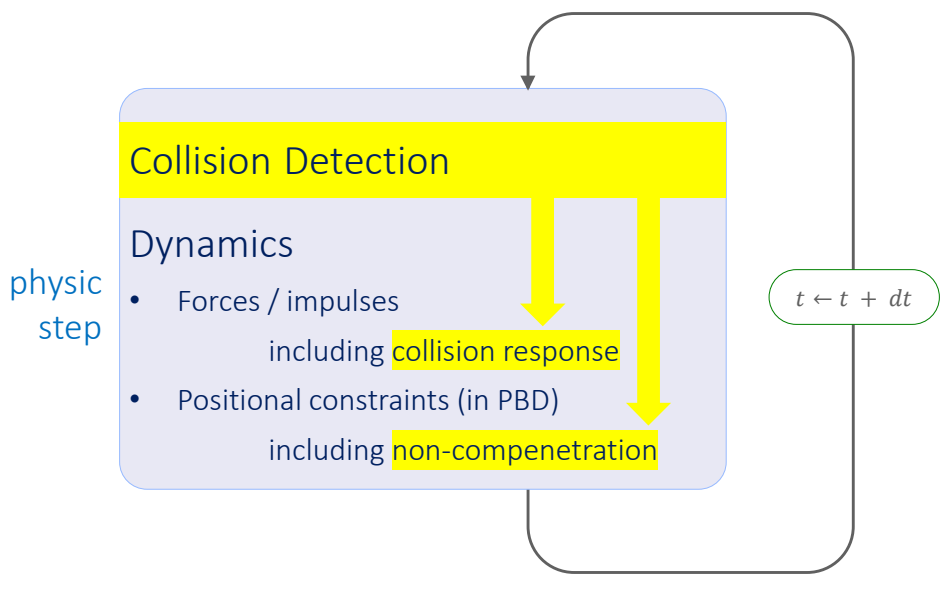
NO COLLISION



COLLISION

83

## Collision detection: When?



84

## Collision detection: strategies

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

85

## 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

86

## Collision detection: Dynamic

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)

- Much more accurate detection
- Bonus:
  - no need to «teleport the object in the safe position».
  - it never left a safe position!
  - It can be 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, should only be used when required

87

## 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 ?

88

## 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!
  - Even excluding static-static pairs: still way too many (quadratic)
- 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** to discard object pairs that actually collided,
  - **ok** to test objects that *didn't* actually collide
- Let's see strategies to do so

89

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

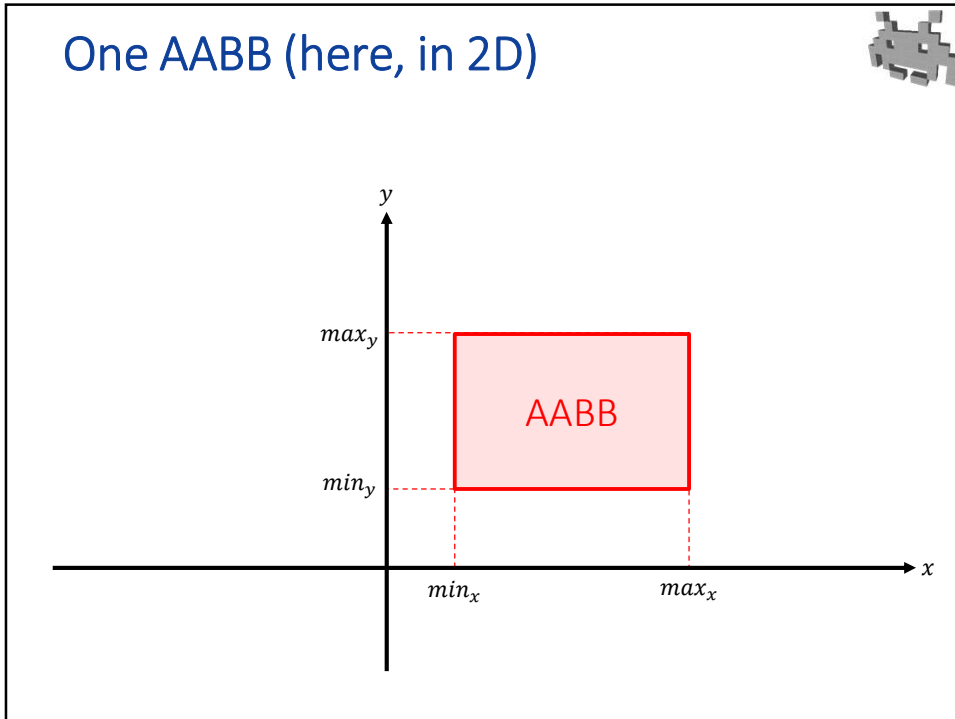
- Classes of solutions:

1) Sorting-based algorithms

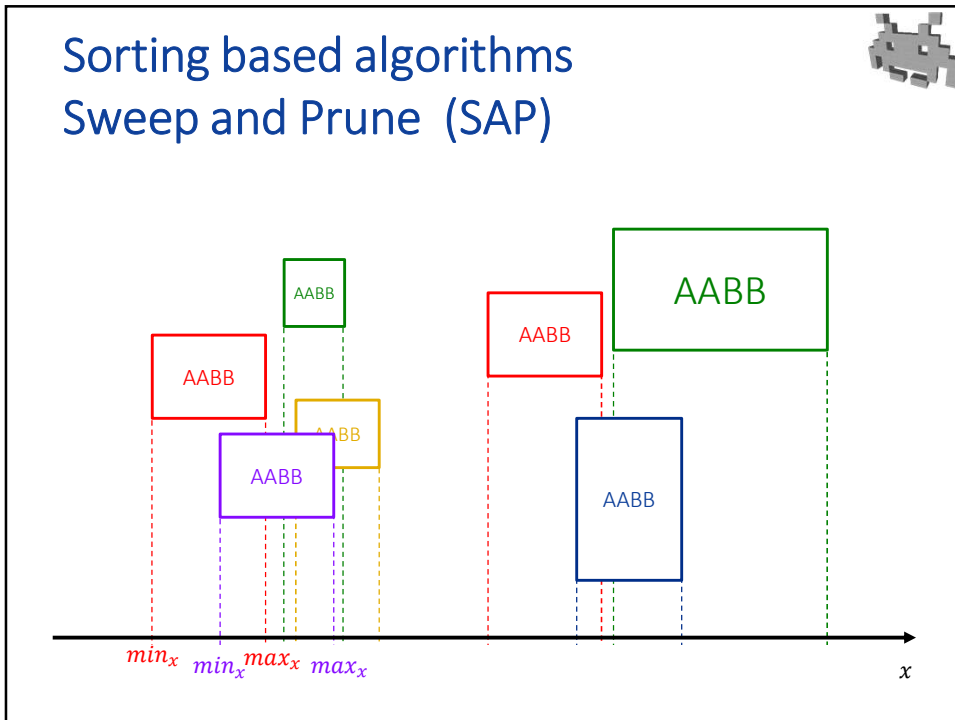
2) **spatial indexing** structures

3) BVH – Bounding Volume Hierarchies

90



91



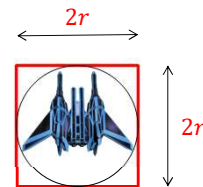
92



## Sweep And Prune (SAP) strategy (aka "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

only  
 $O(n \log n)$

### 3. Sweep the sorted intersections, from smaller to larger

- Quickly detect intersecting intervals in  $x$  (how?)

Even  
faster!  
 $O(n)$

### 4. Prune: among AABB intervals, ignore the ones that don't *also* intersect in both $y$ and $z$

- Test the other pairs for collision

93

## 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

94

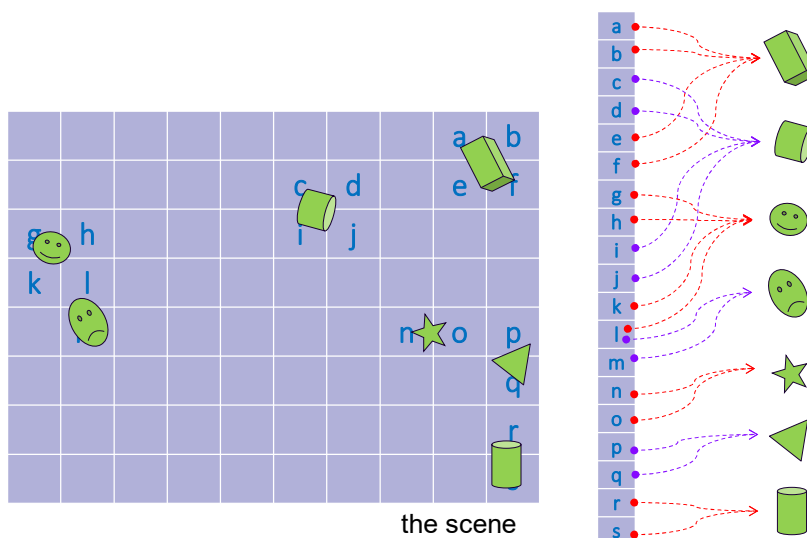
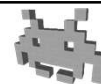
## Spatial indexing structures



- 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**

95

## Regular Grid (or: lattice)



96

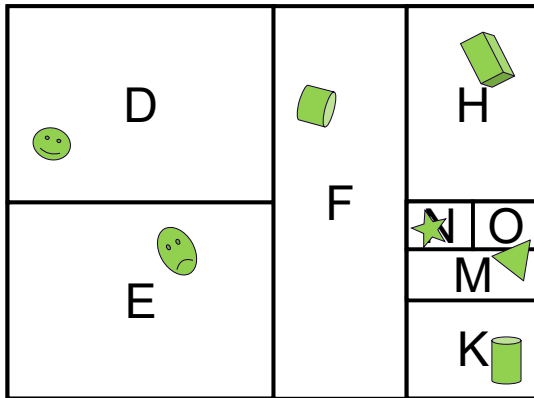
## Regular Grid (or: lattice)



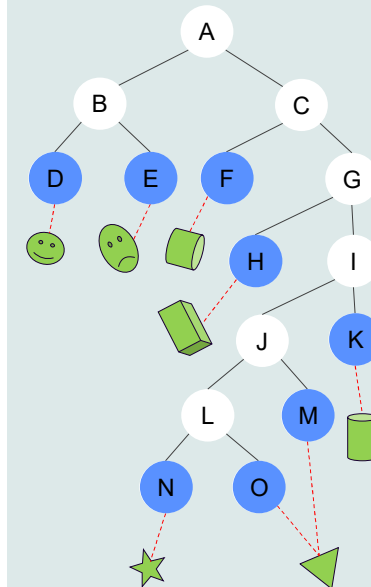
- Array 3D of cells (all the same size)
  - each cell = a list of pointers to collision objects
- Indexing function:
  - Point3D  $\rightarrow$  cell index, (constant time!)
- Construction: (“scatter” approach)
  - for each object B, find all the cells it touches, add a pointer to B to them
- Queries: (“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

97

## kD-trees



the scene



98

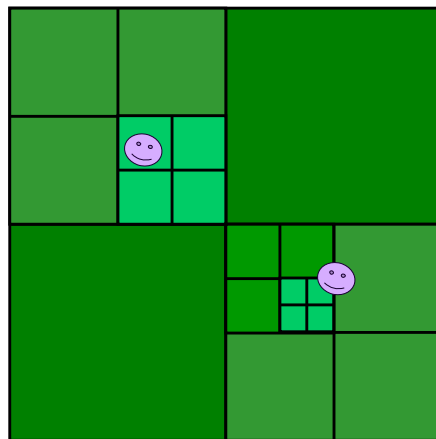
## kD-trees



- 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

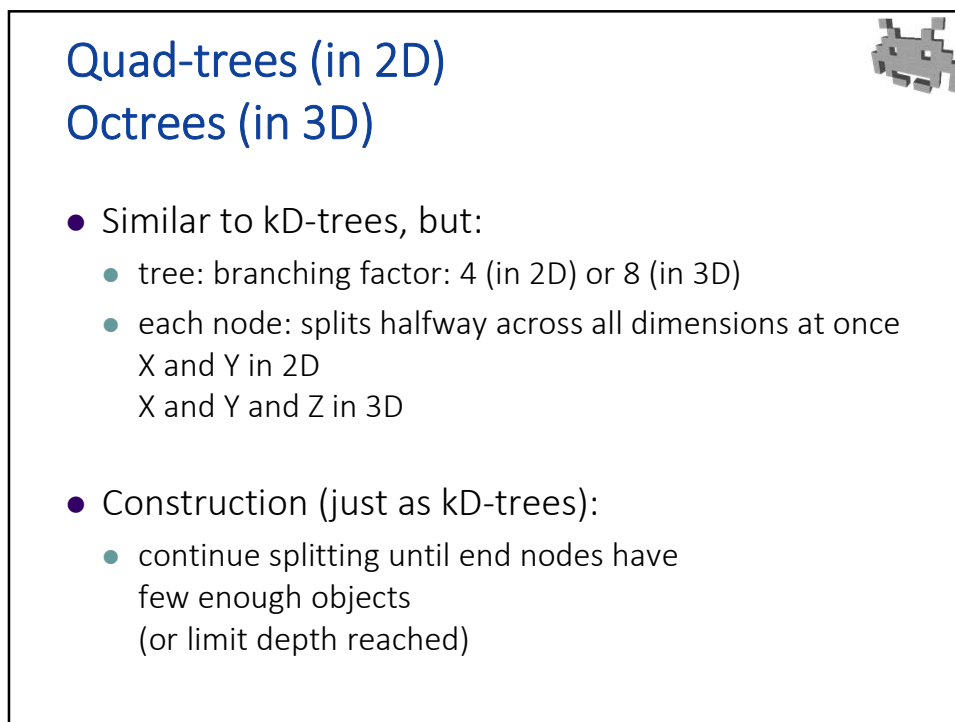
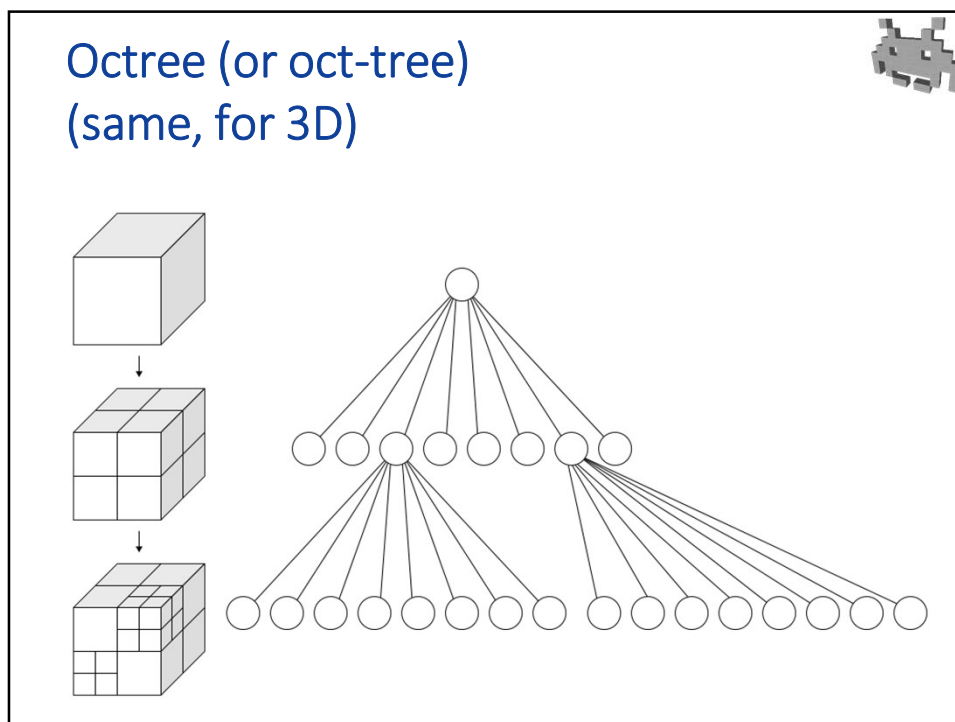
99

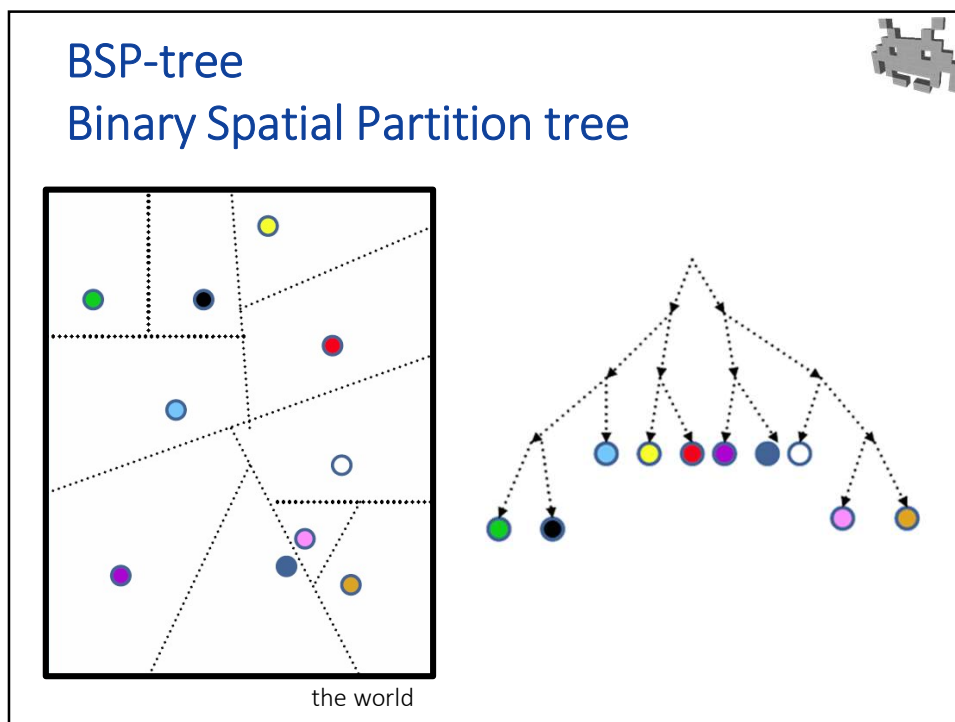
## Quad-Tree (in 2D)



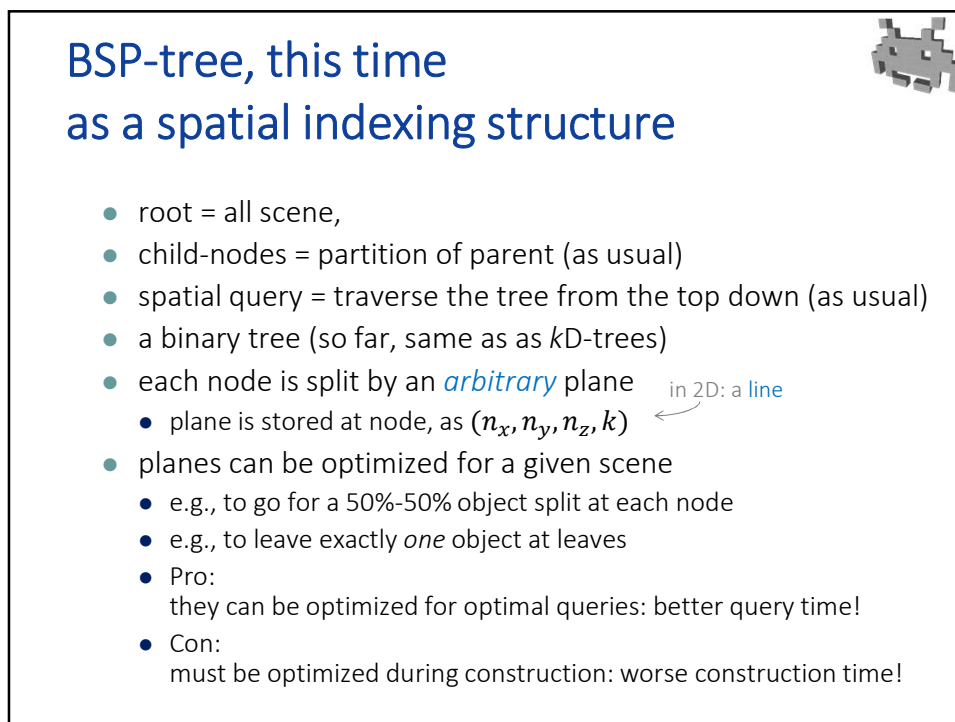
the (2D) world

100





103



104

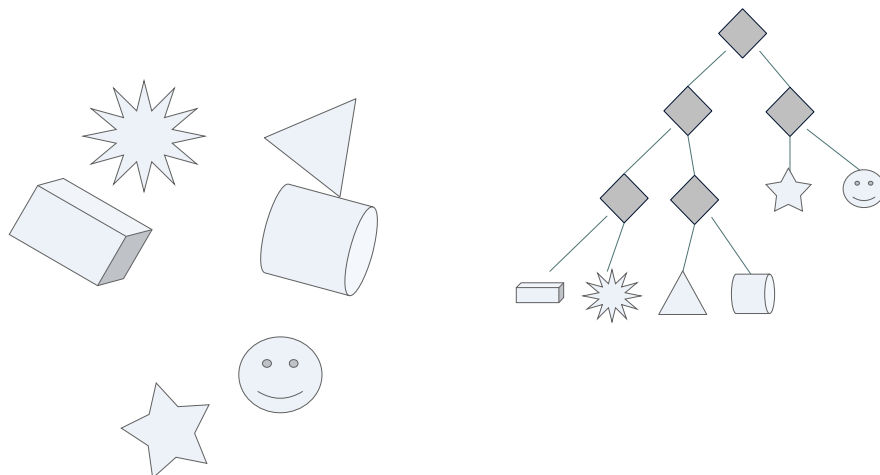
## The «broad-phase» of coll. detection (avoiding quadratic explosion of # of tests)



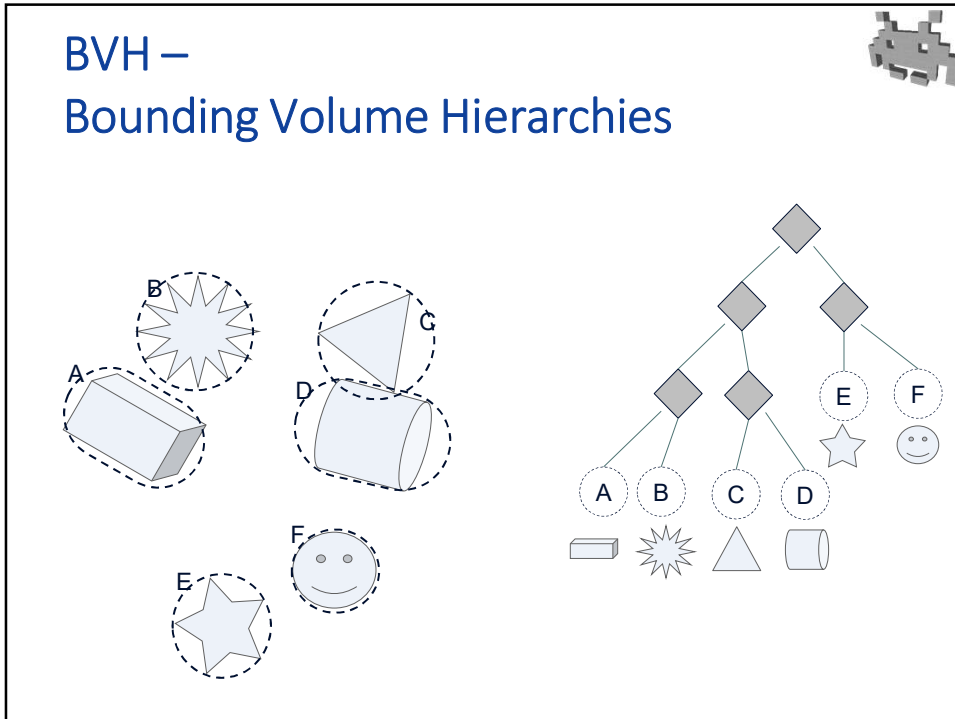
- Classes of solutions:
  - 1) Sorting-based algorithms
  - 2) *spatial indexing* structures
  - 3) BVH – Bounding Volume Hierarchies

105

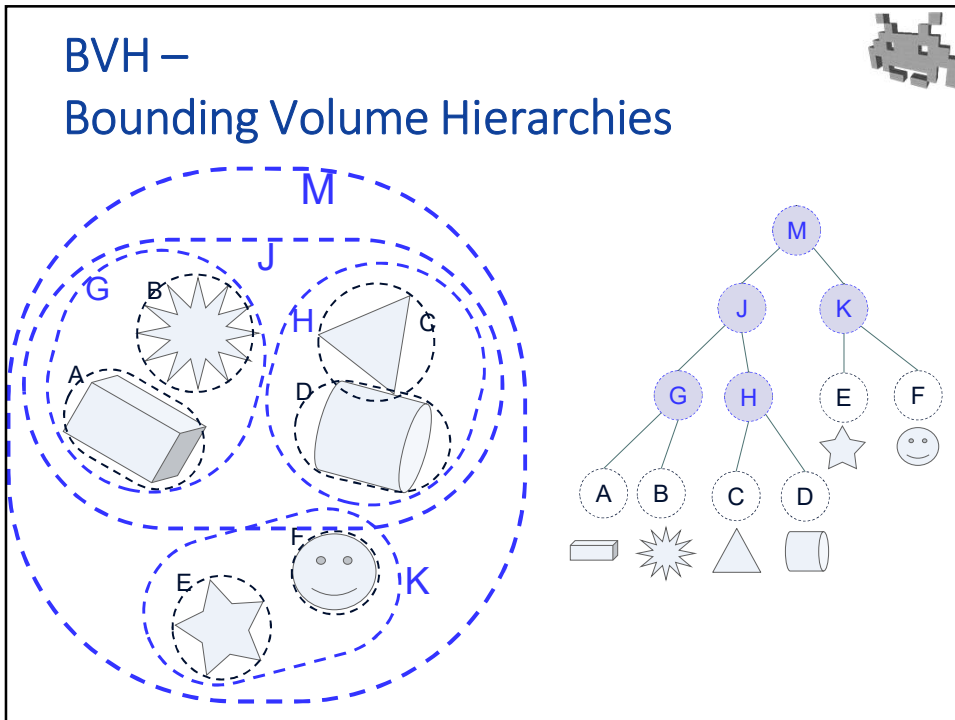
## BVH Bounding Volume Hierarchy



106



107



108



## 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)

109

## 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

111

## 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 know 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)

112

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

113