




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 ●

34

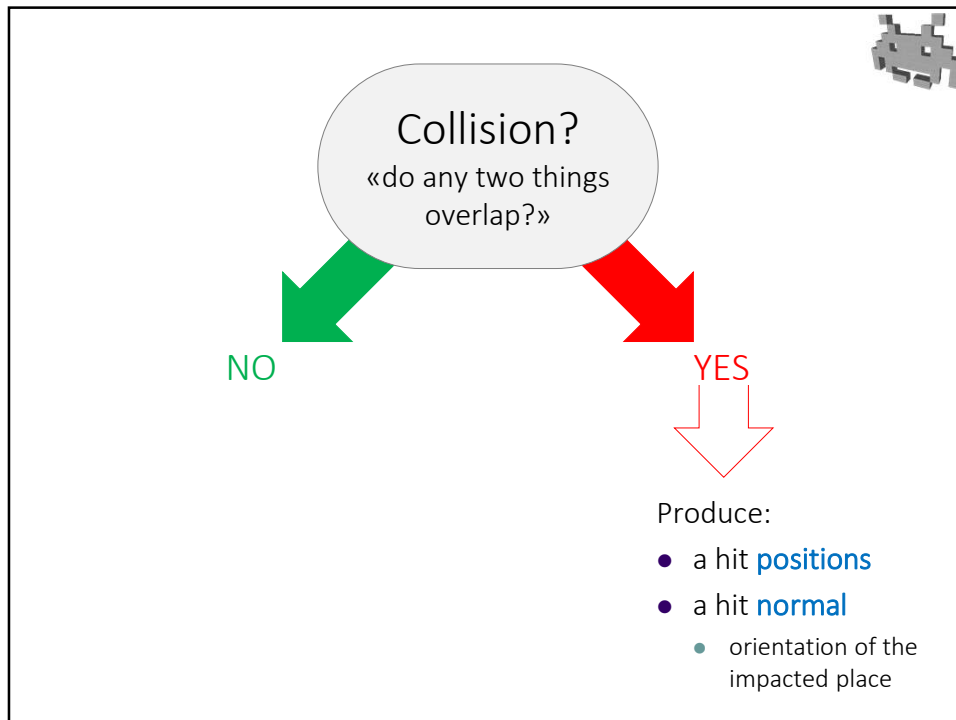
Collision Handling



- **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:

- Collision? **Yes** / **No**
 - «do any 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)

«collision data»
output of detection,
input of response

37

Collision detection



- The usual king of the concerns: *efficiency*
- 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 test

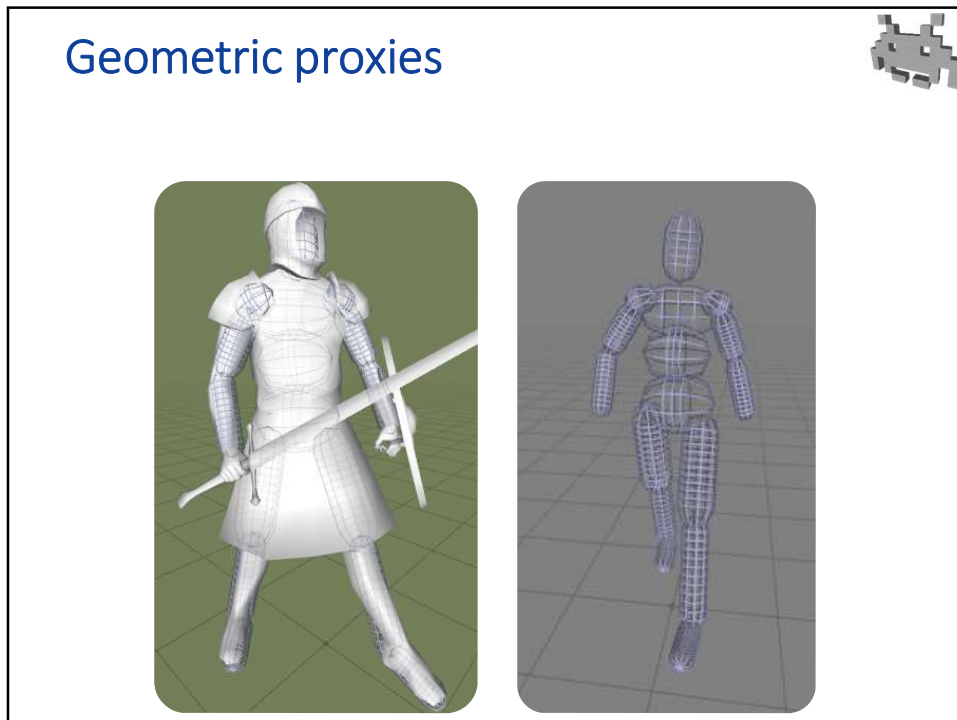
38

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 ?

39



40

Geometric proxies

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

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

Two 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

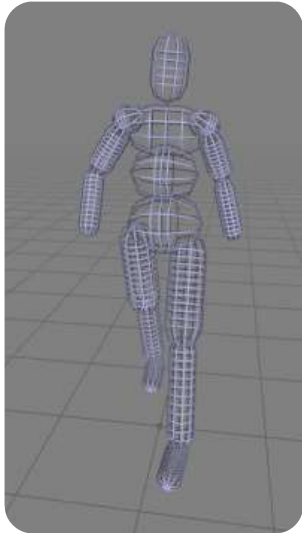
(“hit-box” is a misnomer: it’s not necessarily a “box”)

41

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.



42

Semantic of a geometric proxy

Another proxy, a point, a ray...

`intersection(proxy_A , <something>) ≠ ∅ ?`

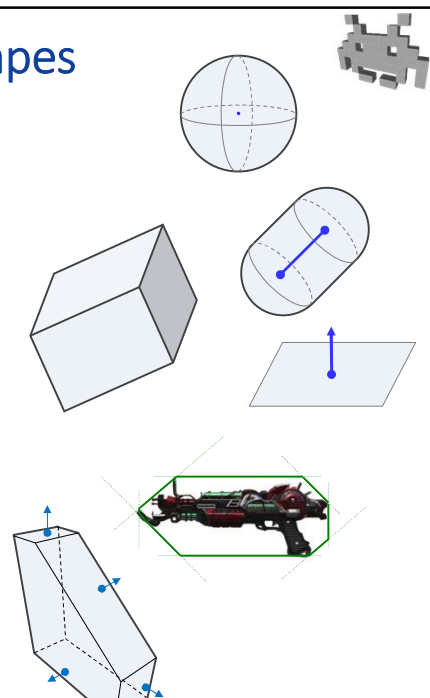
- if `proxy_A` serves as **Bounding Volume** :
 - if NO: no collision
 - if YES: we don't know yetAn «early reject» optimization
- if `proxy_A` serves as **Collider** :
 - if NO: no collision
 - if YES: **collision detected** !
 - Must compute **collision data** from `proxy_A`An approximation of the collision detection

Despite the semantic difference, the same data type can be used for all proxies.

43

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
- ...

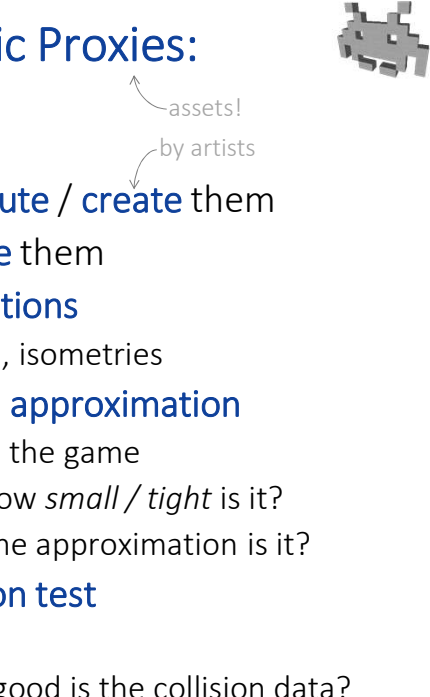


44

🧠 choosing Geometric Proxies: things to consider

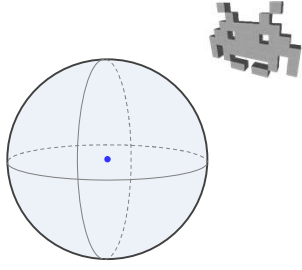
by algorithms assets!
by artists

- Workload needed to **compute** / **create** them
- RAM space needed to **store** them
- Behavior under **transformations**
 - the ones we plan to use, e.g., isometries
- 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 *close* the approximation is it?
- Workload for an **intersection test**
 - with other proxies ...
 - also, is it easy to compute / good is the collision data?



45

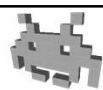
Geometry proxies: Sphere



- 😊 easy to compute automatically
 - only the approximatively 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?

46

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



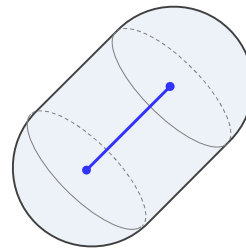
- an implementation choice of the **Physics Engine**
- # of intersection-test algorithms to be *implemented* :
quadratic with # of supported types

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

47

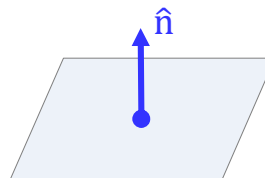
Geometry proxies: «Capsule»



- 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!)

48

Geometry proxies: a half-space



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

49

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)

$(n_x, n_y, n_z, k) \cdot (q_x, q_y, q_z, 1)$

a 4D vector representing the plane

50

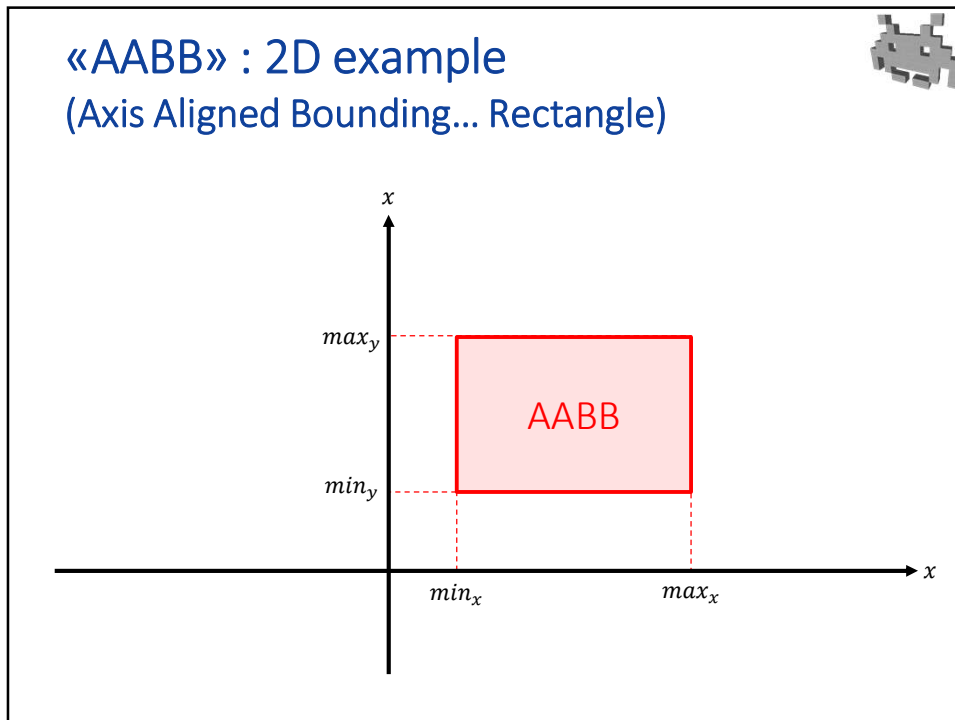
Geometry proxies: «AABB»

As the name implies, almost always used as BOUNDING volume

Axis Aligned Bounding Box

- Consists of three interval
 - $[min_x, max_x] \times [min_y, max_y] \times [min_z, max_z]$ (Cartesian product)
- 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, etc
 - (how?)
- Transforms:
 - $\otimes \otimes \otimes$ cannot be rotated
 - But can be easily scaled / translated

51



52

Geometry proxies: Box

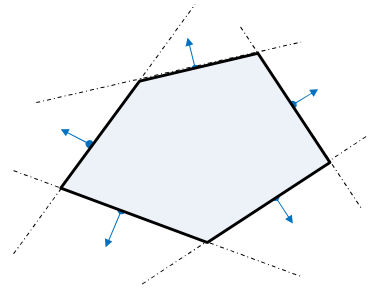
- “Parallelepiped”
 - non axis aligned
 - generalized version of AABB
 - storage:
 - a rotation +
 - an AABB
 - Can be freely transformed
 - note: only if scaling is uniform
 - Tests: still relatively easy (how?)

The diagram shows a 3D coordinate system with three axes. A light blue parallelepiped is shown, which is a 3D box rotated relative to the axes. It is drawn with perspective, showing its top, front, and side faces.

53

Geometry proxies (in 2D): a Convex Polygon

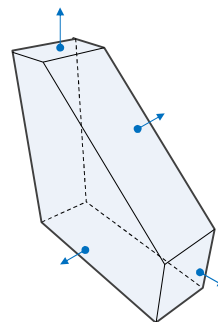
- Intersection of half-planes
 - each delimited by a line
- Stored as:
 - a collection of (oriented) lines
- Test:
 - a point is inside the proxy iff it is in each half-plane
- Flexible (good approximations)... and still moderate complexity



54

Geometry proxies (in 3D): a Convex Polyhedron

- Intersection of half-spaces
- Same as prev, put in 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



55

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



potentially *concave*

not worth it for
a *Bounding Volume* !

- Luxury *Colliders* :)
 - 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 later)
- 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, lecture 6)

56

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 != outside)
 - sometimes: *convex* only
 - completely different internal representation (as a set of bounding planes)

57

BSP-tree (Binary Spatial Partitioning tree)

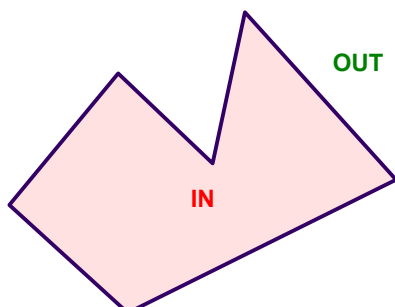


- A way to store a (convex, or concave) polyhedron
- A hierarchical structure
 - a binary tree
 - root = all space, child-nodes = partition of parent
 - each internal node is split by an *arbitrary* plane
 - plane is stored at node, as (n_x, n_y, n_z, k)
 - each leaf: one bit: “inside” or “outside” the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

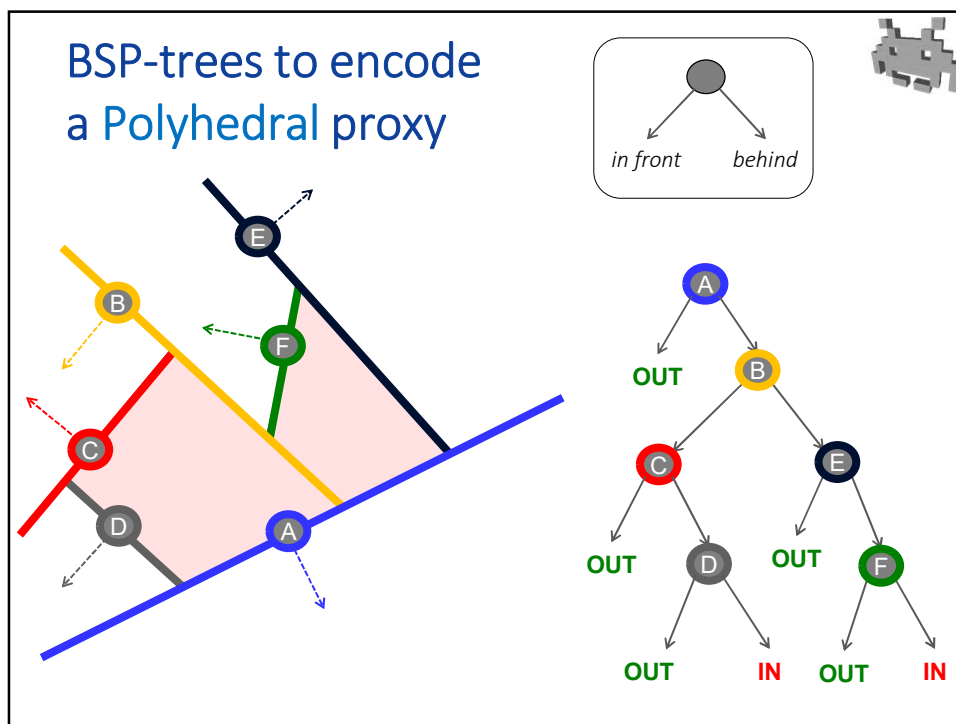
in 2D: a line

58

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



59

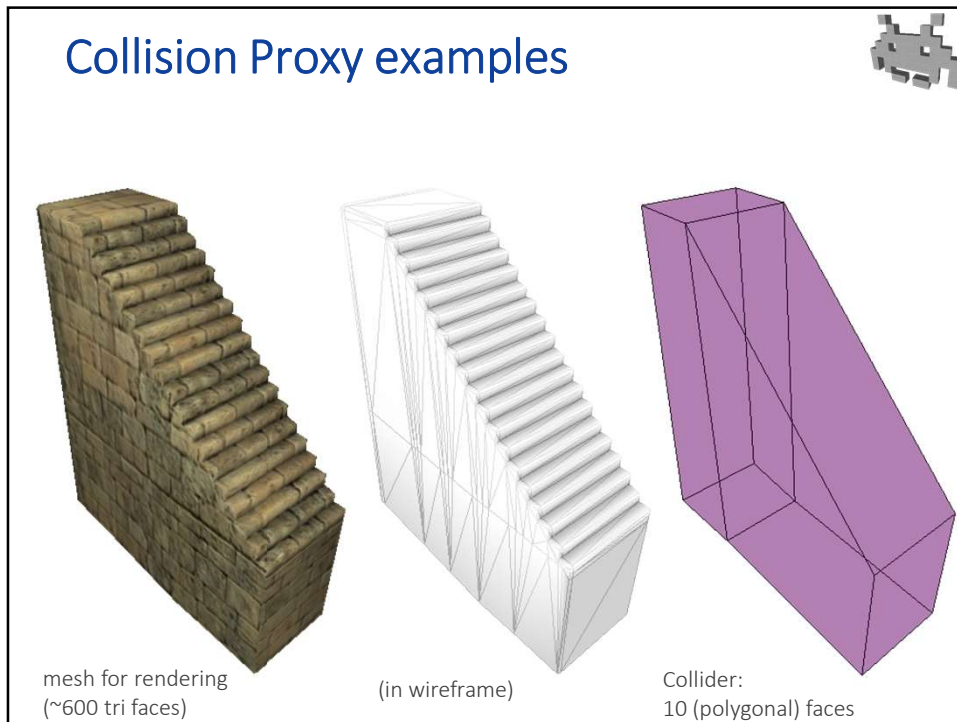


60

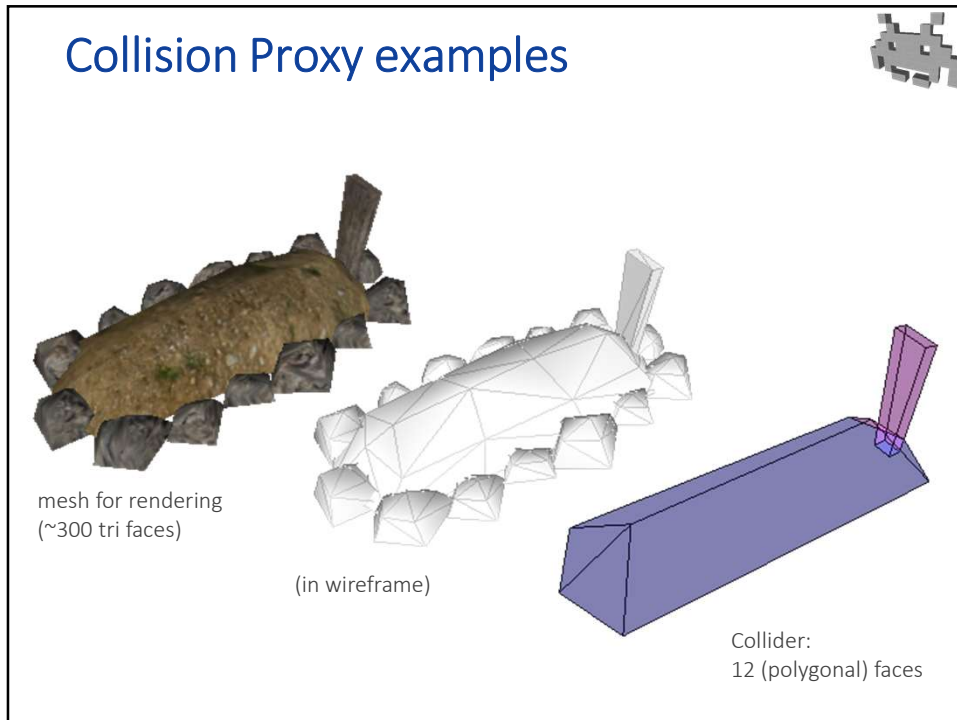
Composite Geometry Proxies

- 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 few proxies
 - note: union of **convex** proxies can be **concave** !
- Still quite efficient to store / test
- Very difficult to construct automatically
 - Open problem

61



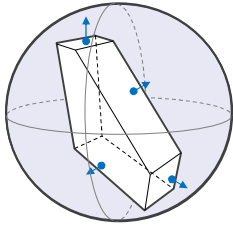
62



63

Bounding Volume + Collision Proxy

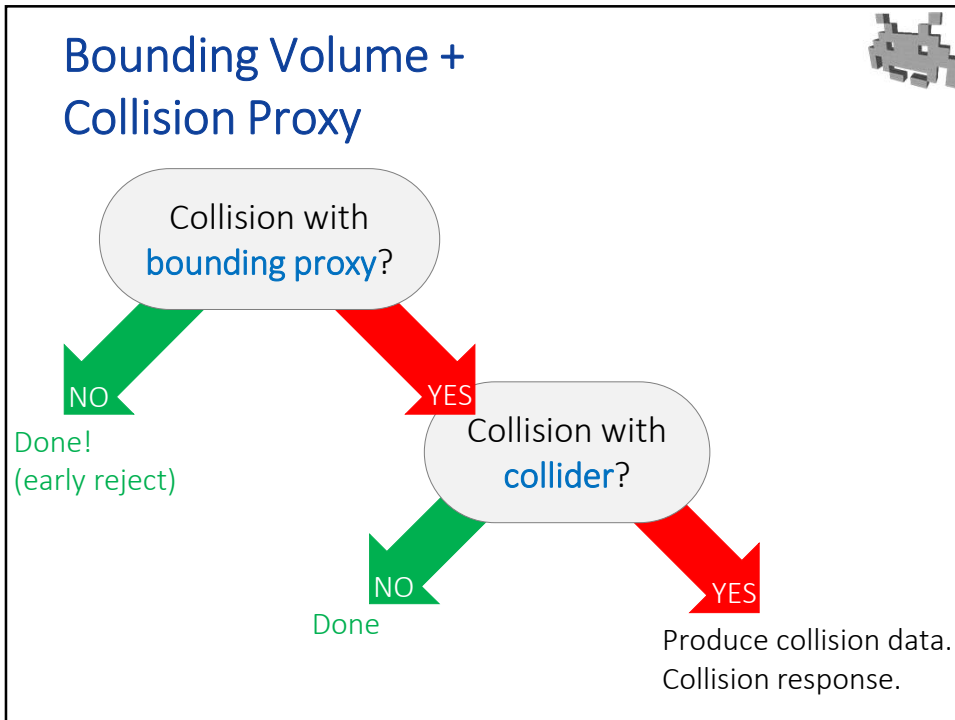
```
if (!intersect( boundingVol, X ) )  
{  
    // nothing to do: early reject!  
}  
else {  
    CollisionData d;  
    if (collide( hitBox, X , &d ))  
    {  
        collision_rensponse( d );  
    }  
}
```



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

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

64



65

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



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

Geometry proxies for colliders are **assets** !

66

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



- “Given an object representation M , build a thigh **bounding volume** for it”
 - a M = 3D model of e.g. a dragon, a castle, a character...
- It’s difficult to find the optimal (smallest possible) bounding volume automatically
- A lot easier to find a “good enough” bounding volume.
- For example, think about an algorithm to find bounding volumes of type...
 - AABB (trivial)
 - Sphere – i.e. a “bounding sphere” (less trivial)
 - Capsule (difficult!)

67

Collision detection: strategies

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

68

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

69

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)

70

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

71

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 ?

72

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 board 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

73

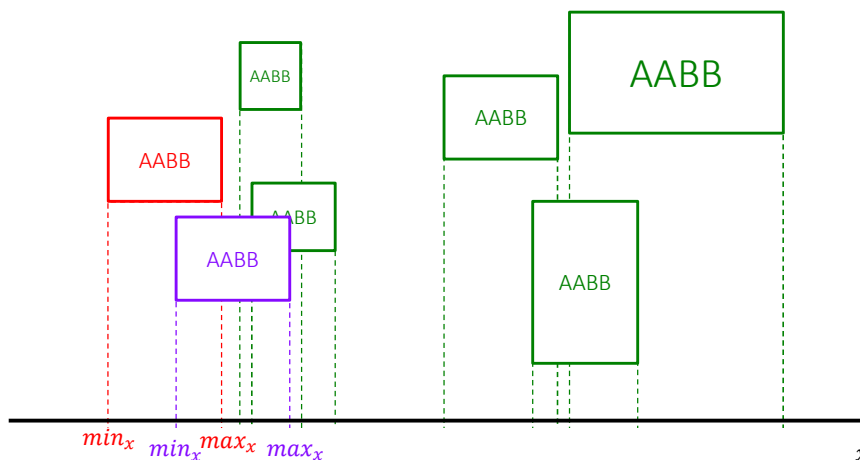
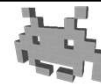
The «broad-phase» of coll. detection (avoiding quadratic explosion of # of tests)



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

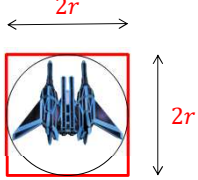
74

Sorting based algorithms Sweep and Prune (SAP)



75


Sweep And Prune (SAP) strategy (or “Sort and Sweep”)



- Bound:**
 - Quickly find the AABB for each collider (in its current rotation + translation)
 - E.g.: use the AABB encapsulating the transformed Bounding Sphere
- 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... only $O(n \log n)$
 - use an *incremental* sorting algorithm, such as quicksort ← Even faster! $O(n)$
- Sweep** the sorted intersections, from smaller to larger
 - Quickly detect intersecting intervals in x (how?)
- Prune:** among AABB intervals, ignore the ones that don't *also* intersect in both y and z
 - Test the other pairs for collision

76

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



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

77

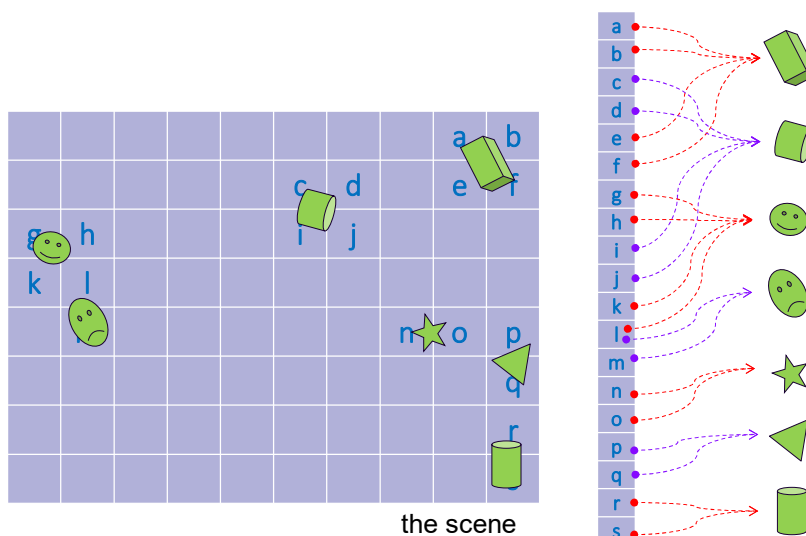
Spatial indexing structures



- 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**

78

Regular Grid (or: lattice)



79

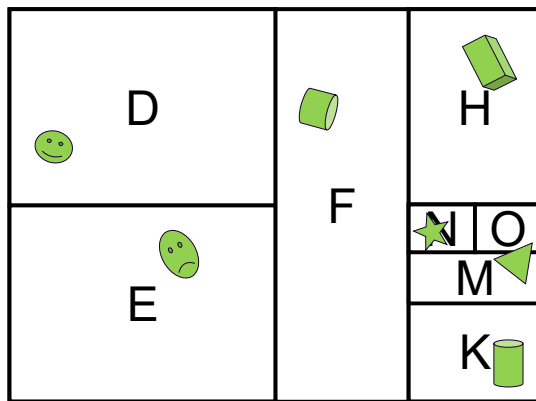
Regular Grid (or: lattice)



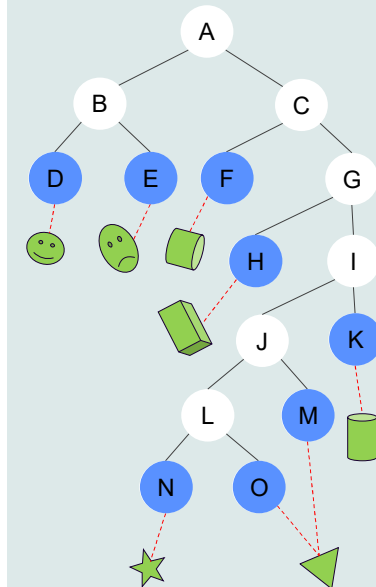
- Array 3D of cells (all the same size)
 - each cell = a list of pointers to collision objects
- Indexing function:
 - Point3D \rightarrow cell index, (constant time!)
- Construction: (“scatter” approach)
 - for each object B, find all the cells it touches, add a pointer to B to them
- Queries: (“gather”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

80

kD-trees



the scene



81

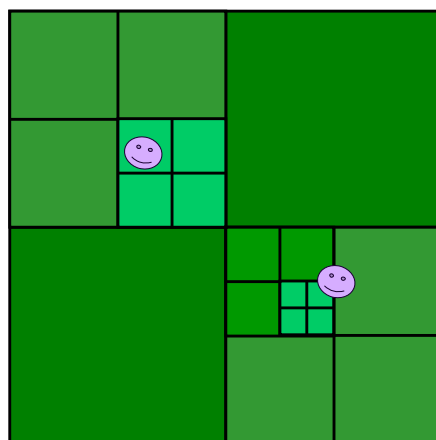
kD-trees



- 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

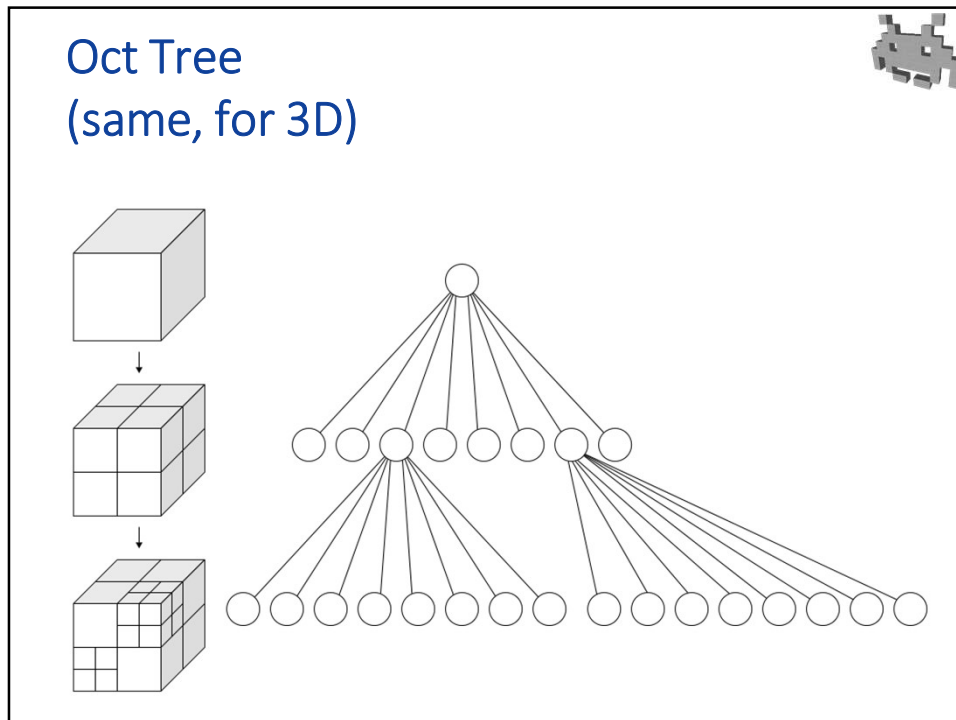
82

Quad-Tree (in 2D)



the (2D) world


83



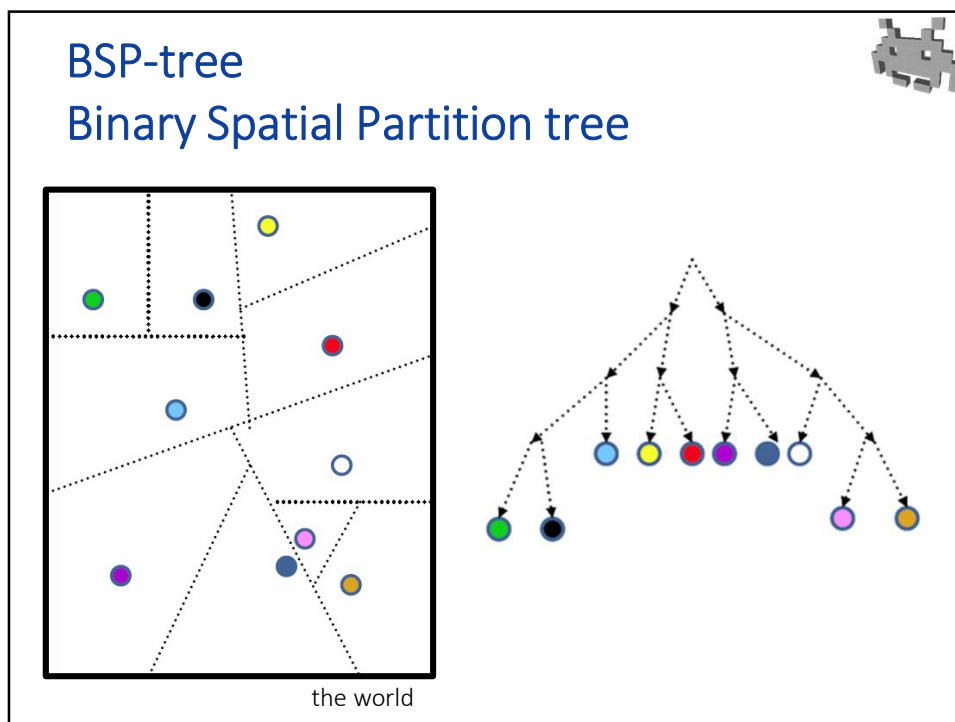
84

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

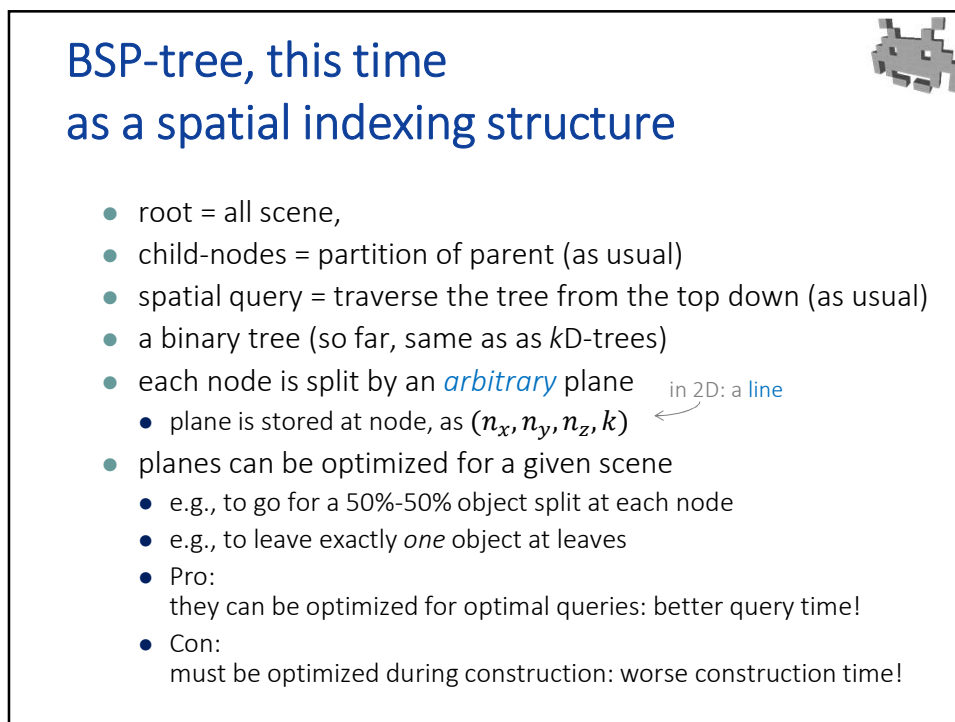
- Similar to kD-trees, but:
 - tree: branching factor: 4 (in 2D) or 8 (in 3D)
 - each node: splits into all dimensions at once
 - X and Y in 2D
 - X and Y and Z in 3D
 - (in the middle)
- Construction (just as kD-trees):
 - continue splitting until a end nodes has few enough objects
 - (or limit level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



85



86



87

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



- Classes of solutions:

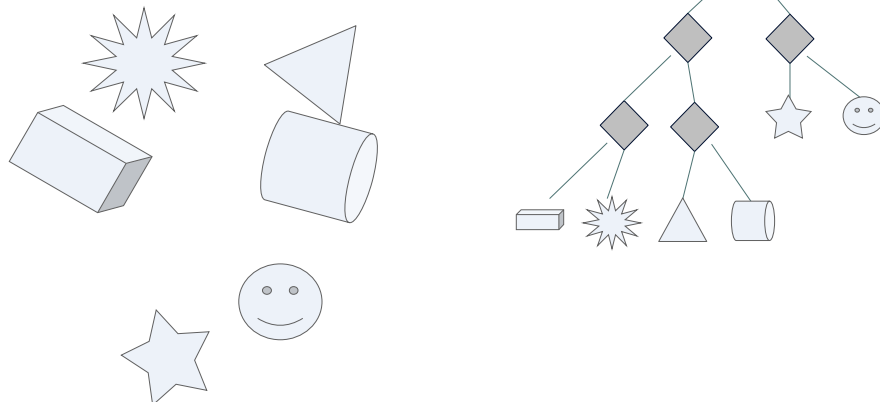
1) spatial indexing structures

2) BVH – Bounding Volume Hierarchies

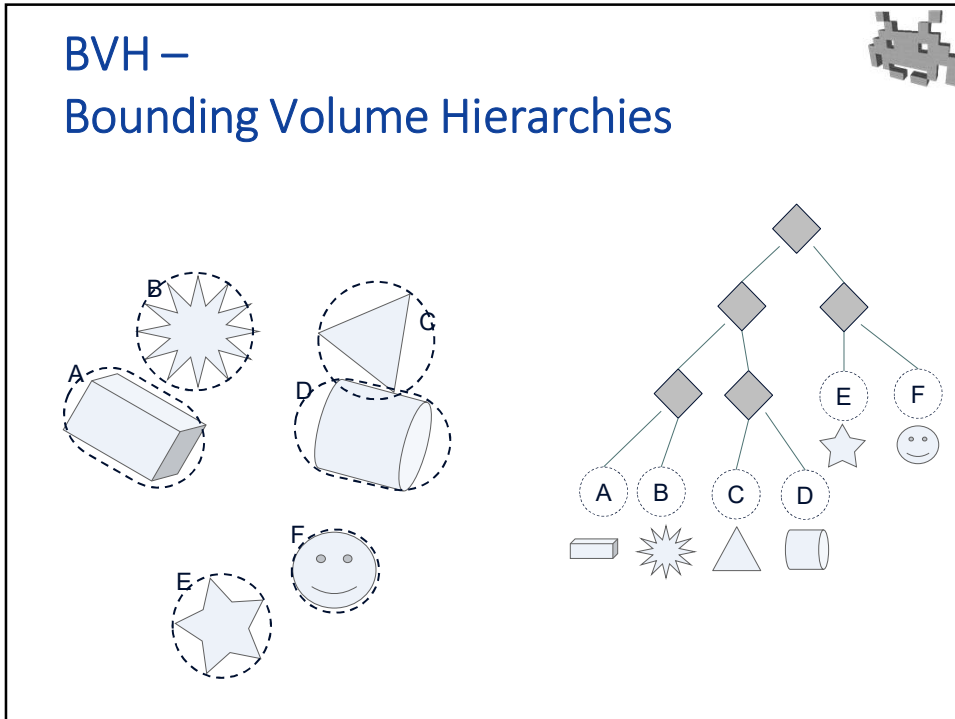
3) Sorting-based algorithms

88

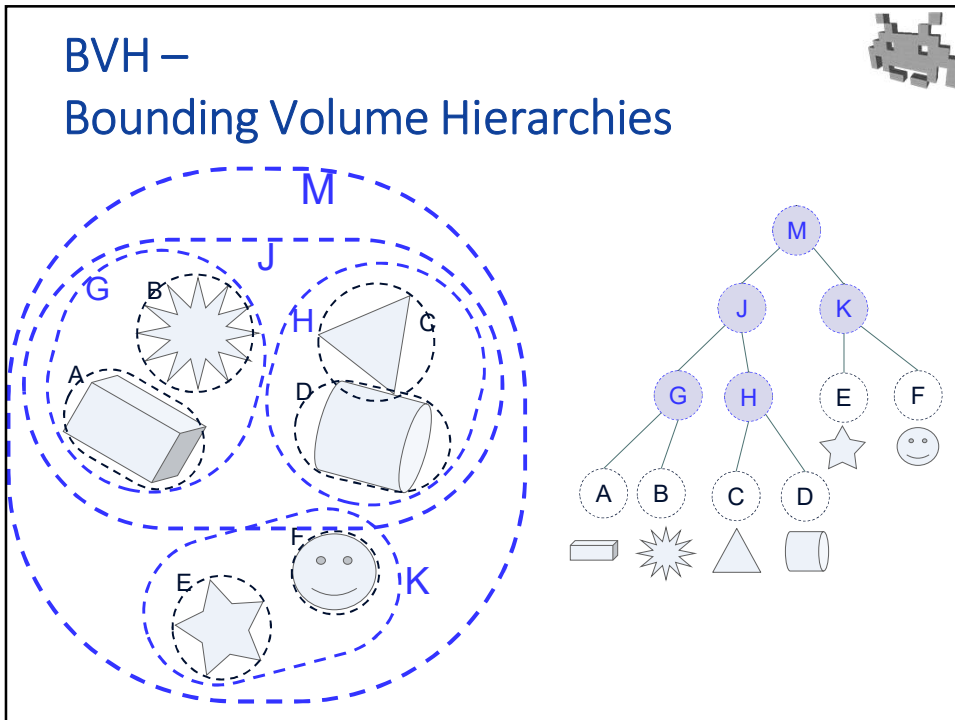
BVH Bounding Volume Hierarchy



89



90



91

BVH

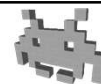
Bounding Volume Hierarchy



- 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)

92

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?**

93

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)

94

End of Game Physics part. 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/>

95