COMP 476

Advanced Game Development

Session 9
Game Physics

(Reading: Millington § 1.3, 7, 12-13)

Lecture Overview

- Collision Detection
- Bounding Volume Hierarchies
- Spatial Partitioning
- Collision Geometry

Introduction

- Physics deals with motions of objects in virtual scene
  - And object interactions during collisions
- Physics increasingly (but only recently, in last few years) *important for games*
  - Similar to advanced AI, advanced graphics
- Now have additional processing for more
  - Duo/Quad/Multi-core processors
  - Physics hardware (Ageia's PhysX PPU) and general GPU (GPGPU; CUDA)
  - Physics middleware (Nvidea PhysX; Havok FX; Box2D) that are optimized

Université Concordia University

Introduction

- **Potential**
  - **New gameplay elements/mechanics (e.g., Half-Life 2, Half-Life Alyx, Inversion)**
  - **Realism (*i.e.*, gravity, water resistance, cloth movement, *etc.*)**
  - **Particle effects**
  - **Improved collision detection**
  - **Rag doll physics (when coupled with animation)**
  - **Realistic motion**

Physics Engine

Build or Buy?

- Physics engine *<u>can be part of a game engine</u>*
- <u>License</u> middleware physics engine
  - Complete solution from day 1
  - Proven, robust code base (in theory)
  - Features are always a trade-off
- Build physics engine <u>in-house</u>
  - Choose only the features you need
  - Opportunity for more game-specific optimizations
  - Greater opportunity to innovate
  - Cost can be easily be much greater

Newtonian Physics

- Sir Isaac Newton (around 1700) described *three laws*, as basis for classical mechanics:

1. A body will remain at rest or *continue to move* in a straight line *at a constant velocity unless acted upon by another force*
2. The acceleration of a body is proportional to the resultant force acting on the body and is in the same direction as the resultant force.
3. For every action, there is *an equal and opposite reaction*

- More recent physics show laws break down when trying to describe universe (thanks, Einstein), but good for computer games

Newtonian Physics

- Generally, an object does not come to a stop naturally, but *forces must bring it to stop*
  - Force can be friction (*i.e.*- ground)
  - Force can be drag (*i.e.*- air or fluid)
  - Can be due to non-perfect collisions
  - External forces: gravitational, electromagnetic, weak nuclear, strong nuclear
    - But gravitational most common in games (and most well-known)
- From dynamics :
  - Force = mass x acceleration (F=ma)

Newtonian Physics

- In games, forces often known (F), so need to calculate acceleration ($\ddot{p}$, meaning $\frac{d^2p}{dt^2}$ or $p^{(2)}(t)$)
  - add up all forces on object (F) and divide by mass (m) to get acceleration ($\ddot{p}$):

$$\ddot{p} = F/m$$

- Acceleration used to update velocity ($\dot{p}$) and then, velocity used to update objects position (p):
  - $p = p + (\dot{p} + \ddot{p}t)t$ (t is the time since last update)
  - Can do same for (x, y, z) positions

UNIVERSITÉ Concordia UNIVERSITY

Newtonian Physics

- **Kinematics** is ***study of motion of bodies and forces*** **(without considering their cause) acting upon bodies**
- **Three main types of bodies:**
  - **Point masses/particles – no angles, so only linear motion (considered infinitely small)**
    - **Particle effects**
  - **Rigid bodies – shapes do not change, so deals with angular (orientation) and linear motion**
    - **Characters and dynamic game objects**
  - **Soft bodies – have position and orientation and** ***can change shape*** **(*i.e.*, cloth, liquids)**
    - **Starting to be possible in real-time**

Rigid-Body Simulation

- In many games (and life!), interesting motion involves *non-constant forces and collision impulse forces*
- Unfortunately, for the general case, *often no closed-form solutions*
- Numerical simulation:

> ***Numerical Simulation*** **represents a series of techniques for incrementally solving the equations of motion when forces applied to an object are not constant, or when otherwise there is *no closed-form solution***

Numerical Integration

Newtonian Equation of Motion

- Family of numerical simulation techniques called ***finite difference methods***
  - The most common family of numerical techniques for rigid-body dynamics simulation
  - Incremental "solution" to equations of motion
- Derived from Taylor series expansion about $x = c$ of property $f(x)$ we are interested in

$$\sum_{i=0}^{n} \frac{f^{(i)}(c)}{i!}(x-c)^i$$

(Taylor series are used to estimate unknown functions)

Numerical Integration

Newtonian Equation of Motion

$$S(t+\Delta t) = S(t) + \Delta t \, d/dt \, S(t) + ((\Delta t)^2/2!) \, d^2/dt^2 \, S(t) + \ldots$$

- In general, do not know values of any higher order. Truncate, remove higher order terms:

$$S(t+\Delta t) = S(t) + \Delta t \, d/dt \, S(t) + O((\Delta t)^2)$$

  - Can do beyond, but always higher order terms
  - $O((\Delta t)^2)$ is called truncation error
- Can use to update properties (position)
  - Called "simple" or "explicit" Euler integration

Explicit Euler Integration

- A "one-point" method solution using the properties at exactly one point in time, *t*, prior to the update time, t+Δt.
  - S(t+Δt) is ***the only unknown value*** so can solve without solving system of simultaneous equations
  - Important – every term on right side is evaluated at t, right before new time t+Δt

- View: S(t+Δt) = S(t) + Δt d/dt S(t)

  new state | prior state | state derivative

- For single particle, S=$(m\dot{p}, p)$ and d/dt S = $(F, \dot{p})$

(§1.3, 7)

Types of Physics Engines

Physics Engines:

Approaches

- There are several different approaches to building a physics engine.
- There are some broad distinctions that we can use to categorize the differences:
  1. Types of Object
  2. Contact Resolution
  3. Impulses and Forces

Physics Engines:

Types of Objects

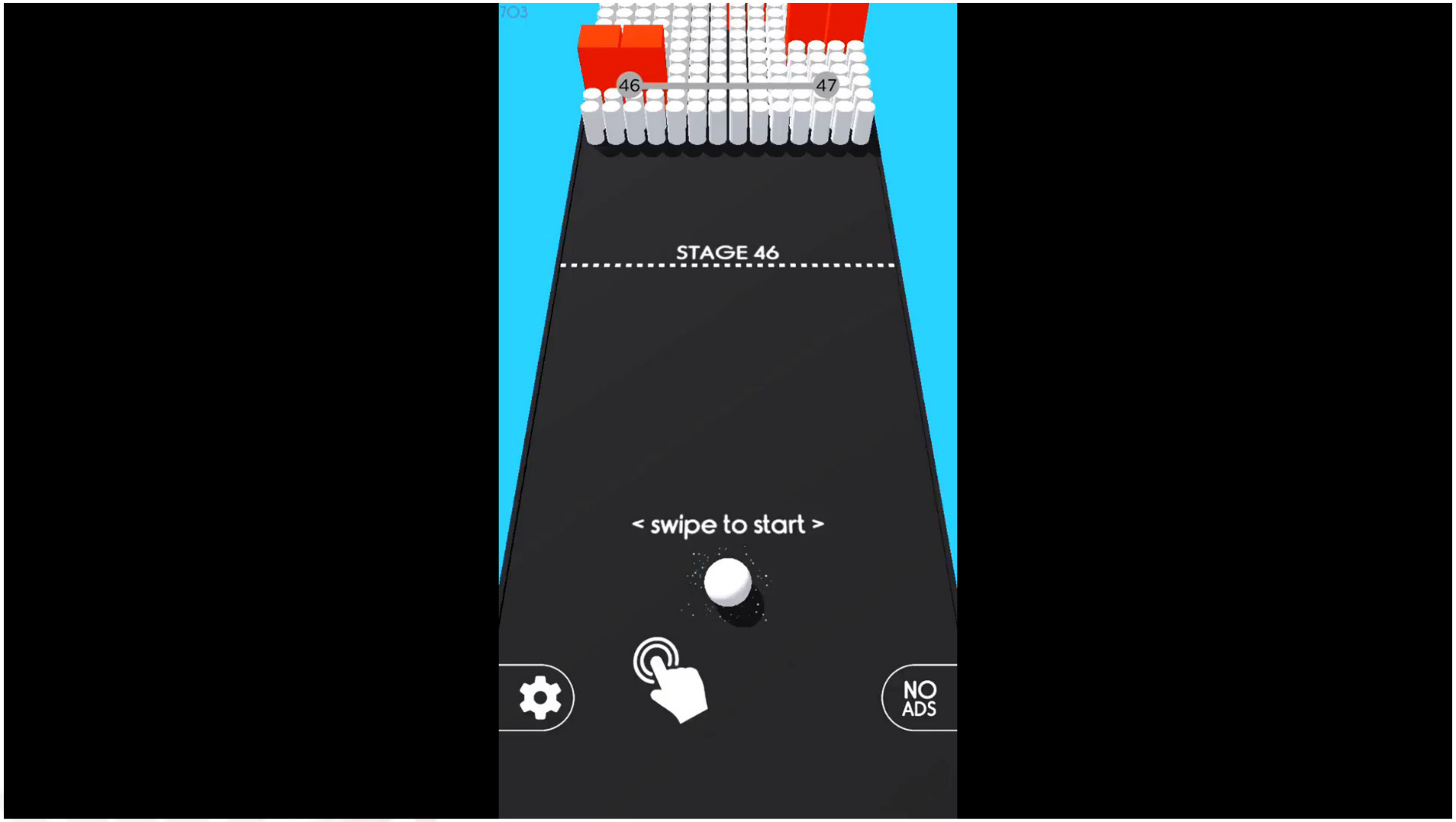
- **Rigid-body Engines**:
  - Treat objects as a whole and work out the way they move and rotate.
- **Mass-aggregate Engines**:
  - Treat objects as if they were made up of lots of little masses (connected to each other by rods).
  - Easier to program because they *don't need to understand rotations*.
  - Difficult to make really firm objects in a mass-aggregate system.
- We will focus on Rigid-body Engines.

Physics Engines:

Contacts

- A lot of the difficulty in writing a rigid-body physics engine <u>is simulating contacts</u>: *locations where two objects touch or are connected*.
- This includes objects resting on the floor (aka resting contacts), objects connected together, and to some extent collisions.
- There are a number of approaches to handle such contacts:
  1. "iterative approach"
  2. "Jacobian-based"
  3. "reduced coordinate approach"

Color Bump 3D



https://www.youtube.com/watch?v=2DnIJClS8Hk

Handling Contacts

- "iterative approach": <u>contacts handled one by one</u>
  - Advantage: speed - each contact resolved fast
  - Disadvantage: one contact can affect another, and these interactions can be significant.
- "Jacobian-based": calculate the exact interaction between different contacts and calculate an overall set of effects to apply to all objects at the same time. Slow and may fail to find solution.
- "reduced coordinate approach": calculate a new set of equations based on the contacts and constraints between objects. Very slow and accurate, but not useful for games.
- We will use the iterative approach.

Université Concordia University

Physics Engines:

Impulses or Forces

- To resolve contacts we could use either impulses or forces.
- Impulses: a change in velocity is caused by a force, but the force acts for such a small fraction of a second that it is easier to think of it as simply a change in velocity.
- Some game engines use:
  - Impulses for collisions; Forces for resting contacts. (rarely done)
  - Forces for both (treating impulses as forces over small period of time)
  - Impulses for both (which we will do)

Overview of Collision Resolution

Simple Collision Resolution

- **Collision**: any situation in which two objects are touching (including objects in contact).
- When two objects collide, their movement after the collision can be calculated from their movement before the collision: ***this is collision resolution***.
- **Closing (Separating) Velocity**, v_c (v_s): the total speed (a scalar) at which the two objects are moving together (apart).
- **Coefficient of Restitution**, c: controls the speed at which the objects will separate after colliding. It depends on the materials in collision.

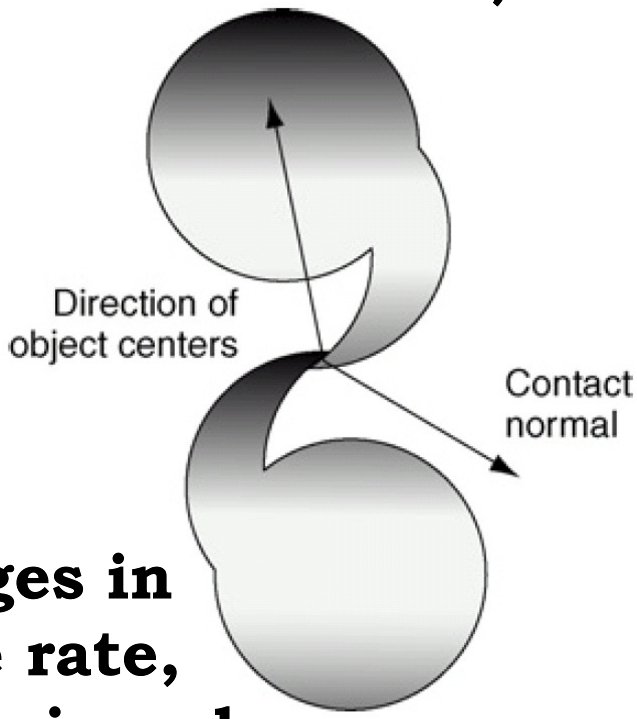
$$v_s' = -cv_s$$

Simple Collision Resolution

- **Collision Direction and Contact Normal**, $\boldsymbol{n}$: the (normalized) direction in which the two objects are colliding (also called the ***collision normal***). For two particles at positions $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$,

$$\boldsymbol{n} = (\boldsymbol{p}_a - \boldsymbol{p}_b)/|\boldsymbol{p}_a - \boldsymbol{p}_b|$$

- For rigid bodies, the normal ***depends on the geometry of the contact***.
- **Impulses**: instantaneous changes in velocity. In terms of the frame rate, collisions are instant so we use impulses instead of using forces (accelerations) to model collisions.



Collision Processing

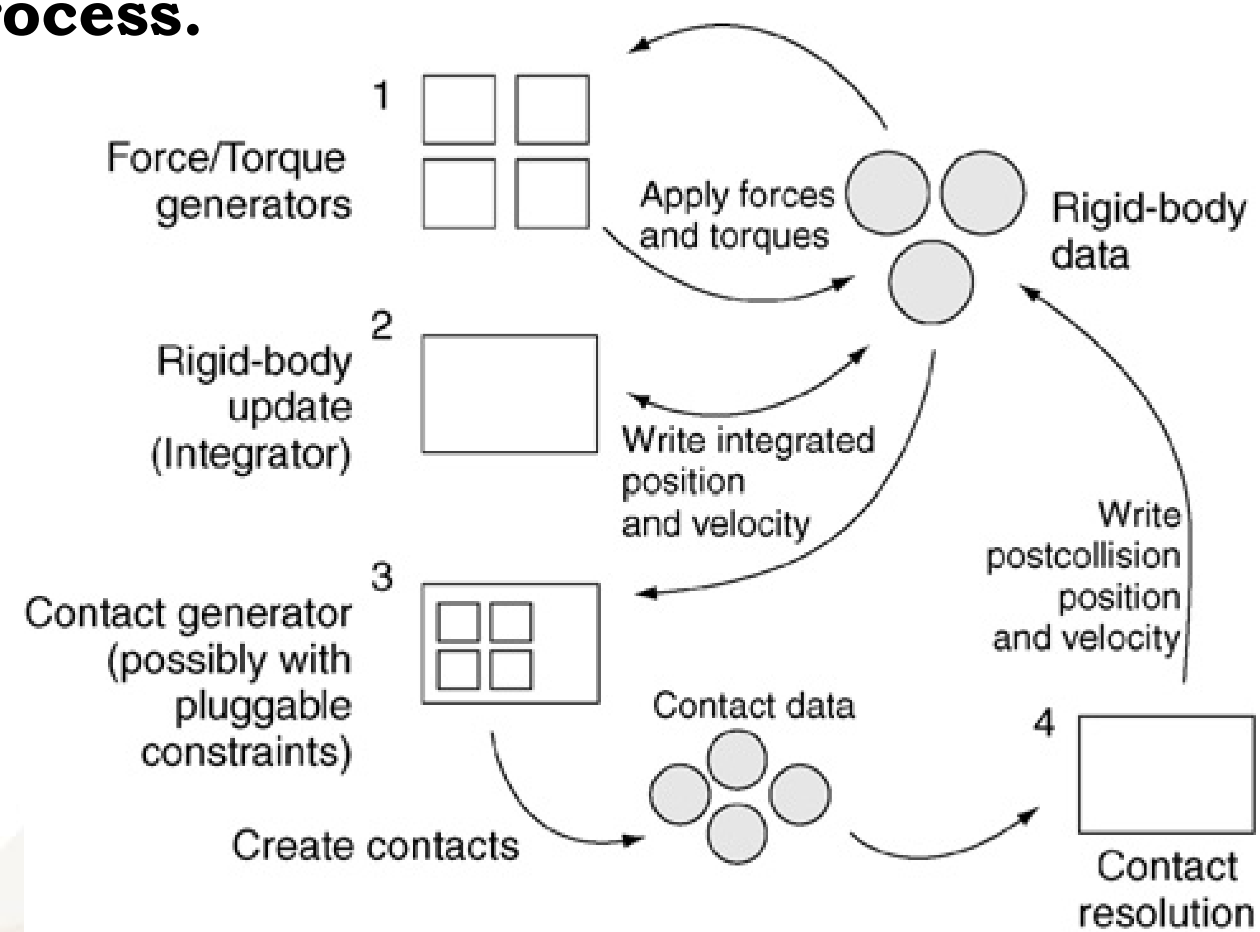
- A *collision detector* is a chunk of code responsible for finding pairs of objects that are colliding or single objects that are colliding with some piece of immovable scenery.
- Collision detection obviously needs to take the *geometries of the objects into account: their shape and size*.
- The collision detection system is responsible for calculating any properties that are geometrical, such as *when and where two objects are touching*, and *the contact normal between them*.

Collision Resolution Process

- Most of the commercial physics middleware packages process ***all the collisions at the same time*** (or at least batch them into groups to be processed simultaneously). This allows them to ***make sure that the adjustments made to one contact don't disturb others***.
- While they are more stable and accurate than the methods we consider, they are *very much more complex and can be considerably slower*.
- Instead our resolution system will look at each collision in turn, ***in order of severity***, and correct it.

Collision Resolution Process

- Below is a schematic of the collision resolution process.



Contact Resolution Algorithm

- We have three bits of code to update the objects being simulated to take account of the contacts:

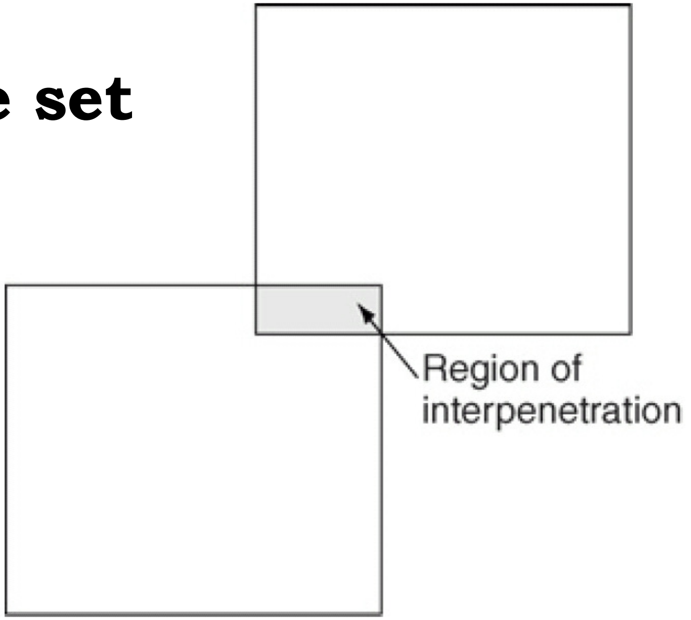
1. The collision resolution function that ***applies impulses to objects*** to simulate their bouncing apart.
2. The interpenetration resolution function that ***moves objects apart*** so that they aren't partially embedded in one another.
3. The resting contact code that sits inside the collision resolution function and ***keeps an eye out for contacts that might be resting*** rather than colliding.

Collision Resolution Pipeline

- The collision resolution routine has two components:
  - a velocity resolution system,
  - and a penetration resolution system.
- These two steps are independent of each other.
- The collision resolver takes the whole set of collisions and the duration of the frame, and it performs the resolution in three steps:
  - it calculates *internal data for each contact*;
  - then it passes the *contacts to the penetration resolver*; and
  - then they *go to the velocity resolver*

Université Concordia University

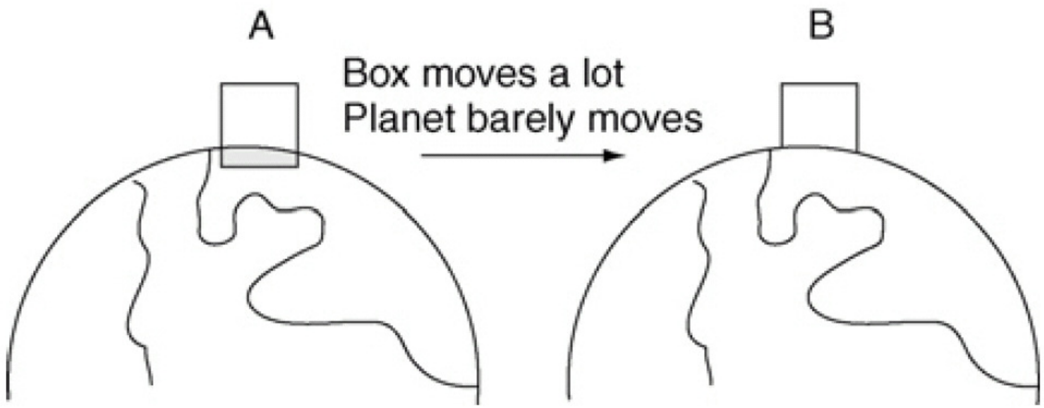
Resolving Interpenetration



- Most simply look through the set of objects and check to see *whether any two objects are interpenetrating*.
- As part of resolving the collisions, we need to resolve the interpenetration.
- When two objects are interpenetrating, we *move them apart just enough to separate them*.
- The calculation of the interpenetration depth depends on the geometries of the objects colliding. Like the closing velocity, the *penetration depth has both size and sign*.

Resolving Interpenetration

- **The penetration depth should be given in the direction of the contact normal. If we move the objects in the direction of the contact normal, by *a distance equal to the penetration depth*, then the *objects will no longer be in contact*.**
- **We also need to work out how much each individual object should be moved (*inversely proportional to their mass*).**



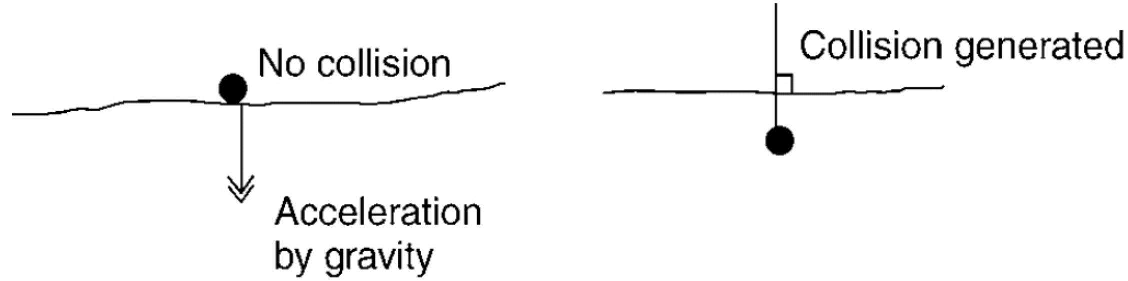
Resolving Velocity

- With penetrations resolved we can turn our attention to velocity.
- We will consider a simple velocity resolution system that works, is stable, and is as fast as possible:

while **(there are collisions with a closing velocity)** ***do***
- **find the collision with the greatest closing velocity.**
- **resolve collision in isolation.**
- **update other contacts based on the changes that were made.**

end while

Resting Contacts



- If you implement and run the collision resolution system, it will work well for medium-speed collisions, but objects at rest (a plate resting on a table, for example) ***may appear to vibrate*** and ***may even leap into the air occasionally***.
- To solve this problem we can do two things:
  1. We need to **detect the contact earlier**.
  2. We need to recognize when an object has velocity that could only have arisen from its forces acting for one frame (*e.g.*, due to gravity).
- **Instead of performing the impulse calculation for a collision, we can apply the impulse that would result in a zero separating velocity.**

Resolution Order

- **If an object has *two simultaneous contacts*, as shown, then changing its velocity to resolve one contact may change its separating velocity at the other contact.**
- **To avoid doing unnecessary work in situations like this, we *resolve the most severe contact first*: the contact with the most negative separating velocity, v_s.**
- **Subtle Complication: If we handle one collision, then we might put back a resolved collision back in collision ⇨ limit # of iterations**

(§12)

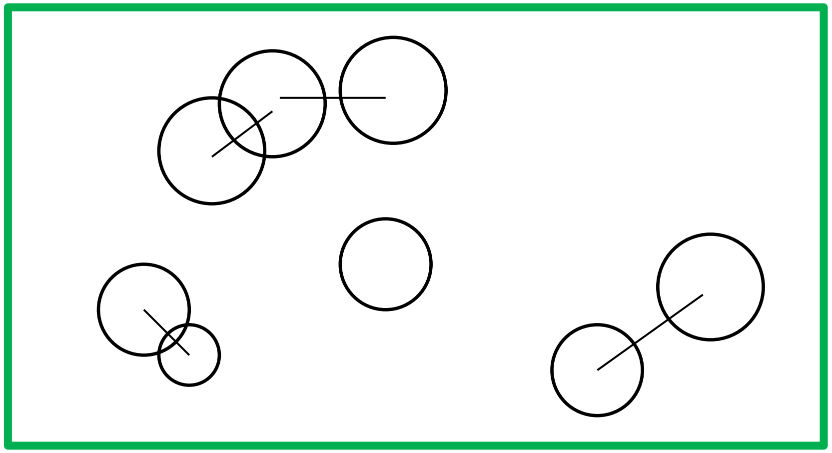
Collision Detection

Collision Detection Pipeline

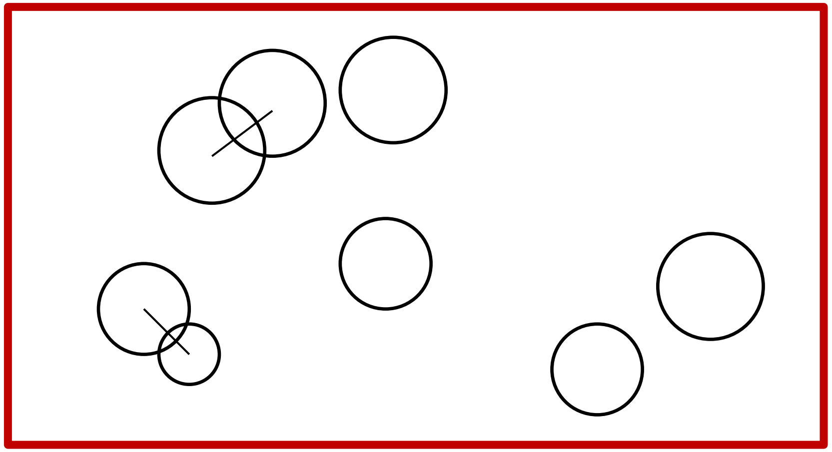
- **To reduce the number of time-consuming collision checks, we can use a two-step process:**
- **Broad phase: find all sets of objects that could possibly be in contact with each other. Typically, this uses heuristics and special data structures to eliminate the vast majority of possible collisions (that are not actually collisions)**
- **Narrow phase: determines which of the candidate collisions are actually in contact. Those in contact are examined to determine the exact data for the contact. Sometimes called contact generation.**

Collision Detection Pipeline

- **Broad phase**
  - Detects potential collisions



- **Narrow phase**
  - Checks each potential collision for actual collision

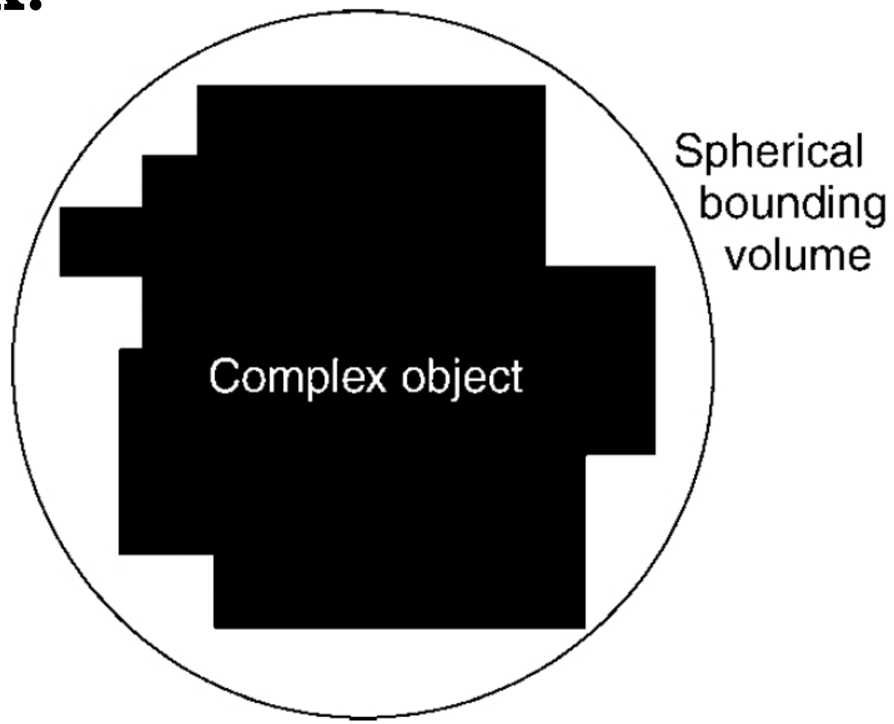


Broad Phase

- **Key features/requirements:**
  - **Should be conservative: OK to generate checks that turn out not to be collisions (false positives), but NOT OK to fail to generate checks that would be collisions.**
  - **Should generate as small a list as possible. In practice, though, many false positives are included.**
  - **Should be *as fast as possible*. No point in generating a smaller list if the detector is too slow.**

Bounding Volumes

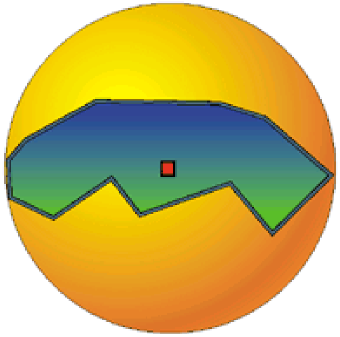
- A bounding volume is an area of space known to contain all of an object.
- Typically a simple shape is used. *e.g.*, a sphere or a box.



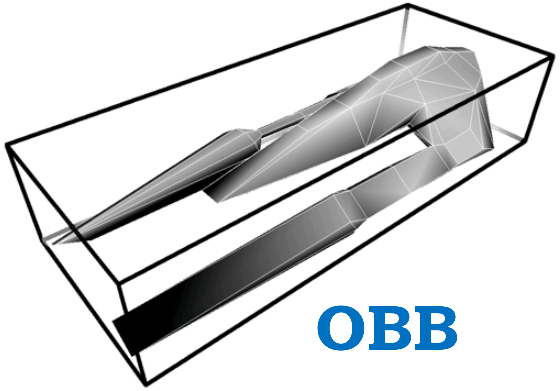
Bounding Volumes

- A simple shape can be used to *perform simple intersection tests*:
  - If the *bounding volumes don't intersect*, then obviously the objects within them don't intersect.
  - If they do intersection, further work is needed to see if objects actually intersect.
- This meets our requirements for broad phase collision detections.
- Ideally, bounding volumes should be *as close fitting to their object as possible*.

Bounding Volumes



- Common Bounding Volumes:
  - **Spheres**: store (centre, radius).
  - **Boxes**: store (centre, half-width).
  - **Rectangular Boxes**: store (centre, half-size(s)).
  - **Axis-Aligned Bounding Boxes** (AABB): Boxes aligned with world coordinates.
  - **Object-Bounding Boxes** (OBB): Boxes aligned with object coordinates.



- For tall, thin objects, a bounding box would fit more tightly, otherwise spheres are often a good place to start.

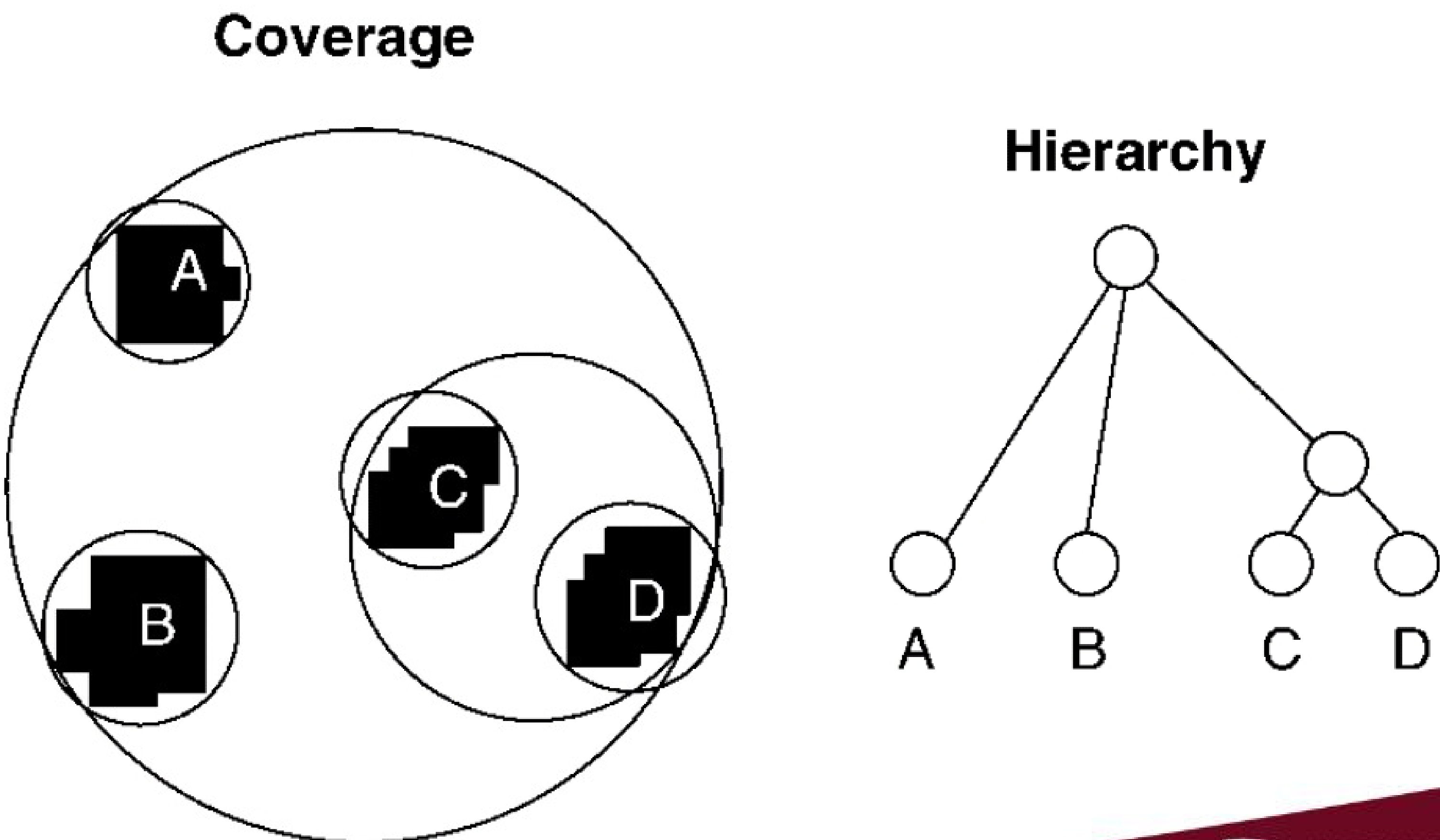
BVH

- We can avoid checking many pairs of objects for contact by ***arranging bounding volumes in hierarchies***.
- A **bounding volume hierarchy** (BVH) is a (typically binary) tree data structure that has:
  - each object in its bounding volume **at leaves**
  - the bounding volume of the parent node is ***big enough to enclose all the objects descended from it***.
- The bounding box at each level of the BVH will typically be chosen ***to best fit the bounding volumes of objects contained within it***, not the objects.

Université Concordia University

BVH

- E.g.; representative of most implementations.



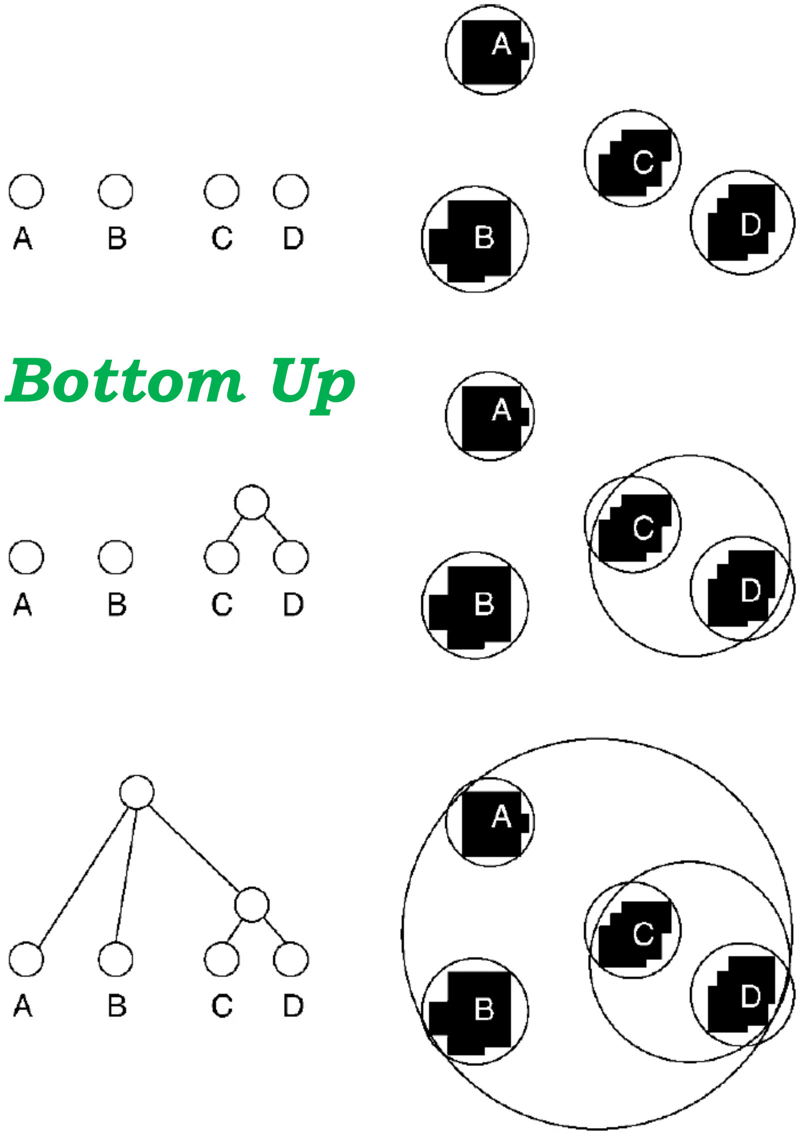
BVH

- A BVH speeds up collision detection:
  - If the bounding volumes of two parent nodes in the tree don't touch, ***none of the*** objects that ***descend*** from those nodes ***can possibly be in contact***.
- Broad Phase Collision Detection
  - If two high level nodes do touch, the children of each node need to be considered. Only combinations of these children that touch can have descendants that are in contact.
    - And so on, recursively...
- Typically much faster than considering each possible pair (sans BVH) in turn

Building a BVH

- **When building a BVH, the hierarchy should have the following properties:**
  - ***Volumes*** **of the bounding volumes should be** ***as small as possible*****.**
  - **Children bounding volumes of any parent should** ***overlap as little as possible*****.**
  - **Tree should be balanced.**
- **For static worlds, the BVH** ***can be built offline*****.**
- **For** ***very dynamic worlds*****, the BVH needs to be rebuilt occasionally during the game.**

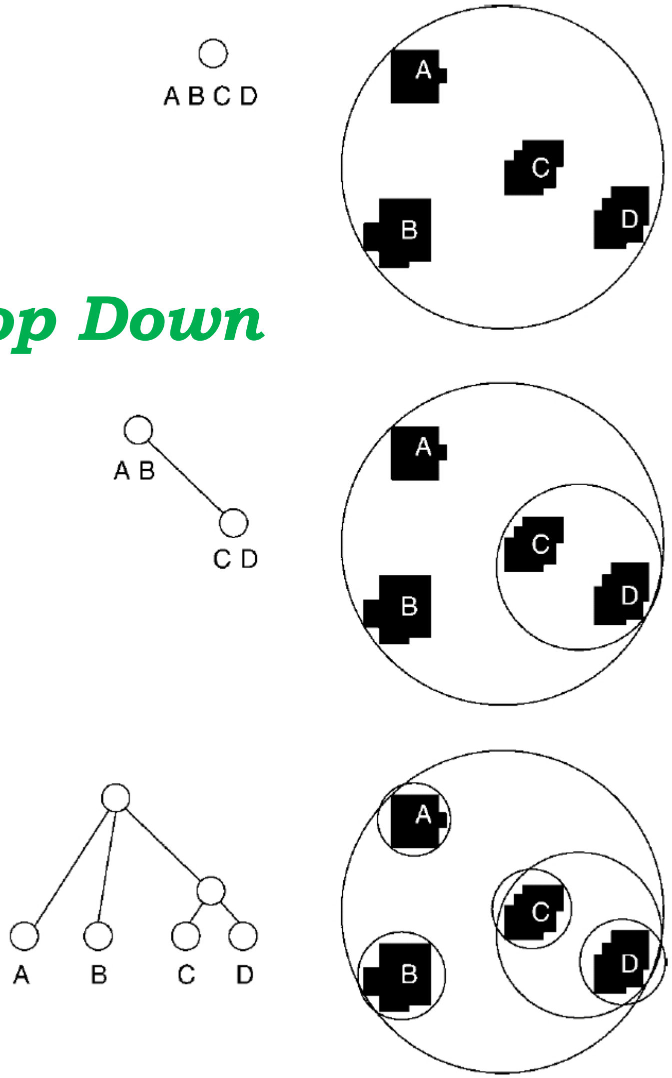
Building a BVH



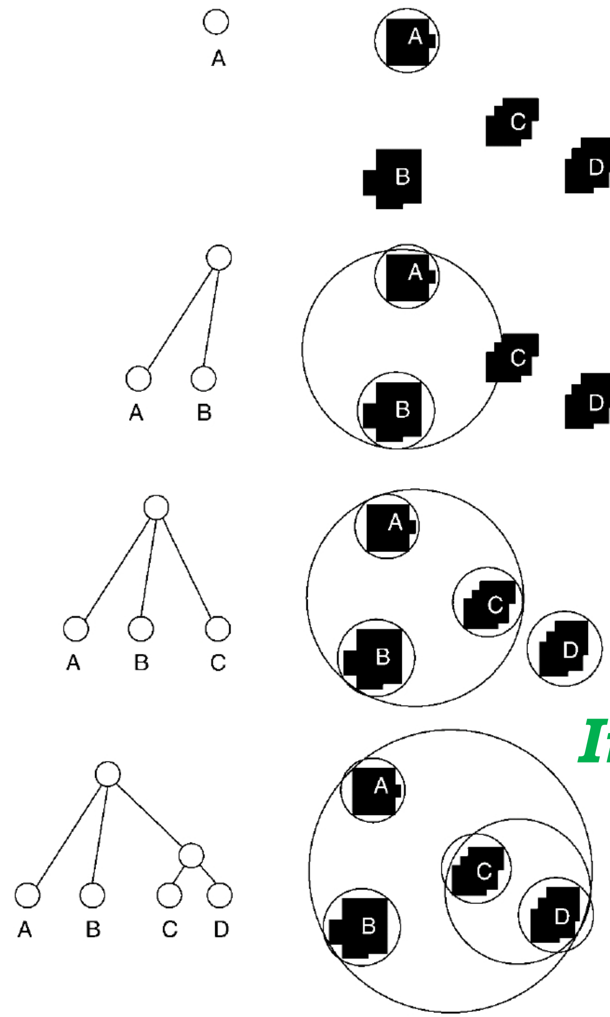
- There are three algorithm families for building a BVH:
  - Bottom Up
  - Top Down
  - Insertion

Building a BVH

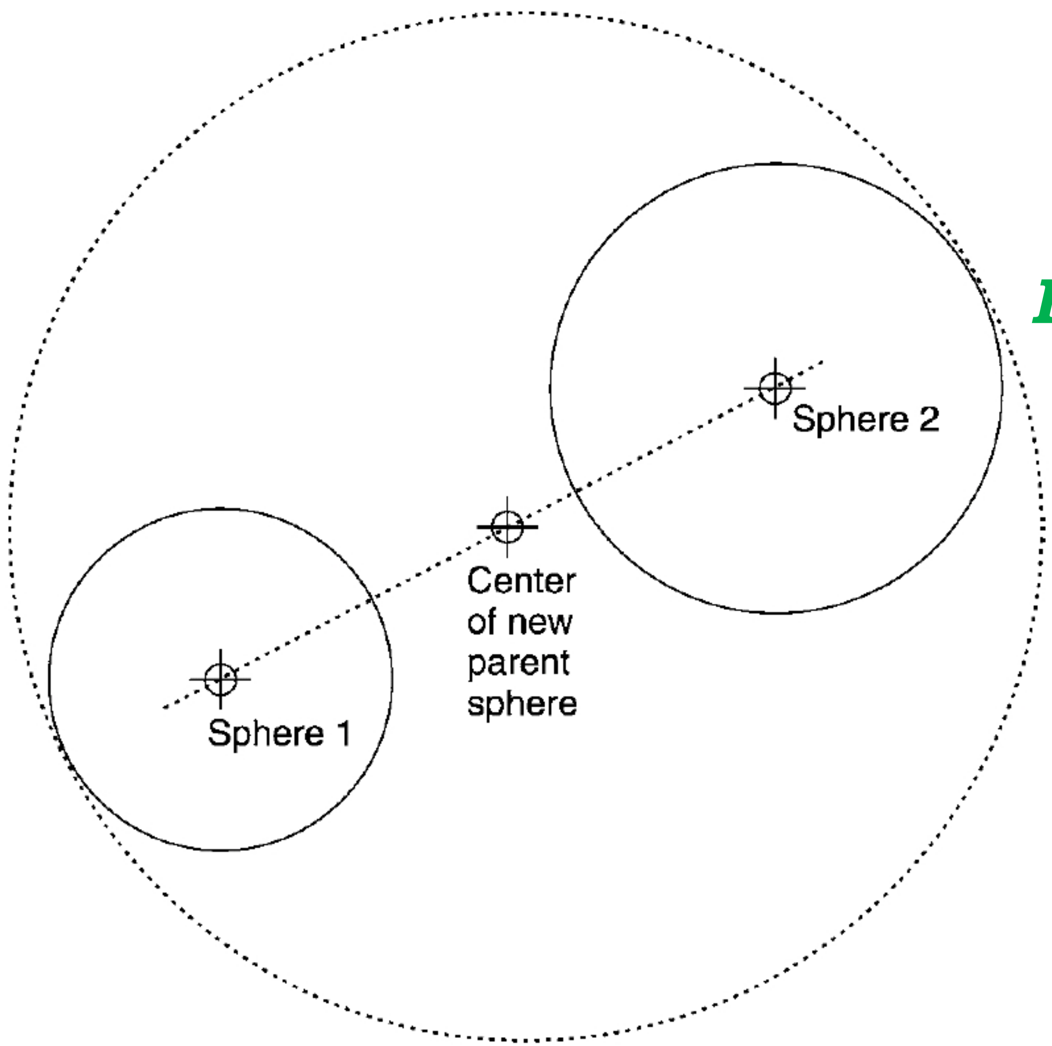
Top Down



Insertion

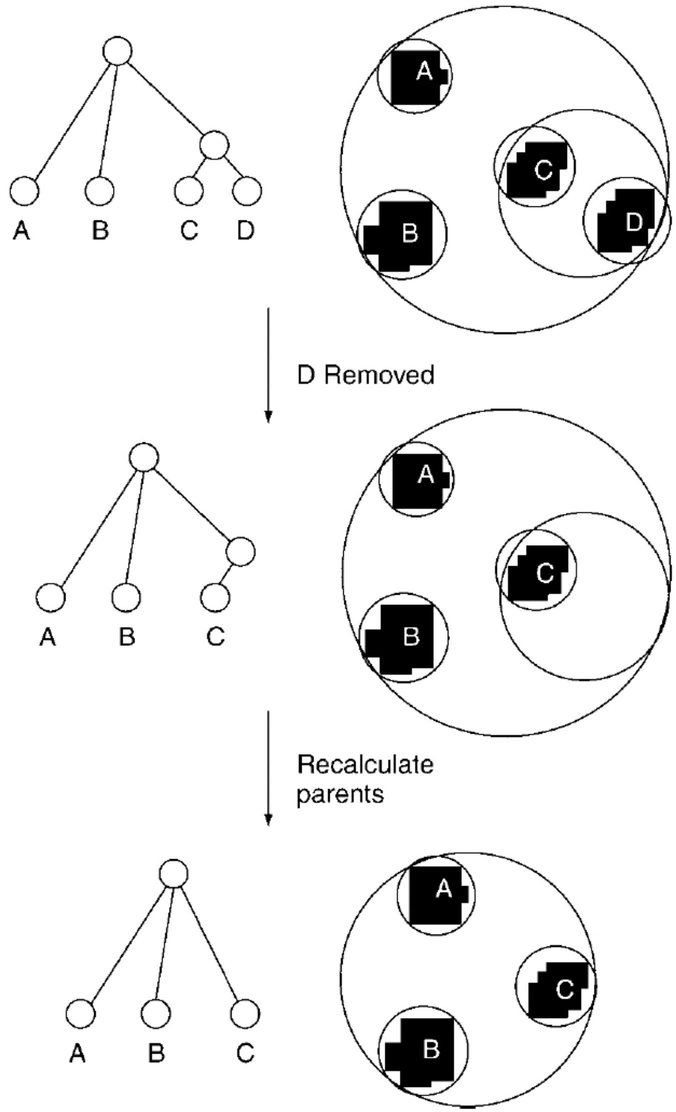


Insertion Hierarchy Building



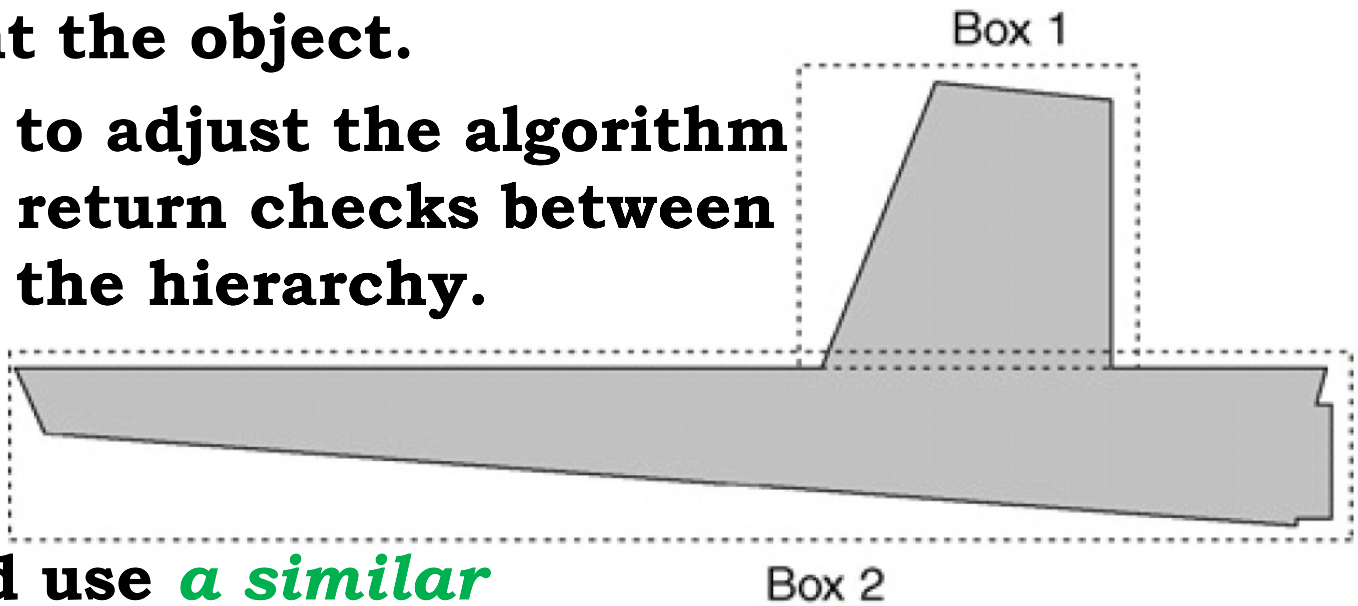
Insertion

Insertion Hierarchy Building



Sub-Object Hierarchy

- Some objects *are large and awkward*. Using a single bounding box would be too large and lead to *too many false positives*.
- Instead we use a *hierarchy of bounding boxes* to represent the object.
- We have to adjust the algorithm to never return checks between boxes in the hierarchy.



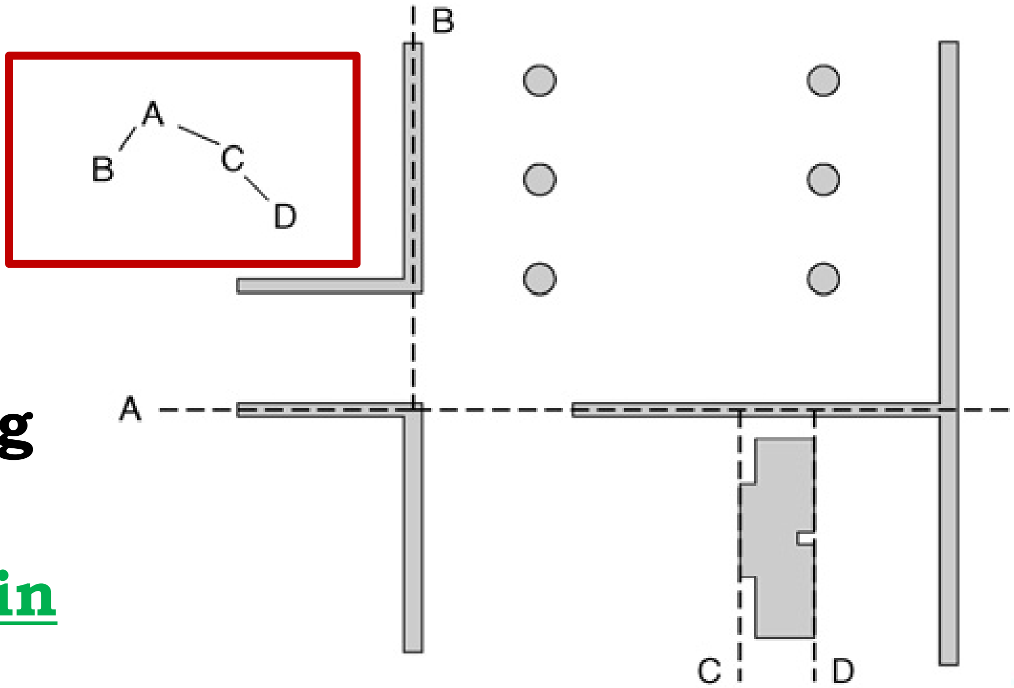
- We could use *a similar approach for entire levels in a game*, …

Spatial Partitioning

- but *for entire levels of a game*, spatial partitioning is a more popular approach.
- There a few differences between BVHs and spatial partitioning:
- BVH: hierarchy depends on *relative positions of objects* represented. Thus hierarchy can change.
- Spatial partitioning: locked to the world. An object in a specific position will be mapped to one location in the data structure.
- Sometimes a combination of both will be used (very commonly, single level bounding boxes will be used even if a BVH is not used).

Binary Space Partitioning

- Again, a binary tree data structure.
- At each node **defines a different plane**. The node has two children, one for each side of the plane.
- *Objects on one side of the plane end up in the subtree* represented by the corresponding child.
- <u>Leaf nodes contain a set of objects</u> (perhaps as a BVH).



Binary Space Partitioning

- BSP trees (and the other spatial partitioning data structures that we will examine) ***have the following issue***:

  How to deal with objects that cross the plane?

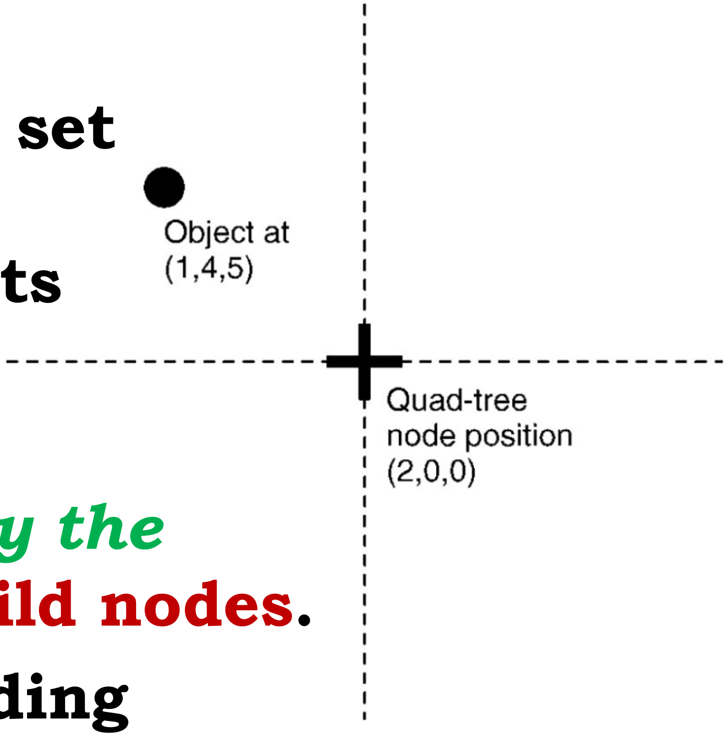
- Some common approaches (each gives a different data structure):
  - they ***can be directly attached to that parent node***; or
  - placed in the child node that they are nearest to; or,
  - more commonly, placed in both child nodes.

Binary Space Partitioning

- Let’s assume we have a BSP tree where objects that intersect a plane are placed (entirely) in both child nodes.
- **Broad Phase Collision Detection**
  - The only collisions that can possibly occur are between ***objects at the same leaf*** in the tree.
  - We can simply consider each leaf of the tree in turn.
  - All pair combinations of those objects at a leaf can be ***sent to the fine collision detector*** for detailed checking.

Quad and Oct Trees

- **Quad-trees** are used for 2D (or 2½D where most objects will be stuck on the ground), and oct-trees for 3D.
- A quad-tree is made up of a set of nodes, ***each with four descendants***. The node splits space into four areas that intersect at a single point.



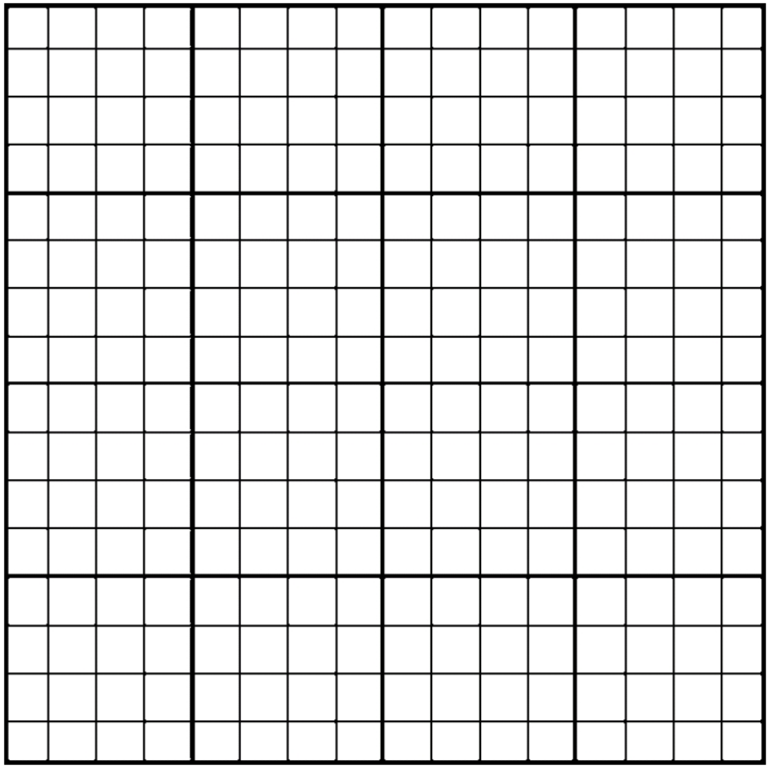
- An ***oct-tree works in exactly the same*** way, but **has eight child nodes**.
- Similar issue as BSPs regarding objects that cross the dividing lines/planes.

Quad and Oct Trees

- Commonly, *each node will be centred in the axis-aligned bounding box* it represents. Then each node creates four (or eight) more boxes of the same size. And so on down the hierarchy.
- **Advantages**:
  - Can *calculate the point on the fly* during recursion down the tree. This saves memory.
  - *Don't need to perform any calculations to find the best location* to place each node's split point. This makes it much faster to build the initial hierarchy.
- **Broad Phase Collision Detection**: As with BSPs

Grids

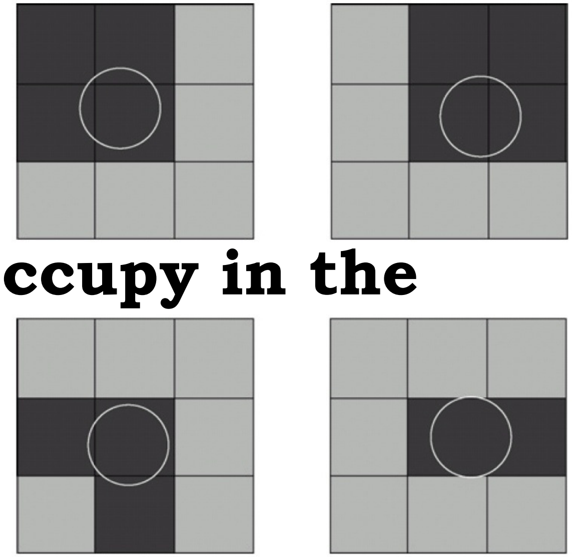
- A grid is an array of locations in which there may be any number of objects.
- Not a tree data structure.
- Much faster to find where an object is located than recursing down a tree.
- Each square in the grid contains *a list of all the objects contained in it*.
- We maintain *a list of all squares containing more than one object*.



Grids

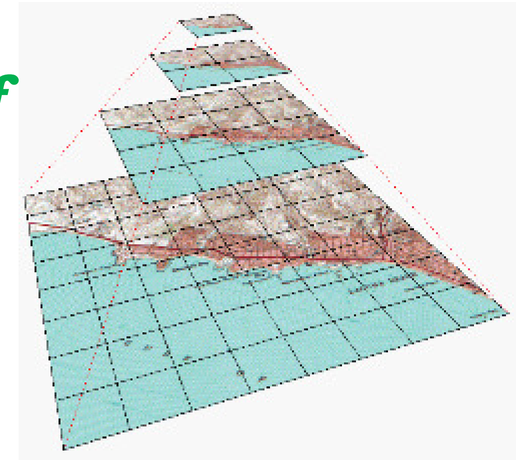
- For objects that *overlap the edge of a square*:
  - It is most common to simply place them into one cell or the other;
  - or place them into all the cells they overlap.
- Just as before, the latter makes it faster to determine the set of possible collisions, but can take up considerably more memory
- Broad Phase Collision Detection
  - Using the latter, the set of collisions can be generated very simply. Two objects <u>can only be in collision</u> if they occupy the same location in the grid.

Grids



- *If objects are larger* than the size of a cell, they will need to occupy in the grid. This can lead to *very large memory requirements* with lots of wasted space.
- If we place an object in just one grid cell (the cell in which its center is located, normally), then *the coarse phase* collision detection *needs to check for collisions with objects in neighboring cells*.
- A hybrid data structure can be useful in this situation, using multiple grids of different sizes. It is normally called a "multi-resolution map".

Multi-Resolution Maps



- A **Multi-Resolution Map** is ***a set of grids with increasing cell sizes***.
- **Objects are added into one of the grids only, in the same way as for a single grid. The grid is selected based on the size of the object.**
- **Often the grids are selected so that each one has cells four times the size of the previous one.**
- **Broad Phase Collision Detection**
  - **For each grid, collisions between each object and objects in the same or neighboring cells are checked. Also, *the object is checked against all objects in all cells in larger-celled grids that overlap*.**

(§13)

Generating Contacts

Generating Contacts

- The broad phase ***produces a list of object pairs that then needs to be checked in more detail*** to see whether the pairs do in fact collide.
- These pairs are ***passed to the fine phase where we do contact generation***: finding all points of contact between colliding objects.
- Often we will have **a two-stage process** of contact generation:
  1. **a fine collision detection** step ***to determine whether there are contacts to generate***
  2. **a contact generation** step ***to work out the contacts that are present***

Collision Geometry

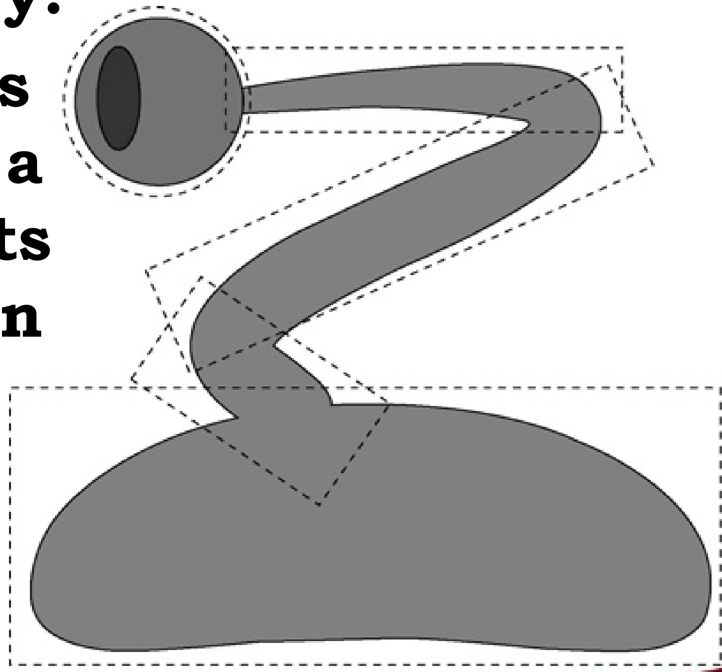
- **Collision Geometry**: ***chunky geometry created just for the physics***.
- **If this chunky geometry consists of certain geometric primitives—namely, spheres, boxes, planes, and capsules (a cylinder with hemispherical ends)—then *the collision detection algorithms can be simpler than for general-purpose meshes*.**
- **This collision geometry isn't the same as the bounding volumes used in coarse collision detection.**
  - **There may be *many different levels of simplified geometry for a scene*.**

Collision Geometry

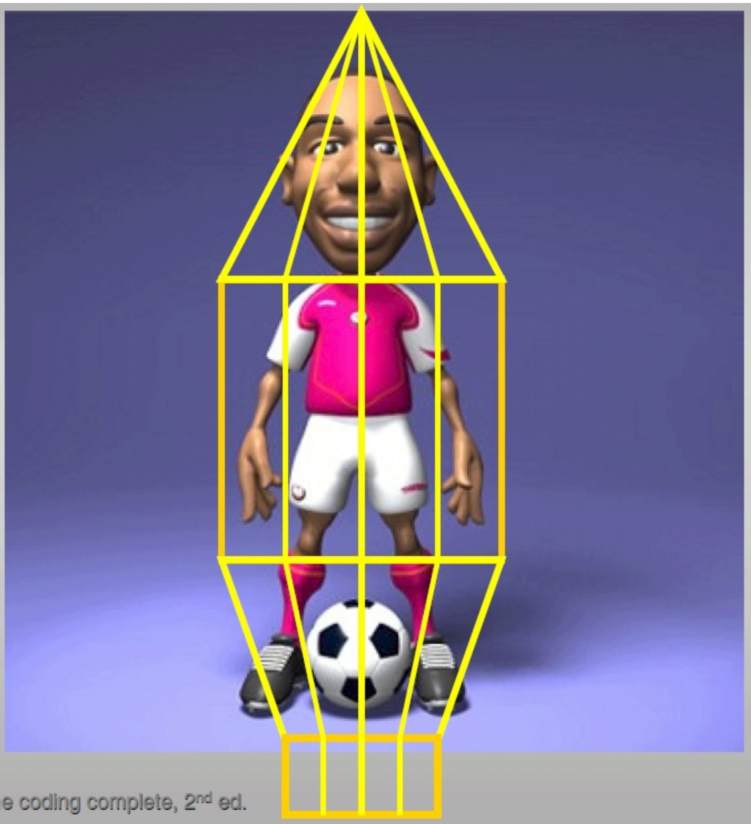
- A special case we need to consider is the *collision of objects with the background level geometry*.
  - The ground or walls, typically represented by planes.
- The primitives your game needs will depend to some extent on the game. We'll only look in detail at spheres and boxes.
- But primitives only get you so far. All primitives can only fit their objects roughly; *there are some objects that don't lend themselves well to fitting with primitives*.

Primitive Assemblies

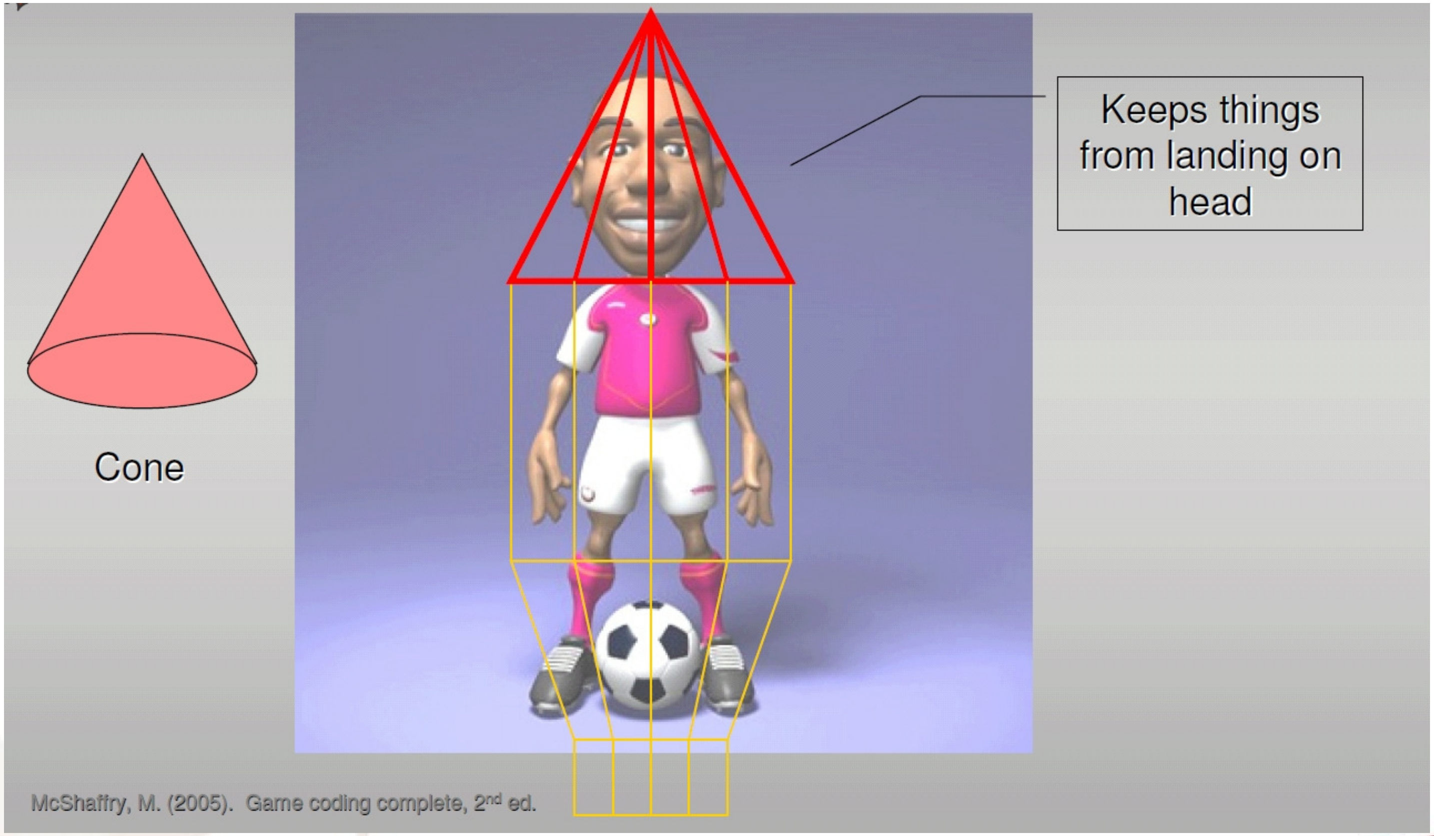
- The vast majority of objects can't easily be approximated by a single primitive shape.
- One approach is to use assemblies of primitive objects as collision geometry.
- We can represent assemblies as *a list of primitives*, with a transform matrix that offsets the primitive from the origin of the object.
- Typically, this collision geometry *is created manually by designers*.



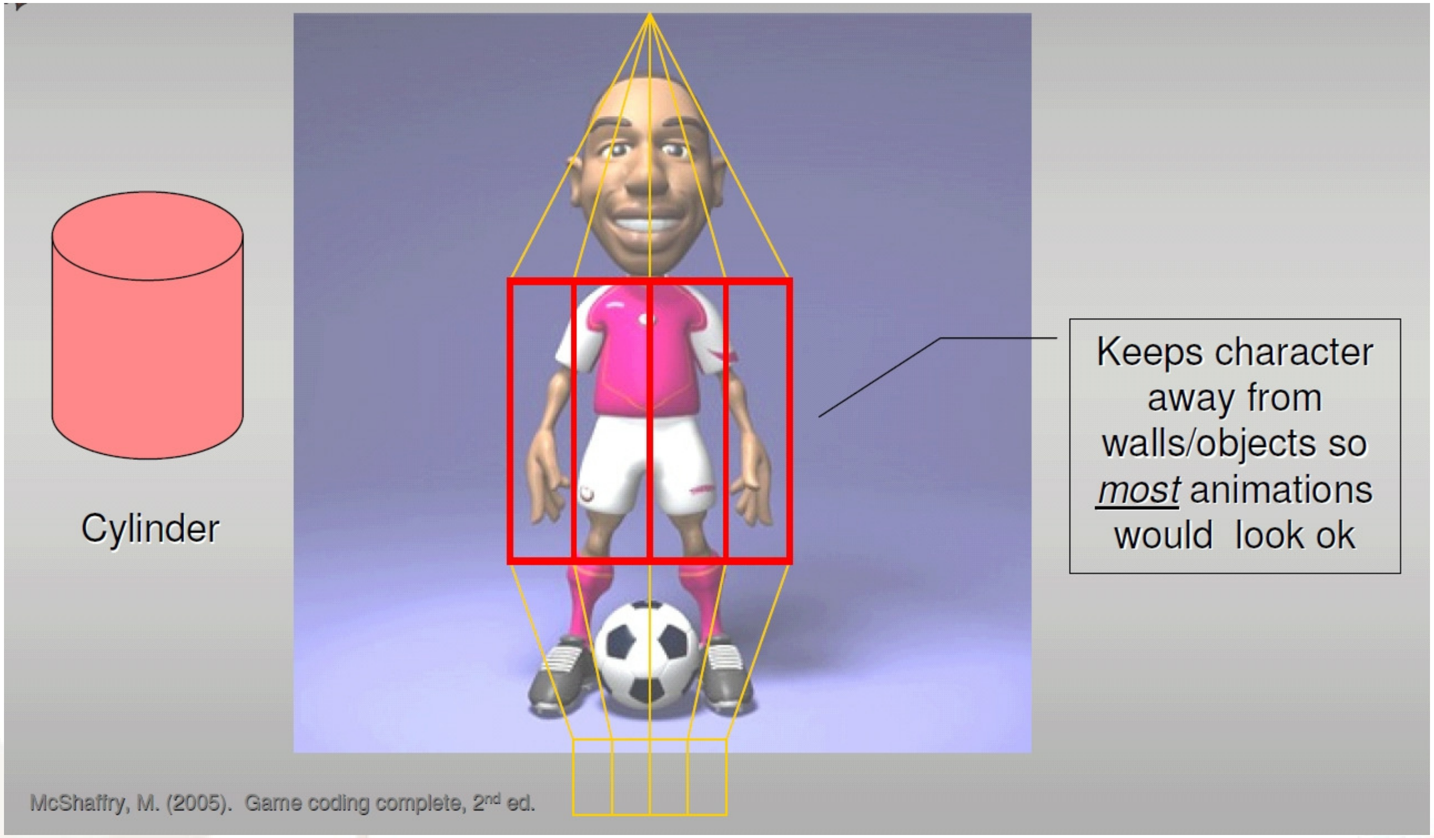
Human Collision Geometry


McShaffry, M. (2005). Game coding complete, 2nd ed.

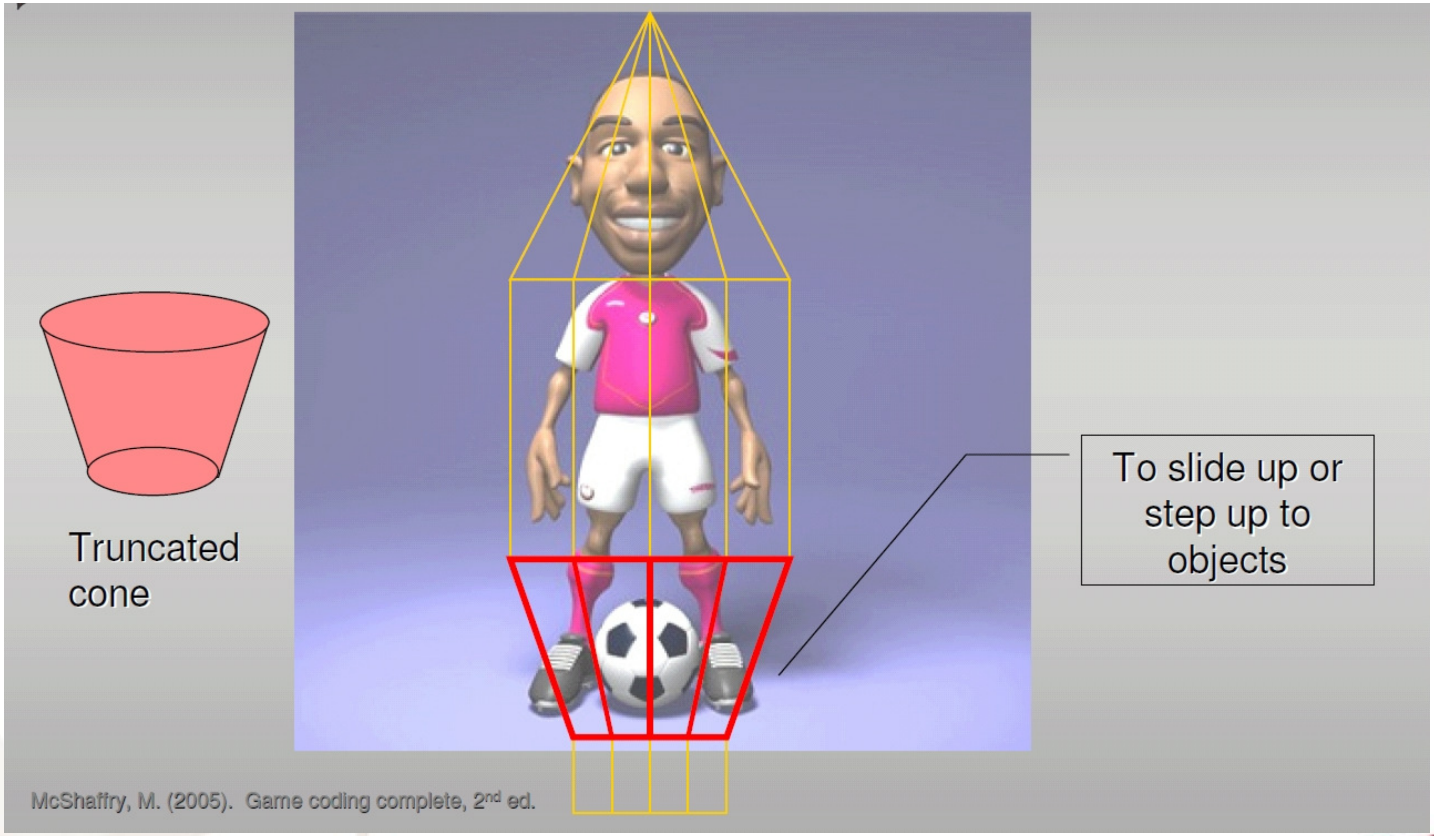
Human Collision Geometry



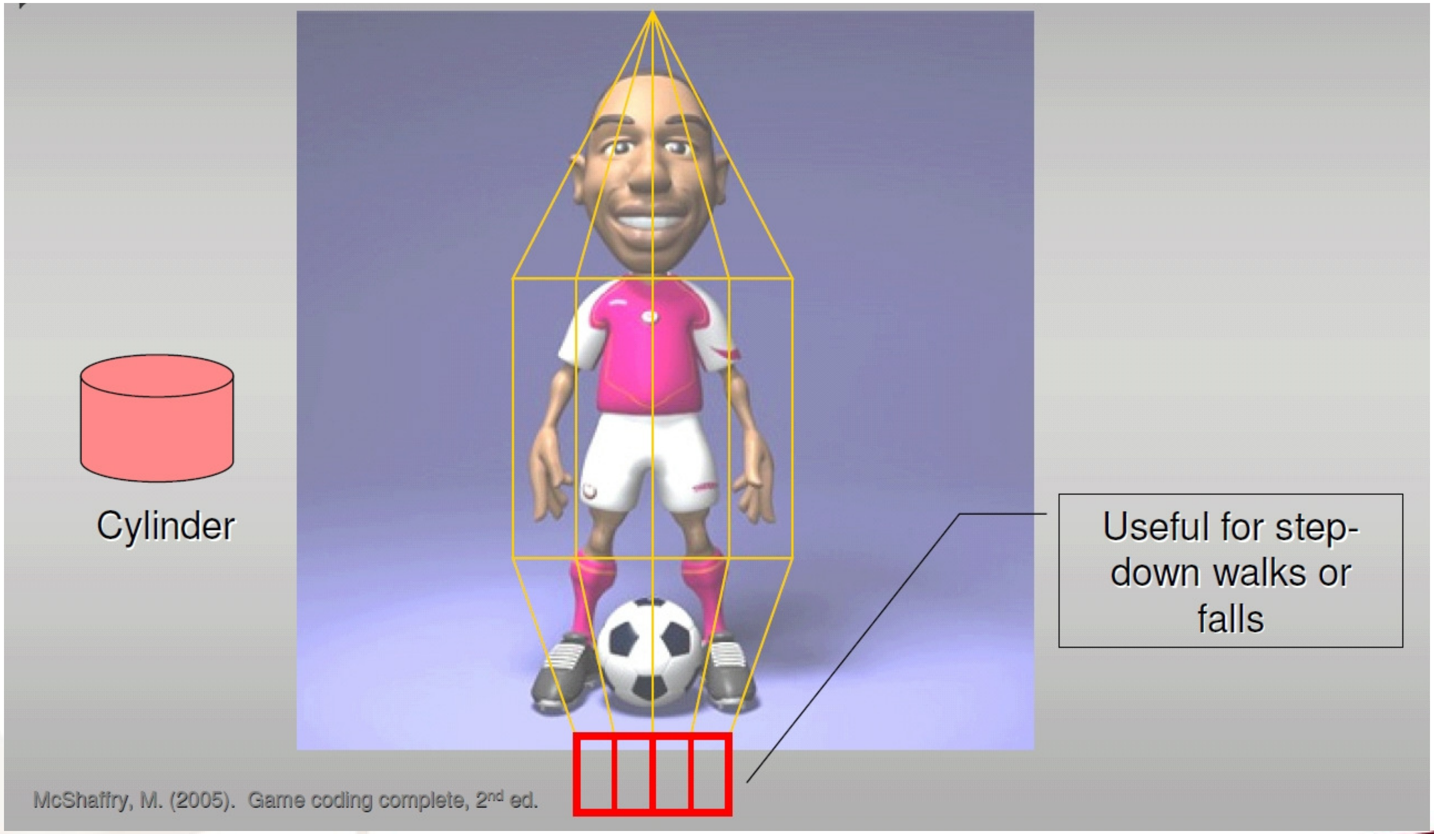
Human Collision Geometry



Human Collision Geometry

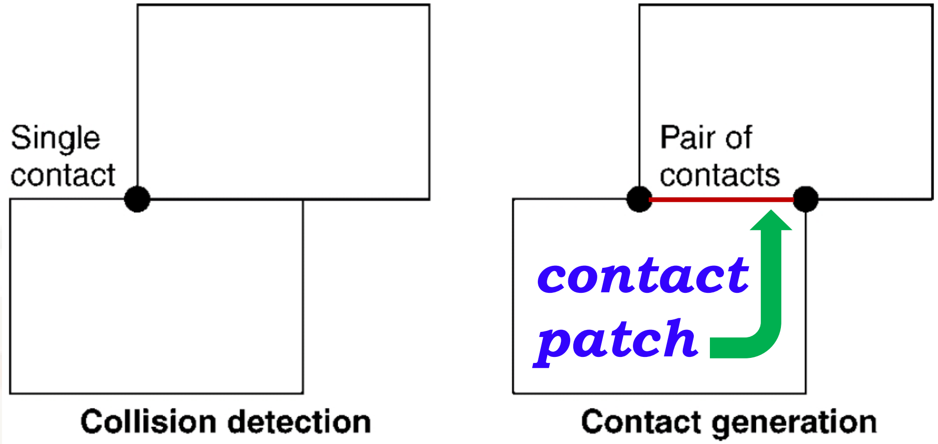


Human Collision Geometry



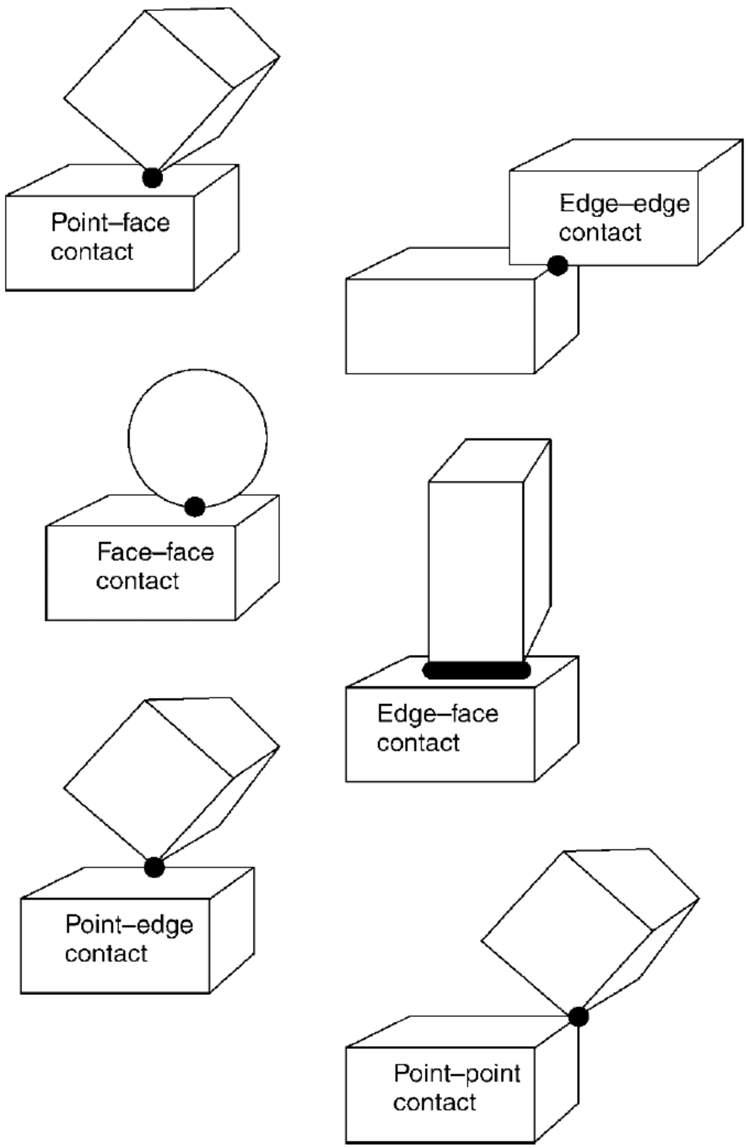
Contact Generation

- Note the following distinction.
- **Collision detection**: determines *whether two objects are touching or interpenetrated*, and normally *provides data on the largest interpenetration point*.
- **Contact generation**: *produces the set of contact points* on each object that **are in contact** (or **penetrating**).



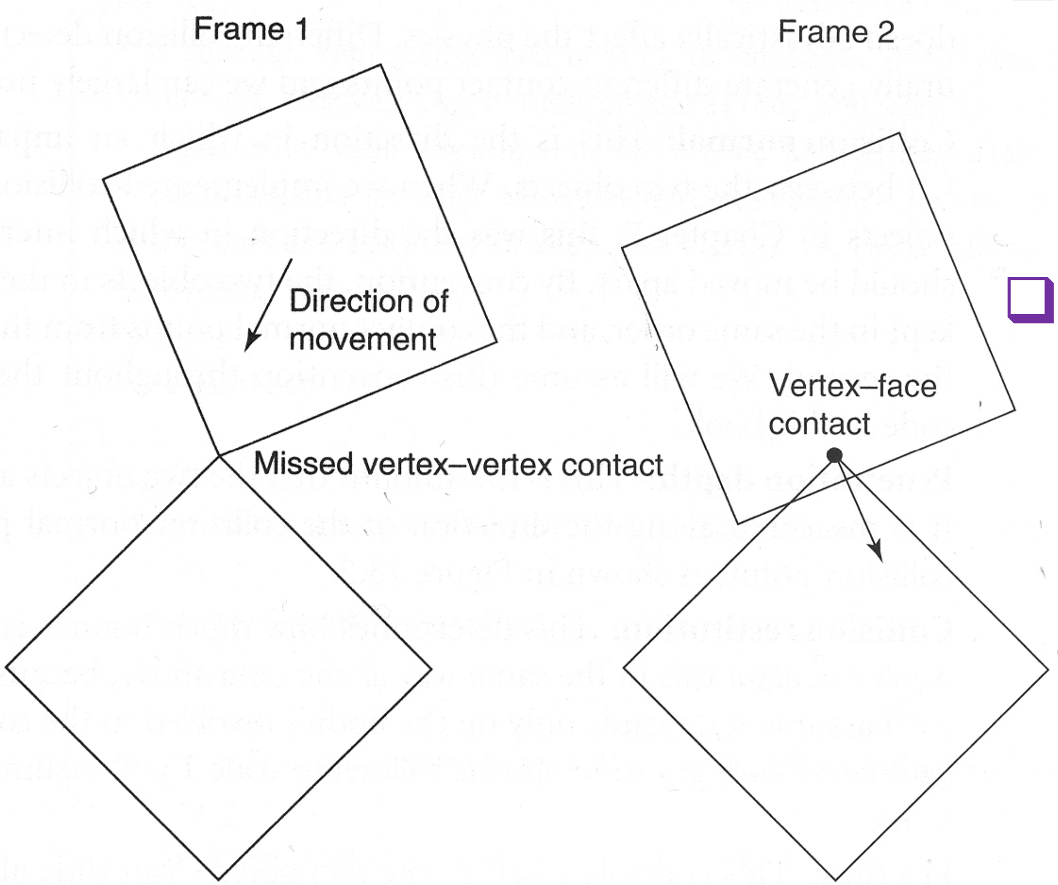
- For physics applications, we need contact generation.

Contact Situations



- **We deal with a set of contact situations as shown.**
- **The simplifications in the figure *generate reasonable physical behaviour*.**
- **These cases are arranged in order of how useful they are (return all at same level).**

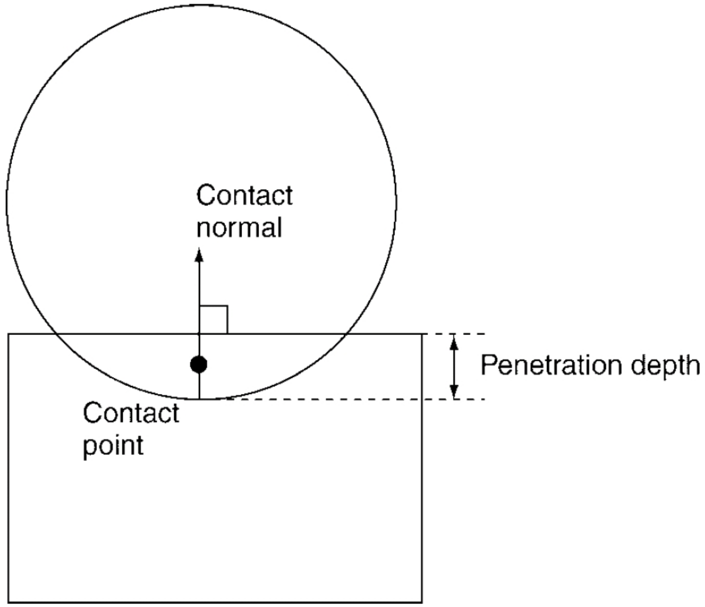
Further Optimization



- **Some contacts are rare and typically difficult to generate good contact data.**
- **So, in the priority orders given above, it is normal to ignore the contacts of the lowest priority group: vertex-vertex, and vertex-edge and parallel edge-edge contacts.**

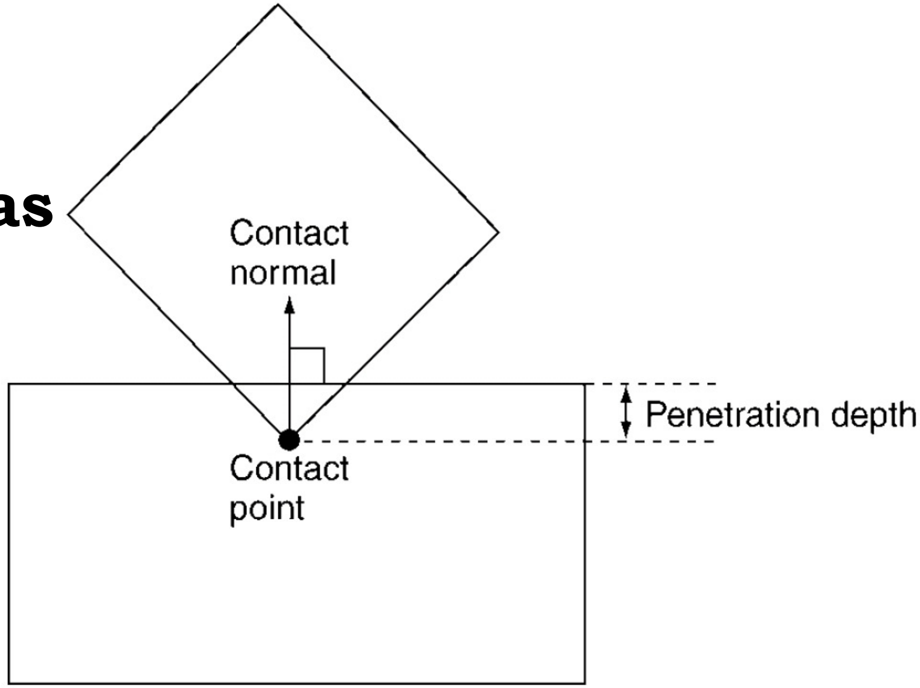
Contact Data

- Several pieces of data used to resolve each contact
- **Collision Point**: point of contact between objects.
- **Collision Normal**: direction that an impulse impact will be felt between two objects.
- **Penetration Depth**: amount two objects are interpenetrating in direction of collision normal.
- **Collision Restitution**: how much "bounce" is in the collision; assumed given.
- **Friction**: whether two objects can slide along the contact; assumed given.



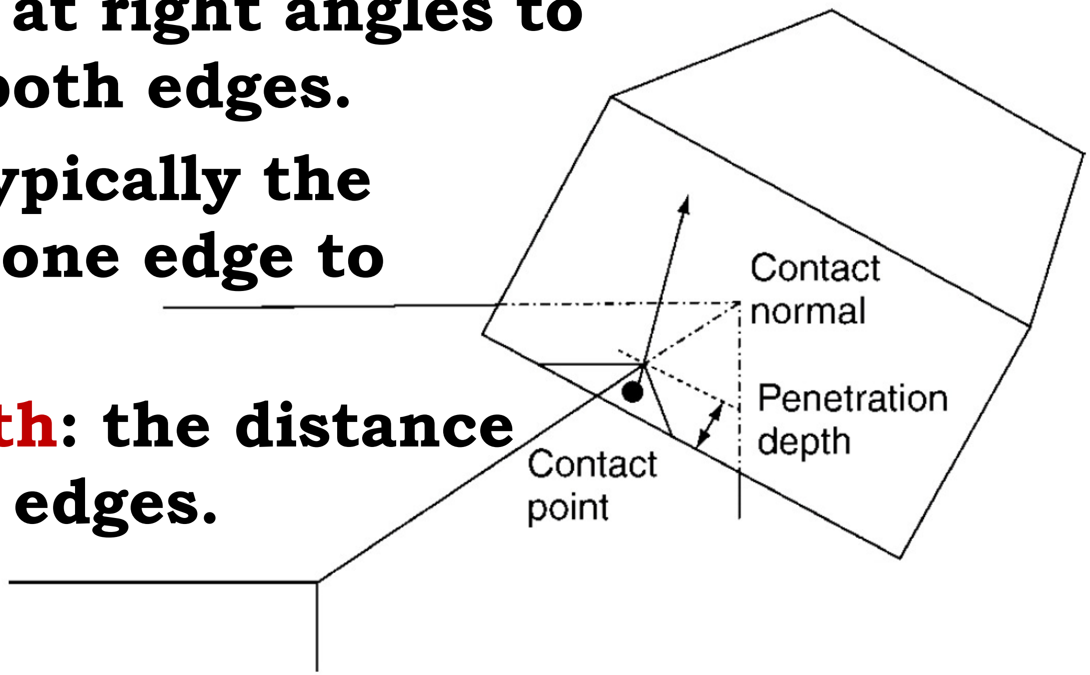
Point-Face Contacts

- One of the two most common and *important types of contact*. Face can be curved or flat.
- Contact Normal: given by the normal of the surface at the point of contact.
- Contact Point: given as the point involved in the contact.
- Penetration Depth: is calculated as the *distance between the object point and the projected point*.



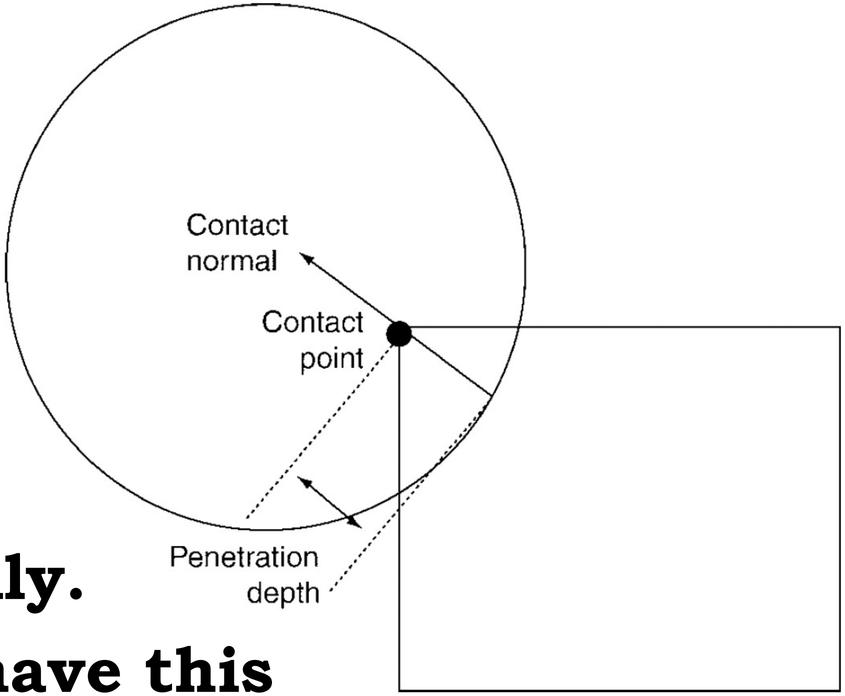
Edge-Edge Contacts

- Second most important type. Critical for resting contacts between objects with flat or concave sides.
- Contact Normal: at right angles to the tangents of both edges.
- Contact Point: typically the closest point on one edge to the other.
- Penetration Depth: the distance between the two edges.



Edge-Face Contacts

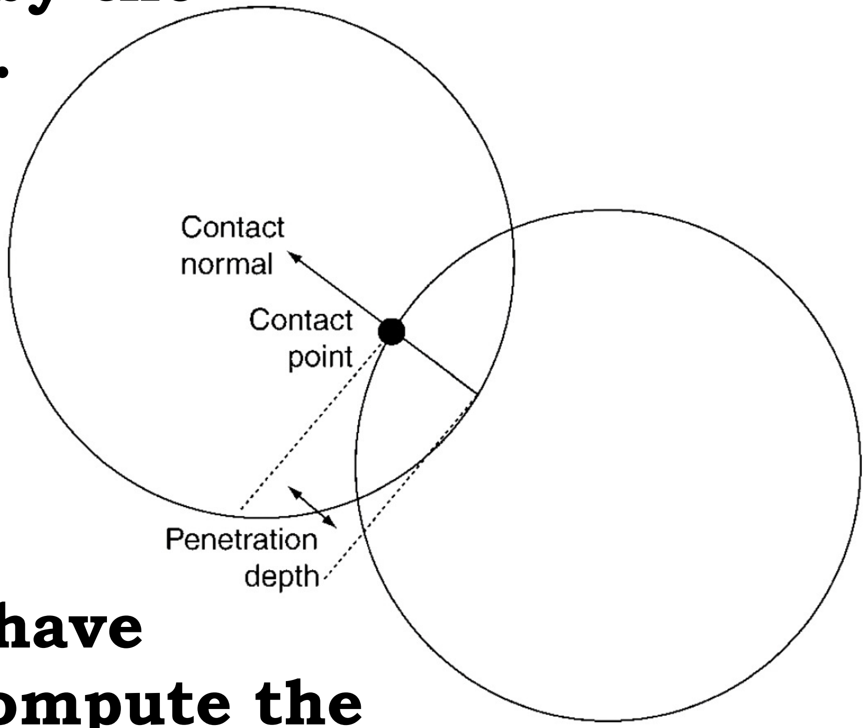
- Only used with curved surfaces.
- **Contact Normal**: given by normal of the face, as before. Edge direction is ignored in this calculation.
- **Contact Point**: is more difficult to calculate for the general case. In the more general case we need to calculate the point of deepest penetration geometrically.



- **Penetration Depth**: we have this from the way we compute the contact point

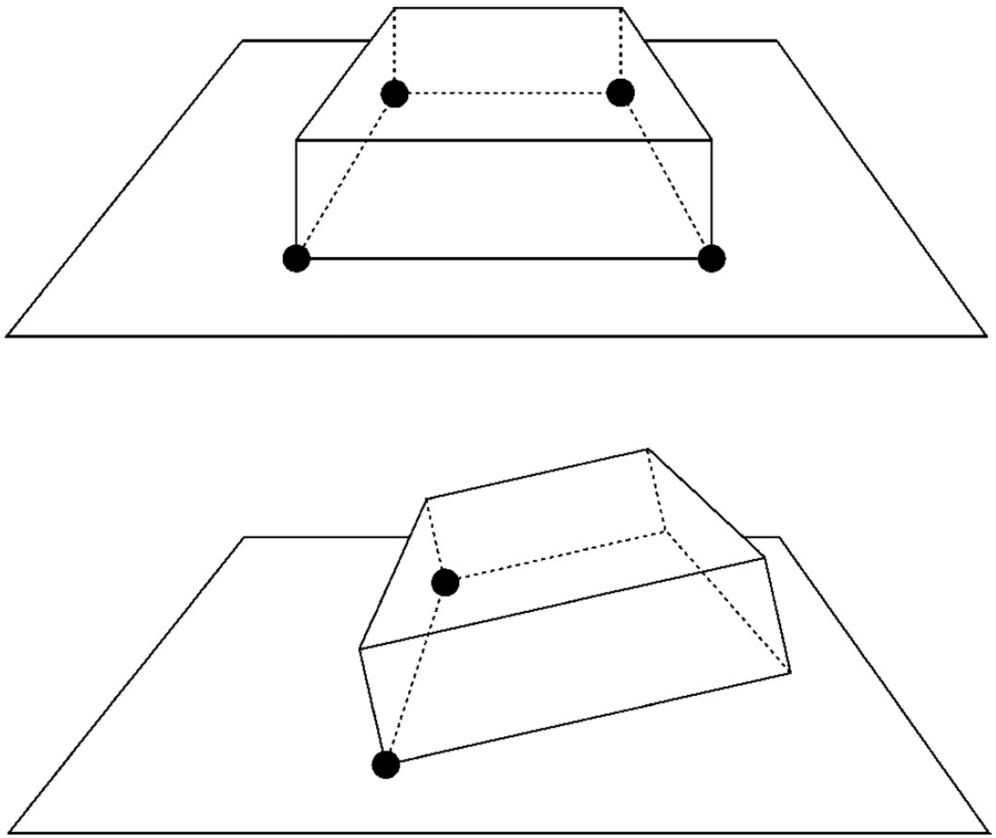
Face-Face Contacts

- Occurs when *a curved surface comes in contact with another face*, either *curved or flat*.
- **Contact Normal**: given by the normal of the first face.
- **Contact Point**: difficult to calculate in the general case. The primitives used often give point of greatest penetration.
- **Penetration Depth**: we have this from the way we compute the contact point



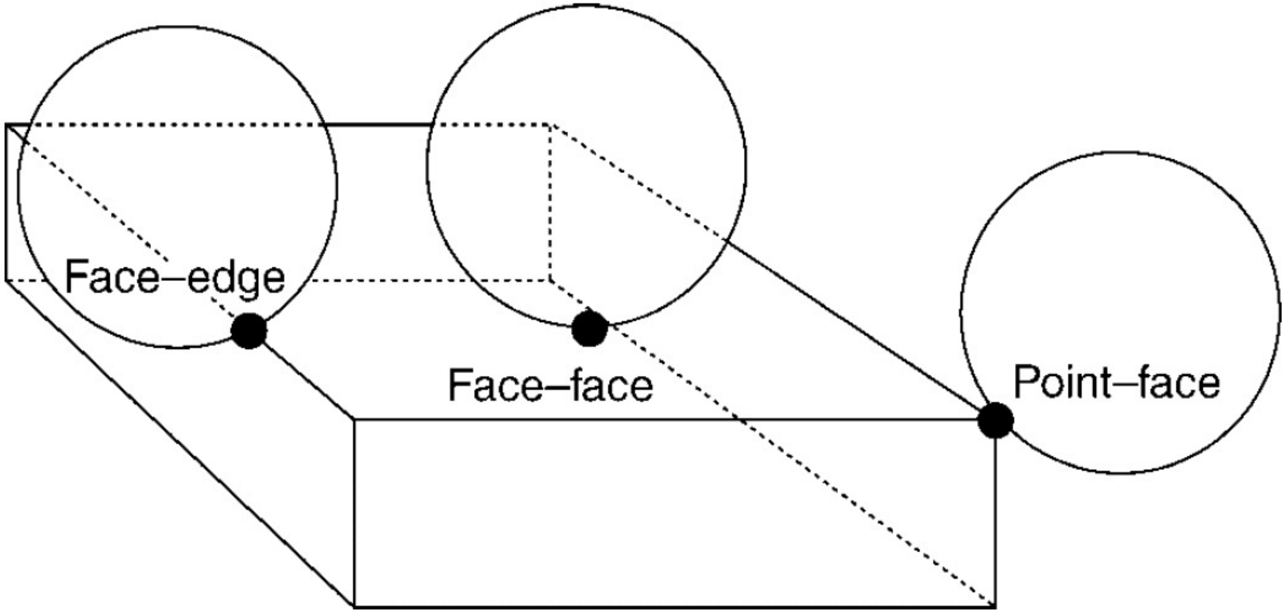
Box-Plane Collision

- **This is the *first algorithm that can return more than one contact*.**



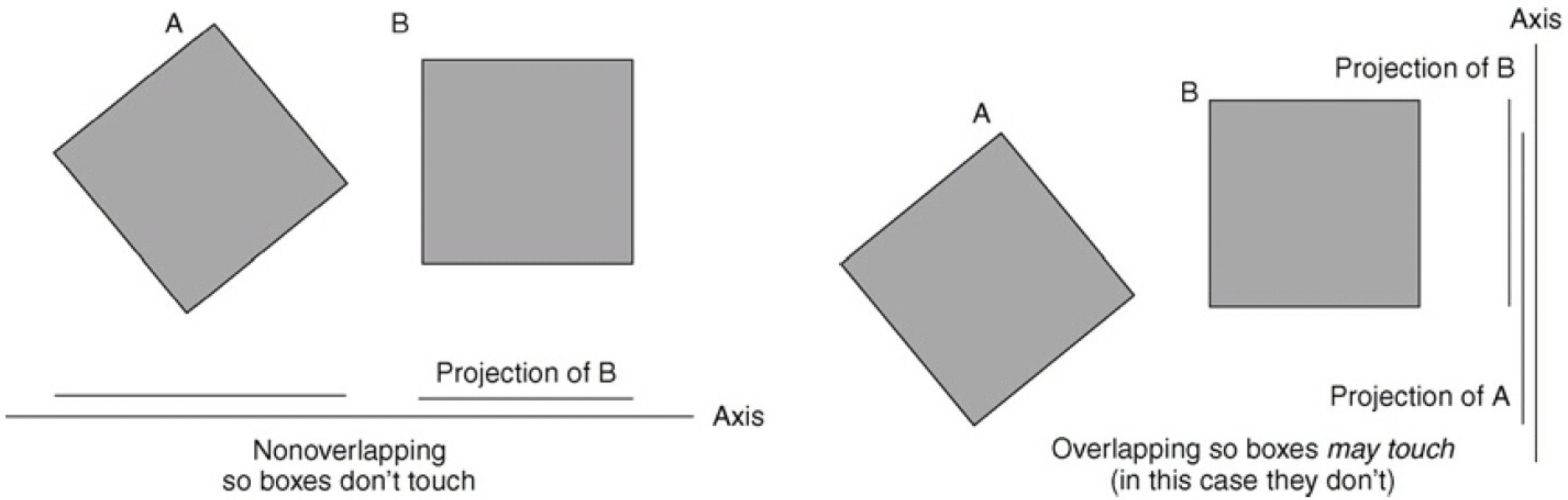
Box-Sphere Collision

- **When *a sphere collides with a box*, we will have <u>just one contact</u>.**
- **But it <u>may be a contact of any type</u>: a face–face contact, an edge–face contact, or a point–face contact**



Separating Axis Tests (SATs)

- **Idea**: ***if we can find any direction in space in which two (convex) objects are not colliding***, **then the two objects are not colliding at all**.



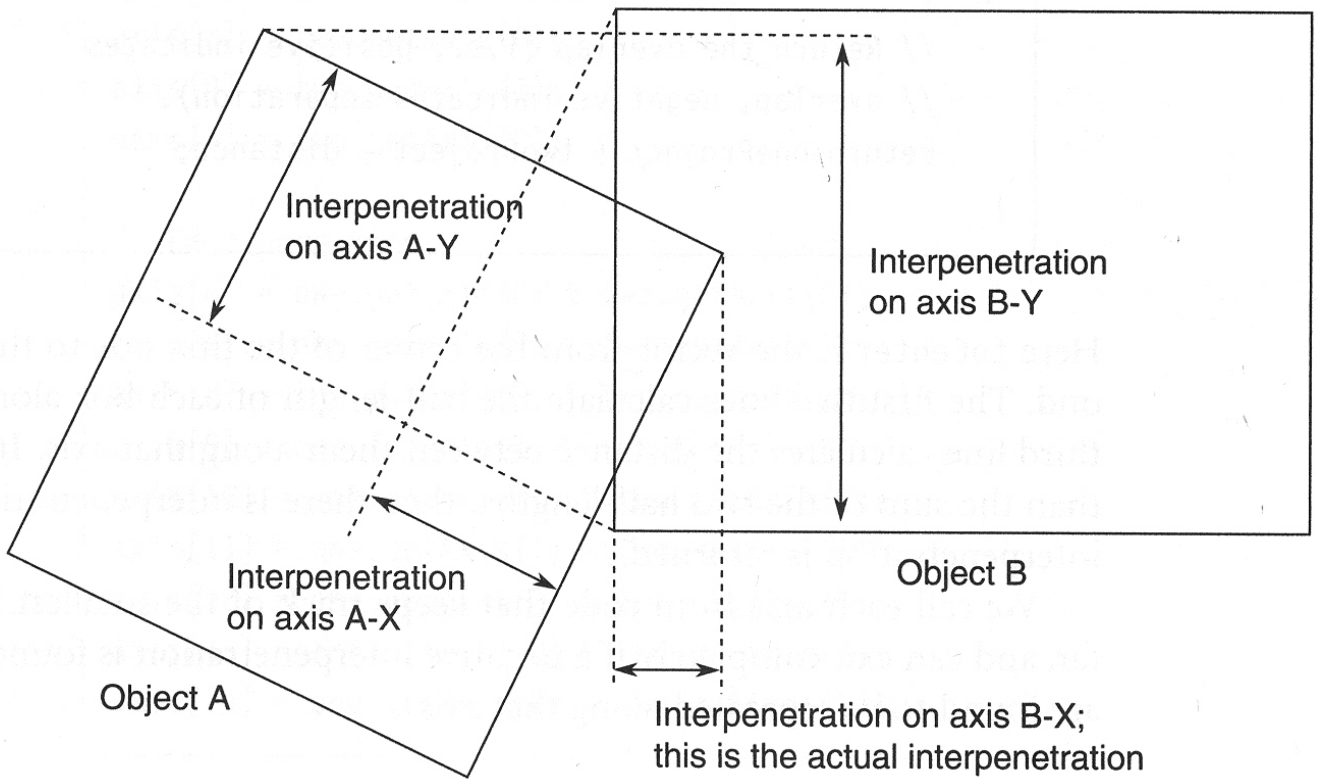
Separating Axis Tests (SATs)

- The set of axes needed are:
  - All faces on both objects give rise to *an SAT axis equal to their face normal*
  - All pairs of edges on different objects create *an SAT axis that is at right angles to both edges*
- For a pair of boxes this gives 15 axes:
  - **3 principal axes** of each box +
  - *9 axes that are perpendicular to each pair of principal axes* from each box (taking the cross product of each pair of principal axes)

Université Concordia University

Separating Axis Tests

- **SATs can tell us *where objects are overlapping and the maximum depth of interpenetration*.**



References/Resources

- Game Physics Engine Development, 2nd Ed., by Ian Millington. [text]
- Chapter 15 Collision and Simple Physics of Game Coding Complete, 3rd Ed., by Mike McShaffry (2009) [text]
- Physics for Games, IMGD 4000, WPI. [PPT]