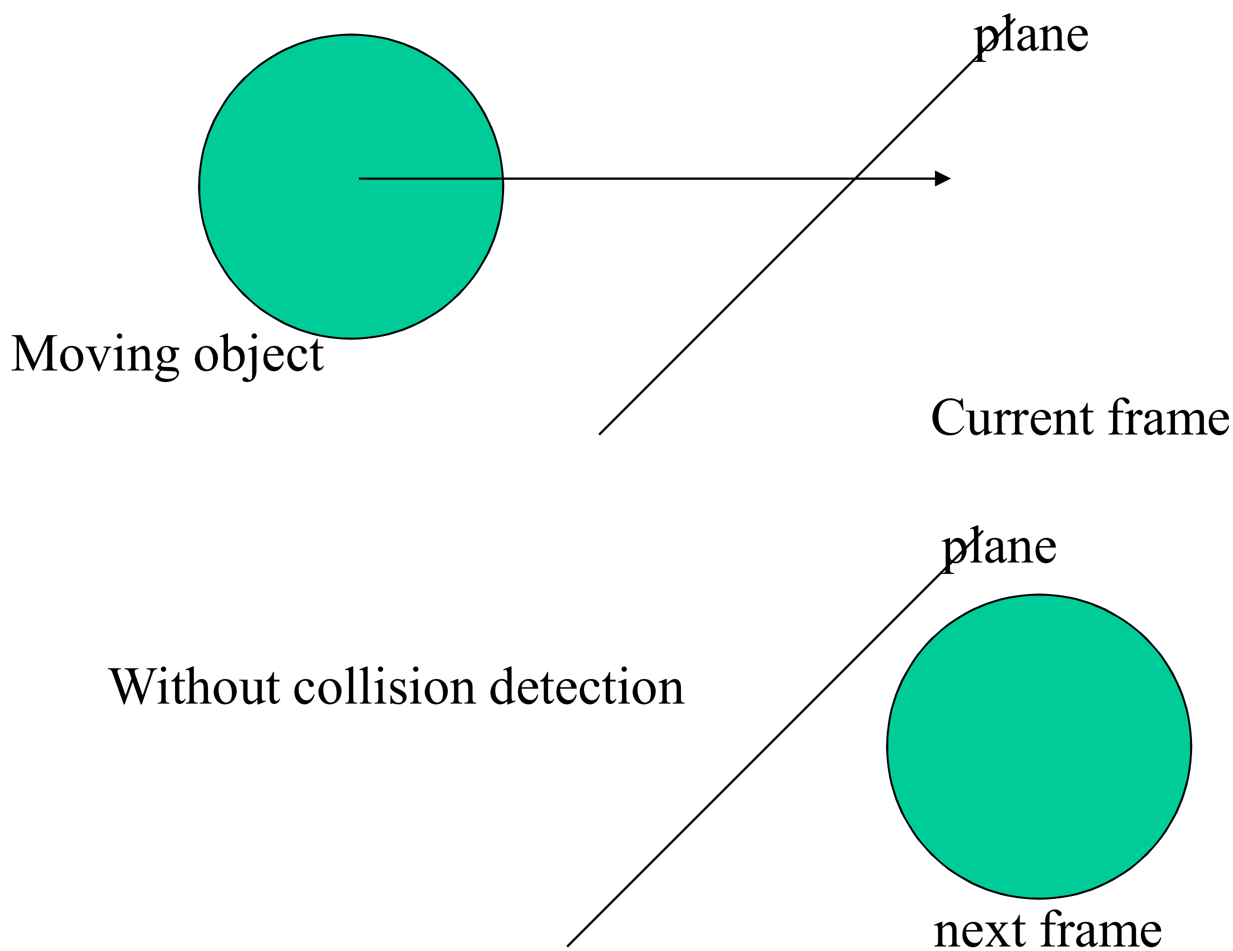
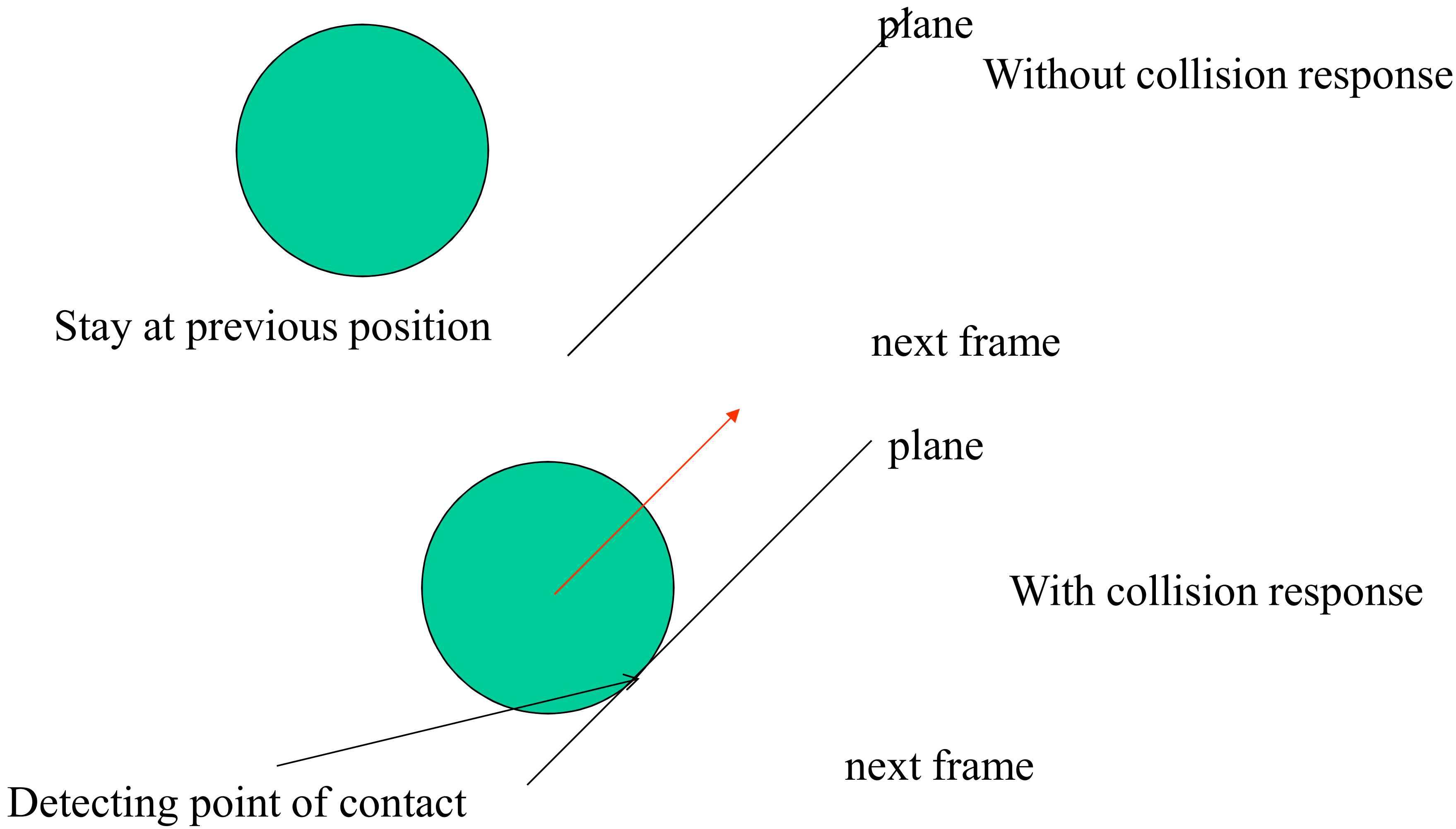
Principle of Computer Game Software

Collision Detection & Physics

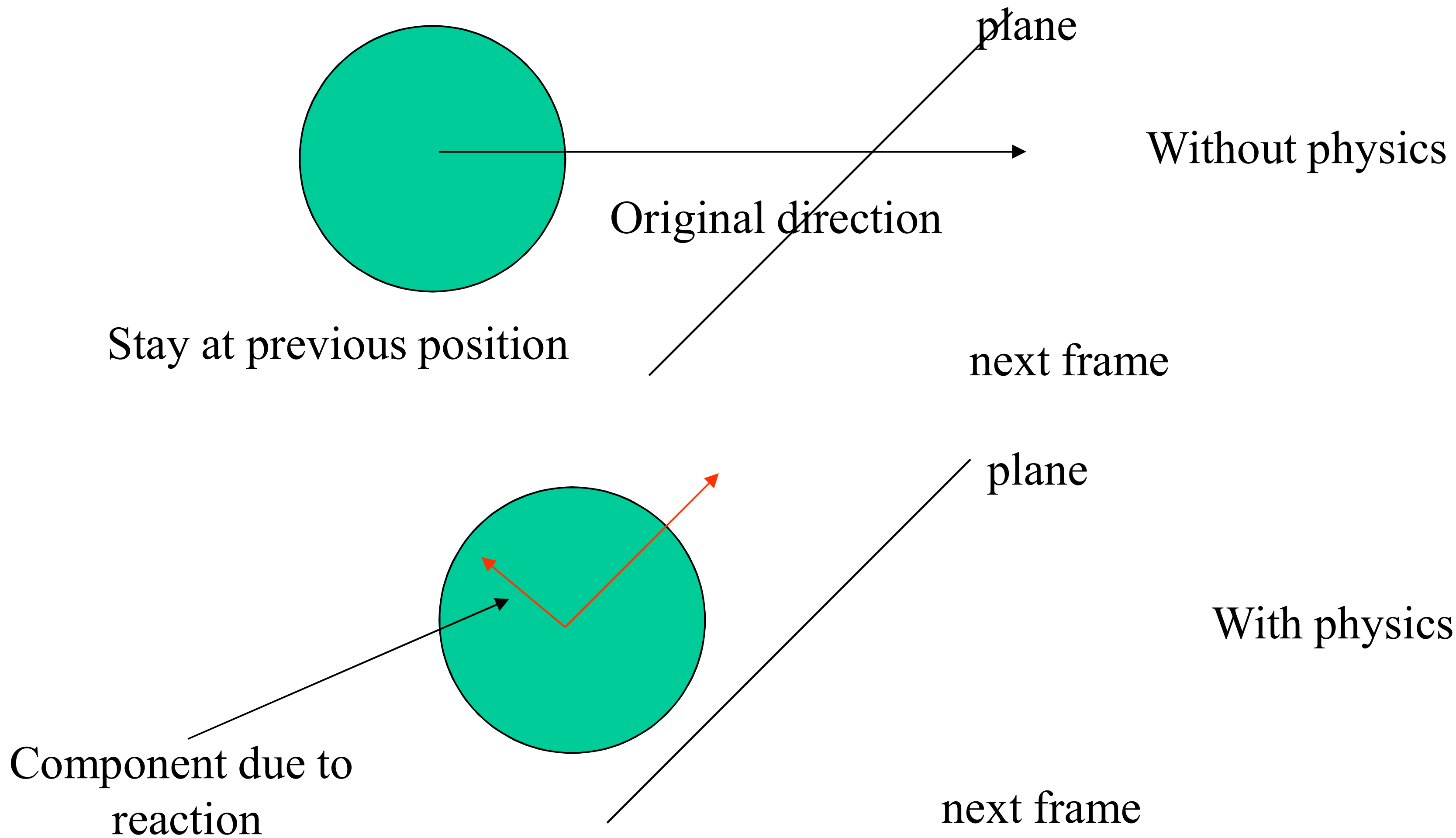
Collision Detection



Collision response



Physics

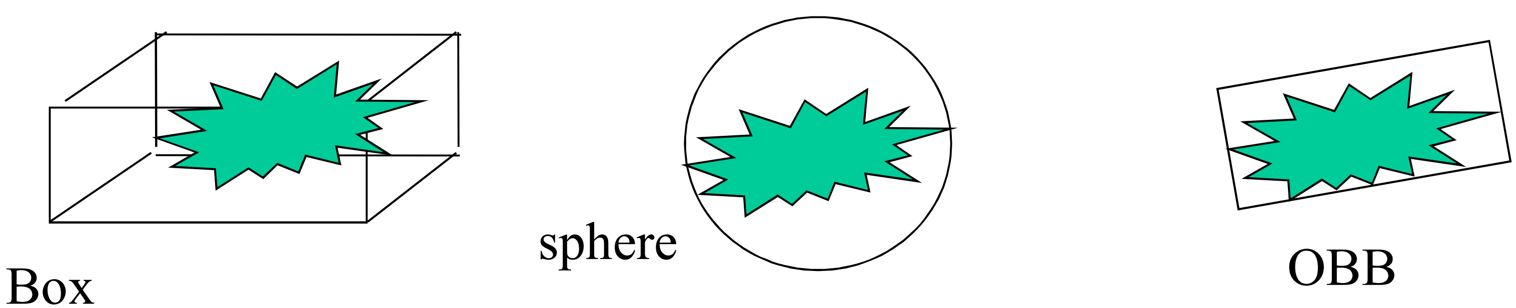

plane
Without physics
Original direction
Stay at previous position
next frame
plane
With physics
Component due to
reaction
next frame

Simple Taxonomy(collision detection)

- Problem faced
  - tons of objects in game world, $O(N^2)$ complexity in detection
  - Same problem in visibility processing!
- Taxonomy
  - Broad phase/narrow phase algorithms
  - Single phase methods
  - Strategies

Broad phase

- Cull away pairs of objects that cannot possibly collide by
  1. Game rules or context
  2. Spatial partitioning in world/local space
  3. Bounding volumes (spheres, AABBs and OBBs) and bounding volume hierarchies



Narrow Phase

- Accurate collision detection by

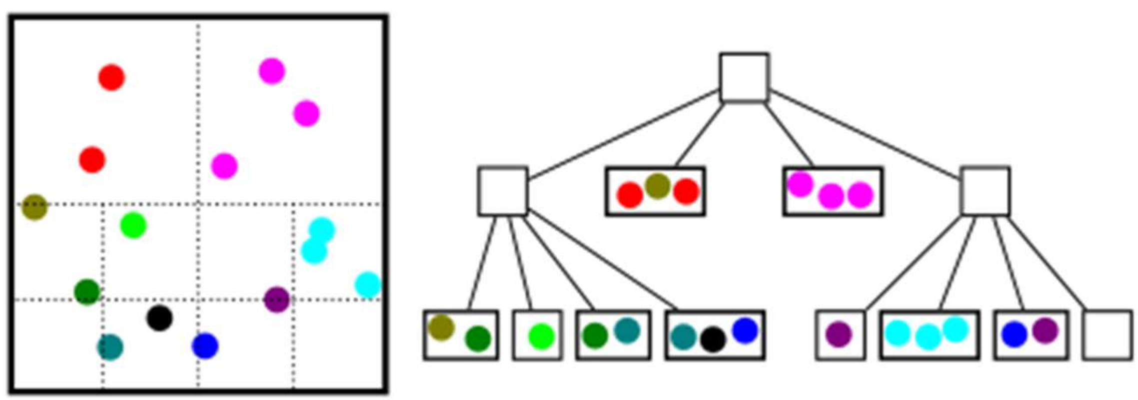
1. Polyhedron-polyhedron testing
2. Separating planes

Strategies

- Exploit spatial/temporal coherence – objects tend to occupy the same region of space from one frame to another
- Pre-calculation – spatial partitioning and bounding volumes involve some degree of pre-calculation e.g. BSP tree

Octree in Collision Detection

- To check for potential colliding pairs, tree is descended and only those regions containing more than one object are examined
- Eliminate testing pairs that distant away
- Drawback : octree must be updated each time step



Geometric Algorithms

- Collision detection & response requires geometrical tests
- Primitives tests involved – point, triangle, sphere, object ..
- Depends on the abstraction of the programmer & speed requirement

Bounding sphere

- Clipping against a plane

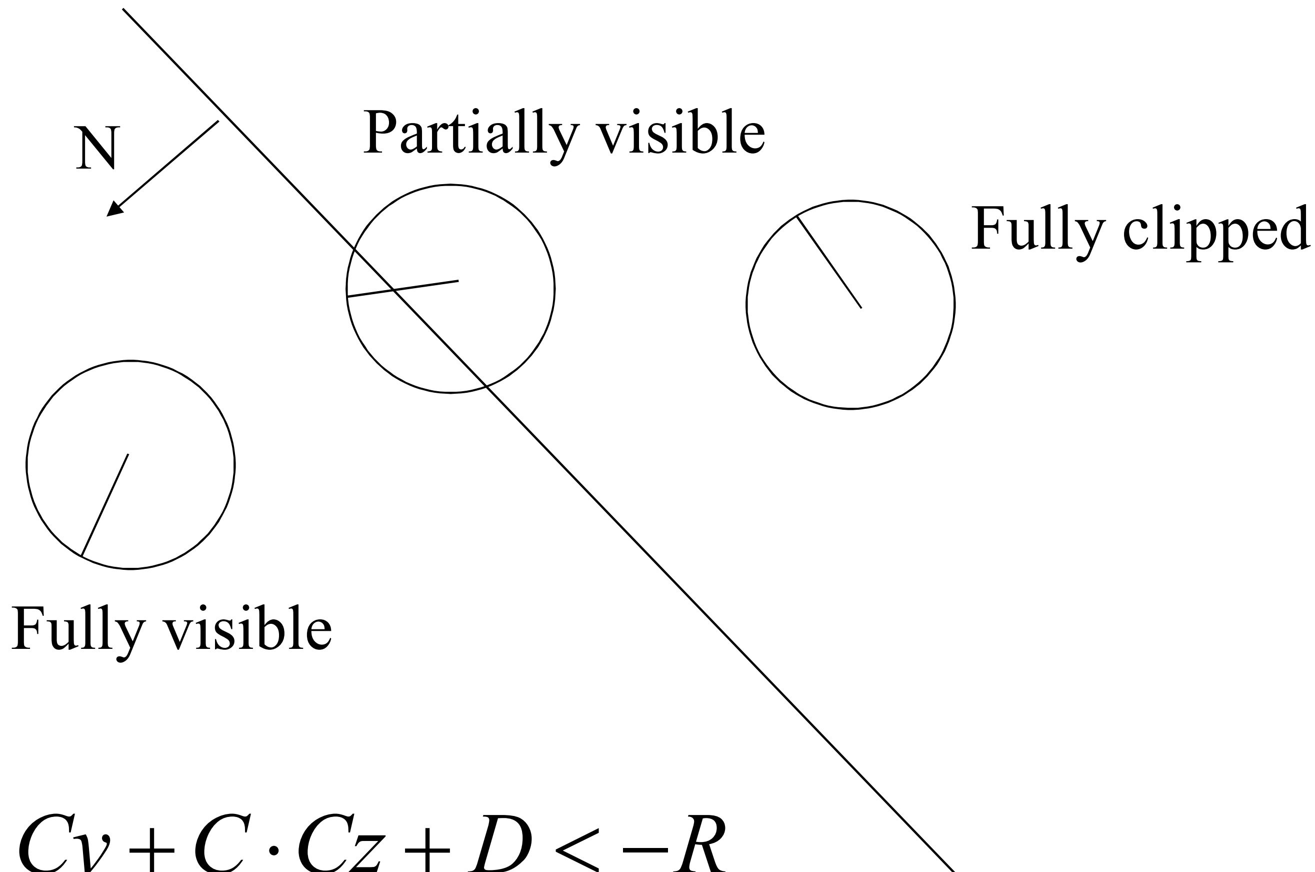
$$Ax + By + Cz + D = 0$$

- For a sphere with center (Cx,Cy,Cz) and radius R,

$$A \cdot Cx + B \cdot Cy + C \cdot Cz + D < -R$$

- Returns *true* if sphere lies completely in hemispace opposite the plane normal

Bounding sphere



$$A \cdot Cx + B \cdot Cy + C \cdot Cz + D < -R$$

Only 3 multiply,4 add and 1 compare

Bounding sphere

- Same as test for a point against the plane

$$A \cdot Px + B \cdot Py + C \cdot Pz + D < 0$$

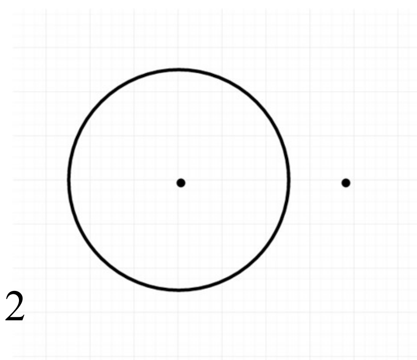
- Implement as clipping against viewing frustum

```
for each clipping plane
     if sphere outside plane
          return false
end for
return true
```

Point-in-Sphere

- Given a point P and a sphere with equation

$$(X - Xc)^2 + (Y - Yc)^2 + (Z - Zc)^2 = R^2$$



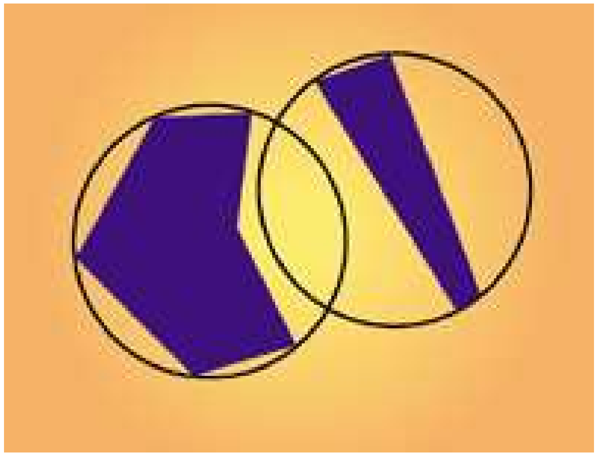
- P is inside sphere iff

$$(Xp - Xc)^2 + (Yp - Yc)^2 + (Zp - Zc)^2 < R^2$$

- Square of radius stored to save computation

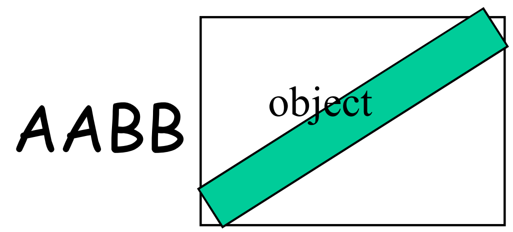
Bounding sphere

- Advantages
  - Cheap in operational cost
  - Rotational invariance
- Disadvantage
  - Not suitable for objects not accordance with sphere e.g. rod shape object will generates a lot of false positives



Bounding Box

- Can provide tighter fit
- Generic or *Axis Aligned*(AABB) – faces parallel to X, Y and Z axes
- To generate AABB



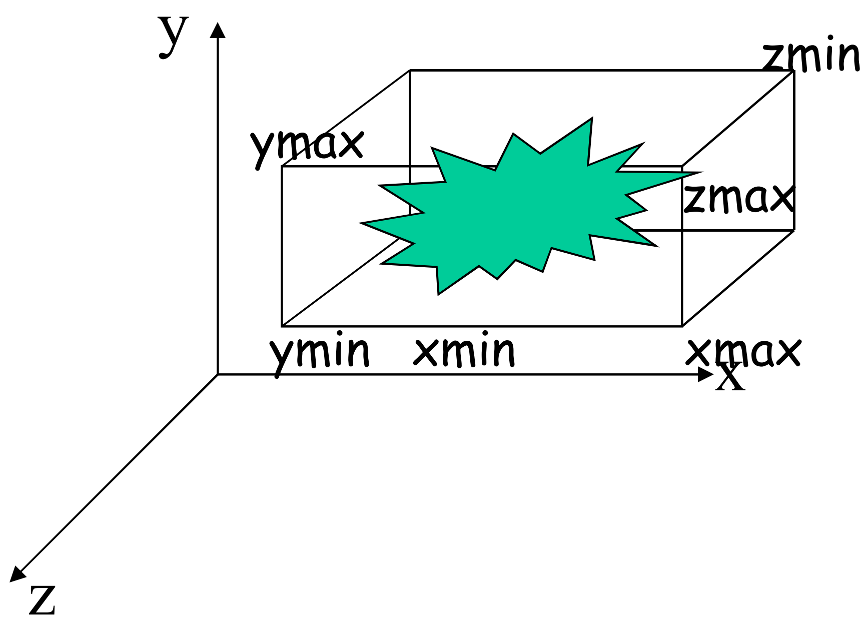
From all object points
select minimum & maximum X value
select minimum & maximum Y value
select minimum & maximum Z value

- Thus defined by two points only

Axis-Aligned Bounding Box(AABB)

- Support plane aligned with X,Y, & Z plane
- As initial test representing an object for early rejection

x – xmax = 0
x + xmin = 0
y – ymax = 0
y + ymin = 0
z – zmax = 0
z + zmin = 0



Point in AABB

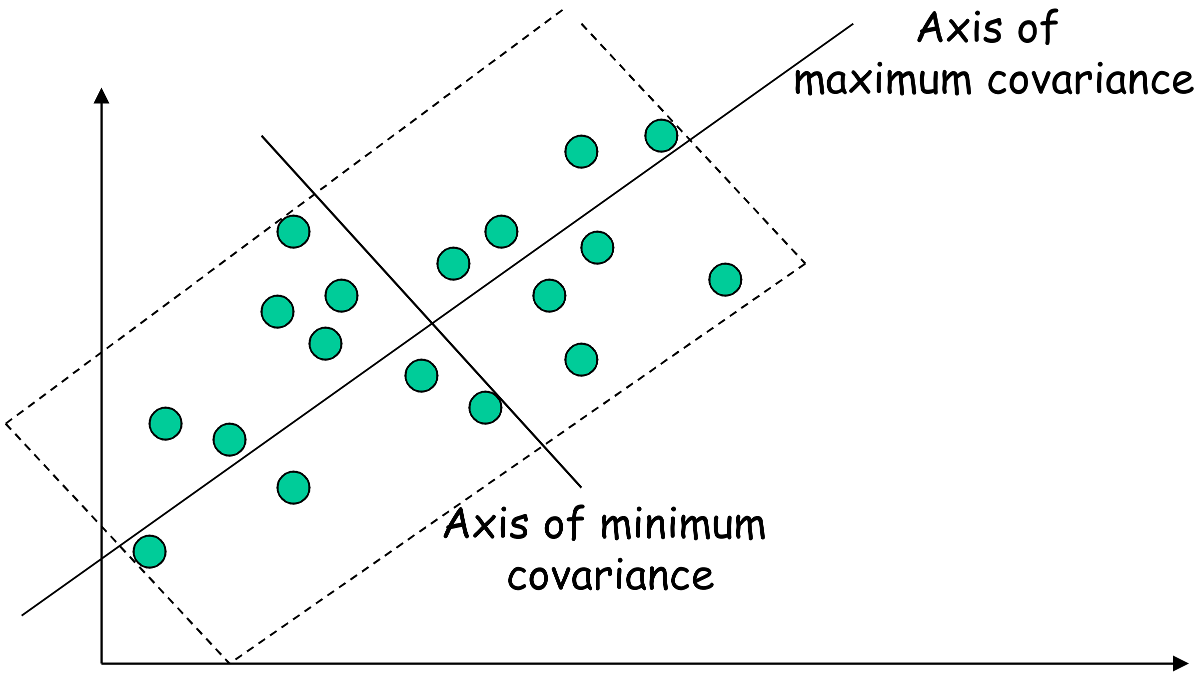
```
bool inside(point p)
{
  if (p.x>xmax) return false;
  if (p.x<xmin) return false;
  if (p.y>ymax) return false;
  if (p.y<ymin) return false;
  if (p.z>zmax) return false;
  if (p.z<zmin) return false;
  return true;
}
```

Use of AABB in Doom3

- Low complexity but needs to be updated as object moves e.g. rotational motion!

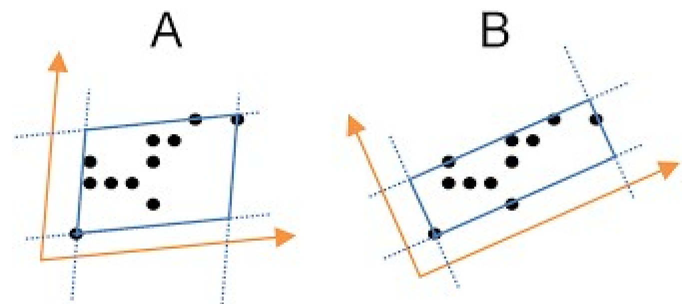
OBB

- Also called *principal component analysis* (**PCA**) in statistics



Oriented Bounding Box(OBB)

- Compute a tight-fit oriented rectangular box for a set of polygons
- Find the covariance matrix of the vertices



$$Mean = \frac{1}{3n}\sum_{i=1}^{n} \vec{p}_i + \vec{q}_i + \vec{r}_i$$

where **p,q**, & **r** are vertices of the *i*th triangle

n : number of triangles

OBB

- 3x3 *covariance* matrix is computed as

$$C_{jk} = \frac{1}{3n}\sum_{\mathrm{i}=1}^{n}(p_{ij} - M_j)(p_{ik} - M_k) + (q_{ij} - M_j)(q_{ik} - M_k) + (r_{ij} - M_j)(r_{ik} - M_k) \qquad 1 \le \mathrm{j,k} \le 3$$

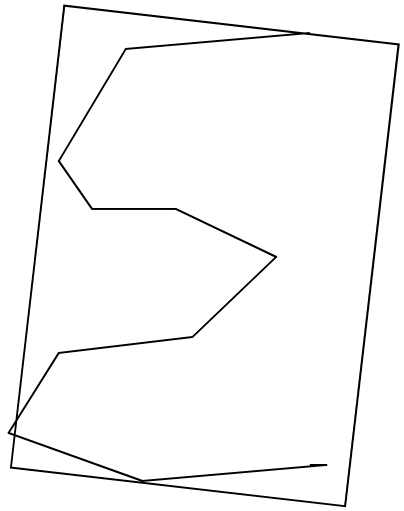
i : triangle index

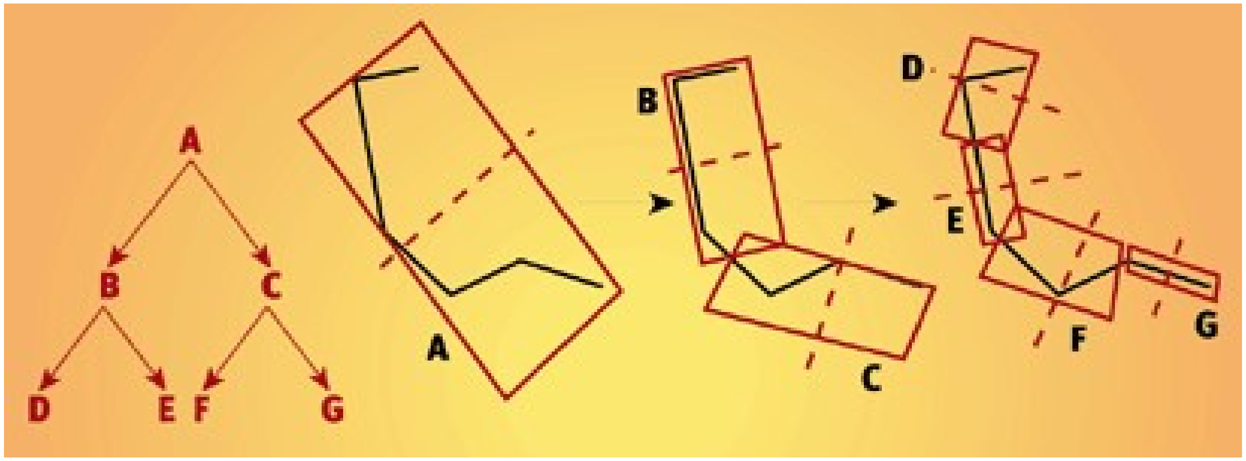
j, *k* : (x,y,z) component of the points

- A symmetric matrix with the *eigen* vectors yields a basis parallel to the face of the OBB

OBB

- Can build a OBB tree by recursively partitioning this tree into two halves using a plane orthogonal to one of its axes
- Too high computational efforts to put into real time application

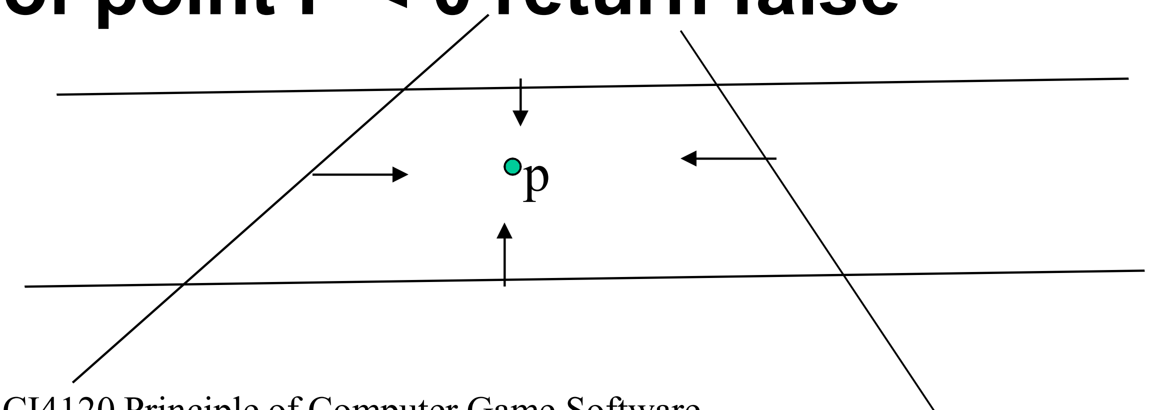




Point in Convex Polygon

- Looping through edges and make sure the point always in same hemispace w.r.t. edges

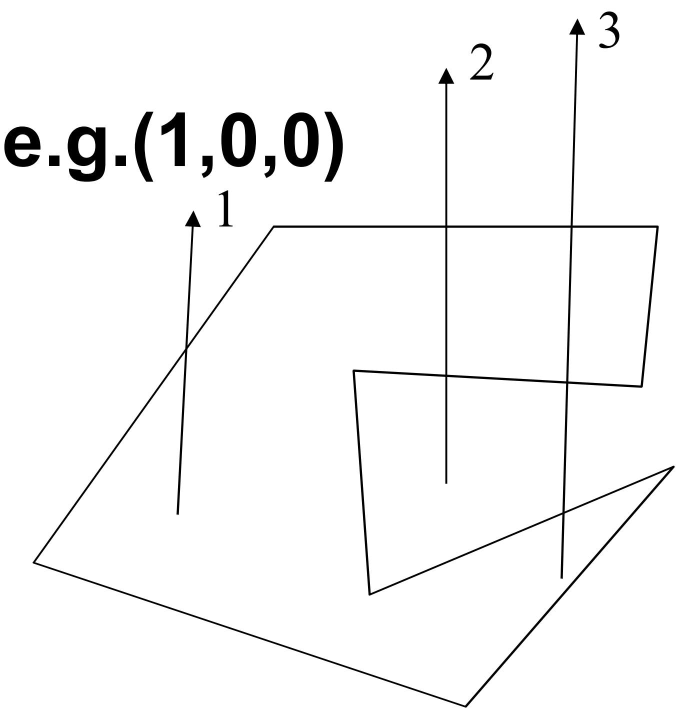
While we haven't done a full cycle
   compute edge vector
   compute up vector from edge & normal
   use edge & up to compute a plane
   if plane value of point P < 0 return false
return true



Point in Polygon(Jordan Curve Theorem)

- A point is inside a closed polygon *iff* number of crossings from a ray emanating from the point in an arbitrary direction and the edges of the polygon is *odd*

Choose an arbitrary direction e.g.(1,0,0)
Build ray vector, count = 0
for each edge
test ray-edge
if crossed
increase count
return count is odd



Point-in-Convex Object

- Similar to polygon test, with change to plane test

sign=sign of point-plane test of the first plane of support of object

for each plane of support

if sign is different from point-plane test of previous plane

return false

return true

Point-in-Convex Object

- Efficiently performed on convex hull of the objects
- Cost is *O*(no. of planes)
- Further optimized by early rejection of point-in-bounding-sphere test
- For concave object, decompose it into set of convex and performing the test

Point-in-Object (3DDDA)

- Processed the mesh into 3D regular grid
- Each grid cell contains those triangles whose barycenter located inside the cell
- Test only those triangles that lie in cell of the point – greatly reduced testing needed
- Further speed up by storing enumerated value (IN, OUT, DON'T KNOW) in each cell
- Drawback : Large memory footprint

Ray–plane Intersection

- Given the parametric form of a line as
  $$R = org + t * dir$$
  dir : direction vector
  org: origin
- And the plane **AX+BY+CZ+D = 0**, the parameter t of where intersection occur is given by

$$t = \frac{-(n \circ org + D)}{n \circ dir}$$

n: normal of plane (A,B,C)

Ray-Triangle test

Compute intersection between ray & support plane of triangle

If there is an intersection point, compute if that point is actually inside the triangle

Combine two primitive tests discussed before

Ray-AABB Test

- Assuming point outside the object

Reject back-facing normals from the six plane

From the 3 planes remaining, compute distance t on ray-plane test

Select the farthest of three distances

Test if that point inside the box

1. 6 dot product check the first step
2. 3 point ray tests
3. A few comparisons for the last step

Ray-Sphere Test

- Given the same parametric form of ray
  $$R = org + \lambda * dir$$
  and a sphere as
  $$(X - Xc)^2 + (Y - Yc)^2 + (Z - Zc)^2 = R^2$$
- rearranging give the form
  $$A\lambda^2 + B\lambda + C = 0$$
- The intersection can be 0, 1 or 2 solution only => the solution is well defined

Ray-Convex Hull Test

- Loop through all the planes of the convex hull
- If all tests are negative(two points lie outside the hull), no intersection
- Otherwise compute intersection point between the plane and ray

Ray-General Object

- Ray-Concave object intersection computation is complex
- Use *Jordan Curve* Theorem
- Since there may have several intersections, need additional information to decide which one to return

Moving Tests

- Need to compute trajectories of objects
- Detect both if collision took place and when it took place
- Sphere-sphere test
- Start with two points moving in space

$$pos_a = pos0_a + v_a t$$
$$pos_b = pos0_b + v_b t$$

Sphere-sphere Test

- Compute the vector difference and take the square of difference vector magnitude

$$pos_a - pos_b = dpos0 + dv \cdot t$$

$$dist = sqrt(A + B \cdot t + C \cdot t^2)$$

- where

$$A = |dpos|^2$$

$$B = 2 * (dpos \circ dv)$$

$$C = |dv|^2$$

Sphere-sphere Test

- Assume r1 & r2 radius of the two spheres, collision happens when

  dist = r1 + r2

- Solving the equation & rearranging terms, collision occur if

$$(dpos \circ dv)^2 > (|dpos|^2 - (r1 + r2)^2)|dv|^2$$

Point-BSP Triangle Set Test

- Traversing the BSP tree to locate the node we are in
- The planes we visit on the way form a convex shape around the point
- do point vs. polygon test to check if lying in valid region

Mesh Vs. Mesh Test

- Usually deals with systems involving many different moving objects
- Requires N**x**N tests => costly computation
- Using sweep and prune – detecting bounding box overlap
- Two BB overlap iff intervals overlap in all x-, y-, and z-axes

Mesh Vs. Mesh Test

Construct 3 lists (x-, y- & z-axes)

For each object

- project BB onto x-, y-, & z- axes
- store projection as in & out in lists

Sort list by coordinate

Determine overlap by scanning the list

Store overlaps in X, Y & Z array

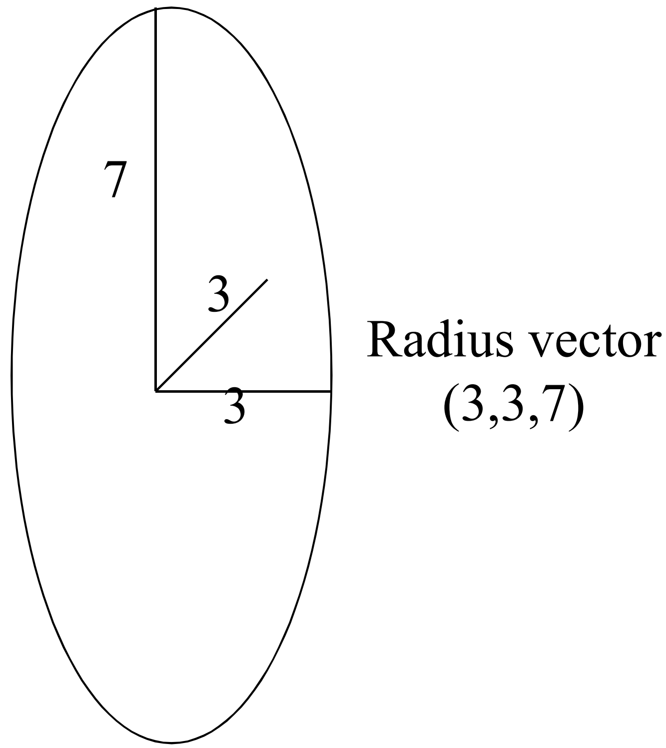
If overlap occur in all three directions

- perform fine collision algorithm

Typical implementation in FPS

- The player is modeled as an ellipsoid
- The ellipsoid is transformed into a sphere by

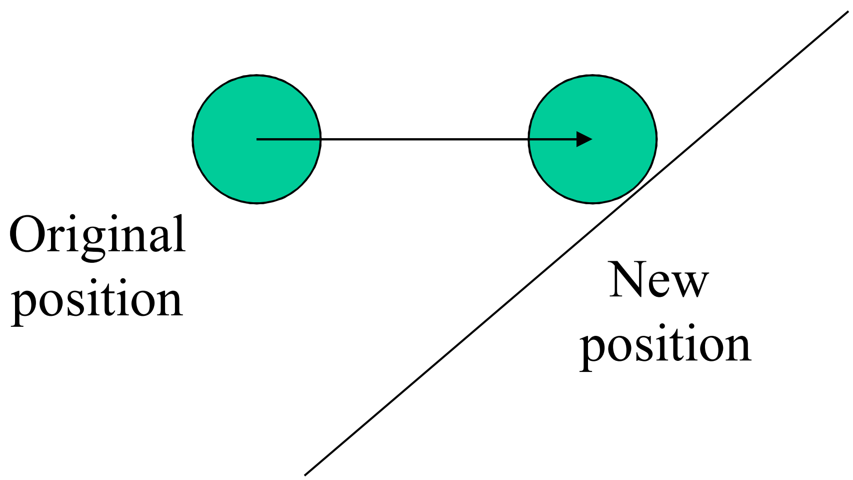
$$M = \begin{bmatrix} \frac{1}{x} & 0 & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$$



(x,y,z) : radius vector of ellipsoid

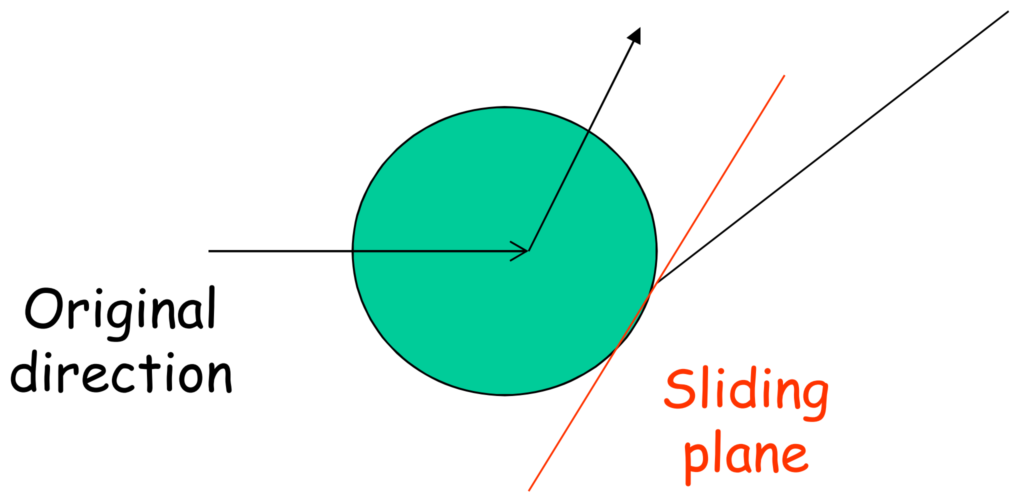
Typical implementation in FPS

- Subsequent calculation in ellipsoid space i.e. player as a sphere, which simplify calculations
- Swept sphere would have to be used to ensure collision is detected



Typical implementation in FPS

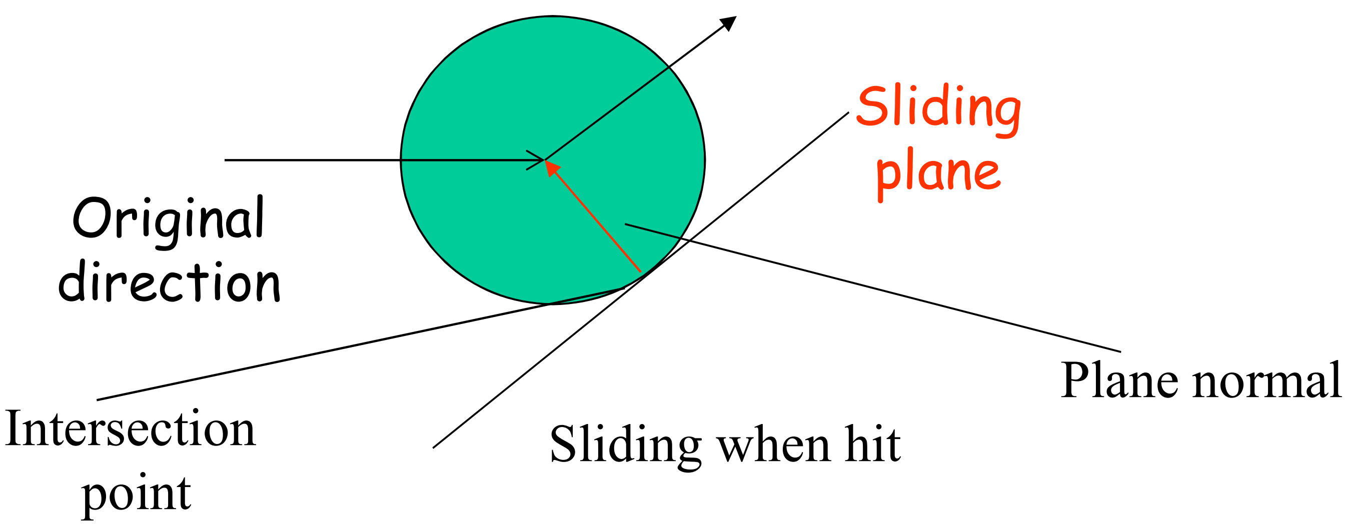
- If collision did happen, sliding occur along the sliding plane
- Sliding plane may not be same as the colliding polygon



Sliding when hit

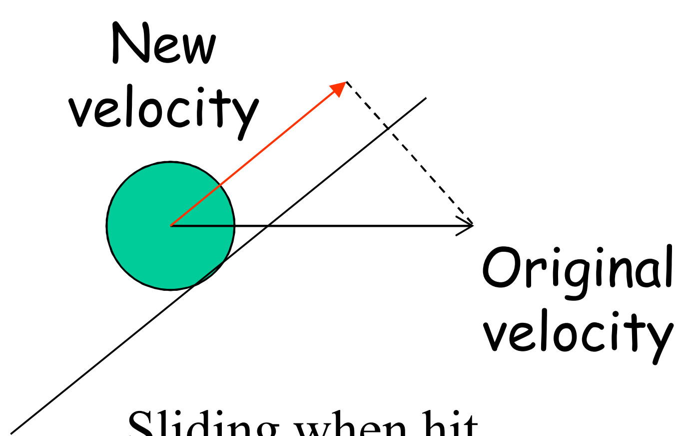
Typical implementation in FPS

- Sliding plane defined by plane normal & intersection point
- Plane normal simply the vector *from intersection point to sphere center*



Typical implementation in FPS

- The new moving velocity is readily obtained by projecting original velocity on the sliding plane



Sliding when hit

Typical implementation in FPS

- Gravity effect can be computed by making two calls to the collision detection/response module : 1 for player move, 1 for gravity
- Simply combining the two vectors may result in unable to walk up staircases

Typical implementation in FPS

- Reference
  - Original proposal by Paul Nettle, link no longer available
  - Refined & elaborated
  - http://www.peroxide.dk/papers/collision/collision.pdf

Recommended readings

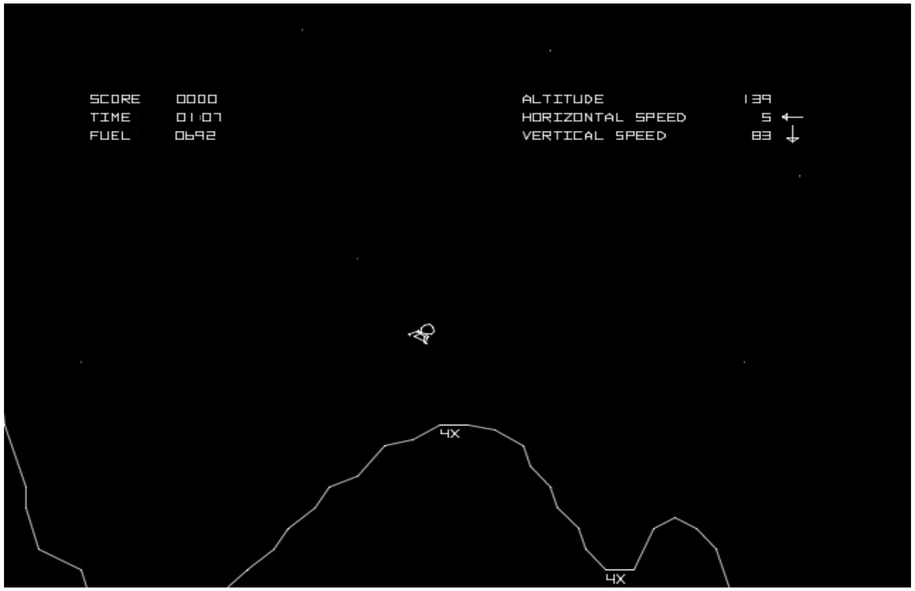
- Chapter 22 of **Core techniques & algorithms in Game Programming**
- Chapter 15 of **3D Games Real-time Rendering & Software Technology**

Physics in Game

- Traditional games did not implement physics – in-game effects are just animations
- Physics can bring more possibility in terms of game play
- With advance of processing power, physics can more accurately mimic our real world e.g. the fall off of enemy after being hit by a bullet, car racing around tight curve.

First Physics in Game

- *Lunar Lander* : the first physics inspired game
- Pilot a landing craft to a safe landing, practise physics concept (gravity & thrust)



http://my.ign.com/atari/lunar-lander

Physics

- Concern with *kinematics* & *dynamics* of rigid bodies in contrary to our usual representation of a single point mass
  - **Kinematics**: motion of body without regard to forces that act on the body (body pose)
  - **Dynamics**: both motion of body and forces that acting on them

Simulating Dynamics

- Interested in dynamics most of the time in game
- Topics — Application

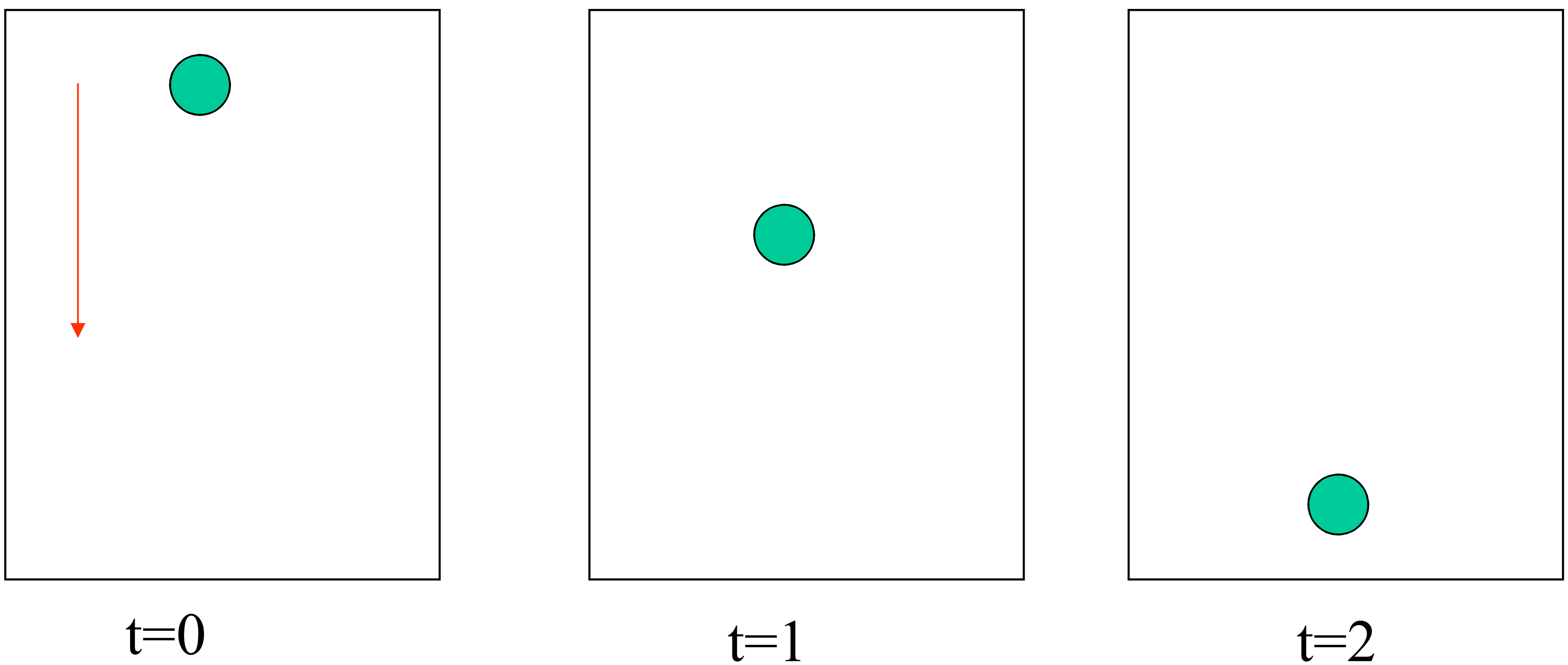
Topics	Application
– Rigid body (articulated)	human
– Cloth dynamic	human
– fluid dynamic	water
–	explosion

Physics

- Typical procedure for simulating physics effect on computer:
- For next time frame
  – Calculate the resultant forces
  – Solve the force equation to obtain motion
  – Update the states of the system

Start from simple

- Suppose a sphere is free falling and we want to show it in our 3D game world.



Representation

- We use (x,y,z) to represent the coordinate of the object, and the displacement at *each time frame* is

  g: gravity

  t : time

$$y = \frac{1}{2} g t^2$$

- all calculations must reference the initial time, thus not suitable for the game world (time is infinitely extended and events happening all the time)

Representation

- We rewrite it into

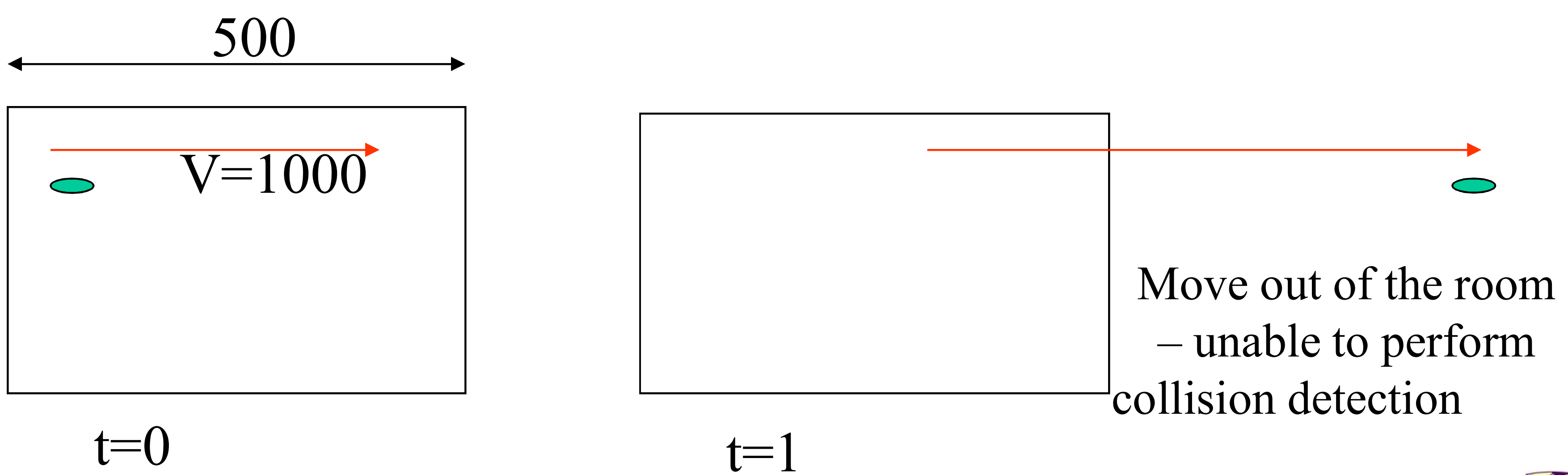
$$y' = y + vt$$ y,y`: old,new position of the object

$$v' = v + gt$$ v,v': old,new velocity of the object

- quantities with ` is the next time frame values of those without `
- acceleration is constant in our case

Euler time integration

- Need additional parameters to represent velocity
- Need careful chosen time frame *t* for different situations when simulated velocity is high w.r.t. sampling rate for collision detection



Euler time integration

- Actually we are solving the following *differential equation* for y

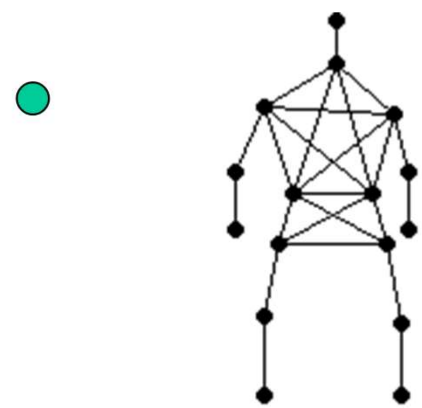
$$\frac{d^2 y}{dt^2} = g$$

- Basically it can be used to solve all kinds of differential equations

Rigid Body Simulation

- Use connected masses to simulate a rigid body
- Connecting stick can be a *spring* to simulate deformable object

Simple connected stick



particle/stick configuration used in *Hitman*

On notations

- Suppose we compute cross product between vector w and n

$$\omega \times \nu$$

- This can be rewritten as

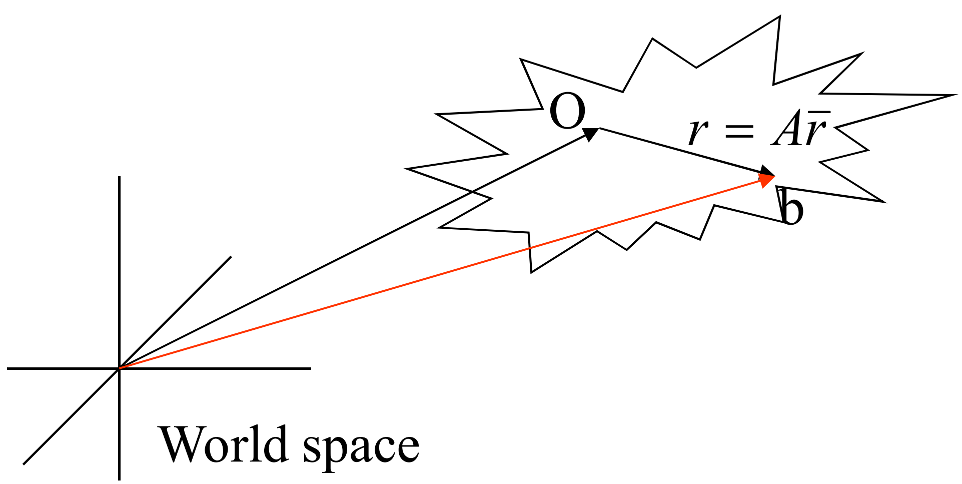
$$\begin{vmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{vmatrix} \begin{vmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{vmatrix} = \widetilde{\omega}\nu$$

Parameters

- 6 Degree of freedom (DOF) : 3 for translation, 3 for rotation
- Rotation representation is still an open problem
- Typical representation: roll/pitch/yaw(RPY) 3x3 matrix, axis-angle, quaternion
- Each suffer from their corresponding shortcomings or *singularities*

Modeling

- Rigid body is modeled as a collection of point masses
- Integrate over the whole body through a number of time steps (*Euler integration*)



O: origin of the rigid body

r : world space position

A: 3x3 rotation matrix

`Lower case:` ***`vector`***
`Upper case:` ***`matrix`***

Kinematics

$$b = o + A\bar{r}$$
$$= o + r$$

Differentiating w.r.t time to give velocity

$$\dot{b} = \dot{o} + \dot{A}\bar{r} + A\dot{\bar{r}}$$
$$= \dot{o} + \omega \times r$$

$\bar{r}$ - constant vector $\Rightarrow$ zero $\dot{\bar{r}}$

ω : angular velocity

Further derivative gives the acceleration

$$\ddot{b} = \ddot{o} + \alpha \times r + \omega \times (\omega \times r)$$

α : angular acceleration

Centripedal acceleration

Dynamics(translational)

$$F = \sum_i m_i \dot{v}_i = \sum_i m_i a_i = Ma_{CM}$$

F: total force exerted on the body

M : mass of whole body

a_{CM}: acceleration of centre of mass

Dynamics(rotational)

- Angular momentum of a point B w.r.t. point A is

$$L_{AB} = r_{AB} \times p_B$$

p_B: linear momentum

- Derivative of angular momentum i.e. torque is given by

$$\dot{L}_{AB} = \tau_{AB} = r_{AB} \times F_B$$

Dynamics

- The angular momentum is thus

$$L_{CM} = \sum_i r_i \times m_i \dot{r}_i$$
$$= \sum_{\mathrm{i}} m_i r_i \times \omega \times r_i$$
$$= \sum_i -m_i r_i \times (r_i \times \omega)$$

- Rewriting cross product into matrix form

$$L_{CM} = \sum_i - m \tilde{r}_i \tilde{r}_i \omega = I_{CM} \omega$$

I_{CM}: Moment of inertia

Rotation Matrix Properties

- A 3D rotation matrix A is orthogonal i.e.

$$AA^t = I$$

- Where $\mathbf{A}^t$ is the transpose of **A**
- Now suppose we have a rotated vector r'

$$r` = Ar$$

- We want to find a matrix B such that

$$B`Ar = ABr$$

Similarity Transform

- Expanding *rhs* by I

$$B`Ar = ABIr$$

$$= ABA^t Ar$$

$$\Rightarrow B` \equiv ABA^t$$

Similarity Transform

- Usage:
- In the rigid body representation

$$r = A\bar{r}$$

- $\bar{r}$ is the object space coordinate w.r.t. object center of mass
- **A** is object to world space rotation

Similarity Transform(2)

- In calculating the moment of inertia for a rigid body (world space)

$$\overline{I_A} = \sum_i - m\tilde{\tilde{r}}_{Ai}\tilde{\tilde{r}}_{Ai}$$

- is the body space moment of inertia, and can be pre-calculated (and its inverse **I**$^{-1}$)

$$I_A^{-1} = A\overline{I^{-1}{}_A}A^t$$

Simulation algorithm

Initialization:

determine body constants: $\bar{I}^{-1}{}_{CM}, M$

determine initial conditions: $r^0{}_{CM}, v^0{}_{CM}, A^0, L^0{}_{CM}$

compute initial auxiliary

$$I_{CM}^{0^{-1}} = A^0 \bar{I}_{CM}^{-1} A^{0^T}$$

$$\omega^0 = \bar{I}_{CM}^{0^{-1}} L_{CM}{}^0$$

Simulation algorithm

Simulation:

compute individual forces & application points: F_i, r_i

compute total forces & torque: $F_T^n = \sum_i F_i, \tau_T^n = \sum_i r_i \times F_i$

integrate quantities: $r_{CM}^{n+1} = r_{CM}^n + h v_{CM}^n$

$$v_{CM}^{n+1} = v_{CM}^n + h \frac{F_T^n}{M}$$

h: time step

$$A^{n+1} = A^n + h \tilde{\omega}^n A^n$$

$$L_{CM}^{n+1} = L_{CM}^n + h \tau_T^n$$

Compute auxiliary quantities: $I_{CM}^{n+1^{-1}} = A^{n+1} \bar{I}_{CM}^{-1} A^{n+1^T}$

$$\omega^{n+1} = I_{CM}^{n+1^{-1}} L_{CM}^{n+1}$$

Further

- Suitable for aircraft modeling
- No contact physics involved i.e. can’t rest on ground
- No simultaneous multiple collision point
- No friction during collision & contact
- Need collision detection

Physics SDK

- Abstract the in-game characters into primitives
- Provide attributes update e.g. position, orientation after each game time step
- Examples :
- Havok http://www.havok.com/
- PhysX http://www.geforce.com/hardware/technology/physx
- Bullet http://www.bulletphysics.com

Good and Bad

- Advantages
  - With physics, artist don’t have to script all behavior (e.g. box falling)
  - Improved realism
  - Player can exercise more control

- Drawback
  - Lack of artist control
  - May conflict with game design

Further

- http://chrishecker.com/Rigid_Body_Dynamics#articles
- https://www.gamasutra.com/view/feature/131312/contact_physics.php
- http://www.newtondynamics.com/
- http://www.havok.com
- https://pybullet.org/wordpress/

Recommended readings

- Chapter 14 of **3D Games Real-time Rendering & Software Technology**
- **Chapter 22 of Core Techniques & Algorithms**
- **http://www.videotutorialsrock.com/opengl_tutorial/collision_detection/text.php**