



University of British Columbia  
 CPSC 314 Computer Graphics  
 Jan-Apr 2005

Tamara Munzner

## Collision Detection

Week 10, Wed Mar 16

<http://www.ugrad.cs.ubc.ca/~cs314/Vjan2005>

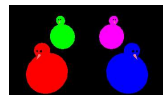
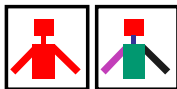
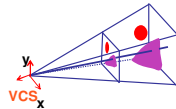
## Project 3

- extra credit explained
  - can earn up to 110% of total points
  - grading scheme: buckets
    - minus, check-minus, check, check-plus, plus
    - plus = extra credit realm

2

## Review: Picking Methods

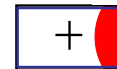
- manual ray intersection
- bounding extents
- backbuffer coding



3

## Review: Select/Hit Picking

- assign (hierarchical) integer key/name(s)
- small region around cursor as new viewport



- redraw in selection mode
  - equivalent to casting pick "tube"
  - store keys, depth for drawn objects in hit list
- examine hit list
  - usually use frontmost, but up to application

4

## Collision Detection

5

## Collision Detection

- do objects collide/intersect?
  - static, dynamic
- simple case: picking as collision detection
  - check if ray cast from cursor position collides with any object in scene
  - simple shooting
    - projectile arrives instantly, zero travel time
- better: projectile and target move over time
  - see if collides with object during trajectory

6

## Collision Detection Applications

- determining if player hit wall/floor/obstacle
  - terrain following (floor), maze games (walls)
  - stop them walking through it
- determining if projectile has hit target
- determining if player has hit target
  - punch/kick (desired), car crash (not desired)
- detecting points at which behavior should change
  - car in the air returning to the ground
- cleaning up animation
  - making sure a motion-captured character's feet do not pass through the floor
- simulating motion
  - physics, or cloth, or something else

7

## Naive Collision Detection

- for each object  $i$  containing polygons  $p$ 
  - test for intersection with object  $j$  containing polygons  $q$
- for polyhedral objects, test if object  $i$  penetrates surface of  $j$ 
  - test if vertices of  $i$  straddle polygon  $q$  of  $j$ 
    - if straddle, then test intersection of polygon  $q$  with polygon  $p$  of object  $i$
- very expensive!  $O(n^2)$

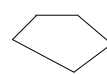
8

## Choosing an Algorithm

- primary factor: geometry of colliding objects
  - "object" could be a point, or line segment
  - object could be specific shape: sphere, triangle, cube
  - objects can be concave/convex, solid/hollow, deformable/rigid, manifold/non-manifold
- secondary factor: way in which objects move
  - different algorithms for fast or slow moving objects
  - different algorithms depending on how frequently the object must be updated
- other factors: speed, simplicity, robustness

9

## Terminology Illustrated

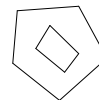


Convex

An object is convex if for every pair of points inside the object, the line joining them is also inside the object

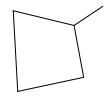


Concave



Manifold

An surface is manifold if every point on it is homeomorphic to a disk. Roughly, every edge has two faces joining it, and there are no isolated vertices.



Non-Manifold

10

## Robustness

- for our purposes, collision detection code is *robust* if
  - doesn't crash or infinite loop on *any* case that might occur
    - better if it doesn't fail on any case at all, even if the case is supposed to be "impossible"
  - always gives some answer that is meaningful, or *explicitly* reports that it cannot give an answer
  - can handle many forms of geometry
  - can detect problems with the input geometry, particularly if that geometry is supposed to meet some conditions (such as convexity)
- robustness is remarkably hard to obtain

11

## Types of Geometry

- points
- lines, rays and line segments
- spheres, cylinders and cones
- cubes, rectilinear boxes
  - AABB: axis aligned bounding box
  - OBB: oriented bounding box, arbitrary alignment
- k-dops – shapes bounded by planes at fixed orientations
- convex, manifold meshes – any mesh can be triangulated
  - concave meshes can be broken into convex chunks, by hand
- triangle soup
- more general curved surfaces, but often not used in games



12

## Fundamental Design Principles

- several principles to consider when designing collision detection strategy
  - if more than one test available, with different costs: how do you combine them?
  - how do you avoid unnecessary tests?
  - how do you make tests cheaper?

13

## Fundamental Design Principles

- *fast simple tests first*, eliminate many potential collisions
  - test bounding volumes before testing individual triangles
- exploit *locality*, eliminate many potential collisions
  - use cell structures to avoid considering distant objects
- use as much *information* as possible about geometry
  - spheres have special properties that speed collision testing
- exploit *coherence* between successive tests
  - things don't typically change much between two frames

14

## Player-Wall Collisions

- first person games must prevent the player from walking through walls and other obstacles
- most general case: player and walls are polygonal meshes
- each frame, player moves along path not known in advance
  - assume piecewise linear: straight steps on each frame
  - assume player's motion could be fast

15

## Stupid Algorithm

- on each step, do a general mesh-to-mesh intersection test to find out if the player intersects the wall
- if they do, refuse to allow the player to move
- problems with this approach? how can we improve:
  - in speed?
  - in accuracy?
  - in response?

16

## Ways to Improve

- even seemingly simple problem of determining if the player hit the wall reveals a wealth of techniques
  - collision proxies
  - spatial data structures to localize
  - finding precise collision times
  - responding to collisions

17

## Collision Proxies

- **proxy**: something that takes place of real object
  - cheaper than general mesh-mesh intersections
- **collision proxy (bounding volume)** is piece of geometry used to represent complex object for purposes of finding collision
  - if proxy collides, object is said to collide
  - collision points mapped back onto original object
- good proxy: cheap to compute collisions for, tight fit to the real geometry
- common proxies: sphere, cylinder, box, ellipsoid
  - consider: fat player, thin player, rocket, car ...

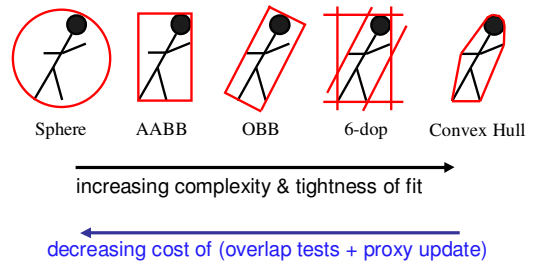
18

### Why Proxies Work

- proxies exploit facts about human perception
  - we are extraordinarily bad at determining correctness of collision between two complex objects
  - the more stuff is happening, and the faster it happens, the more problems we have detecting errors
  - players frequently cannot see themselves
  - we are bad at predicting what should happen in response to a collision

19

### Trade-off in Choosing Proxies



20

### Pair Reduction

- want proxy for any moving object requiring collision detection
- before pair of objects tested in any detail, quickly test if proxies intersect
- when lots of moving objects, even this quick bounding sphere test can take too long:  $N^2$  times if there are  $N$  objects
- reducing this  $N^2$  problem is called *pair reduction*
- pair testing isn't a big issue until  $N > 50$  or so...

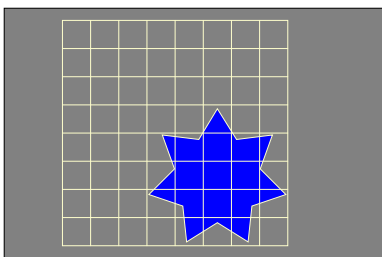
21

### Spatial Data Structures

- can only hit something that is close
- spatial data structures tell you what is close to object
  - uniform grid, octrees, kd-trees, BSP trees, OBB trees, k-dop trees
- for player-wall problem, typically use same spatial data structure as for rendering
  - BSP trees most common

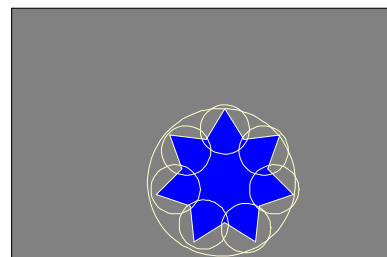
22

### Uniform Grids



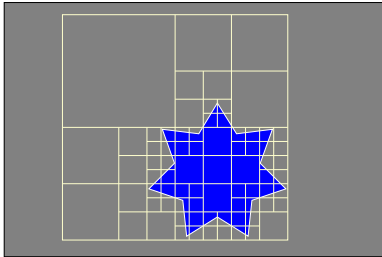
23

### Bounding Volume Hierarchies



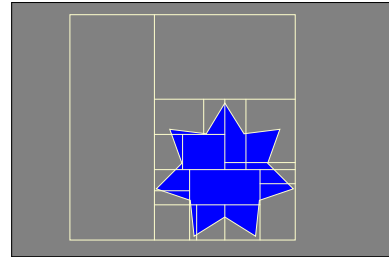
24

### Octrees



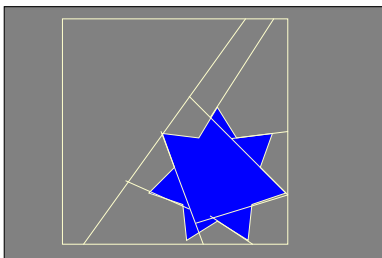
25

### KD Trees



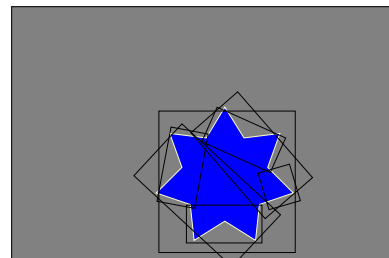
26

### BSP Trees



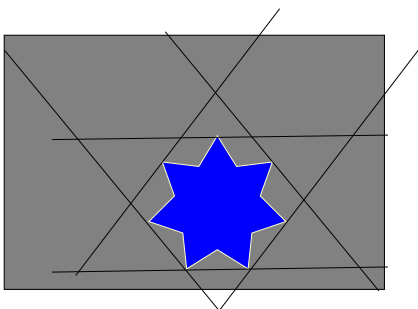
27

### OBB Trees



28

### K-Dops



29

### Testing BVH's

```
TestBVH(A,B) {
  if(not overlap(ABV, BBV)) return FALSE;
  else if(isLeaf(A)) {
    if(isLeaf(B)) {
      for each triangle pair (Ta, Tb)
        if(overlap(Ta, Tb)) AddIntersectionToList();
    }
    else {
      for each child Cb of B
        TestBVH(A, Cb);
    }
  }
  else {
    for each child Ca of A
      TestBVH(Ca, B);
  }
}
```

30

### Optimization Structures

- All of these optimization structures can be used in either 2D or 3D
- Packing in memory may affect caching and performance

31

### Exploiting Coherence

- player normally doesn't move far between frames
- cells they intersected the last time are
  - probably the same cells they intersect now
  - or at least they are close
- aim is to track which cells the player is in without doing a full search each time
- easiest to exploit with a cell portal structure

32

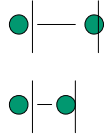
### Cell-Portal Collisions

- keep track which cell/s player is currently intersecting
  - can have more than one if the player straddles a cell boundary
  - always use a proxy (bounding volume) for tracking cells
  - also keep track of which portals the player is straddling
- player can only enter new cell through portal
- on each frame
  - intersect the player with the current cell walls and contents (because they're solid)
  - intersect the player with the portals
  - if the player intersects a portal, check that they are considered "in" the neighbor cell
  - if the player no longer straddles a portal, they have just left a cell

33

### Precise Collision Times

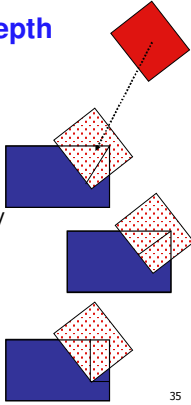
- generally a player will go from not intersecting to interpenetrating in the course of a frame
- we typically would like the exact collision time and place
  - response is generally better
  - interpenetration may be algorithmically hard to manage
  - interpenetration is difficult to quantify
  - numerical root finding problem
- more than one way to do it:
  - hacked (but fast) clean up
  - *interval halving* (binary search)



34

### Defining Penetration Depth

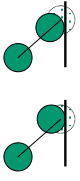
- more than one way to define penetration depth
  - distance to move back along incoming path to avoid collision
    - may be difficult to compute
  - minimum distance to move in any direction to avoid collision
    - often also difficult to compute
    - distance in some particular direction
    - but what direction?
    - "normal" to penetration surface



35

### Hacked Clean Up

- know time  $t$ , position  $x$ , such that penetration occurs
- simply move position so that objects just touch, leave time the same
- multiple choices for how to move:
  - back along motion path
  - shortest distance to avoid penetration
  - some other option



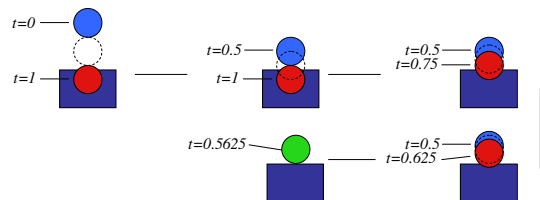
36

## Interval Halving

- search through time for the point at which the objects collide
- know when objects were not penetrating (last frame)
- know when they are penetrating (this frame)
- thus have upper and lower bound on collision time
  - later than last frame, earlier than this frame
- do a series of tests to bring bounds closer together
- each test checks for collision at midpoint of current time interval
  - if collision, midpoint becomes new upper bound
  - if not, midpoint becomes new lower bound
- keep going until the bounds are the same (or as accurate as desired)

37

## Interval Halving Example



38

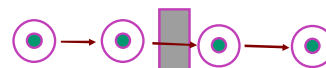
## Interval Halving Discussion

- advantages
  - finds accurate collisions in time and space, which may be essential
  - not too expensive
- disadvantages
  - takes longer than hack (but note that time is bounded, and you get to control it)
  - may not work for fast moving objects and thin obstacles
- method of choice for many applications

39

## Temporal Sampling

- subtle point: collision detection is about the algorithms for finding collisions *in time* as much as space
- temporal sampling
  - aliasing: can miss collision completely!



40

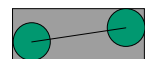
## Managing Fast Moving Objects

- several ways to do it, with increasing costs
- movement line: test line segment representing motion of object center
  - pros: works for large obstacles, cheap
  - cons: may still miss collisions. how?
- conservative prediction: only move objects as far as you can be sure to catch collision
  - increase temporal sampling rate
  - pros: will find all collisions
  - cons: may be expensive, how to pick step size
- space-time bounds: bound the object in space and time, check bound
  - pros: will find all collisions
  - cons: expensive, must bound motion

41

## Prediction and Bounds

- conservative motion
  - assume maximum velocity, smallest feature size
  - largest conservative step is smallest distance divided by the highest speed - clearly could be very small
  - other more complex metrics are possible
- bounding motion
  - assume linear motion
  - find radius of bounding sphere
  - build box that will contain that sphere for frame step
  - also works for ballistic and some other predictable motions
- simple alternative: just miss the hard cases
  - player may not notice!



42

### Collision Response

- for player motions, often best thing to do is move player tangentially to obstacle
- do recursively to ensure all collisions caught
  - find time and place of collision
  - adjust velocity of player
  - repeat with new velocity, start time, start position (reduced time interval)
- handling multiple contacts at same time
  - find a direction that is tangential to all contacts

43

### Related Reading

- Real-Time Rendering
  - Tomas Moller and Eric Haines
  - on reserve in CICS reading room

44

### Acknowledgement

- slides borrow heavily from
  - Stephen Chenney, (UWisc CS679)
    - <http://www.cs.wisc.edu/~schenney/courses/cs679-f2003/lectures/cs679-22.ppt>
- slides borrow lightly from
  - Steve Rotenberg, (UCSD CSE169)
    - [http://graphics.ucsd.edu/courses/cse169\\_w05/CSE169\\_17.ppt](http://graphics.ucsd.edu/courses/cse169_w05/CSE169_17.ppt)

45