University of British Columbia
CPSC 314 Computer Graphics
May-June 2005

Tamara Munzner

Picking, Collision

Week 4, Tue May 31

http://www.ugrad.cs.ubc.ca/~cs314/Vmay2005

News

- extension for P4 proposals
  - now due Thu 6pm, not Wed 4pm
- rearranging lecture schedule slightly
  - picking, collision today
  - textures Thursday (no change)
  - hidden surfaces next week
- reminder
  - final Thu 6/16, P4 due Sat 6/18

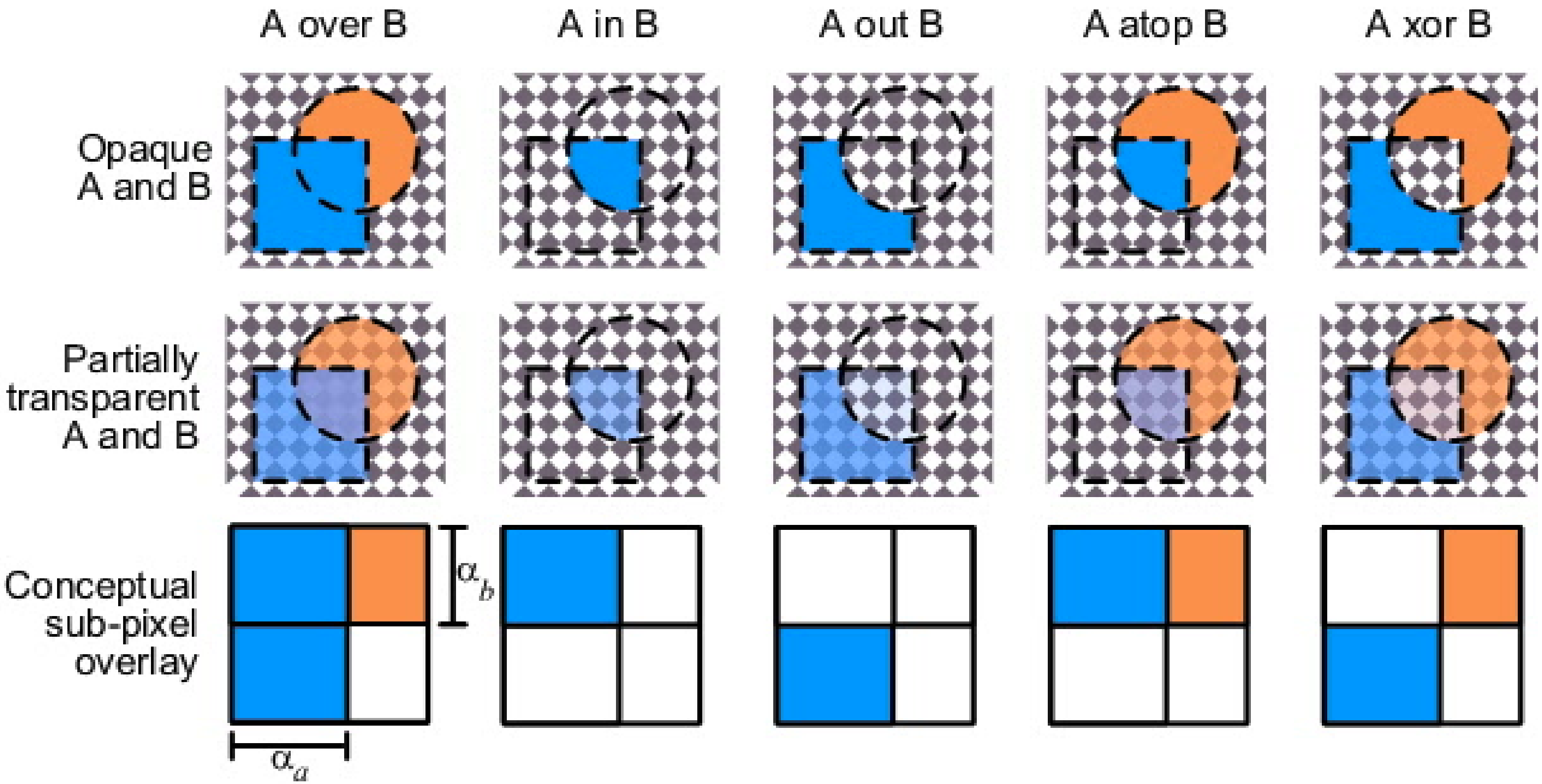
Common Mistakes on H2

- lookat point vs. gaze vector (eye – lookat)
- remember that NDC coordinate range is 2 (from -1 to 1), not 1
- remember homogenise and/or normalize points as needed
- on derivations, need more than just restating definition
- don't forget to flip y axis when converting to display coords

Midterm

- picture IDs out and face up, please
- sit where there is a test paper
- don't open paper until you get the word

Review: Compositing



Correction/Review: Premultiplying Colors

- specify opacity with alpha channel: (r,g,b,α)
  - $\alpha=1$: opaque, $\alpha=.5$: translucent, $\alpha=0$: transparent

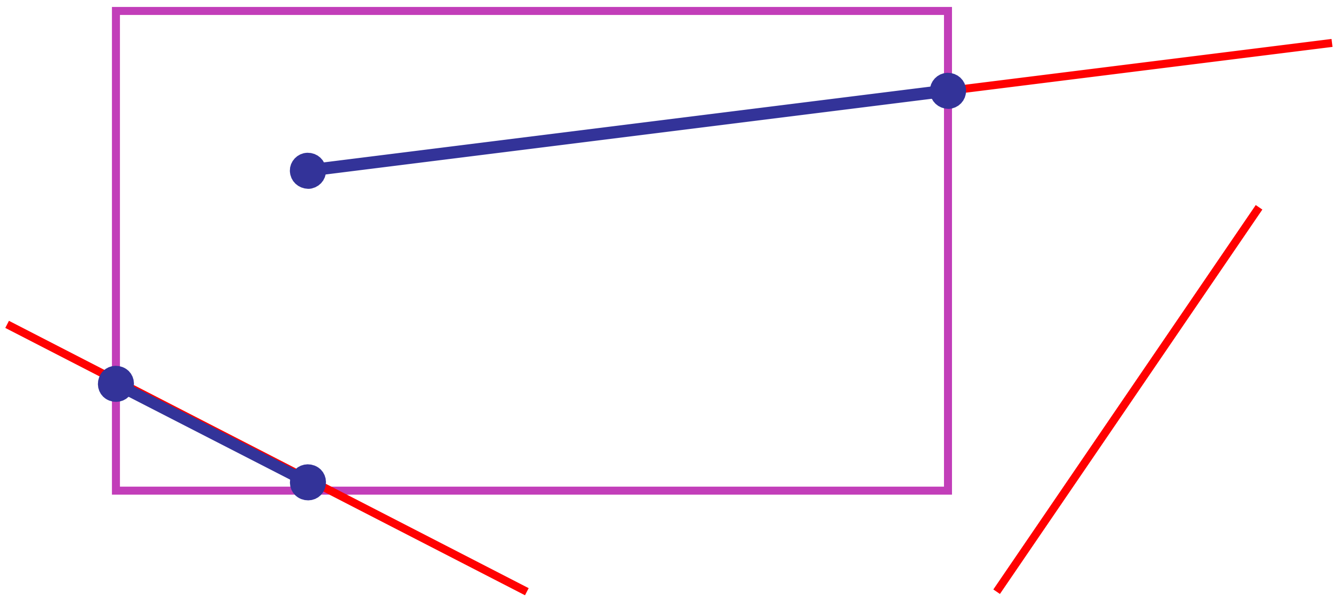
- **A** *over* **B**
  - $\mathbf{C} = \alpha\mathbf{A} + (1-\alpha)\mathbf{B}$

- but what if **B** is also partially transparent**?**
  - $\mathbf{C} = \alpha\mathbf{A} + (1-\alpha)\,\beta\mathbf{B} = \beta\mathbf{B} + \alpha\mathbf{A} + \beta\mathbf{B} - \alpha\,\beta\mathbf{B}$
  - $\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$
    - 3 multiplies, different equations for alpha vs. RGB

- premultiplying by alpha
  - $\mathbf{C'} = \gamma\,\mathbf{C},\ \mathbf{B'} = \beta\mathbf{B},\ \mathbf{A'} = \alpha\mathbf{A}$
  - $\mathbf{C'} = \mathbf{B'} + \mathbf{A'} - \alpha\mathbf{B'}$
  - $\gamma = \beta + \alpha - \alpha\beta$
    - 1 multiply to find C, same equations for alpha and RGB

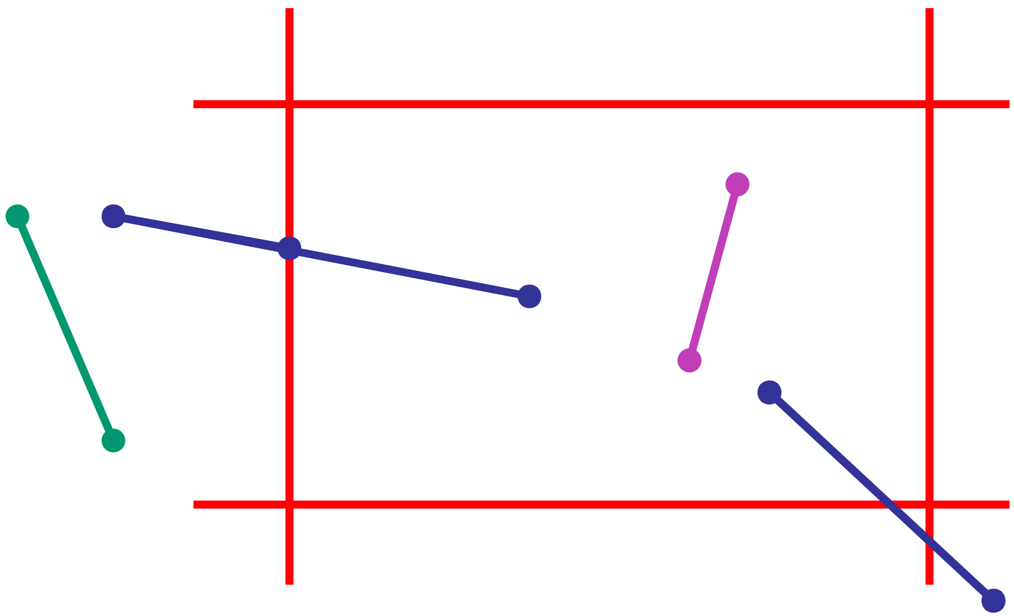
Review: Clipping

- analytically calculating the portions of primitives within the viewport



Review: Clipping Lines To Viewport

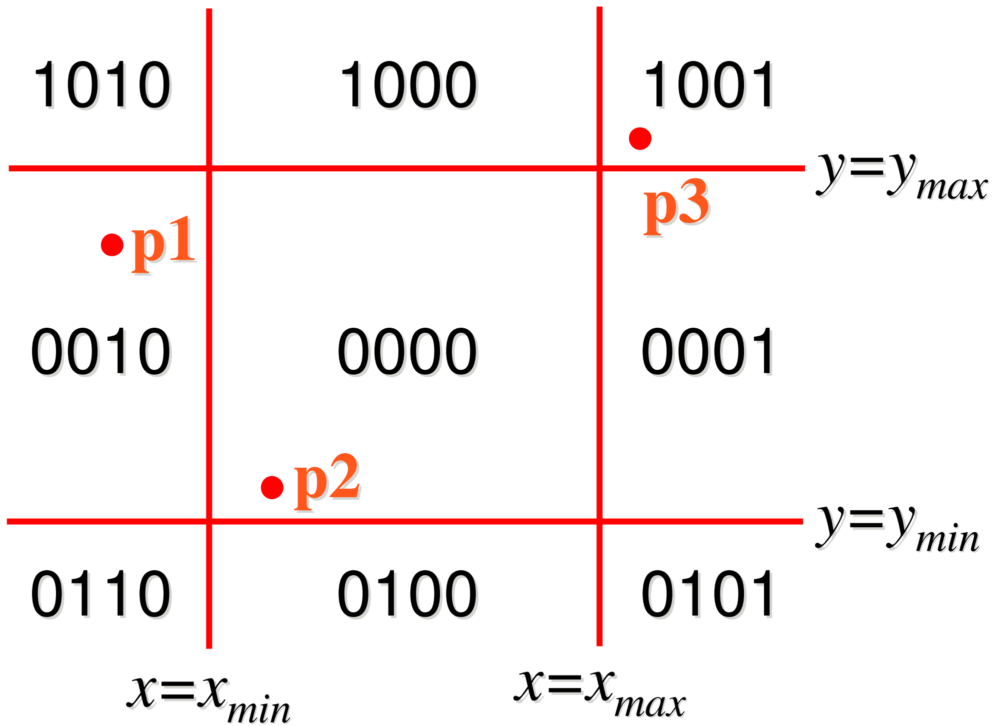
- combining trivial accepts/rejects
  - trivially accept lines with both endpoints inside all edges of the viewport
  - trivially reject lines with both endpoints outside the same edge of the viewport
  - otherwise, reduce to trivial cases by splitting into two segments



Review: Cohen-Sutherland Line Clipping

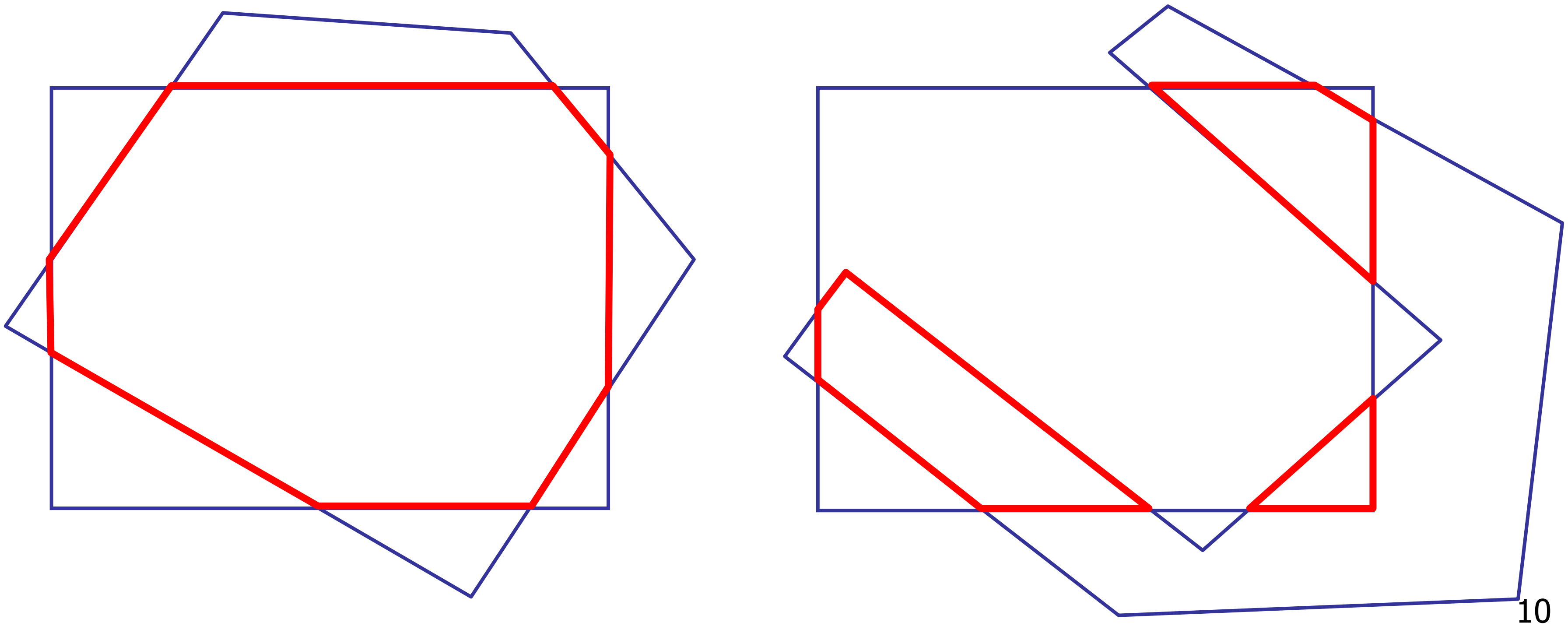
- outcodes
  - 4 flags encoding position of a point relative to top, bottom, left, and right boundary

- OC(**p1**)== 0 && OC(**p2**)==0
  - trivial accept
- (OC(**p1**) & OC(**p2**))!= 0
  - trivial reject



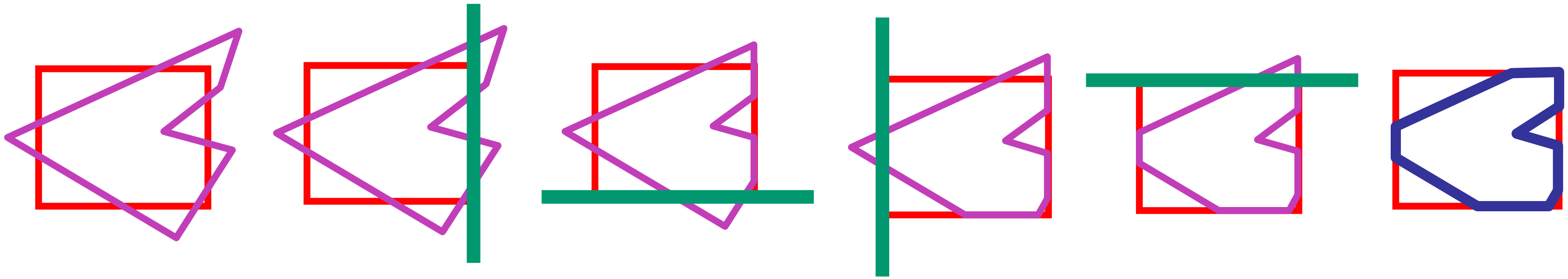
Review: Polygon Clipping

- not just clipping all boundary lines
  - may have to introduce new line segments



Review: Sutherland-Hodgeman Clipping

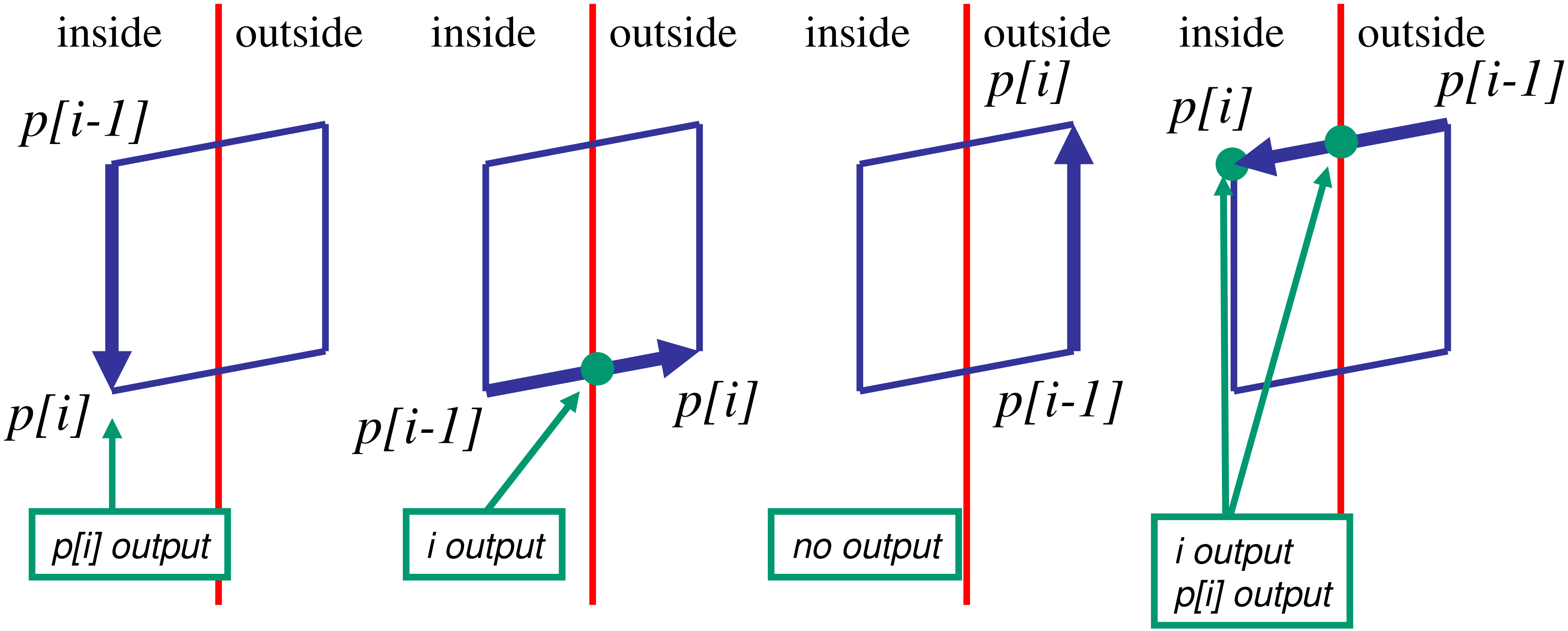
- for each viewport edge
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



- for each polygon vertex
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?

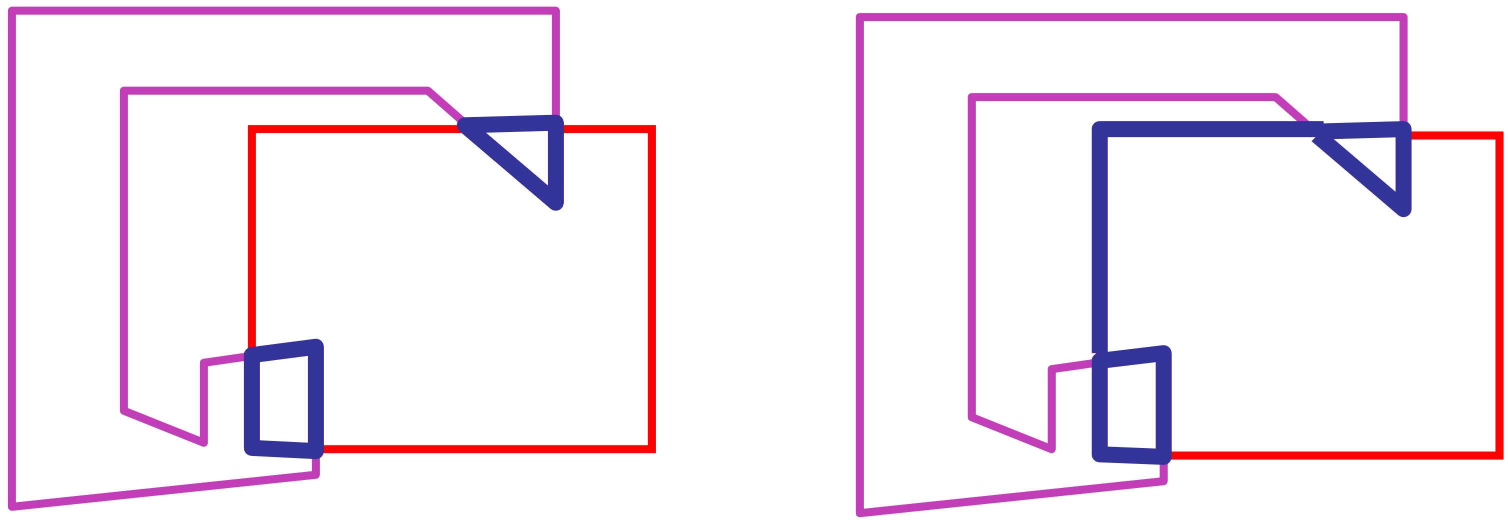
Review: Sutherland-Hodgeman Clipping

- edge from *p[i-1]* to *p[i]* has four cases
  - decide what to add to output vertex list

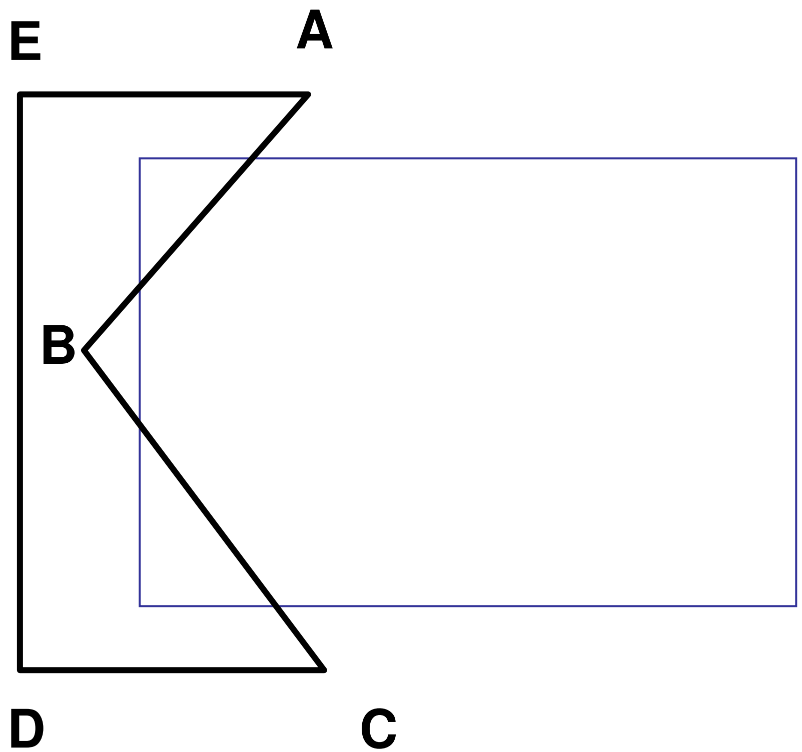


Clarification: Degenerate Edges

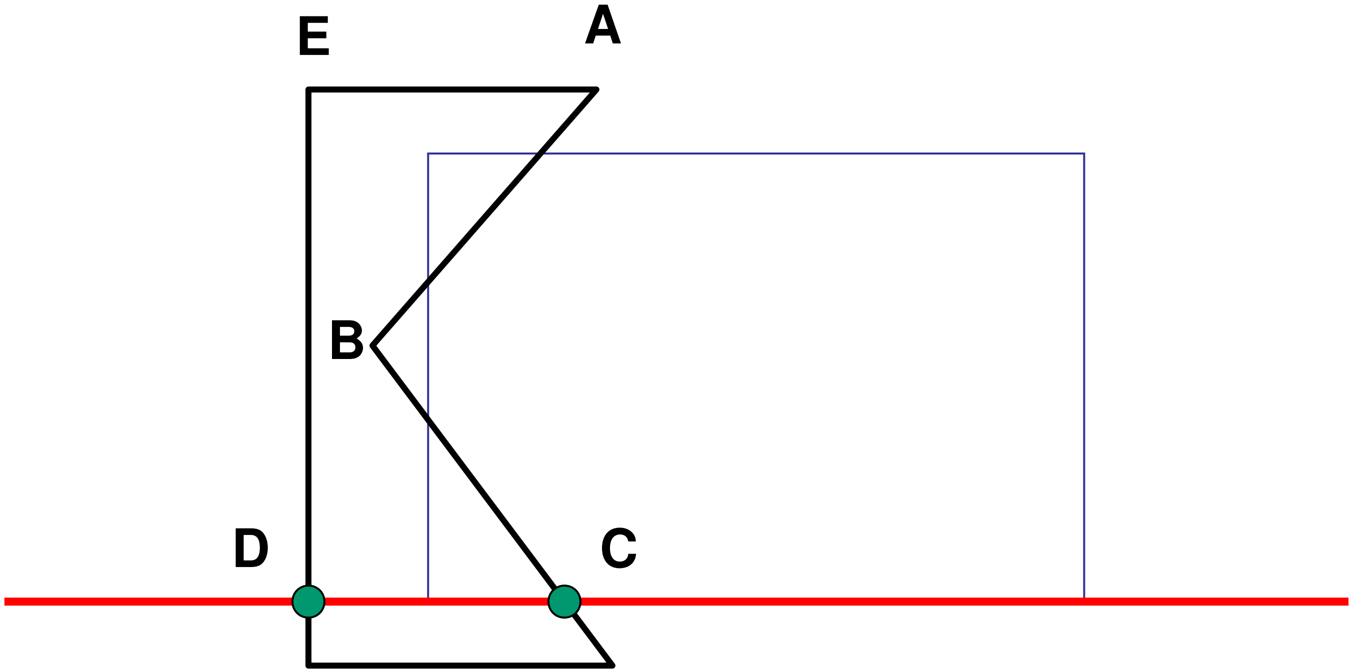
- Q from last time: how does S-H know that there are two disconnected polygons if all it has is a vertex list?
- A: end up with one connected polygon that has degenerate edges



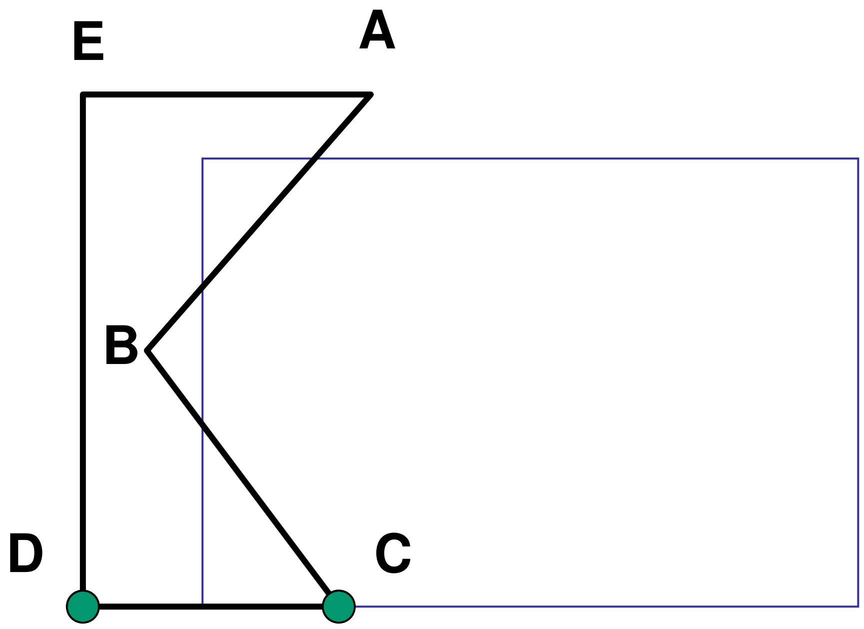
Clarification: Degenerate Edges



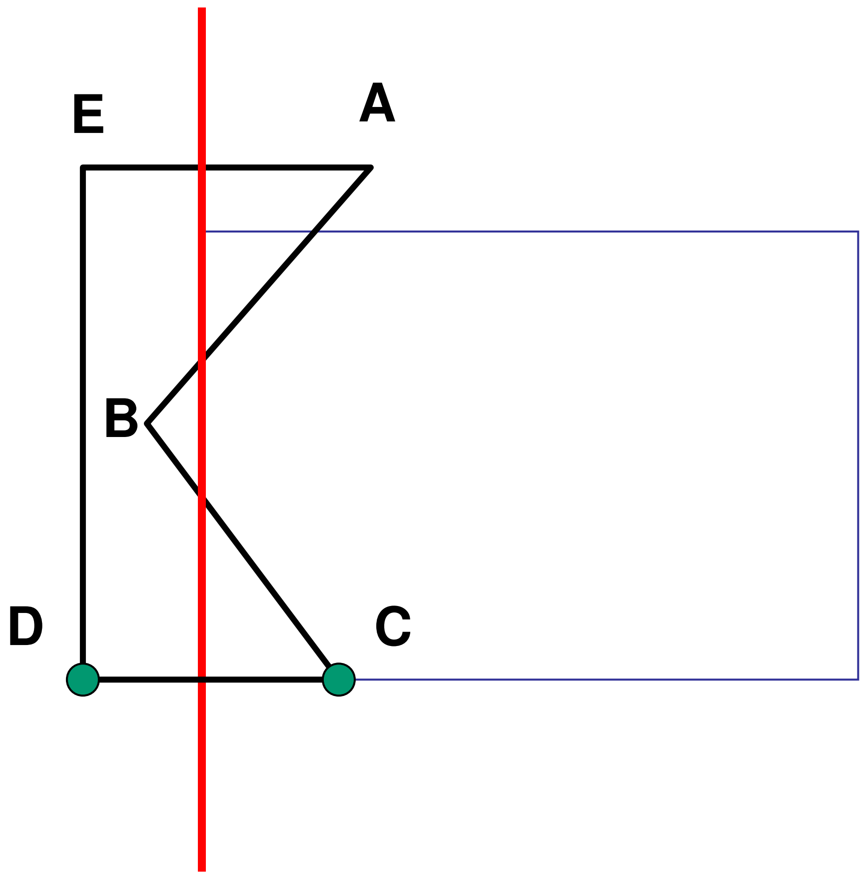
Clarification: Degenerate Edges



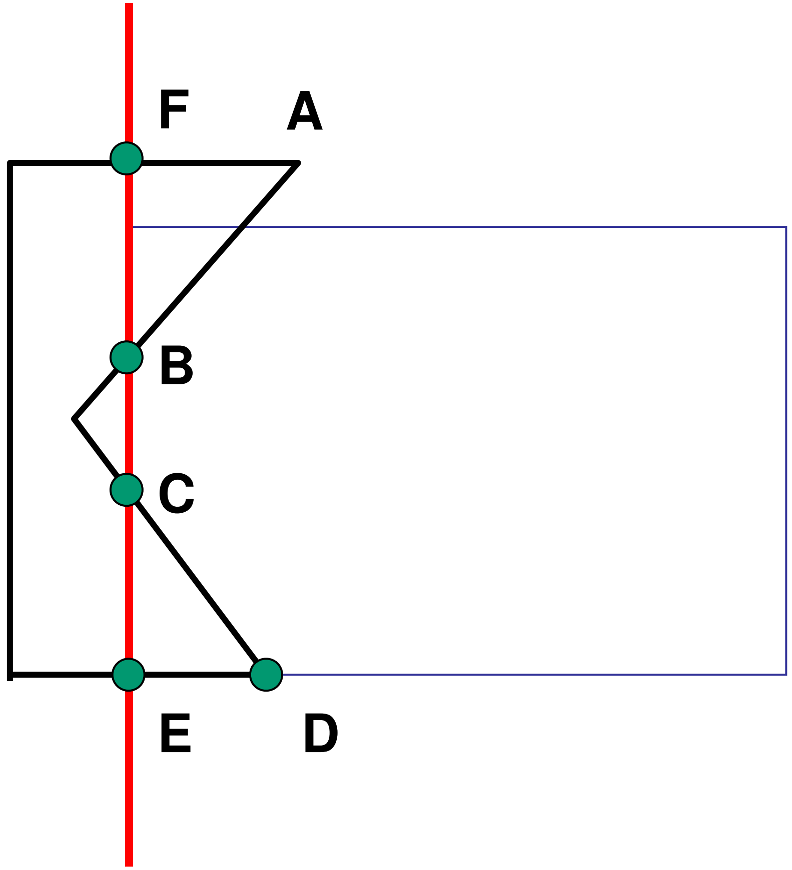
Clarification: Degenerate Edges



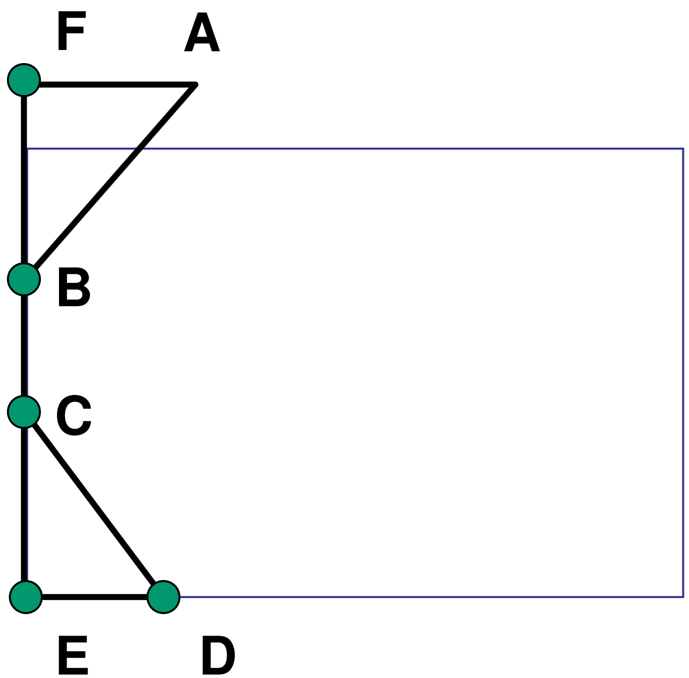
Clarification: Degenerate Edges



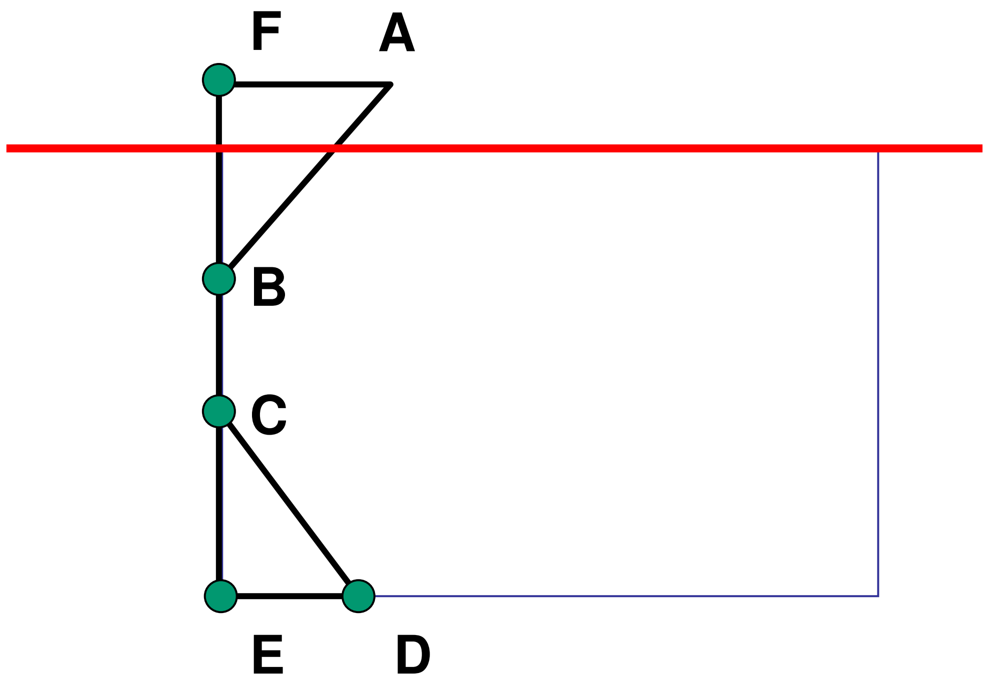
Clarification: Degenerate Edges



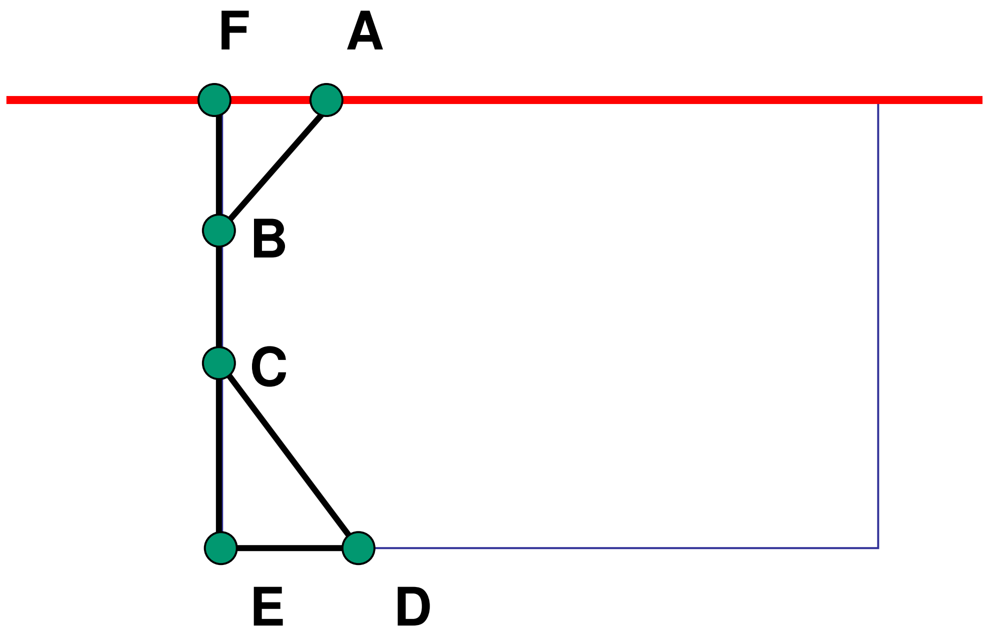
Clarification: Degenerate Edges



Clarification: Degenerate Edges



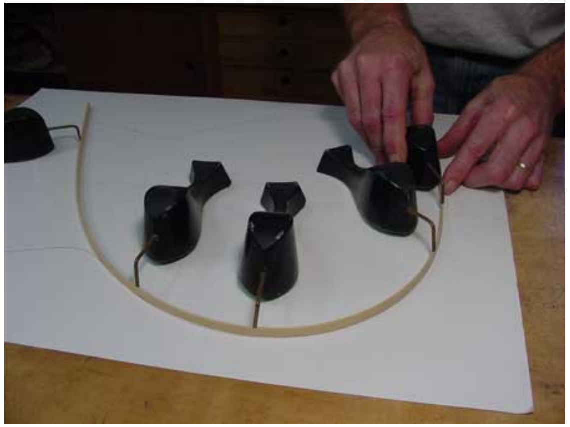
Clarification: Degenerate Edges



Review: Splines

- *spline* is parametric curve defined by *control points*
  - *knots:* control points that lie on curve
  - engineering drawing: spline was flexible wood, control points were physical weights

A Duck (weight)



Ducks trace out curve

Review: Hermite Spline

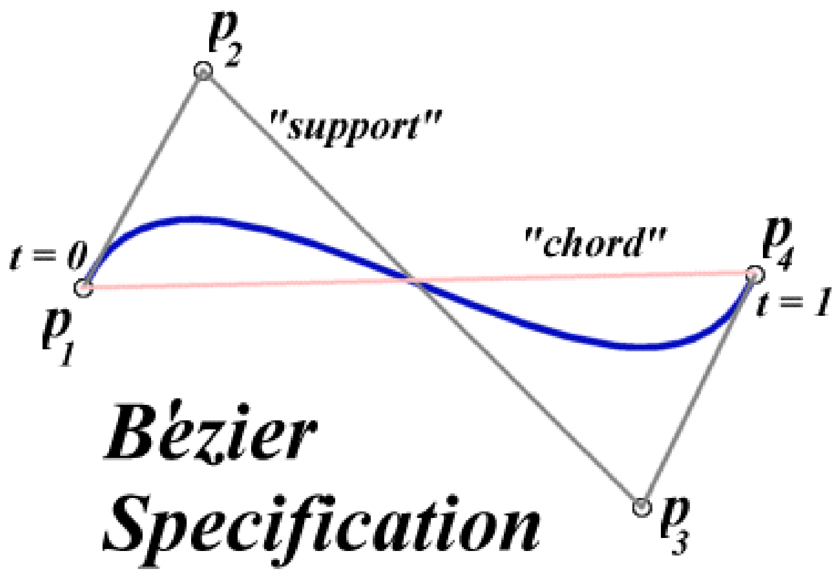
- user provides
  - endpoints
  - derivatives at endpoints

$$x = \begin{bmatrix} x_1 & x_0 & x_1' & x_0' \end{bmatrix} \begin{bmatrix} -2 & 3 & 0 & 0 \\ 2 & -3 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

Review: Bézier Curves

- four control points, two of which are knots
  - more intuitive definition than derivatives
- curve will always remain within convex hull (bounding region) defined by control points

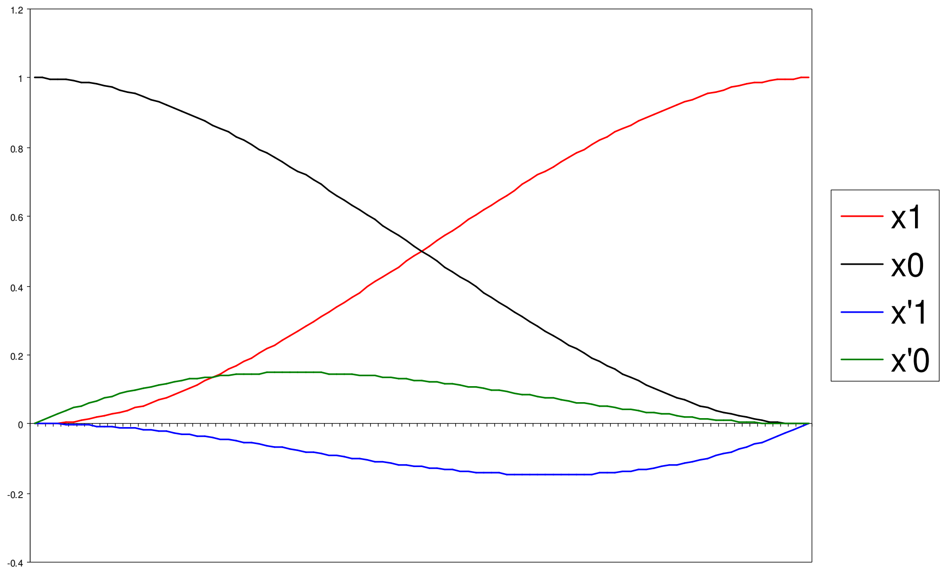
Hermite Specification



Bézier Specification

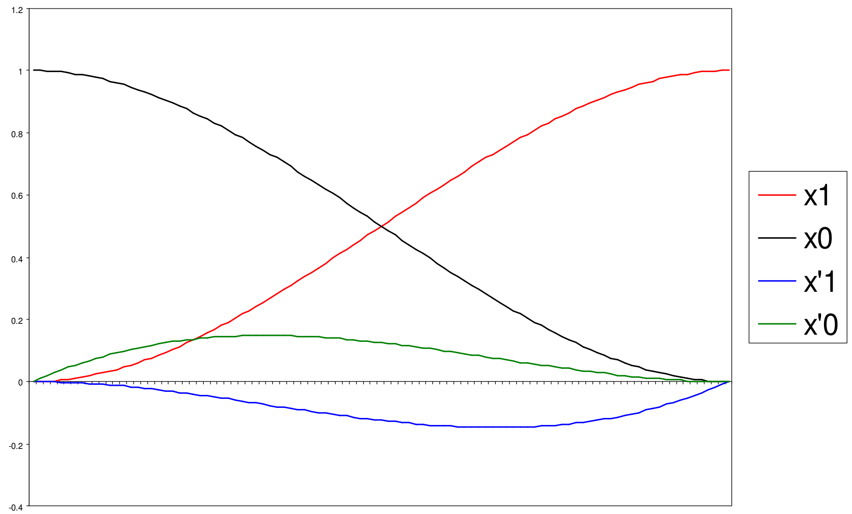
Review: Basis Functions

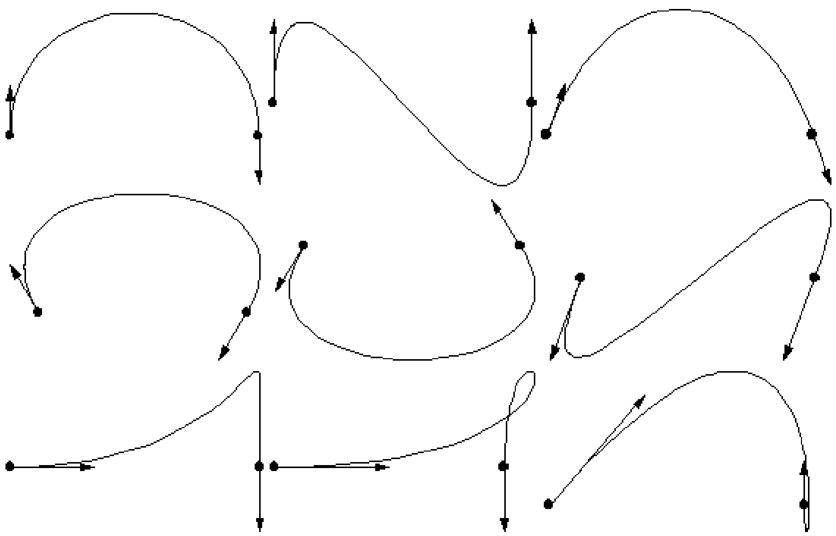
- point on curve obtained by multiplying each control point by some basis function and summing



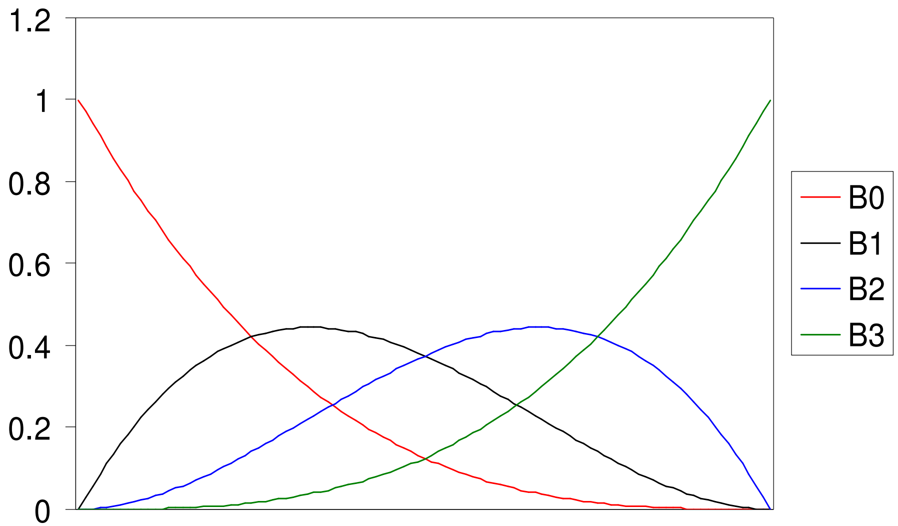
Review: Comparing Hermite and Bézier

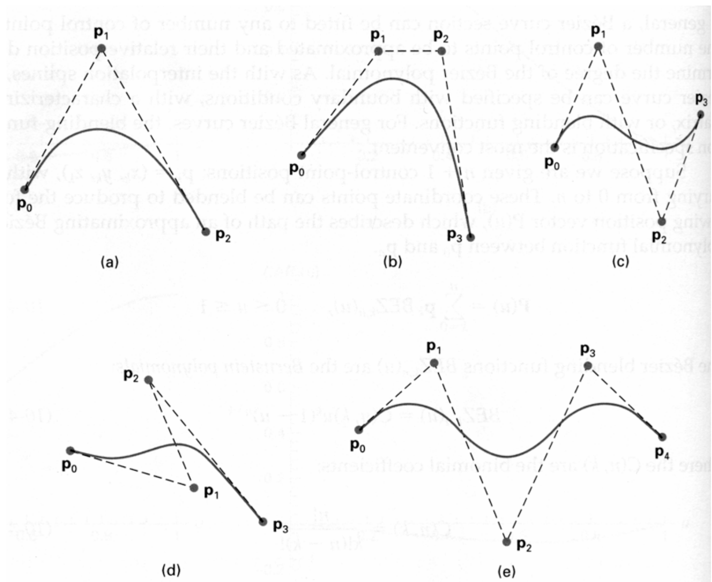
Hermite





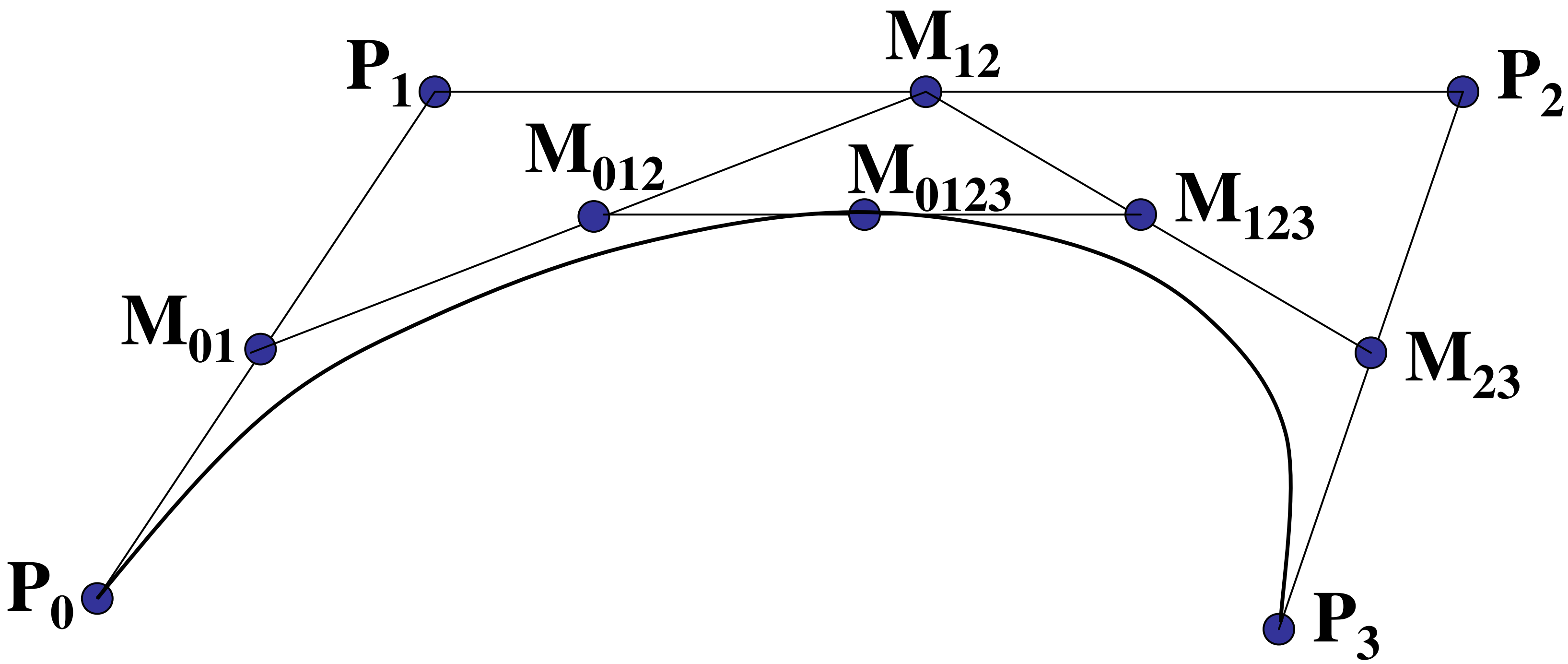
Bézier





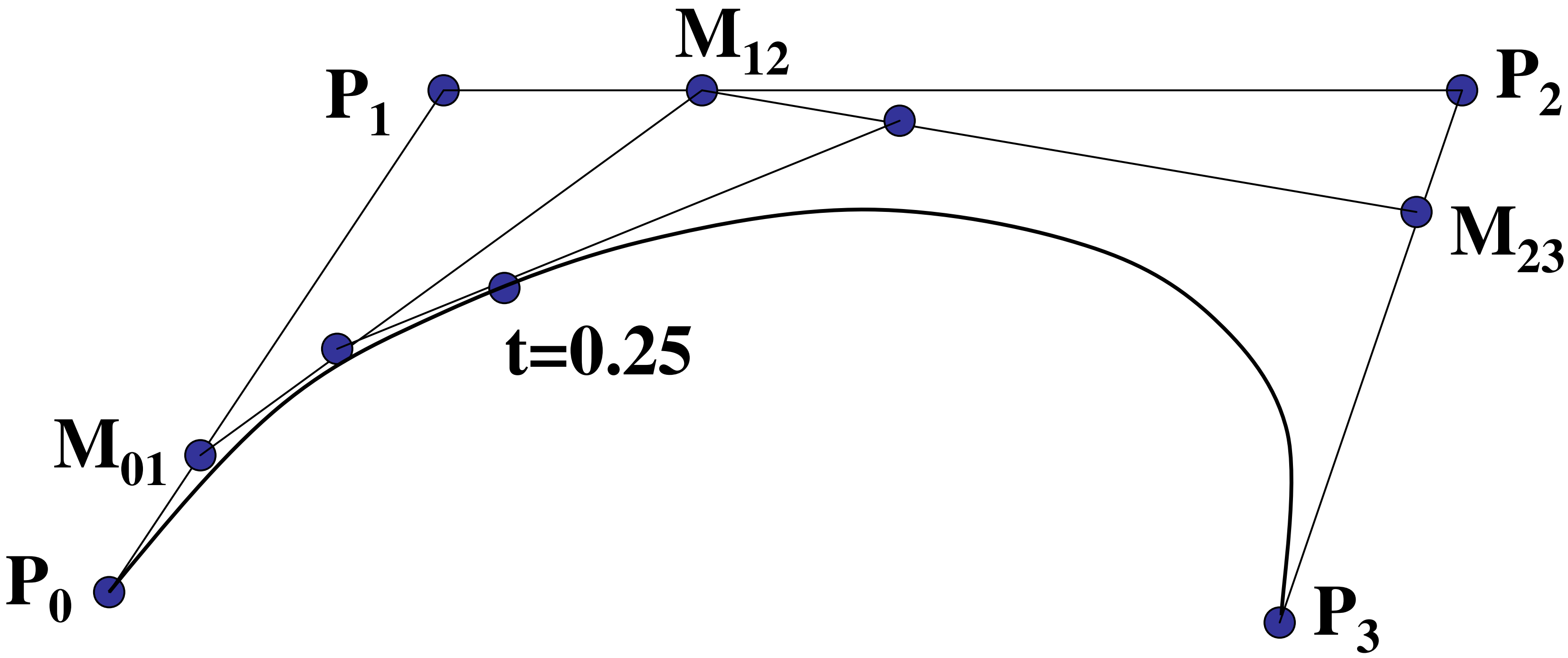
Review: Sub-Dividing Bézier Curves

- find the midpoint of the line joining M_{012}, M_{123}. call it M_{0123}



Review: de Casteljau's Algorithm

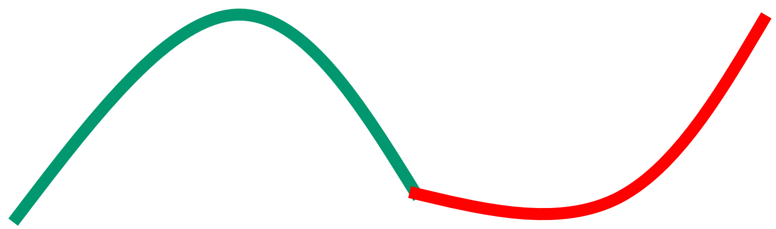
- can find the point on Bézier curve for any parameter value *t* with similar algorithm
  - for *t=0.25*, instead of taking midpoints take points 0.25 of the way



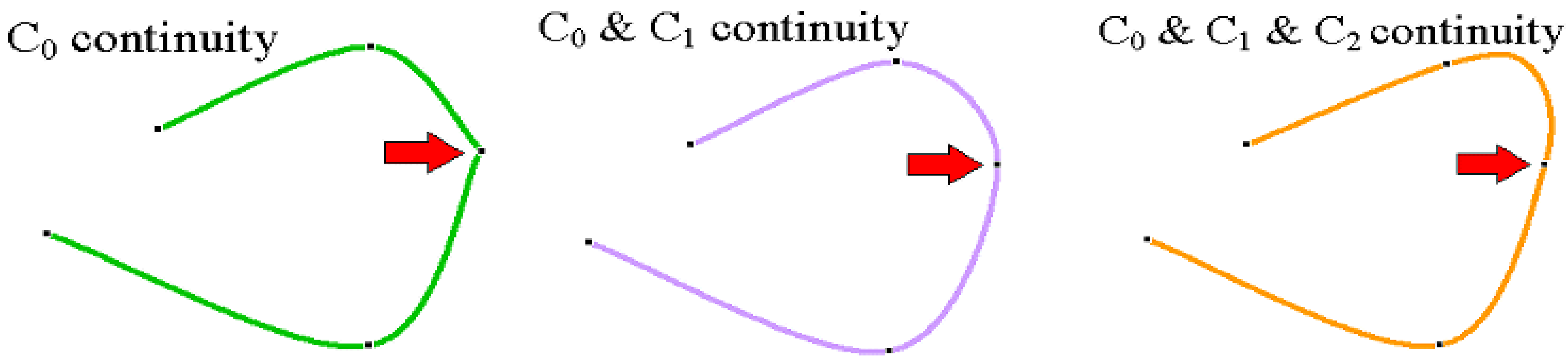
demo: www.saltire.com/applets/advanced_geometry/spline/spline.htm

Review: Continuity

- piecewise Bézier: no continuity guarantees

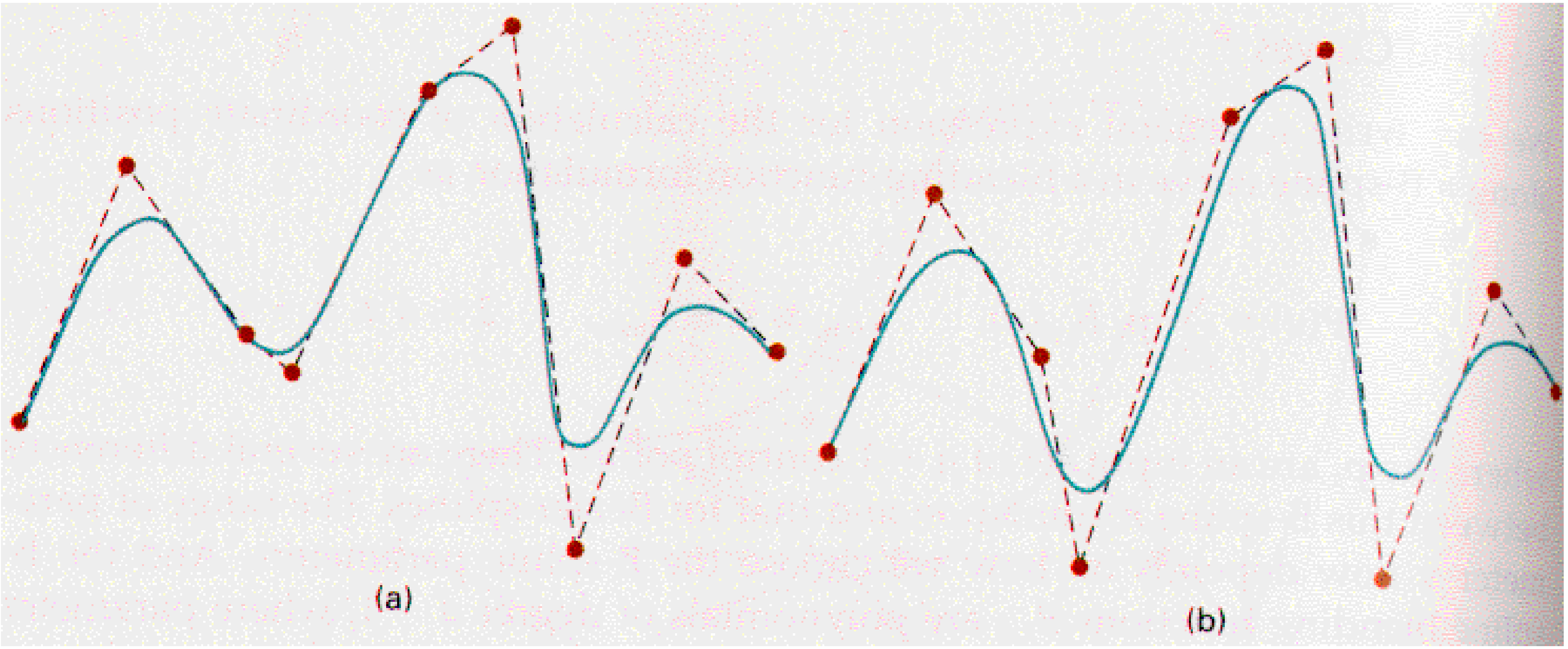


- continuity definitions
  - C^0: share join point
  - C^1: share continuous derivatives
  - C^2: share continuous second derivatives



Review: B-Spline

- C_0, C_1, and C_2 continuous
- piecewise: locality of control point influence



(a)

(b)

Picking

Reading

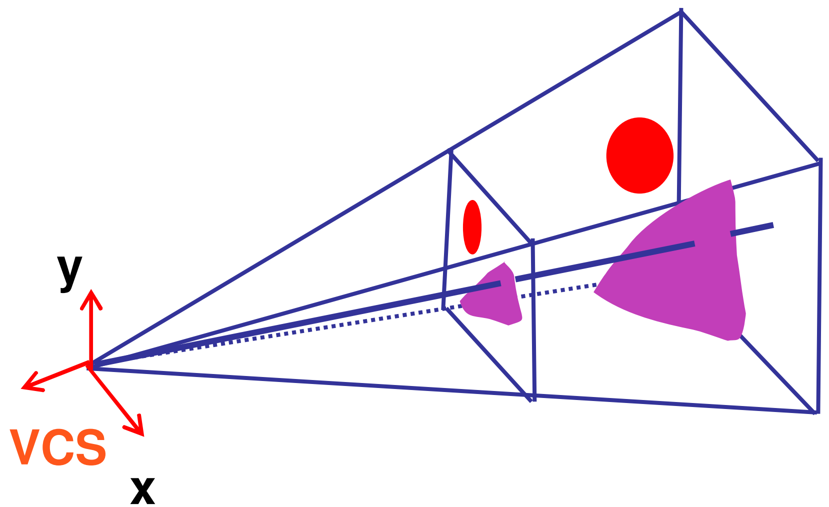
- Red Book
  - Selection and Feedback Chapter
    - all
  - Now That You Know Chapter
    - only Object Selection Using the Back Buffer

Interactive Object Selection

- move cursor over object, click
  - how to decide what is below?
- ambiguity
  - many 3D world objects map to same 2D point
- four common approaches
  - manual ray intersection
  - bounding extents
  - backbuffer color coding
  - selection region with hit list

Manual Ray Intersection

- do all computation at application level
  - map selection point to a ray
  - intersect ray with all objects in scene.
- advantages
  - no library dependence

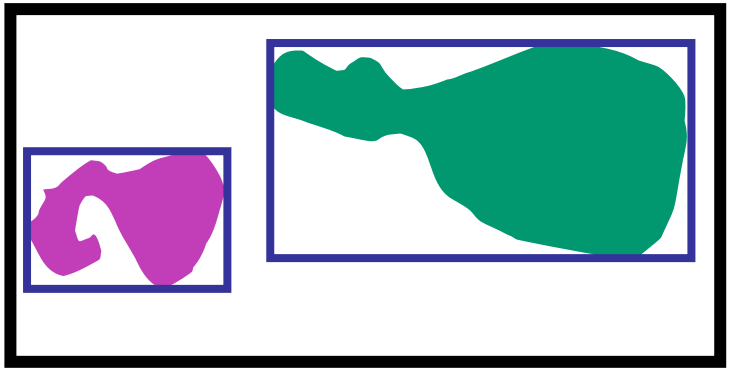


Manual Ray Intersection

- do all computation at application level
  - map selection point to a ray
  - intersect ray with all objects in scene.
- advantages
  - no library dependence
- disadvantages
  - difficult to program
  - slow: work to do depends on total number and complexity of objects in scene

Bounding Extents

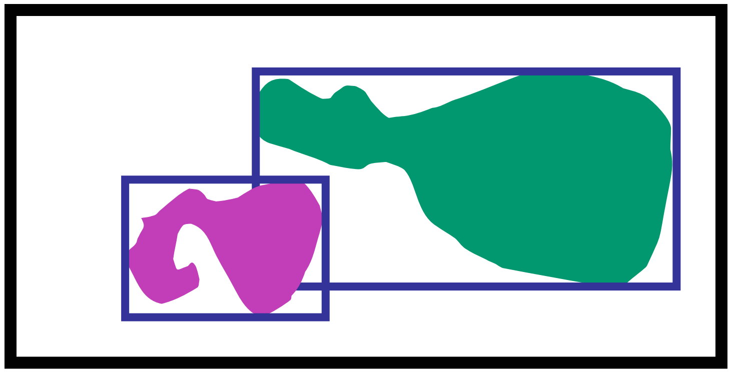
- keep track of axis-aligned bounding rectangles



- advantages
  - conceptually simple
  - easy to keep track of boxes in world space

Bounding Extents

- disadvantages
  - low precision
  - must keep track of object-rectangle relationship
- extensions
  - do more sophisticated bound bookkeeping
    - first level: box check. second level: object check

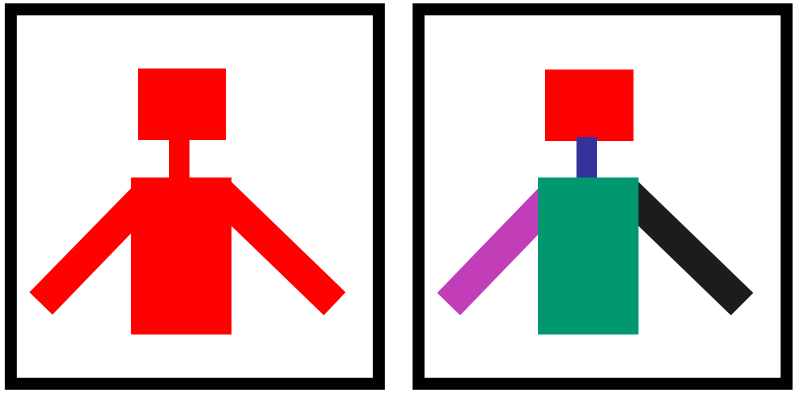


Backbuffer Color Coding

- use backbuffer for picking
  - create image as computational entity
  - never displayed to user
- redraw all objects in backbuffer
  - turn off shading calculations
  - set unique color for each pickable object
    - store in table
  - read back pixel at cursor location
    - check against table

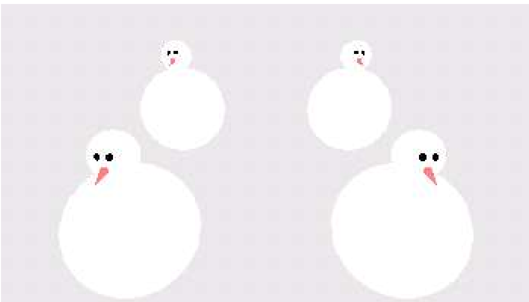
Backbuffer Color Coding

- advantages
  - conceptually simple
  - variable precision
- disadvantages
  - introduce 2x redraw delay
  - backbuffer readback very slow

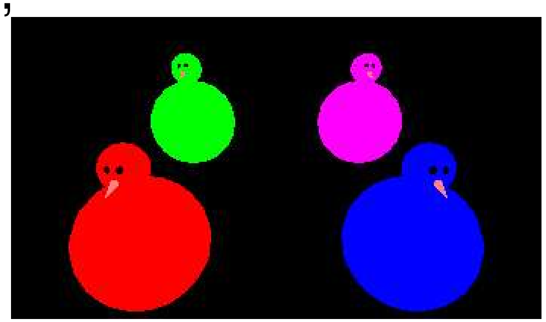


Backbuffer Example

```
glColor3f(1.0f, 1.0f, 1.0f);
for(int i = 0; i < 2; i++)
    for(int j = 0; j < 2; j++) {
      glPushMatrix();
      glTranslatef(i*3.0,0,-j * 3.0);
      glColor3f(1.0f, 1.0f, 1.0f);
      glCallList(snowman_display_list);
      glPopMatrix();
  }
```



```
for(int i = 0; i < 2; i++)
  for(int j = 0; j < 2; j++) {
    glPushMatrix();
    switch (i*2+j) {
      case 0: glColor3ub(255,0,0);break;
      case 1: glColor3ub(0,255,0);break;
      case 2: glColor3ub(0,0,255);break;
      case 3: glColor3ub(250,0,250);break;
    }
    glTranslatef(i*3.0,0,-j * 3.0)
    glCallList(snowman_display_list);
    glPopMatrix();
}
```



http://www.lighthouse3d.com/opengl/picking/

Select/Hit

- use small region around cursor for viewport
- assign per-object integer keys (names)
- redraw in special mode
- store hit list of objects in region
- examine hit list

- OpenGL support

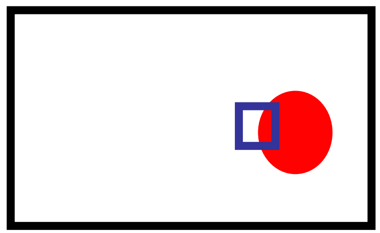
Viewport

- small rectangle around cursor
  - change coord sys so fills viewport

- why rectangle instead of point?
  - people aren’t great at positioning mouse
    - Fitts’s Law: time to acquire a target is function of the distance to and size of the target
  - allow several pixels of slop

Viewport

- tricky to compute
  - invert viewport matrix, set up new orthogonal projection
- simple utility command
  - gluPickMatrix(x,y,w,h,viewport)
    - x,y: cursor point
    - w,h: sensitivity/slop (in pixels)
  - push old setup first, so can pop it later



Render Modes

- glRenderMode(mode)
  - GL_RENDER: normal color buffer
    - default
  - GL_SELECT: selection mode for picking
  - (GL_FEEDBACK: report objects drawn)

Name Stack

- "names" are just integers

  glInitNames()

- flat list

  glLoadName(name)

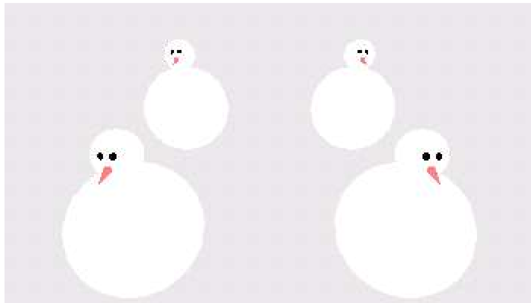
- or hierarchy supported by stack

  glPushName(name), glPopName

  - can have multiple names per object

Hierarchical Names Example

```
for(int i = 0; i < 2; i++) {
  glPushName(i);
  for(int j = 0; j < 2; j++) {
    glPushMatrix();
    glPushName(j);
    glTranslatef(i*10.0,0,j * 10.0);
      glPushName(HEAD);
      glCallList(snowManHeadDL);
      glLoadName(BODY);
      glCallList(snowManBodyDL);
      glPopName();
    glPopName();
    glPopMatrix();
  }
  glPopName();
}
```



http://www.lighthouse3d.com/opengl/picking/

Hit List

- glSelectBuffer(buffersize, *buffer)
  - where to store hit list data
- on hit, copy entire contents of name stack to output buffer.
- hit record
  - number of names on stack
  - minimum and minimum depth of object vertices
    - depth lies in the z-buffer range [0,1]
    - multiplied by 2^32 -1 then rounded to nearest int

Integrated vs. Separate Pick Function

- integrate: use same function to draw and pick
  - simpler to code
  - name stack commands ignored in render mode
- separate: customize functions for each
  - potentially more efficient
  - can avoid drawing unpickable objects

Select/Hit

- advantages
  - faster
    - OpenGL support means hardware accel
    - only do clipping work, no shading or rasterization
  - flexible precision
    - size of region controllable
  - flexible architecture
    - custom code possible, e.g. guaranteed frame rate
- disadvantages
  - more complex

Hybrid Picking

- select/hit approach: fast, coarse
  - object-level granularity
- manual ray intersection: slow, precise
  - exact intersection point
- hybrid: both speed and precision
  - use select/hit to find object
  - then intersect ray with that object

OpenGL Picking Hints

- gluUnproject
  - transform window coordinates to object coordinates given current projection and modelview matrices
  - use to create ray into scene from cursor location
  - call gluUnProject twice with same (x,y) mouse location
    - z = near: (x,y,0)
    - z = far: (x,y,1)
    - subtract near result from far result to get direction vector for ray
- use this ray for line/polygon intersection

Picking and P4

- you must implement true 3D picking!
  - you will not get credit if you just use 2D information

Collision Detection

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

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

From Simple to Complex

- boundary check
  - perimeter of world vs. viewpoint or objects
    - 2D/3D absolute coordinates for bounds
    - simple point in space for viewpoint/objects
- set of fixed barriers
  - walls in maze game
    - 2D/3D absolute coordinate system
- set of moveable objects
  - one object against set of items
    - missile vs. several tanks
  - multiple objects against each other
    - punching game: arms and legs of players
    - room of bouncing balls

Naive General 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! $O(n^2)$

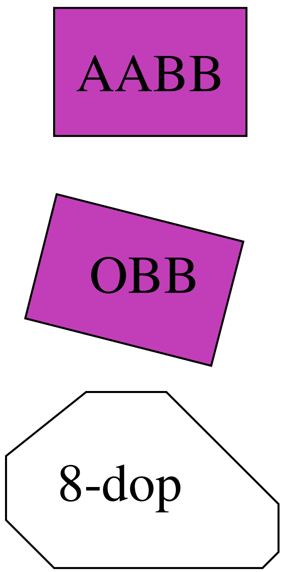
Choosing an Algorithm

- 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

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

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 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

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?

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

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

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?

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

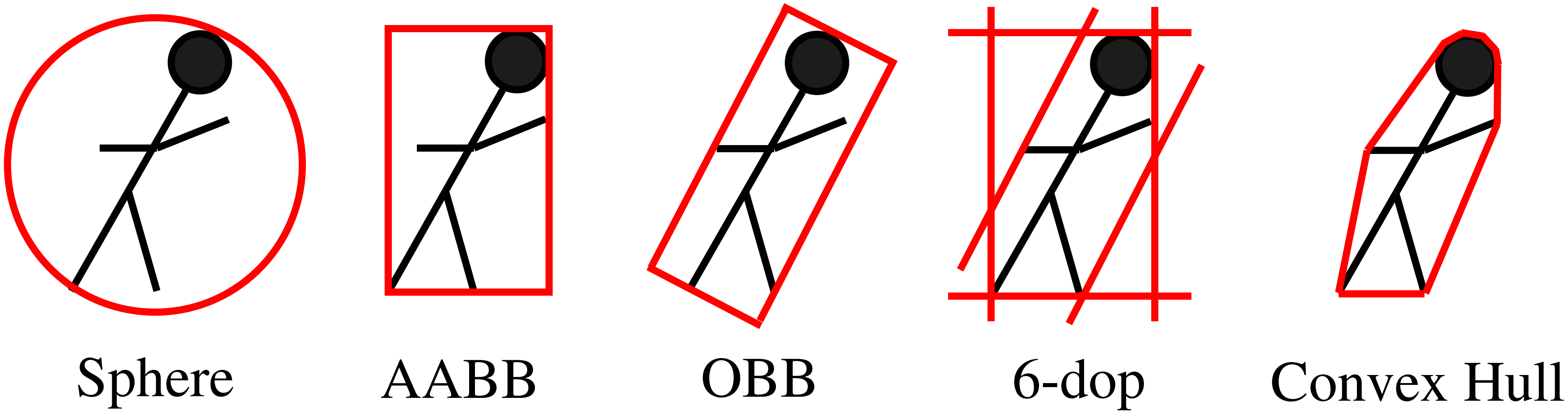
Collision Proxies

- 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 …

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

Trade-off in Choosing Proxies



increasing complexity & tightness of fit

decreasing cost of (overlap tests + proxy update)

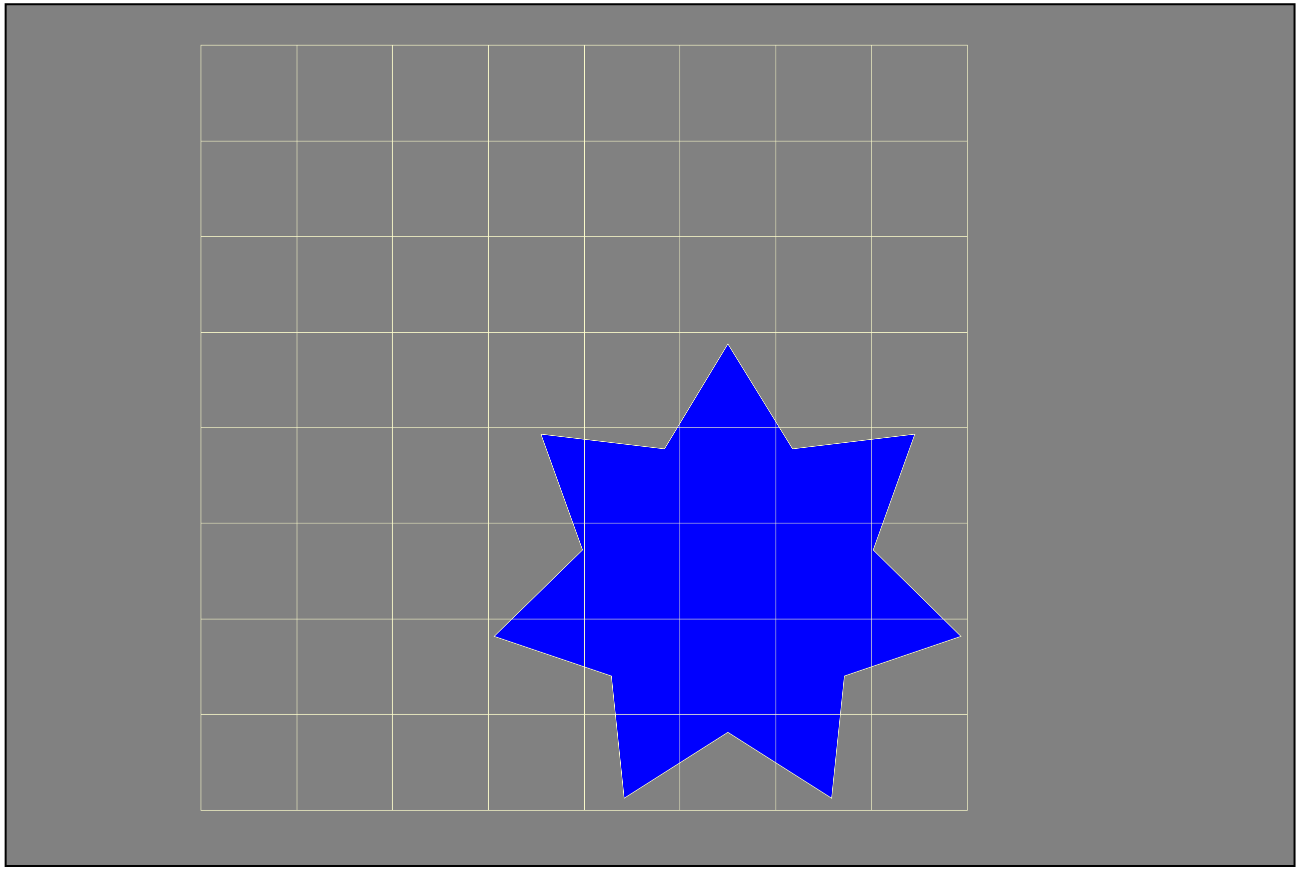
Pair Reduction

- 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…

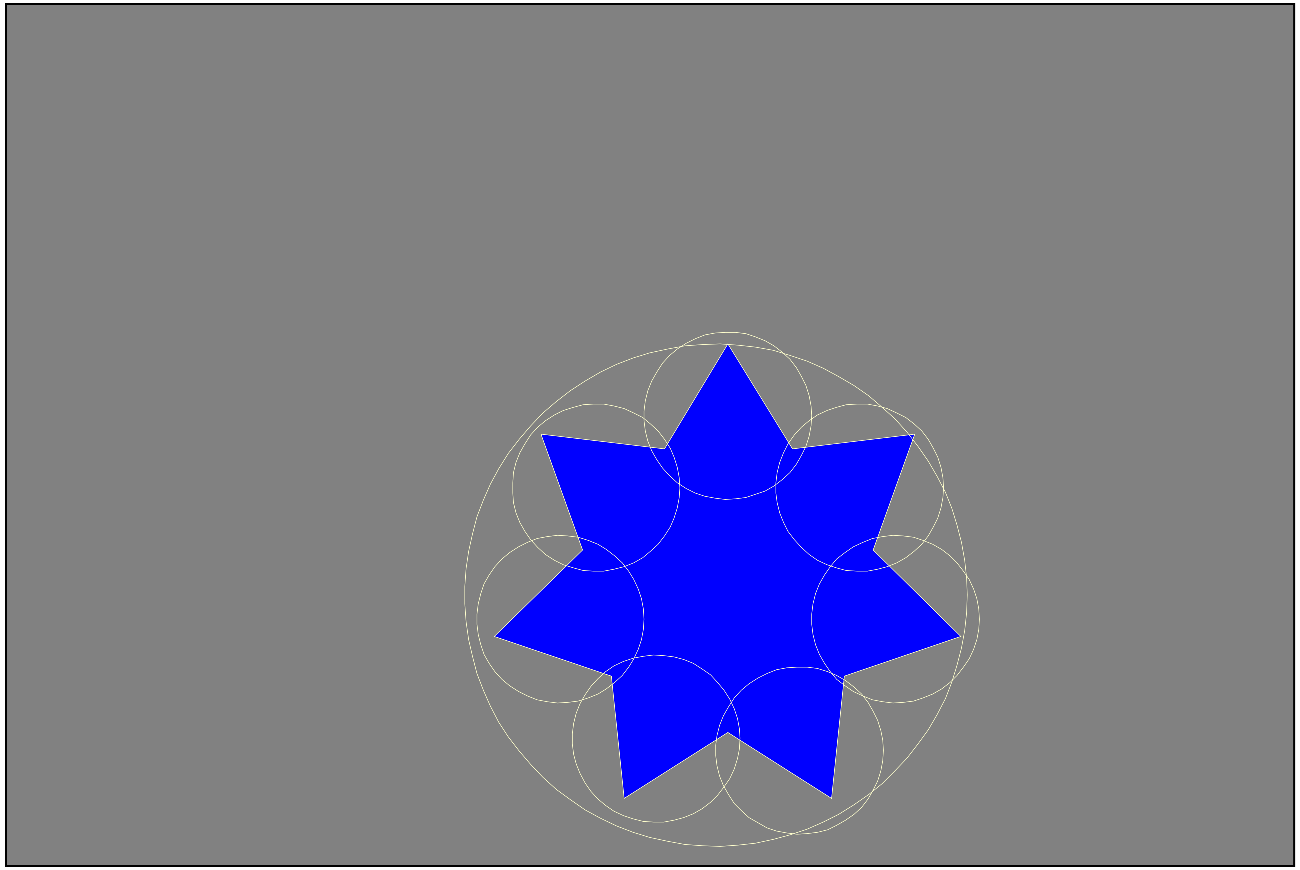
Spatial Data Structures

- 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

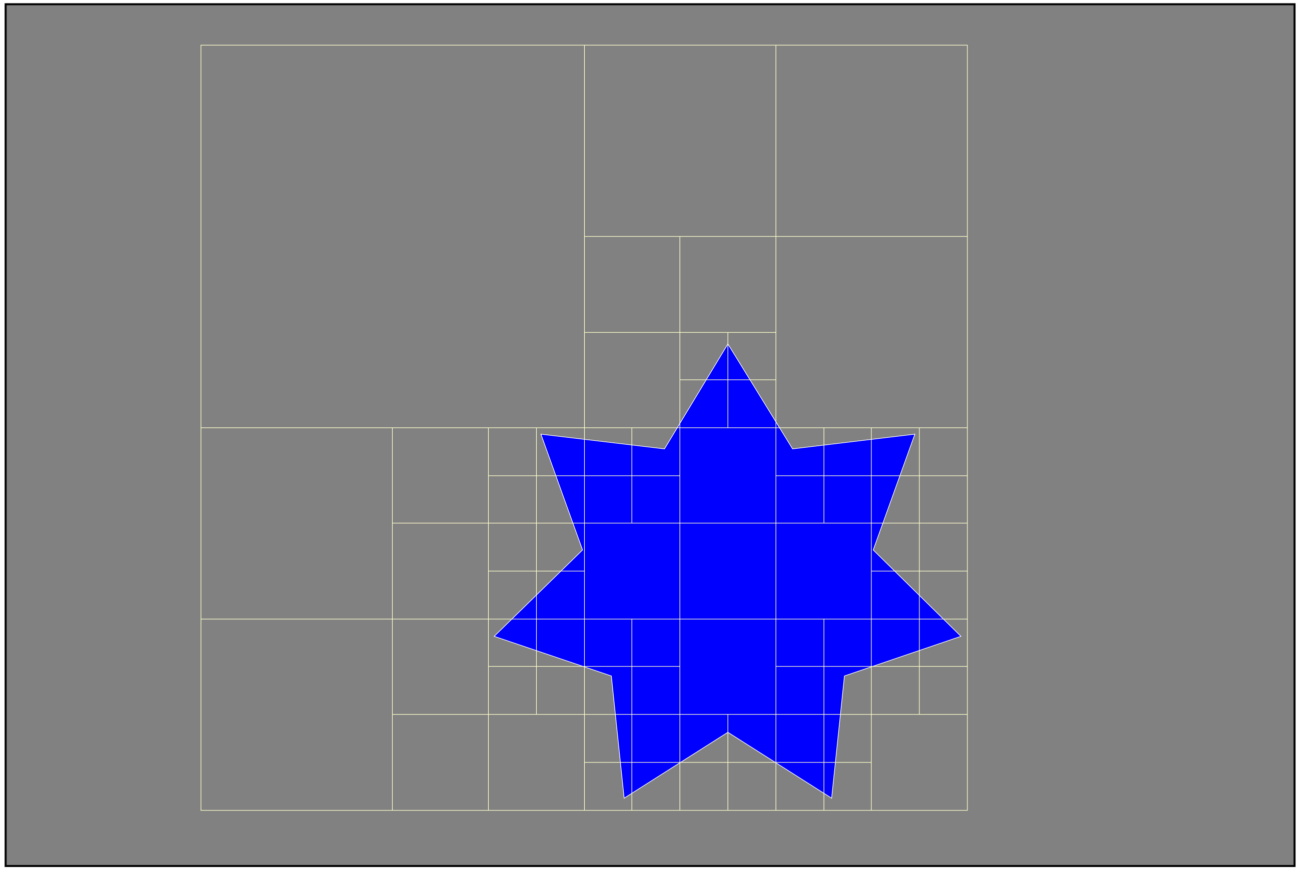
Uniform Grids



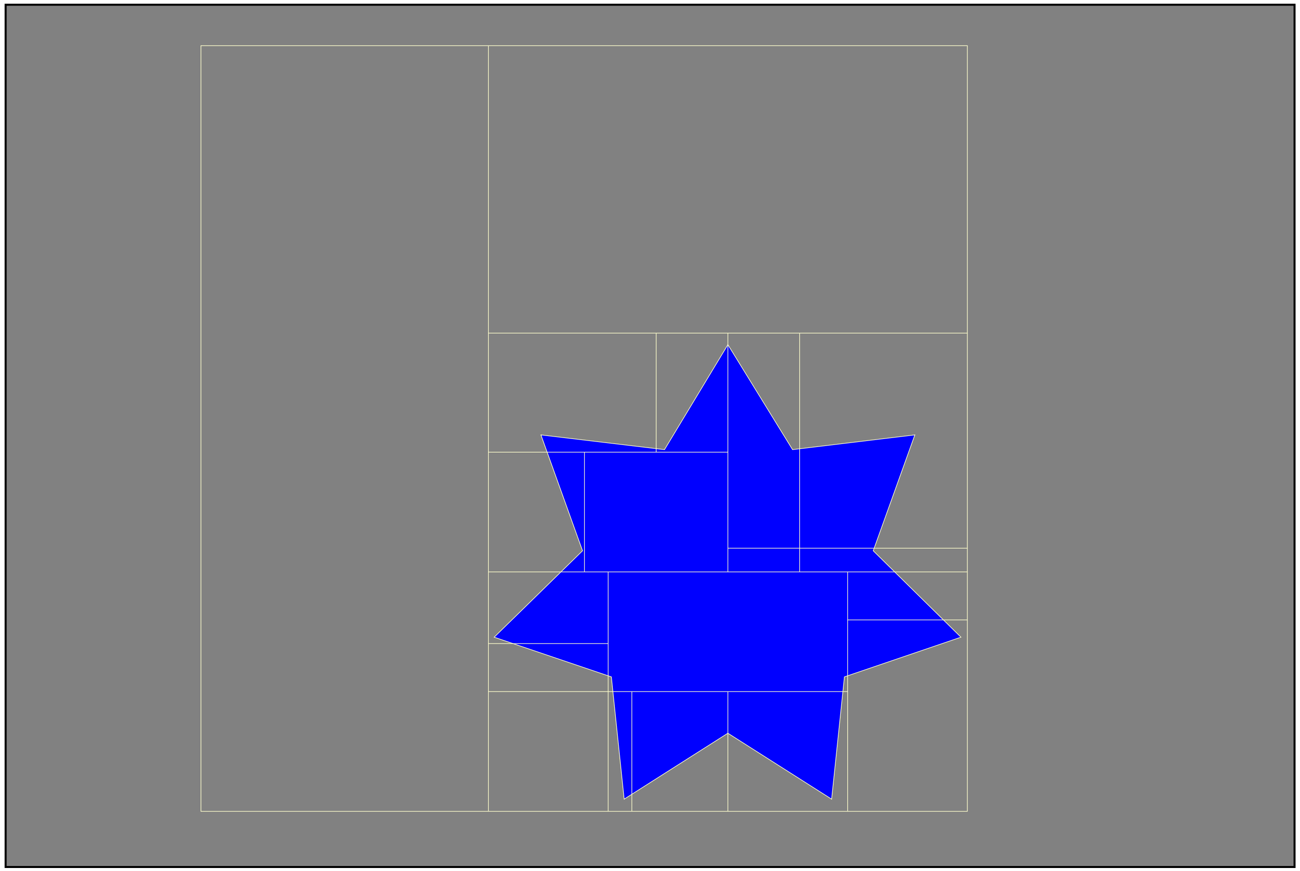
Bounding Volume Hierarchies



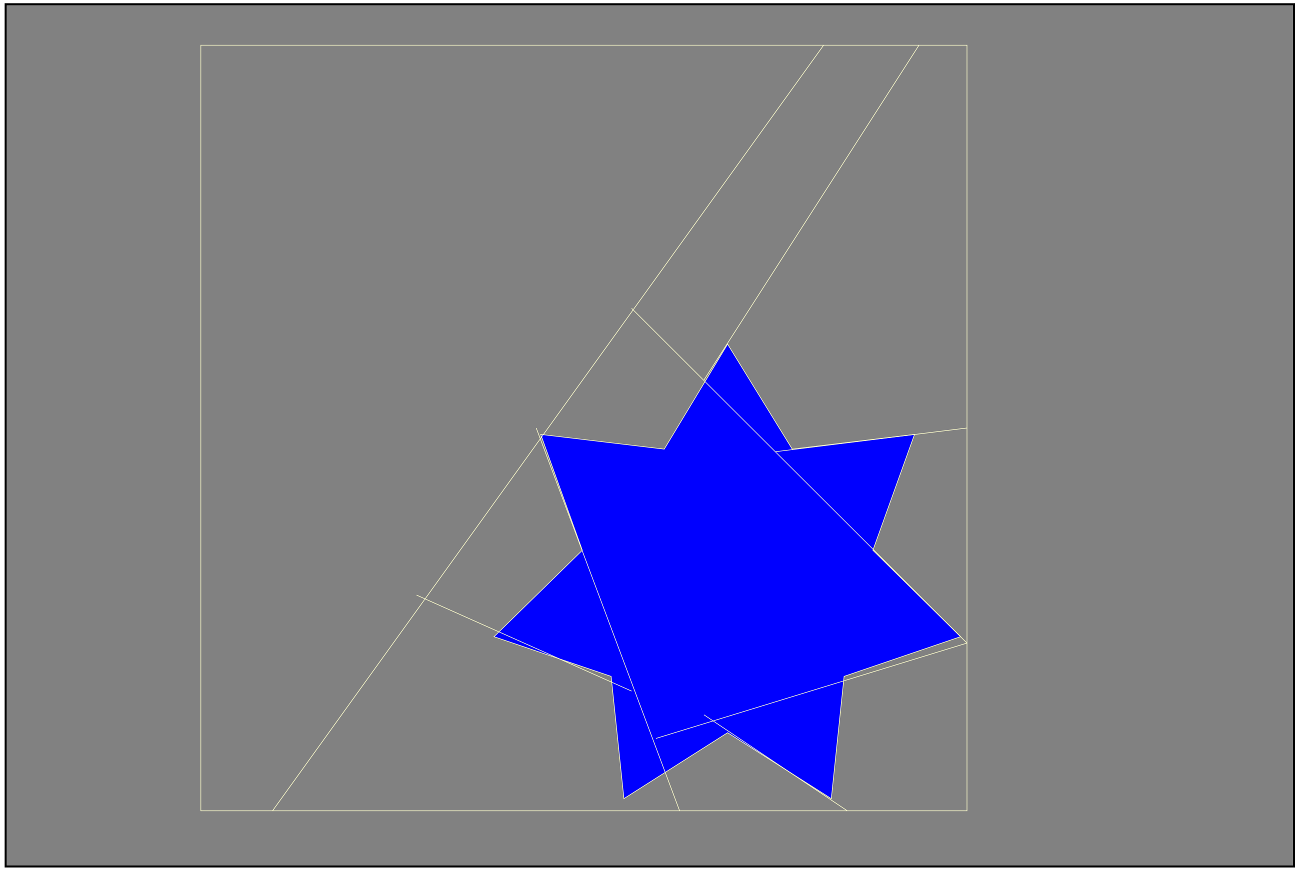
Octrees



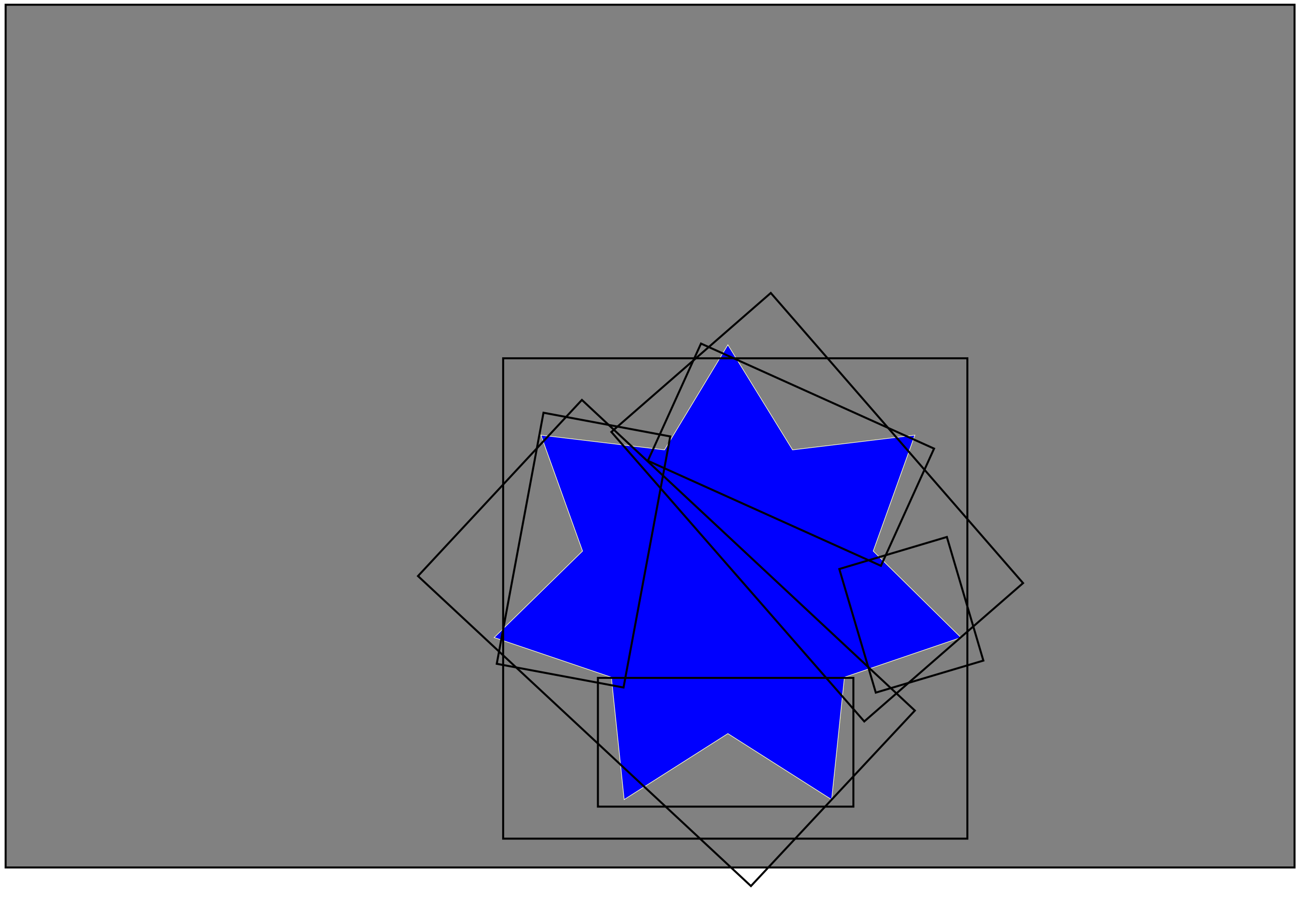
KD Trees



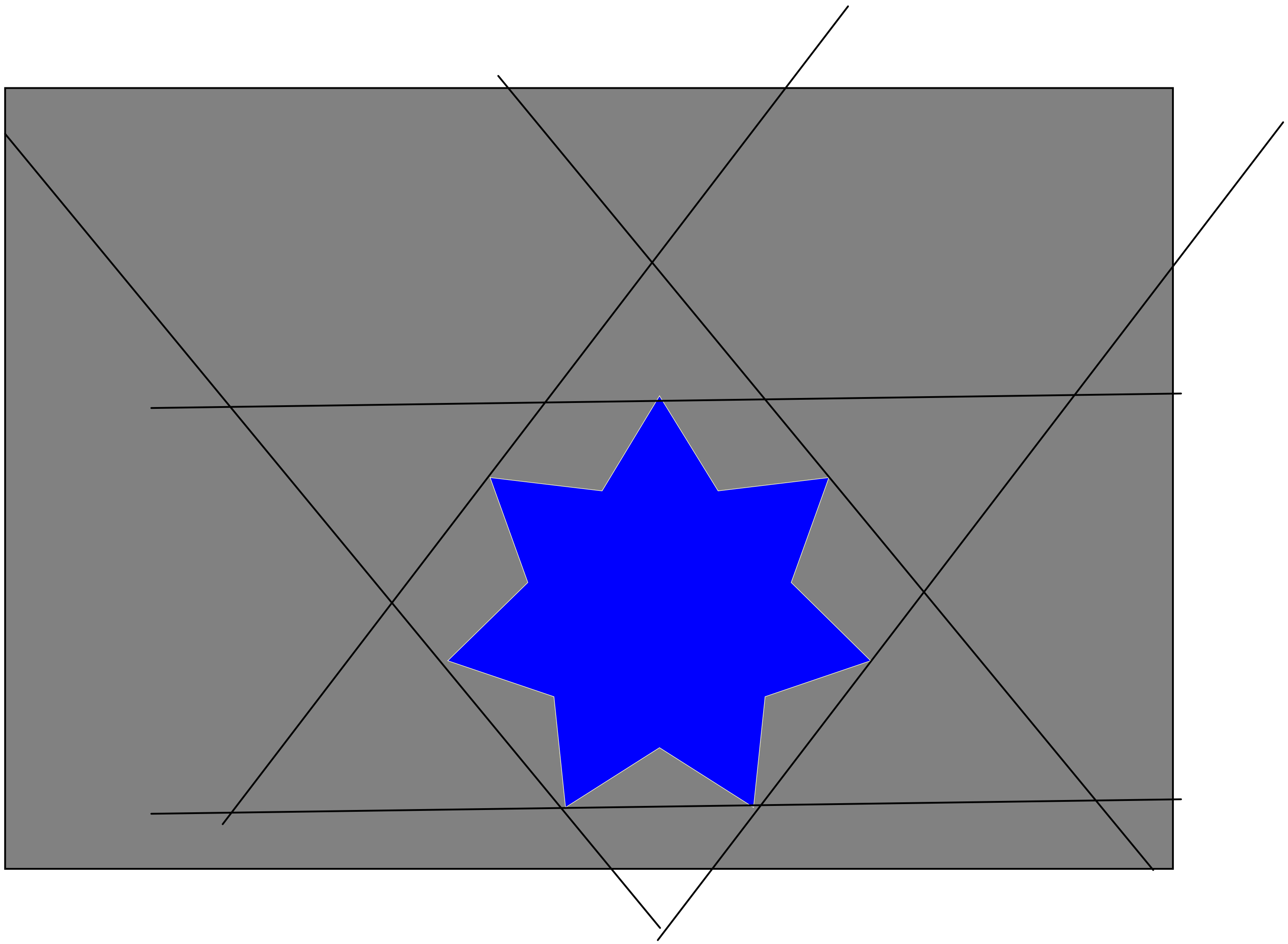
BSP Trees



OBB Trees



K-Dops



Testing BVH's

```
TestBVH(A,B) {
    if(not overlap(A_BV, B_BV) return FALSE;
    else if(isLeaf(A)) {
        if(isLeaf(B)) {
            for each triangle pair (T_a,T_b)
                if(overlap(T_a,T_b)) AddIntersectionToList();
        }
        else {
            for each child C_b of B
                TestBVH(A,C_b);
        }
    }
    else {
        for each child C_a of A
            TestBVH(C_a,B)
    }
}
```

Optimization Structures

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

Exploiting Coherence

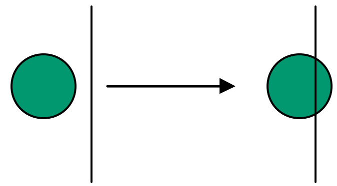
- player normally doesn’t move far between frames
- cells they intersected the last time are
  - probably the same cells they intersect now
  - or at least they are close
- aim is to track which cells the player is in without doing a full search each time
- easiest to exploit with a cell portal structure

Cell-Portal Collisions

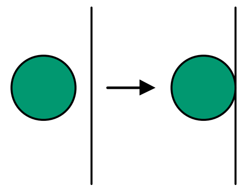
- keep track which cell/s player is currently intersecting
  - can have more than one if the player straddles a cell boundary
  - always use a proxy (bounding volume) for tracking cells
  - also keep track of which portals the player is straddling
- player can only enter new cell through portal
- on 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

Precise Collision Times

- generally a player will go from not intersecting to interpenetrating in the course of a frame



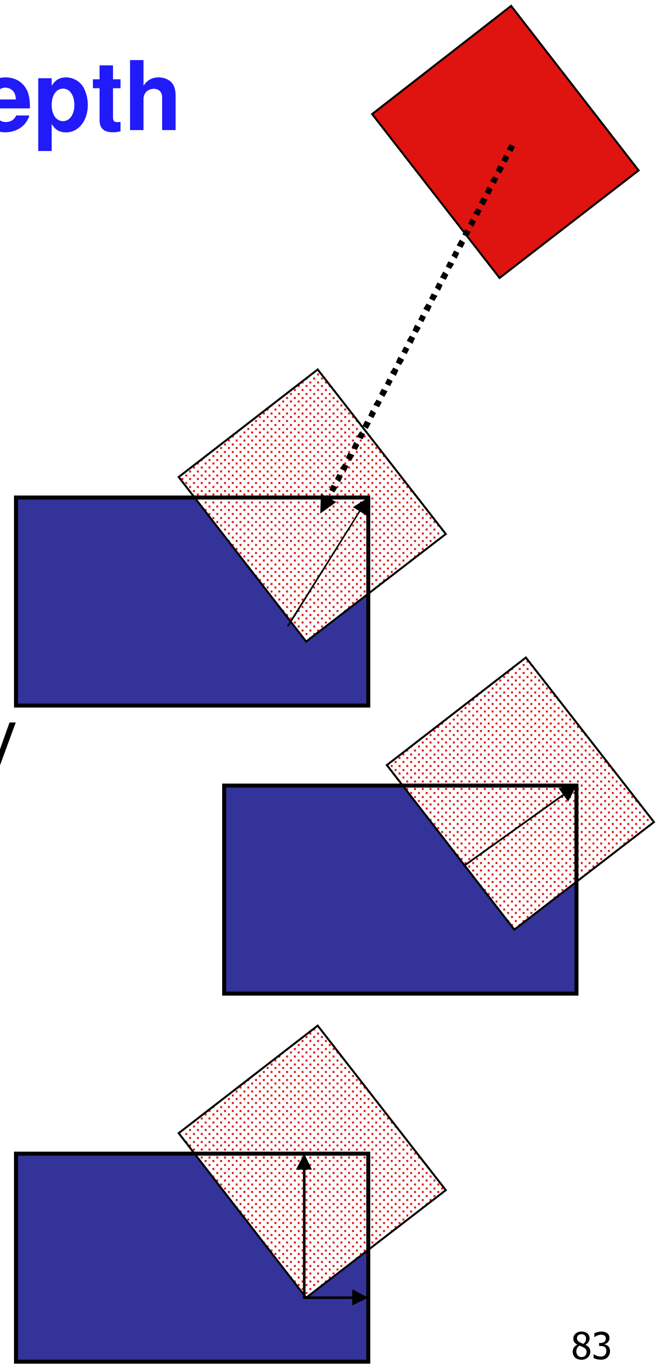
- we typically would like the exact collision time and place



  - 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)

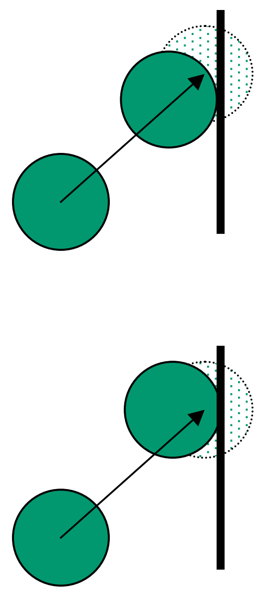
Defining Penetration Depth

- 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



Hacked Clean Up

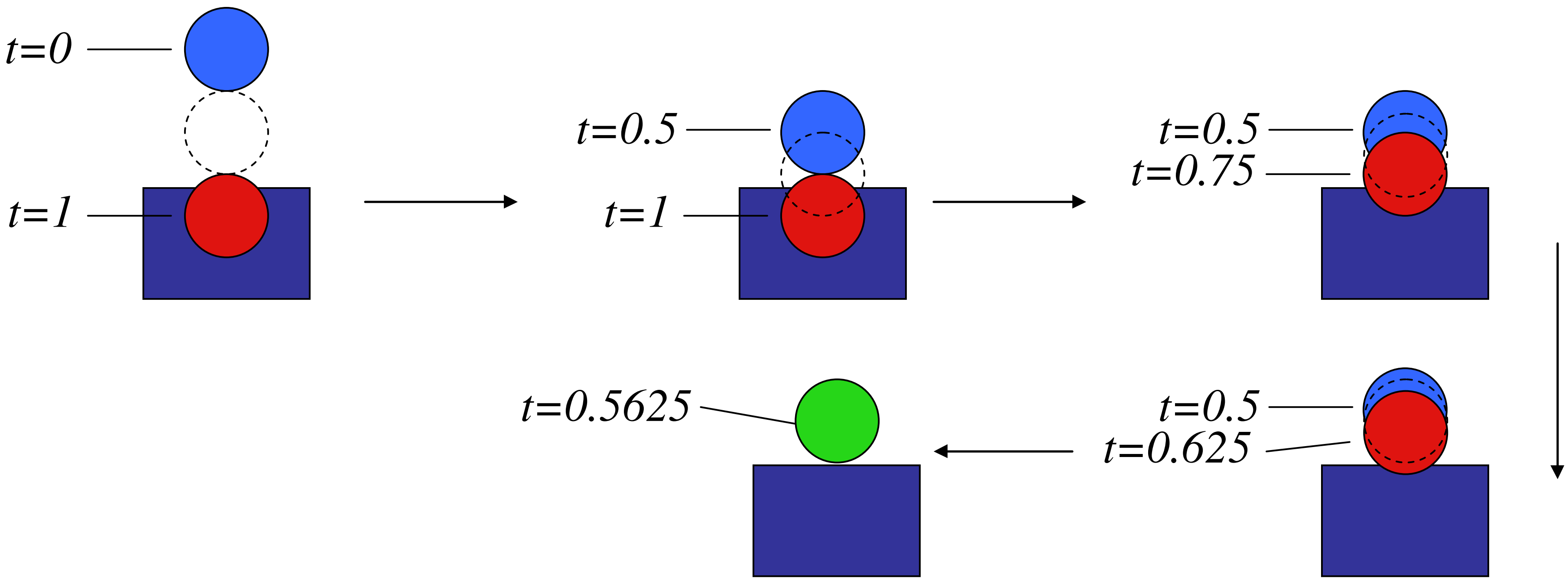
- know time *t*, position *x*, such that penetration occurs
- simply move position so that objects just touch, leave time the same
- multiple choices for how to move:
  - back along motion path
  - shortest distance to avoid penetration
  - some other option



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)

Interval Halving Example



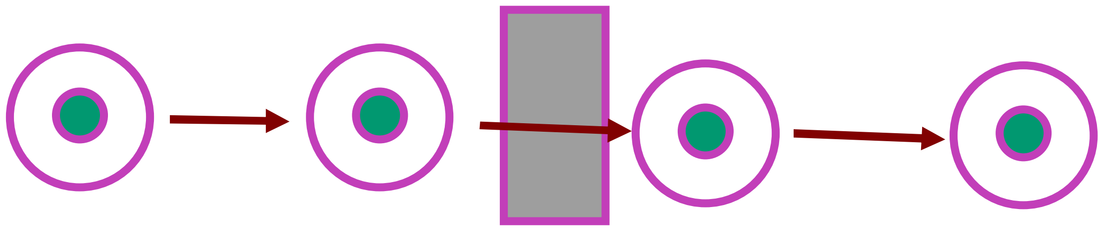
Interval Halving Discussion

- 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

Temporal Sampling

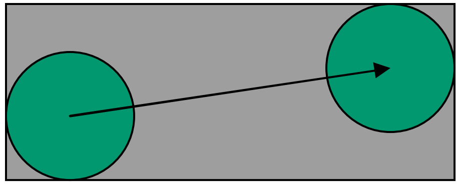
- subtle point: collision detection is about the algorithms for finding collisions *in time* as much as space

- temporal sampling
  - aliasing: can miss collision completely!



Managing Fast Moving Objects

- 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
  - largest conservative step is smallest distance divided by the highest speed - clearly could be very small
    - assume maximum velocity, smallest feature size
    - increase temporal and spatial sampling rate
  - pros: will find all collisions
  - cons: may be expensive, how to pick step size
- simple alternative: just miss the hard cases
  - player may not notice!

Collision Response

- frustrating to just stop
  - 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

Related Reading

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

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