Collision Detection

CS 4730 – Computer Game Design

Back to Physics

- Remember what we have here:
  - Objects have position, velocity, acceleration
  - In a physics routine
    - Collisions are determined
    - Forces collected
    - Numeric integration performed
    - Constraints resolved
    - Frame update and do the whole thing again

Back to Physics

- If there are no collisions, this is easy
- Pick your forces:
  - Gravity
  - Air resistance
  - "The Force"
- Figure out how they affect acceleration
- Do the math
- Update the frame

But With Collisions…

- Consider force
- Force directly changes acceleration
- Bigger mass => Bigger force
- Force puts things in motion, but also can bring things to a halt

Momentum

- Objects stay in motion due to momentum
- Momentum = mass * velocity
- If we do some fancy math:
  - $F = ma = m * (\Delta v / \Delta t)$
  - $F\Delta t = m\Delta v$
- $F\Delta t$ is called an impulse
- An impulse is a change in momentum

Conservation of Momentum

- When objects collide, momentum changes
  - … well, the magnitude is the same, it just goes in another direction
- That's the Law of Conservation of Momentum
- At the point of impact, ignoring other forces, the total momentum of all objects involved does not change

Conservation of Momentum

- Whatever momentum one object loses, the other gains
- This is a transfer of kinetic energy
- How objects react to the kinetic energy is the object's elasticity
- The coefficient of restitution defines how velocity changes before and after impact based on elasticity

Coefficient of Restitution

- If the coefficient is 0.0, then the object is totally inelastic and it absorbed the entire hit
- If the coefficient is 1.0, then the objects is totally elastic and all momentum will still be evident
- The sum of the kinetic energy will be the same

Putting It All Together

- So our final formula is:
- v_{1f} =
  $((e + 1)*m_2*v_2 + v_1*(m_1 – e*m_2)) / (m_1 + m_2)$
- v_{2f} =
  $((e + 1)*m_1*v_1 + v_2*(m_1 – e*m_2)) / (m_1 + m_2)$

Okay… Great!

- Math and physics are great and all…
- … but how do you know if two things collided?

Who’s Colliding?

- Okay, how would you do it?

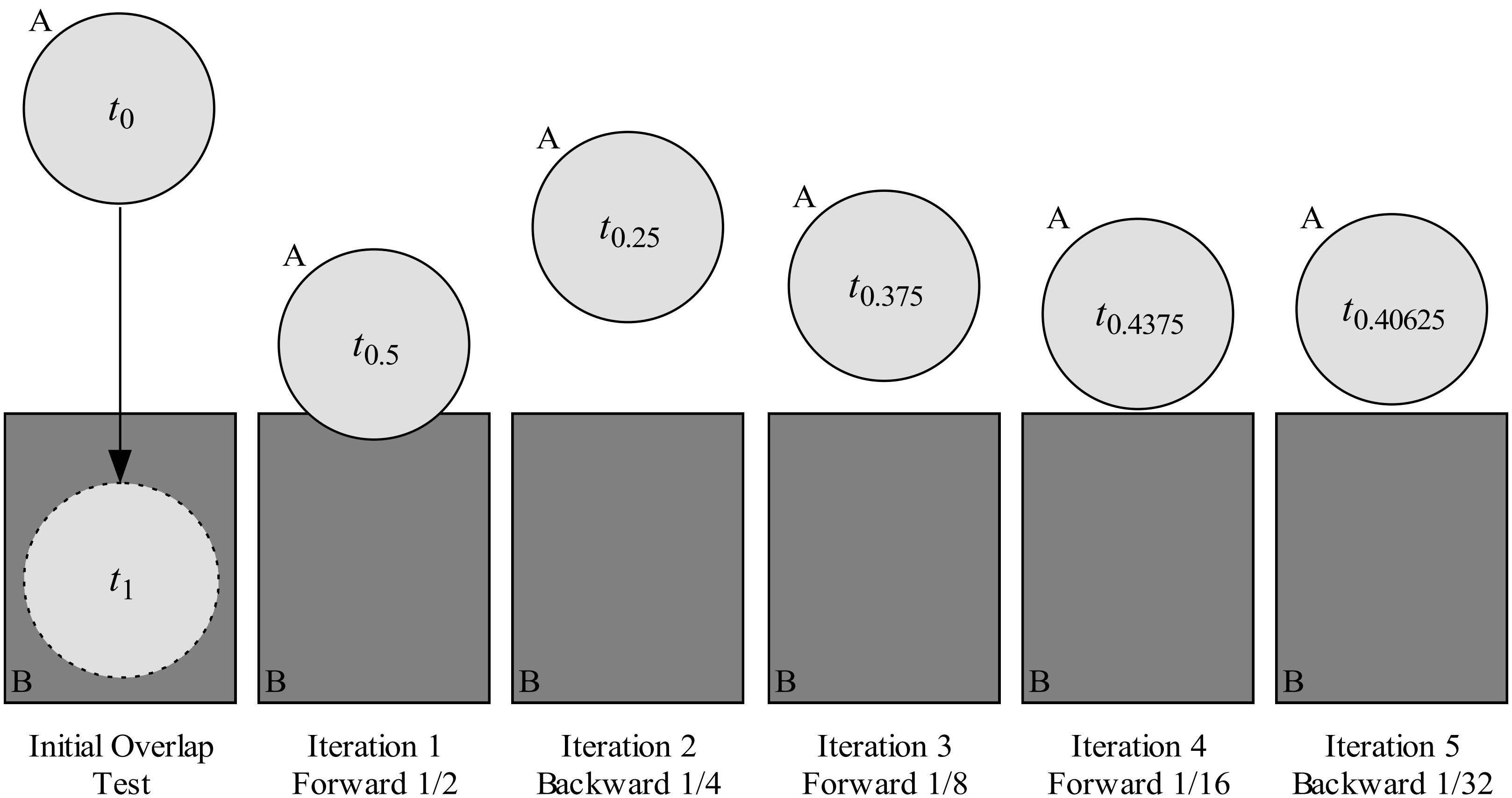
Who's Colliding?

- Compare everything
- Check only around the player
- Check only in a particular quadrant
- Check only around moving objects (i.e. not atRest())
- Remember: "perfect is the enemy of good enough"
- Don't go for perfection! Go for "looks right"

How about the actual collision?

- Overlap testing is probably most common / easiest method
- Does have a bit of error
- For each Δt, check to see if anything is overlapping (using some of the optimizations from the previous slide)

Overlap Testing

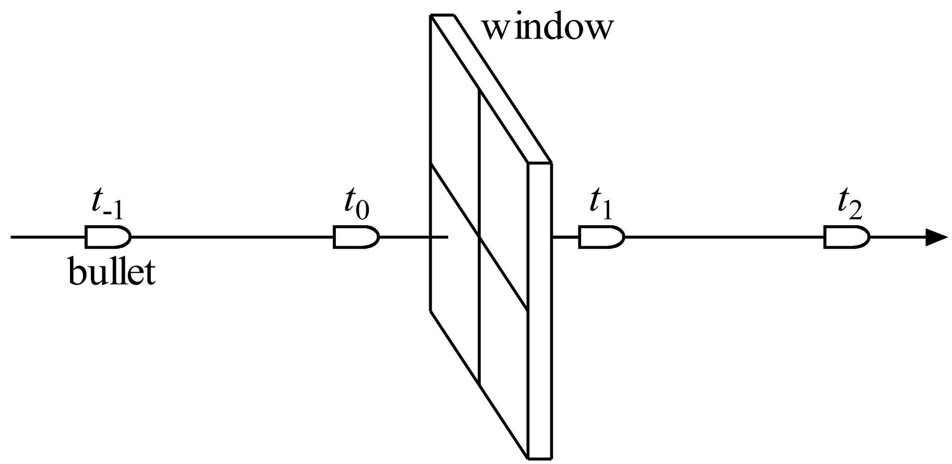


Problems With Overlaps

- What if your Δt is too big?
  - Well, you can fly right through an object before any collision is actually registered
- Kinda hard to do with complex shapes
  - Picture any game sprite
  - None of them are actually simple geometric shapes

Overlap Testing

- Fails with objects that move too fast
  - Unlikely to catch time slice during overlap
- Possible solutions
  - Design constraint on speed of objects
  - Reduce simulation step size

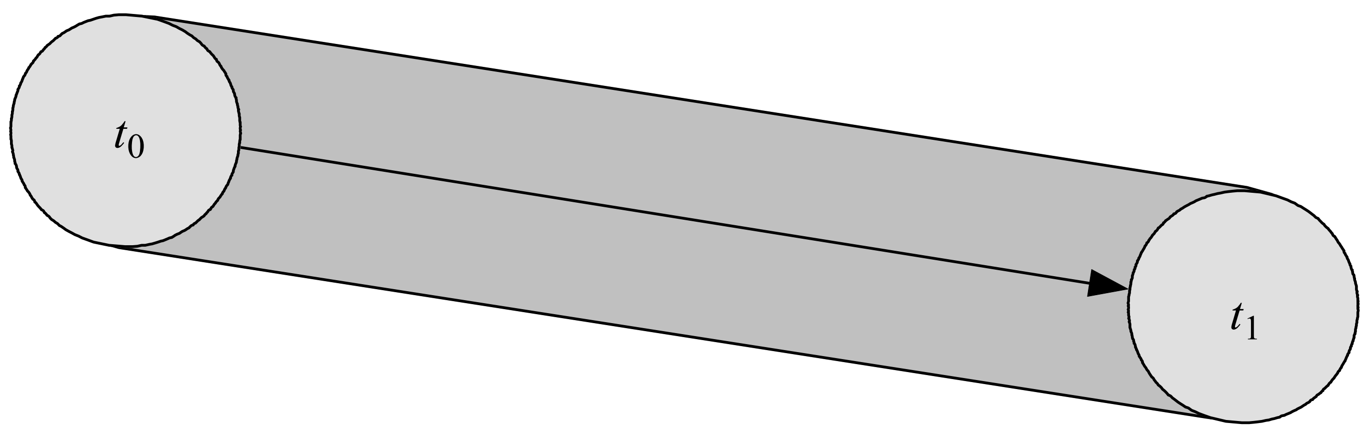


Intersection Testing

- Predict future collisions
- When predicted:
  - Move simulation to time of collision
  - Resolve collision
  - Simulate remaining time step

Swept Geometry

- Extrude geometry in direction of movement
- Swept sphere turns into a "capsule" shape



Limitations

- Issue with networked games
  - Future predictions rely on exact state of world at present time
  - Due to packet latency, current state not always coherent
- Assumes constant velocity and zero acceleration over simulation step
  - Has implications for physics model and choice of integrator

Introducing the Hit Box!

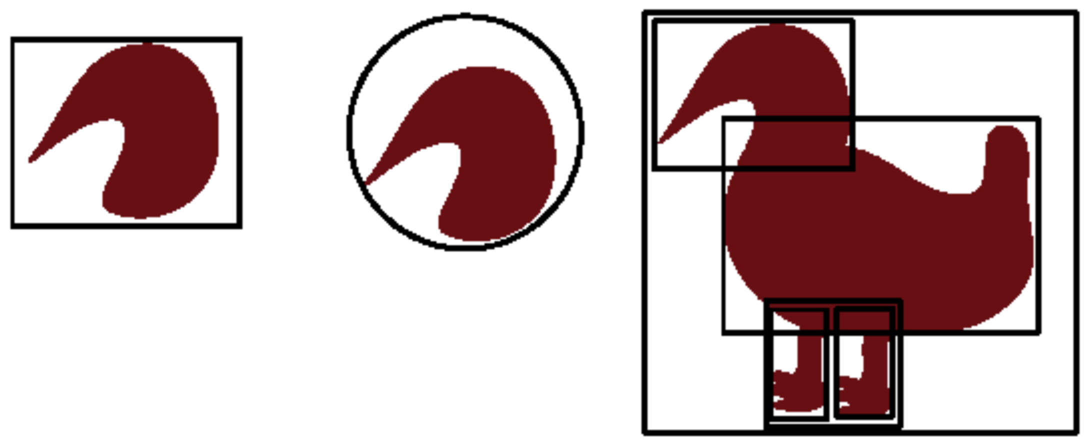
- All of our normal characters in early games were rectangles (or squares)
- Sprite sheets / tile sheets were easy to code and easy to read from
- Thus, characters were broken up into easy-to-render (and check) chunks
- More complex modern games have many more interesting hit boxes (often a set of hit boxes)

MDA of Hit Boxes

- The mechanic of the hit box is essential to having a game that runs at a reasonable speed
- How can the player exploit the hit box?
- What is the end aesthetic result?
- How can we balance between hit box accuracy and game play?

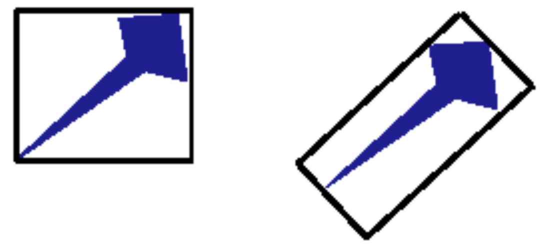
Hit Boxes

- Go for “good enough”
- Efficiency hacks/cheats
  - Fewer tests: Exploit spatial coherence
    - Use bounding boxes/spheres
    - Hierarchies of bounding boxes/spheres

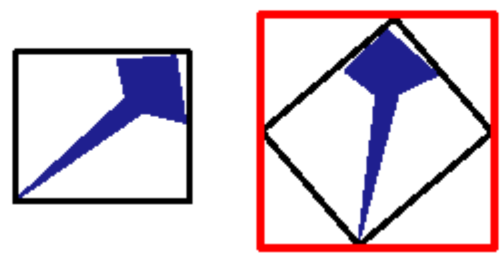


Bounding Boxes

- Axis-aligned vs. Object-aligned



- Axis-aligned BBox change as object moves
- Approximate by rotating BBox



Collision Spheres

- Another option is to put everything in a "bubble"
- Think Super Monkey Ball gone wild
- Sphere collision is cheap to detect!

How Many Hit Boxes?

- In the worst case (with complex objects) this is really a hard problem! O(n^2)
- For each object ***i*** containing polygons ***p***
  - Test for intersection with object ***j*** with polygons ***q***
- For polyhedral objects, test if object ***i*** penetrates surface of ***j***
  - Test if vertices of ***i*** straddle polygon ***q*** of ***j***

Speed Up

- To go faster
  - Sort on one dimension
    - Bucket sort (i.e. discretize in 1 dimension)
  - Exploit temporal coherence
    - Maintain a list of object pairs that are close to each other
    - Use current speeds to estimate likely collisions
  - Use cheaper tests

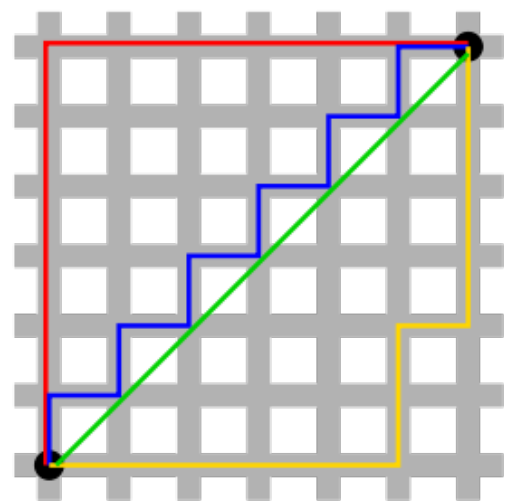
Cheaper Distance Tests

$$d = sqrt((x_1 - x_2)^2 + (y_1 - y_2)^2)$$

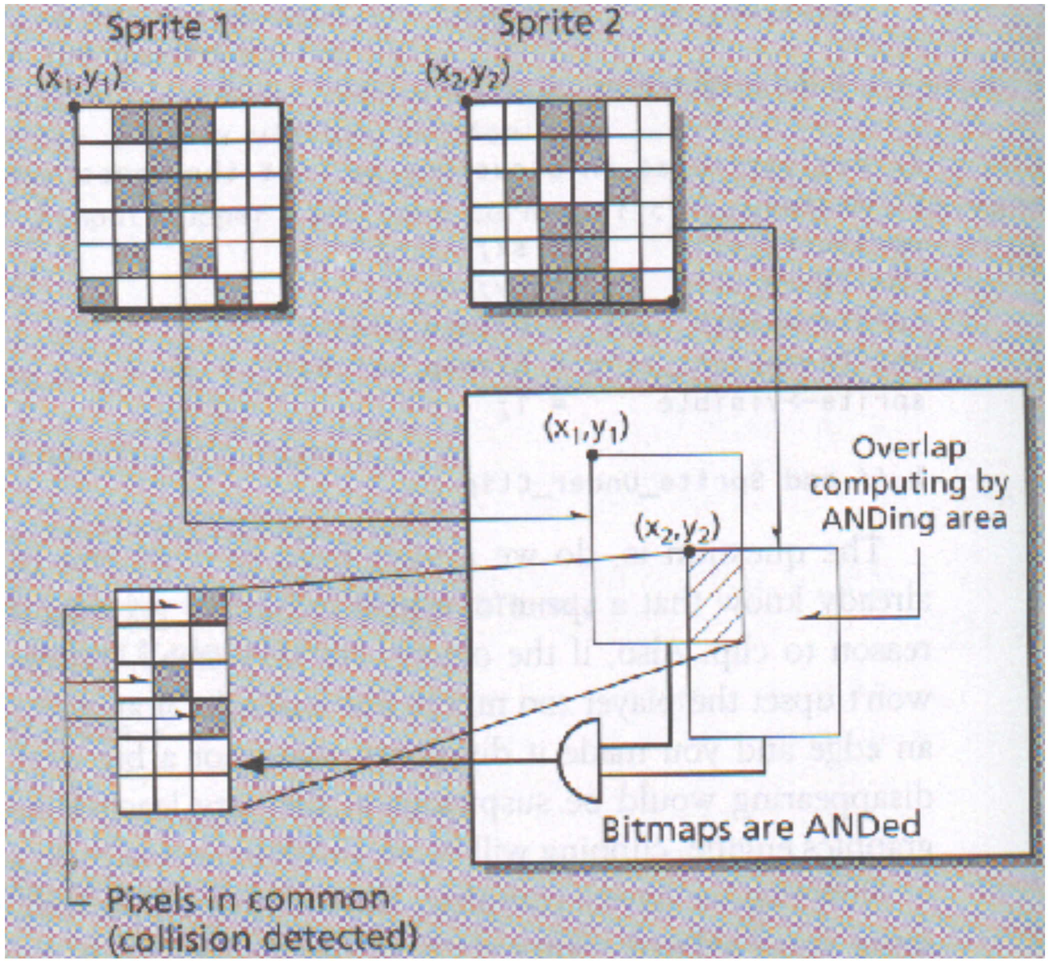
- Cheaper distance calculation:
  - Compare against d^2
- Approximation for comparison:

$$d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$$

  - Manhattan distance

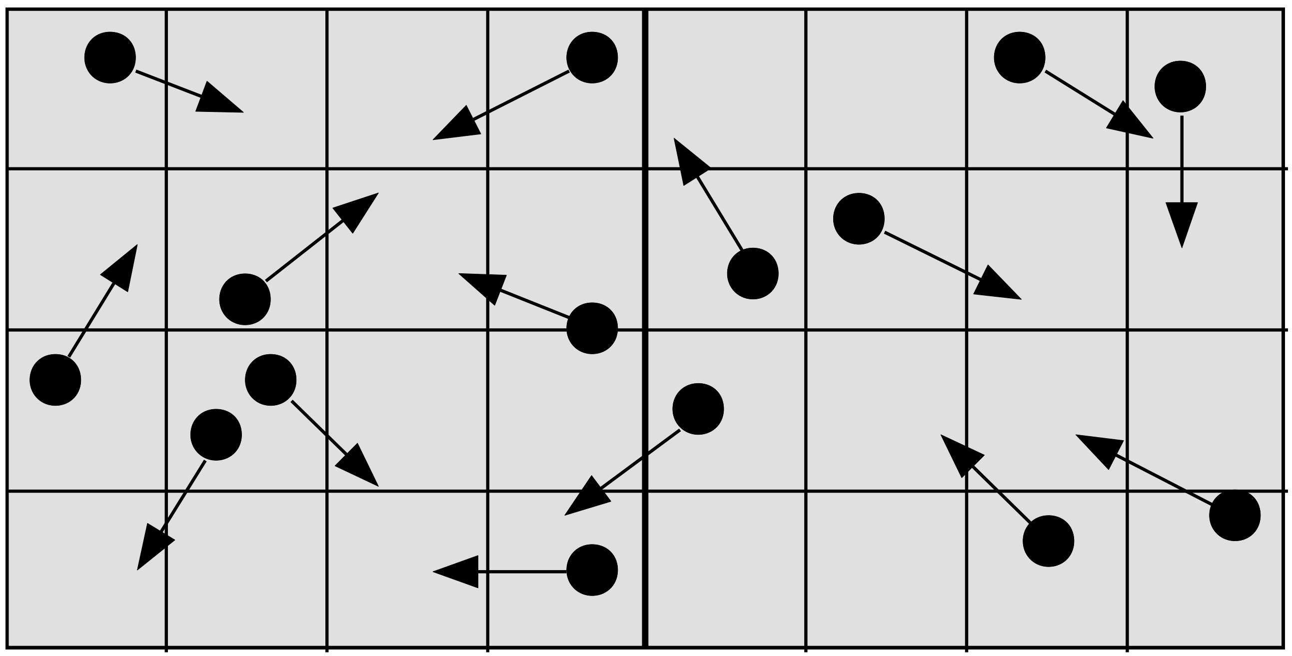


Sprite Collision Detection



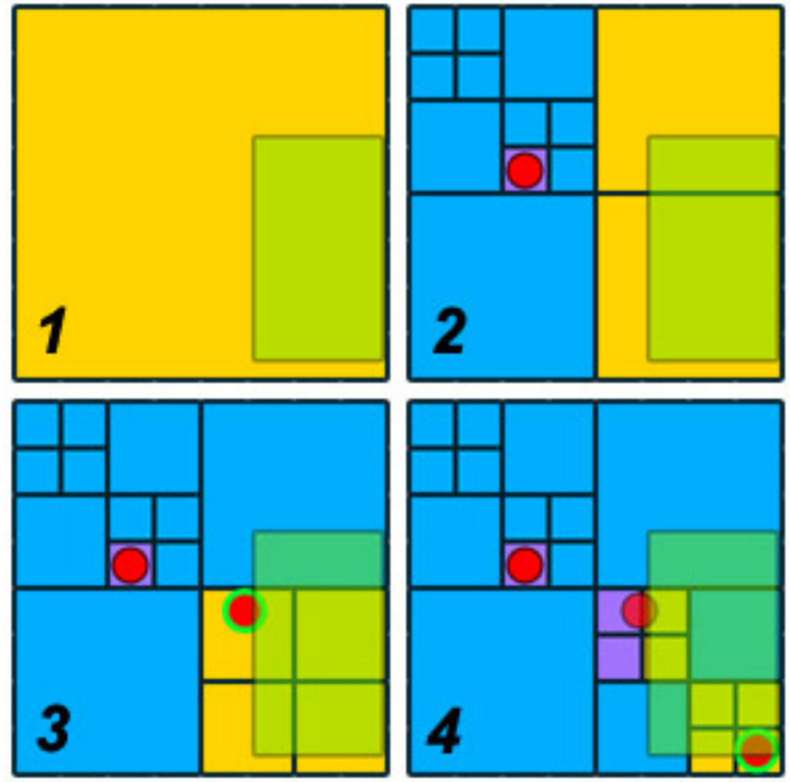
Achieving O(n) Time Complexity

One solution is to partition space



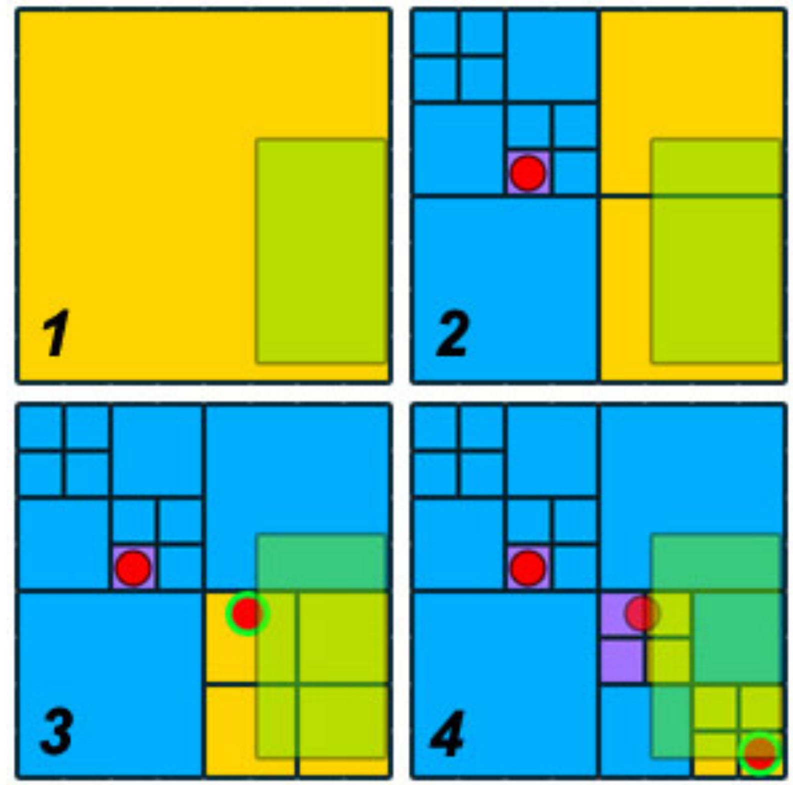
Achieving O(n) Time Complexity

- The box collides with the level 1 node – but there are no objects in level 1
- The box collides with two level 2 nodes – but there are no objects in them either. However, their child nodes need to be checked now.



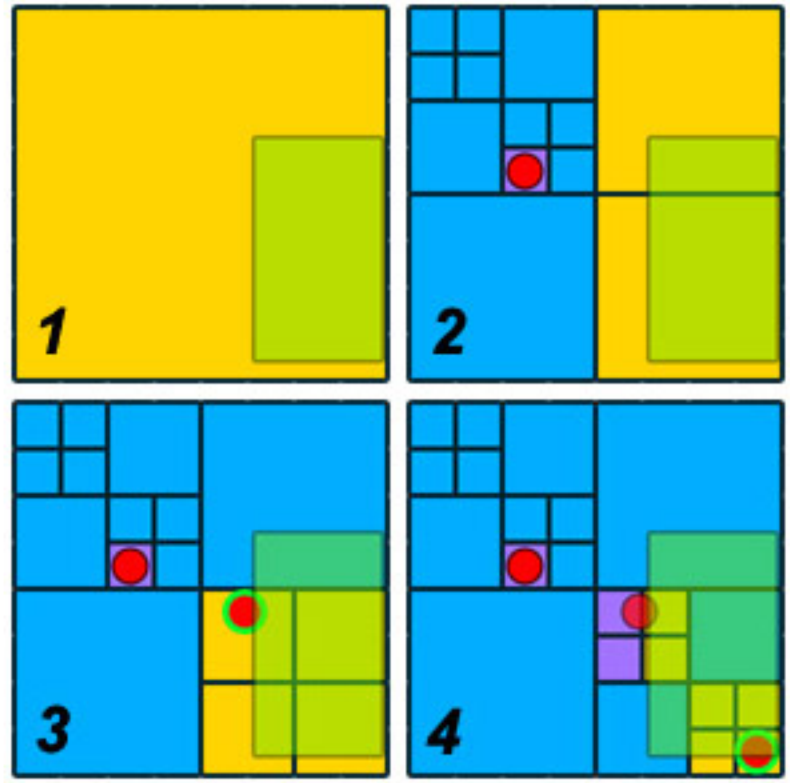
Achieving O(n) Time Complexity

- The box collides with four level 3 nodes, and there is one object in them, which is added to the return list. Note that there are no level 3 nodes in the top-right level 2 node, so it is not queried any further.



Achieving O(n) Time Complexity

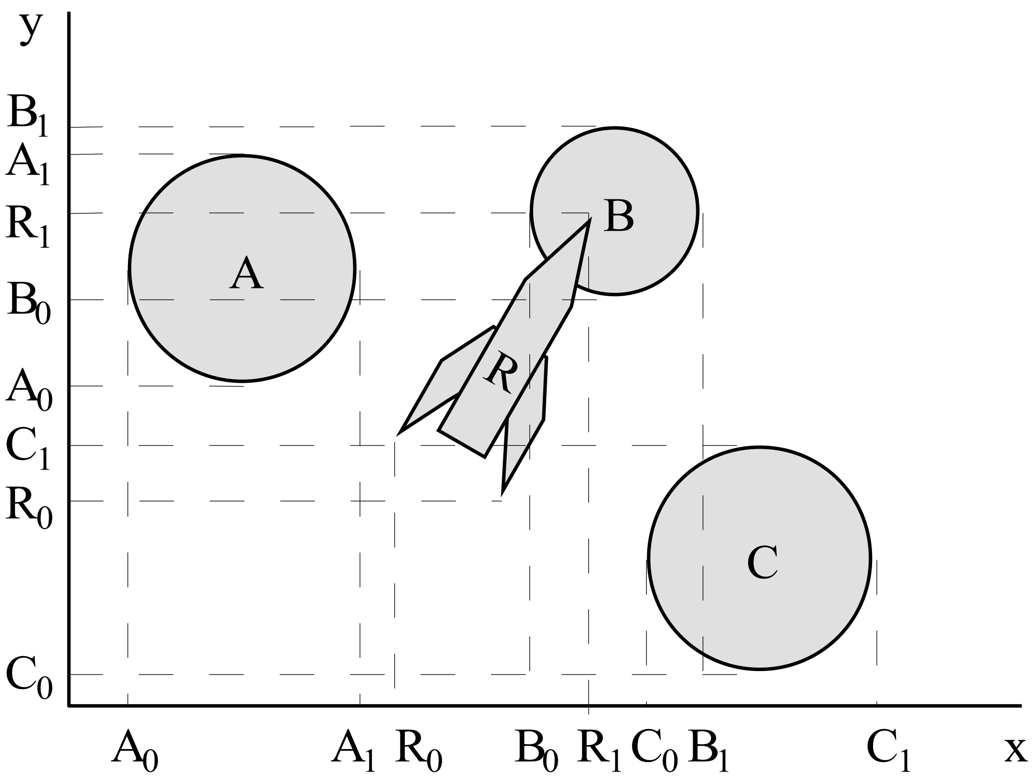
- Finally, the box is colliding with six level 4 nodes, one of which contains another object. Note that the object we just returned was on an edge, so it was contained within the level 3 node instead of a level 4 node.

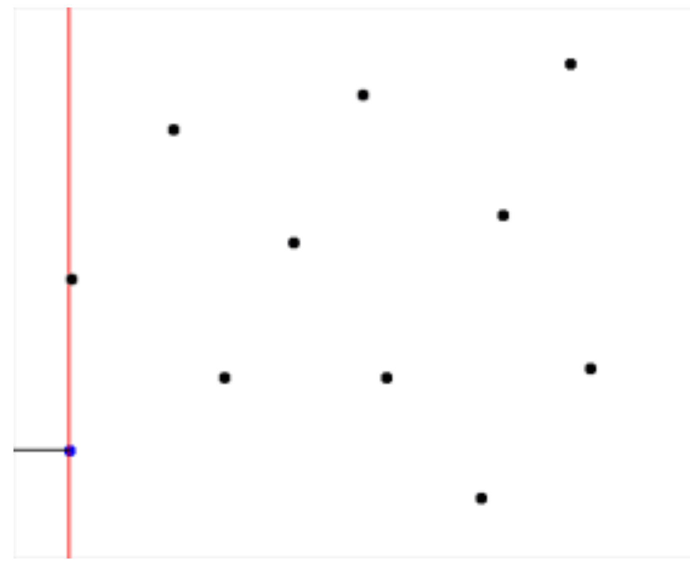


- Credit: http://www.kyleschouviller.com/wsuxna/quadtree-source-included/

Achieving O(n) Time Complexity

Another solution is the plane sweep algorithm





Collision Resolution

- Two billiard balls strike
  - Calculate ball positions at time of impact
  - Impart new velocities on balls
  - Play "clinking" sound effect
- Rocket slams into wall
  - Rocket disappears
  - Explosion spawned and explosion sound effect
  - Wall charred and area damage inflicted on nearby characters
- Character walks through wall
  - Magical sound effect triggered
  - No trajectories or velocities affected

Collision Resolution

- Resolution has three parts
  1. Prologue
  2. Collision
  3. Epilogue

Prologue

- Collision known to have occurred
- Check if collision should be ignored
- Other events might be triggered
  - Sound effects
  - Send collision notification messages

Collision

- Place objects at point of impact
- Assign new velocities
  - Using physics or
  - Using some other decision logic

Epilogue

- Propagate post-collision effects
- Possible effects
  - Destroy one or both objects
  - Play sound effect
  - Inflict damage
- Many effects can be done either in the prologue or epilogue