Collisions

- We’re moving things around
  - Collisions may occur
  - How do we test for them?
Definition of a collision

• Penetration of polygon into another polygon
• This is indicated by some point in one polygon inside the other
Collisions

Ideas on how to tell if we have a collision?
Simple algorithm

for each polygon p1 in objects
  for each polygon p2 in objects
    if (p1 != p2)
      if (!Test(p1, p2))
        Test(p2, p1)
Test(p1, p2)

• Test to see if any vertex in p1 is contained in p2
  – How do we do this?

Would this always work?
What about this case?
Separator lines

• A separator line is a line in space such that all vertices in $p_1$ are on one side and all vertices of $p_2$ are on the other side
  – If a separator line exists, it will contain an edge.
Separator lines
The Separator Equation

• Let $x_1, y_1$ and $x_2, y_2$ be line endpoints
  – Assuming clockwise around the polygon
  – We can make a line equation like this:
    • $a = y_1 - y_2$
    • $b = x_2 - x_1$
    • $c = -a x_1 - b y_1$
    • For any point on the line: $ax + by + c = 0$
    • Easy enough
What about off the line?

$ax + by + c < 0$

$ax + by + c > 0$

Polygon interior side

Polygon exterior side
Test...

for each edge $e$ in $p_2$ // Try each edge
compute line equation $a$, $b$, $c$
possible $\leftarrow$ true // Could be a separator
for each vertex $v$ in $p_1$ // Try each vertex against edge
  $r \leftarrow a \, v_x + b \, v_y + c$ // Line equation
  if $r \leq 0$ // If true, not a separator line
    possible $\leftarrow$ false
    break

if possible = true then
  return false // No collision

return true // Collision possible
private bool Test(CollisionItem c1, CollisionItem c2)
{
    // Symmetrical search for an edge that segments the two...
    if (!TestLR(c1, c2))
        return false;

    if (!TestLR(c2, c1))
        return false;

    // We know we have a collision if we get to here.
    // Create some list of collision items to resolve…

    return true;
}
private bool TestLR(CollisionItem c1, CollisionItem c2)
{
    List<Vector2> v1 = c1.poly.Vertices();
    List<Vector2> v2 = c2.poly.Vertices();
    Vector2 v2a = (Vector2)v2[v2.Count - 1];  // Last vertex in v2

    foreach (Vector2 v2b in v2)
    {
        float a = v2a.Y - v2b.Y;  // Computing the edge line function
        float b = v2b.X - v2a.X;
        float c = -a * v2a.X - b * v2a.Y;

        bool possible = true;
        foreach (Vector2 v1a in v1)
        {
            if (a * v1a.X + b * v1a.Y + c <= 0)
            {
                possible = false;
                break;
            }
        }

        if (possible)
        {
            c2.witness[c1] = cnt2;  // We have a separator line. This is our witness, save it
            return false;
        }

        // Make this the end point for the next pass
        v2a = v2b;
    }

    return true;  // If no separating edges have been found, we potentially have an overlap
}
What's the running time?

• ???
Okay, can we do better?

• Running time is $O(N^2)!$
  – Scary...
Are there cases with better running time?

• What are some other possible collision tests we might need?
Accelerations that really matter

• Bounding box tests
  – Don’t test objects if a bounding box does not overlap

• Witness tests
  – We’ll keep track of separator edges, they may still be valid

Assuming all pairs testing
Bounding boxes

For each object compute min and max in each dimension
private void Move()
{
    verticesM.Clear(); // Destination
    bool first = true;
    foreach (Vector2 v in verticesB)
    {
        float ca = (float)Math.Cos(state.r);
        float sa = (float)Math.Sin(state.r);

        // Bounding box computation
        if (first)
        {
            min[0] = max[0] = vp.X;
            first = false;
        }
        else
        {
            if (vp.X < min[0])
            {
                min[0] = vp.X;
            }
            else if (vp.X > max[0])
            {
                max[0] = vp.X;
            }
            if (vp.Y < min[1])
            {
                min[1] = vp.Y;
            }
            else if (vp.Y > max[1])
            {
                max[1] = vp.Y;
            }
        }
        verticesM.Add(vp);
    }
}
Bounding box overlap tests

• Suppose intervals in one dimension
  – $i_1=[2, 7]$
  – $i_2=[6, 9.001]$
  – $i_3=[4, 8]$
  – $i_4=[11, 14]$
  – $i_5=[15, 16]$
  – $i_6=[17, 21]$

  – What’s the fastest way to determine overlaps of intervals?
Sweep and Prune

Sort begins/ends in order
Scan forward, keeping track of who’s active

\[ i_1 = [2, 7] \]
\[ i_2 = [6, 9] \]
\[ i_3 = [4, 8] \]
\[ i_4 = [11, 14] \]
\[ i_5 = [15, 16] \]
\[ i_6 = [17, 21] \]

Sort begins/ends in order
Scan forward, keeping track of who’s active

\[ O(n \log n + k) \]
How to extend to 2D?

• ???
Extending to 2D

- For two bounding boxes to overlap, they must overlap in both dimensions
- I keep a hash of all objects overlapped by this object
  - Hash object to integer
  - If integer is 2, we overlap
// All of the items we know about to test for collisions
class CollisionItem
{
    // Bounding box overlaps (CollisionItem to integer count)
    public Hashtable overlap = new Hashtable();
    public Polygon poly; // The polygon
}

// We also keep an array of beginning and end points.
class EndItem
{
    public bool begin; // Is this a beginning?
    public CollisionItem item; // The item we are referring to
}

List<EndItem> boxx = new List<EndItem>(); // Bounding box items in X axis
List<EndItem> boxy = new List<EndItem>(); // Bounding box items in Y axis
List<CollisionItem> items = new List<CollisionItem>(); // All of our items
// Adds a polygon to the collision system
public void Add(Polygon p)
{
    // Create an object for the item
    CollisionItem c = new CollisionItem();
    c.poly = p;
    items.Add(c);

    // Create a two bounding box items for this polygon
    EndItem beg = new EndItem();
    beg.begin = true;
    beg.item = c;

    EndItem end = new EndItem();
    end.begin = false;
    end.item = c;

    // Add to the two array lists
    boxx.Add(beg);
    boxx.Add(end);

    boxy.Add(beg);
    boxy.Add(end);
}

What would it take to remove a polygon?
public bool Test()
{
    anyCollide = false;

    foreach (CollisionItem c in items) // Set all CollisionItems to an empty overlap set
        c.overlap.Clear();

    Sort(boxx, 0); // Insertion sort the two lists
    Sort(boxy, 1);

    Sweep(boxx); // Sweep the lists
    Sweep(boxy);

    foreach (CollisionItem c1 in items)
    {
        foreach (CollisionItem c2 in c1.overlap.Keys)
        {
            if ((int)c1.overlap[c2] < 2) // Does it overlap in both dimensions?
                continue;

            if (Test(c1, c2)) // If so, test for polygon overlap
            {
                anyCollide = true; // If we have a collision…
            }
        }
    }

    return anyCollide;
}
private void Sweep(List<EndItem> box)
{
    Hashtable active = new Hashtable(); // Set of all active items
    foreach (EndItem b1 in box) // Test a dimension...
    {
        if (b1.begin)
        {
            foreach (CollisionItem b2 in active.Keys) // New box overlaps with every item in the active list
            {
                CollisionItem c1, c2;
                if (b1.item.poly.Id < b2.poly.Id) // Overlap is symmetrical. To prevent two entries, we
                { // keep track of overlaps with increasing id values.
                    c1 = b1.item;
                    c2 = b2;
                }
                else
                {
                    c2 = b1.item;
                    c1 = b2;
                }

                if (c1.overlap.Contains(c2))
                    c1.overlap[c2] = (int)c1.overlap[c2] + 1; // Previous overlap, increment the count
                else
                    c1.overlap[c2] = 1; // First time overlap.
            }

            // Add to the active items
            active.Add(b1.item, b1);
        }
        else
        {
            // Remove from list of actives
            active.Remove(b1.item);
        }
    }
}
You might have noticed...

Sort(boxx, 0);     // Insertion sort the two lists

You’re probably thinking:

“Why would the instructor ever use an insertion sort? I’m smarter, so I’ll use QuickSort here and win brownie points.”
Question

• When is insertion sort faster than Quicksort?
  – Or Heapsort or just about any sort?
Coherence

• *Coherence* – the property of computer graphics that things don’t change much over small distances
  – In particular, objects don’t move very far
  – Insertion sort is perfect when the list is close to being sorted already
    • We keep the lists and just move what has gotten out of order, usually a small distance.
    • We get expected O(N) running time!
Witness tests

• Another element of coherence
  – A separator edge in one frame is often still one in the next frame.
  – Keep the edge as a witness, test it first
private bool TestWLR(CollisionItem c1, CollisionItem c2)
{
    if (!c2.witness.Contains(c1)) // See if a witness exists
        return true;

    int w = (int)c2.witness[c1]; // It does. Let's try it...

    List<Vector2> v1 = c1.poly.Vertices();
    List<Vector2> v2 = c2.poly.Vertices();

    Vector2 v2b = (Vector2)v2[w]; // Determine vertices for witness edge
    Vector2 v2a;
    if (w == 0)
        v2a = (Vector2)v2[v2.Count - 1];
    else
        v2a = (Vector2)v2[w - 1];

    float a = v2a.Y - v2b.Y; // Compute the line function
    float b = v2b.X - v2a.X;
    float c = -a * v2a.X - b * v2a.Y;

    bool possible = true;
    foreach (Vector2 v1a in v1)
    {
        if (a * v1a.X + b * v1a.Y + c <= 0) // If r <= zero, we're on the wrong side of the line
        {
            possible = false;
            break;
        }
    }

    if (possible)
        return false; // We have a separator line. Witness is still good=

    c2.witness.Remove(c1); // Witness is invalidated
    return true;
}
What do you do when a collision occurs?

- We generally need to know **when** the collision has occurred
  - Time steps may have stepped over it...
Penetration

• When a collision occurs, we compute the penetration of the vertex into the polygon

• We consider a valid collision to be $0 < p < \varepsilon$ for some small $\varepsilon$.
  – I used 0.0001
  – Greater than $\varepsilon$ is too deep
Computing Penetration

• If you normalize a, b in the line equation, 
  \( p = -(ax + by + c) \) is the penetration

• What algorithm do we use to determine the penetration?
Algorithm

for each vertex v in p1
    possible ← true  // Until we know otherwise
    bestR ← BIGNUM  // Candidate for best R for this vertex
    for each edge e in p2
        compute a,b
        normalize a,b
        compute c
        r ← ax + by + c
        if r > 0
            possible ← false
            break
        else if −r < bestR  // So far...
            bestR ← −r
    if possible then
        return bestR

This algorithm might miss a second vertex penetration...
private bool TestVE(CollisionItem c1, CollisionItem c2)
{
    ArrayList v1 = c1.poly.Vertices();
    ArrayList v2 = c2.poly.Vertices();

    foreach (Vector2 v1a in v1) // Loop over all of the vertices
    {
        bool possible = true; // Candidate vertex
        Vector2 v2a = (Vector2)v2[v2.Count - 1];

        float bestR = 1e10f; // Least penetration
        Vector2 bestl = new Vector2(); // Best intersection point
        Vector2 bestN = new Vector2(); // Best intersection normal

        foreach (Vector2 v2b in v2) // Loop over all of the edges. Is this vertex contained by all?
        {
            float a = v2a.Y - v2b.Y; // Compute the edge line function
            float b = v2b.X - v2a.X;
            float len = (float)Math.Sqrt(a * a + b * b); // Normalize these values
            a /= len; b /= len;

            float c = -a * v2a.X - b * v2a.Y; // Compute c and r...
            float r = a * v1a.X + b * v1a.Y + c;

            if (r > 0) // If r <= zero, we're on the inside side of the line
            {
                possible = false; break; // This is not a separator line
            }
            else if (-r < bestR)
            {
                bestR = -r; // Amount of penetration
                bestN.X = a; // Edge normal
                bestN.Y = b;
                bestl.X = v1a.X + a * bestR; // Where the actual
                bestl.Y = v1a.Y + b * bestR; // intersection would be
            }

            // Make this the end point for the next pass
            v2a = v2b;
        }

        if (possible)
        {
            bestCollide.p1 = c1.poly;
            bestCollide.p2 = c2.poly;
            bestCollide.v1 = bestl;
            bestCollide.N2 = bestN;
            bestCollide.r = bestR;
            return true;
        }
    }

    return false;
}
Determining the collision time

- We want to step by delta
  - What if we penetrate too deep?

- Watch out for “Tunnelling”
  - Going all of the way through and out the other side
  - How do we prevent Tunnelling?
  - You’ll see this in Step 9.
Tunneling issues

• If smallest object is 0.1 and v is 1, we can’t step more than 0.1s
  – Why

• What if we figure in acceleration?
  – Step 9 really demonstrates this...
Bisection Search

• If delta is too deep, try delta/2

• We need the ability to try a step and back off if it is too much
  – This is why the state is in a class
The Advance Algorithm

// We need to advance by delta
while delta > 0
    step ← delta // How much we will try
do
    badcollision ← false
    for each polygon p in objects // Euler step everyone
        save state and advance by step
        if collision.Test() = true
            if any collision is too deep
                badcollision ← true // We can’t go this far, backtrack
                step ← step/2 // Halve our step size
                for each polygon p in objects // And back up
                    restore state
            else
                process the collisions
        else
            if !badcollision
                delta ← delta – step
    while badcollision = true // End of do…
private void Advance()
{
    long time = stopwatch.ElapsedMilliseconds;       // How much time as elapsed?
    float delta = (time - lastTime) * 0.001f;       // Delta time in milliseconds
    lastTime = time;

    while (delta > 0)
    {
        float step = delta;       // Step size we will try
        bool badcollision;

        do
        {
            badcollision = false;
            foreach (Polygon p in objects.Values) { p.SaveState(); p.Step(step); }

            if (collision.Test())       // Test for collision
            {
                foreach (Collision.Collide c in collision.Collides) // My system returns a list of collisions
                {
                    if (c.r > deepest && c.p1.V.X != 0 && c.p1.V.Y != 0 && c.p2.V.X != 0 && c.p2.V.Y != 0)
                    {
                        step /= 2; badcollision = true; break;
                    }
                    c.p1.V = new Vector2(0, 0); // Stop the moving polygon
                    c.p1.RV = 0;
                    c.p1.DeltaP(c.n2.X * c.r, c.n2.Y * c.r); // Move it so it is not longer colliding
                    c.p2.V = new Vector2(0, 0); // Zero velocity
                    c.p2.RV = 0;                // Zero rotational velocity
                }
            }
            if (badcollision)
            {
                // Restore the state after any bad collision
                foreach (Polygon p in objects.Values) { p.RestoreState(); }
            }
            else
            {
                delta -= step;
            }
        } while (badcollision);
    }
}
Example

- Suppose we want to step 0.016 seconds.
  - delta = 0.016, step = delta
  - We try to step and get a too deep collision
    - Back up and try a step of 0.008
  - We try a step of 0.008 and get no collision
    - delta = delta – step, so delta = 0.008 (left)
  - delta = 0.008, step = delta
  - We try a step of 0.008 and get a too deep collision
    - Back up and try a step of 0.004
  - We try a step of 0.004 and the collision is just right
    - Process the collision, delta = delta – 0.004 = 0.004
  - We try a step of 0.004 and no collision
    - delta = delta – 0.004 = 0
  - Loop exits, we are done...
What to do when you collide?

• Some adhoc response
  – Jump back a bit for example

• Remove colliding object
  – Might be one or both

• Stop
  – Can we just set $v$ and $rv$ to zero?

• Bounce
  – Physics response to the collision
Stop?

• Easiest solution, just move $p_1$ away by the overlap distance
  – See example in the previous slide
  – Also in Step 9 in more detail
Bounce

• Physics response assuming inelastic collision
Object collisions

- Imagine objects A and B colliding

Assume collision is point on edge
What happens?

- ???

Where do we get $n$ from?
Our pal Newton...

- Newton’s law of restitution for instantaneous collisions with no friction
  - $\varepsilon$ is a coefficient of restitution
    - 0 is total plastic condition, all energy absorbed
    - 1 is total elastic condition, all energy reflected

$\mathbf{v}^+ \cdot \mathbf{n} = -\varepsilon \mathbf{v}^- \cdot \mathbf{n}$

What’s the consequence of “no friction”?
Velocity at a point

- The velocity at a point is a combination of a linear and rotational component

\[ v_p = v_a + \omega_a r_{\perp} \]

- Vector from center of mass to point

\[ (x, y)_{\perp} = (-y, x) \]
Impulse felt by each object

- Newton’s third law: equal and opposite forces
  - Force on A is \( jn \) (\( n \) is normal, \( j \) is amount of force)
  - Force on B is \( -jn \)
  - So...

\[
\begin{align*}
  v_A^+ &= v_A^- + \frac{j}{m_A} n \\
  v_B^+ &= v_B^- - \frac{j}{m_B} n
\end{align*}
\]

We’ll need to know \( j \)

No rotation for now...
Solving for $j$...

$$v^+ \cdot n = -\varepsilon v^- \cdot n$$

$$(v_A^+ - v_B^+) \cdot n = -\varepsilon (v_A^- - v_B^-) \cdot n$$

$$v_A^- \cdot n + \frac{j}{m_A} n \cdot n - v_B^- \cdot n + \frac{j}{m_B} n \cdot n = -\varepsilon (v_A^- - v_B^-) \cdot n$$

$$j = \frac{-(1+\varepsilon)(v_A^- - v_B^-) \cdot n}{n \cdot n \left( \frac{1}{m_A} + \frac{1}{m_B} \right)}$$

Then plug $j$ into...

$$v_A^+ = v_A^- + \frac{j}{m_A} n$$

$$v_B^+ = v_B^- - \frac{j}{m_B} n$$

Assumes $n$ is normalized
What about rotation?

\[ v_A^+ = v_A^- + \frac{j}{m_A} n \]

\[ \omega_A^+ = \omega_A^- + \frac{r_{\perp A} \cdot jn}{I_A} \]

\[ j = \frac{-(1 + \varepsilon)(v_A^- - v_B^-) \cdot n}{\left( \frac{1}{m_A} + \frac{1}{m_B} \right) + \frac{(r_{\perp A} \cdot n)^2}{I_A} + \frac{(r_{\perp B} \cdot n)^2}{I_B}} \]
private bool Bounce(Polygon p1, Polygon p2, Vector2 V1, Vector2 N2, float r)
{
    Vector2 r1 = V1 - p1.P;  // A vector from the center of mass to
    Vector2 r2 = V1 - p2.P;  // the intersection point for both objects

    Vector2 p1v = new Vector2(p1.V.X + p1.RV * -r1.Y, p1.V.Y + p1.RV * r1.X);  // Velocity at v1 for both objects.
    Vector2 p2v = new Vector2(p2.V.X + p2.RV * -r2.Y, p2.V.Y + p2.RV * r2.X);  // This is velocity at the point.

    // What is the relative velocity
    float vrel = Vector2.Dot(N2, (p1v - p2v));
    if (vrel >= 0)
        return true;  // Return if not moving towards each other

    if (r > epsilon)
        return false;  // Are we too deep (bad penetration)?

    Vector2 vdiff = new Vector2(p1v.X - p2v.X, p1v.Y - p2v.Y);  // Difference in velocities at the point

    // Numerator of the fraction for both J values
    float num = -(1 + epsilon) * Vector2.Dot(vdiff, N2);

    // j for the velocity change
    float j = num / (1 / p1.Mass + 1 / p2.Mass);

    // j for the angular velocity change
                      SQR(-r1.Y * N2.X + r1.X * N2.Y) / p1.Inertia + SQR(-r2.Y * N2.X + r2.X * N2.Y) / p2.Inertia);

    p1.RV = p1.RV + jr * (-r1.Y * N2.X + r1.X * N2.Y) / p1.Inertia;
    p2.RV = p2.RV - jr * (-r2.Y * N2.X + r2.X * N2.Y) / p2.Inertia;


    return true,
}
What are we leaving out?

• Simplified model
  – Omits:
    • Objects in continuous contact
    • Requires simultaneous solution
      – Simulated without and generally works okay