Lecture 5
Announcements
Notes About Retries

• E-mail the TAs when you want a retry graded
  – Will try to grade it before the next due date, but no guarantees
• Must handin each week separately
• Getting an incomplete on a retry isn’t the end
  – Extra retries must be done the week after standards
  – Must submit a finished version eventually
instanceof

- Not reflection
- Will still work with the obfuscator
- Better to use method overriding + double dispatch
- Still very bad design
Tou 1 Feedback

- Games look great!
- We’re excited for Tou2 :D
Mid-Semester Feedback Forms

• Will go out soon
• Tell us how we’re doing
• Suggestions for upcoming special lecture topics
Detection vs. Response

- Detection is “are these two entities (meaning their shapes) colliding?”
  - Double dispatch to use the right algorithm
- Response is “because these two entities are colliding, what should they do?”
  - Double dispatch to make each respond correctly (I hit a bullet vs. I hit an enemy vs. a bullet hits an enemy)
- Resolution is “how do I make these two entities not be colliding anymore?”
  - What if we want realistic physics — atoms can’t overlap, but they don’t always bounce off of each other?
  - Next week!
QUESTIONS?
LECTURE 5
Collision Detection II
Polygons: Harder Than They Look

- Lots of places that could collide (overlap)
- Irregular shapes
- Test every point?
  - What if both polygons are huge?
  - Can get false positives if pixels are close together
Polygon Definitions

• Polygon
  – Bounded by 3 or more straight line segments (edges)
  – Edges meet at vertices

• Convex Polygon
  – Every interior angle < 180°
  – Any line through the shape crosses only twice
  – Boundary never crosses itself

• Concave Polygon
  – Interior angle > 180°
  – Line through the shape crosses more than twice
Math Assumptions

- Positive y direction is up
- In viewports (where you do everything), can have an arbitrary coordinate system – choose “math coordinates” if you want
  - Think of how you could use `AffineTransform` to do this
- Easier to talk about math with +y pointing up
- If +y points down, beware that orientation reverses
Point in Polygon

- Think of the border of a polygon as a path of vectors
  - Counterclockwise order!!!
- For convex polygons, points in the interior will be on the same side of all the vectors

On the left of all the vectors
On the right of this vector
Point in Polygon

- To determine what side of a vector $v$ a point is on:
  - Draw another vector $p$ from the base of the vector to that point
  - Take cross product $v \times p$
  - If result is positive, it’s on the left

$\text{Cross-product} < 0$

$\text{Cross-product} > 0$
Point in Polygon Algorithm

1. Iterate over the edges (counterclockwise)
2. Construct a vector $\mathbf{v}$ along the edge
3. Construct a vector $\mathbf{p}$ to the point
4. Take the cross-product $\mathbf{v} \times \mathbf{p}$
5. If all cross-products are positive, point is inside
6. If any cross-product is negative, it’s outside
Shape-Polygon

• What about Circle-Polygon, AAB-Polygon, Polygon-Polygon collisions?
• Can all be handled with the same algorithm
• Based on Separating Axis Theorem
Projection

- Imagine a light source with parallel rays
- Shape is between light source and axis
- “Shadow” cast on axis is shape’s projection onto that axis
Projection Demo

- Thin red line is axis
- Thick blue line is rectangle’s projection onto axis
Intersection

- Shapes are intersecting when ALL possible projections overlap
- No matter what direction you look from, you can’t see between the shapes
Separating Axis Theorem

• If two convex shapes are not overlapping, there exists an axis for which the projection of the shapes will not overlap.
• If we can find an axis along which the projection of the two shapes does not overlap, then the shapes aren’t colliding.
Caveat: Only Convex Shapes

• SAT only applies to convex shapes
  – Can’t draw a line between concave shapes, therefore no separating axis
• Don’t need to support concave polygons
• Compound shapes can be concave, but each component is convex
Separating Axes

• Consider an axis on which projections do not overlap
• There’s a line perpendicular to it that goes between the shapes
• This is the line of sight
Separating Axes

- Line of sight = a line that can be drawn between two separate shapes
- Separating Axis = axis perpendicular to that line, onto which shapes are projected
Finding Separating Axes

• If shapes are very close to colliding, a line parallel to an edge will always get through

• Minimal separating axis is always perpendicular to a shape’s edge
Finding Separating Axes

• Lines perpendicular to each shape edge are all the separating axes you need
  – Sometimes called “edge normals” (confusing)
• Consider each edge a vector, take perpendicular
General SAT Algorithm

1. Create a vector for each edge of each shape
2. Take the perpendicular vector to get a separating axis
3. For each axis, project both shapes onto it
4. If the projections don’t overlap, no collision
5. If the projections overlap on every axis, the shapes are colliding
Interactive Demo

- Thin green lines are separating axes
- Red lines are projections of triangle, blue lines are projections of circle
- When shapes intersect, all projections overlap
Special Cases: Circles

• What’s perpendicular to the edges of a circle?
• Take the vector from the center to the closest vertex of the polygon
• No perpendicular – this is a separating axis
Special Cases: AABs

- Four axes perpendicular to the edges of an AAB
- Two of them are parallel, why test four?
- For an AAB, separating axes are just x and y axes
Implementation Notes

• To construct vectors for polygon edges, iterate around points counterclockwise

• Two kinds of perpendicular: \((-y, x)\) and \((y, -x)\)
  – Make sure you’re consistent
Implementation Notes

• Remember to check BOTH polygons’ separating axes
  – Otherwise false positives

• Checking for overlap:
  \[ \text{min1} \leq \text{max2} \]
  \[ \text{&& \ min2} \leq \text{max1} \]
An interesting aside on SAT

• The SAT is actually N-dimensional…

• To check two N-d convex shapes:
  – Find separating axes (N-1 dimensions)
  – Project all points onto this axis
  – Use convex hull to get a (N-1)-D polygon
  – Run (N-1)-D SAT on these
  – Two N-d shapes overlap if all (N-1)-d projections overlap
LECTURE 5
Tips for Tou II
Representing Polygons

- Polygon is an ordered list of points
- Order must be consistent
  - Best to use counterclockwise
  - Essential for edge perpendiculars to work
- Life will be easier if Polygon is immutable
  - Adding/changing points may ruin ordering
- Make your own shape hierarchy
  - Java’s is terrible
Drawing Polygons

- **Graphics2D** has `drawPolygon`, but it's only `int` precision and has awkward parameters.
- Java AWT's **Polygon** (for `draw(Shape)`) is equally bad.
- **Path2D** is the "new" polygon:
  - Call `moveTo` for the first vertex.
  - Call `lineTo` repeatedly in the order of vertices, finishing at the first.
- Don't store a polygon as **Path2D**, just create one when you need to draw.

```java
public void fill(MyPolygon poly) {
    Path2D path = new Path2D.Float();
    Vec2f last = poly.getLastVertex();
    path.moveTo(last.x, last.y);
    for (Vec2f point : poly.getVertices()) {
        path.lineTo(point.x, point.y);
    }
    javaGraphics.fill(path);
}
```

//Alternatively, in a game entity...

```java
public void onDraw(Graphics2D g) {
    g.fill(polygon.toPath2D());
}
```
Sprites are Still Possible

• Multiple shapes are required, so you can’t build this game out of square sprites
• Give units a “bounding shape” that mimics visible outline of sprite
Behavior Trees are Useful

- Give the boss an AI instead of hardcoded behavior
- Keep track of the player’s current level/difficulty
- Create different attack patterns
Encapsulate Separating Axes

- Projection is tedious, can be messy
- A `SeparatingAxis` class is a good idea
  - project methods for each shape
- May also want a `Range` or `Projection` class

```java
public class SeparatingAxis {
    public SeparatingAxis(Vec2f direction) {...}
    public Range project(Circle c) {...}
    public Range project(AAB a) {...}
    public Range project(Polygon p) {...}
    public Range project(Compound c) {...}
}
```
Tips for Tou 2

JAVA TIP OF THE WEEK
Known Bad Cases

- Some things you know will make your code blow up
- Float values should never be NaN
- Vectors should never try to divide by zero
- Polygons should always be convex
Failures Can Be Unhelpful

• Your shapes suddenly disappear…but why?
• An object with a fraction threw an ArithmeticException…but when did the denominator get set to 0?
Solution: Asserts

- Part of Java language
- Evaluate a Boolean expression, throw an error if it’s false
- Optionally attach a message (will appear in thrown error)

```
assert <expression> : <message>;

class Fraction {
    private int numerator;
    private int denominator;
    ...
    void setDenominator(int d) {
        assert d != 0 : “Denominator was set to ” + d;
        denominator = d;
    }
}
```
Using Asserts

- Must be enabled at runtime to have effect
  - Use the -ea flag
  - Disabled by default
- Have no effect when disabled
  - OK to do expensive computations in asserts
Using Asserts

- When an assertion fails, it throws a `java.lang.AssertionError`.
- You can set an Exception Breakpoint on this in Eclipse to debug as soon as an assertion fails.
When To Use Asserts

- Checking invariants
- Validating control flow
  - `assert false` if control should never reach a line
- Checking preconditions
  - Exceptions better on public-facing methods

```java
private Vec2f computeNewPosition(Vec2f old) {
    assert old.x >= 0 && old.y >= 0;
    ...
}

public double average(List<Integer> nums) {
    if (nums.isEmpty())
        throw new IllegalArgumentException("Can’t average an empty list");
    ...
}
```
When To Use Asserts

- Checking postconditions
  - Even on public methods
- Checking data structure validity (class invariants)
  - Tree remains balanced
  - Polygon remains convex

```java
public class BalancedTree<T> {
    private boolean isBalanced() {
        ...
        return true; // Example return value
    }
    public void add(T data) {
        ...
        assert isBalanced();
    }
}

public class Polygon {
    public Polygon(Vec2f[] points) {
        ...
        assert isConvex();
    }
    private boolean isConvex() {
        ...
        return true; // Example return value
    }
}
```
Tips for Tou II

QUESTIONS?
GAME DESIGN 4

Difficulty
What is difficulty?

• Remember: games are “problem-solving activities approached with a playful attitude”

• So: the difficulty of solving the problems in the game determines how hard it is
Why does it matter?

• Too easy: your players get bored and quit
• Too hard: your players get frustrated and quit
• Just like controls: it makes or breaks your game!
Difficulty is a spectrum

acceptable different for every player!
Brief history of difficulty in games

• In the old days, games were designed to be hard to drive replay value (and profit!)

• After save systems were invented, the overall difficulty of games has decreased

• General trend has been easier to cater to the casual audience
Components of difficulty

• Learning curve
  – Depends on complexity of controls and mechanics
• Punishment for failure
• Difficulty of sub problems
  – How do I defeat this enemy?
  – How do I make a Tetris with these blocks?
What makes a problem difficult?

• Clarity of problem statement
• Clarity of having reached a solution
• Transparency of solution
• Difficulty of executing the solution
Fair vs. Unfair difficulty

In a fair game...
• The player is responsible for failure
• The player clearly understands the objective
• The player knows what they are capable of doing

In an unfair game...
• Random factors determine failure
• The player doesn’t know they’re trying to do
• The player doesn’t know what they can do
Case studies: Kirby

• Overall a fairly easy platformer
• Just enough challenge to be engaging
• Various solutions (different absorbable powers) make the game fun
Case studies: Super Meat Boy

• Difficult platformer featuring short levels
• Executing the solution is the primary source of difficulty
• Punishment for failure is tiny, player immediately respawns on death
• Get to see all failed attempts on success!
Case studies: Dark Souls

- Relentlessly difficult fantasy adventure
- Unlike most modern games, does not tell the player everything about the game
- As a result, the game feels mysterious and exploration feels rewarding
- Like Super Meat Boy, success feels fairly earned
Case studies: Silver Surfer

• Player’s collision shape is huge
• Player can collide with terrain
• One hit = death
• No continues
• Way too difficult, even for NES games
Case studies: Big Rigs

- Player collides with nothing
- Physics do not exist
- AI player cannot actually win
- Essentially no challenge
Case studies: Dwarf Fortress

- ASCII graphics, complex mechanics, and horrible UI create an incredibly steep learning curve
- Barrier to entry drives players away from an incredibly deep and engaging experience
Gauging the difficulty of your game

- As the programmer and designer, you know your game inside and out
- General rule: actual difficulty is always at least one step up than what you think it is
- Playtesting is the best way to test how hard your game is
- No playtesters? Play your game with various levels of effort (from lazy to tryhard)
Adjusting difficulty

Play with the following:

• Learning curve
  – How hard is it to learn to play the game?

• Degree of punishment for failure
  – How much is the player set back for messing up?

• Difficulty of subproblems
  – How hard is it to do things in the game?

• Number of distinct subproblems
  – How many different problems need to be solved?

• Clarity of problems (only if you know what you’re doing!)
  – How well does the player know what they’re supposed to be doing?
QUESTIONS?
TIPS FOR TOU 2
Difficulty for Tou 2

- General trend for cs1971 students: too difficult!
  - The TA’s have to be able to beat it for grading...
- Think from the player’s perspective – they will not know everything you do
- Lazy playthroughs are an easy way to check if your game is too hard
Variety is the key

• Variety is a great way to make your game more engaging without spiking the difficulty
• Enemy variety gives the player lots of interesting subproblems to solve
• Different weapon types give the player different ways to solve each problem
• Introduce the variety gradually — all at once will be overwhelming
Non-sticky controls

• Setting velocity on key press feel sticky
• Instead, keep track of a Set of Vec2f’s
• When a direction is pressed, add a normalized vector to the set, and remove it when the direction is released
  – E.g. right pressed $\rightarrow$ add (1,0) to the set
• On tick, sum the vectors in the set, normalize it, and multiply by the player’s speed
The usual good stuff

• Good controls 😊
  – Follow conventions, be ergonomic, etc.
• Think from the player’s perspective!
  – Always do this!
• Remember your audience: 1971 students and TA’s
Game Design Tips for Tou 2

QUESTIONS?
Tou 1 Playtesting!

Hooray!