Lecture 5

Announcements
Nin: Your last (solo) project

- Platformer game
  - Physics
  - More physics
  - Externalizing game logic
Announcements

QUESTIONS?
Lecture 5

Final Project Pitches
This Week

• Today in class: pitch final project ideas!
• Afterwards: talk to your classmates on Slack! Just use #general.
• Form groups (2-4 people per group)
  ○ Reach out to the TAs about working alone
• By **Saturday midnight**, fill out this [Google Form](#) with your group members
  ○ Also on the website
• We’ll reach out to you about scheduling a design check
Final Projects

QUESTIONS?
Griffin

- Among Us, but better!
  - Simple learning curve
  - Great party game
  - Clean and simple visuals

- Potential features
  - Procedurally generated maps that work with the Among Us design
  - Multiplayer (client-server interactions)
  - Minigames within screens
  - Modify existing map tiles (e.g. sabotaging)
Andrew

• Platformer/Metroidvania
  ○ Explore a hopefully predesigned map with a short story
  ○ A few upgrades which will allow you to explore hidden/new areas of the map
  ○ Maybe even a boss fight

• Potential features
  ○ Floating objects
  ○ Upgrades which can be picked up and improve your equipment
  ○ More extensive AI!
Daniel

- **Open-ended RPG**
  - Story-driven RPG with the freedom to do literally anything
  - Be a criminal and kill off important NPCs!
  - Or be the hero and save the world!
  - Or a bit of both!
  - Disclaimer: can be hard to code.

- **Potential features**
  - Engine that handles constant changes
  - Needs to have code that can handle anything the user does
Alexander

- **Top-down adventure game**
  - For each region, defeat enemies to protect a neighboring town
  - Returns to town between levels to upgrade stats, or move onto the next town for the storyline
  - Snappy combat, decent amount of strategy
  - Inputs are logged into queues and sequences of buttons do different attacks (like Smash)
  - Eventually be able to stack massive combos and mow down the enemy
  - Items!

- **Potential features**
  - Combat system for combos
  - Animations that animate different parts of the GameObject, and animation graphs/nodes
  - More than 1 collision component/animation component for each GameObject
  - TileMap for handling tiles
  - Rendering in order of game world position
  - Light sources
  - Full viewport effects (e.g. red tint for low HP)
Eliza

- **The Witch**
  - You're a witch living outside a protestant town in the late 1600s, and you've found the potion recipe to bring back the dead in an ancient book. The potion calls for the following ingredients:
    - 1 virgin's blood
    - 1 preacher's finger
    - 1 lock of adulterer's hair
    - 1 blasphemer's tongue
    - 6 ear lobes, all from different people than each other
  - To get the ingredients, you'll need to trap people by enchanting them into following you into your traps, or collect the ingredients while they are on the stocks (i.e. the adulterer and blasphemer are always out on the stocks, and you need to get to them without being caught).
  - Collect certain items from the forest to set your own traps. You can brew love spells to temporarily enchant villagers by collecting specific ingredients from the forest. You also need to avoid being caught by the town watch, who patrol the border between town and woods.
  - Might include something about only being able to leave the woods at night, where day and night are on regular cycles. If you get caught by the town watch, you're put on trial and hanged.
Eliza, cont’d

• The Witch
  ○ You have a lair in the woods which is your home base, with a cauldron in front where you brew spells, which you can then cast at any time (e.g. the love spell is a launched projectile and the sleep spell spreads out in a radius.)
  ○ You have a book on hand always which has the ingredients for every spell.
  ○ The woods can have abundant, re-spawning, collectible berries, herbs, and frogs that are brewed to make the basic spells, i.e. you don't need to worry about limited resources for these spells so much as brewing enough spells to do your main mission.
  ○ If you get all the ingredients and brew the resurrection potion, you win!
  ○ (Work in progress, story will be more fleshed out)

• Potential features
  ○ Inventory system
  ○ Lots of NPC behavior AI
  ○ Focus shift between panes
Giuse

• Small-scale RPG like The World Ends With You
  ○ Split-screen combat
  ○ Trends and styles of equipment for stat boosts

• Potential features
  ○ Networking/split-screen co-op
  ○ Combat encounters
  ○ Random enemy/boss behaviors
Miru

- Platformer
  - Hollow Knight/Celeste like game with tight controls and fast paced movement
  - Maybe some questing/equipment, but nothing too complicated
  - Prioritize being able to *schmove* your character

- Potential features
  - Whatever physics that isn’t in Nin
  - Questing system!
Cooper

• Binding of Isaac + Hollow Knight
  ○ Lots of looting
  ○ Platforming and story
  ○ Discovery, danger, death

• Potential features
  ○ Better physics than Nin
  ○ Fun attacks
  ○ Comprehensive system for loot
Jonathan

- Indie RPG
  - Like Undertale, Oneshot, or LISA
  - Storywriting
  - Well-designed gameplay mechanics

- Potential features
  - Inventory system and items
  - Parties
  - Music loaders
Jarrett

- Pirate ship that fight each other!
  - Controlled by 1 or 2 players
  - Progress through levels of increasing difficulty
  - Theoretically never-ending game
  - Variety of enemy ships based off of real historical pirate ships
  - Procedurally generated maps and enemies

- Potential features
  - Multiplayer
  - Cooperative vs Competitive
  - Expanded controls for the player
  - Customizable sprites
  - Unlockable sprites and power ups
Sal

• Roguelike fast-paced puzzle game
  ○ Procedural generation!! Lots of it
  ○ Each game is new and unexpected
  ○ Solve many easy puzzles as fast as possible
  ○ Downwell with fewer enemies and more about reaction time

• Potential features
  ○ Physics things not in Nin
  ○ Additional camera features (e.g. dynamic zooming)
Carlos

- 2D scrolling platformer
  - Like Celeste or Shovel Knight
  - Recreate/add movement mechanic (like dash) to make each level a puzzle
- Potential features
  - Whatever physics that isn’t in Nin
  - Level loading and progression
  - New collision system that allows event triggering
• Dungeon crawler: fight through monster-infested woods to find your friend
  ○ Inspired by Chinese mythology, play as a dragon prince who can change into a dragon at will
  ○ Add snake (yes, the 1976 game) mechanics to dungeon combat
  ○ Add a home base where you can purchase upgrades for your attacks and abilities!
  ○ Aesthetically, simplistic, nostalgic pixel-art style similar to Link to the Past/Minish Cap-era Zelda games, or Mother 3.

• Potential features
  ○ Ability to transfer player control
  ○ Randomly generating obstacles
  ○ Permanent and temporary upgrades
Lecture 5

Collection Detection II
Polygons

- Lots of places that could collide (overlap)
- Irregular shapes
- Can’t test every screen 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**
  - Has an interior angle > 180°
  - There exists a line through the shape that crosses more than twice
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
Point in Polygon

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

Cross-product $< 0$

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 negative, point is inside
6. If any cross-product is positive, it’s outside
Shape-Polygon

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

• 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

- Projection axis not necessarily horizontal or vertical
- Dashed line is the axis
- Red and blue lines are projections of the shapes
Intersection

- Two shapes intersect when ALL possible projections overlap
  - No matter what direction you look from, you can’t see between the shapes
- But there are infinitely many projections!
  - We’ll show we only need to project against each shape’s edge’s perpendicular.
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
- In this class, you don’t need to support concave polygons
- You can make compound shapes that are concave, but each component is convex
Separating Axis

- 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 Axis

- 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”
- Consider each edge a vector, and take the perpendicular
- Each green arrow here represents a separating axis
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 there exists an axis without an overlap, there is no collision
5. If the projections overlap on every axis, the shapes are colliding
Special Cases: Circles

- Take the vector from the center to the closest vertex of the polygon
  - A line of sight would have to pass between them
- No perpendicular – this vector is already 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 counter-clockwise

• Direction of the vector matters!
  ○ There are 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
  - \( \text{min1} \leq \text{max2} \) && \( \text{min2} \leq \text{max1} \)
  - Should be the same from your AAB
    Interval overlaps
QUESTIONS?
Collision Detection II

POLYGON MTV
Recap: MTV in one dimension

- In 1D, convex shapes are line segments (intervals)
- These have a 1D MTV
  - Similar to overlap
  - But it has a sign
- Write a method that computes this
- Use it to find shapes’ MTV
Computing MTV

1. For each axis, project both shapes onto that axis, and find the 1D MTV of their projections
   ○ Make sure to normalize the axis before projection
2. Find the axis with the minimum 1D MTV
3. The 2D MTV is the minimum 1D MTV times that (normalized) axis

Note: normalizing and projection are in the Vec2d source code!
Computing intervals’ MTV

// use Double instead of double so we can return null

Double intervalMTV(Interval a, Interval b):
    Double aRight = b.max - a.min
    Double aLeft  = a.max - b.min
    if aLeft < 0 || aRight < 0:
        return null
    if aRight < aLeft:
        return aRight
    else:
        return -aLeft
Computing polygons’ MTV

Vec2d shapeMTV(Shape a, Shape b):

Double minMagnitude = +infinity
Vec2d mtv = null

for Vec2d axis in allAxes:
    axis = norm(axis)
    Double mtv1d = intervalMTV(a.proj(axis), b.proj(axis))
    if mtv1d is null:
        return null
    if abs(mtv1d) < minMagnitude:
        minMagnitude = abs(mtv1d)
        mtv = axis.smult(mtv1d)

return mtv
Recap: Special Cases

• Circle vs Polygon
  ○ Use these axes:
    - Polygon’s edge normals
    - Vector from circle center to closest vertex

• AAB vs Polygon
  ○ Use these axes:
    - Polygon’s edge normals
    - x axis and y axis
QUESTIONS?
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 hyperplane
  - 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

Physics II
Physics II

VELOCITY & ACCELERATION
Velocity

- Rate at which position changes
  - $\Delta x/\Delta t$
- Can’t increment position each tick
  - Frame rates vary
- Multiply by time elapsed
  - $\text{pos} += \text{vel} \times t$
Acceleration

- The rate that velocity changes
  - $\Delta v / \Delta t$
- Useful for gravity, springs, wind, etc.
  - vel += acc * t
- Fun fact: the next derivatives of position are called jerk, snap, crackle, and pop
Which order to update?

Position first (Euler)

\[
\begin{align*}
pos &= \text{vel} \times \text{time} \\
\text{vel} &= \text{acc} \times \text{time}
\end{align*}
\]

Velocity first (Symplectic Euler)

\[
\begin{align*}
\text{vel} &= \text{acc} \times \text{time} \\
pos &= \text{vel} \times \text{time} \\
&\quad \cdot \text{more stable, use this}
\end{align*}
\]
Physics II

COLLISION RESPONSE
Changing velocity for collision

- Just reverse the object’s velocity?
  - \( \text{vel} = -\text{vel} \)
- Reverse the y component?
  - \( \text{vel.y} = -\text{vel.y} \)
  - Doesn’t work if we bounce against a wall!
Two moving-object collisions

• Reverse both velocities?
• Doesn’t always work
• **Apply equal and opposite impulses**
  ○ An impulse is an instantaneous force (also known as the change in momentum)
  ○ Objects will change their own velocities based on mass, so heavier objects will see less acceleration
  ○ Simulate real-world physics
Units (throwback to AP Physics C or IB HL Physics)

**Without mass**
- position \( \vec{x} \)  
- velocity \( \vec{v} = \Delta \vec{x} / \Delta t \)  
- acceleration \( \vec{a} = \Delta \vec{v} / \Delta t \)  
- (no equivalent)

**With mass**
- (no equivalent)
- momentum \( \vec{p} = m \vec{v} \)
- force \( \vec{F} = \Delta \vec{p} / \Delta t \)
- impulse \( \Delta \vec{p} \)

\( m \) for position, velocity, and acceleration, and \( \text{kg} \cdot \text{m/s} \) for momentum, force, and impulse.
Implementing force and impulse

- `applyForce()` accumulates force
- `applyImpulse()` accumulates impulse
- `onTick()` applies force and impulse, clearing them for next frame
- Static (immovable) objects shouldn’t be affected by `applyForce()` and `applyImpulse()`

```
class PhysicsComponent {
    double mass;
    Vec2d pos, vel;
    Vec2d impulse, force;

    void applyForce(Vec2d f) {
        force += f;
    }

    void applyImpulse(Vec2d p) {
        impulse += p;
    }

    void onTick(float t) {
        vel += t*force/mass + impulse/mass;
        pos += t*vel;
        force = impulse = 0;
    }
}
```
Building a gravity component

- Use `applyForce()` to apply a gravitational force
  - Newton's second law: $F = ma$
  - $F_g = \text{mass} \times \text{gravitational constant}$
- Don’t apply as an impulse; otherwise, fall speed will be dependent on the frame rate
- Play around with $g$: if things are falling too slowly, crank it up
  - Watch videos of other platforming games to get a sense of natural movement
Impulse collision response

- Translate objects out of collision
  - Each by MTV/2
  - Or proportional to mass in direction of MTV
- Apply some impulse proportional to MTV to each object
  - Details in next section
QUESTIONS?
Note about Velocity

- When working with collisions, we only care about the velocity in the direction of the collisions.
- Your engine is 2D, so your velocities are 2D vectors.
- For all the math in the next slides, we’ll need our velocity as a scalar.
  - We only want the component parallel to the MTV.
- To do this, take the dot product of the velocity and the normalized MTV.
Restitution

- Property of physical entities
- Amount of energy lost in collision
- Value between 0 and 1
  - 0 is perfectly inelastic (objects stick together)
  - 1 is perfectly elastic (momentum entirely preserved, e.g. pool balls)
- The coefficient of restitution (COR) between two entities is the geometric mean of their restitutions: \( \sqrt{r_1r_2} \)
Correct Collisions

- How do we find the physically correct collision response?
- i.e. given $u_a$ and $u_b$, what are $v_a$ and $v_b$?
  - $u$: initial velocity
  - $v$: final velocity
  - $a$ and $b$: objects
- Use physical definition of the COR:
  - $\frac{v_b - v_a}{u_a - u_b}$
Final Velocities: finding $v_a$ and $v_b$

- Conservation of momentum:
  \[ m_a u_a + m_b u_b = m_a v_a + m_b v_b \]

- Solving for $v_a$
  \[ v_a = \frac{m_a u_a + m_b u_b - m_b v_b}{m_a} \]

- Substitute COR equation for $v_b$
  \[ \text{COR} \cdot (u_a - u_b) + v_a = v_b \]
  \[ v_a = \frac{m_a u_a + m_b u_b - m_b \left( \text{COR} \cdot (u_a - u_b) + v_a \right)}{m_a} \]
Final Velocities: finding $v_a$ and $v_b$

- Solve again for $v_a$:
  - \[ v_a = \frac{m_a u_a + m_b u_b - m_b \text{COR} \cdot (u_a - u_b) - m_b v_a}{m_a} \]
  - \[ v_a \left(1 + \frac{m_b}{m_a}\right) = \frac{m_a u_a + m_b u_b - m_b \text{COR} \cdot (u_a - u_b)}{m_a} \]
  - \[ v_a = \frac{m_a u_a + m_b u_b - m_b \text{COR} \cdot (u_a - u_b)}{m_a + m_b} \cdot \frac{m_a}{m_a + m_b} \]
  - \[ v_a = \frac{m_a u_a + m_b u_b - m_b \text{COR} \cdot (u_a - u_b)}{m_a + m_b} \]
  - \[ v_a = \frac{m_a u_a + m_b u_b + m_b \text{COR} \cdot (u_b - u_a)}{m_a + m_b} \]
Final Velocities: finding $v_a$ and $v_b$

- Similarly for $v_b$
  
  $$v_b = \frac{m_a u_a + m_b u_b + m_a \text{COR} \cdot (u_a - u_b)}{m_a + m_b}$$
Final Velocities

- Can’t just set velocities directly to $v_a$ and $v_b$!
  - Might interfere with other collisions (that happen later on the same tick)
- Use impulse instead
  - Instead of calculating $v_a$ directly, we just find the impulse that needs to be applied
Velocity Difference

- Impulse causes a change in velocity: we want to change from \( u \) to \( v \)
  \[ I_a = m_a (v_a - u_a) \]

- To find the change in velocity, use our previous equation for \( v_a \)
  \[ v_a - u_a = \frac{m_a u_a + m_b u_b + m_b \text{COR} \cdot (u_b - u_a) - u_a (m_a + m_b)}{m_a + m_b} \]
  \[ v_a - u_a = \frac{m_b u_b + m_b \text{COR} \cdot (u_b - u_a) - u_a m_b}{m_a + m_b} \]
  \[ v_a - u_a = \frac{m_b (u_b + \text{COR} \cdot (u_b - u_a) - u_a)}{m_a + m_b} \]
  \[ v_a - u_a = \frac{m_b (u_b - u_a) (1 + \text{COR})}{m_a + m_b} \]
Final Impulse

- Multiply the mass back into our impulse equation:
  \[ I_a = \frac{m_a m_b (u_b - u_a) (1 + \text{COR})}{m_a + m_b} \]

- Same process for \( v_b \):
  \[ I_b = \frac{m_a m_b (u_a - u_b) (1 + \text{COR})}{m_a + m_b} \]

- Note that neither equation depends on final velocities; we can calculate them directly from the **known values** mass, initial velocity, and COR \( \sqrt{r_1 r_2} \)

- Implement these equations and apply these impulses upon collision!
Static Shapes

- If a is static, then you can treat it as if it had infinite mass
- Take the limit of $I_b$ as the mass of a goes to infinity

$$I_b = m_b(u_a - u_b)(1 + \text{COR})$$

- Vice-versa if b is static
- You should special-case this
Putting it all together

Physically correct collision response:

1. Calculate COR with the restitution properties of the shapes
2. Project velocities onto MTV (which become $u_a$ and $u_b$)
3. Apply impulse formula to calculate impulses
4. Apply corresponding impulse to each shape’s PhysicsComponent
Reminders

- Directions matter! Be consistent about which way is positive/negative.
- Our `project0nto()` method gives a vector.
- Our `dot()` method gives a scalar.
- For physics equations, you want a scalar velocity, so use `dot()`.
- To use `dot()` for projection, you need to give it a normalized vector.
QUESTIONS?
Lecture 5

Tips for Nin I
More Accurate Sprites

- Use polygons to give units a “bounding shape” that mimics the visible outline of sprite
Behavior Trees/GOAP

- Give the enemies an AI instead of hardcoded behavior
- Keep track of the player’s current level/difficulty
**Axis / Range Classes**

- Projection is tedious, can be messy
- An `Axis` class is a good idea
  - project methods for each shape
- May also want an `Interval` or `Projection` class
- `Intervals` should be able to compare themselves to other `Intervals`

```java
public class Axis {
    public Axis(Vec2f direction) {...}
    public Interval project(Circle c) {...}
    public Interval project(AAB a) {...}
    public Interval project(Polygon p) {...}
    public Interval project(Compound c) {...}
}
```
Fixed Timestep

- Collisions can break if you have a really long tick
- This might happen, so consider:
  - Give the physics world some constant time on each tick
  - Tick as many times as possible on each game tick
  - Called separately from tick() and lateTick()
Known Bad Cases

• Some things you know will make your code blow up
• Floating point values should never be NaN
• Vectors should never try to divide by zero
• Make sure there are never any NaNs
• Polygons should always be convex
• NaNs will ruin your life
• Vectors have isNaN() methods
• Use assert liberally
Tips for Nin I

QUESTIONS?
Lecture 5

Difficulty
What is difficulty?

- Games are “problem-solving activities approached with a playful attitude”
- 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
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 clear a line in 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
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)
- Remember: TAs need to be able to beat your game in order to grade it
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?
‘Til Next Week!

• Nin I released today
• Final groups due by Saturday (10/24)