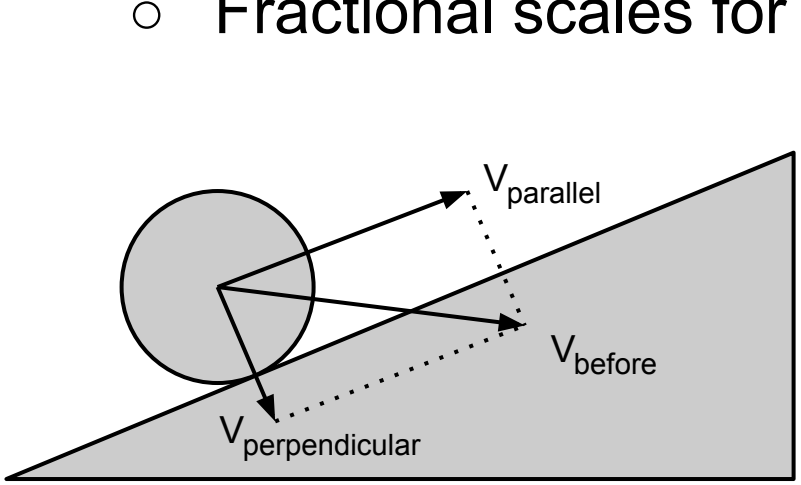

Collision Response for
Player Movement

Introduction

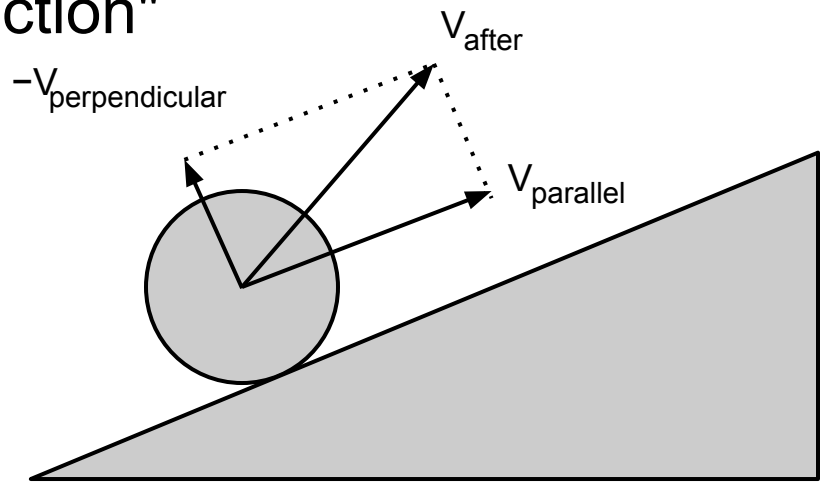
- Where we left off
  - World modeled using triangles
  - Player approximated with ellipsoid
  - Sweep test: find where ellipsoid intersects world when moving from A to B
- Need to respond to collision
  - Bounce, slide, ...
- Remember, goal is not to be physically accurate!
  - Player as an ellipsoid isn't physically meaningful anyway
  - Want to respond how the player intends to move

Bouncing

- Split velocity into components parallel and perpendicular to the contact plane
  - Keep parallel component the same
  - Negate the perpendicular component
  - Fractional scales for "friction"



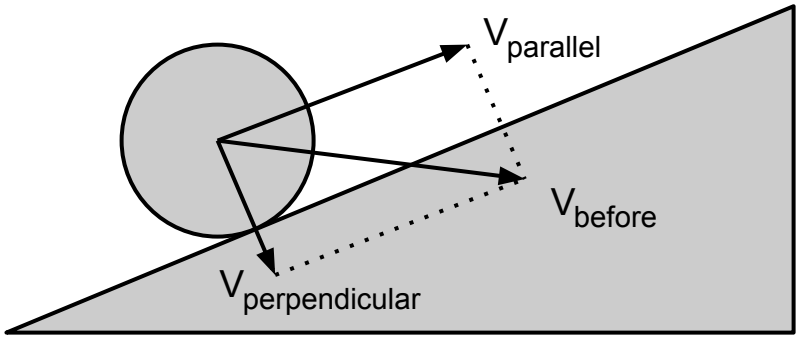
Before



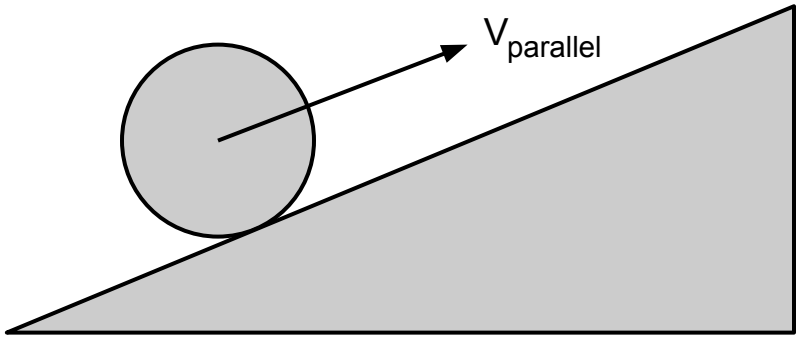
After

Sliding

- Split velocity into components parallel and perpendicular to the contact plane
  - Keep parallel component the same
  - Zero the perpendicular component



Before



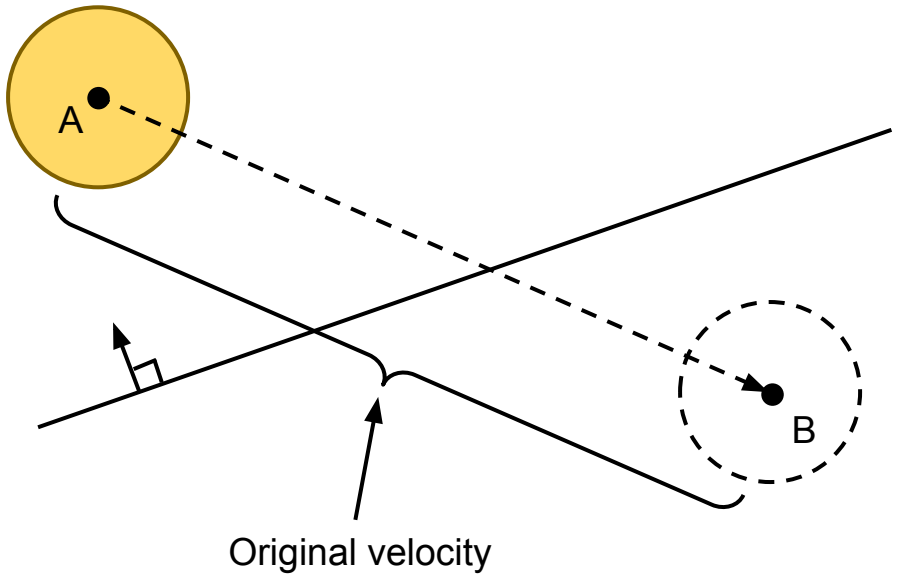
After

Sliding for Player Movement

- Move player up to contact point using time of collision
  - Don't want to stop the player, then they are stuck
  - Slide the player instead
- Project the rest of the velocity onto the contact plane
  - Make sure not to use the original velocity, or players will speed up in collisions!
- Move player again with another collision test
  - This is recursive, although only need 2-3 iterations

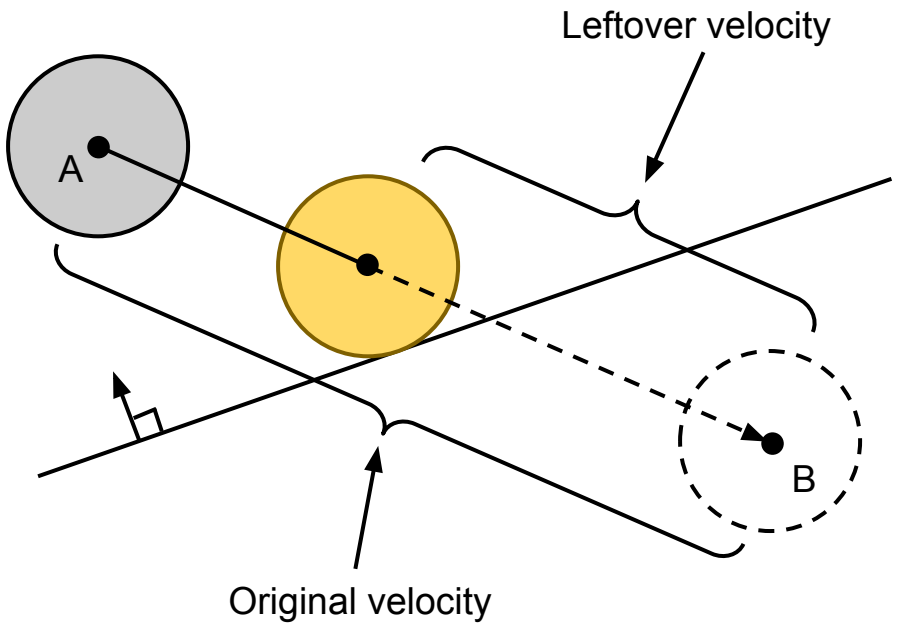
Sliding for Player Movement

- Iteration one of player sliding algorithm



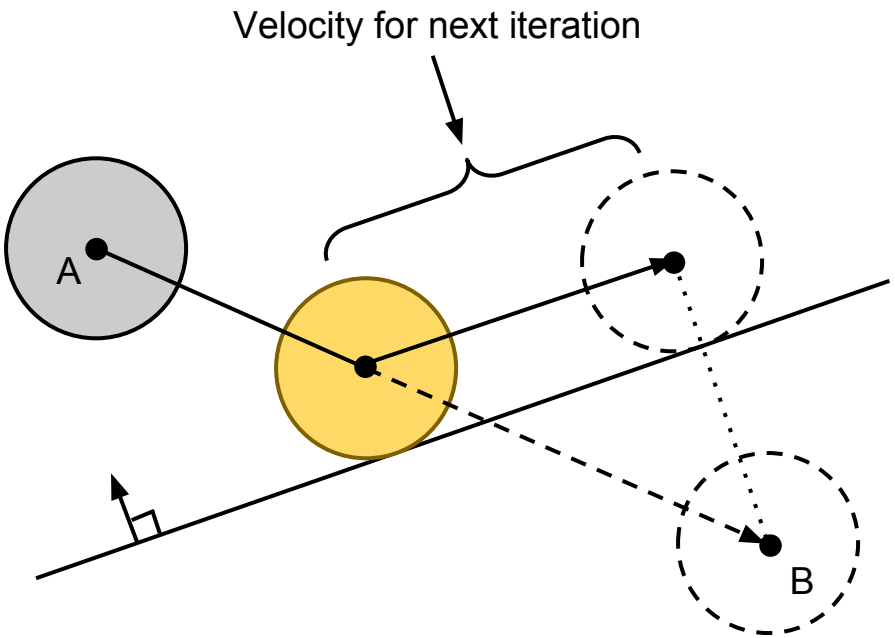
Sliding for Player Movement

- Iteration one of player sliding algorithm



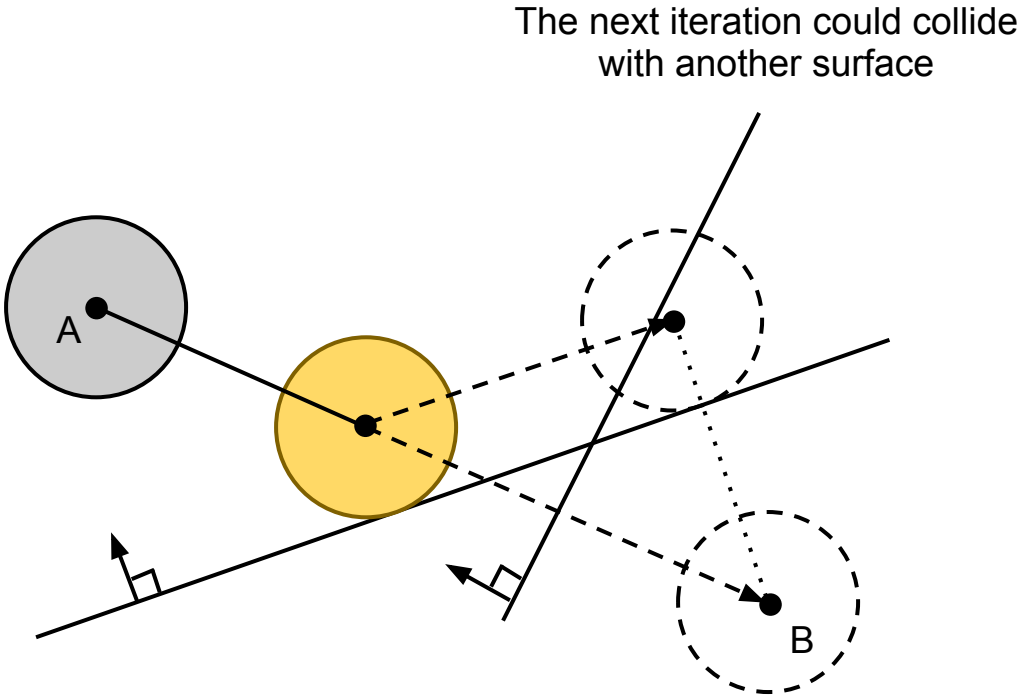
Sliding for Player Movement

- Iteration one of player sliding algorithm



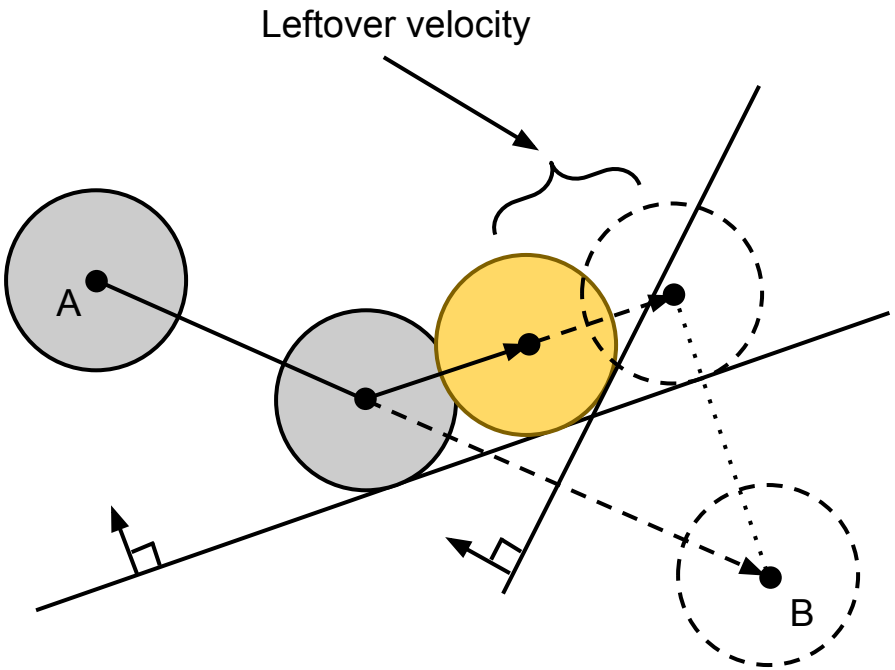
Sliding for Player Movement

- Iteration two of player sliding algorithm



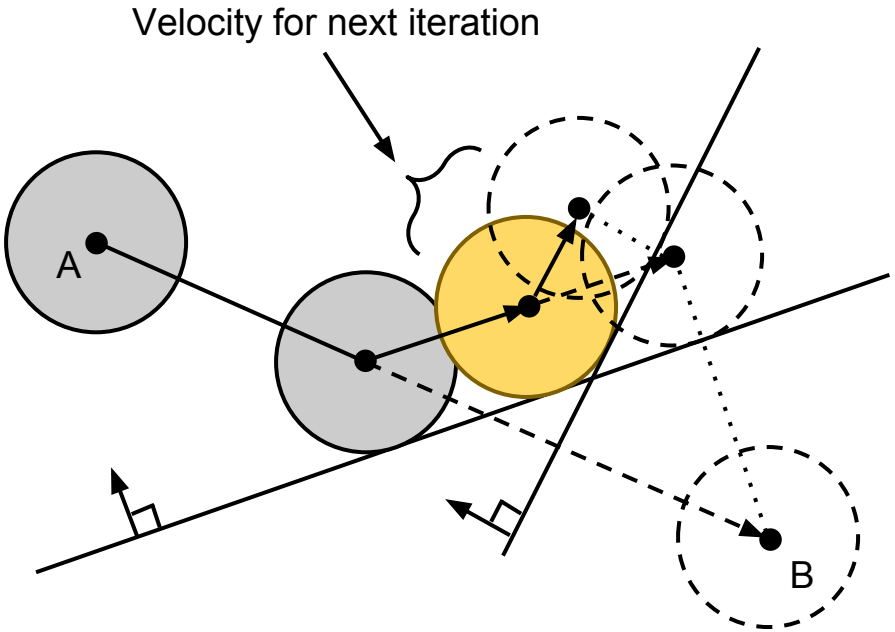
Sliding for Player Movement

- Iteration two of player sliding algorithm



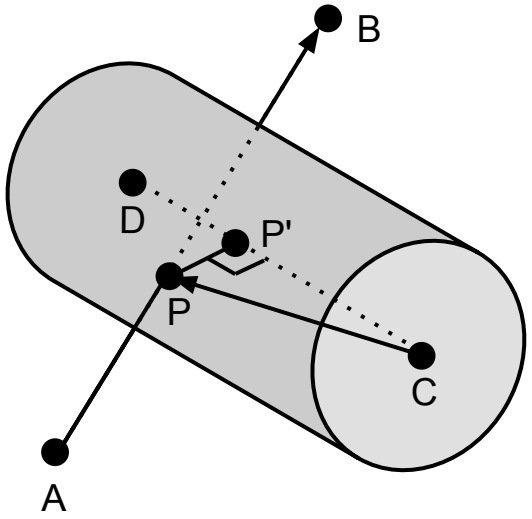
Sliding for Player Movement

- Iteration two of player sliding algorithm



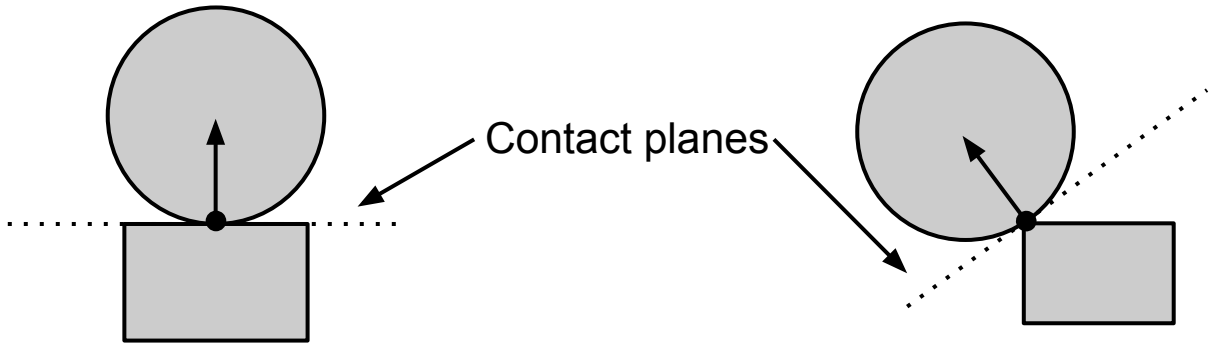
Computing the Contact Point

- Split into three cases
  - Interior: Use ray-plane intersection point P
  - Vertex: Use vertex position
  - Edge: Project vector from edge to intersection point P onto edge to get contact point on edge P'
  - Remember to do this in sphere space convert to ellipsoid space



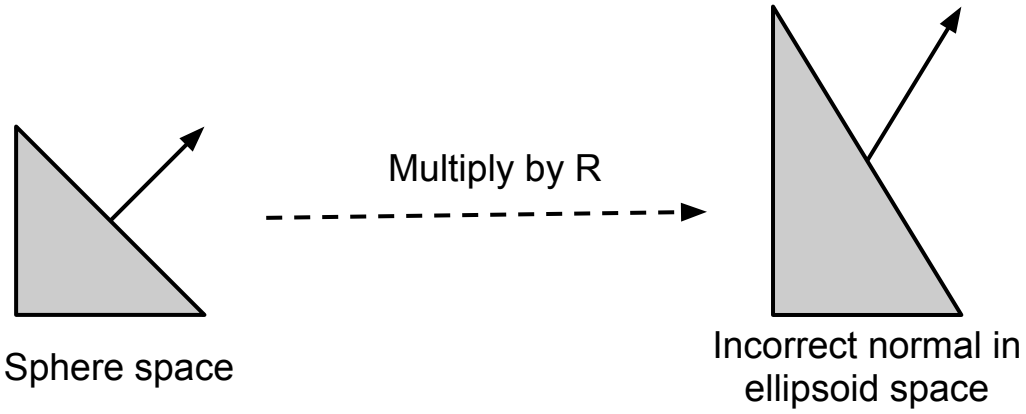
Computing the Contact Plane

- Plane through contact point and tangent to ellipsoid
  - Contact plane isn't always the triangle plane
- Computing the contact normal
  - R = ellipsoid radius vector (r_x, r_y, r_z)
  - E / R = ellipsoid center in sphere space
  - P / R = contact point in sphere space
  - Normal in sphere space: E / R − P / R = (E - P) / R



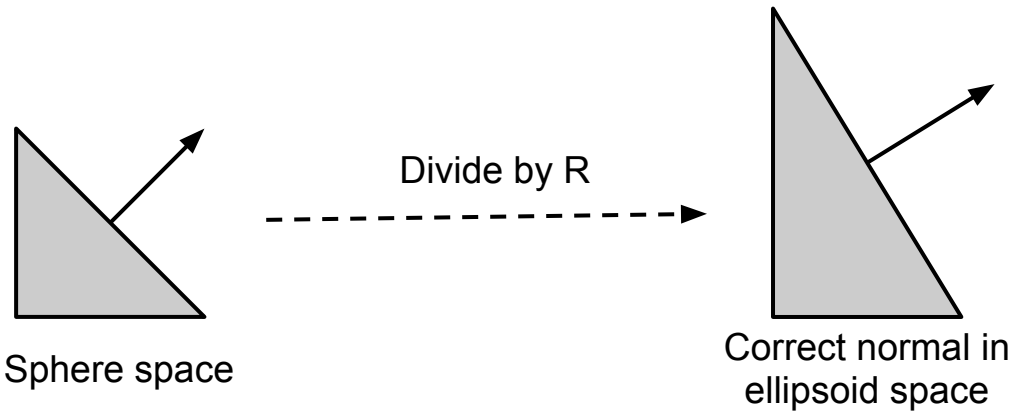
Contact Plane in Ellipsoid Space

- Currently we have the contact normal in sphere space
  - Need to convert to ellipsoid space
- First attempt: Multiply by ellipsoid radius R
  - Works for points
  - Does not work for vectors!



Contact Plane in Ellipsoid Space

- Instead, transform normal by inverse-transpose matrix
  - Results in dividing by R instead of multiplying
  - Normal in ellipsoid space = ((E − P) / R) / R

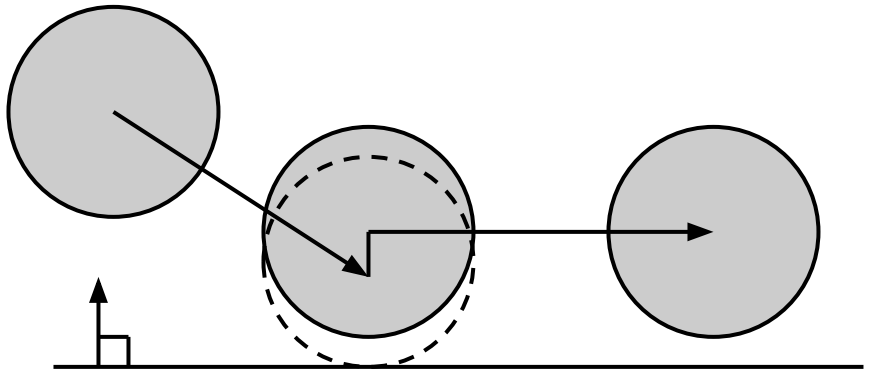


Precision Issues

- Sliding method as described doesn't work!
  - Due to limited floating-point precision, ellipsoid unlikely to be exactly on plane
  - May end up slightly above or slightly penetrating
  - If slightly penetrating, collision detection may fail
  - Player may slip through the world

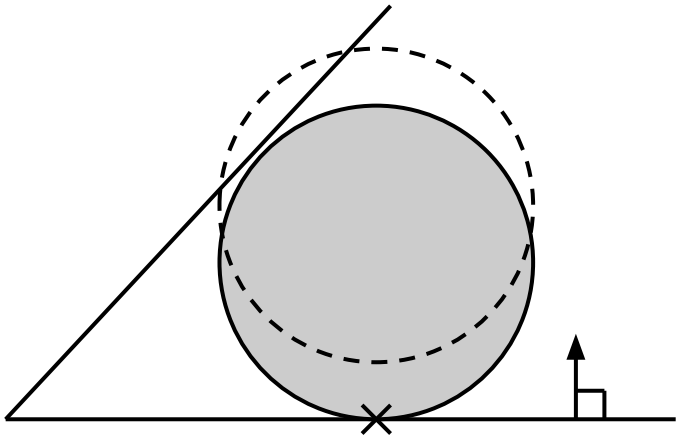
Precision Issues

- Solution
  - Push player a small amount away after every collision
  - Push along contact normal
- Does this always work?



Precision Issues

- Nope, still doesn't always work
  - Fails in corners that aren't obtuse
  - Player may be pushed into another triangle!



Precision Issues

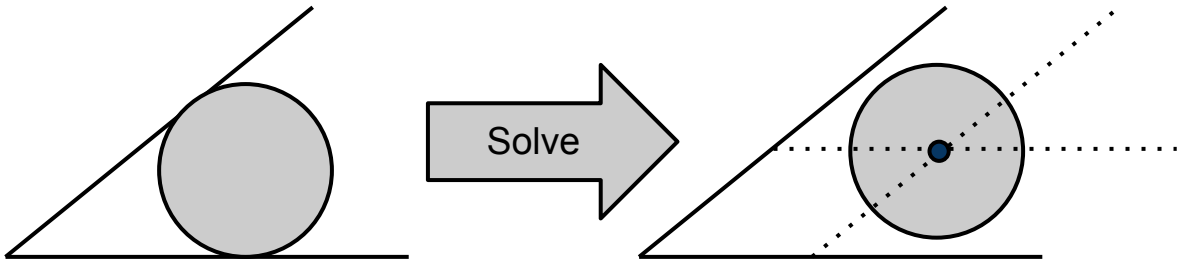
- Need to push player away from multiple constraints simultaneously
- This case also has iterative and analytic solutions
  - We'll talk about an analytic solution next, but you don't have to implement it

Analytic Player Offset

- Find all triangle interiors the player is intersecting
  - We are ignoring vertices and edges
  - Project player center onto plane of triangle and see if it's inside triangle bounds
- Create a set from their planes
  - Adjacent triangles with identical planes only contribute one plane
- Solve for new position given old position and set of planes

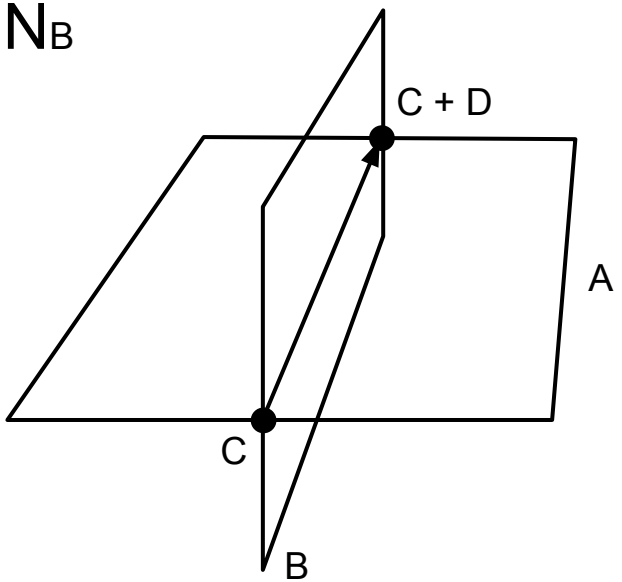
Analytic Player Offset

- Only really need to handle cases with 2 and 3 planes
  - Four or more won't happen in any reasonable world



Plane-Plane Intersection

- Given planes as 4-vectors:
  - $A = (N_A, d_A)$, $B = (N_B, d_B)$
- The line of intersection C + D * t is:
  - $k = N_A \bullet N_B$
  - $C = (N_A * (d_A - d_B * k) + N_B * (d_B - d_A * k)) / (1 - k^2)$
  - $D = N_A \times N_B$



Plane-Plane Intersection

- Can use this to solve for correct player offset
  - 2 planes: Project player position onto line of intersection of both planes
  - 3 planes: Set player position to intersection of first plane and line of intersection of other two planes
- All planes are offset by sphere radius + epsilon along their normals from the triangles they came from

Recap

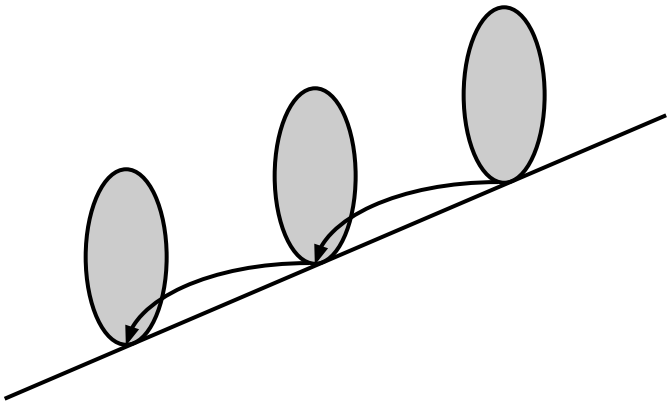
- We now have collision detection and analytic collision response for an ellipsoidal player
  - Yay!
- But we're not done yet...
  - We can tweak our collision response to better handle certain scenarios (ramps, ladders, etc.)
  - We'll now present a few of these improvements (hacks) that are present in our platformer demo

Problem: Sliding While Standing

- Player should not accelerate down slopes due to gravity
- Normally this is handled by static friction
  - Force resisting relative motion of two surfaces in contact
- Fix: Hack to simulate static friction
  - Split movement into horizontal/vertical
  - Only do sliding for horizontal movement
  - May want sliding on steep slopes, could conditionally enable sliding on vertical movement by slope steepness

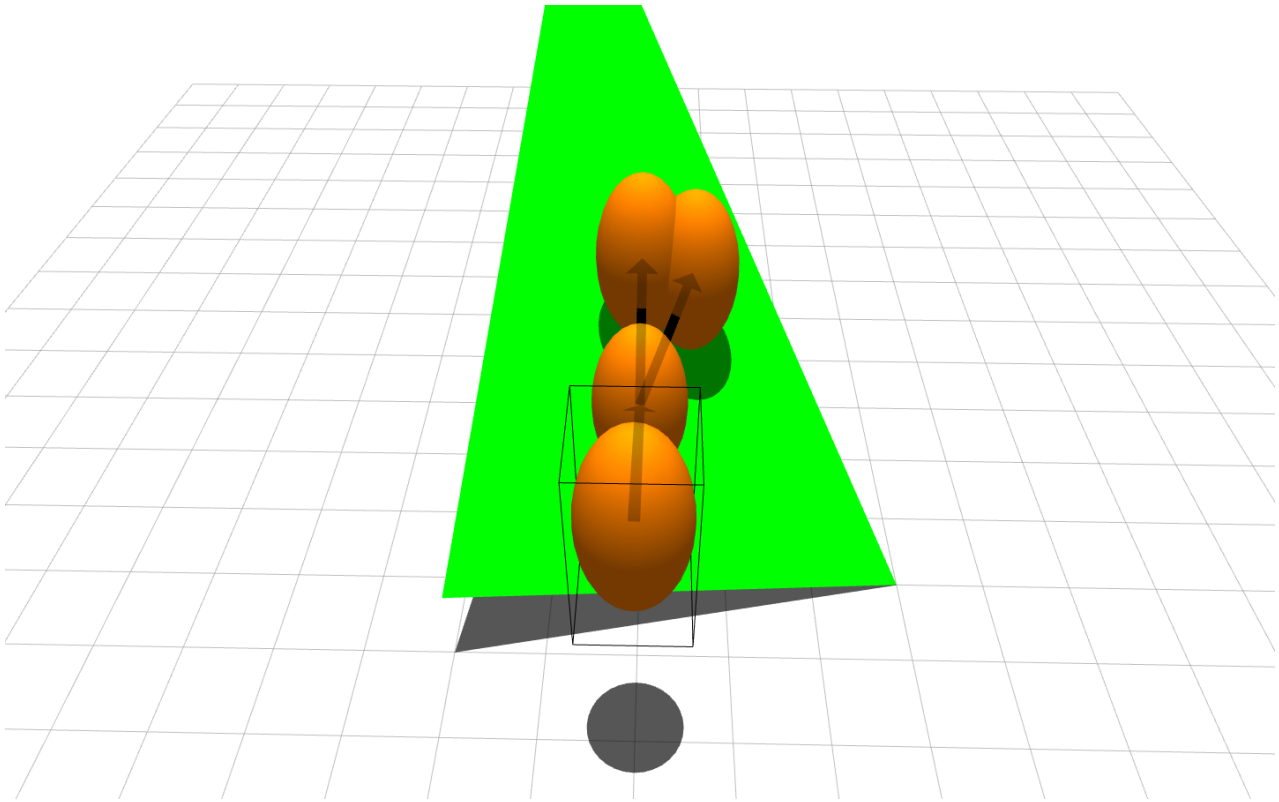
Problem: Bouncing Down Ramps

- Player bounces if up/down velocity is set to 0 on contact
- Fix: Set up/down velocity to small negative number when on ground instead



Problem: Deflection on Ramps

- Projecting the velocity leads to deflection



Problem: Deflection on Ramps

- Projecting the velocity leads to deflection
  - Player's intent is to move in a straight line
  - Current algorithm projects target sliding position to closest point on contact plane
- Actually want to set target sliding position to point on plane straight ahead of player instead
  - We are requiring this hack for your week 2 handin

Problem: Deflection on Ramps

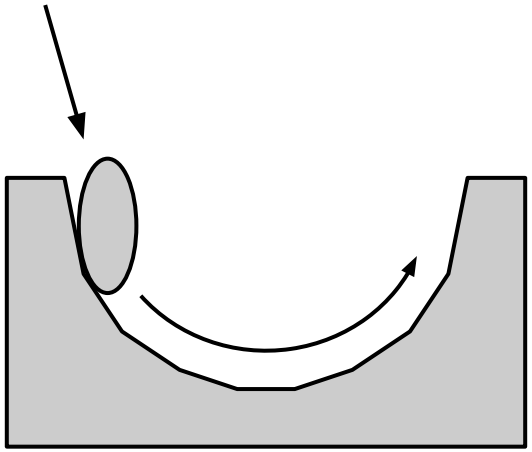
- Fix: Modify horizontal movement
  - Slide toward point on contact plane straight ahead but with same length as old deflection method
  - Given leftover velocity V and contact normal N:
    - New velocity direction = $V - (0, 1/N.y, 0) * (N \bullet V)$
    - New velocity length = $||V - N * (N \bullet V)||$
- No sliding for vertical walls (N.y == 0)
  - Can't move any further while moving straight
  - How to fix this:
    - Set new velocity = $V - N * (N \bullet V)$ like before only for vertical walls

Surface Types

- What if we want to add a ladder?
  - Or an icy surface, sticky surface, rotating platform, etc.
- Annotate faces as special surfaces
- Surface type returned as part of collision detection
- Use surface type to adjust collision response

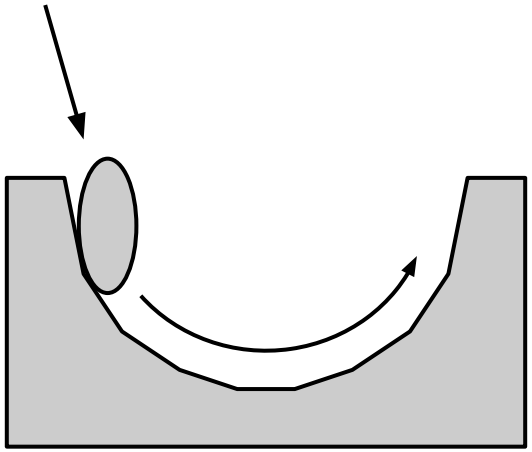
Frictionless Surfaces

- Modeling frictionless surfaces
  - Half pipe: player should have same speed on enter and exit
  - Must approximate curved surfaces with flat polygons when using polygonal collision detection
- Problem: Projecting the velocity when sliding loses energy



Frictionless Surfaces

- Fix: Force energy conservation
  - Same velocity direction as before
  - New velocity length = $||V_{original}||$
- Don't want to always redirect velocity
  - Hack is only to correct for polygonal approximation
  - Shouldn't redirect when hitting surface head-on
  - Only when glancing off surface ($N \bullet V > -0.5$)



Case Study: Hacks in Real Games

- Hack to fix player-world intersections
  - If player-world intersection is detected, try nudging slightly along all 3 axes to see if player is now free
  - Used in Quake 2 by id Software
- Simple method for climbing over steps and small objects
  - Before moving, lift player up a small amount
  - After moving, drop player down that same amount (unless player intersects an object)
  - Used in the game MDK2 by BioWare

Case Study: Hacks in Real Games

- Easy collision response against a heightfield
  - p.y = max(p.y, terrainHeight(p.x, p.z))
  - Used in many RTS engines
- Maximize floating-point precision
  - Floats have more precision near the origin
  - World is divided into sectors, each with its own local coordinate space
  - When player moves between sectors, objects positions are updated to the new origin
  - Used in Dungeon Siege by Gas Powered Games

Conclusion

- Collision response is a pile of hacks
  - Optimal player movement is not physically correct
  - Floating point precision is tricky
  - What we presented is definitely not the only way to do it

Platformer: Week 2

Demo

C++ Tip of the Week

- Template metaprogramming
  - The C++ type system is Turing complete (i.e. can be used for computation)
  - Discovered by accident during C++ standardization
  - Compile-time programming: programs generating programs
  - Abuses template specialization
- C++ templates are a functional language
  - Recursion instead of iteration
  - Immutable variables
    - Create a variable that holds a type via `typedef`
    - Create a variable that holds an int via `enum`

C++ Tip of the Week

- Simple example: compile-time factorial

```
// Recursive template for general case
template <int N> struct factorial {
    enum { value = N * factorial<N - 1>::value };
};

// Use template specialization for base case
template <> struct factorial<0> {
    enum { value = 1 };
};

int result = factorial<5>::value; // == 5*4*3*2*1 == 120
```

C++ Tip of the Week

- Another example: compile-time linked list

```
// Compile-time list of integers
template <int A, typename B> struct node {
    enum { num = A };
    typedef B next; };
struct end {};

// Compile-time sum function
template <typename L> struct sum {
    enum { value = L::num + sum<typename L::next>::value }; };
template <> struct sum<end> {
    enum { value = 0 }; };

typedef node<1, node<2, node<3, end> > > list123;
int total = sum<list123>::value; // == 1 + 2 + 3 == 6
```

C++ Tip of the Week

- Drawbacks
  - Much longer compile times (computation via template instantiation is inefficient)
  - No debugger, only page-long error messages
  - Turing completeness brings the halting problem

```
// This code will infinite-loop the compiler
template <typename T> struct loop {
    loop<T*> operator->();
};
loop<int> i, j = i->fail;
```