Class 3

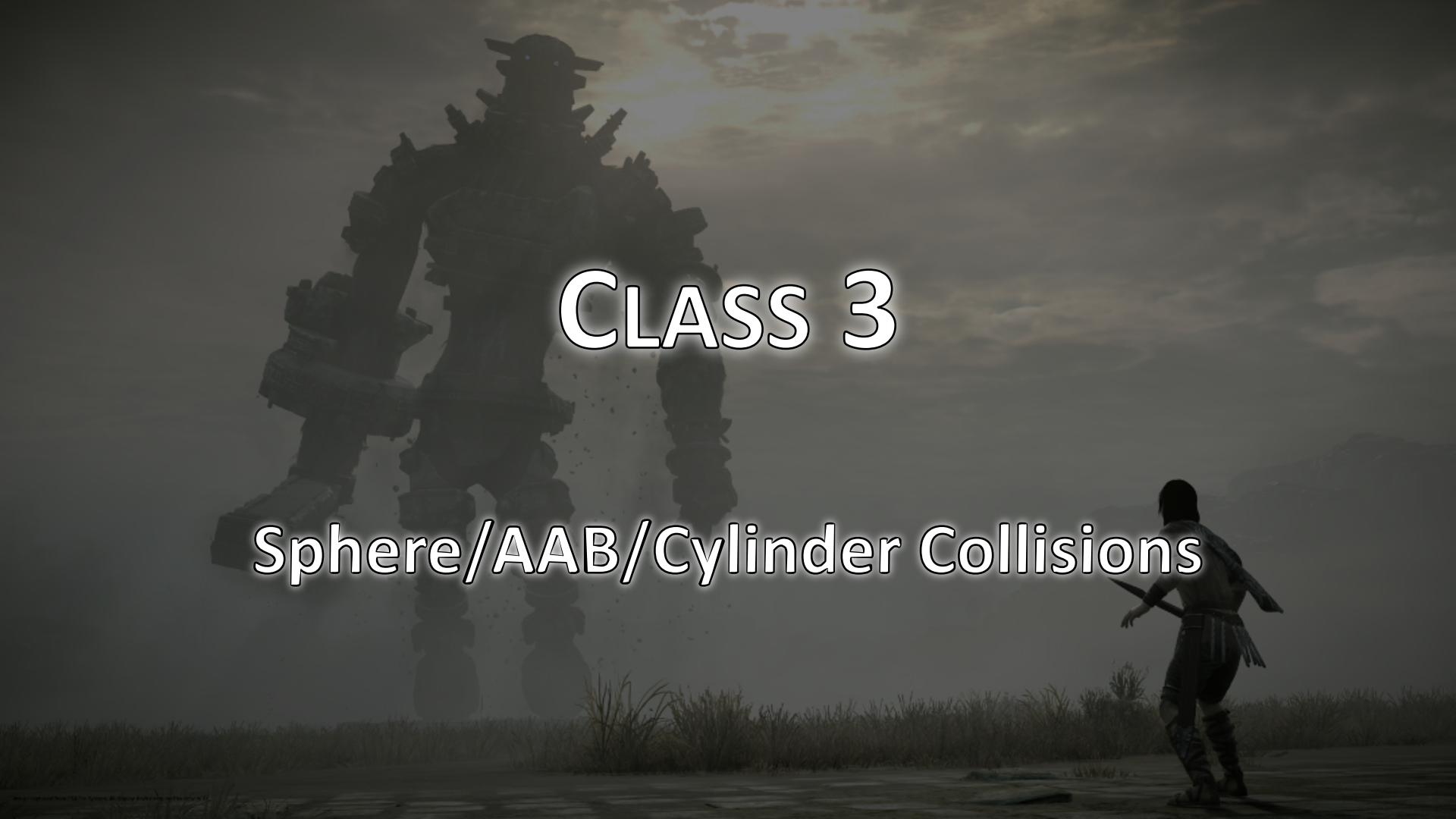
Ellipsoid/Triangle and Sphere/AAB/Cylinder Collisions

Platformer

- A game that minimally involves platforms
- Not based on any game in particular
  - Super Mario 64?
  - Team Fortress 2?
- Completely up to you to make unique gameplay

Platformer

QUESTIONS?

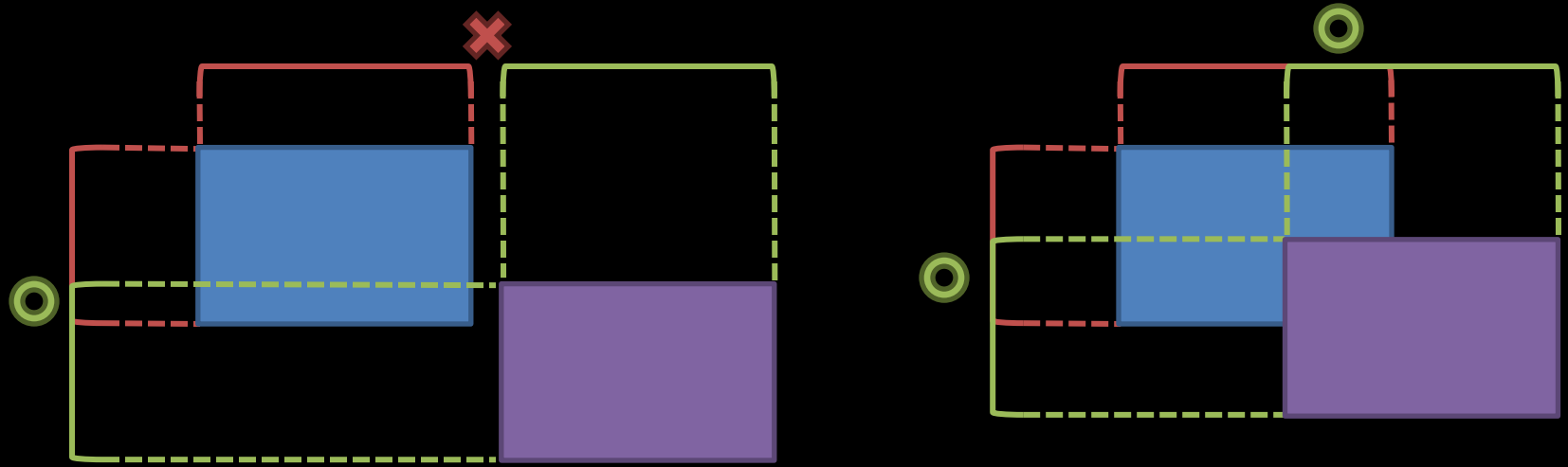

Class 3
Sphere/AAB/Cylinder Collisions

New Types of Collisions

- Box-box
- Cylinder-box
  - Combination of line-line and circle-box in 2D
- Sphere-box
  - Generalization of circle-box to 3D

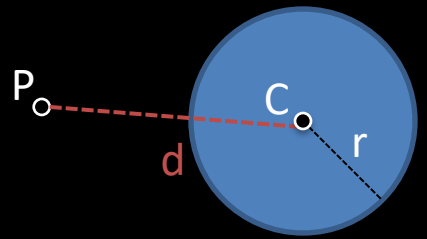
AAB-AAB

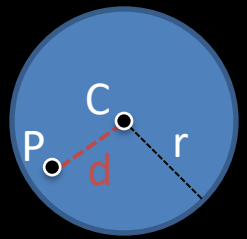
- Ensure overlap on each axis:
  - $A_{xmin} \leq B_{xmax}$ AND $A_{xmax} \geq B_{xmin}$
  - $A_{ymin} \leq B_{ymax}$ AND $A_{ymax} \geq B_{ymin}$



Point-Circle

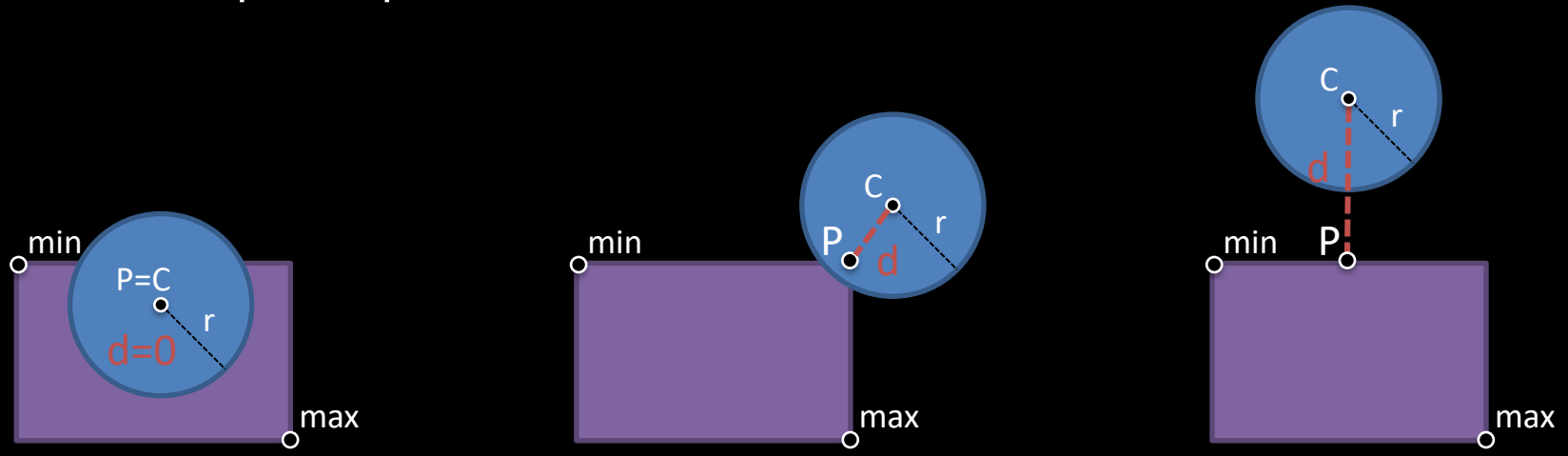
- Check if the distance between the point and the center is less than or equal to the radius
- $\|P - C\|^2 \leq r^2$





Circle-AAB

- Check if closest point to circle on AAB is in circle
  - Closest point: clamp C.x, C.y to [min.x, max.x], [min.y, max.y]
  - Then just do point-circle collision



Sphere-AAB

- Easy to generalize to 3D from last 2 slides

Class 3

Ellipsoid/Triangle Collisions

Triangle Collisions

MOTIVATION

Current system is nice…

- The current collisions we have looked at only work in an axis-aligned world
- This is suitable for lots of games, such as Minecraft

…but not always great

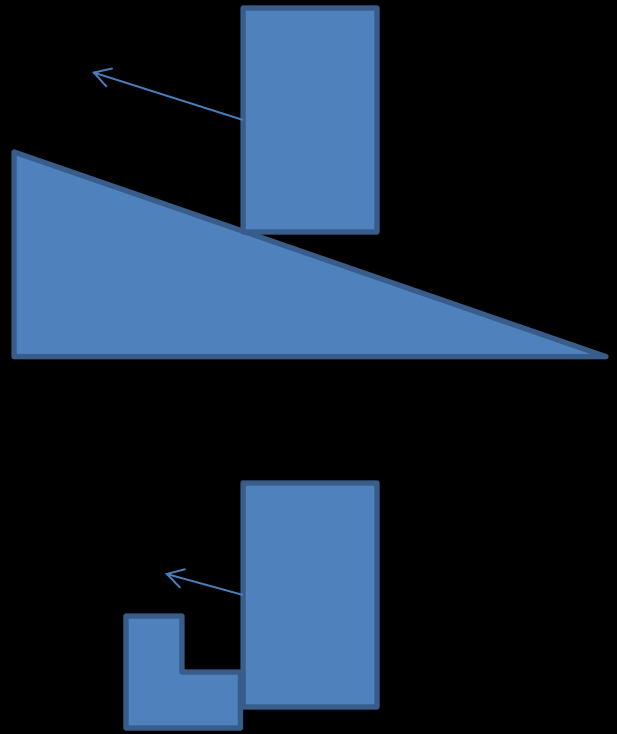
- What if I want:
  - Slopes/ramps/curved surfaces
  - Non 90 degree angles
  - Environment objects of varying size

What do we really want?

- Arbitrary environment representation
  - Not restricted to a grid or size
- Arbitrary shapes in that environment
  - Allow for sloped surfaces
  - Allow for approximated curved surfaces
- We want TRIANGLES!
- What shape should entities be?

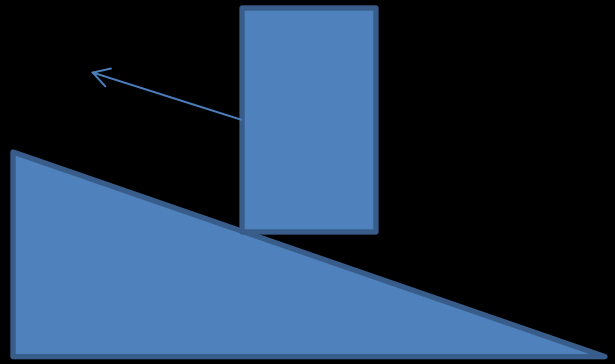
Shape: AABB

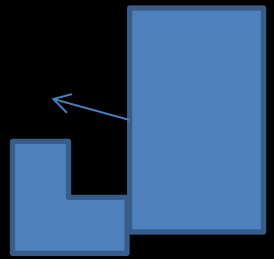
- Pros:
  - Simple collision test for axis-aligned worlds
- Cons:
  - Entities don't have same diameter in all directions
  - Complicated collision test for arbitrary worlds
  - Entities "hover" on slopes
  - Stairs need special handling



Shape: Cylinder

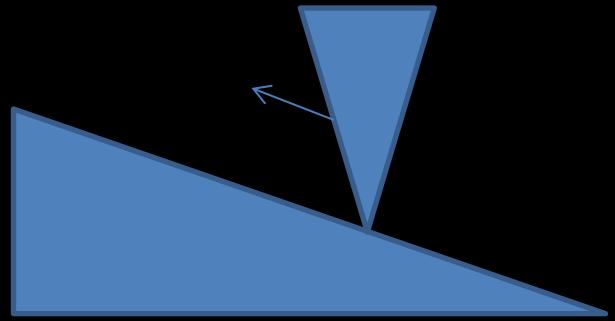
- Pros:
  - Entities have same diameter in all directions
- Cons:
  - Collisions even more complicated by caps
  - Same slope hover problem
  - Same stairs problem





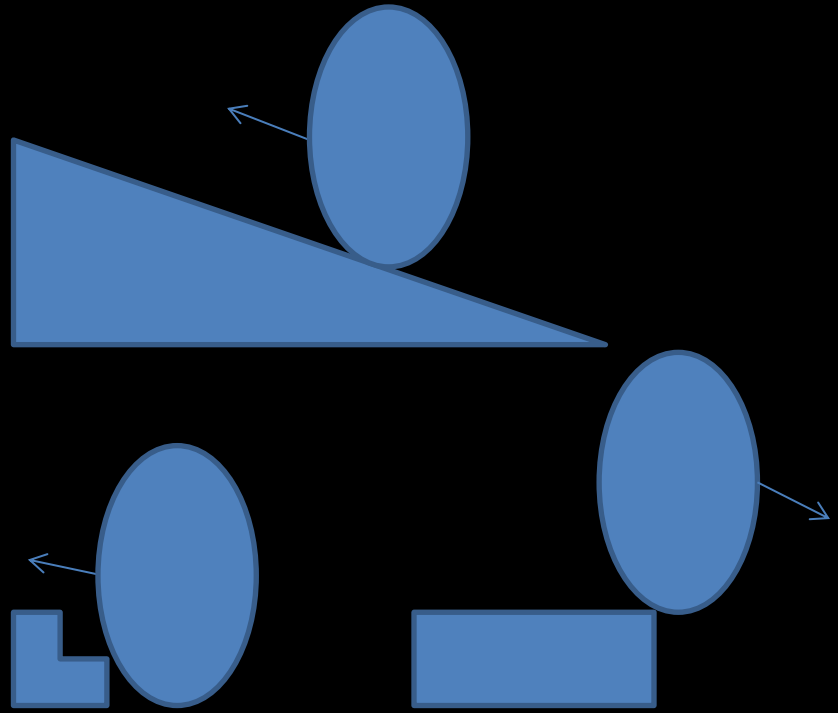
Shape: Upside-down cone

- Pros:
  - Entities don't hover on slopes
  - Entities naturally climb stairs (kinda)
- Cons:
  - Still more complicated collision tests
  - Sliding like this may be undesirable



Shape: Ellipsoid

- Pros:
  - Simpler collisions than any of the others for arbitrary triangle world
  - Entities closer to the ground on slopes
  - Entities still climb stairs (if they're low enough)
- Cons:
  - Entities "dip" down a bit going off edges



With this system…

- Environment represented as an arbitrary mesh of triangles
- Entities represented as ellipsoids
- We need to build:
  - A basic mesh representation
  - Ellipsoid-triangle collisions
  - Ellipsoid raycasting
  - Triangle raycasting
  - Navigation through the world

Non-Voxel Collisions

QUESTIONS?

Ellipsoid/Triangle Collisions

ELLIPSOID RAYCASTING

Raycasting a circle

- Before we try 3D, let's think in 2D
- Ray: position and direction
  - $\vec{r}(t) = \vec{p} + t\vec{d}$
  - $\vec{d}$ is a normalized vector
- Make every circle a unit circle at the origin (simpler to raycast)
  - Translate circle center and ray origin by -(circle center)
  - Scale circle and ray origin and direction relative to radius $(1/r)$
    - DO NOT RE-NORMALIZE the ray direction vector
- Plug ray equation into equation for unit circle at the origin:

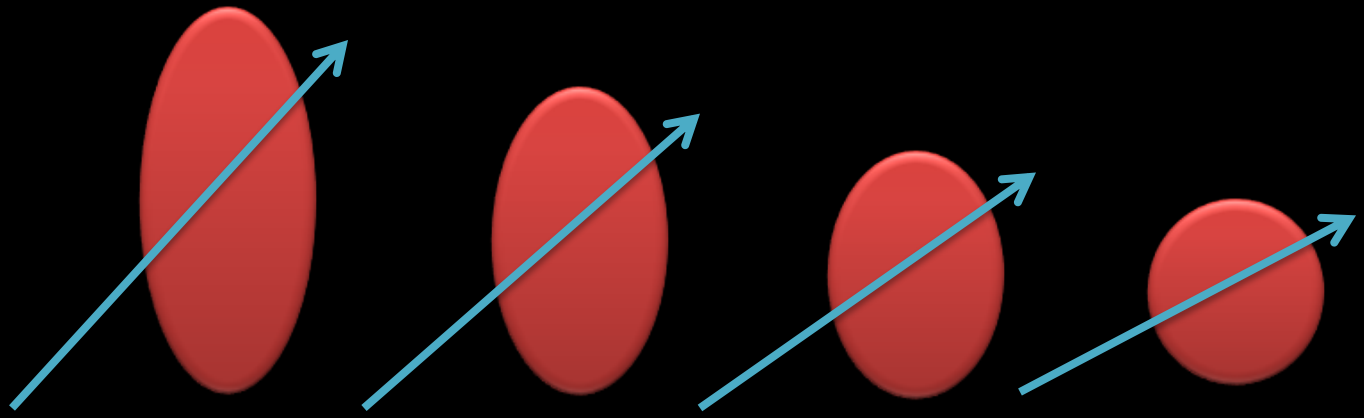
$$x^2 + y^2 = (\vec{p}.x + \vec{d}.x * t)^2 + (\vec{p}.y + \vec{d}.y * t)^2 = 1$$

- t is the only real variable left, solve with quadratic formula
  - t gives you the intersection point for both the unit circle with the transformed ray, and the original circle with the untransformed ray
    - Because we haven't re-normalized the direction

Raycasting a Sphere

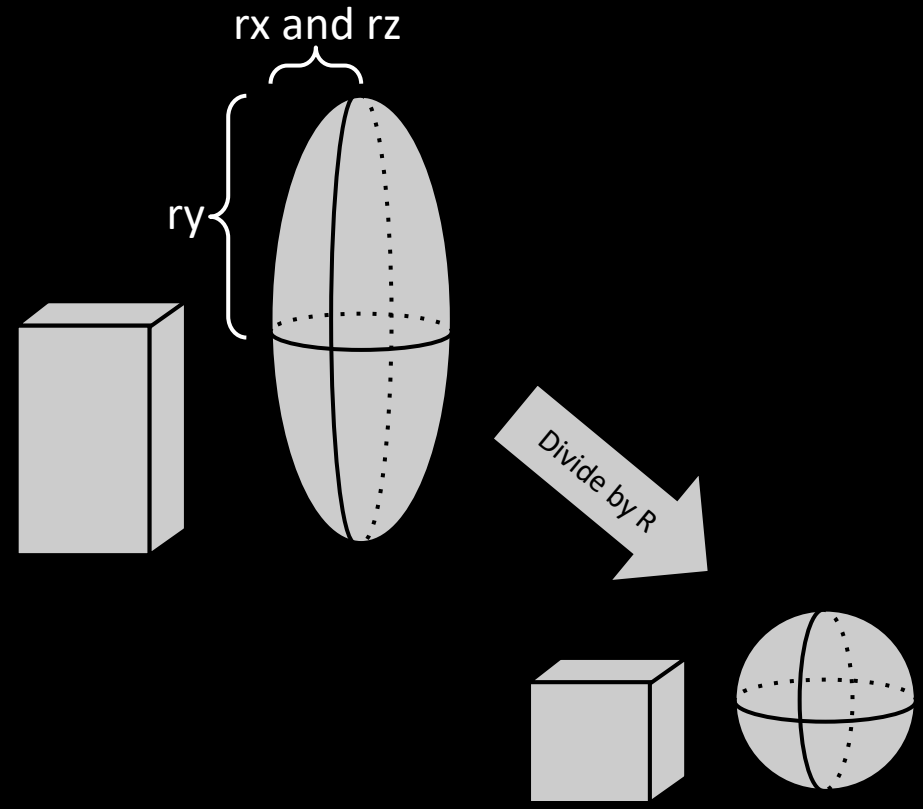
- Unit sphere at the origin: $x^2 + y^2 + z^2 = 1$
  - Same transformations to both sphere and ray
- Same ray equation (3 components)
- Solve for t:
  - Calculate discriminant $(b^2 - 4ac)$
    - < 0 means no collision (no real roots to quadratic)
    - $= 0$ means one collision (one root, ray is tangent to sphere)
    - > 0 means two collisions (two roots)
- Plug t into ray equation to get 3D intersection

Raycasting an Ellipsoid



Change of space

- Sphere intersections are way easier than ellipsoid intersections
- Squish the entire world so the ellipsoid is a unit sphere!
  - Do detection in that space, convert back
- Change of vector spaces:
  - Ellipsoid radius R = (rx, ry, rz)
  - Use basis (rx,0,0), (0,ry,0), (0,0,rz)
  - Ellipsoid space to sphere space: component-wise division by R!



Raycasting an Ellipsoid

- Convert from ellipsoid space to unit sphere space
  - Don't forget to transform to origin as well as scale
- Solve sphere equation for the new ray
- Plug t into the original ray equation to get intersection point

Raycasting II – Ellipsoid Raycasting

QUESTIONS?

Ellipsoid/Triangle Collisions

TRIANGLE RAYCASTING

Raycasting to the environment

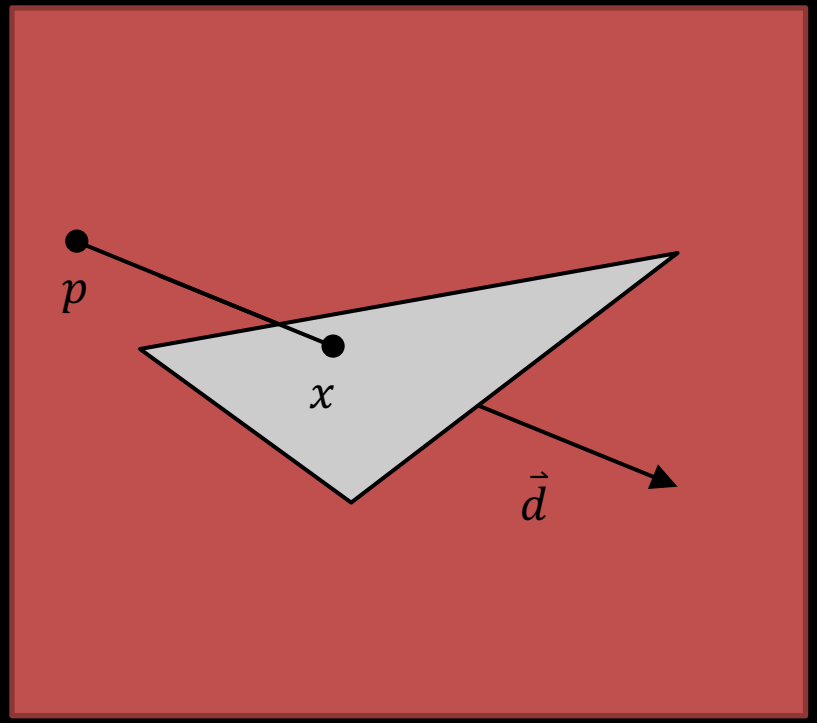
- We can raycast to ellipsoids, great
- Need some way to be able to raycast to our environment as well
- This can be used for gameplay like bullets, lasers, line of sight, etc…
- More importantly, you will use this in your sphere-triangle collision detection

Raycasting to the environment

- Our environment is made up entirely of polygons
- All polygons can be decomposed into triangles
  - Even ellipsoids are approximated by triangles when being drawn
- So to raycast the environment, raycast to each triangle, and take the closest intersection

Ray-triangle intersection

- Given: Ray casted from $\vec{p}$ in the direction of $\vec{d}$
  - Ray equation $\vec{r}(t) = \vec{p} + t\vec{d}$
- Goal: find $\vec{x}$, the point on the triangle
- There might not be a point $\vec{x}$ which exists in that triangle
- But there is a point $\vec{x}$ that exists in the plane of that triangle (unless the ray is parallel to the plane)
  - t value might just be negative (the point is in the opposite direction of the ray)

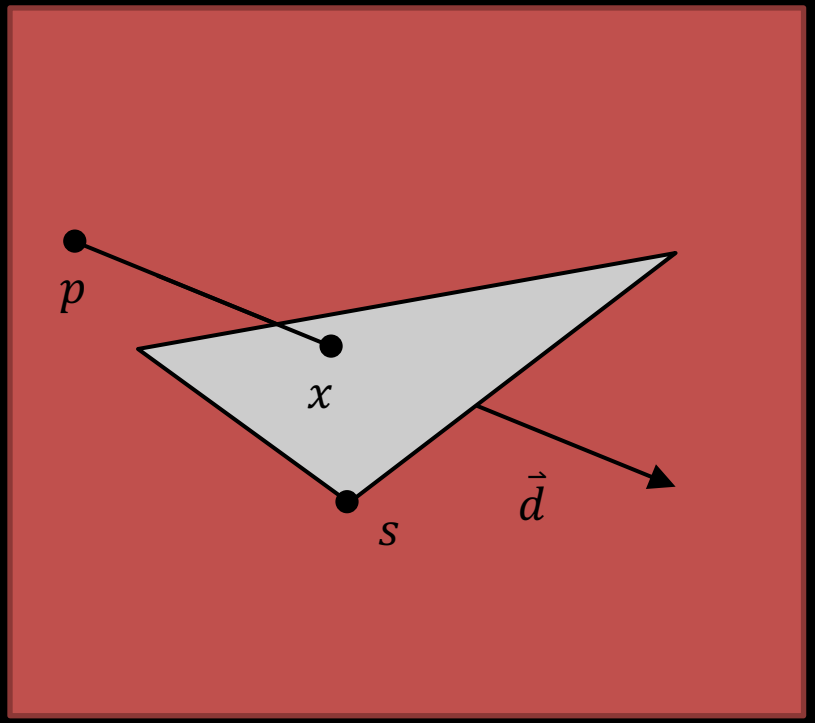


Ray-triangle intersection

- Point x on triangle plane if $(x - s) \cdot n = 0$
  - Where s is any point on the plane, such as one of the vertices
  - n is the normal of the plane
- And $x = p + t*d$
- Solve for t

  $(p + t*d - s) \cdot n = 0$

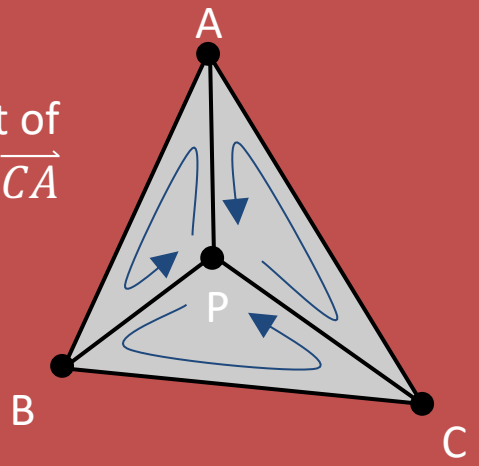
  $t = ((s - p) \cdot n) \,/\, (d \cdot n)$

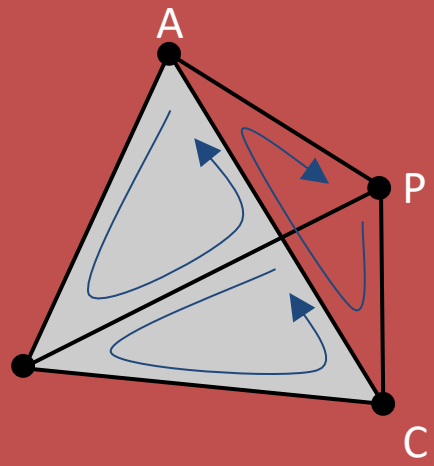


Ray-triangle intersection

- So now we know the point P at which the ray intersects the plane of the triangle
  - But is that point inside the triangle or outside of it?
- Point P (on plane) is inside triangle ABC iff P is on the left of all of the edges (assuming that edges are defined in counter-clockwise order i.e. $\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CA}$)

P is to the left of $\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CA}$





P is to the right of $\overrightarrow{CA}$

Ray-triangle intersection

- A point P is to the left of edge AB if the cross product $AB \times AP$ is in the same direction as the triangle normal
  - $BC \times BP$, and $CA \times CP$ are the other cross products
- Can calculate normal of a triangle with cross product of two of its edges

$$N = (B - A) \times (C - A)$$

- Now you can compare to see if two vectors are in the same direction by seeing if their dot product is positive

$$(AB \times AP) \cdot N > 0$$

Ellipsoid/Triangle Collisions

QUESTIONS?

ELLIPSOID TRIANGLE COLLISIONS

The basics

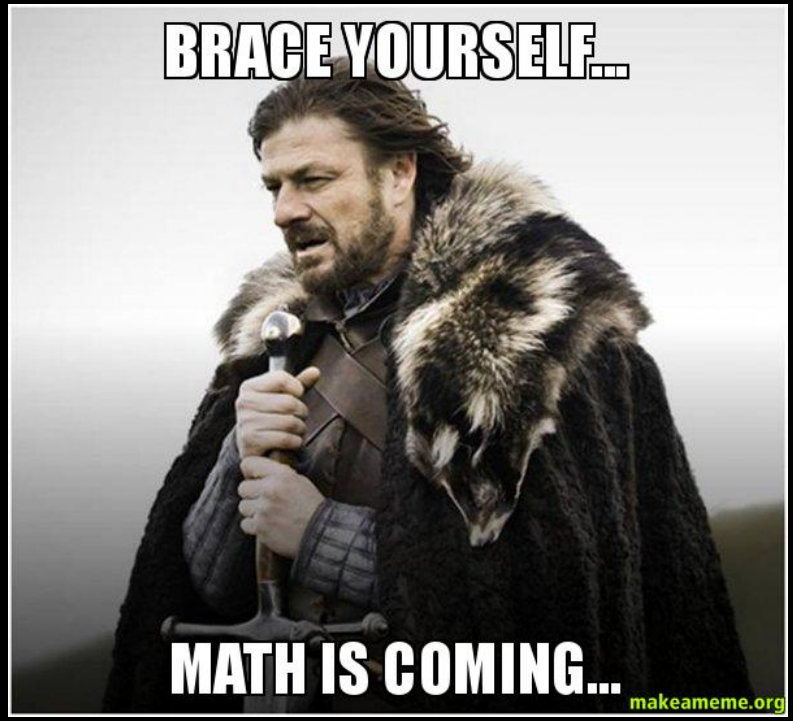
- Entity represented by an ellipsoid
- World represented by a set of triangles
- Continuous collision detection
  - Analytically compute the time of and point contact, translate object to that point
- Basic idea: formulate motion of the entity as a parametric equation, solve for intersection
  - Only works for simple motion (straight lines)

General algorithm

- Compute the line the player follows in one update
  - Kinda like raycasting start position to end position
- Do ellipsoid-triangle sweep test for all triangles and take the closest result
  - Can optimize this using spatial acceleration data structure to test relevant triangles
  - Closest indicated by smallest positive t value (proportion of update taken resulting in collision)
- Compute remaining translation, sweep again
  - Cut off after a certain number of translations
  - You'll do this next week

WARNING

- There is A LOT of vector math we're about to get into
- You DO NOT need to understand all of it
  - Though it may help with debugging
- This is not a math class
  - Don't memorize the derivations
  - Don't re-invent the wheel



Collisions III

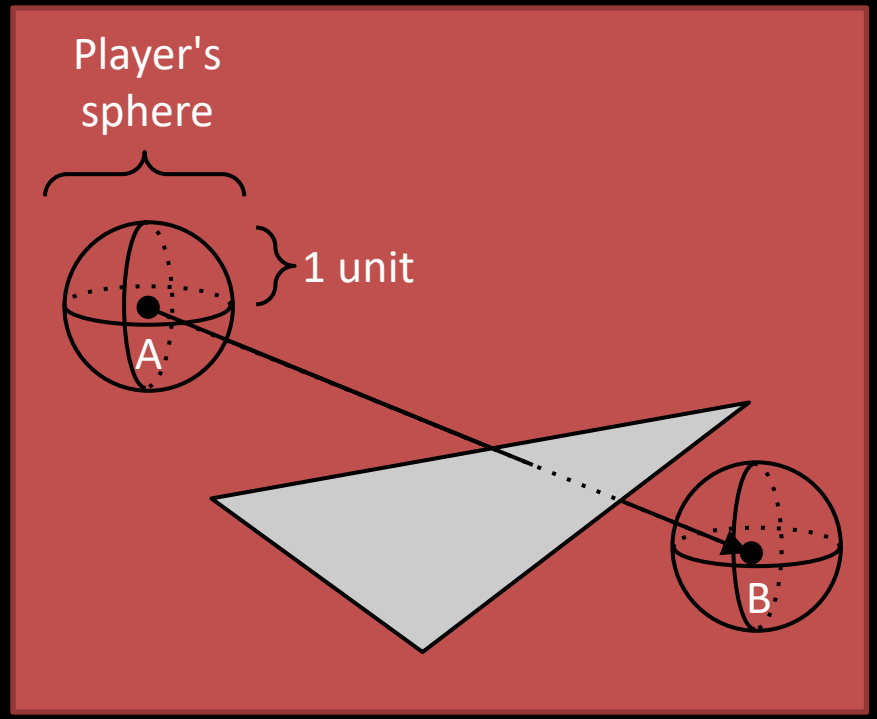
ELLIPSOID-TRIANGLE COLLISIONS

Ellipsoid-triangle collisions

- Analytic equation for a moving sphere:
  - Unit sphere moving from A at $t = 0$ to B at $t = 1$
  - Location of center: $A + (B - A)t$
  - Point P on the sphere at t if $\|[A + (B - A)t] - P\|^2 = 1$
- Solve for t in unit sphere space
  - Value stays the same in ellipsoid space!
- Split collision detection into three cases:
  - Triangle interior (plane)
  - Triangle edge (line segment)
  - Triangle vertex (point)

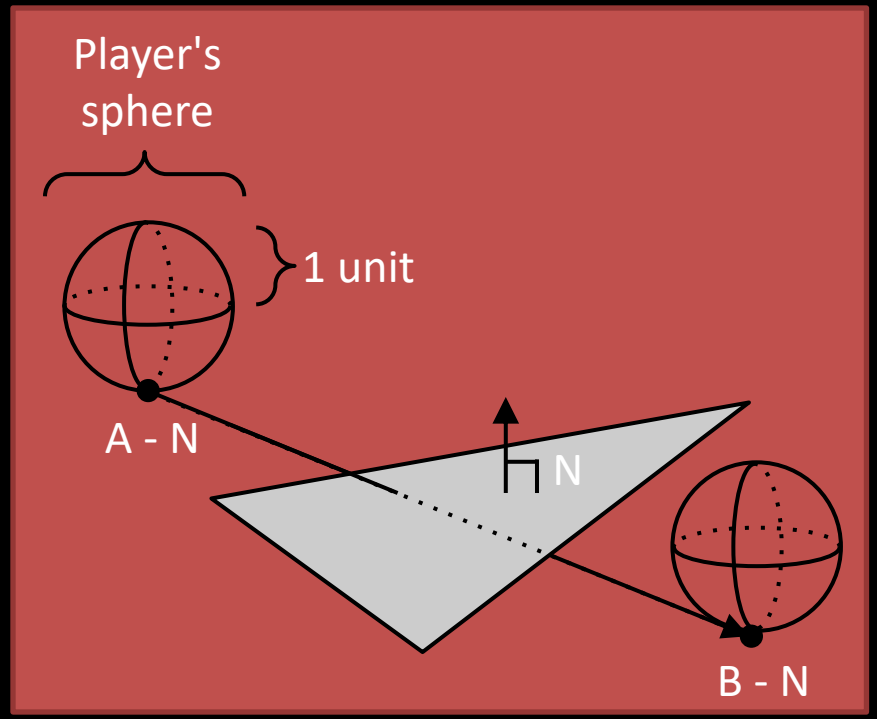
Sphere-interior collision

- Intersect moving sphere with a plane
- If intersection is inside triangle, stop collision test
  - Interior collision always closer than edge or vertex
- If intersection is outside triangle, continue test on edge and vertices
  - NO short circuit



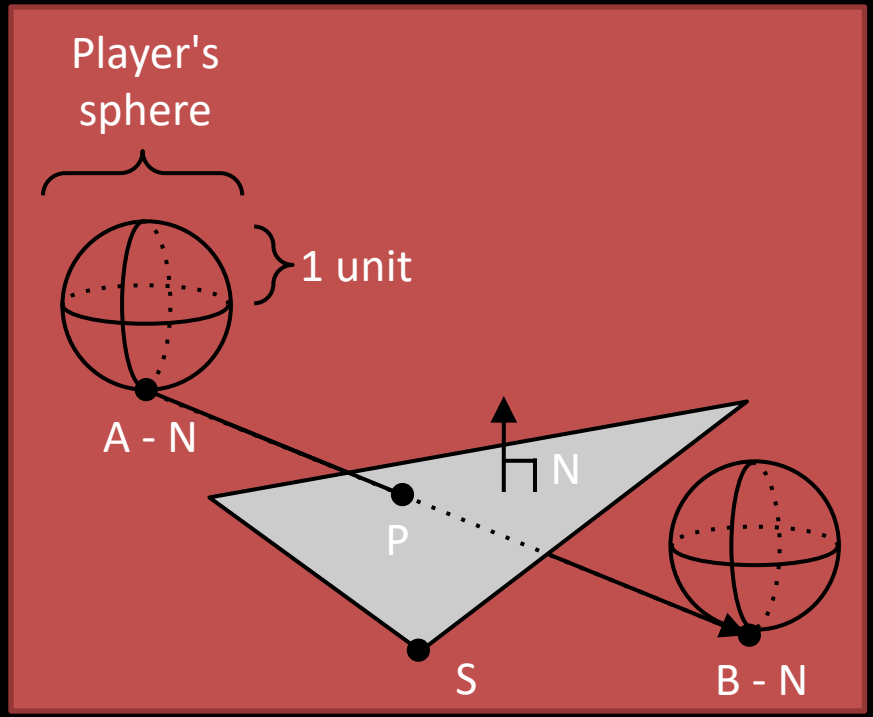
Sphere-interior collision

- Sphere-plane intersection:
  - Same thing as ray plane using the point on the sphere closest to the plane!
  - Given plane with normal N, closest point is $A - N$
    - We assume that the sphere starts "above" the triangle
    - Don't care about colliding a sphere starting below the triangle, this should never happen



Sphere-interior collision

- Point P on plane if $(P - S) \cdot n = 0$
  - Where S is any point on the plane, such as one of the vertices
- Turn the problem into ray/triangle!
- Again, when the sphere hits the plane
  - May not be in the triangle!
  - Repeat your point-in-triangle test!



Ellipsoid/Triangle Collisions – Ellipsoid-Interior

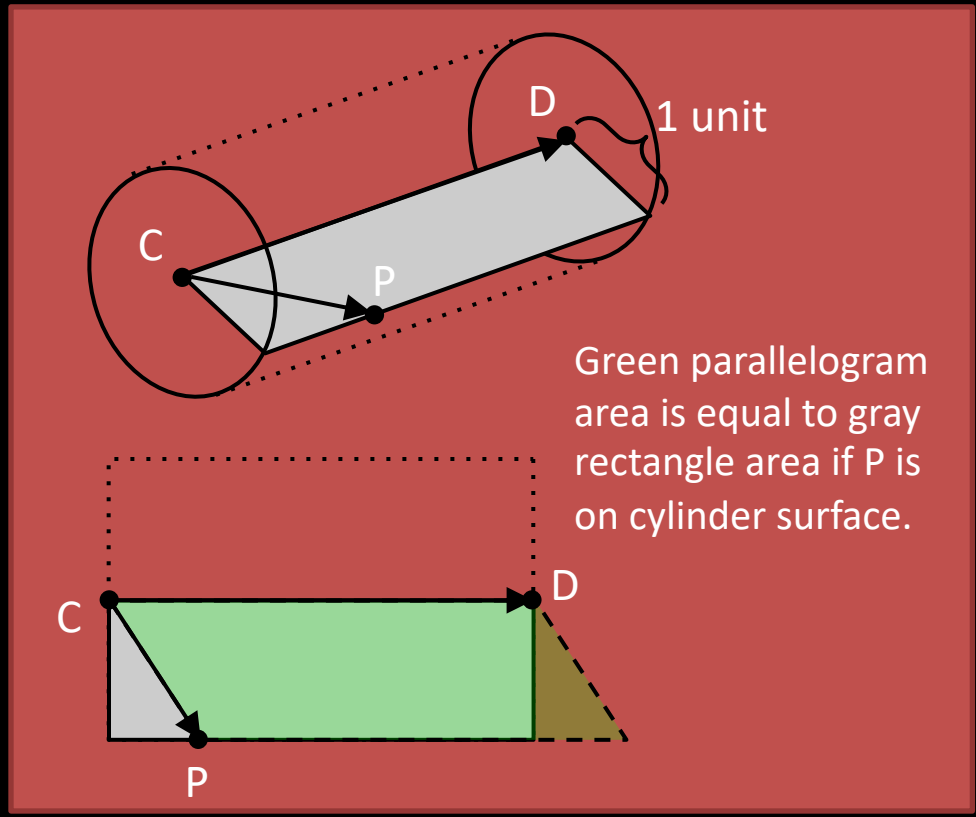
QUESTIONS?

Sphere-edge collision

- Sphere vs. edge is the same as sphere vs. line segment
  - Intersect moving sphere with the infinite line containing the edge
  - Reject intersection if it occurs outside the line segment
- How do we collide a moving sphere with a line?
  - Really just finding when sphere center passes within 1 unit of line
  - If we treat the line as an infinite cylinder with radius 1, and the motion of sphere center as ray we can use ray-cylinder intersection

Analytic sphere-edge collision

- Area of parallelogram formed by two vectors is the length of their cross product
- Defining the surface of an infinite cylinder with vectors
  - Given two points C and D along cylinder axis
  - Point P on surface if
    - Area of gray parallelogram = Area of green parallelogram
    - This means that their height is equal, which means that the distance of p to the line segment is 1



Analytic sphere-edge collision

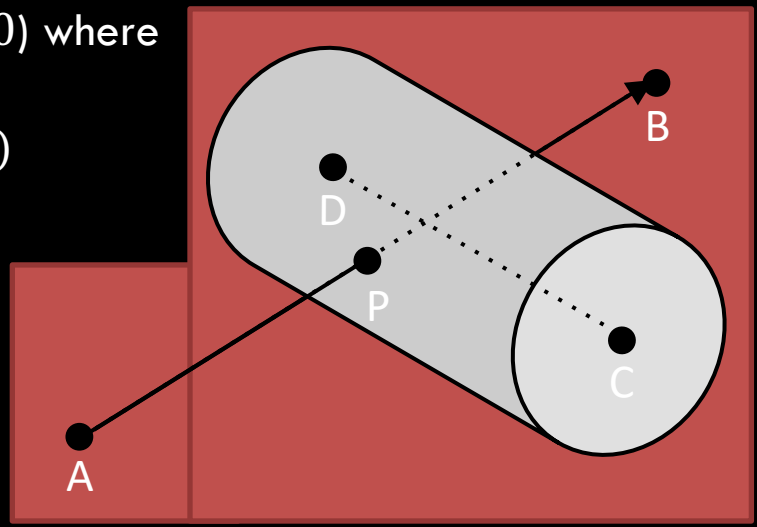
- Set $P = A + (B - A)t$
- Since the area of a parallelogram formed by two vectors is the length of their cross product, we know we know that the point P satisfies $\|(P - C) \times (D - C)\|^2 = \|D - C\|^2$. So, we have

  $\|([A + (B - A)t] - C) \times (D - C)\|^2 = \|D - C\|^2$
- Solving for t, you get quadratic ($at^2 + bt + c = 0$) where

  $a = \|(B - A) \times (D - C)\|^2$

  $b = 2((B - A) \times (D - C)) \bullet ((A - C) \times (D - C))$

  $c = \|(A - C) \times (D - C)\|^2 - \|D - C\|^2$
- Solve using quadratic equation, use lesser t value



Analytic sphere-edge collision

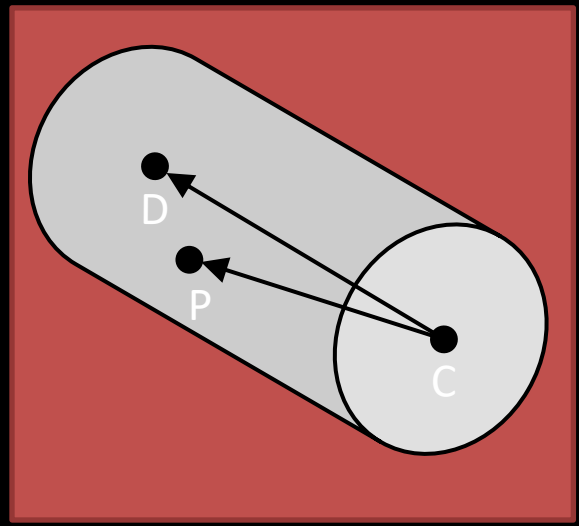
- Discard intersection if not between C and D
  - Will be handled by vertex collision test
- To check if intersection is between C and D:
  - Get vector from C to intersection point P

    $P - C$
  - Project this vector onto cylinder axis

    $(P - C) \bullet \frac{D-C}{\|D-C\|}$
  - Check if projection is in the range $(0, \|D - C\|)$

    $0 < (P - C) \bullet \frac{D-C}{\|D-C\|} < \|D - C\|$
  - Optimized by multiplying by $\|D - C\|$:

    $0 < (P - C) \bullet (D - C) < \|D - C\|^2$

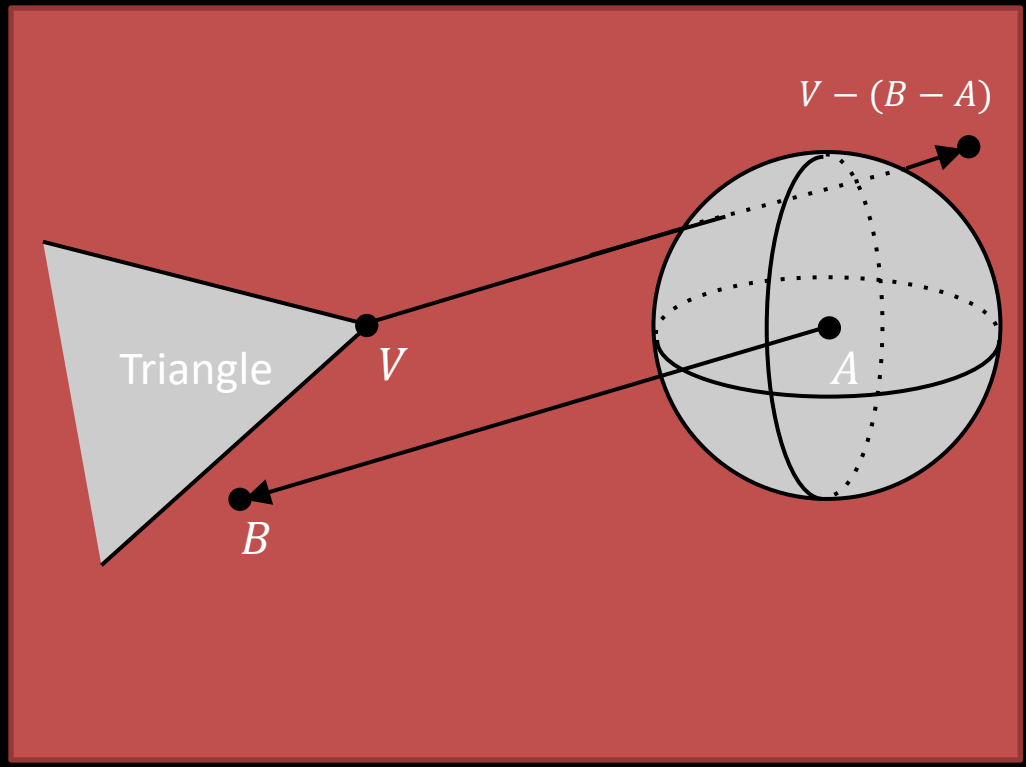


Ellipsoid/Triangle Collisions – Ellipsoid-Edge

QUESTIONS?

Analytic sphere-vertex collision

- Collision test against a triangle vertex V
- How do we collide a moving sphere against a point?
  - We know how to do a ray-sphere intersection test
  - Moving sphere vs. point is equivalent to sphere vs. moving point
    - Where the point moving in opposite direction



Ellipsoid/Triangle Collisions – Ellipsoid-Vertex

QUESTIONS?

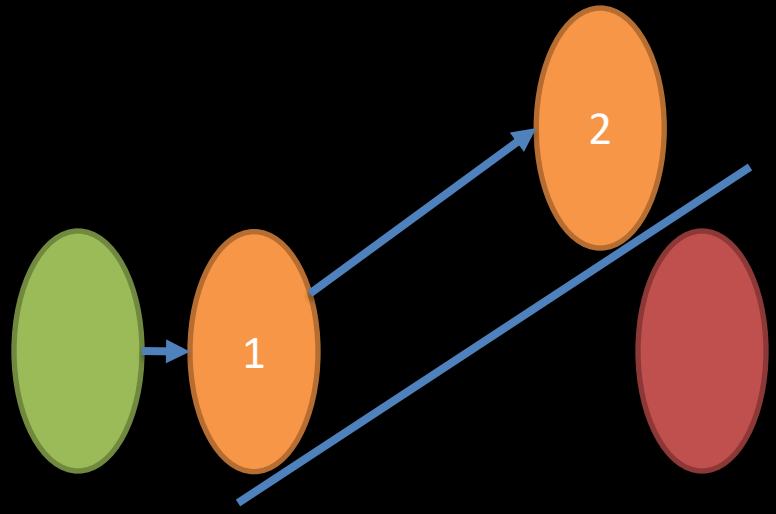

Class 3
Walking Around a Mesh

A few more details…

- We have figured out how to detect ellipsoid triangle collisions, but we have not discussed how to respond to them in a helpful way
- If we tried to walk around a triangle mesh without considering some extra details, we would stick to the ground!
- Why? Notice that we haven't discussed minimum translation vectors!

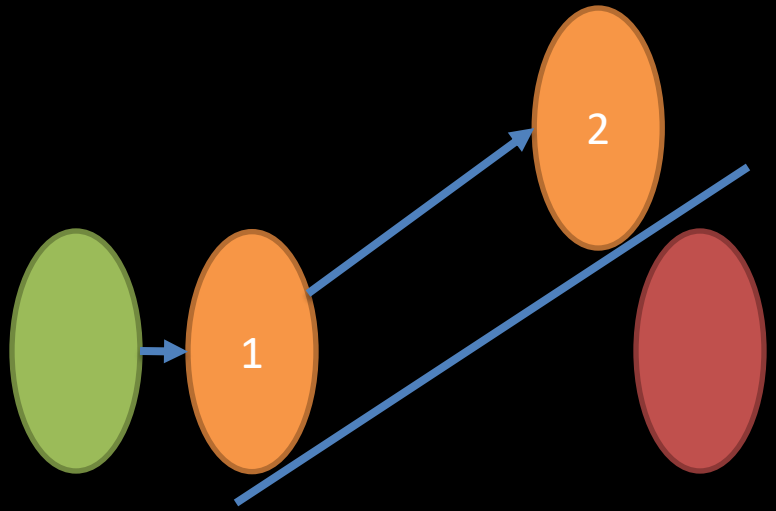
The “MTV Slide”

- The green ellipsoid is the starting position of the ellipsoid during the tick
- The red ellipsoid is the ending position of the ellipsoid during the tick, **assuming no collisions happened**
- The first orange ellipsoid is the position of the ellipsoid at the first collision (with the blue triangle) that happens as the green ellipsoid moves in the direction of the red ellipsoid
- The second orange ellipsoid is the same as the first orange ellipsoid except it has been slid forward in the direction of the red ellipsoid in such a way that it does not collide with the blue triangle



The "MTV Slide"

- We will call this behavior the "MTV slide"
  - Adding this functionality prevents us from sticking to the ground!
- The next slide has pseudocode explaining how to implement this
  - Essentially, you take the vector connecting the first orange ellipsoid to the red ellipsoid and subtract away the component of that vector that is not orthogonal to the triangle's normal
  - Then you add the resulting vector to the position of the first orange ellipsoid to get the second orange ellipsoid
- Note that the second ellipsoid could run into **another triangle**
  - So, we need to repeat our collision detection and MTV slide steps for some maximum number of translations, or until we come across an iteration where we can safely slide without hitting another triangle



MTV Slide Pseudocode

```
ellipsoidTriangleCollisions(initial_pos, final_pos):
  collisions = []
  curr_pos = initial_pos
  for i in range(MAX_TRANSLATIONS):
    c = getClosestCollision(initial_pos, next_pos)
    if (c.t == INFINITY):
      return (collisions, next_pos)
    else:
      curr_pos = c.pos
      d = final_pos – curr_pos
      // you might need to play around with d_corrected in order to correctly handle
      // downward-facing walls
      d_corrected = d - dot(d, c.normal) * closest_collision.normal
      next_pos = curr_pos + d_corrected
      collisions.append(c)
  return (collisions, curr_pos)
```

Improving Movement

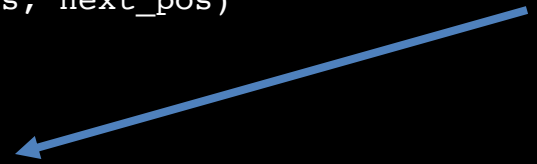
- Even with the MTV slide, we may stick to a wall if we are touching a wall and any component of our horizontal velocity is in the direction of the wall
  - In other words, if we “hug” a wall and try to move parallel to it, we will stick to the wall
  - This occurs because we repeatedly collide with the wall
  - We will also get weird behavior in the collision debugger
- We can try to fix this problem by nudging the first orange ellipsoid slightly away from the triangle at every iteration…

MTV Slide Pseudocode

```
ellipsoidTriangleCollisions(initial_pos, final_pos):
  collisions = []
  curr_pos = initial_pos
  for i in range(MAX_TRANSLATIONS):
    c = getClosestCollision(initial_pos, next_pos)
    if (c.t == INFINITY):
      return (collisions, next_pos)
    else:
      curr_pos = c.pos + c.normal * .01 // nudge the ellipsoid slightly away from the triangle
      d = final_pos – curr_pos
      // you might need to play around with d_corrected in order to correctly handle
      // downward-facing walls
      d_corrected = d - dot(d, c.normal) * closest_collision.normal
      next_pos = curr_pos + d_corrected
      collisions.append(c)
  return (collisions, curr_pos)
```

MTV Slide Pseudocode

```
ellipsoidTriangleCollisions(initial_pos, final_pos):
  collisions = []
  curr_pos = initial_pos
  for i in range(MAX_TRANSLATIONS):
    c = getClosestCollision(initial_pos, next_pos)
    if (c.t == INFINITY):
      return (collisions, next_pos)
    else:
      curr_pos = c.pos + c.normal * .01 // nudge the ellipsoid slightly away from the triangle
      d = final_pos – curr_pos
      // you might need to play around with d_corrected in order to correctly handle
      // downward-facing walls
      d_corrected = d - dot(d, c.normal) * closest_collision.normal
      next_pos = curr_pos + d_corrected
      collisions.append(c)
  return (collisions, curr_pos)
```



Improving Movement

- Unfortunately, this nudging method can push us **through** a nearby triangle and make us fall into the floor!
- An expensive way to deal with this is to nudge the ellipsoid and then check whether we collided with another triangle by nudging
  - This is an expensive fix because we could end up nudging the ellipsoid back and forth repeatedly as the ellipsoid escapes a crevice
  - But it works!

Final MTV Slide Pseudocode

```
ellipsoidTriangleCollisions(initial_pos, final_pos):
  collisions = []
  curr_pos = initial_pos
  for i in range(MAX_TRANSLATIONS):
    c = getClosestCollision(initial_pos, next_pos)
    if (c.t == INFINITY):
      return (collisions, next_pos)
    else:
      curr_pos = doNudge(curr_pos, c)
      d = final_pos – curr_pos
      // you might need to play around with d_corrected in order to correctly handle
      // downward-facing walls
      d_corrected = d - dot(d, c.normal) * c.normal
      next_pos = curr_pos + d_corrected
      collisions.append(c)
  return (collisions, curr_pos)
```

Final MTV Slide Pseudocode

```
doNudge(curr_pos, collision):
  nudge = collision.normal
  pos_nudged = collision.center + nudge * .01
  for i in range(MAX_NUDGES): // MAX_NUDGES can be something like 3
    nudge_collision = getClosestCollision(curr_pos, pos_nudged)
    if (nudge_collision.t == INFINITY):
      curr_pos = pos_nudged
      break
    else:
      // this code is necessary when we hit an edge shared by two triangles in exactly the wrong way
      // this code tests whether we are nudged into the same triangle twice, and nudges you in the
      // OPPOSITE direction if that happens
      if (length(nudge_collision.normal – nudge) < EPSILON || length(nudge_collision.normal + nudge) < EPSILON):
        nudge = -nudge_collision.normal
      else:
        nudge = nudge_collision.normal
      pos_nudged = nudged_collision.center = nudge * .01
  return curr_pos
```

Player Controls

- Keep track of whether you are on the ground
  - If you are on the ground, then your y velocity is 0
  - If you are not on the ground, then your y velocity is not zero and gravity is acting upon you
- You are on the ground if you collide with a triangle with an upward-facing normal (i.e. dot(n,(0,1,0) > 0)
- If no movement keys are pressed (forward, backward, left, right) and we are on the ground, then we don't need to move at all during the tick
- These methods are a bit hacky, but they get the job done

Player Controls Pseudocode

- The next slide has some pseudocode for handling player movement
- There are lots of ways of solving this problem, so you should experiment!

Player Controls Pseudocode

```
// m_pos, m_on_ground, m_y_vel are member variables
handlePlayerMovement():
  old_pos = copy(m_pos) // get a copy of the current position
  moving_laterally = false // indicates whether we pressed WASD
  key_presses = getKeyPresses()
  look = camera.getLook()
  dir = normalize(vec3(look.x,0,look.z))
  perp = vec3(dir.z,0,-dir.x)

  if (WASD_pressed(key_presses)): // if we pressed forward, backward, left, or right
    moving_laterally = true
    m_pos = translateHorizontally(key_presses, m_pos, dir, perp) // translate using dir and perp

  if jumpButtonPressed():
    m_on_ground = false
    m_y_vel = JUMP_SPEED

  y_vel += GRAVITY * seconds // GRAVITY is some negative number
  if (m_on_ground):
    if (y_vel < 0):
      y_vel = 0

  m_pos.y += y_vel // now m_pos contains the position of the player, assuming there are no collisions

  m_pose, collisions = ellipsoidTriangleCollisions(old_pos, m_pos) // see ”mtv slide” pseudocode

  m_on_ground = false // we need to assume that we are not on the ground
  for collision in collisions:
    if (dot(vec3(0,1,0), collision.normal) > 0):
      m_on_ground = true

  if (!moving_laterally && m_on_ground):
    m_pos = old_pos
```

Class 3

Tips for Ellipsoid/Triangle Collisions

Tips for Ellipsoid/Triangle Collisions

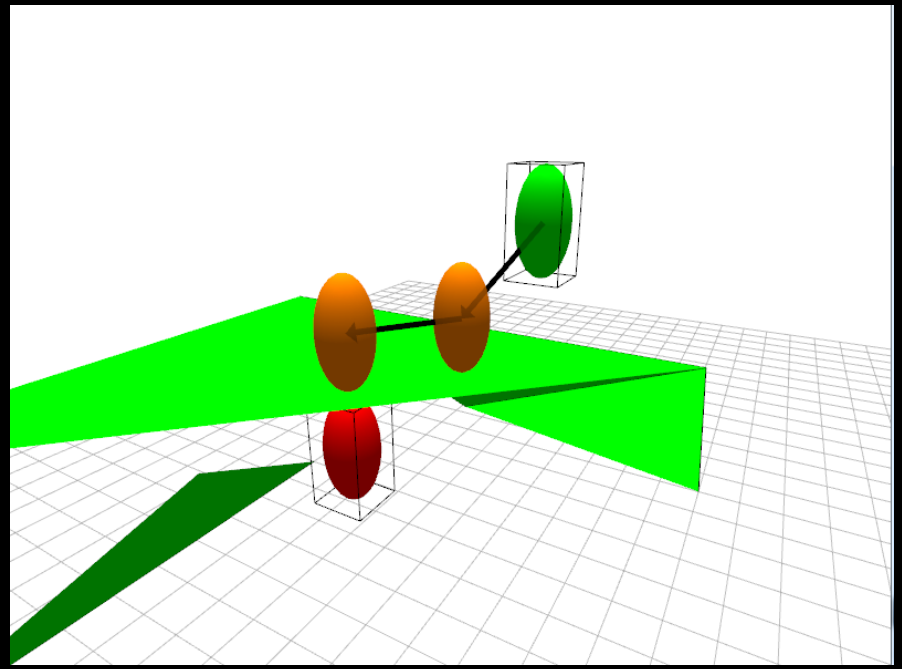
COLLISION DEBUGGER

“No, I don’t need a debugger”

- Physics/collision bugs are the hardest type of bugs to track down
- It will be much easier for you to find your mistakes in a controlled environment than for you to make them in your own code
- It’s easier to test to make sure you’ve done it correctly

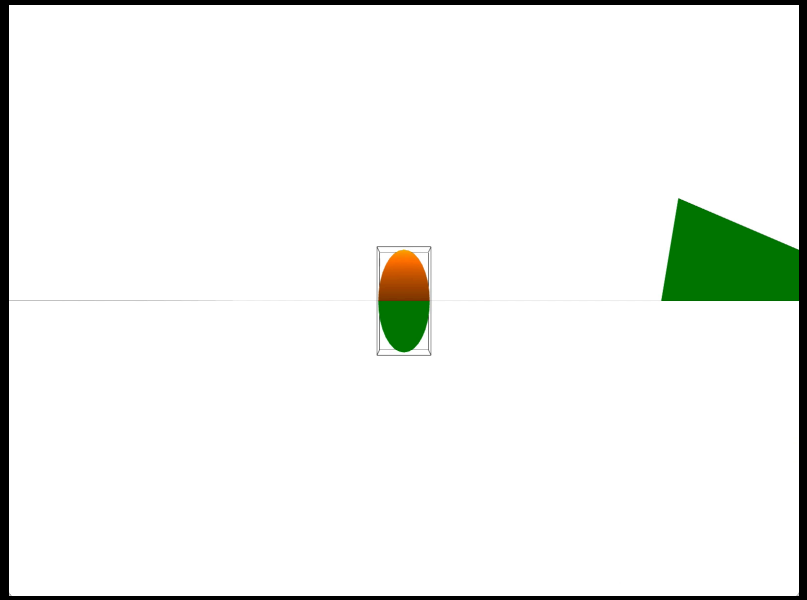
How does it work?

- You can move around two of the ellipsoids here
  - The green ellipsoid represents an entity at the beginning of the tick
  - The red ellipsoid represents an entity at the end of the tick (without collision)
- The other two ellipsoids are determined by the placement of the first two
  - The first orange ellipsoid represents where the entity will end up via colliding with the green triangles
  - The second orange ellipsoid represents where the entity slides to after hitting the surface (after the “MTV slide”)

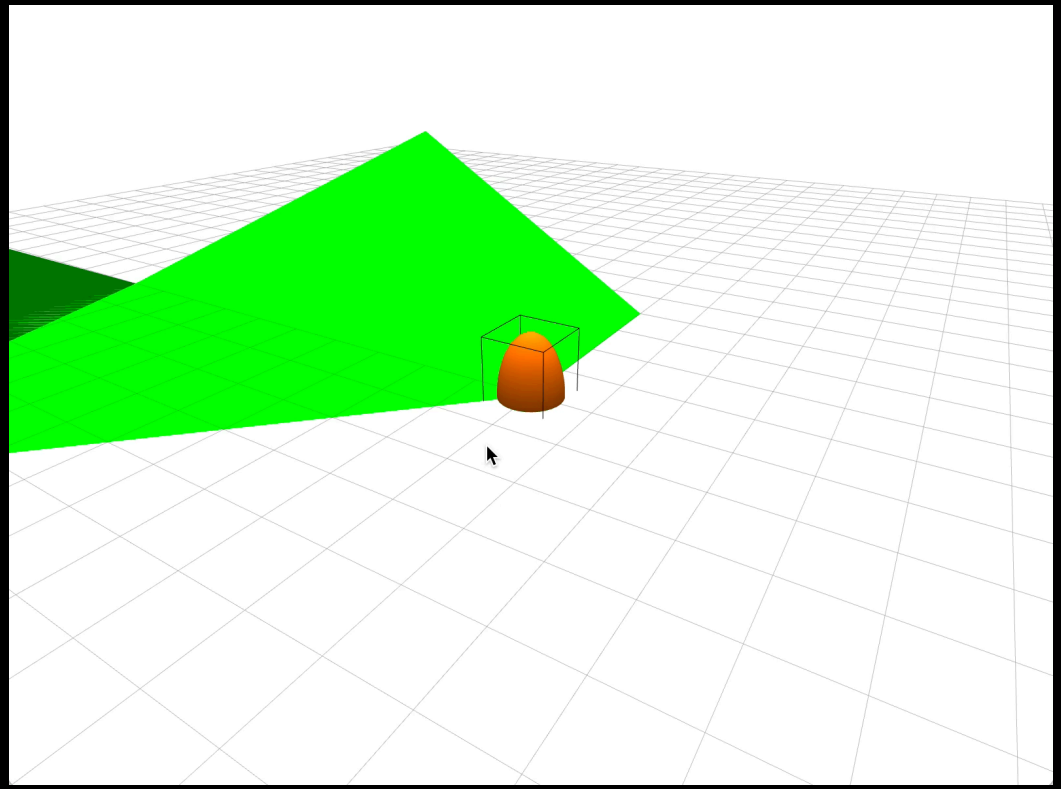


Stencil Demonstration Video

- You can move the green and red ellipsoids around by clicking and dragging the boxes around them
- Notice that dragging different sides of the box will move the ellipsoid in different ways



Solution Demonstration Video



Collision Data

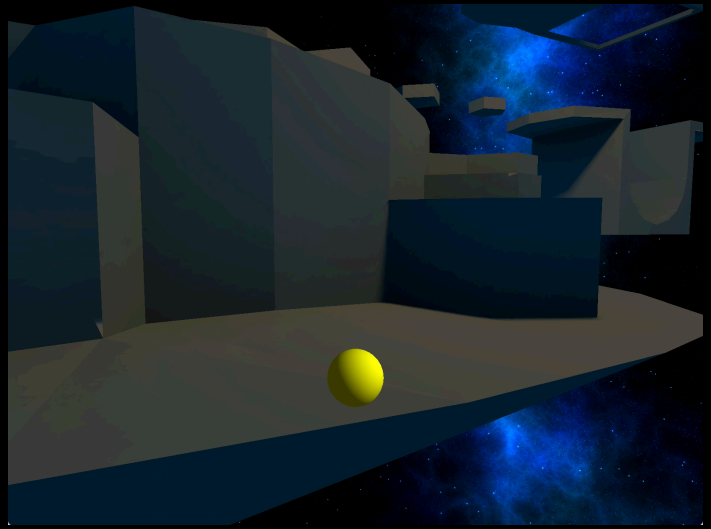
- Your collision code should return a struct, minimally containing:
  - t-value in [0,1]
  - Normal
  - Point of contact
- You may want to put “fancier” stuff in later

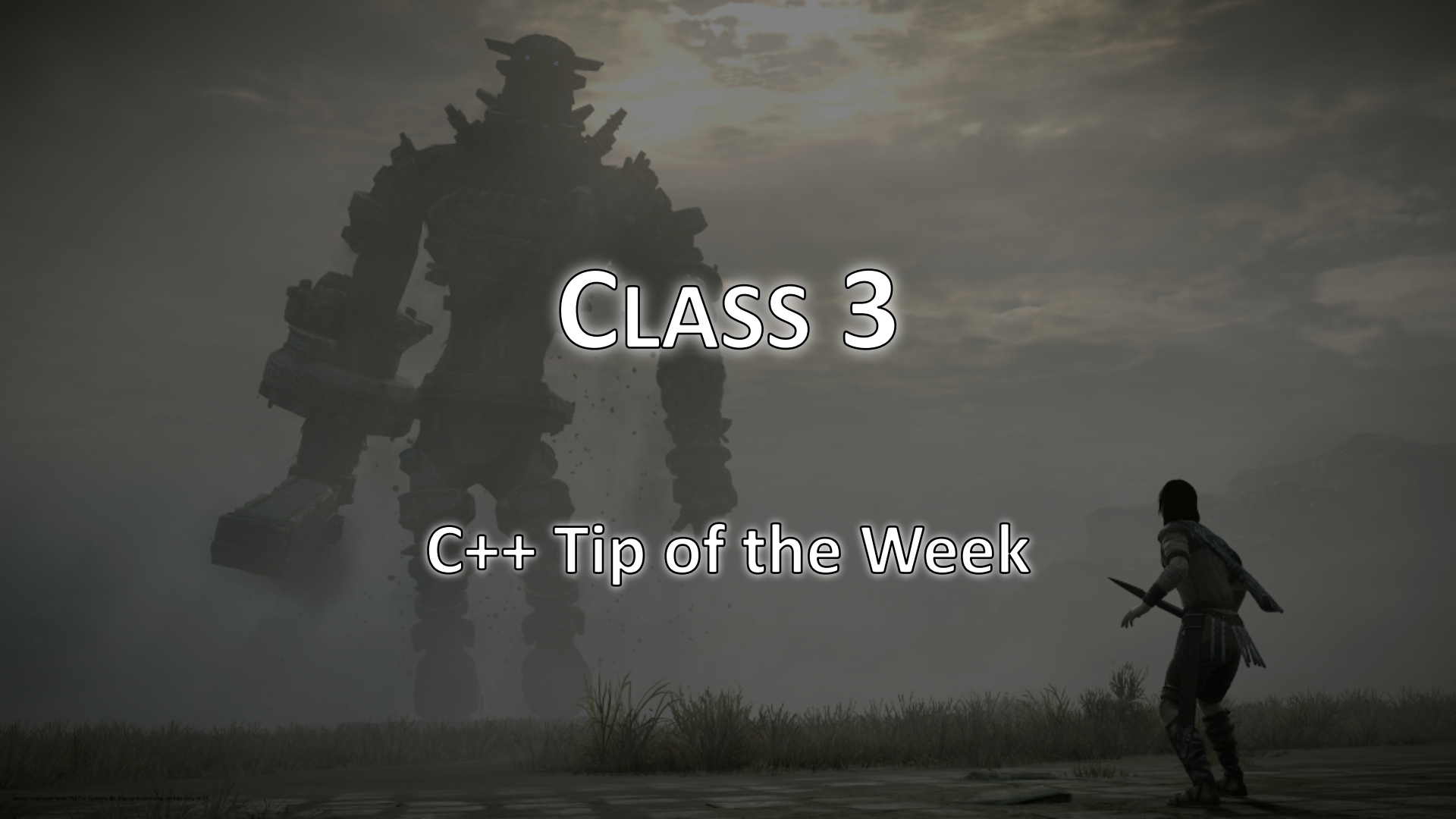
More stuff

- About two sided triangles …
- Don’t worry about colliding ellipsoids with triangles that they are already “inside”
- Don’t worry about colliding ellipsoids with triangles that they are “below”

Goal for Platformer 1

- Use the collision debugger to implement your collisions, and then use them to walk around an arbitrary mesh (see video)




Class 3
C++ Tip of the Week

C++ Tip of the Week

PARAMETRIZED INHERITANCE

Parametrized inheritance

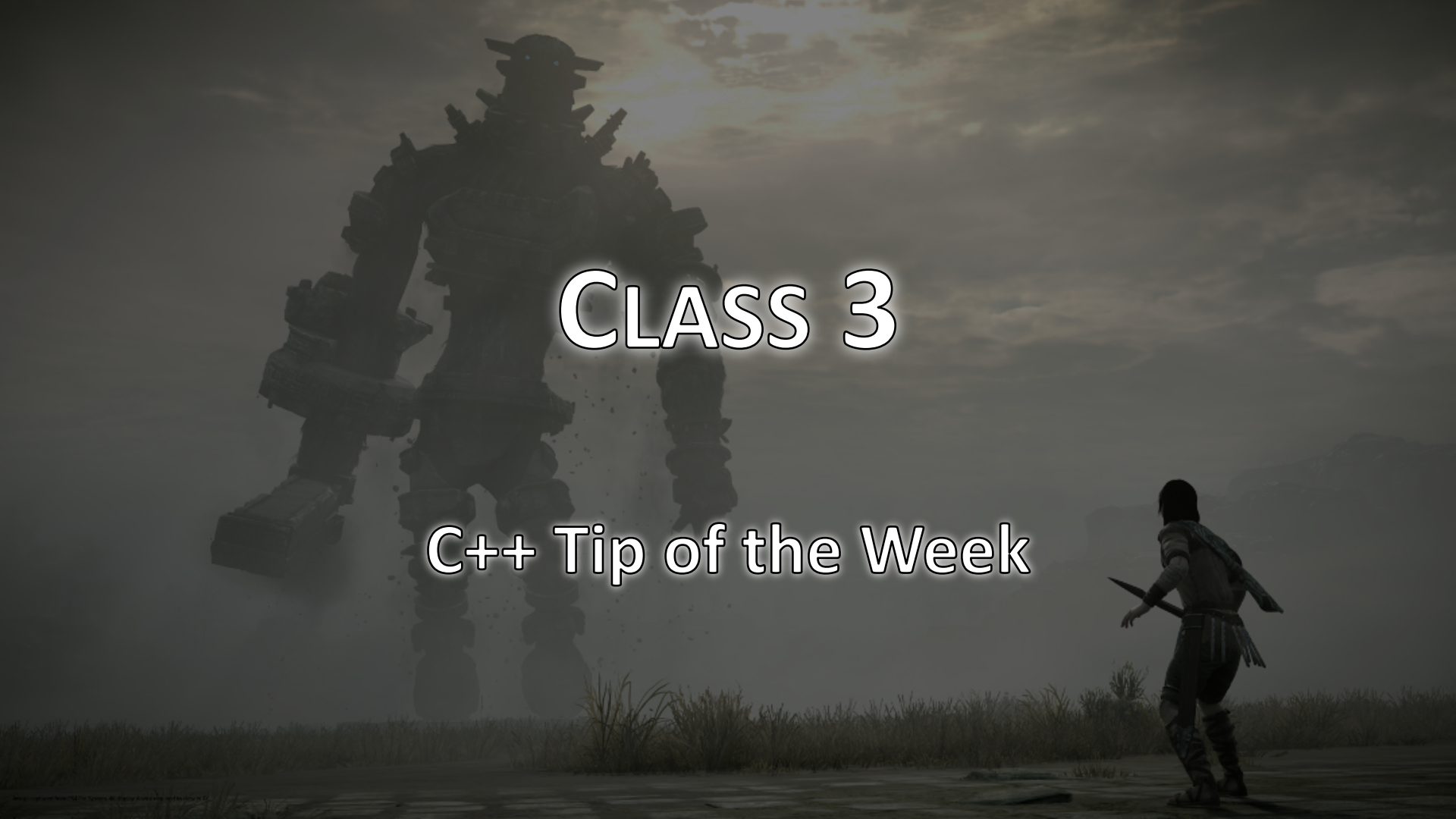
```
// (Parent varies at compile time)
template<class Parent>
class Kid : public Parent
{
public:
   Kid () : Parent() { … };
   method() {
                Parent::doThis(true);
                doThat();
                Parent::doThis(false); }
   doThat() { … };
}
```

```
// call Dad::doThis, Kid::doThat,
Dad::doThis
Kid<Dad> f1;   f1.method();

// call Mom::doThis, Kid::doThat,
Mom::doThis
Kid<Mom> f2;   f2.method();

Kid<Parent> f3; f3.doThat();
// the compiler just wrote 3 “Kid”
classes for us
```

- Kinda like generics in Java
- But the thing in the braces is just text replaced by the compiler when given actual argument


Class 3
C++ Tip of the Week

C++ Tip of the Week (Part II)

OPERATOR OVERLOADING

Wait, Operator Overloading?

- In C++, you can tell basic operators to work with classes (or enums!)
  - The basic arithmetic operations are commonly overloaded (+, -, *, /)
    - ++, --, <<, and >> are also often overloaded
- GLM overloads many operators to make vector math convenient

Operator Overloading

- There are many legitimate uses of operator overloading
- But it can be very easy to misuse it
- In general, only use it to objectively make code clearer (to anyone who reads it)
  - even if `myColor%(BLUE->RED[-7])` makes sense to you

Operator Overloading

- You can even overload the function operator () for classes
  - Then you can call objects of that class like functions
  - But you could just give that class a *named* function, and call that function from your objects
- You can overload the assignment operator = for classes too

Operator Overloading

- The only operators you *can't* overload are:

  :: . (dot) ? : (ternary) sizeof

- Meaning you *can* overload pretty much everything else:

  `% ^ | & ~ > < == ! [] () new -> delete`

- https://isocpp.org/wiki/faq/operator-overloading

C++ Tips of the Week

QUESTIONS?