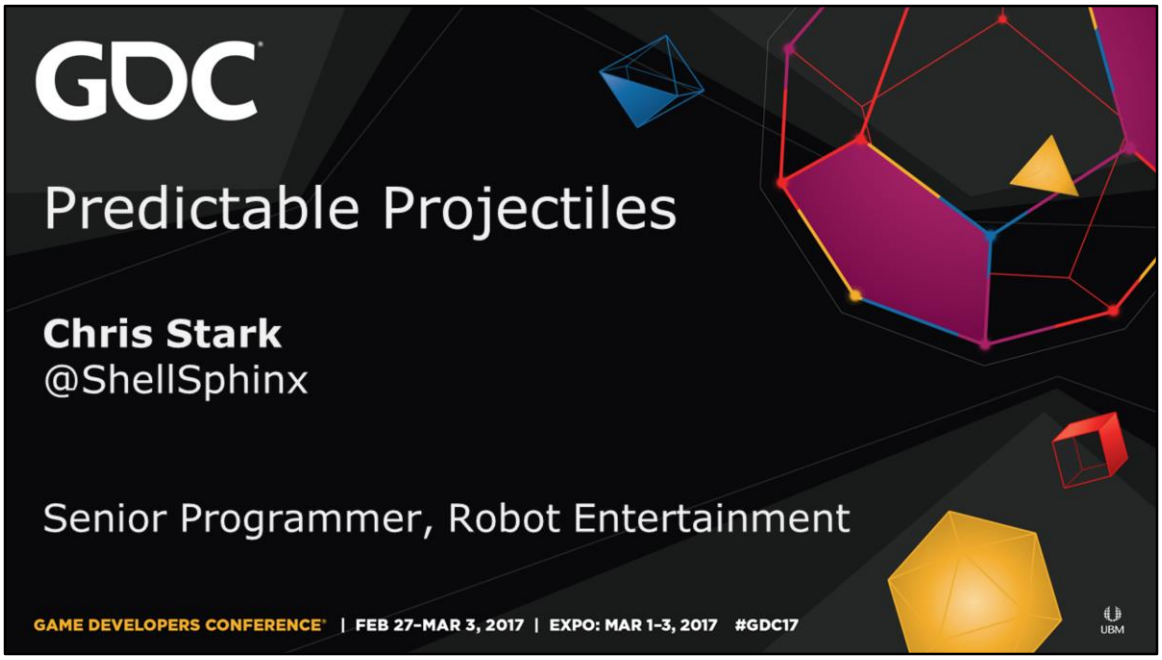
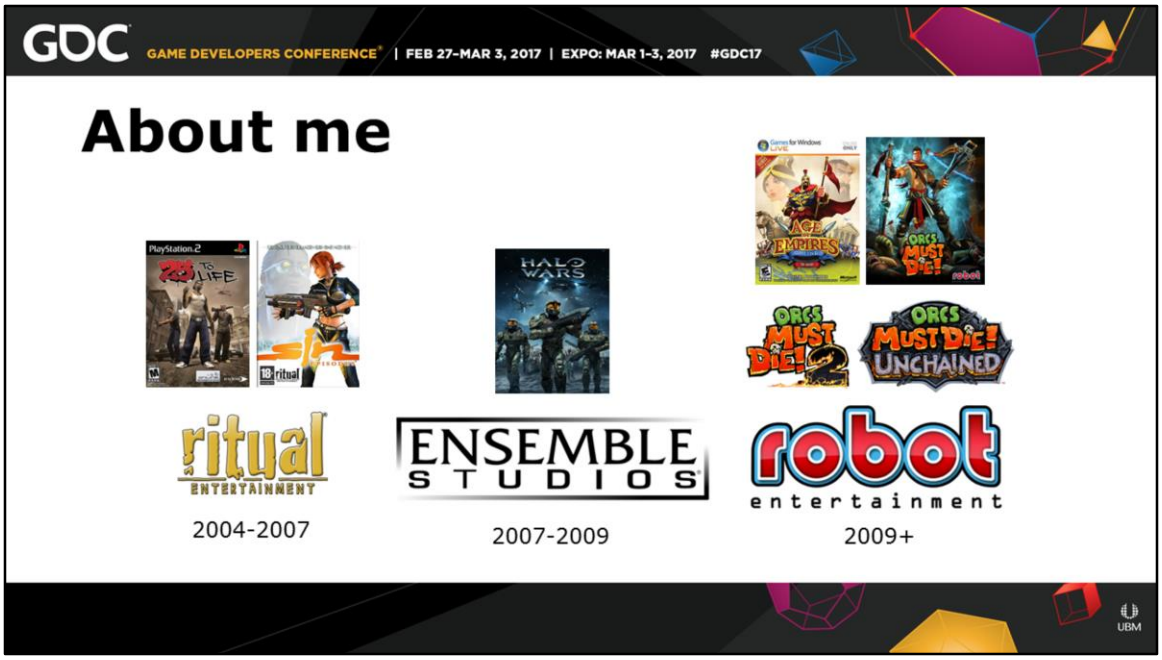


I'm very grateful for all the help and support I've gotten from my co-workers at Robot Entertainment.

The original version of these slides have animated .GIF images, which do not animate in this .PDF. I've tried to get still frames that capture the movie as best I can.

You can download the original slides at:
http://www.mathforgameprogrammers.com/



I've worked in the industry now for around 13 years at several different companies
Been at Robot for 8 years, primarily working on OMD franchise<next>

Orcs Must Die!

For those of you not familiar with Orcs, here's a quick rundown:
One Warmage (and friends) vs hordes of AI orcs, trolls, ogres, and worse, using traps and weapons to hold them back.
Tons of AIs that want to shoot, stab you, and blow you up.

Overview

- Player Movement
  - AI wants to shoot player
  - Linear and ballistic projectiles
  - Need to lead the player

Going to talk about my experiences doing ranged AIs on this franchise
Will be talking about math, code, and design
  Design drives the math and code; fun foremost!

Most action games have ranged AI,
  We're going to be specifically talking about ones that shoot/throw projectiles today.
  Some of this still relevant for hitscan.

Projectiles slow enough to dodge
Being able to dodge makes AIs more fun to play against
Gives player counter-play to ranged units

As a result, can't shoot at current position, the player won't be there when the projectile gets there

Overview

If you don’t lead the player, you get this. Looks pretty bad, doesn’t it?

Overview

Here’s what you’d like to see.

Overview

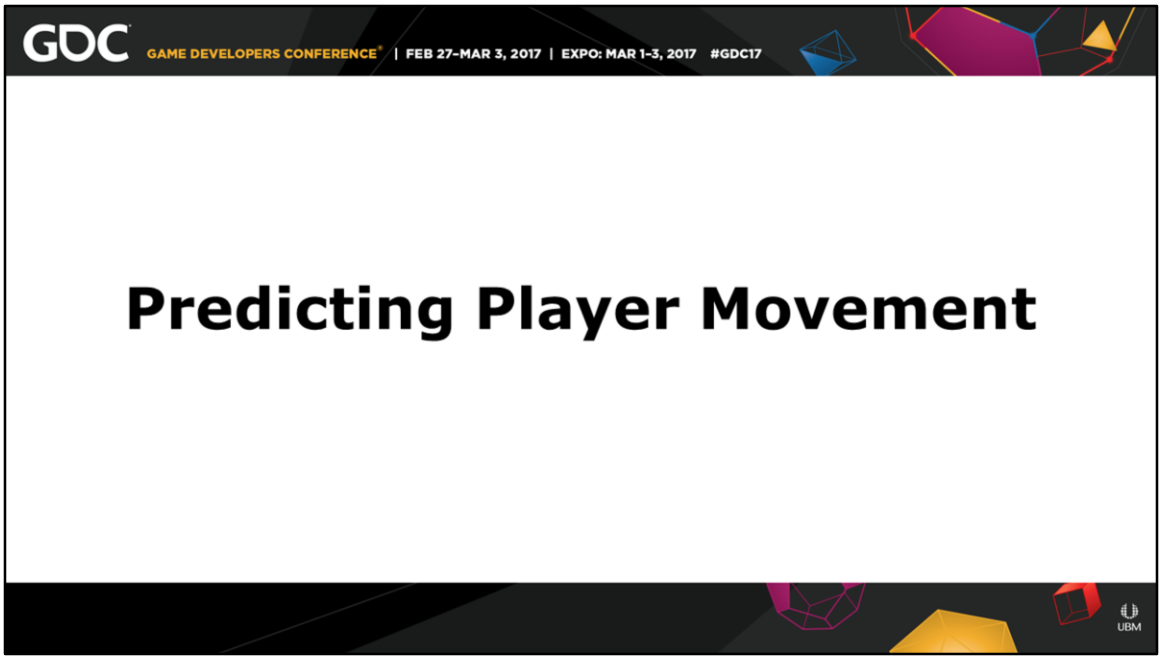
- Player Movement
  - AI wants to shoot player
  - Linear and ballistic projectiles
  - Need to lead the player
- Linear Projectiles

First class of projectiles is linear projectiles,

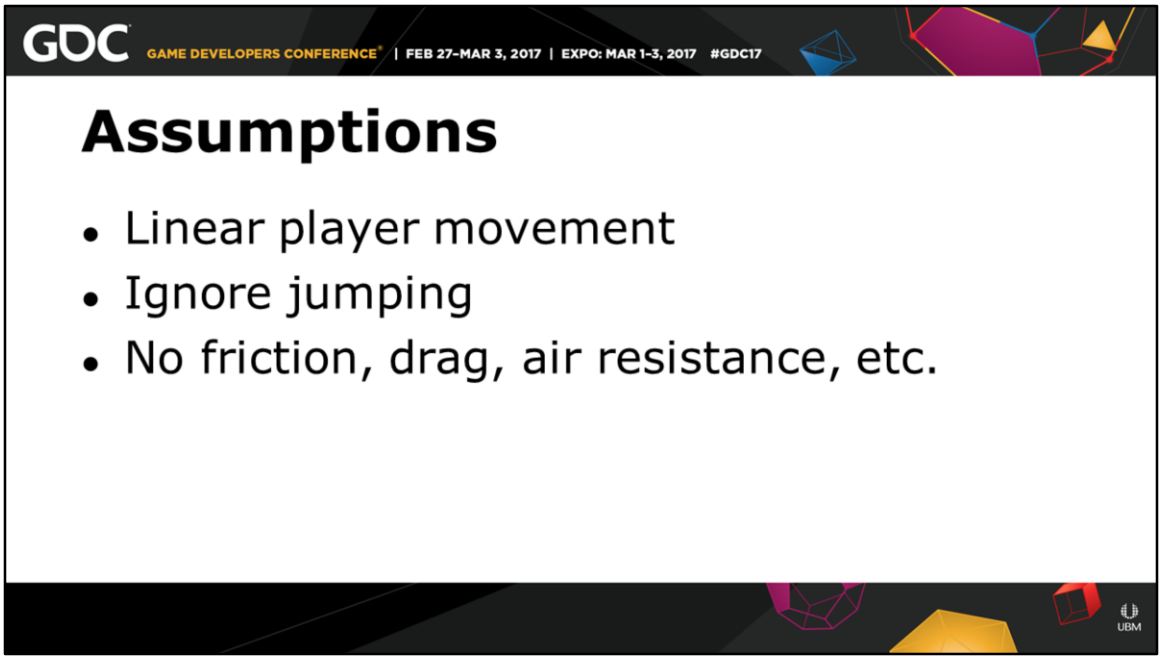
Overview

- Player Movement
  - AI wants to shoot player
  - Linear and ballistic projectiles
  - Need to lead the player
- Linear Projectiles
- Ballistic Projectiles

And the other is ballistic.
We'll be talking through how to aim, debug, and miss with both types of projectiles, and a few things I ran across while implementing them.

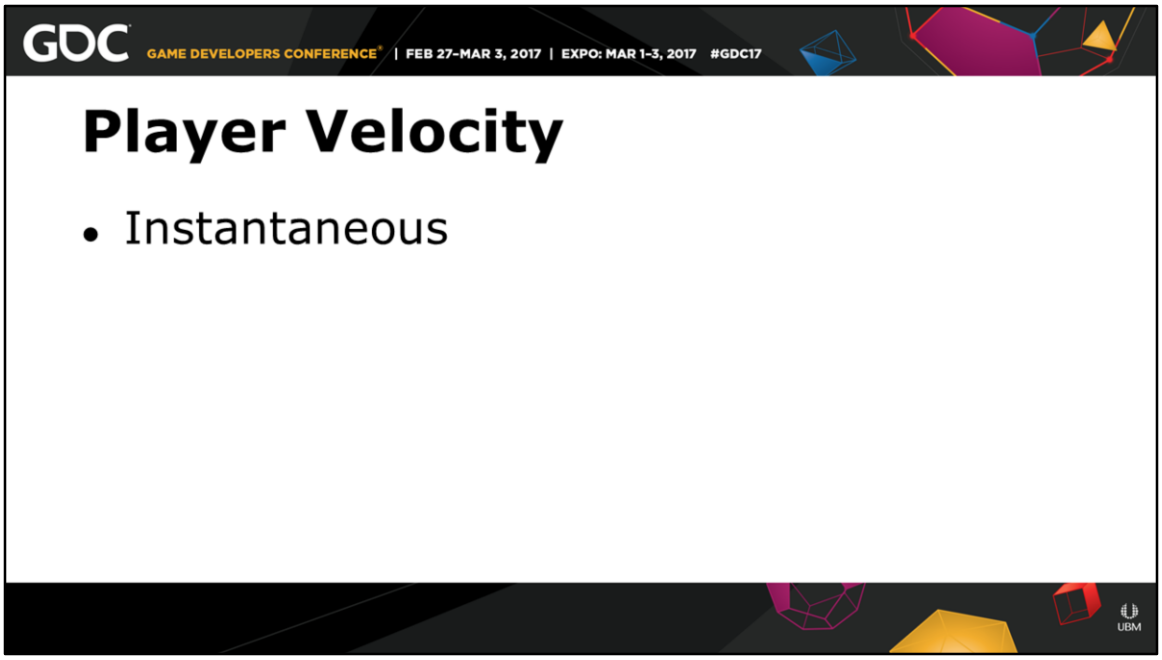


Let’s start off by talking about predicting player movement.



First, a few assumptions we're going to make
Want to keep the physics, math, and ultimately code, as simple as possible.

We're typically dealing with a short enough time frame we don't have to take rotations into account
We also want to ignore the player's vertical velocity when jumping, as it will throw off our results
Operating in a vacuum--not taking those into account greatly simplifies math



We need to know which way the player is going.
Easiest thing to do is see how fast the player is going right now and use that.

Player Velocity

- Instantaneous

(Over movie)—Works well when player moves like this

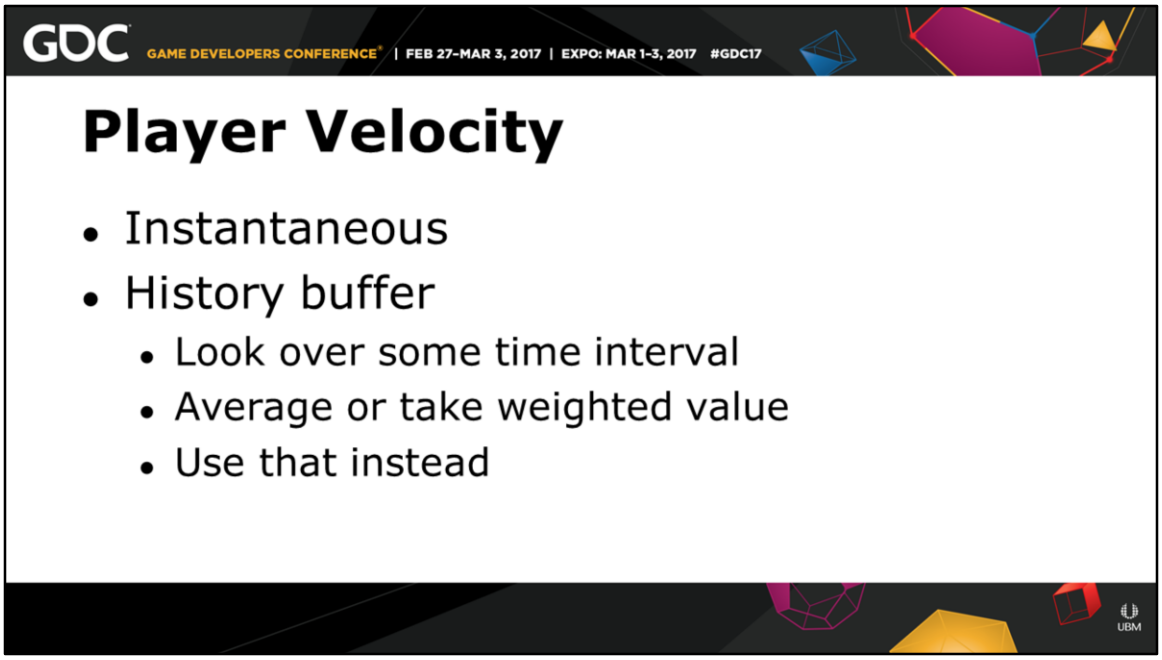
Player Velocity

- Instantaneous

(Over movie)—But not like this.

Need to handle players dodging, strafing, and starting/stopping.

Can we do better?



We can use a history buffer.
Better, more accurate result

Tweak until you have the right feel for players to still be able to dodge.

Predicting Player Movement

x_{0w}

Where:

x_{0w} = Original Warmage position

x_{0w}

Now that we've got a velocity, let's talk about how we use that to predict the player's movement.

Start with warmage in a position.

In our first two games, the player's origin was on the ground.
So make sure you use the center of the warmage's bounds for position and not the ground.

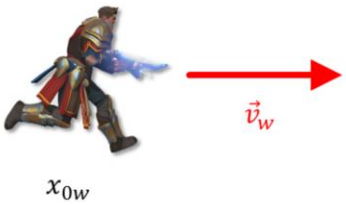
Predicting Player Movement

x_{0w}

Where:

x_{0w} = Original Warmage position

$\vec{v}_w$ = Warmage velocity



x_{0w}

We know what the warmage's velocity is from our history buffer.

Predicting Player Movement

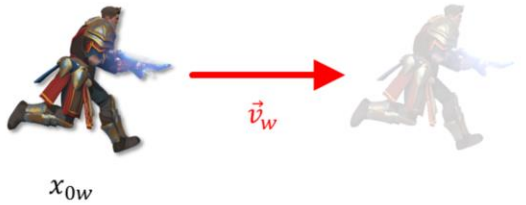
$x_{0w} + \vec{v}_w t$

Where:

x_{0w} = Original Warmage position

$\vec{v}_w$ = Warmage velocity

t = time



Going to be moving for t seconds

Predicting Player Movement

$x' = x_{0w} + \vec{v}_w t$

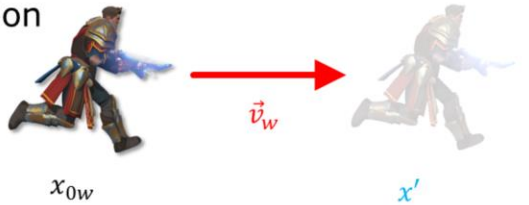
Where:

x' = New position

x_{0w} = Original Warmage position

$\vec{v}_w$ = Warmage velocity

t = time



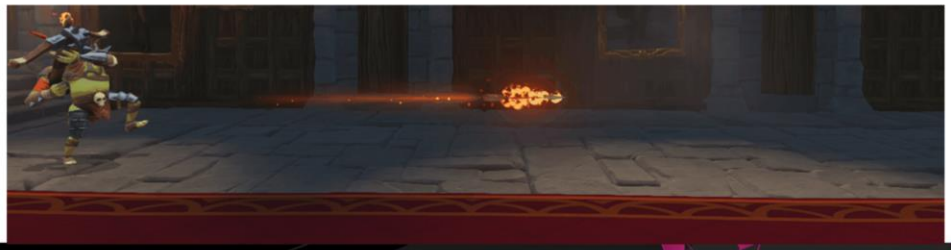
Which gives us our final predicted position, x'.

We'll see this again in a minute.

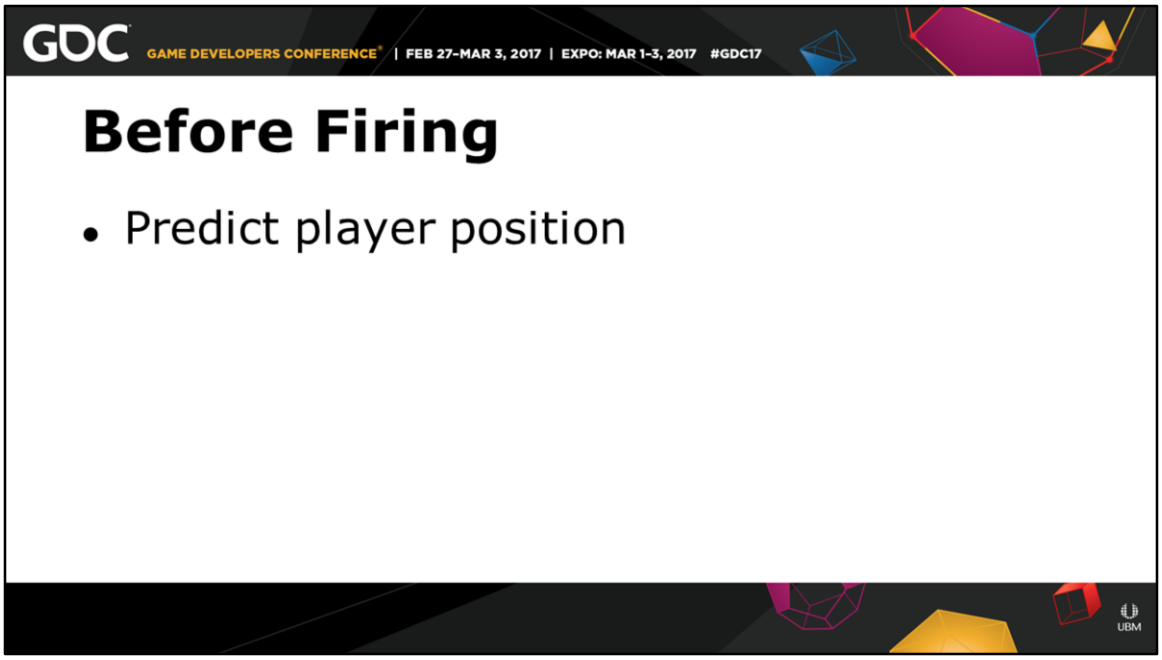
Linear Projectiles

Linear Projectiles

- Move at a constant speed
- Constant direction

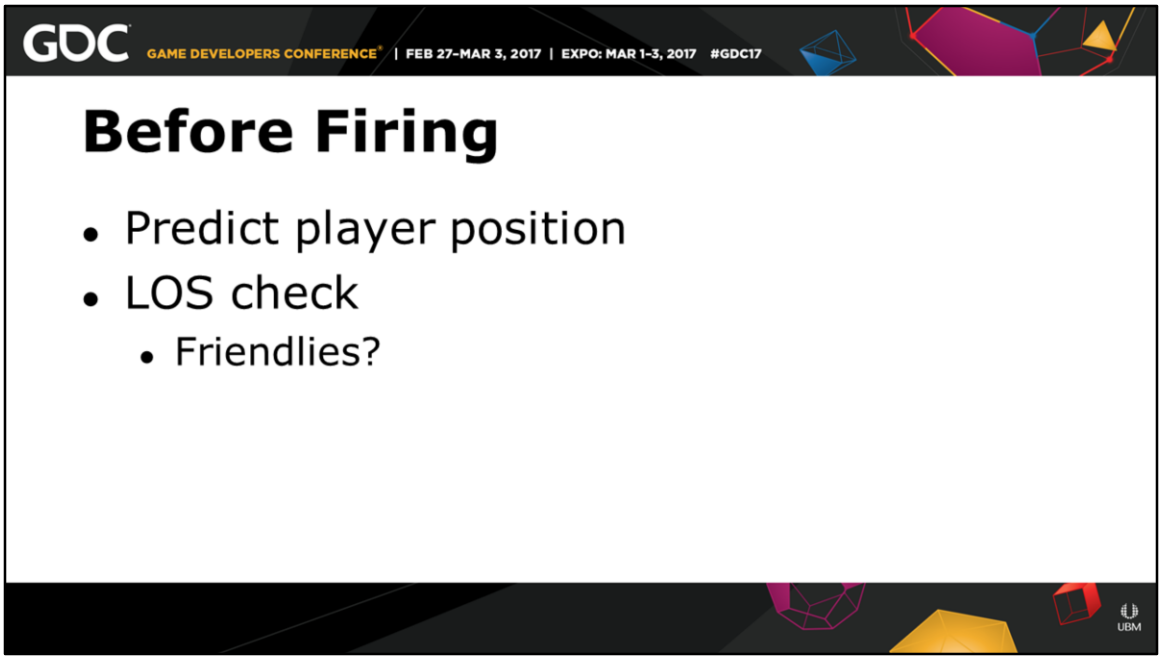


This is a linear projectile.
No gravity—goes in straight line forever.

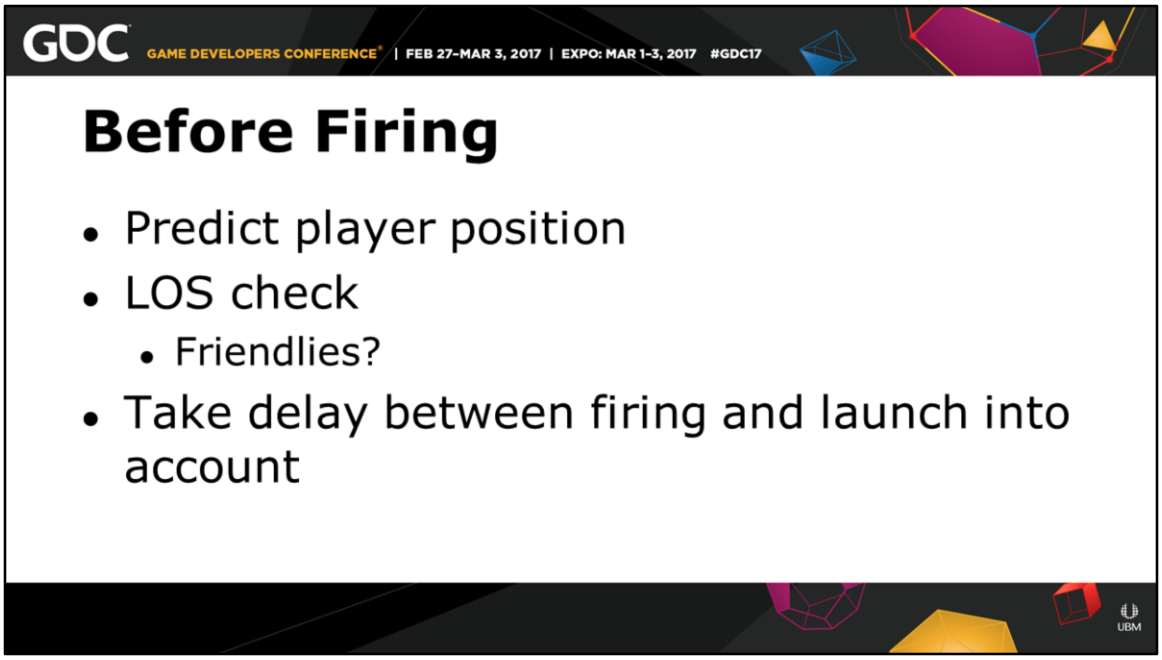


Before you go firing off at the player, there's a few things to do first.

We'll talk about how to predict the position here in a moment.



Don’t want to fire through walls
We changed our minds a couple of times on firing through friendly units.
Wound up not due to player confusion as to where the shot came from, but YMMV.
May want to do LOS check for friendlies too.



If you do the math before firing animation plays:
Add delay between fire and launch to t for warmage movement.

Predicting Position

x_{0p}

Where:

x_{0p} = Projectile launch position

Start with an initial position.

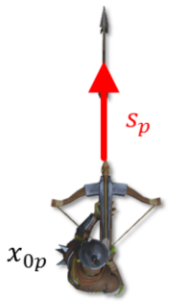
Predicting Position

x_{0p}

Where:

x_{0p} = Projectile launch position

s_p = Projectile speed



We have a constant speed, which in OMD we let design set.

We also have a maximum range that design can set, to help keep entity counts down so projectiles don't potentially live forever.

Predicting Position

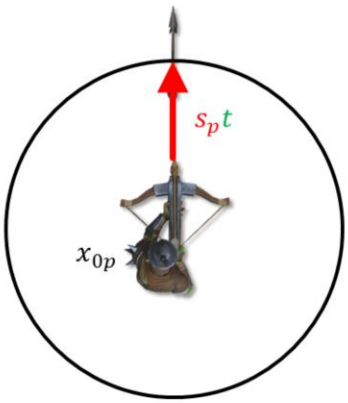
$x_{0p} + s_p t$

Where:

x_{0p} = Projectile launch position

s_p = Projectile speed

t = Time since launch



It runs for some amount of time t.

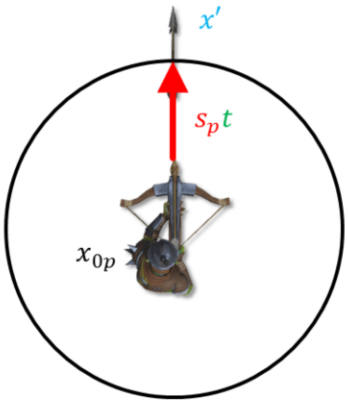
AI can fire in any direction, forms a circle (sphere in 3D)
Radius of circle is sp * t

Predicting Position

$x' = x_{0p} + s_p t$

Where:

- x' = New position
- x_{0p} = Projectile launch position
- s_p = Projectile speed
- t =Time since launch

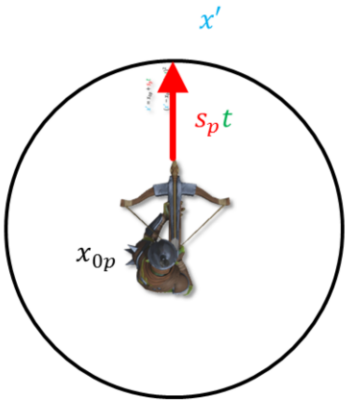


X’ is on the circle.

Predicting Position

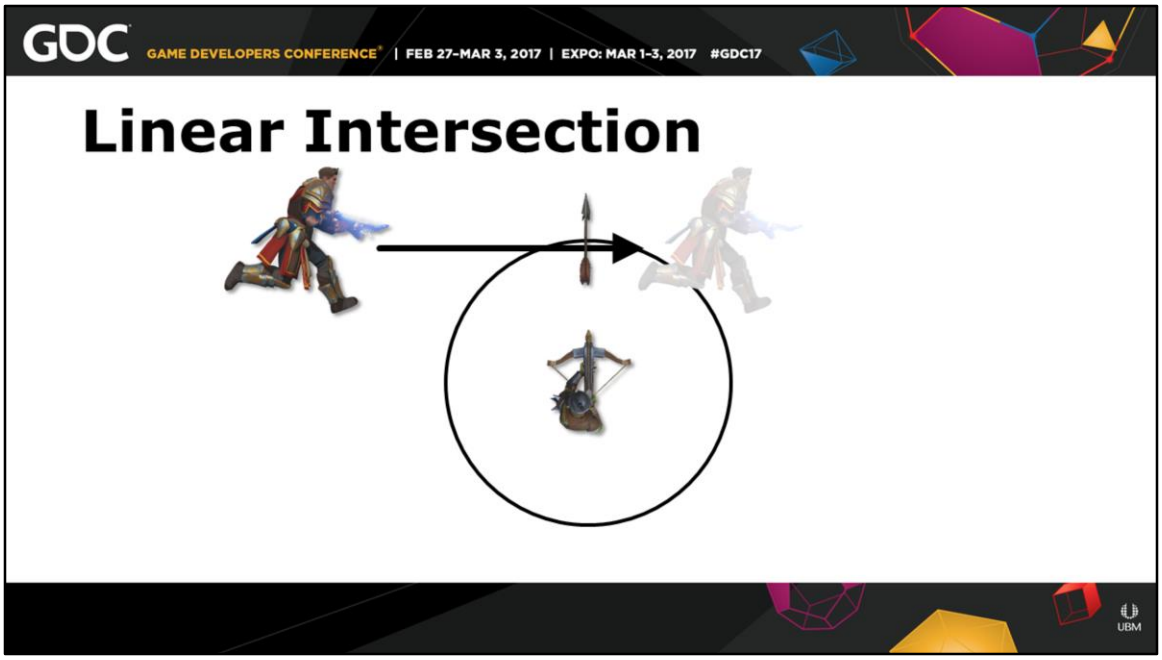
$$x' = x_{0p} + s_p t$$

$$(x' - x_{0p})^2 = (s_p t)^2$$



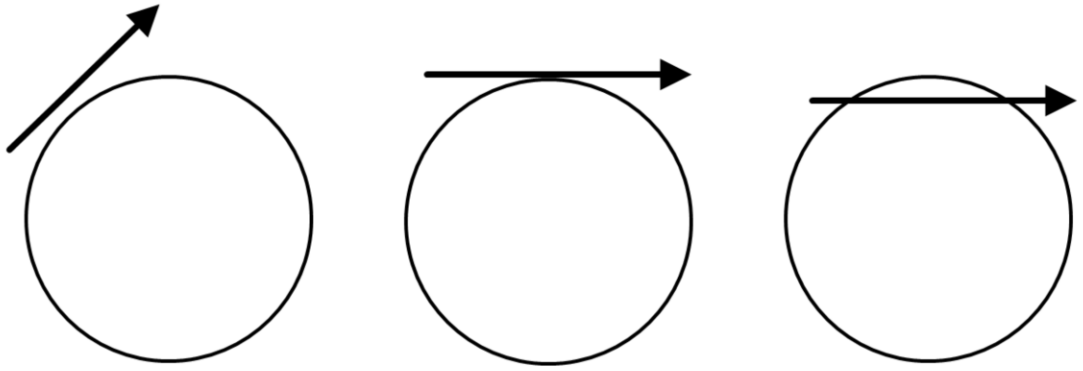
This can also be written as the equation for a circle, like so.

Going to do all math in 2D, works in 3D too.



The result when we put the player's movement and projectile fire together is a circle vs line intersection.

Linear Intersection



Thinking about it visually, we'll have 0, 1, or 2 solutions.

Linear Intersection

$x' = x_{0w} + \vec{v}_w t$ Predicted Player Movement

$(x' - x_{0p})^2 = (s_p t)^2$ Projectile Position

Here are our equations that we came up with for predicted player movement and projectile movement.

Linear Intersection

$x' = x_{0w} + \vec{v}_w t$ Predicted Player Movement

$(x' - x_{0p})^2 = (s_p t)^2$ Projectile Position

X' is the same in both of these equations; it's our intersection point.

Linear Intersection

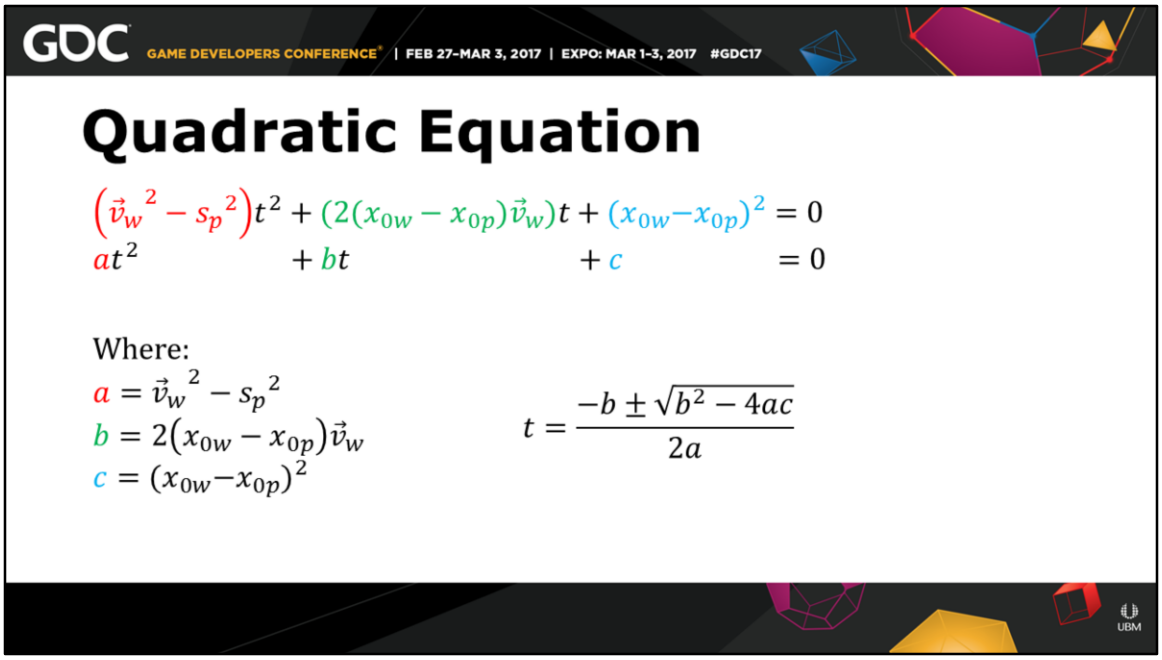
$x' = x_{0w} + \vec{v}_w t$ Predicted Player Movement

$(x' - x_{0p})^2 = (s_p t)^2$ Projectile Position

X' is the same in both of these equations; it's our intersection point.

We don't know t, which is our intersection time.

If you solve these equations for t, you'll wind up with…

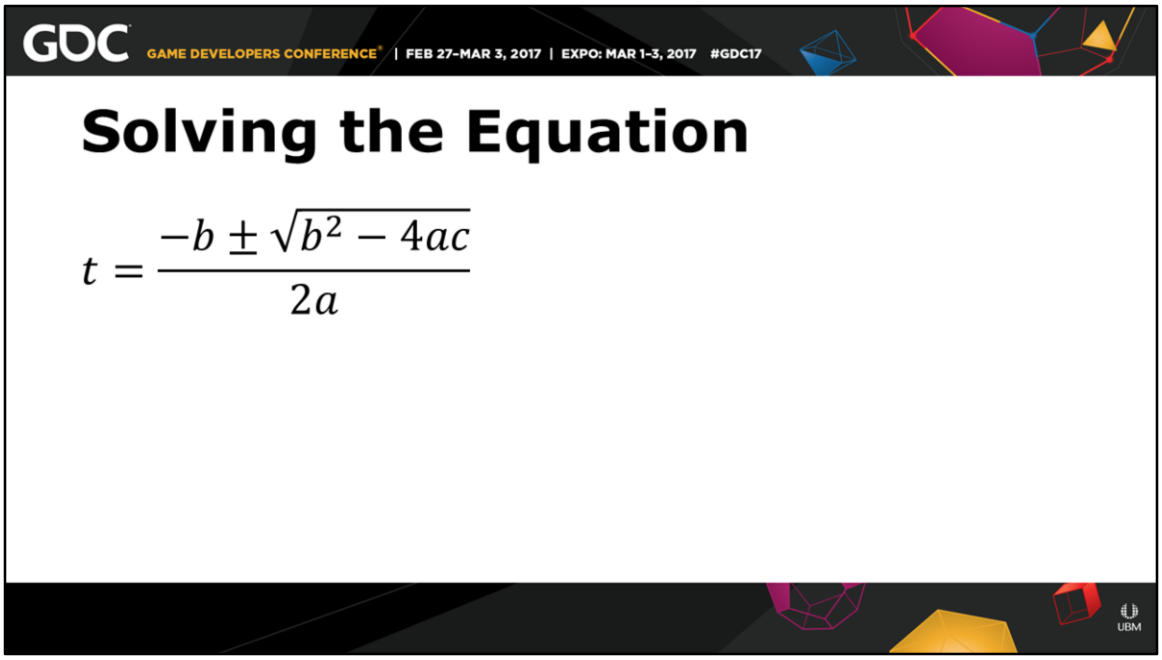


A quadratic equation, can solve with quadratic formula.

A, b, and c are all scalars.

Vector multiplies in a, b, c are just dot products.

For folks interested in how to get from those equations to here, it's all detailed out in the online slides.



Let’s take our quadratic formula.

Just code it up and go, right?
No.
There’s a few special cases on solving the quadratic equation from code to think about.

Solving the Equation

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

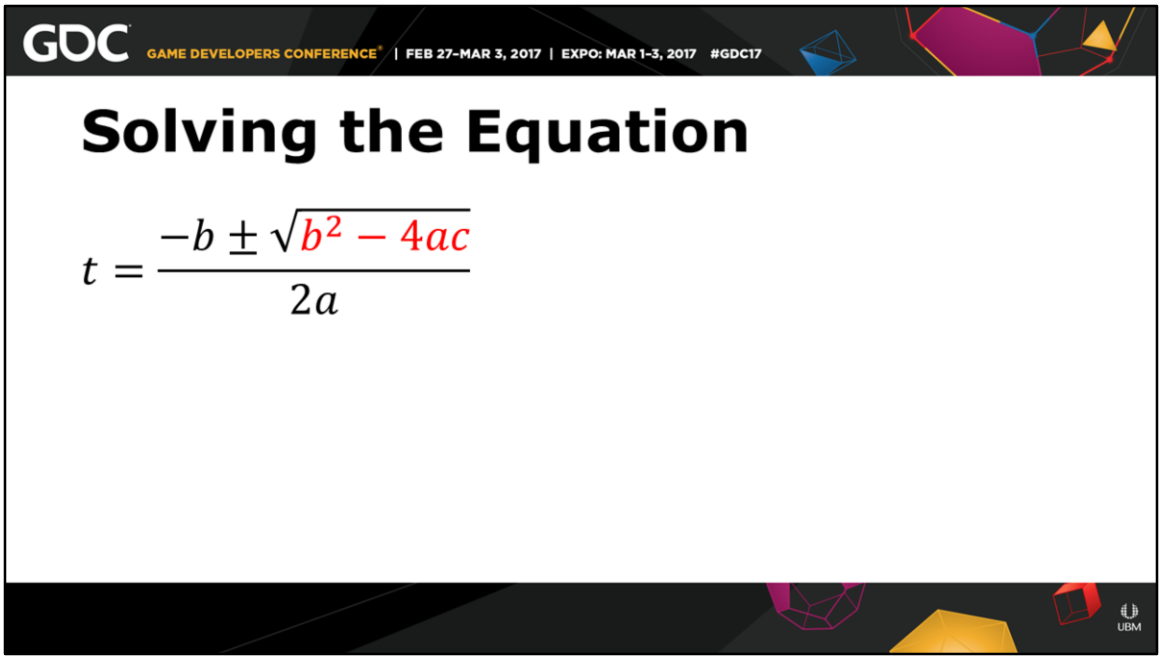
If a is 0, remember that we're solving $at^2 + bt + c = 0$,

Solving the Equation

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$t = -\frac{c}{b}$$

so we can still solve it.



If discriminant is negative, out of luck
If 0, one solution.

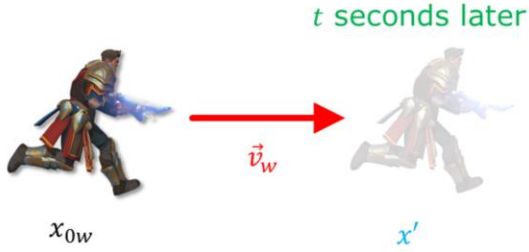
Otherwise:
Take smallest positive solution.
Negative solutions are invalid.

Almost never have 2 valid solutions, one will almost always be negative.

Firing

- Plug t into Warmage movement
- Get the location we're aiming at:

$$x' = x_{0w} + \vec{v}_w t$$



Take t from our solution and get location.

Can use x' to get a direction vector to fire in

Firing

Firing Direction:
$\|x' - x_{0p}\|$

Final Velocity:
$\vec{v}_p = \|x' - x_{0p}\| s_p$

Final velocity is the direction vector between our launch position and the warmage's position * our projectile speed

No valid solution

- Don't fire

You may wind up with no valid solutions.

If you've done your math before firing, you can just do nothing.

No valid solution

- Don't fire
- Fire at current location.

Fire at the player's current location.

If you do your math at firing time, like we did, found that it looks bad if nothing is fired, so we just fired at the player's current location.

Debugging

Once you have everything in place
To double check math, set up debug drawing in game.

Debugging

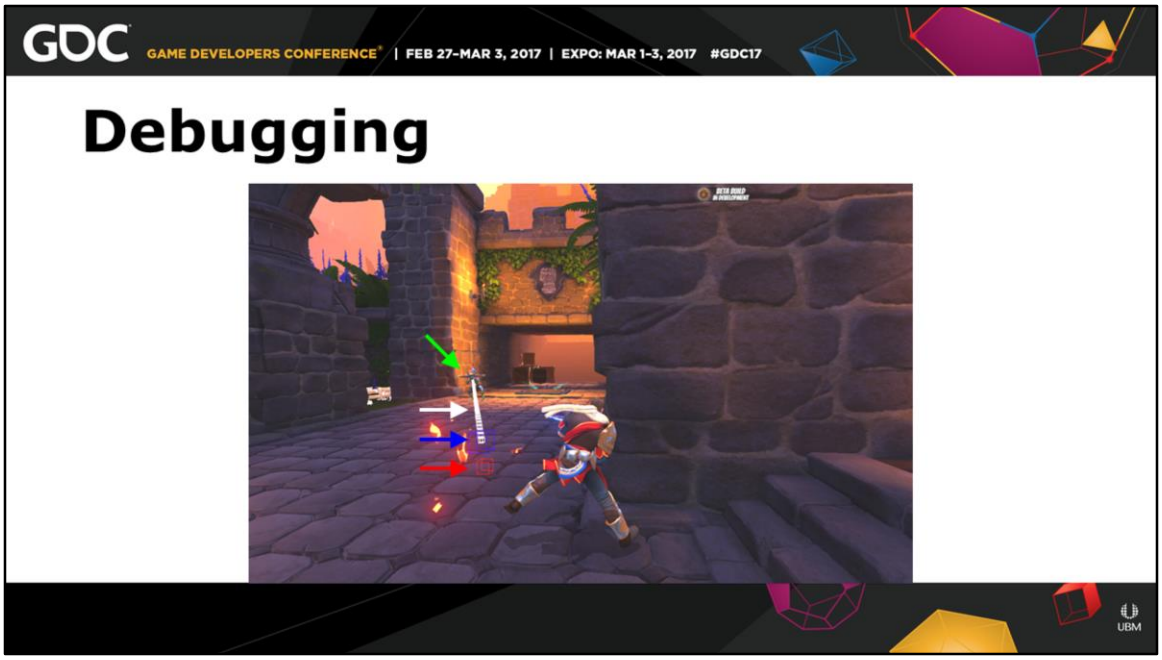
Draw launch point (green)—it's under the orc here so can't see.

Debugging

predicted point (red)

Debugging

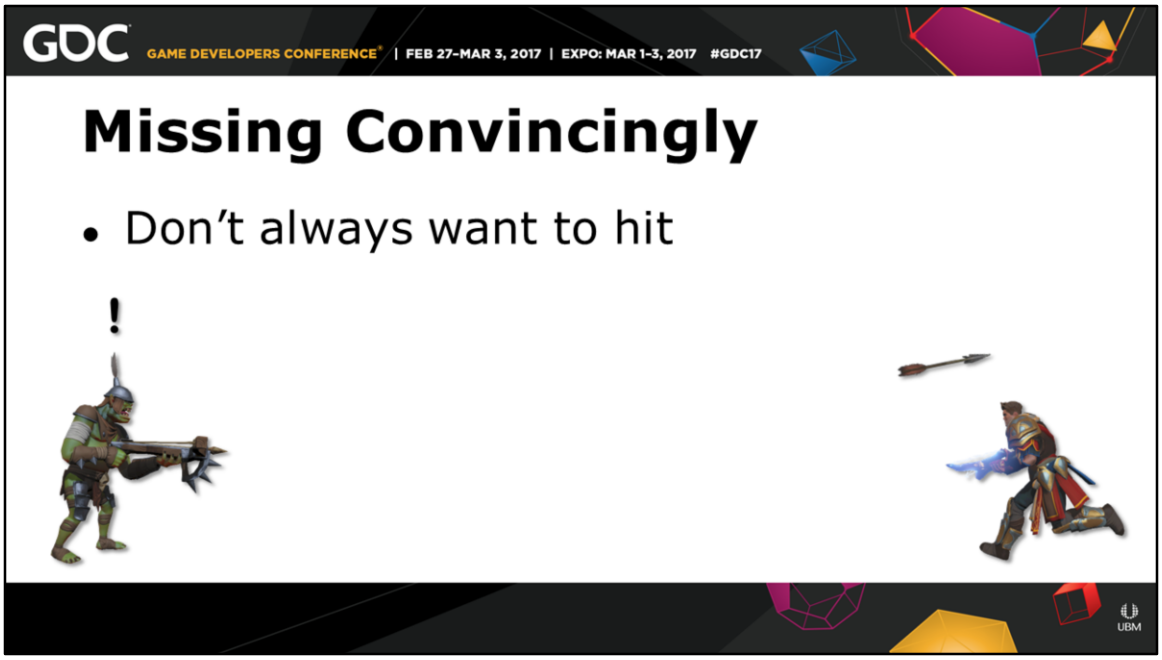
projectile travel (white)



and projectile impact (blue)

Explain that difference between impact and predicted is due to player collision cylinder.

Lines up well here.



Now that we've talked about hitting, let's talk about missing.

Writing fun AI, not aimbots
Had accuracy setting
Horizontal and vertical

Missing Convincingly

- Don't always want to hit
  - [-Accuracy, Accuracy]

Picked random number in range for both directions
Defaulted both to 2 degrees (tweaked multiple times!)
So 4 degree cone.

Want to be careful with too much up or down, looks bad, makes AI look dumb.
They are orcs, but we don't want them to look *too* dumb!

Used these to generate a rotation matrix and apply to initial projectile direction.

Missing Convincingly

At the start of the game, that was good enough.
Then we added barricades, which block AI movement and can be damaged by Ais.
Archers wound up being very rough on barricades, which wasn't fun.

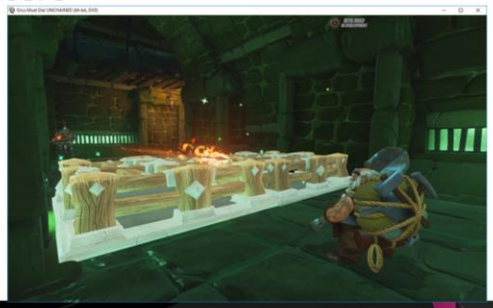
Missing Convincingly

- Added tilt, range 0-1
- [-Accuracy * tilt, Accuracy * (1 – tilt)]

Added a tilt value to shrink and truncate the cone
.5 is centered, 0 removes cone bottom, 1 removes cone top.
We defaulted to 0.75, which got the majority of shots clear of barricades.
Don’t want all down removed.
Want to incent players to not fight around barricades.

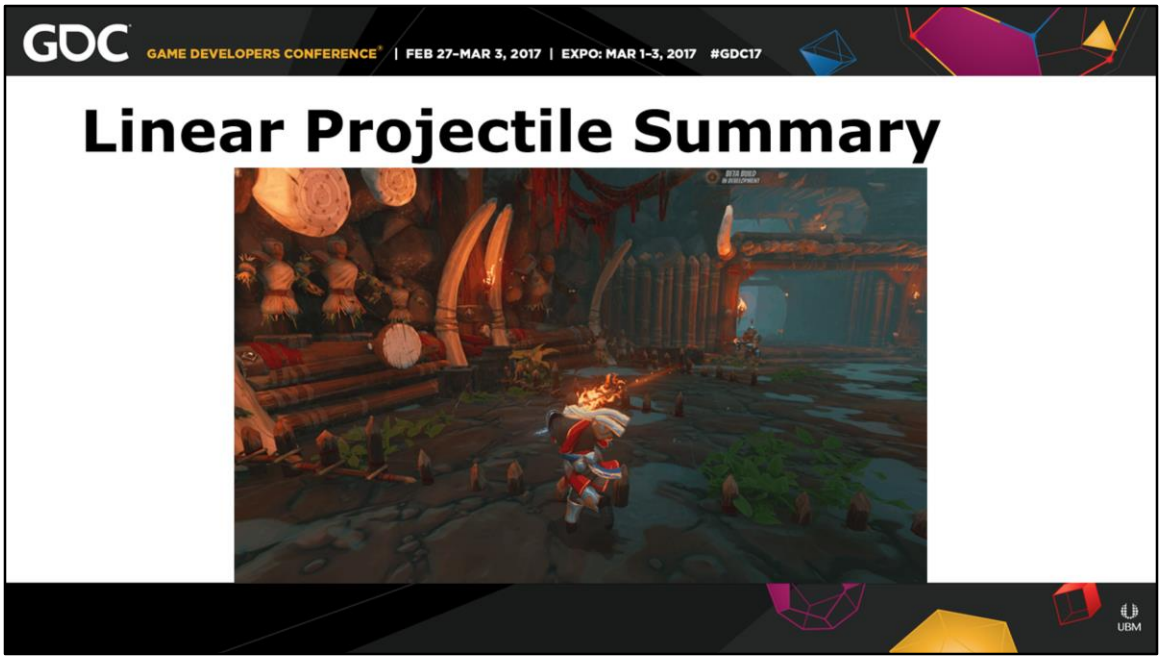
Missing Convincingly

- Still wasn't enough!
- Short characters



We also have some really short characters whose center was below the top of the barricades.
Wound up looking at their center:
If below the top of the barricade, move aim point up above barricade
But no higher than the top of the character's bounds.

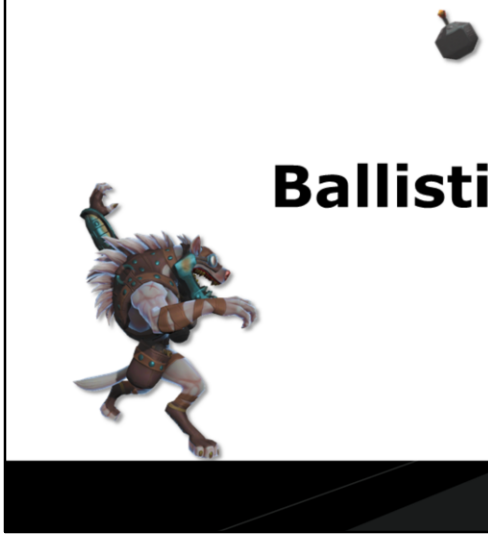
Threw a warning if characters were too short, but have never had it come up.



Here's our final result.

Linear projectiles
- Have speed
- Solve for t to get intersection time/location

Missing
- Use a cone
- Different angle for horizontal/vertical
- May need additional game-dependent tweaks



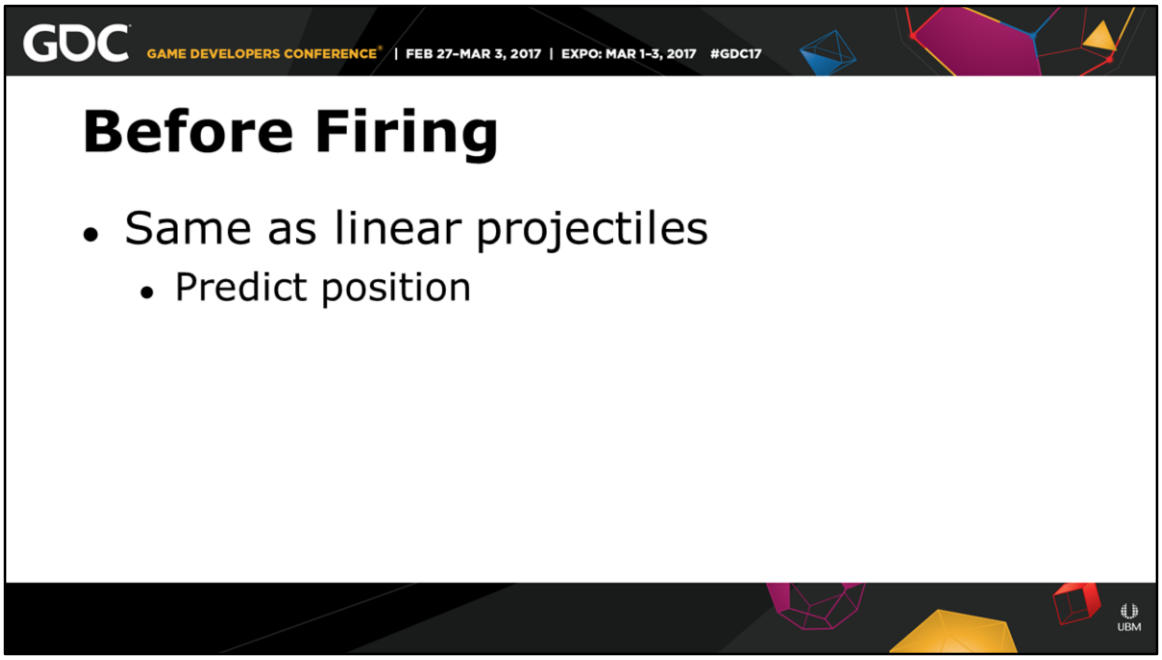
We have one other type of projectile to consider, ballistic projectiles.

Ballistic Projectiles

- Initial fixed velocity
- Affected by gravity
- Arc as a result

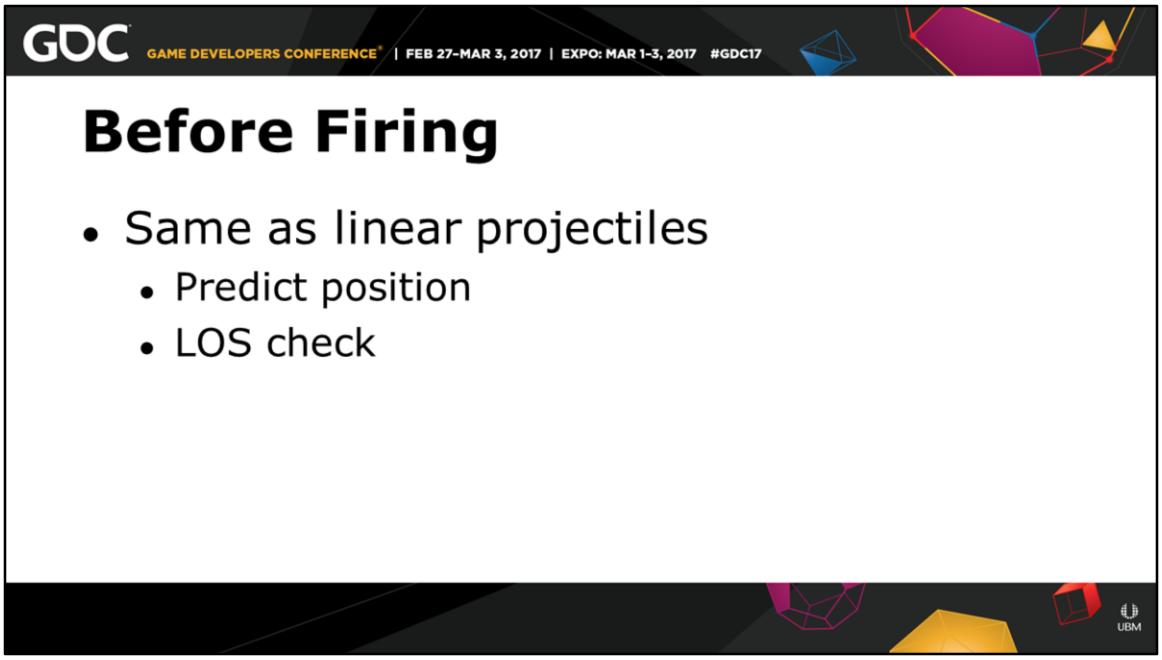
This is a ballistic projectile.

Initial velocity has an up component to it, unlike linear projectiles.



You really want to do the same set of operations before firing.

You may want to predict on the ground instead of the character's center since grenades can bounce—less bouncing around.



In theory, unlike linear projectiles, you'd want to do a series of traces to approximate the arc to make sure you don't hit geo.

We just did a straight line check between the two—almost always had rectangular hallways without obstructions up top, so good enough—saved us some perf. YMMV.

Before Firing

- Same as linear projectiles
  - Predict position
  - LOS check
  - Take delay between firing and launch into account

Just like before.

Ballistic Projectiles

$$x' = -\frac{1}{2}gt^2 + \vec{v}_{0p}t + x_{0p}$$

This is your basic ballistic motion equation.
After seeing how we solved linear projectiles, you'd assume that we solve this similarly, by setting x' equal and solving.

Ballistic Projectiles

$$\left(-\frac{1}{2}gt^2 + \vec{v}_{0p}t + x_{0p}\right)^2 = (s_w t)^2$$

...

$$\left(\frac{1}{4}g^2\right)t^4 + \left(g\vec{v}_{0p}\right)t^3 + \left(g x_{0p} + {\vec{v}_{0p}}^2 - {s_w}^2\right)t^2 + 2\left(\vec{v}_{0p} \cdot x_{0p}\right) + {x_{0p}}^2 = 0$$

a b c d e

Quartic equation:
$ax^4 + bx^3 + cx^2 + dx + e = 0$

You wind up getting a quartic equation as a result.

Have some references in the online slides if you're curious about how to solve these, most notably Graphics Gems I.

We actually wound up cheating the math a bit instead, and did some approximations.

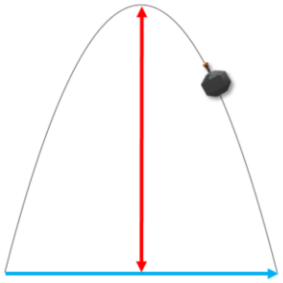
What We Did

- Started with an arc

We started with an arc

What We Did

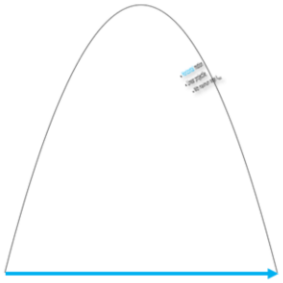
- Started with an arc
- Horizontal and vertical motion



You can actually break the arc's motion down into two parts, horizontal and vertical.

What We Did

- Horizontal motion
- Linear projectile
  - With maximum range R_{max}

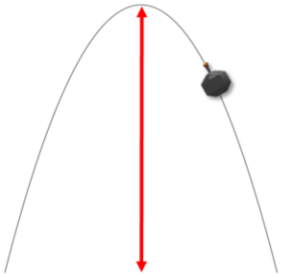


The horizontal motion is actually the same as the linear projectile solution we just came up with, with the key difference of having a max range.

We'd solve it for t and use that to get our intersection point.

What We Did

- Vertical motion



We want that intersection point to be what we hit with our combined motion.

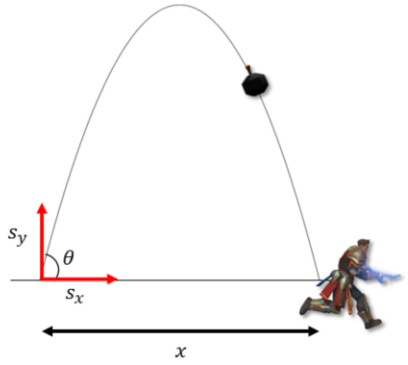
Let’s solve for our throw so the projectile will land at that point.

Vertical motion

$$x' = -\frac{1}{2}gt^2 + s_p t$$

$$x = s_p t \cos(\theta)$$

$$y = -\frac{1}{2}g_y t^2 + s_p \sin(\theta) t$$



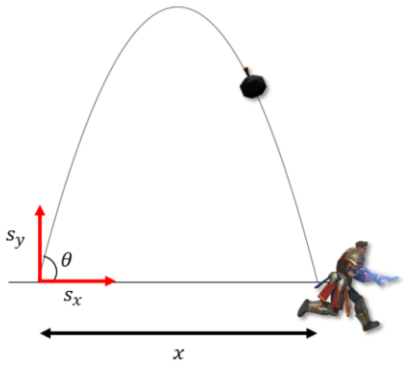
Break projectile equation into x and y components, which are our distances to target:
X-horizontal
Y-vertical

Want to solve for θ, which we can use to calculate our horizontal and vertical throw speeds.

Two equations, two unknowns.

Vertical motion

$$-\frac{gx^2}{2{s_p}^2}\tan^2(\theta) + x\,tan(\theta) - \frac{gx^2}{2{s_p}^2} - y = 0$$

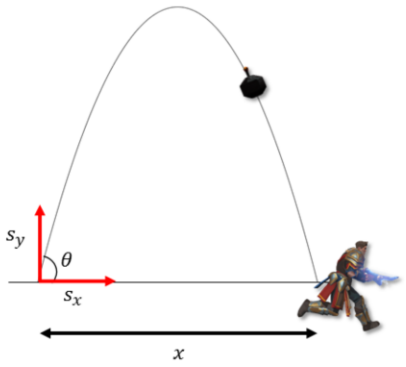


Once you factor that through, you get this,

Vertical motion

$$-\frac{gx^2}{2{s_p}^2}\tan^2(\theta) + x\,tan(\theta) - \frac{gx^2}{2{s_p}^2} - y = 0$$

a b c



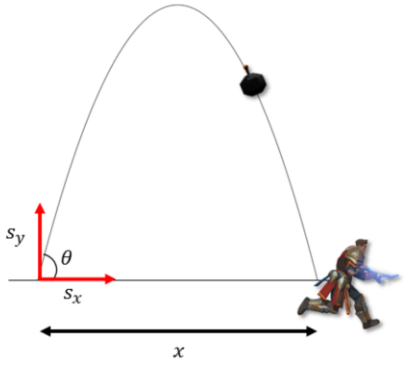
which can be solved via the quadratic equation.

Vertical motion

$$-\frac{gx^2}{2{s_p}^2}\tan^2(\theta) + x\,tan(\theta) - \frac{gx^2}{2{s_p}^2} - y = 0$$

a b c

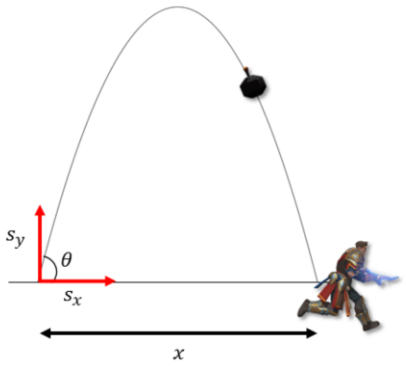
$$\boldsymbol{\theta} = tan^{-1}\left(\frac{{s_p}^2 \pm \sqrt{{s_p}^4 - g\left(gx^2 + 2y{s_p}^2\right)}}{gx}\right)$$



Which you can use to get θ, the launch angle

Vertical motion

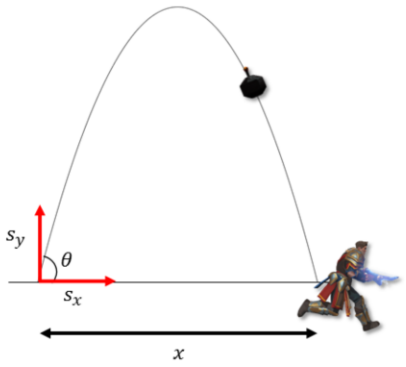
- No solution
  - Throw directly at player with gravity
- Two solutions



For one solution, you're good.
Keeping gravity on is important to feel.
We just picked the smallest throw angle if we had two solutions.
Felt it looked better.

Vertical motion

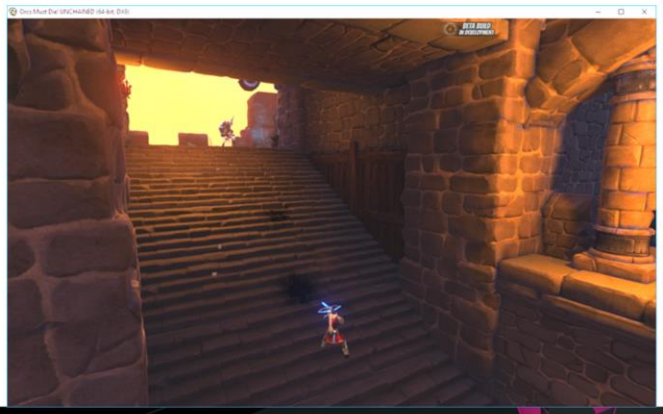
$$s_x = s_p \cos(\theta)$$
$$s_y = s_p \sin(\theta)$$



Now that we have theta, sx and sy are easy to calculate.

Now we need a horizontal and vertical vector to throw along.

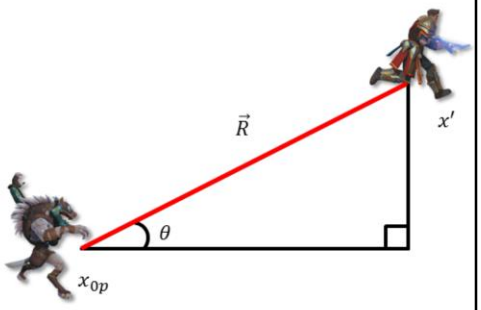
Vertical motion



That way it all works in a situation like this.

Vertical motion

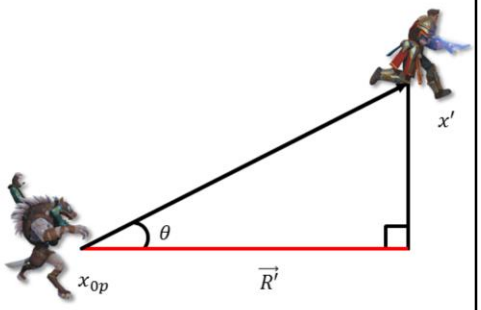
- Calculate throw vectors



R, is our vector between launch and target, which may be non-horizontal.

Vertical motion

- Calculate throw vectors



Project it onto the ground plane to get R’ (math works out so you drop the up component of your vector).

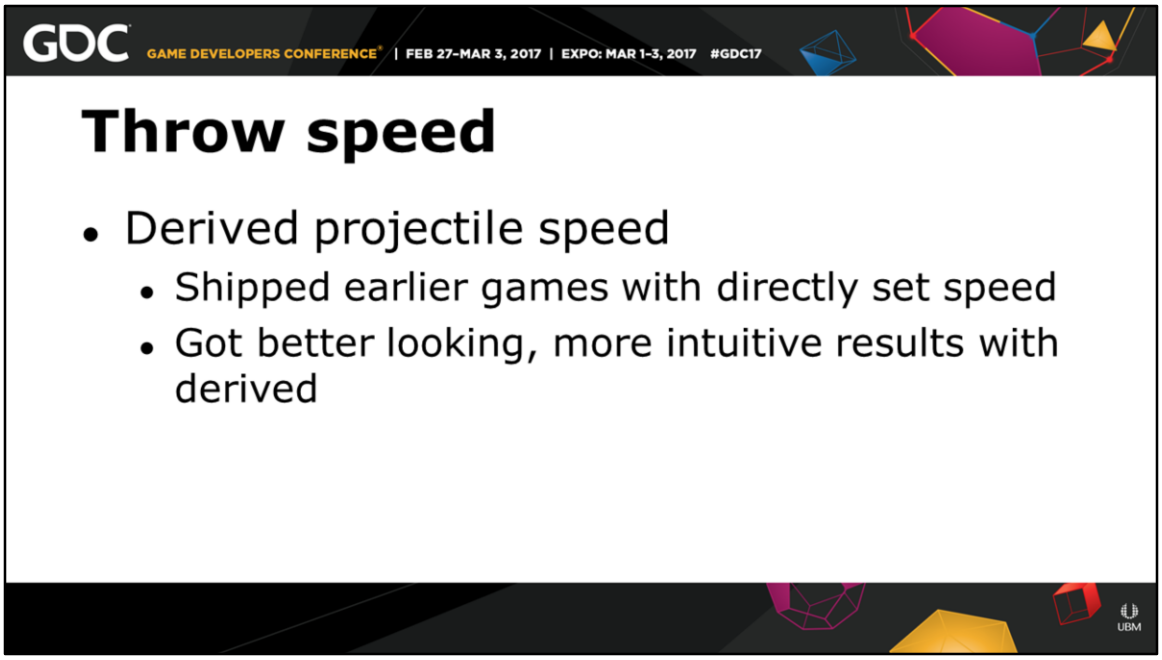
Vertical motion

- Add together horizontal and vertical.

$$\vec{v}_p = \left\|\overrightarrow{R'}\right\| s_x + (0, 0, s_y)$$

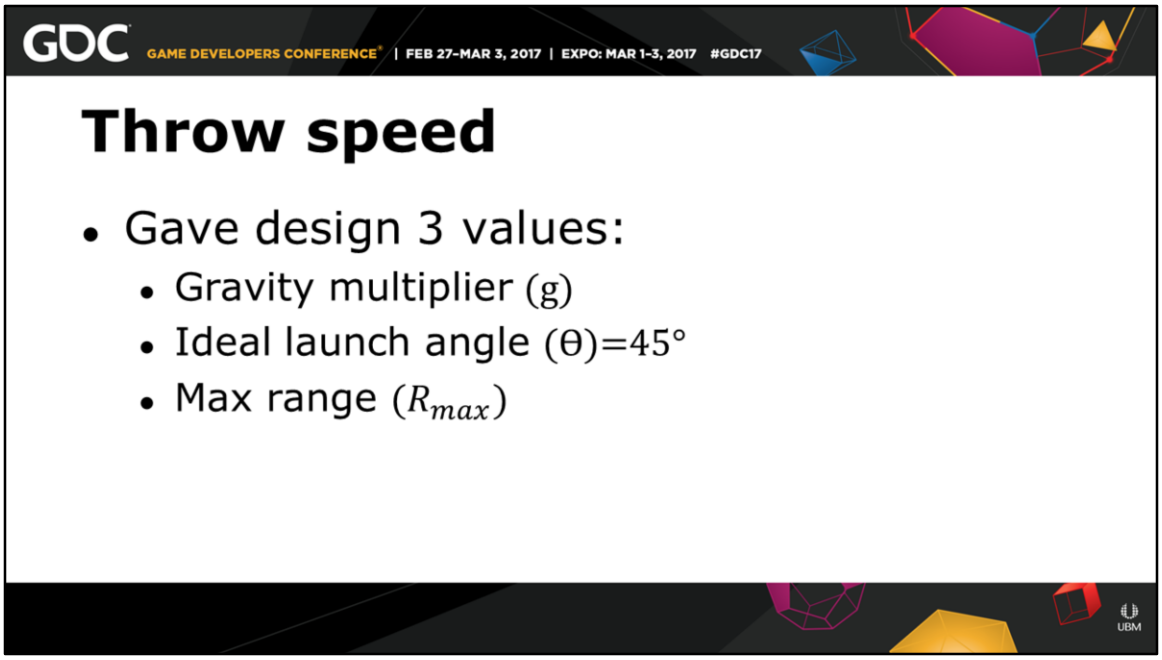
Add your horizontal and vertical throws together, and you're in the grenade throwing business!

Using the quadratic is not a perfect solution, as it can slightly alter your impact time, but it was accurate enough for our needs—plus our grenades are AOE. ☺
It will still hit your impact position.



On our earlier games, we let design set the throw speed directly just like for linear projectiles.

We found that giving them some values describing the behavior of the projectile and deriving the speed from that gave us a better looking result, and was more intuitive.



Gravity multiplier helps you make it feel heavier (large values) or get more arc (small values).

We defaulted the ideal launch angle to 45 degrees—since it's the maximum distance you can get, we'd get the lowest possible speed for our range, even if we didn't wind up throwing at a 45 degree angle. Don't think design ever changed it.

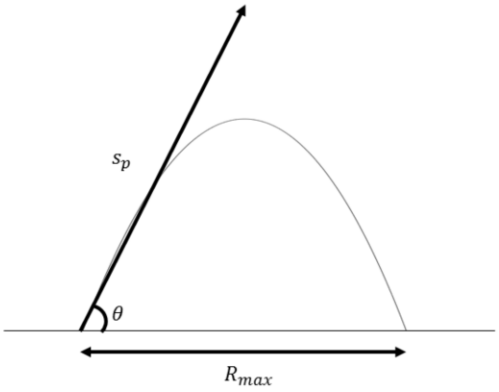
Max range had a constraint on how large it could be due to how we implemented.

Throw speed

$$x' = -\frac{1}{2}gt^2 + s_p t$$

$$x(t) = s_p t \cos(\theta)$$

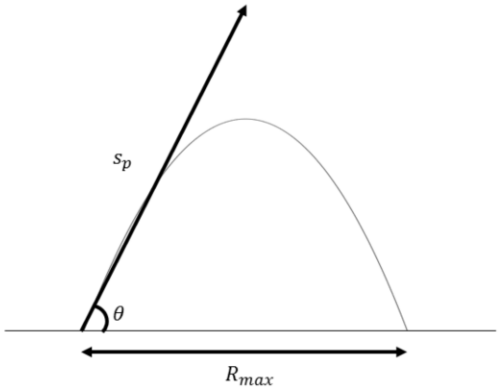
$$y(t) = -\frac{1}{2}g_y t^2 + s_p \sin(\theta) t$$



Break projectile equation into x and y components, just like we did before.

Throw speed

$$x(t) = R_{max}$$
$$y(t) = 0$$
$$\ldots$$
$$s_p = \sqrt{\frac{R_{max} g}{\sin(2\theta)}}$$



This time though, solve for speed where the projectile comes back to the ground.

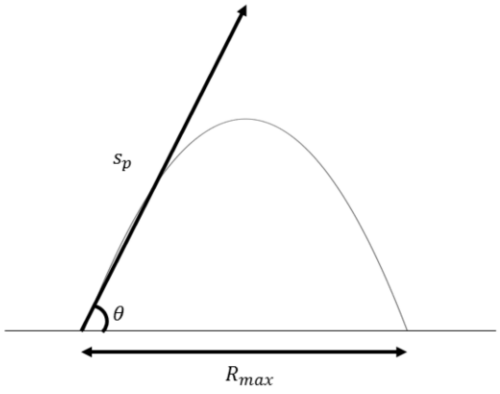
This is the speed we used in our quadratic equation solution earlier.

Once again, details are all in the online slides.

Throw speed

$$x(t) = R_{max}$$
$$y(t) = 0$$
$$\dots$$
$$s_p = \sqrt{\frac{R_{max}\, g}{sin(2\theta)}}$$

$$\boldsymbol{s_x = s_p\, cos(\theta)}$$



And this is what we used for our speed on the linear intersection. The cosine comes from trig identities (SOHCAHTOA).

Max Range

- Minimum ceiling height
- Capped max range to not hit

$$y(t) = h = -\frac{1}{2}gt^2 + s_p t \sin(\theta)$$

...

$$R_{max} = \frac{4h}{tan(\theta)}$$

Art and design agreed on a minimum value for ceiling height.

Don't forget to take into account what height the projectile is being launched from (and that this can vary per character). Your actual height is the ceiling height minus that number.

Solved equation when projectile was at it's peak (h)

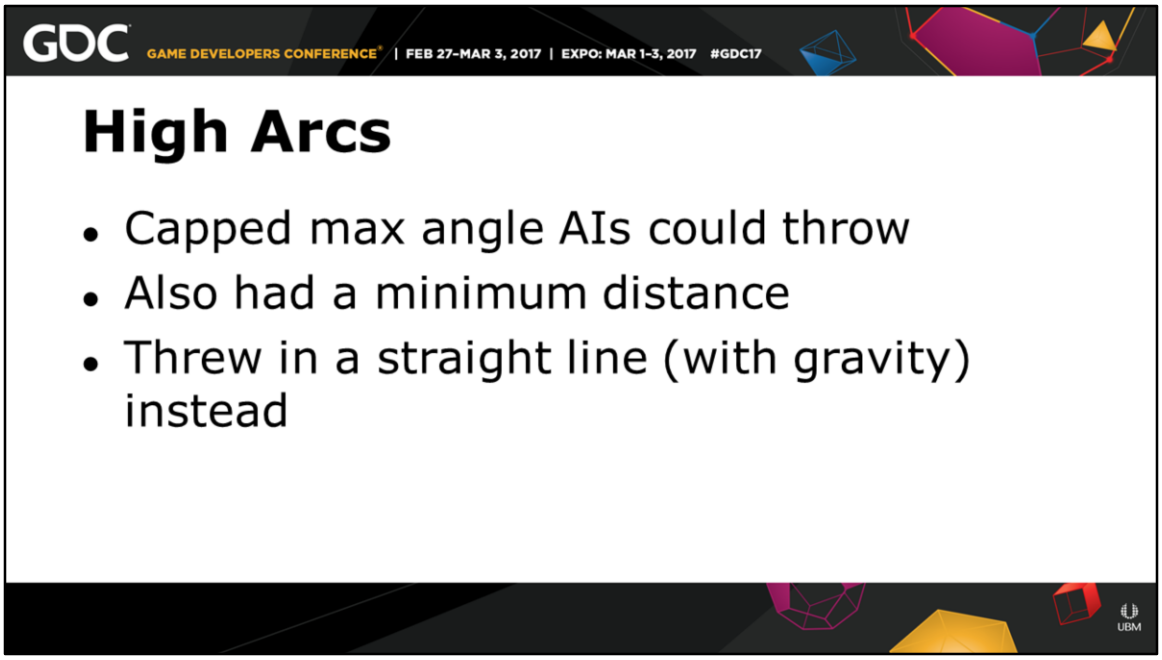
Debugging

Also pretty much the same. Here's what it looks like when we're debugging.

Can impact more than once due to hitting the ground and bouncing.

High Arcs

One issue we noticed was that when players got close to grenade-throwing AIs, you could get very high arcs on the grenades.



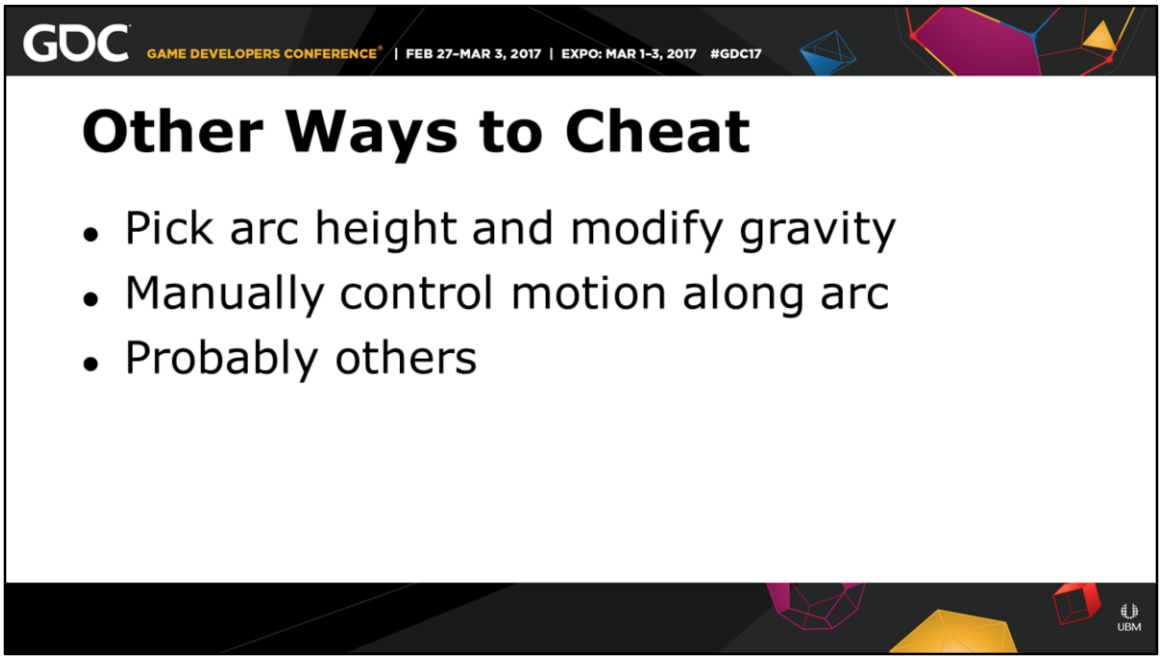
We wound up making two changes to solve this.

We used 70 degrees for our max angle in OMDU, and 2-3 meters for our min range, seemed to work well

We later wound up adding a minimum range to all of our ranged mobs.
When players moved too close they stopped attacking.
Still kept our other fixes in place, but this reduced how much they were needed.

Also made AI feel better—shooting you point blank looked kind of silly.

Also gave players another way to counter-play ranged AIs



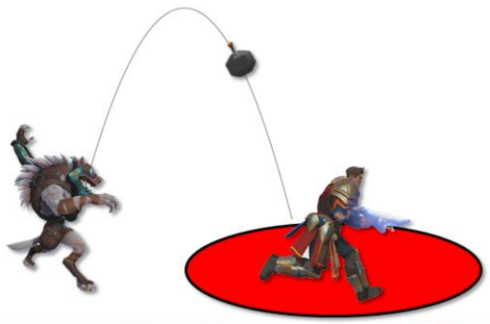
These will let you throw as far as you want, but with tradeoffs for how it looks and feels.

More information on these in the references.

Remember, if it looks good and plays well, no one cares if you cheat physics!

Missing Convincingly

- Random point in circle



Let’s talk about missing with ballistic projectiles.

In the earlier games in the franchise, we picked a random point in a circle around the player’s predicted position instead of the predicted position.

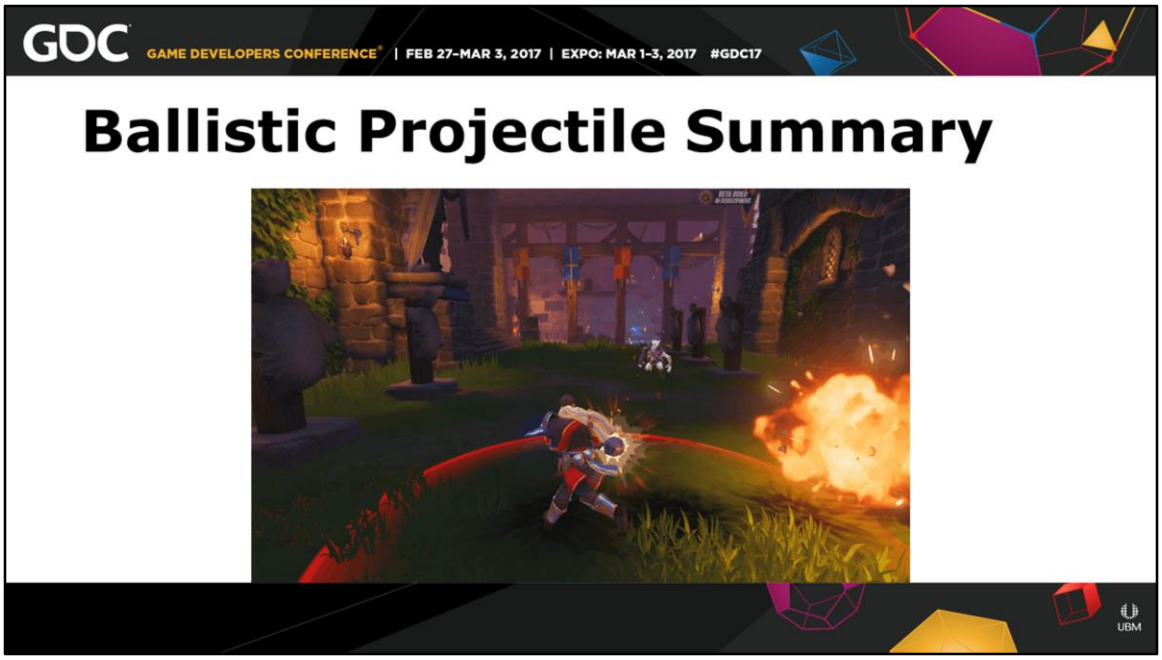
This worked fine, we shipped with it.

Missing Convincingly

- Alert players to grenades

We changed this because players would frequently not see the grenades if they were looking away from the unit that threw them.
Which meant they got blown up a lot and it wasn't fun, so we tried another approach in our latest game.

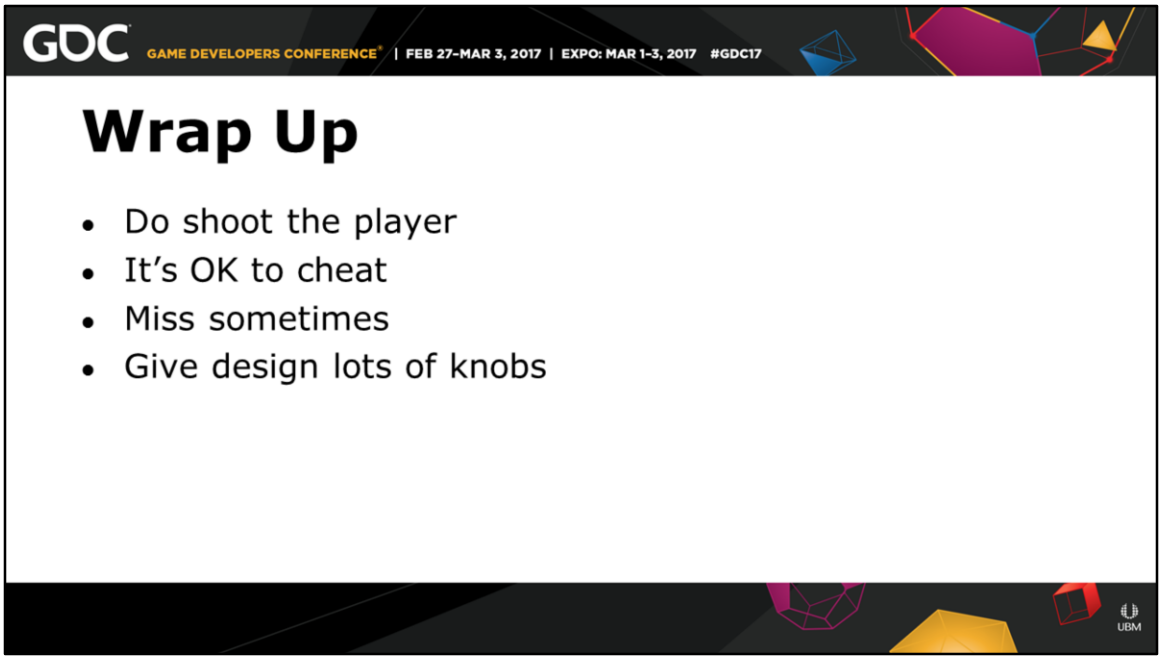
Now grenade throwing units don't miss.
Instead, we did a few things to add counterplay:
We added a fuse that started on impact and waited two seconds.
Beefed up impact sounds and fuse sound.
Added red decal on ground to tell players where they'd take damage.
Gave the player a better chance of running away



And here it is all together.

Ballistic projectiles
- Variable speed/arc
- Came up with approximate solution and solved with quadratic euqation
- May need to compensate for high arcs or other edge cases.

Missing
- Aim around player
- Make avoidable



Aim ahead of the player.
Fun is the priority.
Give players a chance to react to AI attacks, aimbots are not fun.
Make sure design can control the core parameters, then derive math/code/fun from there.

Questions?

@ShellSphinx

Thanks!

References

Schwarze, Jochen. "Cubic and Quartic Roots." *Graphics Gems*, edited by Andrew S. Glassner, Academic Press, 1990, Pages 404-407, 738.

http://mathforum.org/dr.math/faq/faq.cubic.equations.html

https://playtechs.blogspot.com/2007/04/aiming-at-moving-target.html

https://blog.forrestthewoods.com/solving-ballistic-trajectories-b0165523348c

http://luminaryapps.com/index.php?cID=143


GDC GAME DEVELOPERS CONFERENCE | FEB 27-MAR 3, 2017 | EXPO: MAR 1-3, 2017 #GDC17
Bonus Derivations / Refactoring
UBM

The fun part!

Linear Intersection (1/2)

$\boldsymbol{x}' = x_{0w} + \vec{v}_w t$ Warmage

$(\boldsymbol{x}' - x_{0p})^2 = (s_p t)^2$ Projectile

Multiply projectile equation through on left side

$\boldsymbol{x}'^2 - 2x'^{x_{0p}} + {x_{0p}}^2 = (s_p t)^2$

Now we substitute in for x′ from the Warmage's equation

$(x_{0w} + \vec{v}_w t)^2 - 2(x_{0w} + \vec{v}_w t)x_{0p} + {x_{0p}}^2 = (s_p t)^2$

See Slide 31

Want to get x'

We have one unknown: t, so we're going to solve for that.

Linear Intersection (2/2)

$$(x_{0w} + \vec{v}_w t)^2 - 2(x_{0w} + \vec{v}_w t)x_{0p} + {x_{0p}}^2 - \left(s_p t\right)^2 = 0$$

$${x_{0w}}^2 + 2x_{0w}\vec{v}_w t + (\vec{v}_w t)^2 - 2(x_{0w} + \vec{v}_w t)x_{0p} + {x_{0p}}^2 - \left(s_p t\right)^2 = 0$$

$${x_{0w}}^2 + 2x_{0w}\vec{v}_w t + {\vec{v}_w}^2 t^2 - 2x_{0p}x_{0w} - 2x_{0p}\vec{v}_w t + {x_{0p}}^2 - {s_p}^2 t^2 = 0$$

$$\left({\vec{v}_w}^2 - {s_p}^2\right)t^2 + (2(x_{0w} - x_{0p})\vec{v}_w)t + {x_{0w}}^2 - 2x_{0p}x_{0w} + {x_{0p}}^2 = 0$$

$$\left({\vec{v}_w}^2 - {s_p}^2\right)t^2 + (2(x_{0w} - x_{0p})\vec{v}_w)t + (x_{0w} - x_{0p})^2 = 0$$

Wait a minute…that looks familiar.

Line 1: get it all on the same side
Line 2: Multiply through the squared expression
Line 3: Get rid of everything in parenthesis
Line 4: Start regrouping
Line 5: Refactoring a bit

Vertical Motion—Solving for Angle (1/2)

$$x = s_p t \cos(\theta)$$

$$y = s_p t \sin(\theta) - \frac{1}{2} g t^2$$

$$t = \frac{x}{s_p \cos(\theta)}$$

$$y = s_p \frac{x}{s_p \cos(\theta)} \sin(\theta) - \frac{1}{2} g \left(\frac{x}{s_p \cos(\theta)} \right)^2$$

$$y = \frac{x \sin(\theta)}{\cos(\theta)} - \frac{g x^2}{2 {s_p}^2 \cos^2(\theta)}$$

$$y = x\, tan(\theta) - \frac{g x^2}{2 {s_p}^2} (1 + \tan^2(\theta))$$

See slide 65

Vertical Motion—Solving for Angle (2/2)

$$x\ tan(\theta) - \frac{gx^2}{2{s_p}^2}(1+\tan^2(\theta)) - y = 0$$

$$x\ tan(\theta) - \frac{gx^2}{2{s_p}^2} - \frac{gx^2}{2{s_p}^2}\tan^2(\theta) - y = 0$$

$$-\frac{gx^2}{2{s_p}^2}\tan^2(\theta) + x\ tan(\theta) - \frac{gx^2}{2{s_p}^2} - y = 0$$

Quadratic equation!

Vertical Motion—Solving for Angle Quadratic (1/3)

$$-\frac{gx^2}{2{s_p}^2}\tan^2(\theta) + x\,tan(\theta) - \frac{gx^2}{2{s_p}^2} - y = 0$$

$$tan(\theta) = \frac{-x \pm \sqrt{x^2 - 4\left(-\frac{gx^2}{2{s_p}^2}\right)\left(-\frac{gx^2}{2{s_p}^2} - y\right)}}{2\left(-\frac{gx^2}{2{s_p}^2}\right)}$$

$$tan(\theta) = \frac{s^2\left(-x \pm \sqrt{x^2 - 4\left(-\frac{gx^2}{2{s_p}^2}\right)\left(-\frac{gx^2}{2{s_p}^2} - y\right)}\right)}{-gx^2}$$

See slide 68

Vertical Motion—Solving for Angle Quadratic (2/3)

$$tan(\theta) = \frac{{s_p}^2\left(-x \pm \sqrt{x^2 - 4\left(-\frac{gx^2}{2{s_p}^2}\right)\left(-\frac{gx^2}{2{s_p}^2} - y\right)}\right)}{2gx^2}$$

$$tan(\theta) = \frac{-{s_p}^2 x \pm \sqrt{{s_p}^4 x^2 - 4{s_p}^4\left(\frac{g^2x^4}{4{s_p}^4} + \frac{gx^2}{2{s_p}^2}y\right)}}{-gx^2}$$

$$tan(\theta) = \frac{{s_p}^2 x \pm \sqrt{{s_p}^4 x^2 - \left(g^2x^4 + 2{s_p}^2 gx^2 y\right)}}{gx^2}$$

Vertical Motion—Solving for Angle Quadratic (3/3)

$$tan(\theta) = \frac{{s_p}^2 \pm \sqrt{{s_p}^4 - \left(g^2x^2 + 2{s_p}^2gy\right)}}{gx}$$

$$tan(\theta) = \frac{{s_p}^2 \pm \sqrt{{s_p}^4 - g\left(gx^2 + 2y{s_p}^2\right)}}{gx}$$

$$\theta = tan^{-1}\left(\frac{{s_p}^2 \pm \sqrt{{s_p}^4 - g\left(gx^2 + 2y{s_p}^2\right)}}{gx}\right)$$

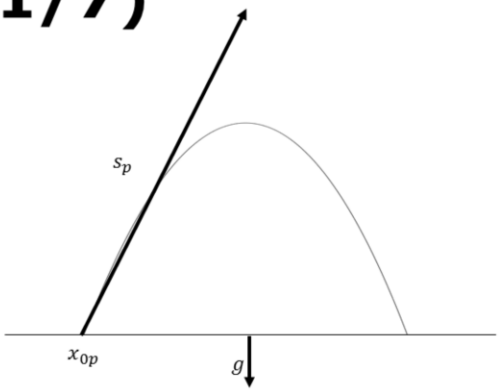
Throw Speed (1/7)

Projectile motion:

$$x' = -\frac{1}{2}gt^2 + s_p t + x_{0p}$$

Where:

- x' = Future position
- g = Gravitational acceleration
- t = Time
- s_p = Initial velocity
- x_{0p} = Initial position (assuming 0)



See slide 77

Note that we're just dropping x0p off after this point since it's 0

Throw Speed (2/7)

Initial projectile speed:

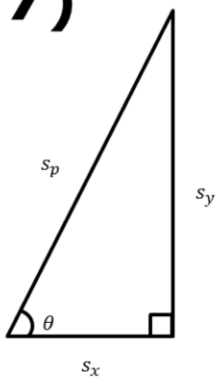
$s_x = s_p \cos(\theta)$

$s_y = s_p \sin(\theta)$

Where:

s_p = Throw speed

θ = Ideal projectile launch angle (45°)



We can take that diagram and make a triangle out of it.

Then can get x and y components of speed with trig identities (SOHCAHTOA)

Throw Speed (3/7)

Projectile motion:

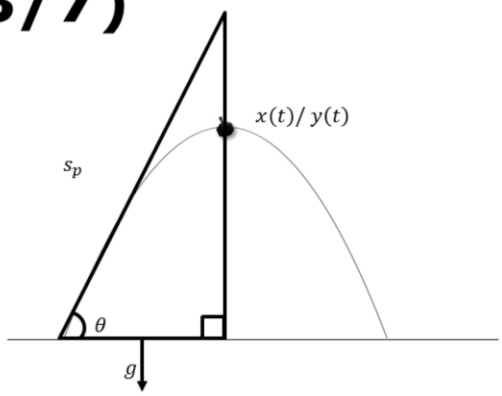
$$x' = -\frac{1}{2}gt^2 + s_p t$$

$$x(t) = -\frac{1}{2}g_x t^2 + s_p \cos(\theta)t = s_p t \cos(\theta)$$

$$y(t) = -\frac{1}{2}g_y t^2 + s_p \sin(\theta)t$$

Where:

$x(t)$ = Future x position
$y(t)$ = Future y position



Take projectile trajectory and get formulae for x and y future positions

No gravitational acceleration in x direction, so ½ gt^2 drops out for x(t)

Throw Speed (4/7)

$$x(t) = R_{max} = s_p t \cos(\theta)$$

$$y(t) = 0 = s_p t \sin(\theta) - \frac{1}{2} g t^2$$

...

$$t = \frac{2 s_p \sin(\theta)}{g}$$

Substitute in for t in $x(t)$ and refactor...

$$x(t) = R_{max} = \frac{{s_p}^2 \sin(2\theta)}{g}$$

Solve for when you land.

X position is Rmax (max range) when you land.
Y position is 0 when you launch and at the land.

Assuming flat ground—we're just using this to get rough numbers, so it's OK.

Solve y equation for t

Throw Speed (5/7)

$$y(t) = 0 = s_p t \sin(\theta) - \frac{1}{2} g t^2$$

$$s_p t \sin(\theta) - \frac{1}{2} g t^2 = 0$$

$$s_p t \sin(\theta) = \frac{g t^2}{2}$$

$$s_p t = \frac{g t^2}{2 \sin(\theta)}$$

$$t = \frac{g t^2}{2 s_p \sin(\theta)}$$

$$\frac{1}{t} = \frac{g}{2 s_p \sin(\theta)}$$

$$t = \frac{2 s_p \sin(\theta)}{g}$$

Solve y for t

Throw Speed (6/7)

$$t = \frac{2s_p \sin(\theta)}{g}$$

Substitute in for x...

$$x = s_p \frac{2s_p \sin(\theta)}{g} \cos(\theta)$$

$$x = \frac{{s_p}^2 2\sin(\theta)\cos(\theta)}{g}$$

$$x = R_{max} = \frac{{s_p}^2 \sin(2\theta)}{g}$$

R is our horizontal range, so it's really just another name for X.

We really want to solve for s, though.

Throw Speed (7/7)

$$R_{max} = \frac{{s_p}^2 \sin(2\theta)}{g}$$

$$\frac{R_{max}}{{s_p}^2} = \frac{\sin(2\theta)}{g}$$

$$\frac{{s_p}^2}{R_{max}} = \frac{g}{\sin(2\theta)}$$

$${s_p}^2 = \frac{R_{max} g}{\sin(2\theta)}$$

$$s_p = \sqrt{\frac{R_{max} g}{\sin(2\theta)}}$$

Max Range Cap (1/5)

- We had minimum ceiling height that art and design agreed on
- Capped our max throw range to not hit
- Starting with our projectile equation for y:

$$y(t) = h = -\frac{1}{2}gt^2 + s_p t \sin(\theta)$$

$$v_y(t) = 0 = -gt + s_p \sin(\theta)$$

...

$$R_{max} = \frac{4h}{tan(\theta)}$$

See slide 80

We want to solve when the projectile is at its peak, h

vy(t) is the derivative of y(t)

At the peak, y velocity is 0

Max Range Cap (2/5)

$$y(t) = s_p t \sin(\theta) - \frac{1}{2} g t^2$$

$$v_y(t) = s_p \sin(\theta) - gt$$

$$0 = s_p \sin(\theta) - gt$$

$$t = \frac{s_p \sin(\theta)}{g}$$

Substitute into y(t)

$$y(t) = h = s_p \frac{s_p \sin(\theta)}{g} \sin(\theta) - \frac{1}{2} g \left(\frac{s_p \sin(\theta)}{g} \right)^2$$

Vy is 0 at max height

Max Range Cap (3/5)

$$h = s_p \frac{s_p \sin(\theta)}{g} \sin(\theta) - \frac{1}{2} g \left(\frac{s_p \sin(\theta)}{g} \right)^2$$

$$h = \frac{{s_p}^2 \ sin^2(\theta)}{g} - \frac{1}{2} g \frac{{s_p}^2 \ sin^2(\theta)}{g^2}$$

$$h = \frac{{s_p}^2 \ sin^2(\theta)}{g} - \frac{{s_p}^2 \ sin^2(\theta)}{2g}$$

$$h = \frac{{s_p}^2 \ sin^2(\theta)}{2g}$$

Max Range Cap (4/5)

$$h = \frac{{s_p}^2 \ sin^2(\theta)}{2g}$$

$$R = \frac{{s_p}^2 \sin(2\theta)}{g}$$

$$\frac{h}{R} = \frac{{s_p}^2 \ sin^2(\theta)}{2g} \frac{g}{{s_p}^2 \sin(2\theta)}$$

$$\frac{h}{R} = \frac{{s_p}^2 \ sin^2(\theta)}{2} \frac{1}{{s_p}^2 \sin(2\theta)}$$

$$\frac{h}{R} = \frac{{s_p}^2 \ sin^2(\theta)}{2{s_p}^2 \sin(2\theta)}$$

Now that we have h, we actually want R, so divide both sides by R

Max Range Cap (5/5)

$$\frac{h}{R} = \frac{sin^2(\theta)}{2\sin(2\theta)}$$

$$\frac{h}{R} = \frac{sin^2(\theta)}{4\sin(\theta)\cos(\theta)}$$

$$h = \frac{R\ tan(\theta)}{4}$$

$$R = \frac{4h}{tan(\theta)}$$