# Iterating on a Dynamic Camera System

Phil Wilkins

Hello, good morning, and welcome.

My name is Phil, I’m a programmer at Sony Santa Monica, and I'm here to talk to you about the camera system we use in the God of War series of games.

Three years ago I gave a version of this talk that covered the core of the system I developed for the first two games. This is an iteration and an expansion on that talk, to include the new technology developed for three, and to cover some of the detail I left out before.

# Overview

| | |
|---|---|
| • Selection | Environment, Combat, Scripting, Filtering |
| • Blending | Blend Tree, Weights, Modes, Parameters |
| • Dynamics | Animated, Dynamic, Combat |
| • Targeting | Hero, Creatures, Damping, Weighting, Prioritisation |

So the camera system, and my talk, break down into four broad areas. Selection deals with choosing the cameras we're interested in. Blending turns them into a single camera, ready to pass to the rendering system. Dynamics describes how we calculate a cameras position and orientation. And finally, Targeting deals with what we're actually looking at.

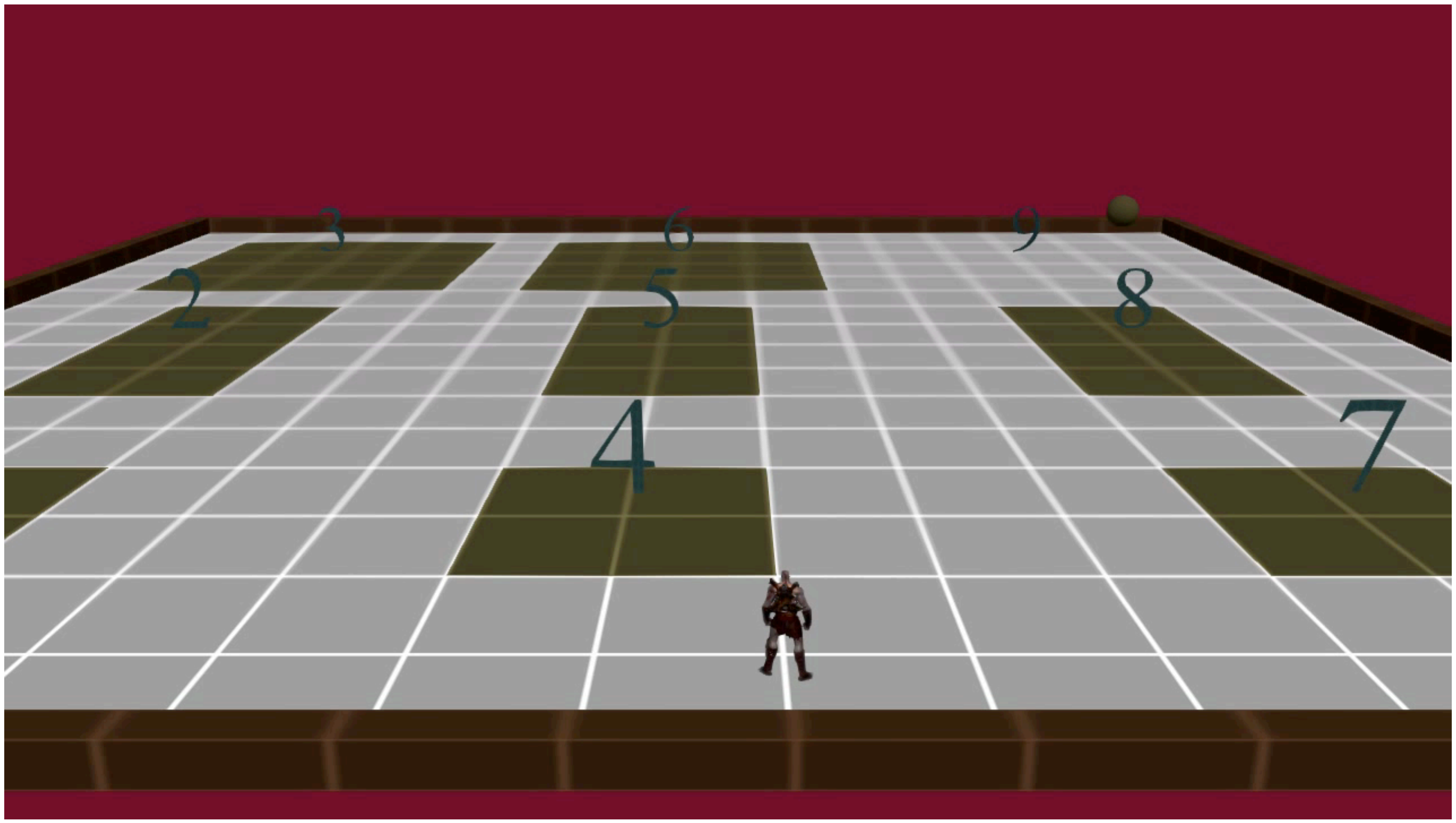
## Overview

| | |
|---|---|
| • Selection | Environment, Combat, Scripting, Filtering |
| • Blending | Blend Tree, Weights, Modes, Parameters |
| • Dynamics | Animated, Dynamic, Combat |
| • Targeting | Hero, Creatures, Damping, Weighting, Prioritisation |

OK, so at any one time we have a large number of potential cameras to choose from. A typical level might have around 50 cameras defined in it. Many of which are designed to be run simultaneously.

They're submitted for selection by one of three systems. Zones placed in the environment, actions triggered by the combat system, or entity scripting.

## Selection : Environment



The navigable areas of the environment, are marked up with zones. These zones each reference one or more cameras. When our hero enters a zone, the associate camera is selected.

## Selection : Combat

Cameras can also be submitted by the combat system. When Kratos grabs this grunt, he goes into a branching move sequence. Each of these moves can have an animated camera associated with it.

## Selection : Scripting

Finally we have scripted sequences.

The player triggers something in the environment, and the scripting system tells the camera system to start and stop a particular camera to highlight an event in game.

## Selection : Filtering

- Ignore submitted camera based upon:
  - Ignore list in active cameras
  - Player state based filtering:
    - On Ground, In Combat, Climbing, Flying, Swimming, Jumping, Falling, On Rope, Wall Hang, Wall Press
  - Prioritisation

Once we have a set of submitted cameras, we filter out the ones we don’t care about.
Each camera has a list of other cameras to either allow or ignore transition to.
We filter based on the players state. For example, a camera can be marked to only be valid if the player is falling.
This allows us to use different cameras for different transitions through the same area.
Finally from the remaining cameras, we ignore all those that have a priority level less than the highest one submitted.

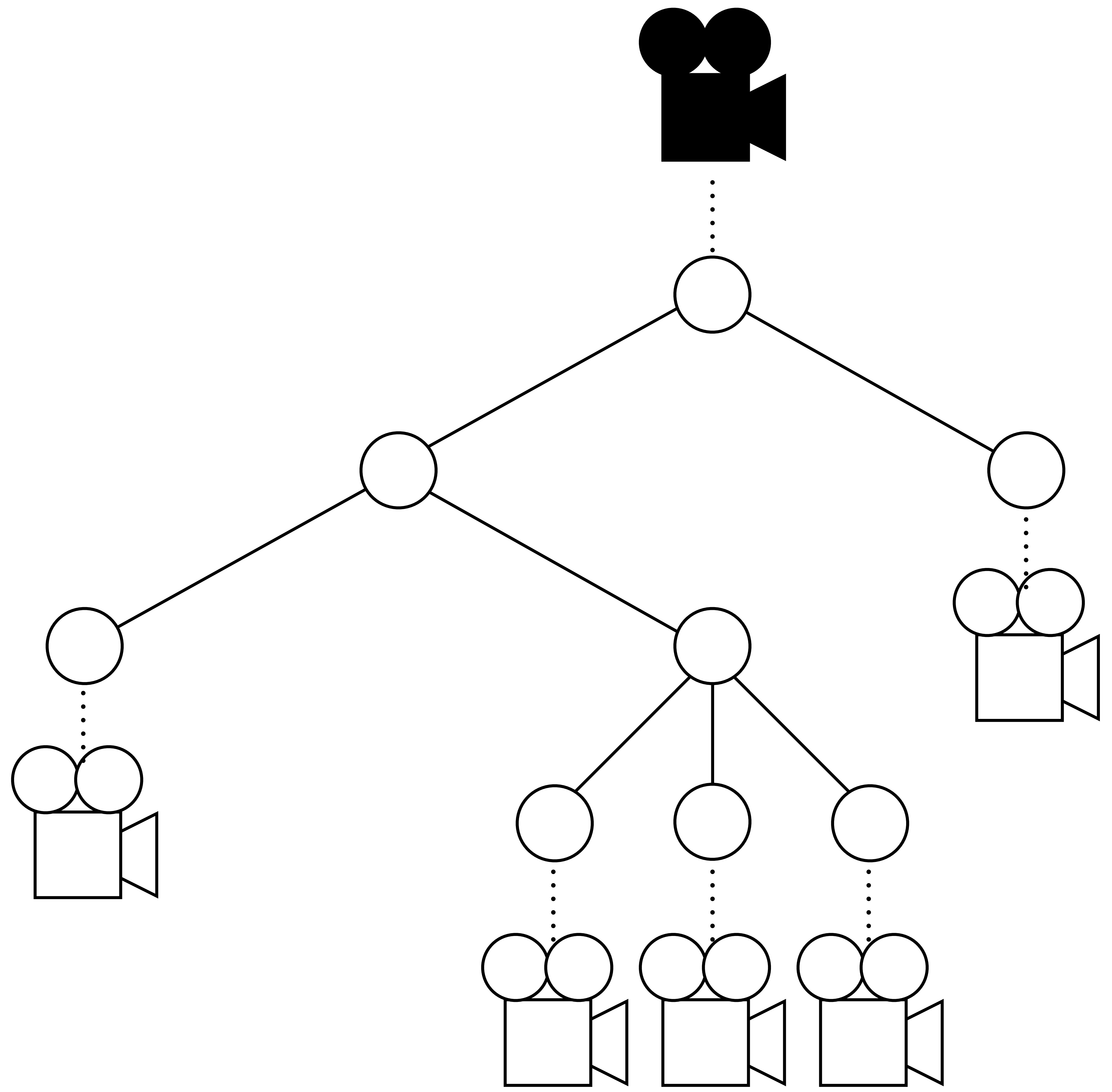
This gives us a list of cameras that should be active.

## Overview

| | |
|---|---|
| • Selection | Environment, Combat, Scripting, Filtering |
| • Blending | Blend Tree, Weights, Modes, Parameters |
| • Dynamics | Animated, Dynamic, Combat |
| • Targeting | Hero, Creatures, Damping, Weighting, Prioritisation |

Next we reconcile that with the blending system.

## Blending : Blend Tree

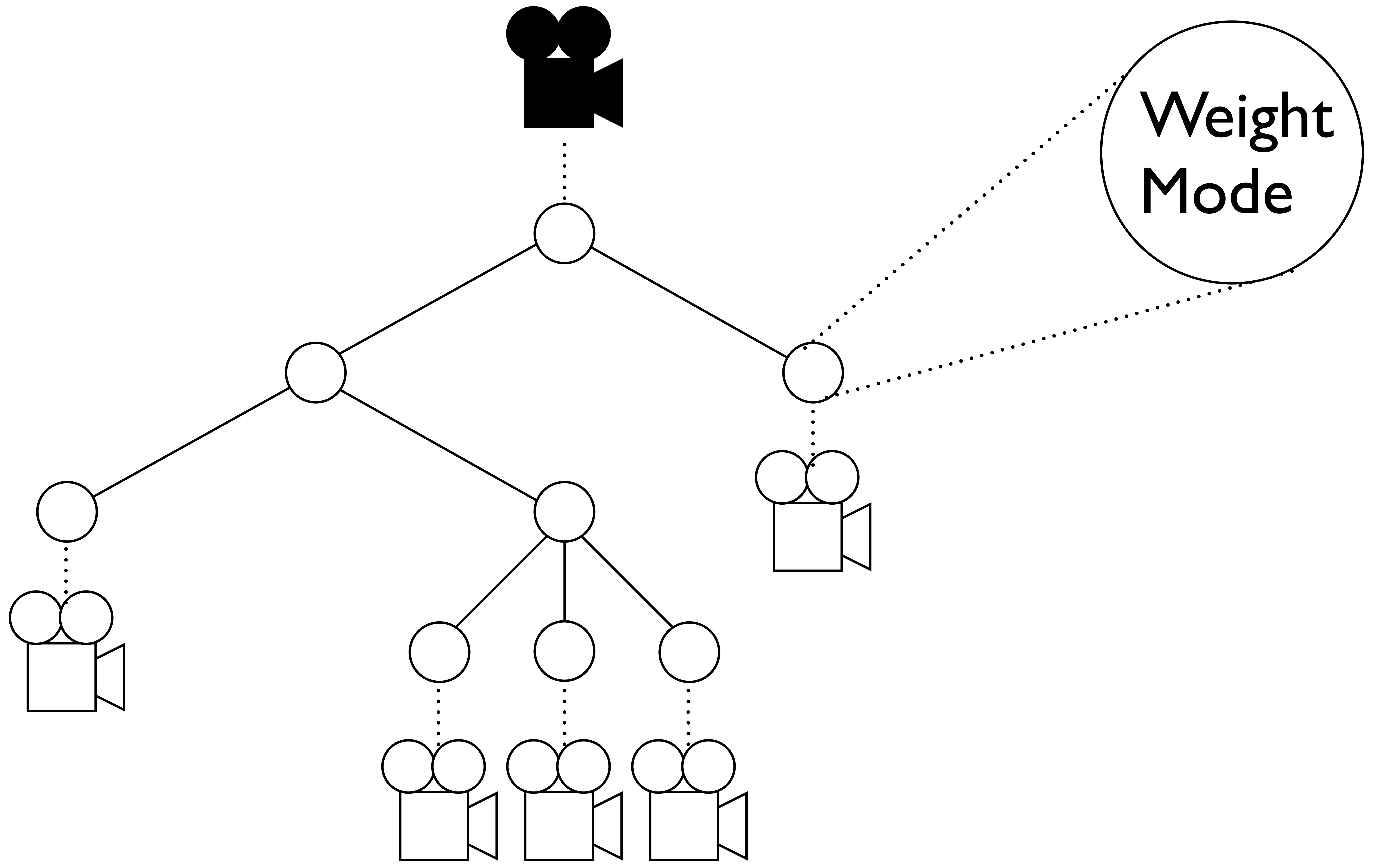


The system maintains a blend tree. The tree is constructed and maintained based upon a cameras priority, how it’s set to blend, and whether or not it was actually submitted this frame.

Old cameras are faded out, and new ones are inserted into the tree at the appropriate point.

*Each node has a weight, and mode that determine how it uses it’s inputs to calculate it’s output state.

# Blending : Blend Tree



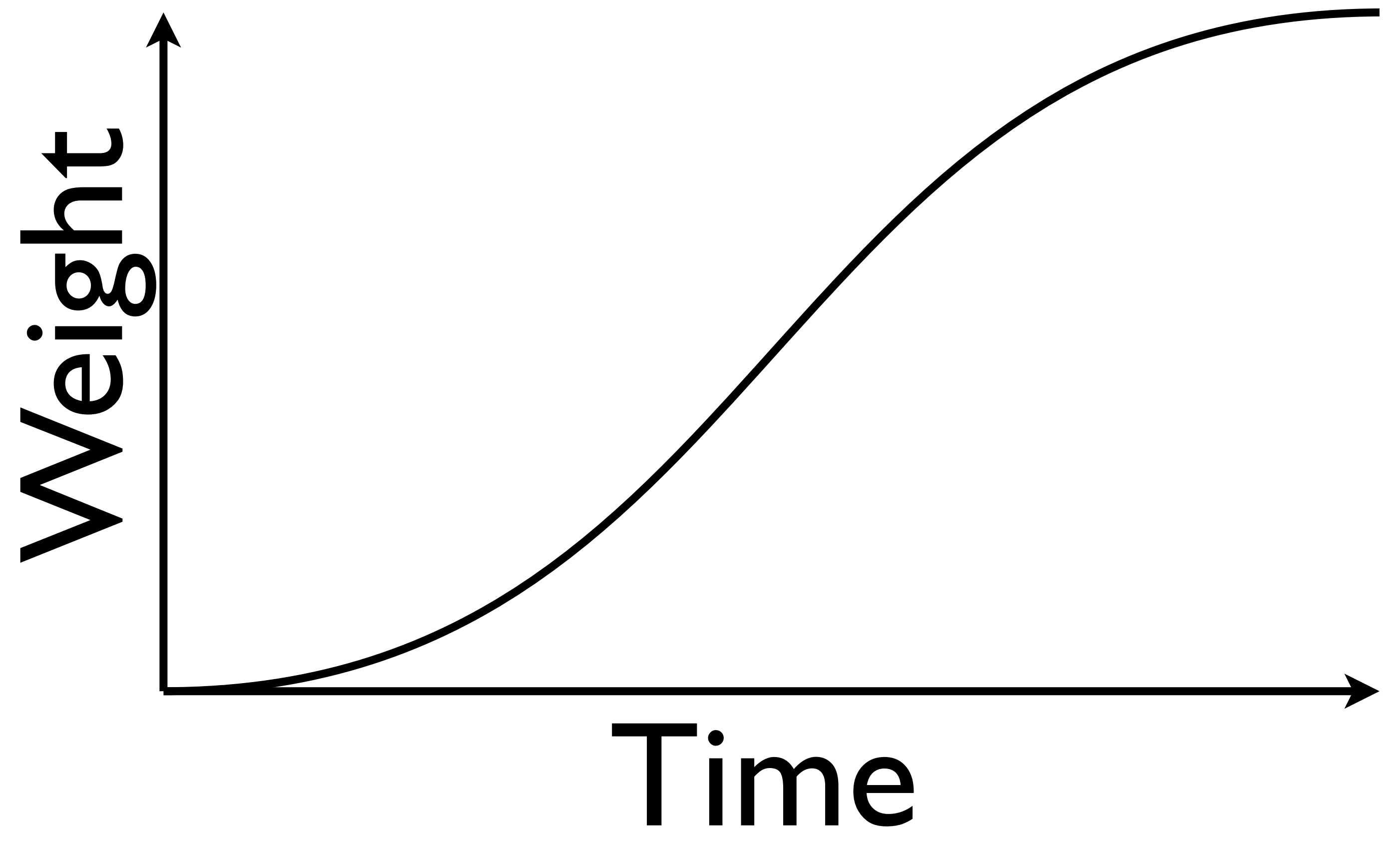
The system maintains a blend tree. The tree is constructed and maintained based upon a cameras priority, how it’s set to blend, and whether or not it was actually submitted this frame.

Old cameras are faded out, and new ones are inserted into the tree at the appropriate point.

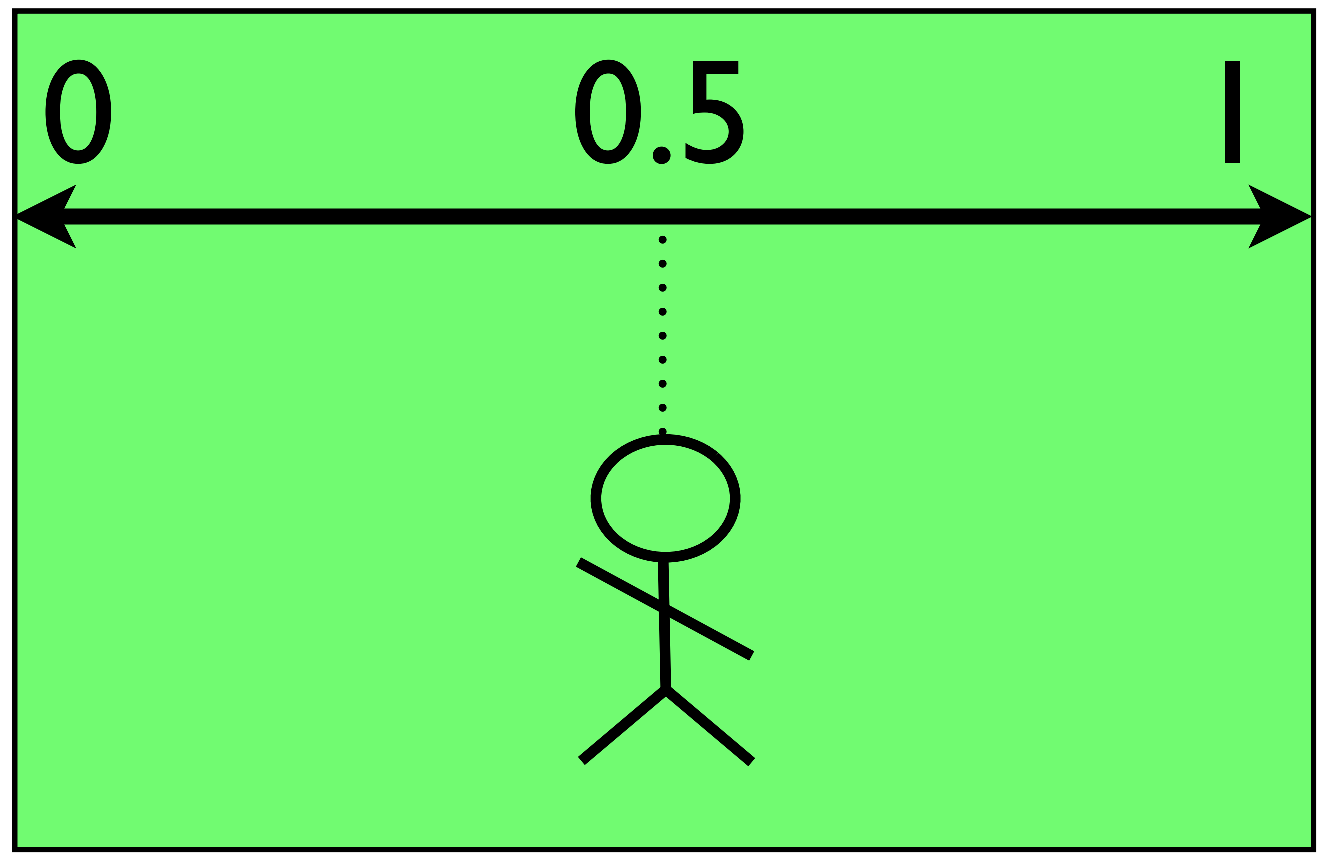
*Each node has a weight, and mode that determine how it uses it’s inputs to calculate it’s output state.

## Blending : Weights

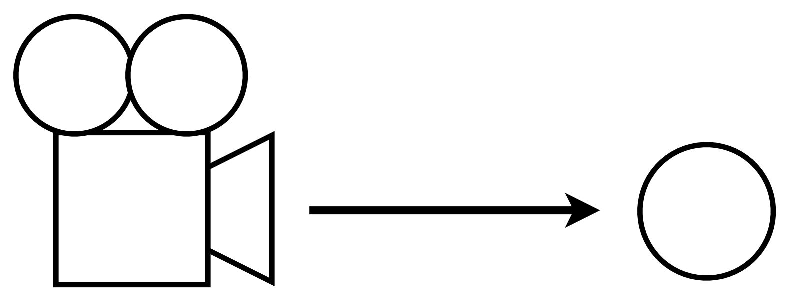
- Timer drives Hermite spline



- Position within zone



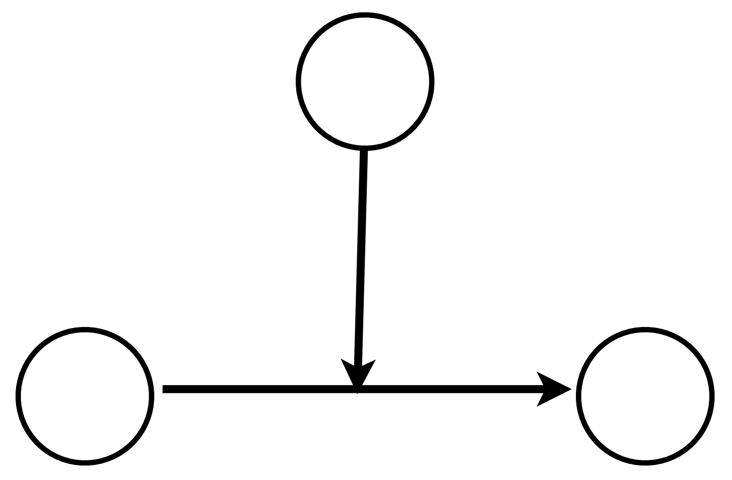
- Driven by camera logic



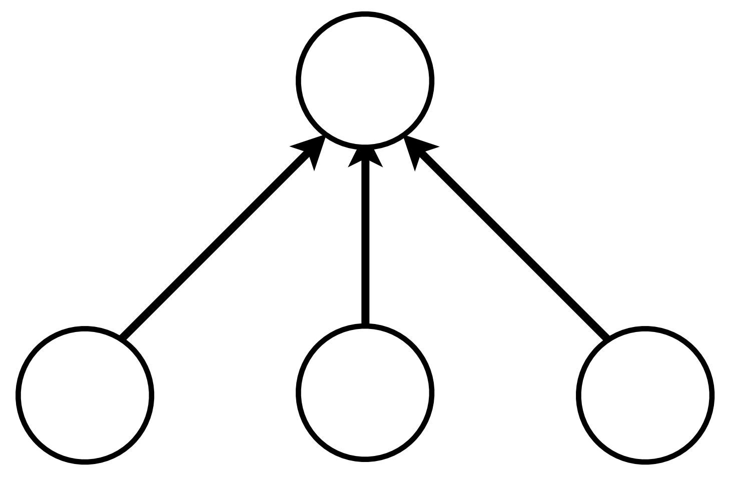
Blend weights can be generated from timers. These timers aren't used raw, but instead, drive a Hermite spline to add ease. Usually a timer runs forwards, as we generally replace old cameras by fading new cameras in over them, but sometimes, like at the end of a combat camera sequence, we run one backwards to reveal the underlying camera.
Weights can be driven from the heros position within a zone. These are calculated by the collision system. The most useful weight is proportion across the zone, parallel to a vector. This allows us to crossfade between two cameras as we move across the zone.
Or they can be calculated by the camera itself. We have an experimental camera that controls it's weight based upon which direction it’s moving on it's rail.

## Blending : Modes

- Crossfade
  - Uses weight to blend second node over first node
  - Timers and self weighted nodes



- Average
  - Sums and normalises weights of all children to determine contribution
  - Zone position weights



The mode determines how the children of that node are blended.

The crossfade mode blends the second node over the first node, as the weight increases from zero to one. This is used for timers, and self weighted nodes.

The average mode blends across all the input nodes using their individual weights to produce an average. We use this for zone position based blends.

## Blending : Operations

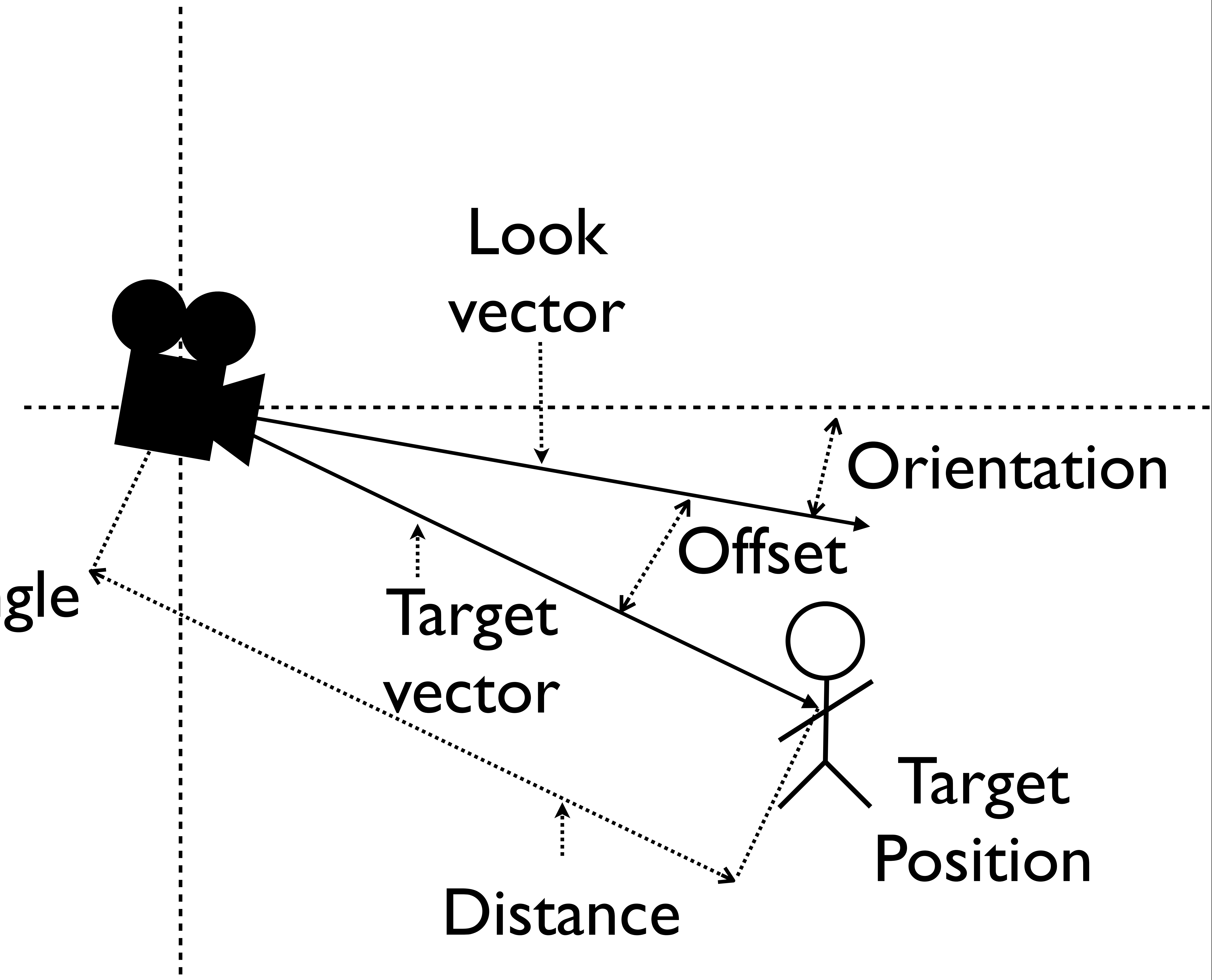
- Decomposed into a sequence of binary operations between cameras, with accumulators used to store intermediary values
- Cameras decomposed into aesthetically pleasing parameters that are individually interpolated
    - Camera orientation
    - Target position in world space
    - Target position in camera space

Once we have our blending tree, with all the weights determined, we reduce this to a list of blending operations between pairs of cameras.
Each camera is converted into a set of parameters which can then be interpolated to produce an aesthetically pleasing camera motion.
There are three major parameters, camera orientation, target position in world space, and target position in camera space.

Lets see that as a diagram.

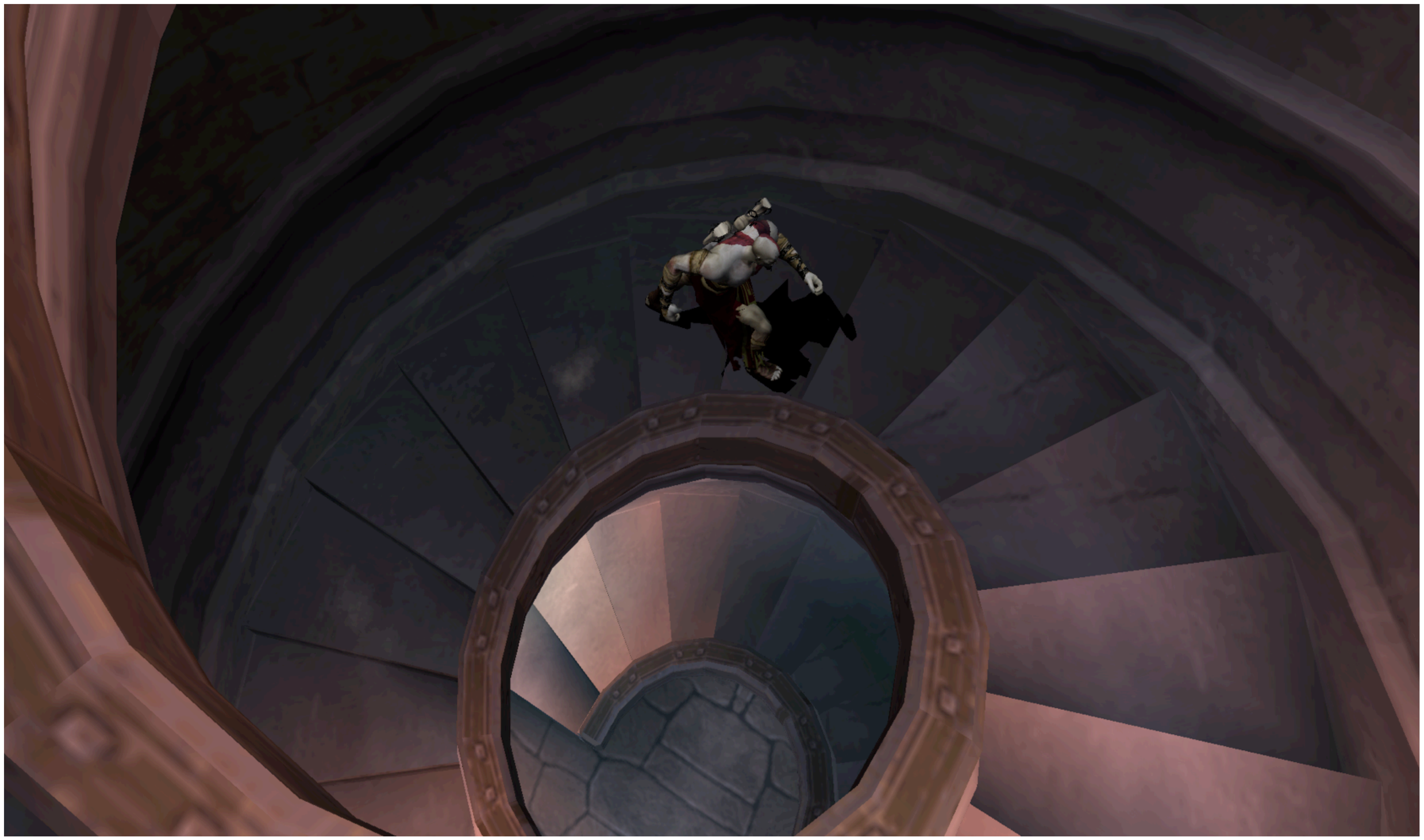
# Blending : Model Parameters

- Target position (X,Y,Z)
- Offset in spherical coordinates (Azimuth, Elevation, Distance)
- Orientation as an Euler angle (Yaw, Pitch, Roll)



The camera is expressed relative to the target, so the camera will tween around it, and not through it.
The position of the target in camera space is stored as the offset of the target vector from the look vector, by two angles, azimuth and elevation, or horizontal and vertical. With the distance from the camera to the target completing the spherical coordinate form.
Finally the orientation of the camera is stored in euler form. Eulers tween nicely for cameras, as unlike quaternions, they don't introduce any spurious roll, which looks ugly.

## Blending : Gimbal Lock



Of course Eulers suffer from gimbal lock, and do nasty things when you look straight up and down. Now we try and work around that by asking designers to avoid those shots, but like that's going to happen.

We solved this by keeping the camera ever so slightly tilted from vertical.

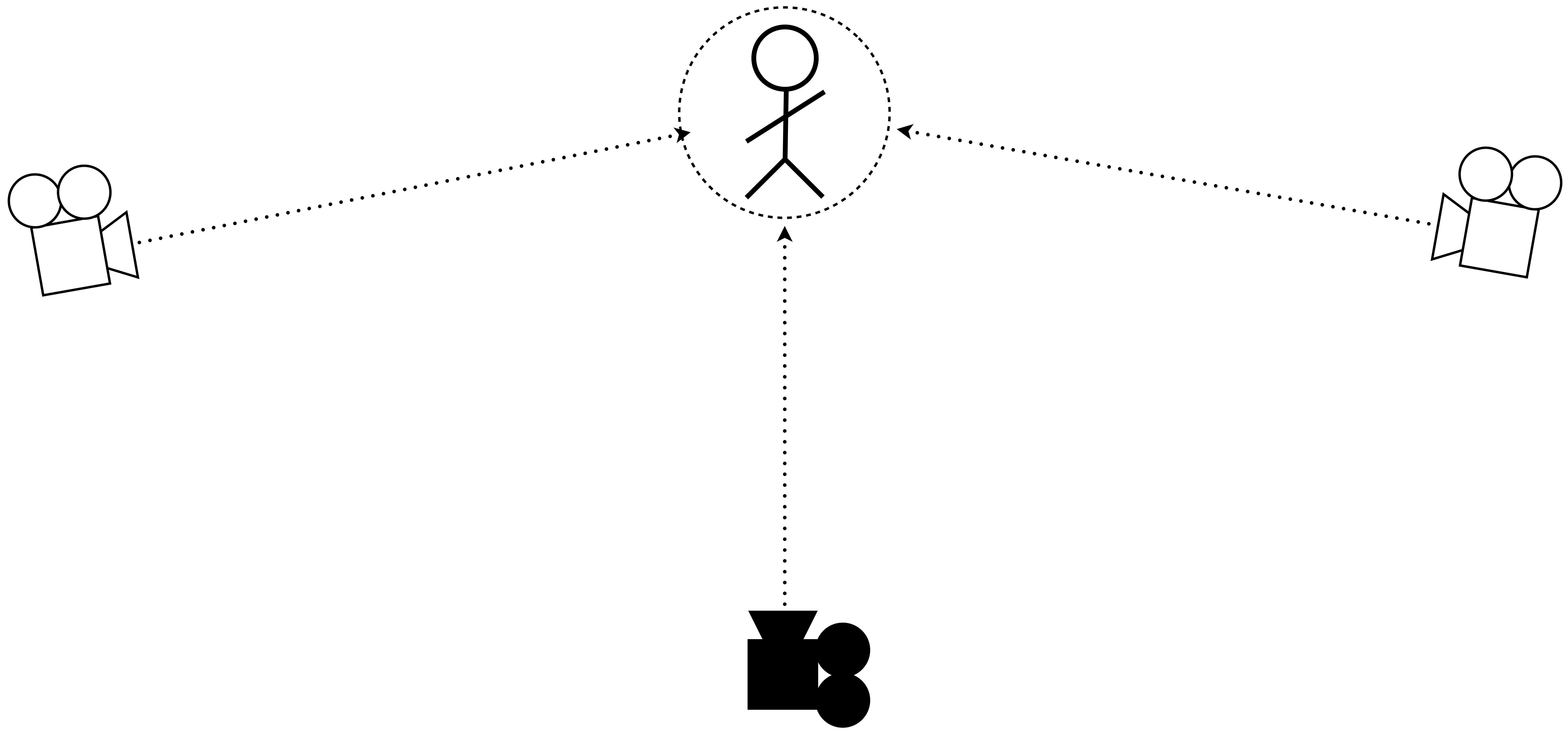
## Blending : Gimbal Lock

But then they throw stuff like this at you, and yes, he's really going straight up here, we didn't just put the set on it's side, although we probably should have done.

So we solve this, by adding an extra transform, which allows us to change the direction of the gimbal lock for the orientation euler. We don't change this very often, most of the time it's an identity rotation, but it’s there for these exceptional cases.

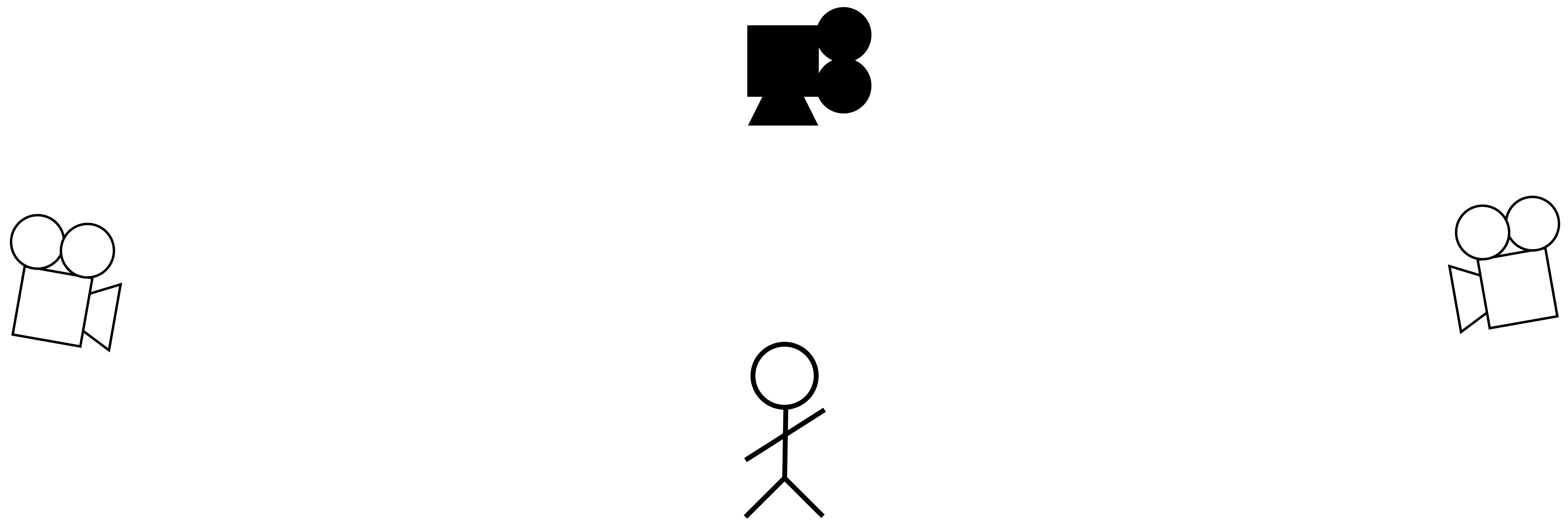
Now there was one exceptional case that we ended up cutting from the game...

## Blending : Yaw Blending



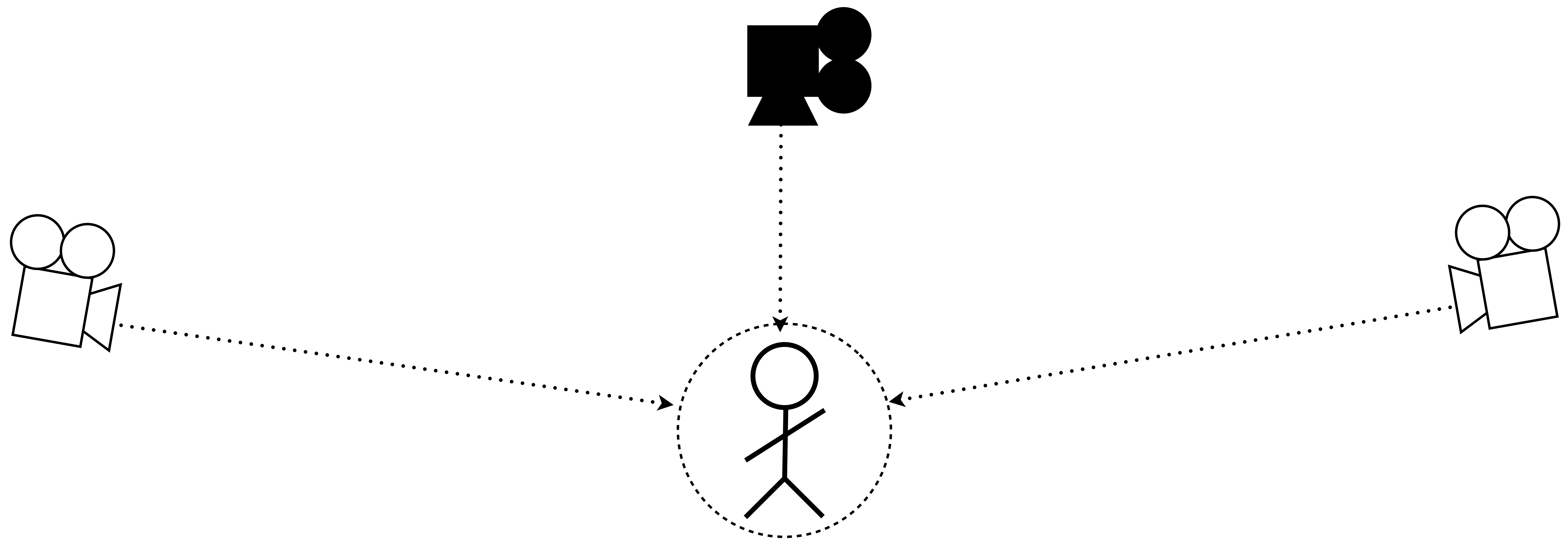
So there’s another problem.
Azimuth, Yaw, and Roll all wrap around. So for each of these parameters we have two ways that any two cameras can blend. The long way, and the short way.
It’s rarely an issue for Azimuth, since the target is almost invariably in front of the camera, and Roll is almost always 0.
For Yaw though, it can be a problem. This system is not static, everything’s in motion. Which means that which side is the shortest, can change, causing a massive pop as the camera flips from one side of the player to the other.
*
Now this pathological case is fairly easy to spot, but sometimes it crops up in an intermediary step in the middle of a long chain of blends.

# Blending : Yaw Blending



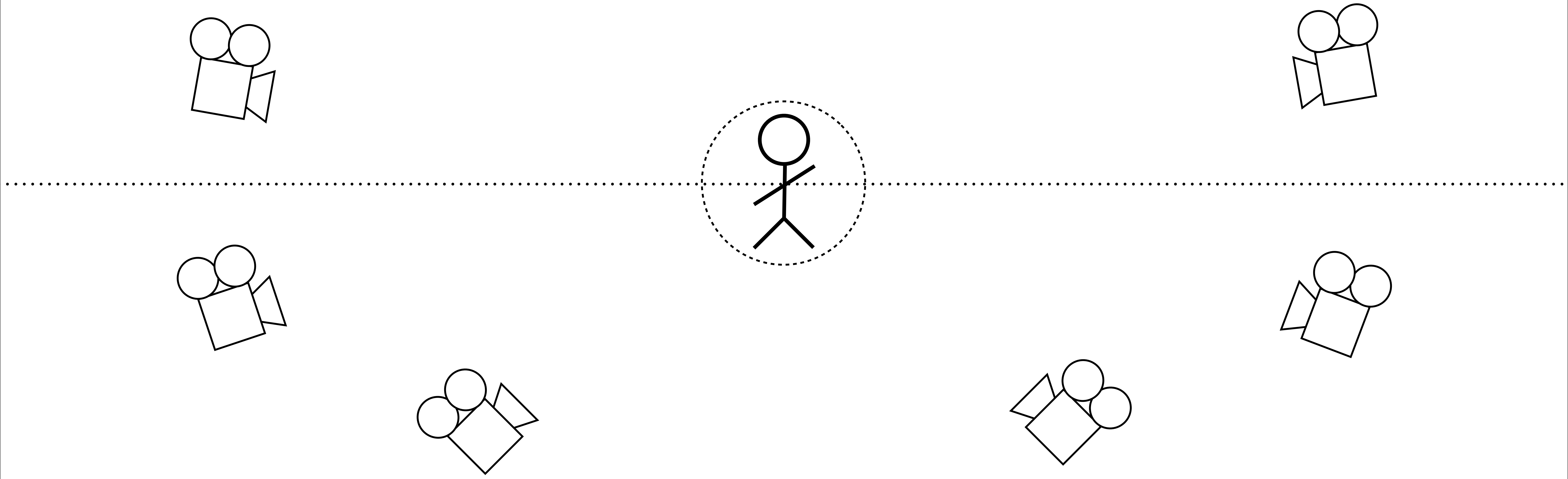
So there’s another problem.
Azimuth, Yaw, and Roll all wrap around. So for each of these parameters we have two ways that any two cameras can blend. The long way, and the short way.
It’s rarely an issue for Azimuth, since the target is almost invariably in front of the camera, and Roll is almost always 0.
For Yaw though, it can be a problem. This system is not static, everything’s in motion. Which means that which side is the shortest, can change, causing a massive pop as the camera flips from one side of the player to the other.
*
Now this pathological case is fairly easy to spot, but sometimes it crops up in an intermediary step in the middle of a long chain of blends.

# Blending : Yaw Blending



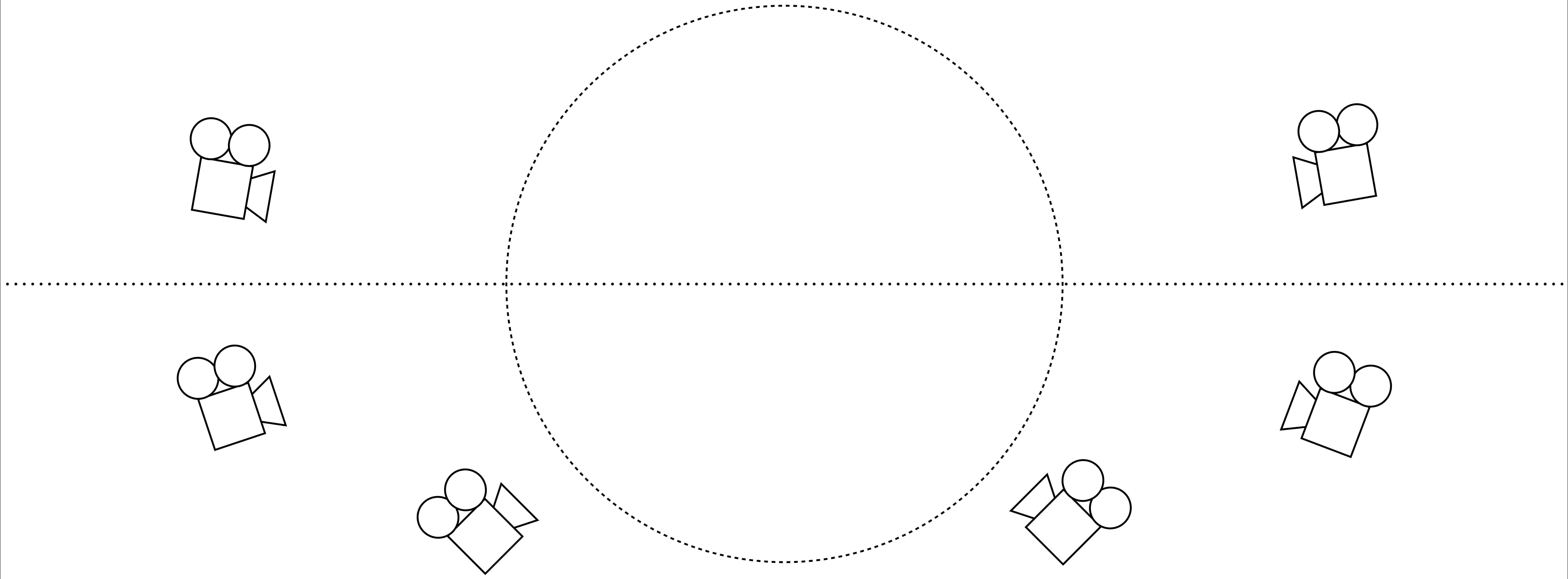
So there’s another problem.
Azimuth, Yaw, and Roll all wrap around. So for each of these parameters we have two ways that any two cameras can blend. The long way, and the short way.
It’s rarely an issue for Azimuth, since the target is almost invariably in front of the camera, and Roll is almost always 0.
For Yaw though, it can be a problem. This system is not static, everything’s in motion. Which means that which side is the shortest, can change, causing a massive pop as the camera flips from one side of the player to the other.
*
Now this pathological case is fairly easy to spot, but sometimes it crops up in an intermediary step in the middle of a long chain of blends.

## Blending : Yaw Blending



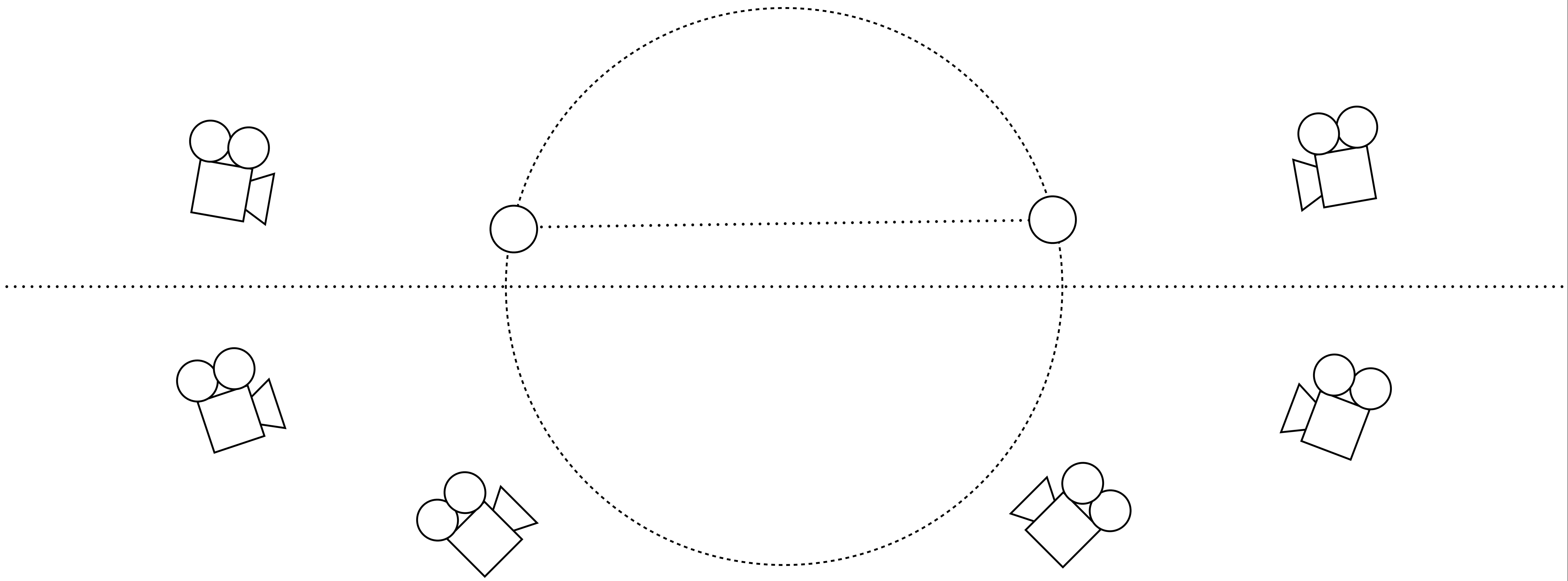
For example, the designers intent with a set of cameras like this, is probably to control the camera in a path below the player. But for the two cameras above the line, the shortest path is above the player.

# Blending : Yaw Blending



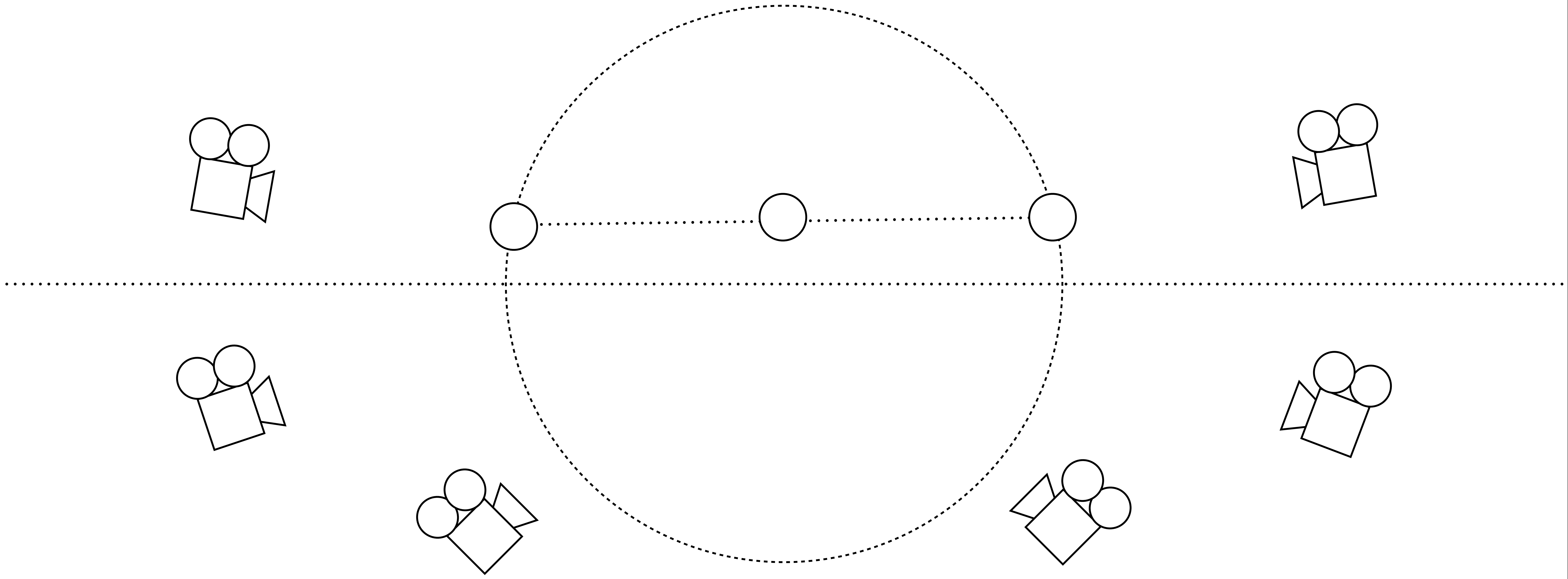
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



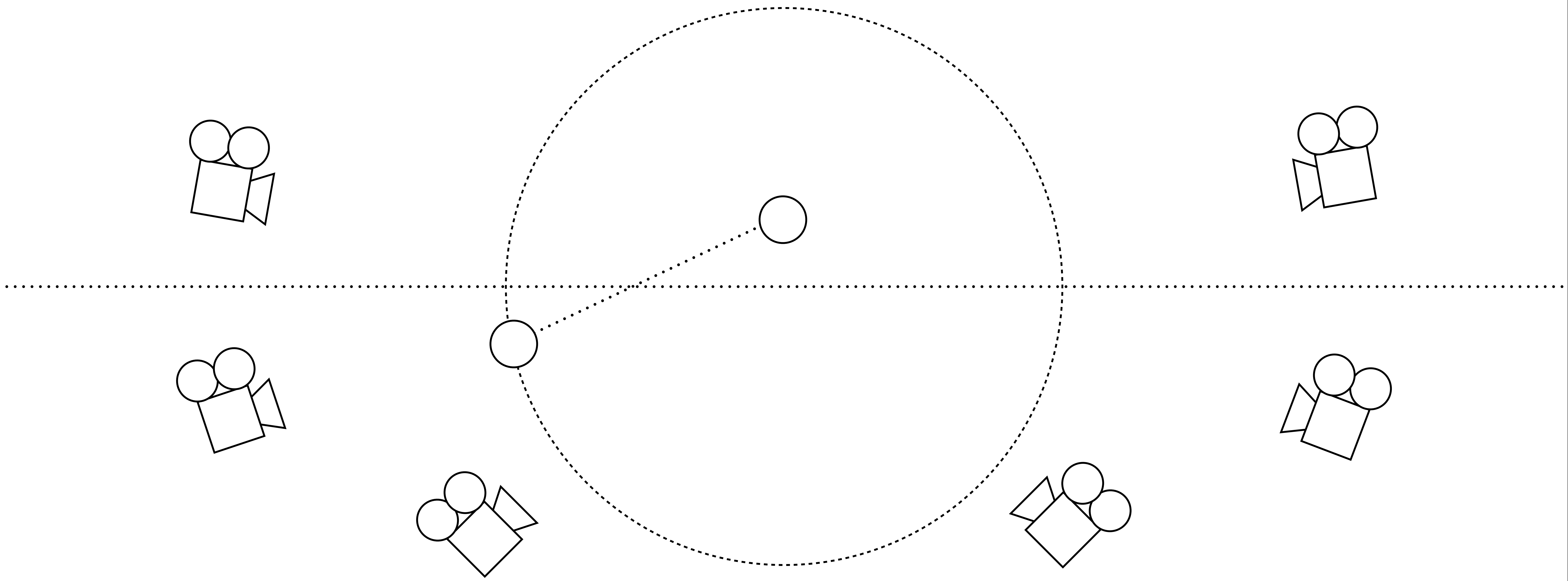
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



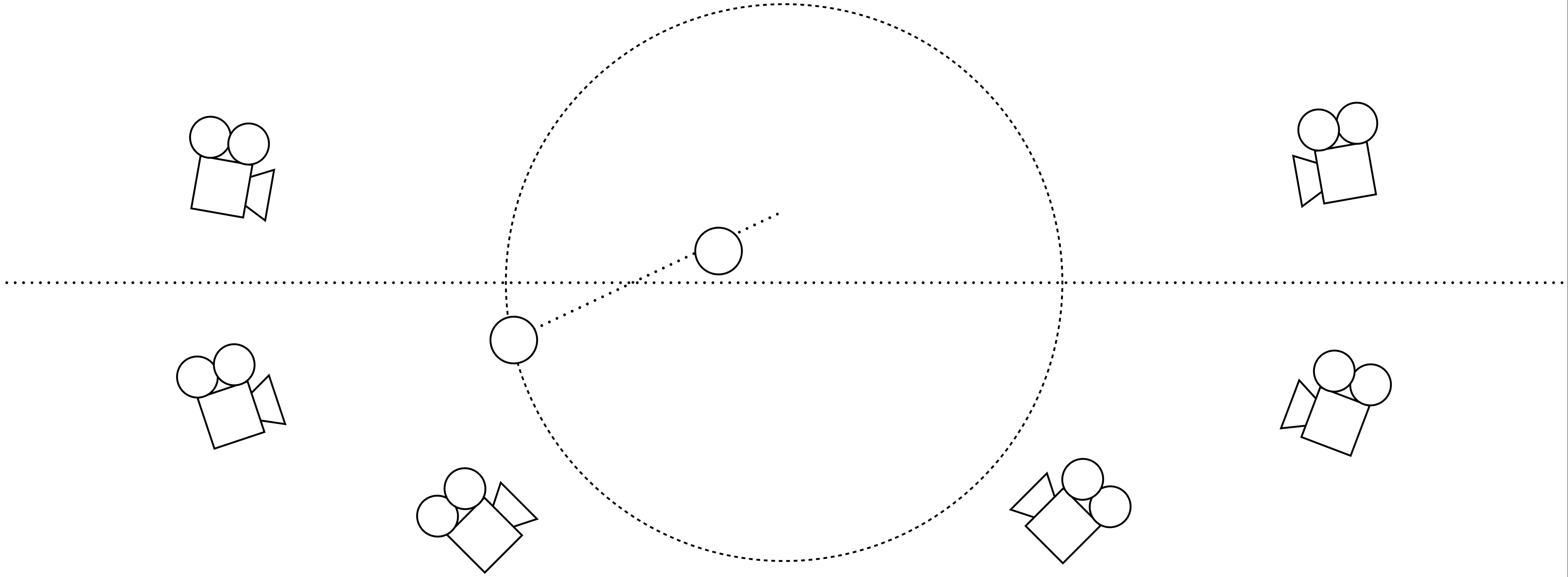
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

## Blending : Yaw Blending



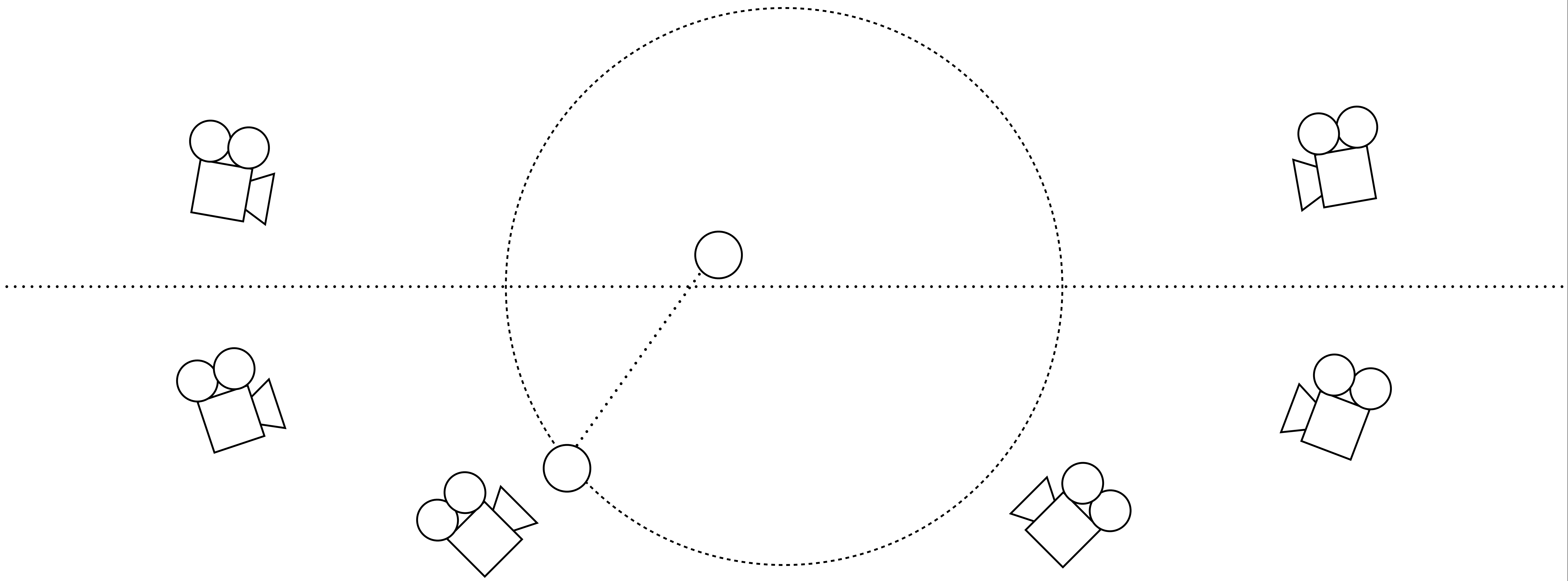
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



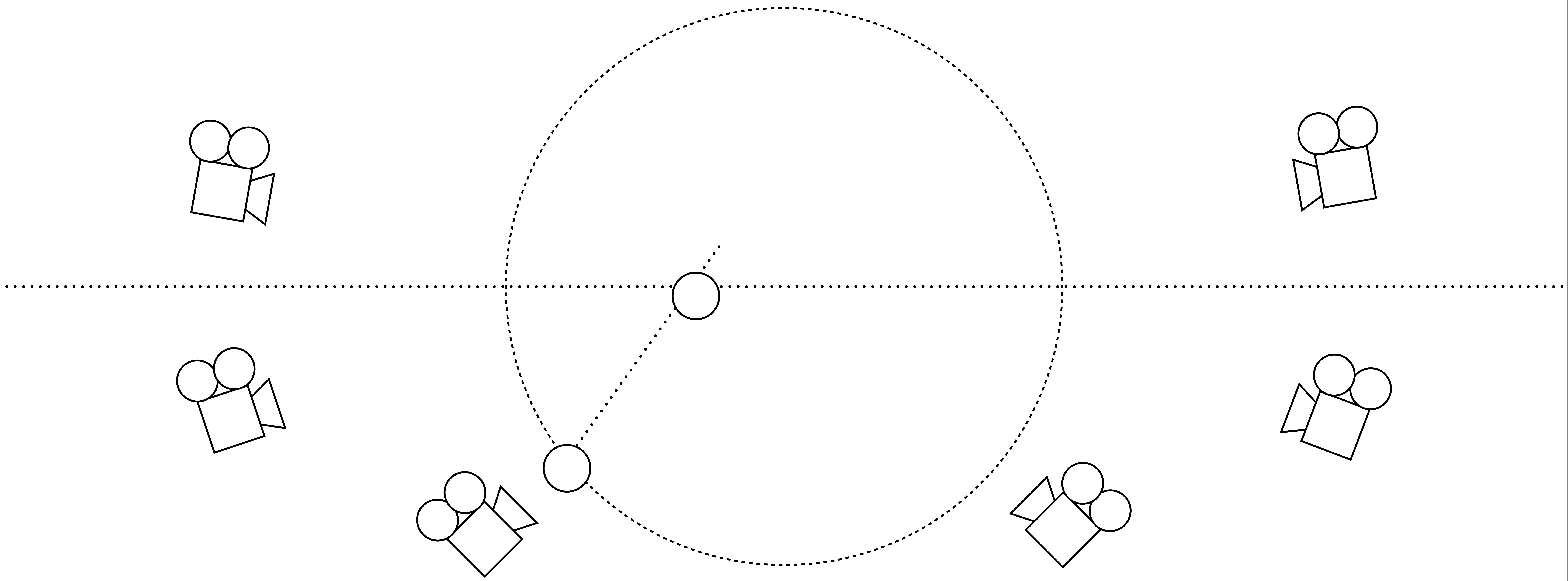
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



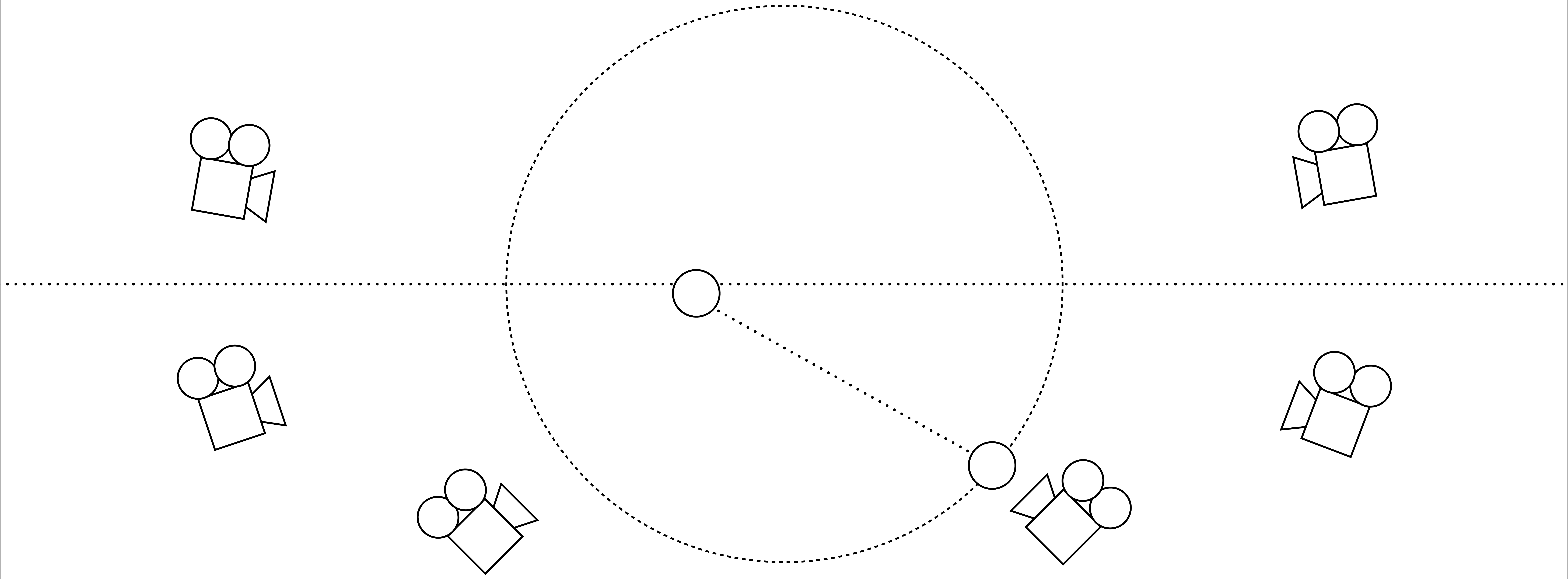
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



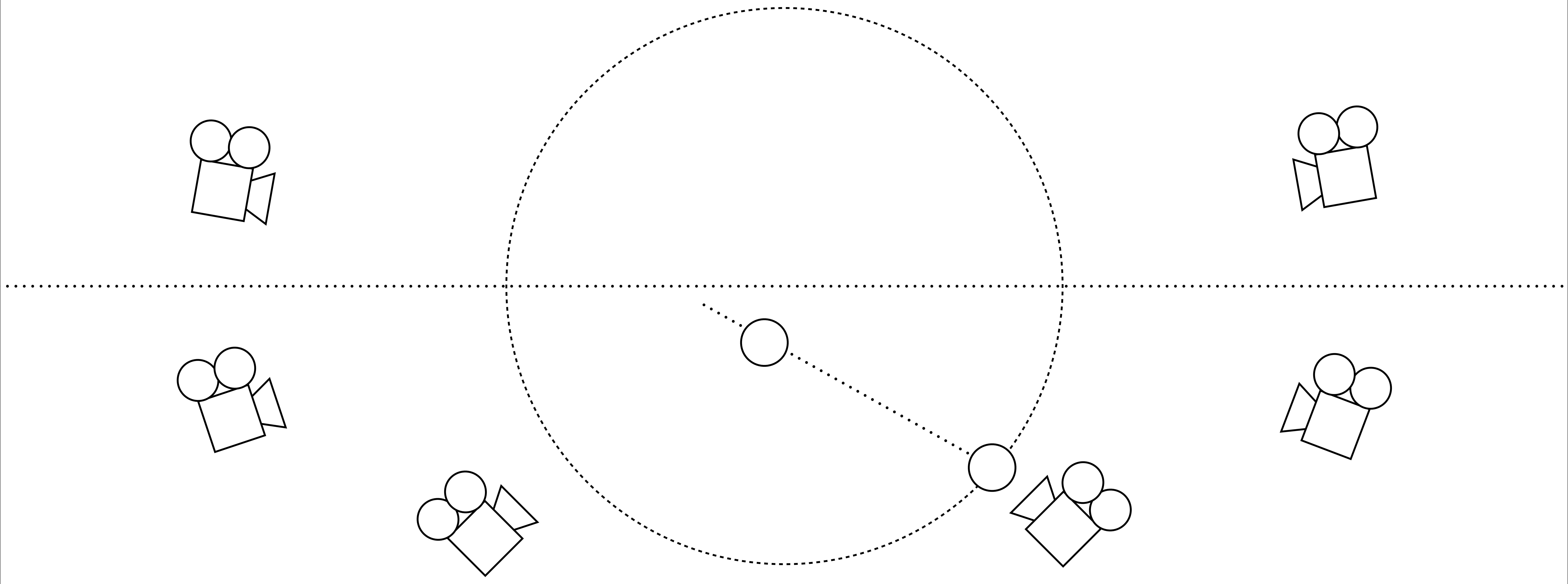
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

## Blending : Yaw Blending



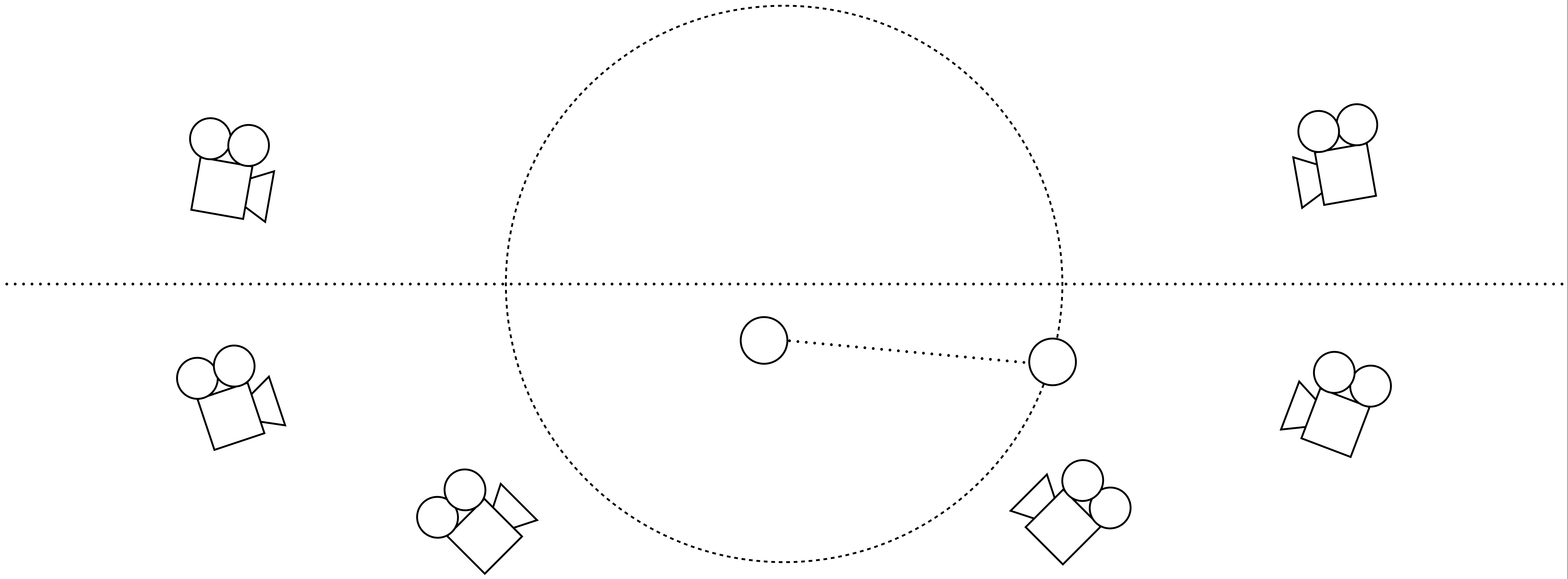
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



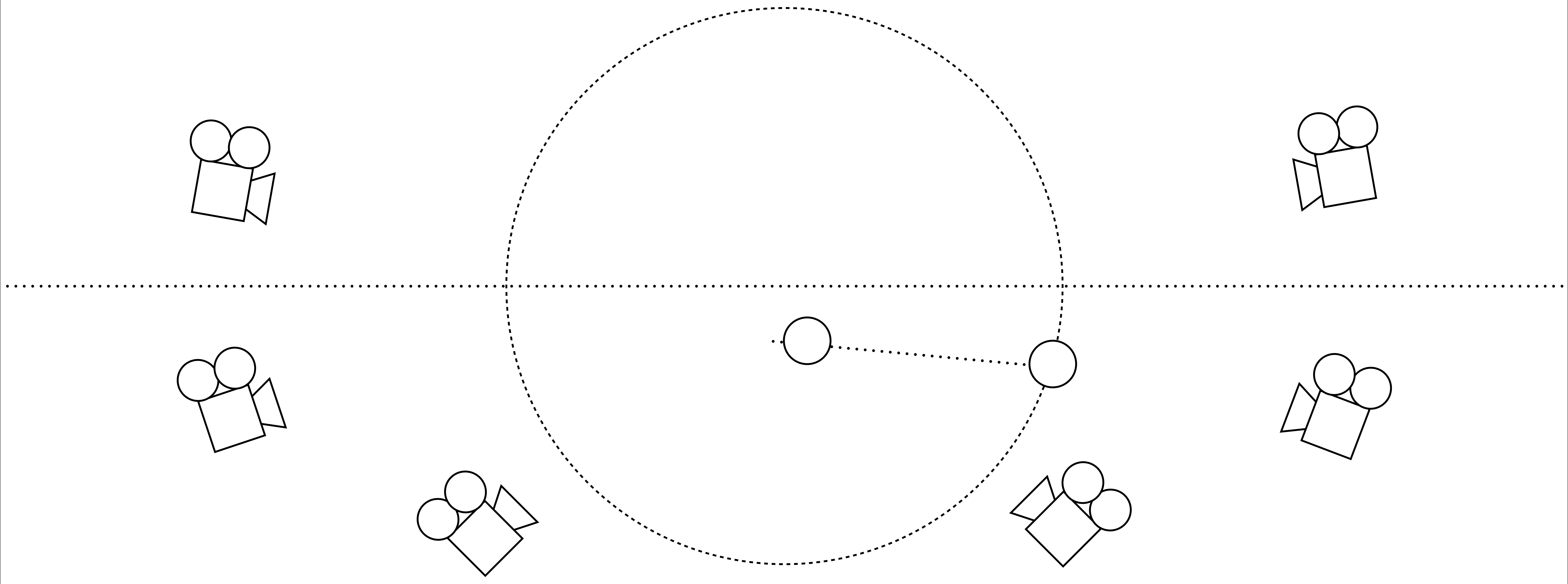
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



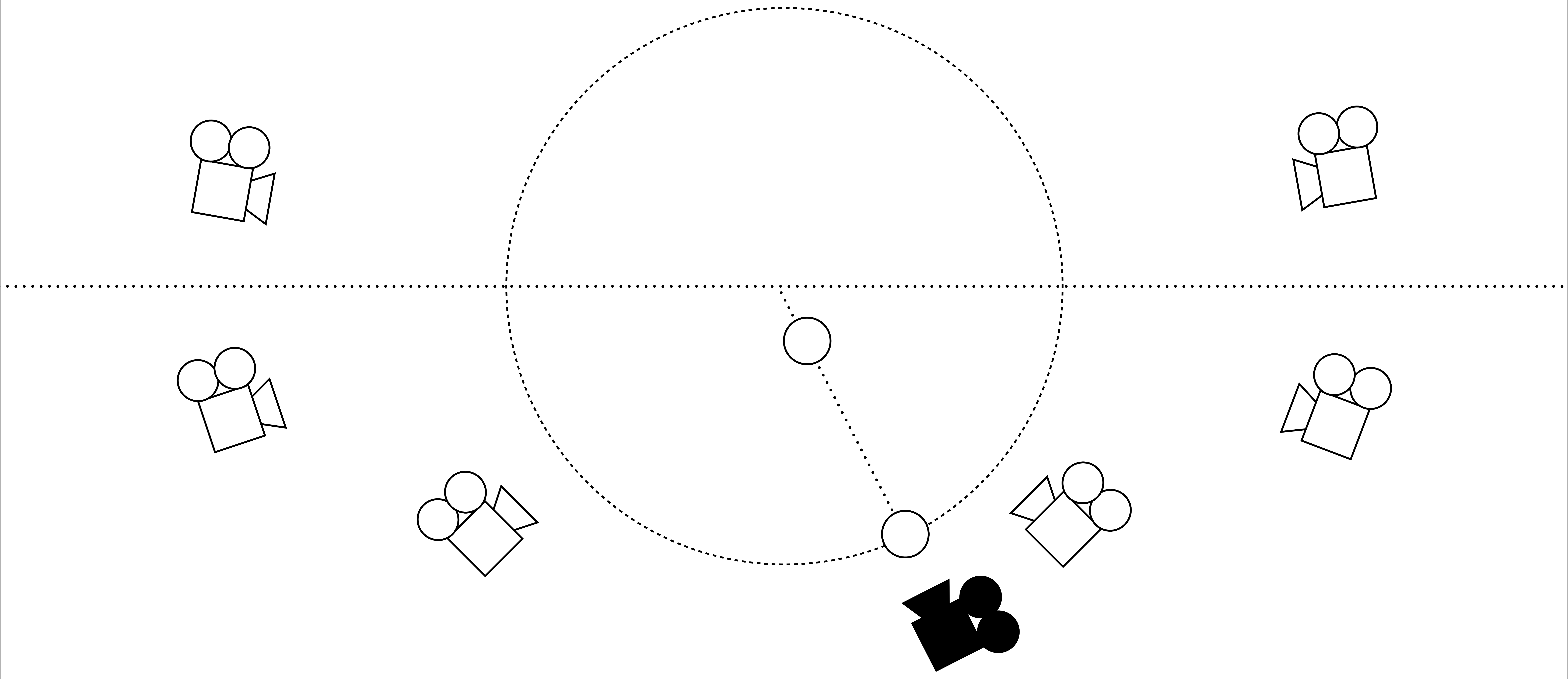
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



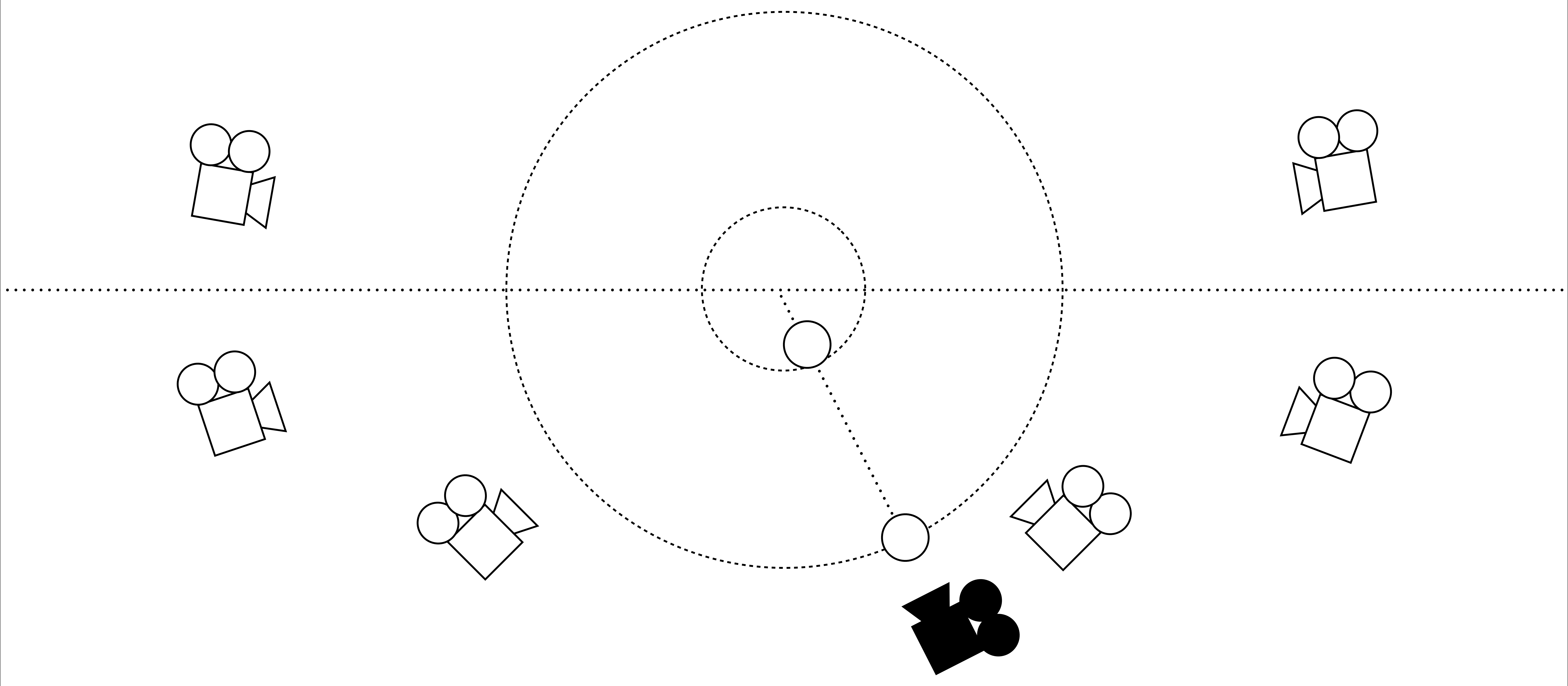
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



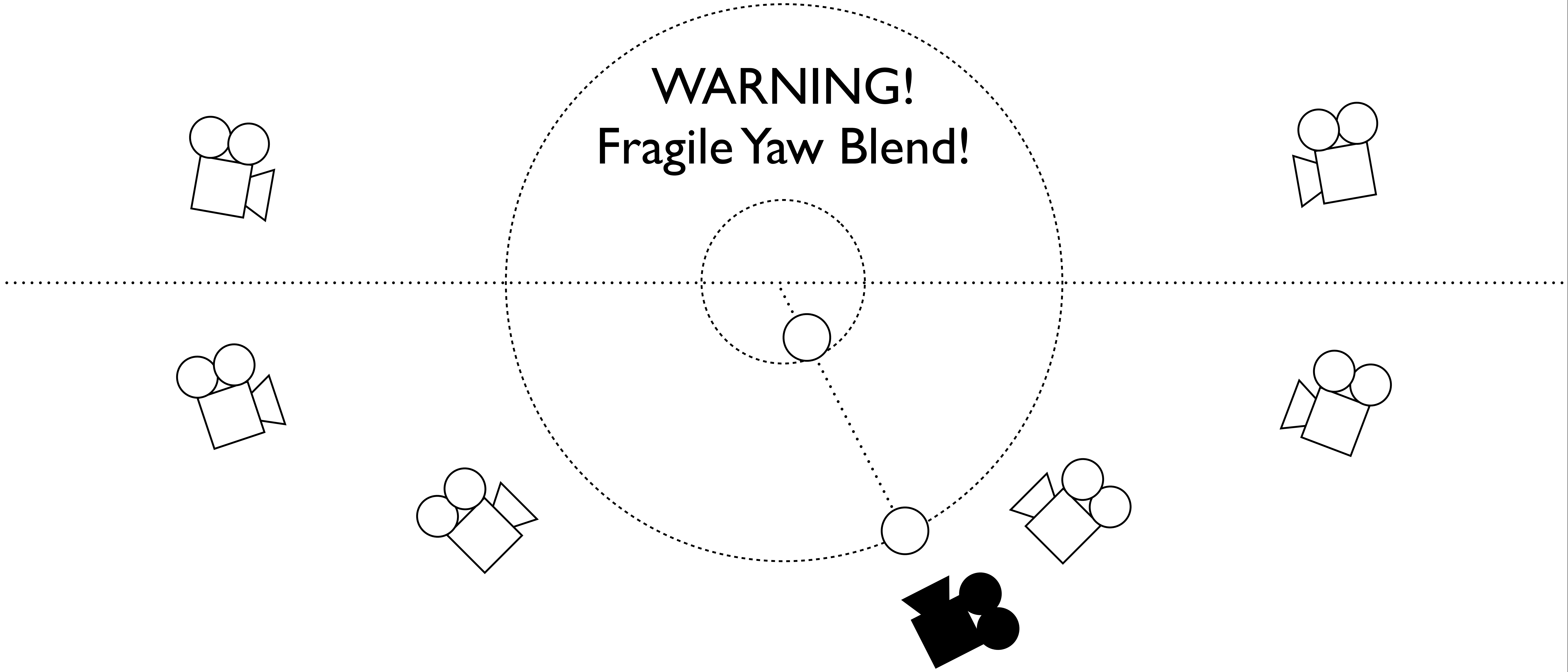
The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

## Blending : Yaw Blending



The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

# Blending : Yaw Blending



The solution we use for this problem, is to express yaw for each input camera, as a 2d unit vector. * blending that * for each blend in turn **** and converting it back into an angle when it comes time to generate the camera transform for the renderer. This doesn't entirely solve the problem, but it does reduce it, and it gives us a metric for fragile blends that we can use to warn the camera designer. * If the length of the vector gets too small, warning text appears on screen.
*

## Blending : Model Parameters

- Target world position as a vector
- Offset of target in camera space, in spherical coords
- Camera pitch and roll as angles
- Camera yaw as 2d vector
- Gimbal lock orientation as a quarternion
- Angle of View

So here’s our final list of blending parameters
OK, so that's enough to create camera position and orientation, which just leaves angle of view.

## Overview

| | |
|---|---|
| • Selection | Environment, Combat, Scripting, Filtering |
| • Blending | Blend Tree, Weights, Modes, Parameters |
| • Dynamics | Animated, Dynamic, Combat |
| • Targeting | Hero, Creatures, Damping, Weighting, Prioritisation |

So,how do we actually generate a camera in the first place.

## Dynamics

- Animated Camera
- Dynamic Camera
- Combat Camera

Well there are three basic cameras, that approximately match the three different submission methods.
The Animated camera is used mostly for cinematic sequences, and generally triggered by the scripting system.
The Dynamic camera deals with environmental situations, and is generally zoned.
And the Combat camera when we want to get a close up shot of a fight sequence, is submitted by the combat system.

OK let's start off with a simple one.

## Animated Camera

- Converts hand animated camera directly into blending parameters
- Cinematic sequences of all lengths
- Optional static target for blending purposes, uses aim point from Maya

The animated camera, takes a camera animated by, err, an animator, and maps it directly to the blending parameters. We typically use this for cinematic sequences.

To control blending you can set it to use the aim point from Maya as it's target point

## Animated Camera : Drive Rail

- Drive rail is a NURBS curve
- Calculate parameter value for nearest point on curve to hero
- Convert to animation time
- Apply new time to animated camera
- Many attributes of dynamic camera can also be animated
- Change dynamic constraints based on hero position

It became more powerfull in GoW3 when we added the ability to drive animation based on the nearest point on a NURBS curve we call the Drive Rail.

This is pretty simple, basically we map the length of the animation to the length of the rail. Calculate the nearest point on the rail to the hero each frame, and set the animation time based on that mapping.

The bonus is that, you can animate most of the interesting parameters of the Dynamic camera. So you can use this as another way of varying dynamics, based on the players position in the world.

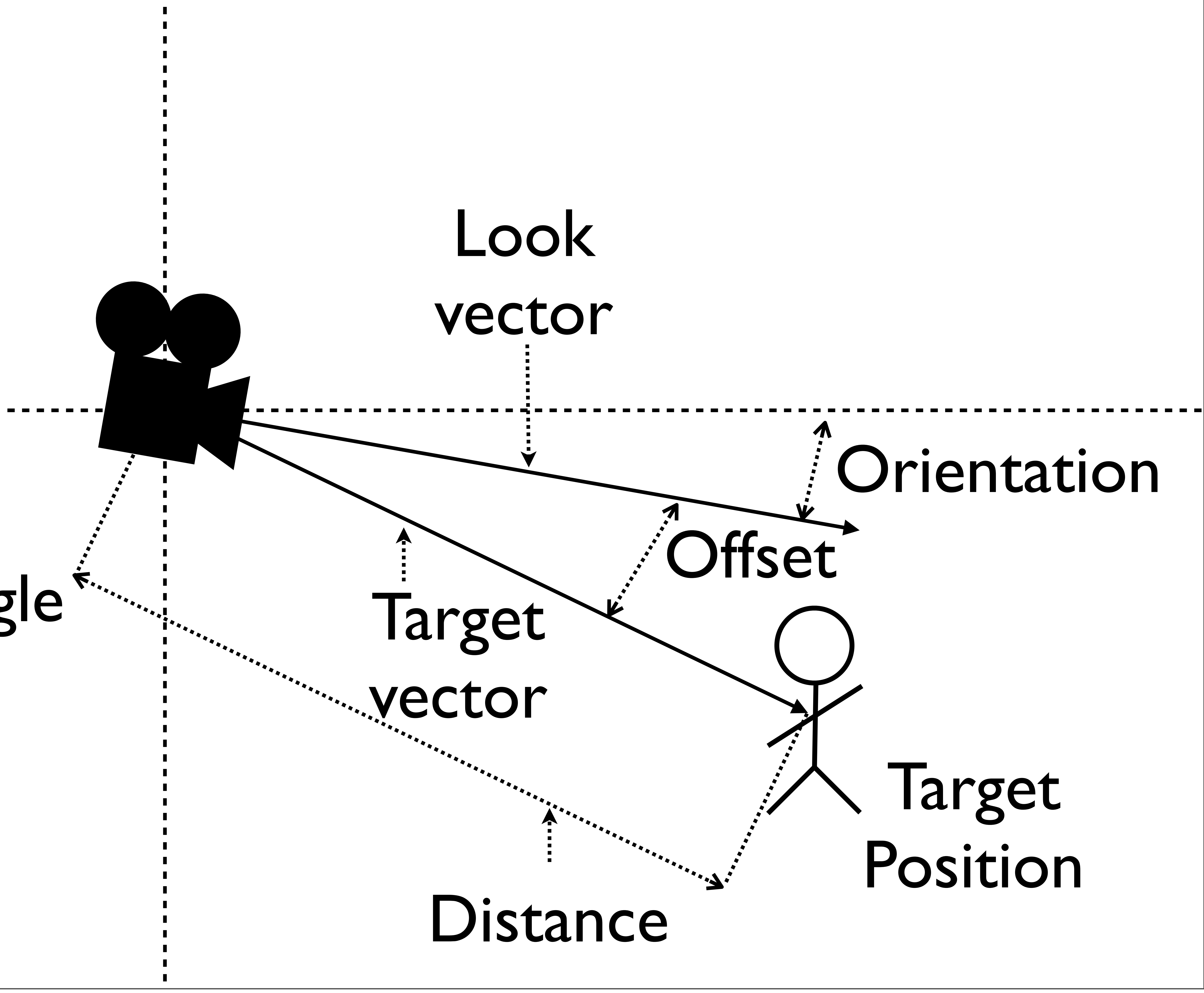
## Dynamics

- Animated Camera
- Dynamic Camera
- Combat Camera

So lets talk about the Dynamic camera, and how those parameters work.

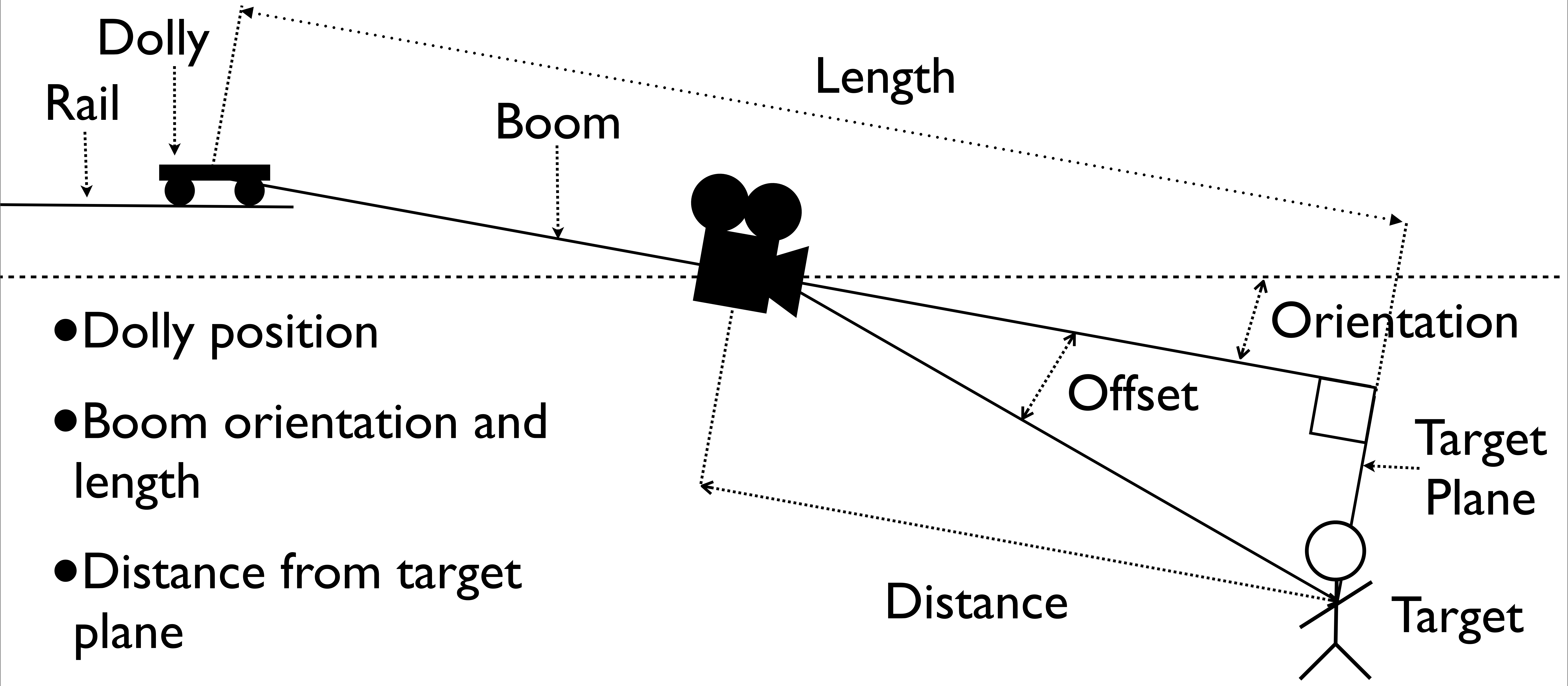
# Dynamic Camera : Model

- Target position (X,Y,Z)
- Offset in spherical coordinates (Azimuth, Elevation, Distance)
- Orientation as an Euler angle (Yaw, Pitch, Roll)



The dynamic camera model is very similar to model we use for blending. The world position of the target. The targets offset from the camera, in spherical coordinates, and the orientation of the camera as an Euler.

# Dynamic Camera : Model



- Dolly position
- Boom orientation and length
- Distance from target plane

There's a couple of extra bits we care about. There's the Dolly, which is either a fixed point in space, or a point on a NURBS curve called the Rail.

And there's the boom. This a vector coincident with the look vector, starting at the Dolly, and ending at the target plane.

We use the boom to calculate and constrain the orientation and distance properties of the camera.

We also convert the offset distance into the distance from the camera to the target plane, as that stops constraining the offset from changing the size of the target on screen.

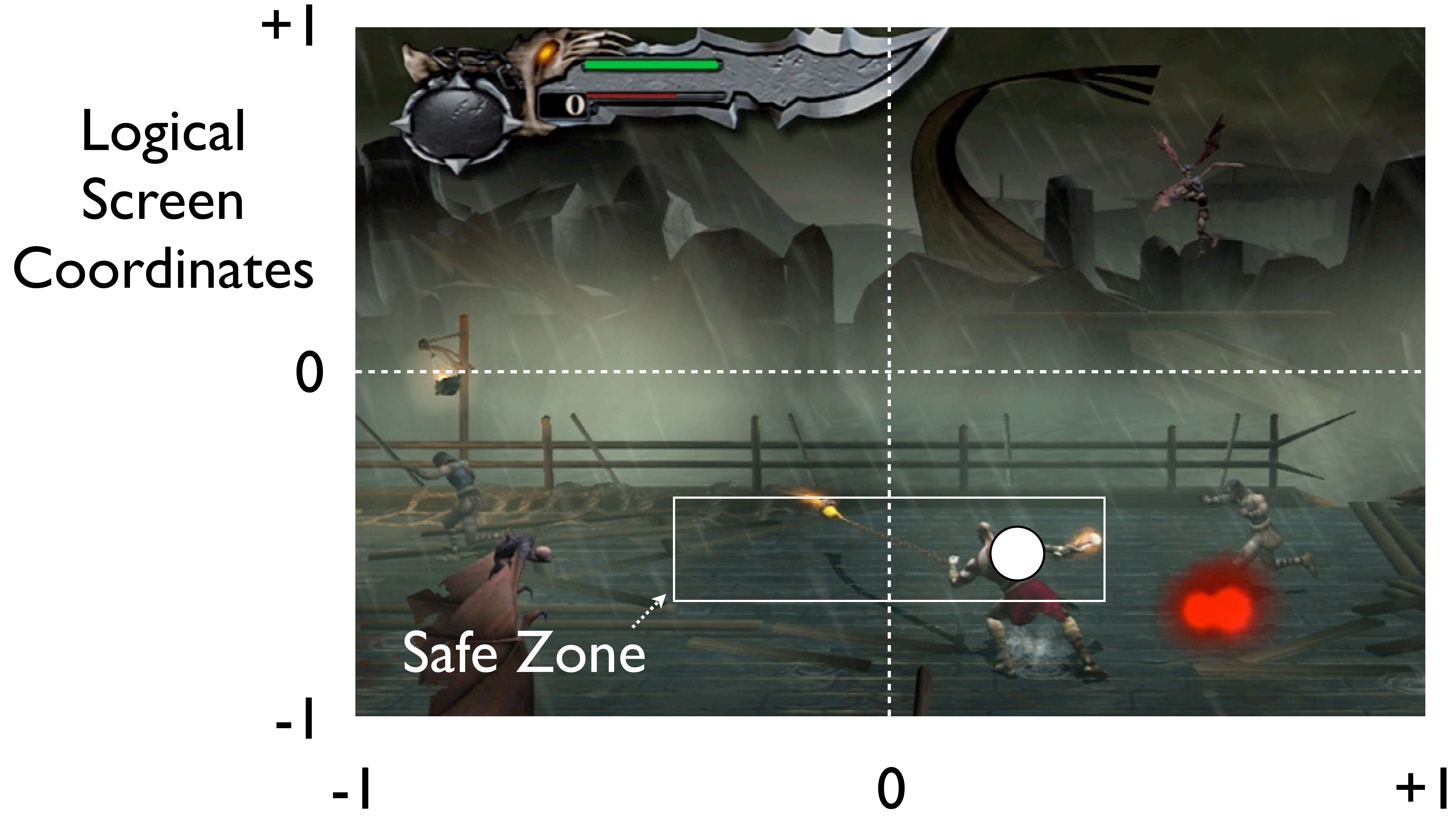
## Dynamic Camera : Constraints

| Constraint | Model Parameter |
|---|---|
| Framing | Offset Azimuth and Elevation |
| Dolly | Length and Orientation of Boom |
| Distance | Distance from Camera to Target plane |
| Orientation | Tilt and Yaw of Camera |

So here's the set of constraints, in the order they're applied.

First we set the framing, by constraining the offset azimuth and elevation, the horizontal and vertical components of the target in camera space.
Then, we update the position of the dolly to constrain the length and orientation of the boom.
Then we constrain the distance of the camera from the target plane.
And finally we constrain the orientation of the camera.

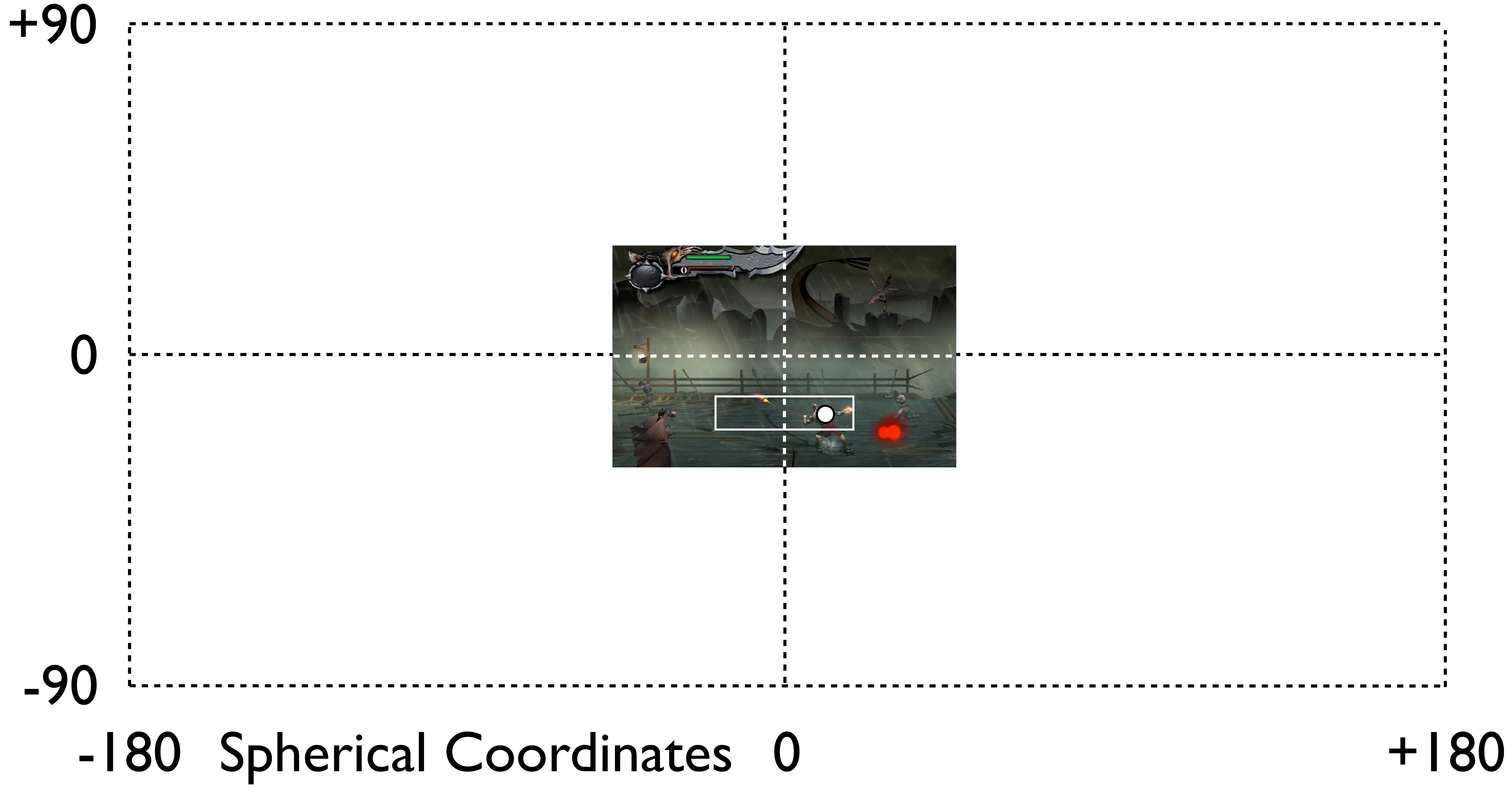
## Constraints : Framing



The framing constraint is specified as a range of logical screen coordinates from -1 to +1, that defines a rectangle on screen, known as the safe zone.

The idea being that if Kratos moves out of the safe zone, we'll move the camera to put him back in it.

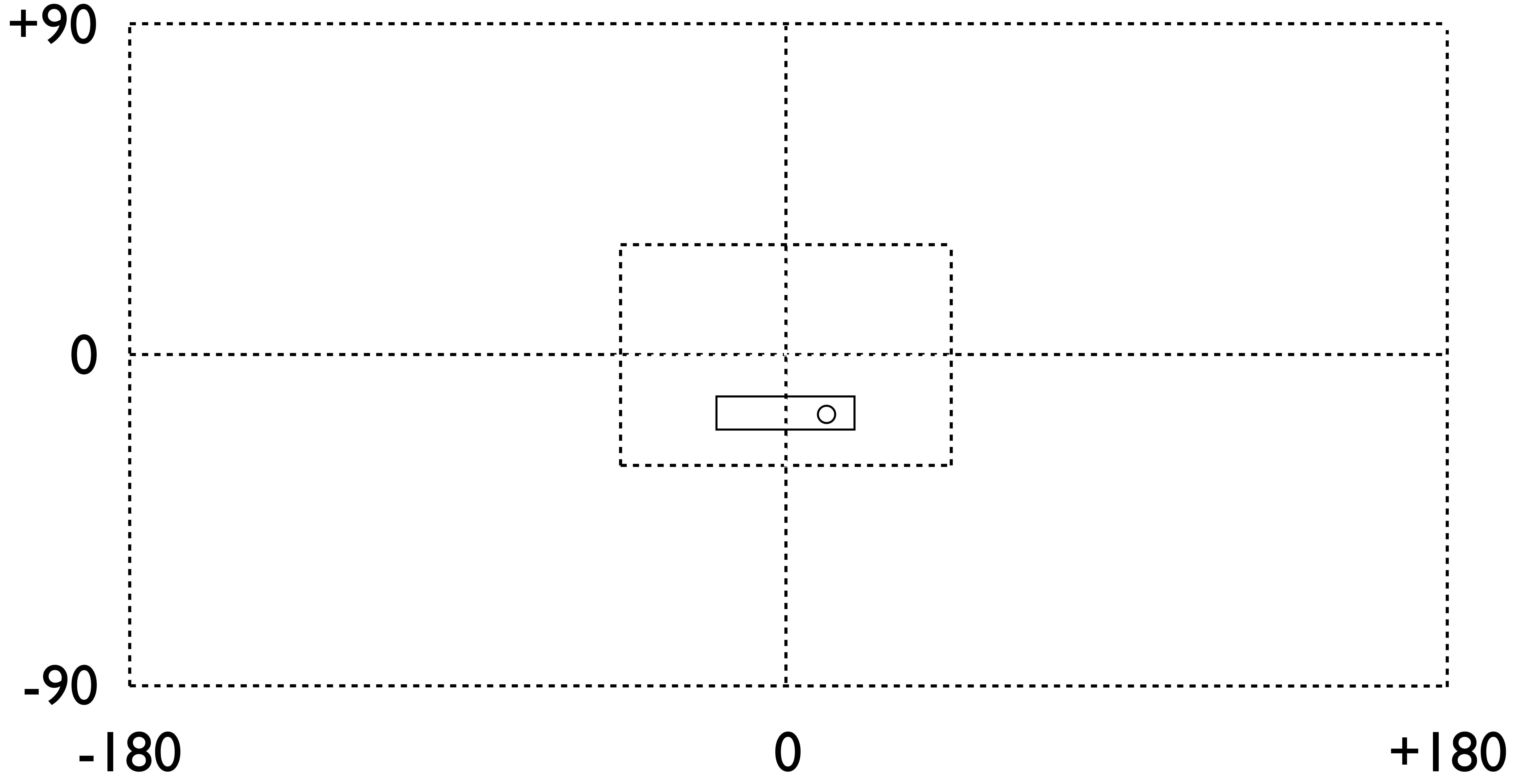
# Constraints : Framing



So the first thing we do is to use the field of view and aspect ratio to convert the safe zone into a pair of spherical angle ranges. We make this calculation each frame, because these constraint values may be animated. Obviously the spherical projection is distorted towards the top and bottom, but since horizontal fov never opens up that much, the error is small in our operating range, so we can ignore it.

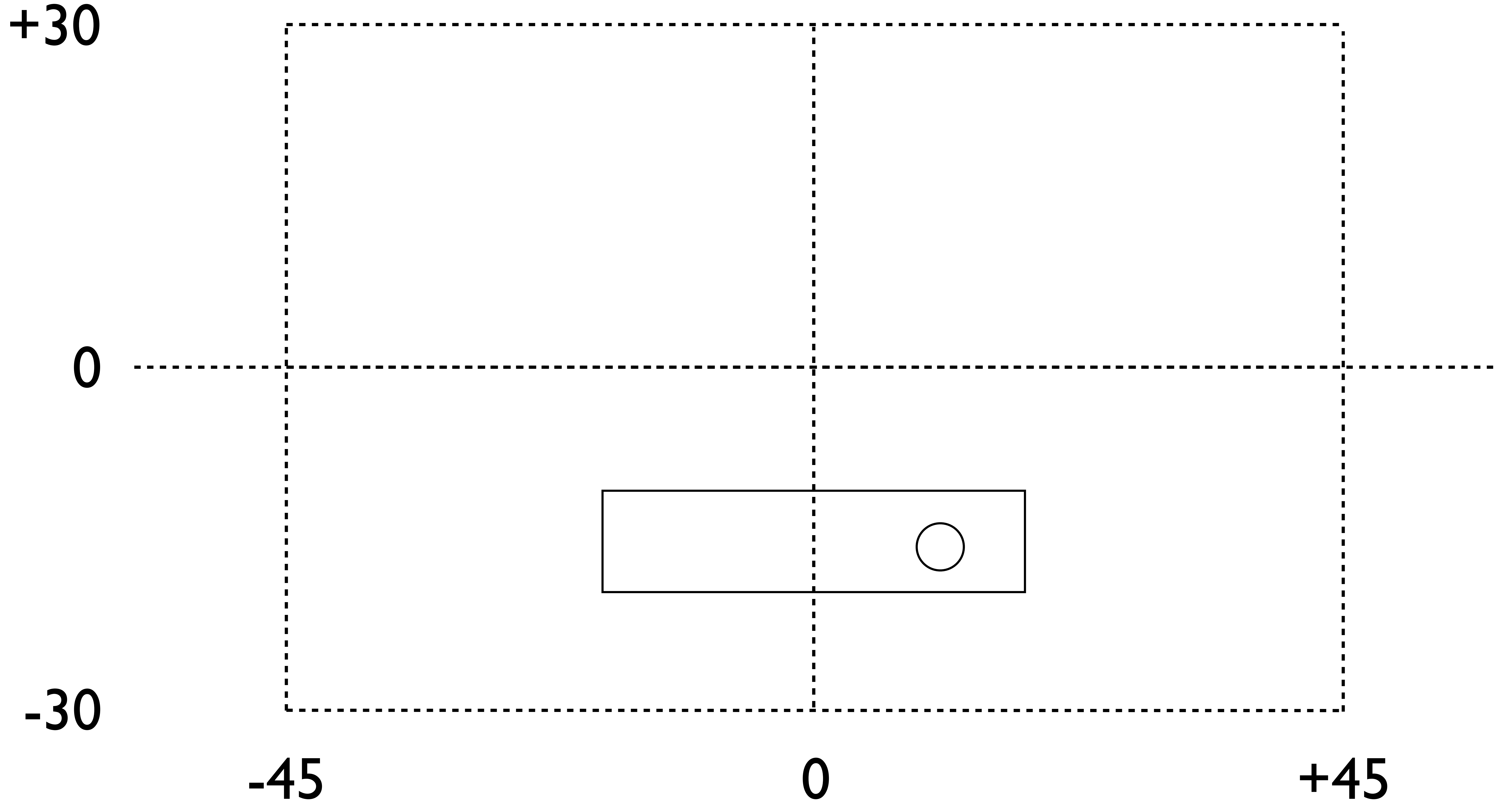
Let's clean that up a little...

# Constraints : Framing



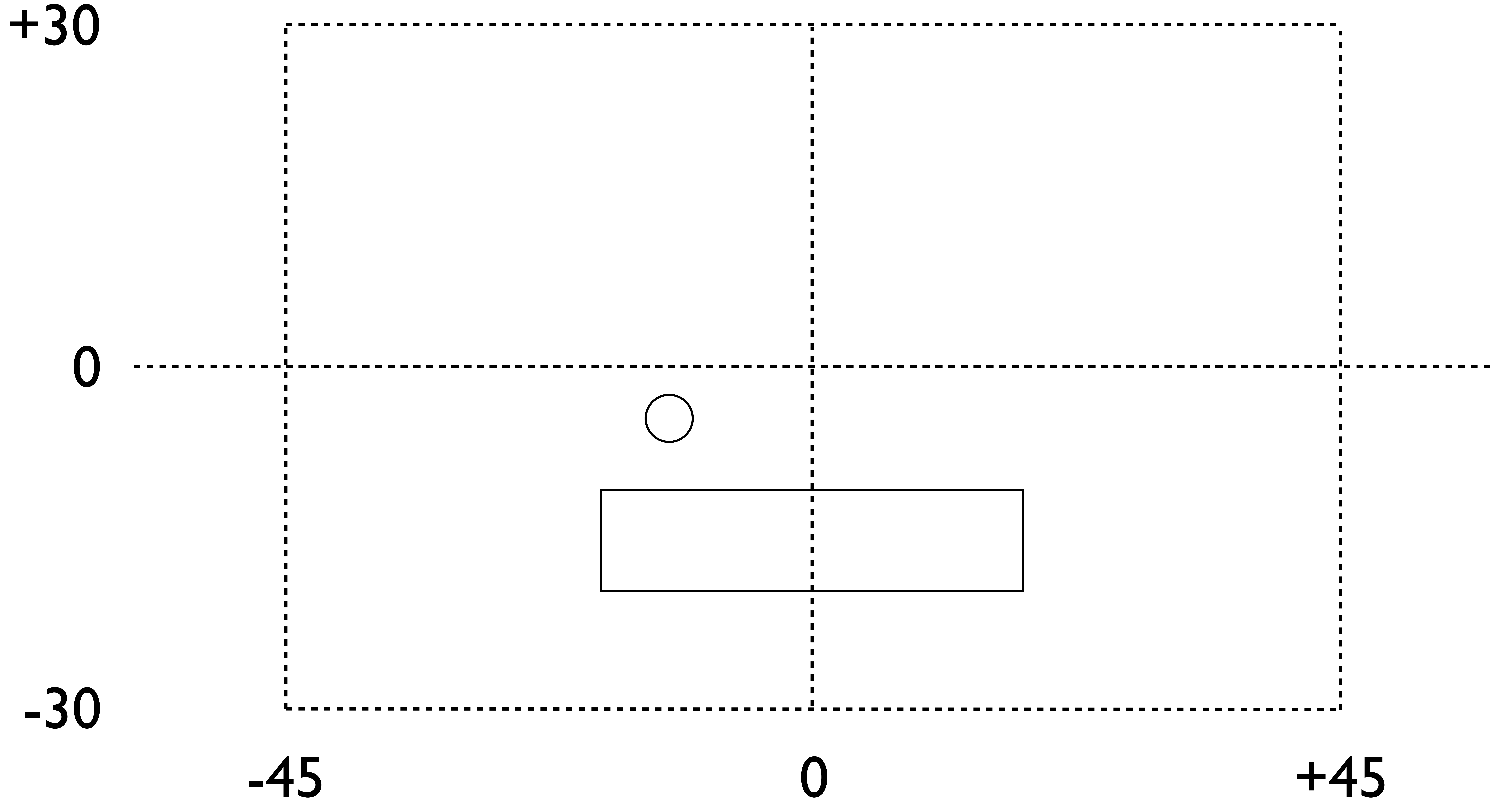
And zoom in so we can see what we're doing

# Constraints : Framing



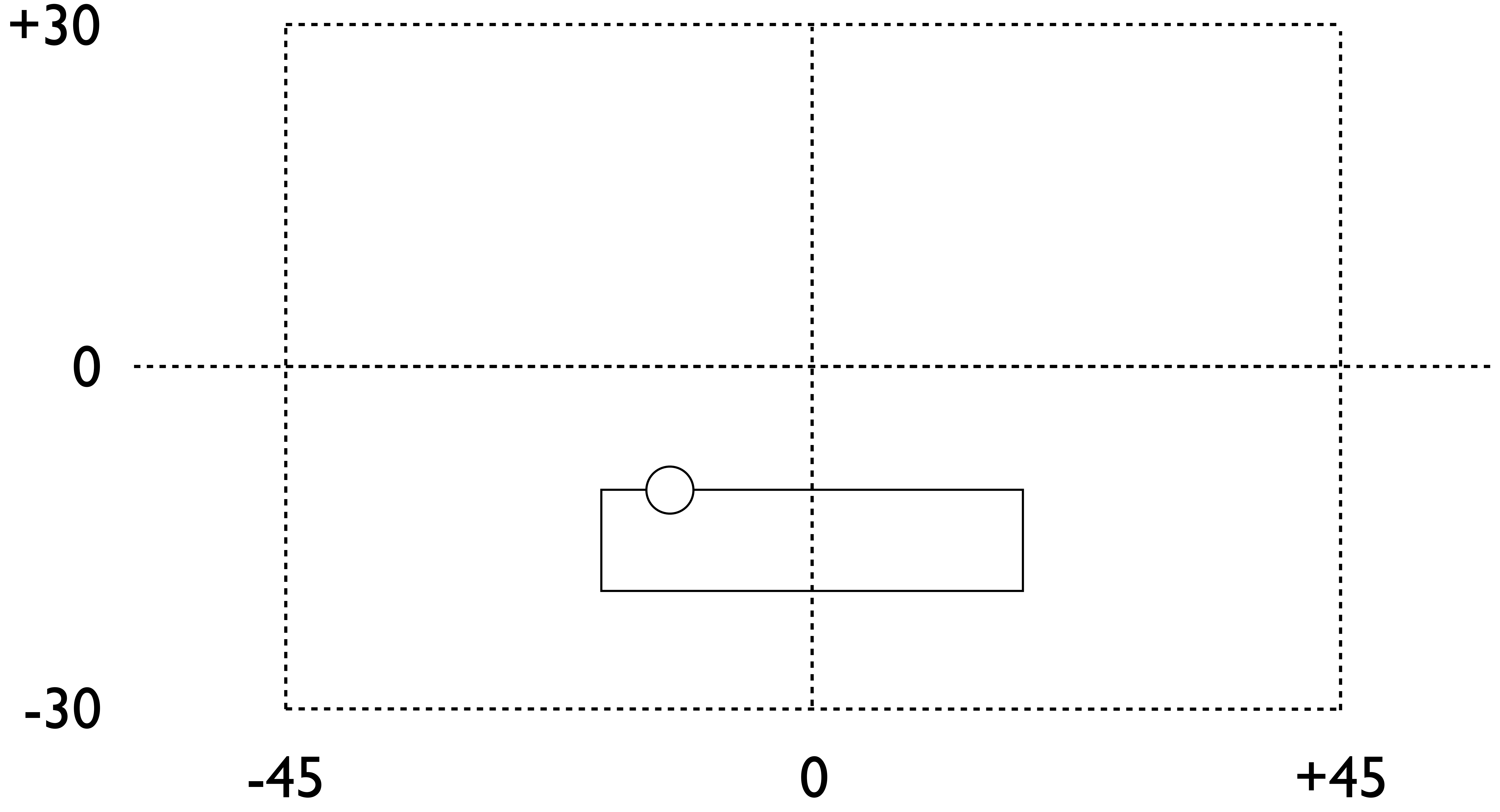
So first we update the offset angles for the updated position of the target.
*
Then we clamp them to their respective ranges. Here it’s already within the horizontal constraint, but needs to be moved down to match the top of the vertical constraint.
*
Because we don’t change orientation or distance at this stage, this has the effect of tracking the camera vertically with respect to the player.

## Constraints : Framing



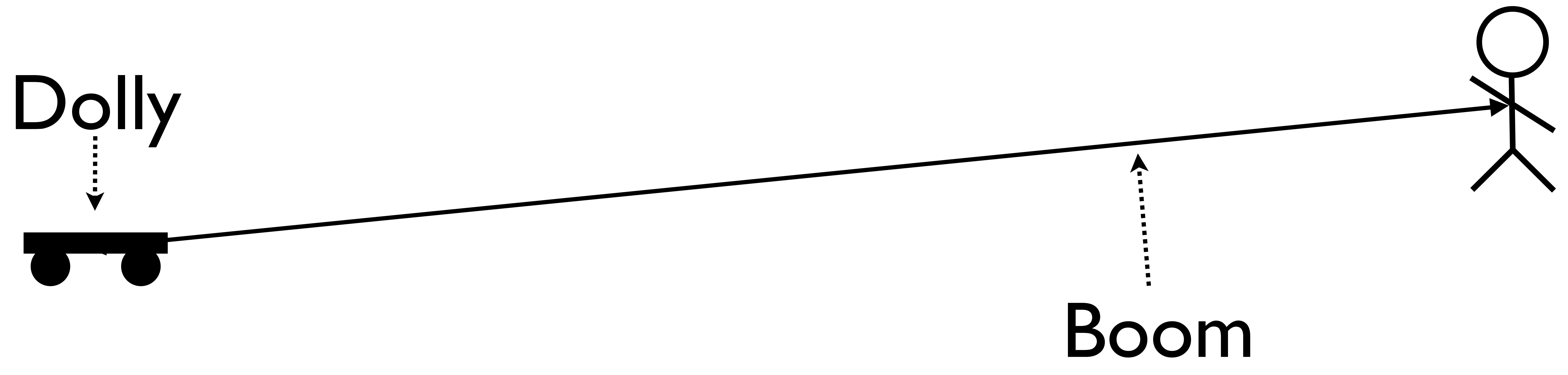
So first we update the offset angles for the updated position of the target.
*
Then we clamp them to their respective ranges. Here it's already within the horizontal constraint, but needs to be moved down to match the top of the vertical constraint.
*
Because we don't change orientation or distance at this stage, this has the effect of tracking the camera vertically with respect to the player.

## Constraints : Framing



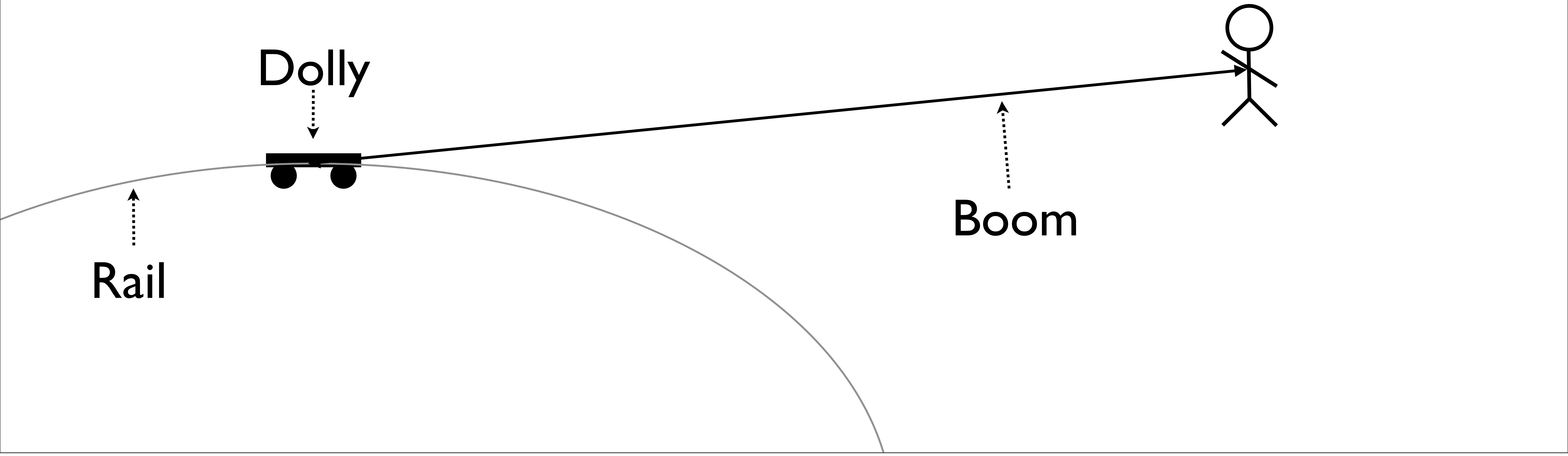
So first we update the offset angles for the updated position of the target.
*
Then we clamp them to their respective ranges. Here it’s already within the horizontal constraint, but needs to be moved down to match the top of the vertical constraint.
*
Because we don’t change orientation or distance at this stage, this has the effect of tracking the camera vertically with respect to the player.

# Constraints : Dolly & Boom



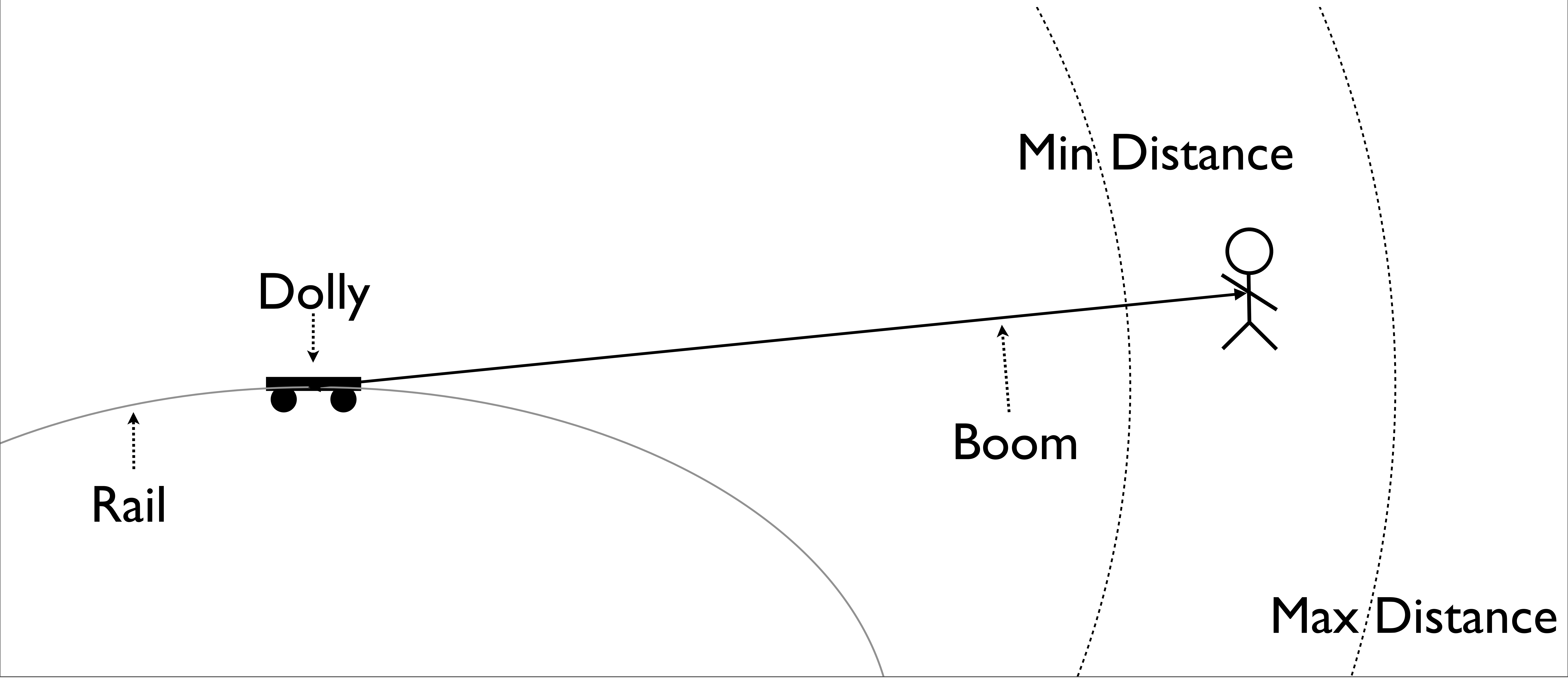
Now before we constrain distance and orientation, we have to calculate the boom, which means determining the position of the dolly. If there’s no rail, then we just use a fixed point, in space.
*If there is a rail though, then we need to move the move the dolly along it to satisfy the constraints.
*There’s a distance constraint
*and there are angle constraints, symmetrical either side of a fixed vector through the dolly, or relative to the tangent on the rail at the dolly.
* These are used to generate weights, for a minimisation function that moves the dolly to best satisfy the constraints.

# Constraints : Dolly & Boom



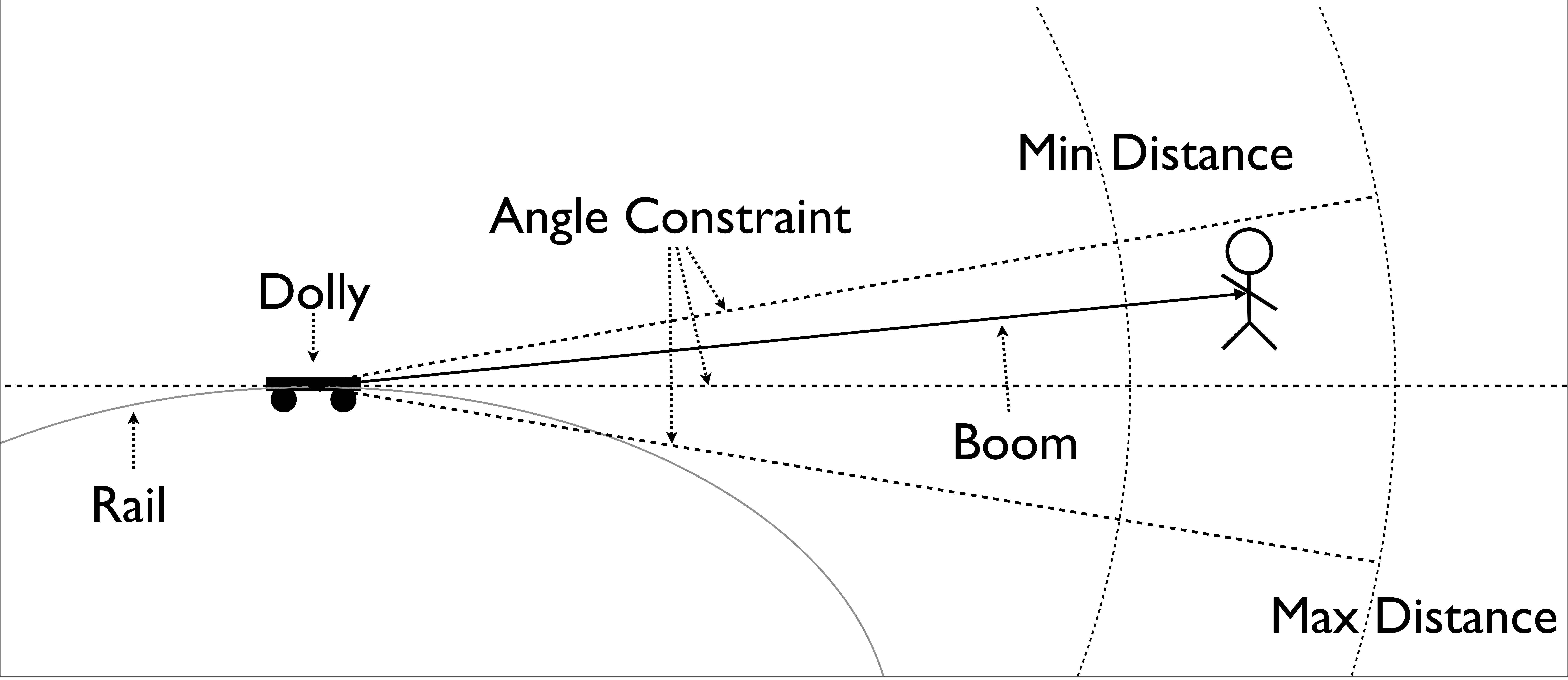
Now before we constrain distance and orientation, we have to calculate the boom, which means determining the position of the dolly. If there's no rail, then we just use a fixed point, in space.
*If there is a rail though, then we need to move the move the dolly along it to satisfy the constraints.
*There's a distance constraint
*and there are angle constraints, symmetrical either side of a fixed vector through the dolly, or relative to the tangent on the rail at the dolly.
* These are used to generate weights, for a minimisation function that moves the dolly to best satisfy the constraints.

## Constraints : Dolly & Boom



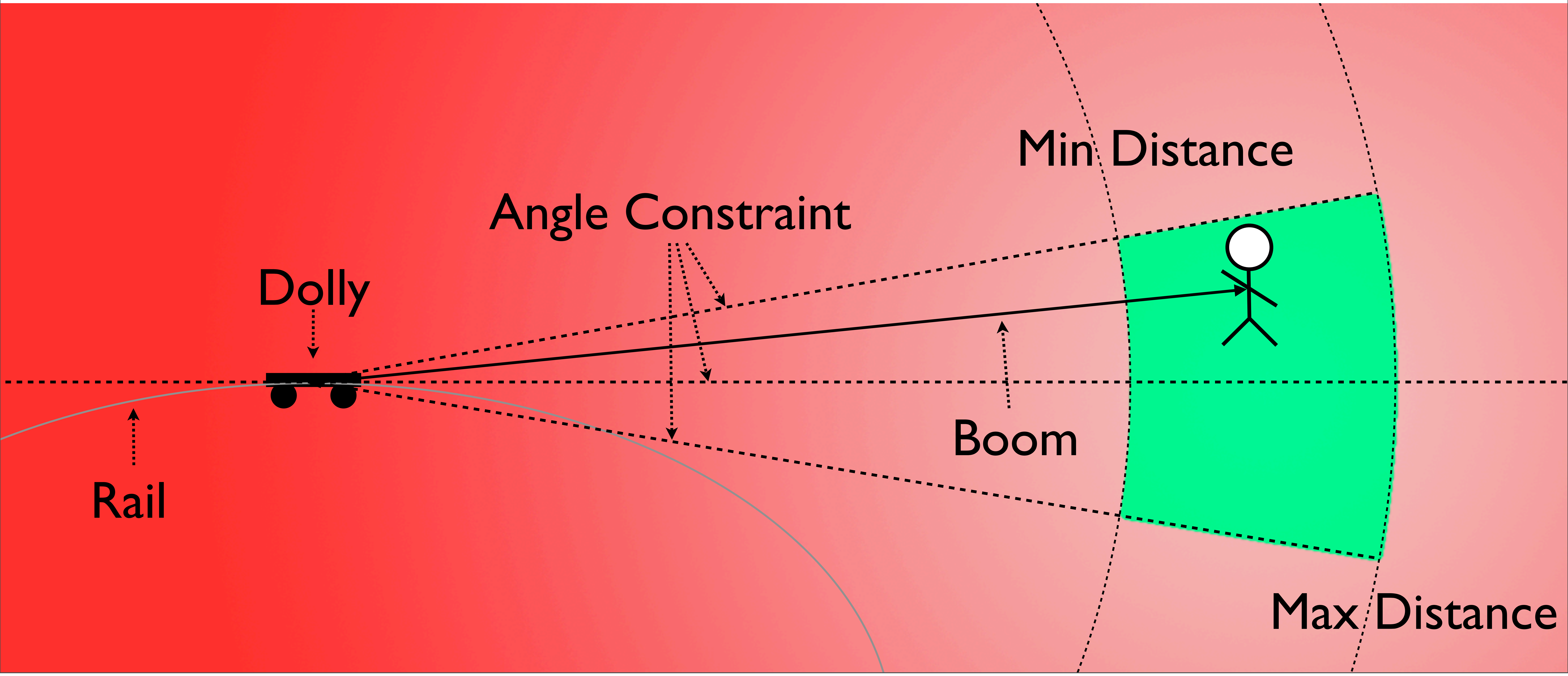
Now before we constrain distance and orientation, we have to calculate the boom, which means determining the position of the dolly. If there's no rail, then we just use a fixed point, in space.
*If there is a rail though, then we need to move the move the dolly along it to satisfy the constraints.
*There's a distance constraint
*and there are angle constraints, symmetrical either side of a fixed vector through the dolly, or relative to the tangent on the rail at the dolly.
* These are used to generate weights, for a minimisation function that moves the dolly to best satisfy the constraints.

# Constraints : Dolly & Boom



Now before we constrain distance and orientation, we have to calculate the boom, which means determining the position of the dolly. If there's no rail, then we just use a fixed point, in space.
*If there is a rail though, then we need to move the move the dolly along it to satisfy the constraints.
*There's a distance constraint
*and there are angle constraints, symmetrical either side of a fixed vector through the dolly, or relative to the tangent on the rail at the dolly.
* These are used to generate weights, for a minimisation function that moves the dolly to best satisfy the constraints.

# Constraints : Dolly & Boom



Now before we constrain distance and orientation, we have to calculate the boom, which means determining the position of the dolly. If there's no rail, then we just use a fixed point, in space.
*If there is a rail though, then we need to move the move the dolly along it to satisfy the constraints.
*There's a distance constraint
*and there are angle constraints, symmetrical either side of a fixed vector through the dolly, or relative to the tangent on the rail at the dolly.
* These are used to generate weights, for a minimisation function that moves the dolly to best satisfy the constraints.

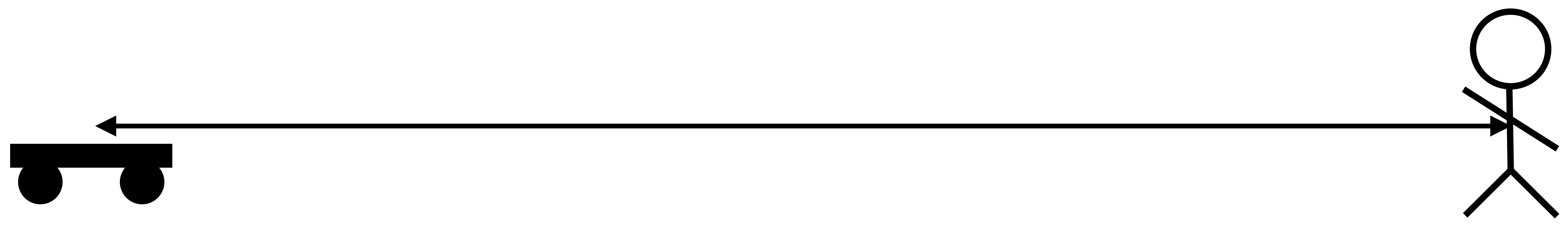
## Constraints : Minimisation

- Position expressed as single parameter to weighting function
- Weighting function calculates weight proportional to target distance outside of constraints at given position
- Minimisation function performs local search of parameter space to determine nearest minima
- Output damped to remove precision artifacts

So the minimisation function basically takes a weighting function, and an initial parameter value.
The weighting function converts the parameter into a dolly position,
works out how far outside the constraints the target would be, and returns that as a weight.
The minimisation function then searches the parameter space near the initial parameter in order to find a local minima.
During early development of the minimisation function, it suffered from small errors that made motion a little jerky, so in order to smooth it out, we added some simple damping. Now although the function is pretty smooth itself these days, the designers liked the effect of the damping, and it remains in the system.

## Constraints : Distance

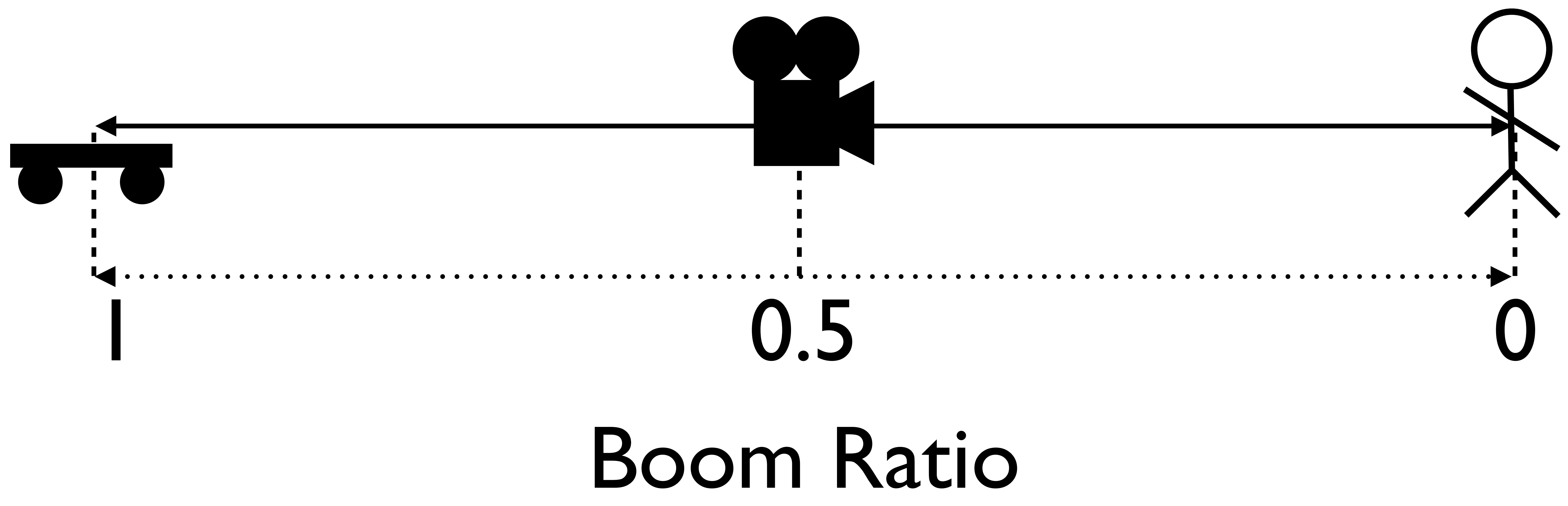
- Distance from Boom Ratio
- Constrain to Dolly
- Constrain to Target
- Optionally set to fixed distance



Now we have a boom, we use it calculate the distance attribute.
*For this we use the Boom Ratio parameter, which defines a logical position on the boom, from 1..0 with 1 being the dolly, and 0 being the target plane.
*Next we apply a min / max constraint to the distance from the dolly.
*And then the same again to the target. We do the target distance constraint last, to ensure that the camera never goes crashing through the hero, and clips his mesh.
Alternatively we can set the distance to a fixed value.

# Constraints : Distance

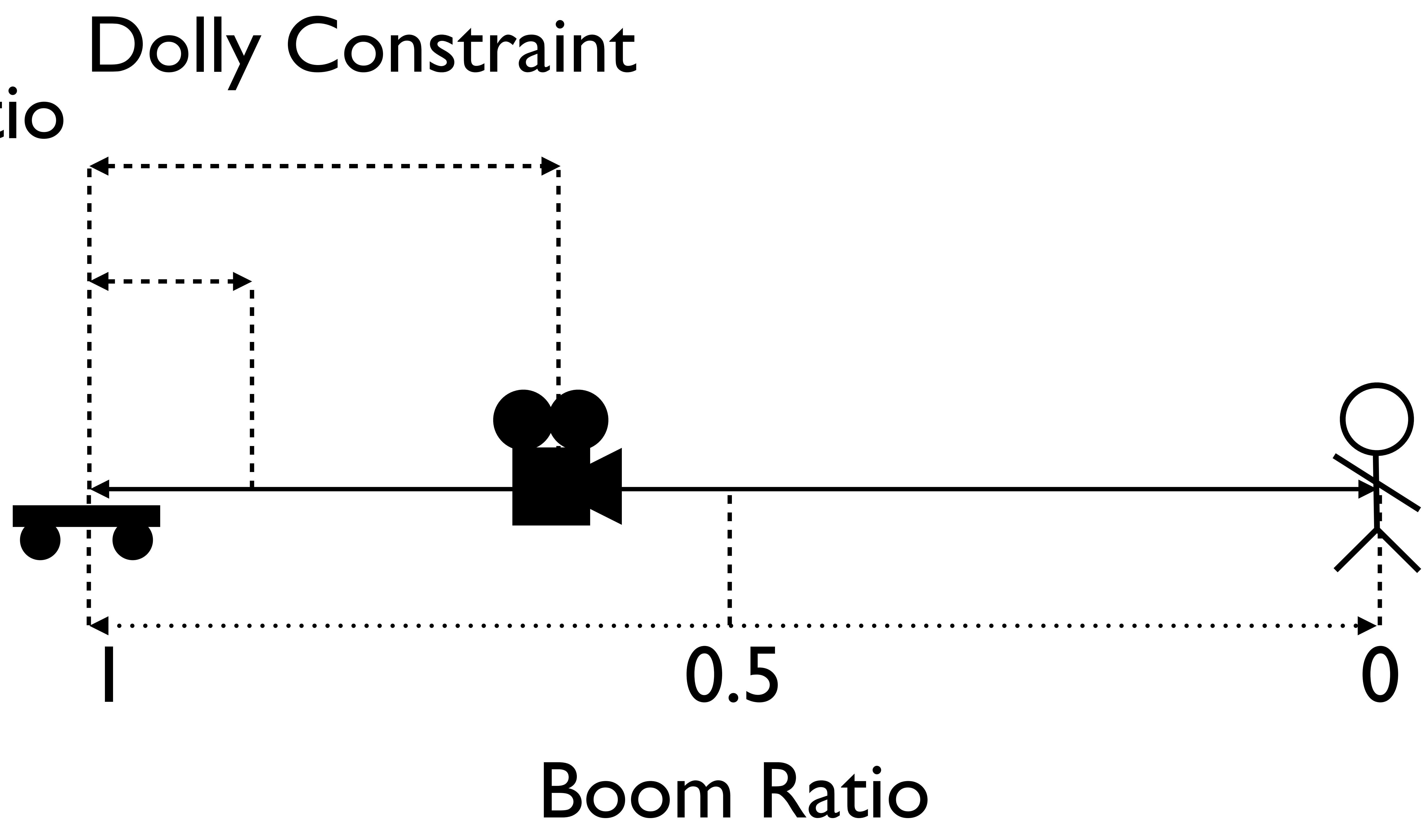
- Distance from Boom Ratio
- Constrain to Dolly
- Constrain to Target
- Optionally set to fixed distance



Now we have a boom, we use it calculate the distance attribute.
*For this we use the Boom Ratio parameter, which defines a logical position on the boom, from 1..0 with 1 being the dolly, and 0 being the target plane.
*Next we apply a min / max constraint to the distance from the dolly.
*And then the same again to the target. We do the target distance constraint last, to ensure that the camera never goes crashing through the hero, and clips his mesh.
Alternatively we can set the distance to a fixed value.

# Constraints : Distance

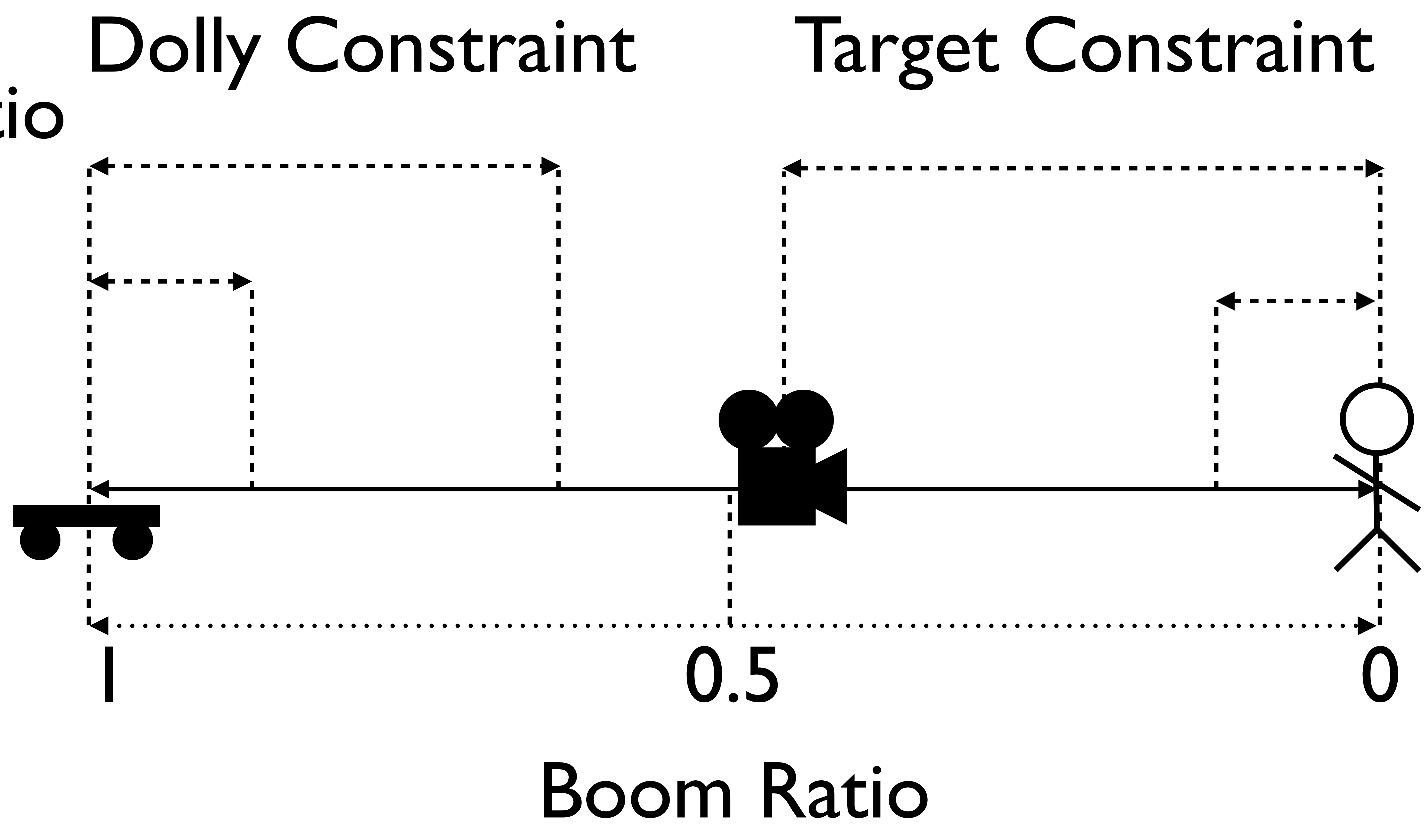
- Distance from Boom Ratio
- Constrain to Dolly
- Constrain to Target
- Optionally set to fixed distance



Now we have a boom, we use it calculate the distance attribute.
*For this we use the Boom Ratio parameter, which defines a logical position on the boom, from 1..0 with 1 being the dolly, and 0 being the target plane.
*Next we apply a min / max constraint to the distance from the dolly.
*And then the same again to the target. We do the target distance constraint last, to ensure that the camera never goes crashing through the hero, and clips his mesh.
Alternatively we can set the distance to a fixed value.

# Constraints : Distance

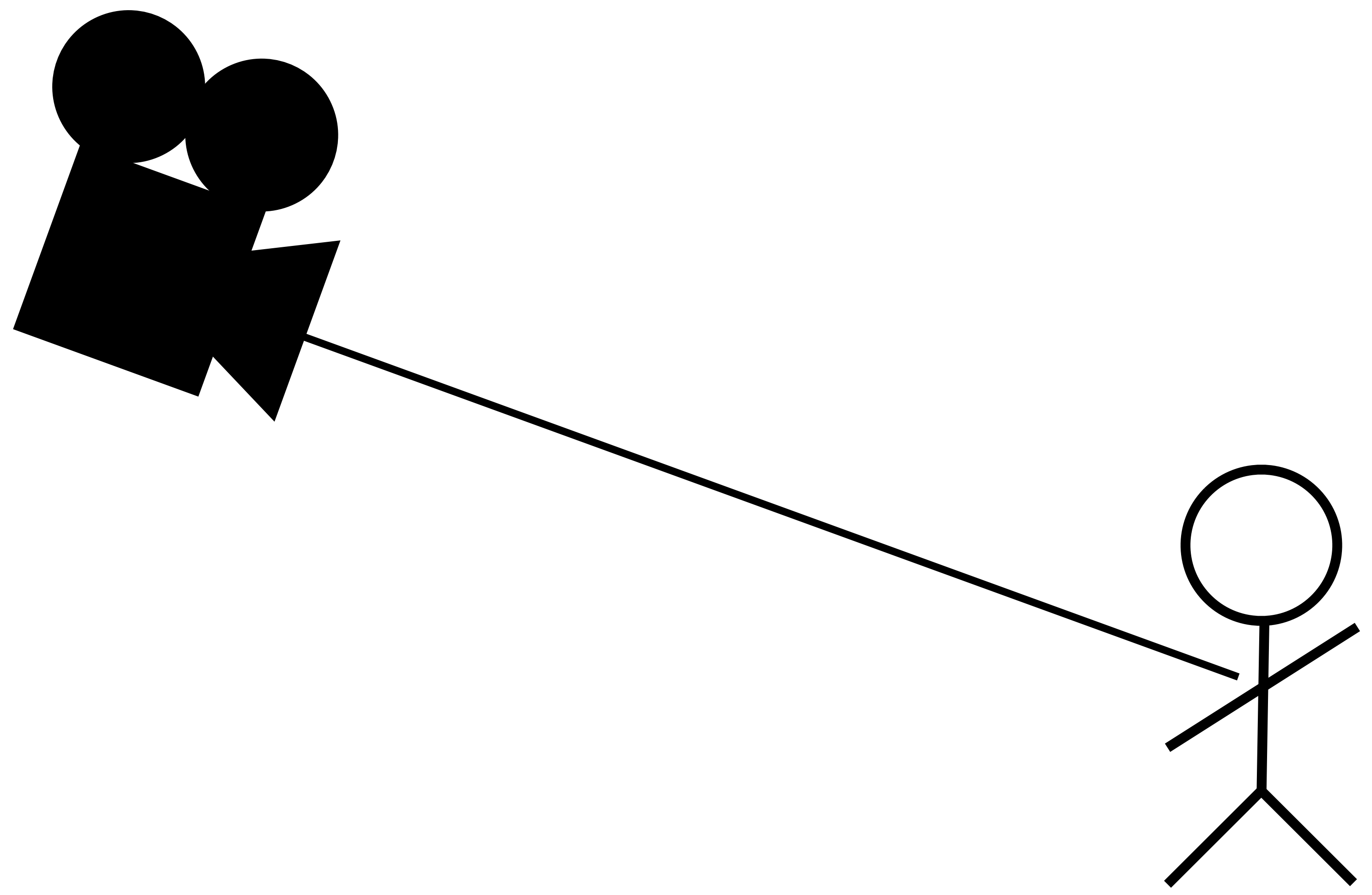
- Distance from Boom Ratio
- Constrain to Dolly
- Constrain to Target
- Optionally set to fixed distance



Now we have a boom, we use it calculate the distance attribute.
*For this we use the Boom Ratio parameter, which defines a logical position on the boom, from 1..0 with 1 being the dolly, and 0 being the target plane.
*Next we apply a min / max constraint to the distance from the dolly.
*And then the same again to the target. We do the target distance constraint last, to ensure that the camera never goes crashing through the hero, and clips his mesh.
Alternatively we can set the distance to a fixed value.

## Constraints : Orientation

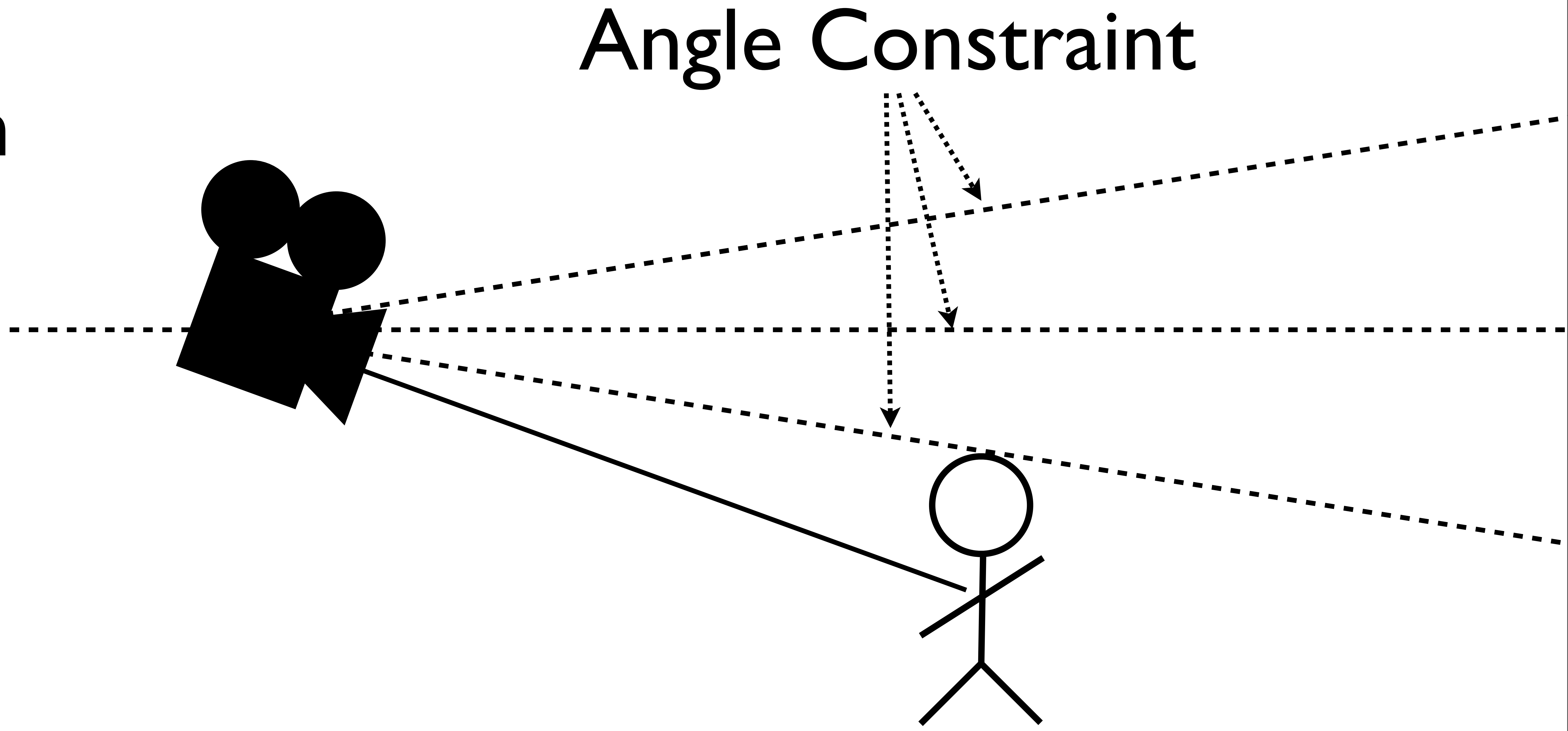
- Yaw and Pitch seperated
- Set Orientation from Boom
- Constrain Orientation to valid range
- Rotation applied around camera position
- Optionally set to fixed angle



Finally we calculate and constrain orientation. This is done for the pitch and yaw angles separately. Roll is not generated directly, and is set to 0 at this stage.
First we set the orientation of the camera, from the orientation of the boom.
* The we check this angle against the same angle constraints we used to constrain the boom.
* If the orientation is outside the constraint, then we rotate the camera to fit them. This happens at the camera, so we have to recalculate distance and offset to keep the camera from moving.
Alternatively, like distance, we can set the orientation to a fixed angle.

# Constraints : Orientation

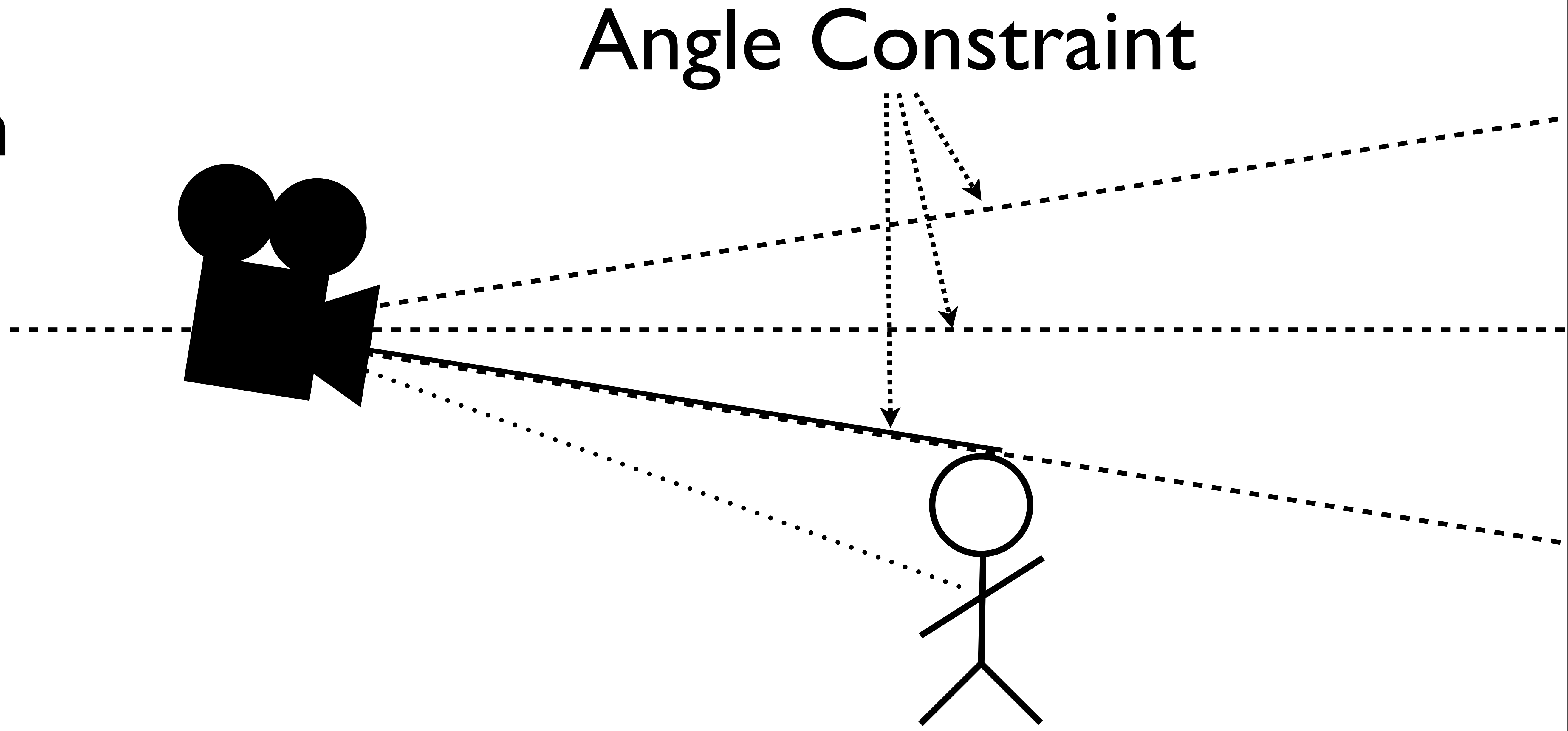
- Yaw and Pitch seperated
- Set Orientation from Boom
- Constrain Orientation to valid range
- Rotation applied around camera position
- Optionally set to fixed angle



Finally we calculate and constrain orientation. This is done for the pitch and yaw angles separately. Roll is not generated directly, and is set to 0 at this stage.
First we set the orientation of the camera, from the orientation of the boom.
* The we check this angle against the same angle constraints we used to constrain the boom.
* If the orientation is outside the constraint, then we rotate the camera to fit them. This happens at the camera, so we have to recalculate distance and offset to keep the camera from moving.
Alternatively, like distance, we can set the orientation to a fixed angle.

# Constraints : Orientation

- Yaw and Pitch seperated
- Set Orientation from Boom
- Constrain Orientation to valid range
- Rotation applied around camera position
- Optionally set to fixed angle



Finally we calculate and constrain orientation. This is done for the pitch and yaw angles separately. Roll is not generated directly, and is set to 0 at this stage.
First we set the orientation of the camera, from the orientation of the boom.
* The we check this angle against the same angle constraints we used to constrain the boom.
* If the orientation is outside the constraint, then we rotate the camera to fit them. This happens at the camera, so we have to recalculate distance and offset to keep the camera from moving.
Alternatively, like distance, we can set the orientation to a fixed angle.

## Constraints : Conflict

- Applying one constraint inevitably pushes the system out of another constraint
- Try to minimise constraint conflict
- Best case, system settles
- Worst case, system explodes
- In case of oscillation, relax constraints

Almost inevitably, whenever you apply one constraint, you push the model out of compliance with another constraint. Now we can try to minimise that effect, by making the constraints affect independent parameters of the model, but sometimes that's just not possible. As we saw with the rotation constraint, sometimes you affect the whole model. Usually, these conflicts settle themselves quickly and quietly, but sometimes they oscillate, and in the worst case, they explode in a nightmare scenario of positive feedback hell.

Fortunately this is pretty easily fixed by opening up some of the constraints. In my experience it's usually the orientation constraints. Opening them up from 0 to 5 degrees can make the difference between broken mess, and a perfectly smooth camera.

## Dynamic Camera : Constraints

- Maya Camera Position - X,Y,Z
- Maya Camera Orientation - Pitch, Yaw
- Safe Zone - Top, Bottom, Left, Right
- Boom Ratio
- Distance from Dolly - Min, Max
- Distance from Target - Min, Max
- Angle Constraints - Pitch, Yaw
- Angle Constraint Flags - Rail Relative, Move Dolly

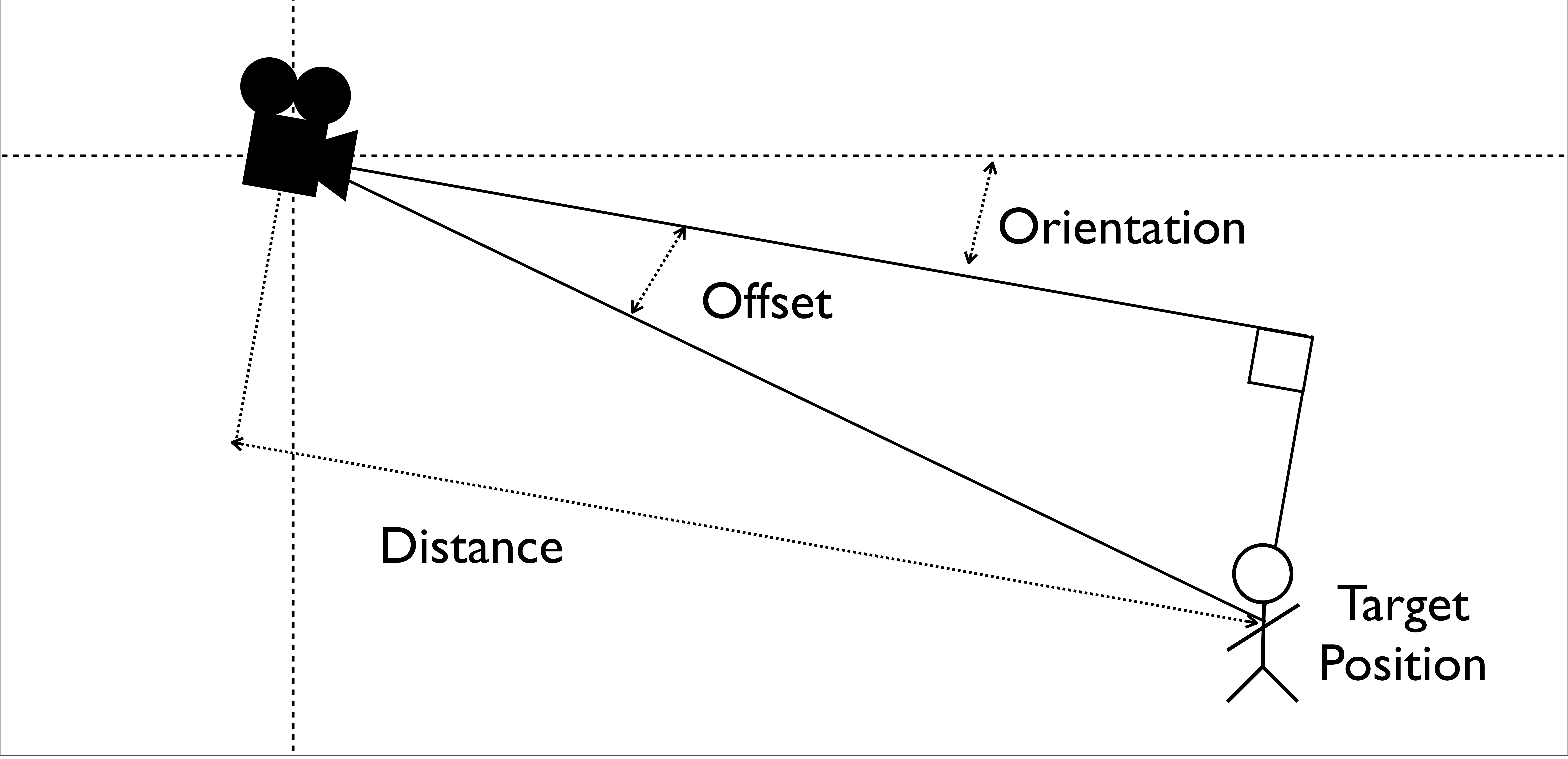
So here's a list of all the constraints the designers have control over.

Note that we use the dolly distance and angle constraints for the boom, and for the camera itself. This keeps the constraint parameters simple.

As I mentioned previously, all these parameters can be animated and the Maya camera can be attached to an animated objected, such as a Titan.

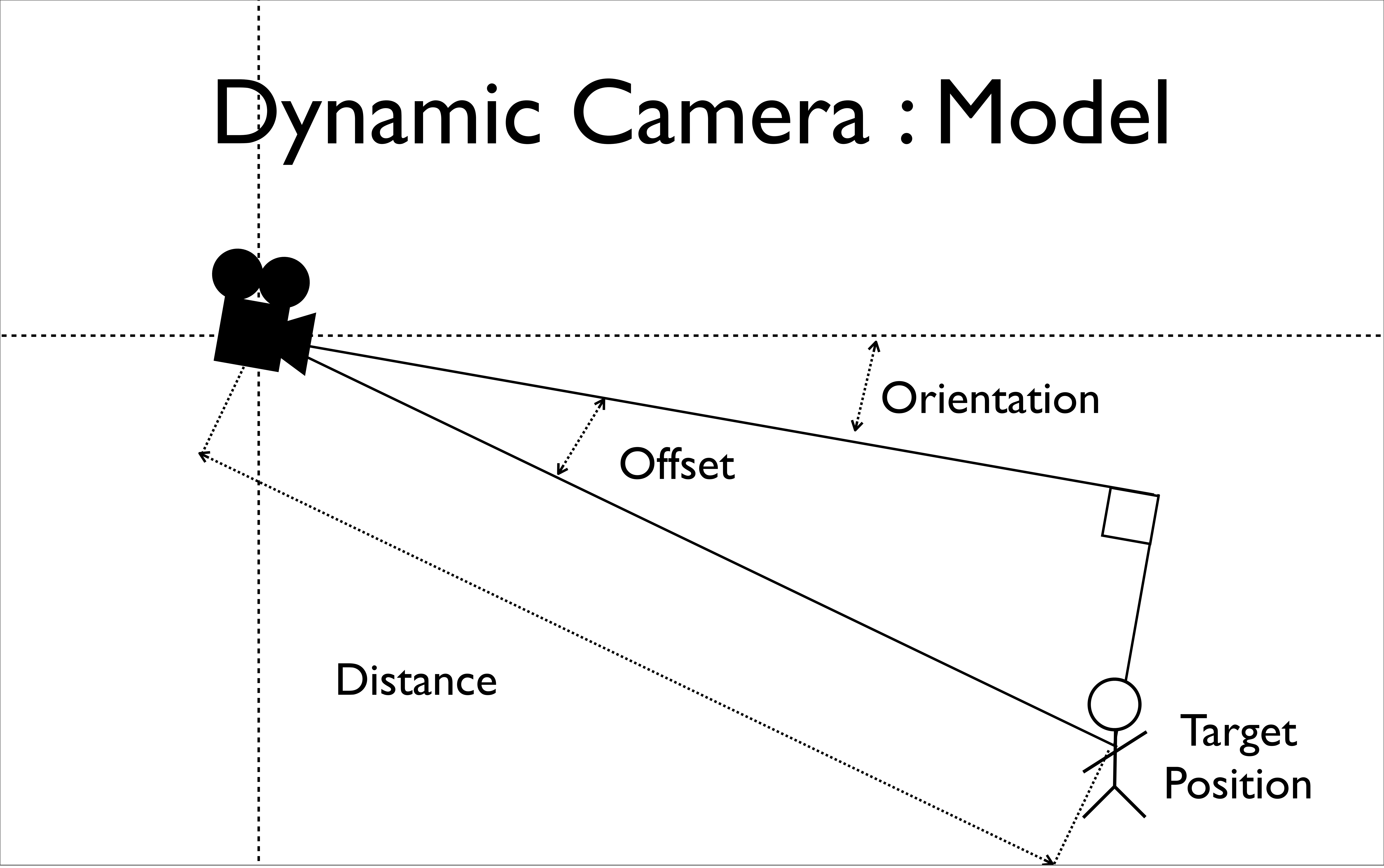
The two exceptions would be the two flags at the bottom there. Rail Relative determines whether or not the angle constraints are expressed relative to the tangent at the dolly. And Move Dolly controls whether the angle constraints are applied to the Dolly.

# Dynamic Camera : Model



So having generated and constrained Offset, Orientation, and Distance, we now have our updated camera position and orientation. Finally we convert Distance back into blending format, and we're ready to send it to the blender.

# Dynamic Camera : Model



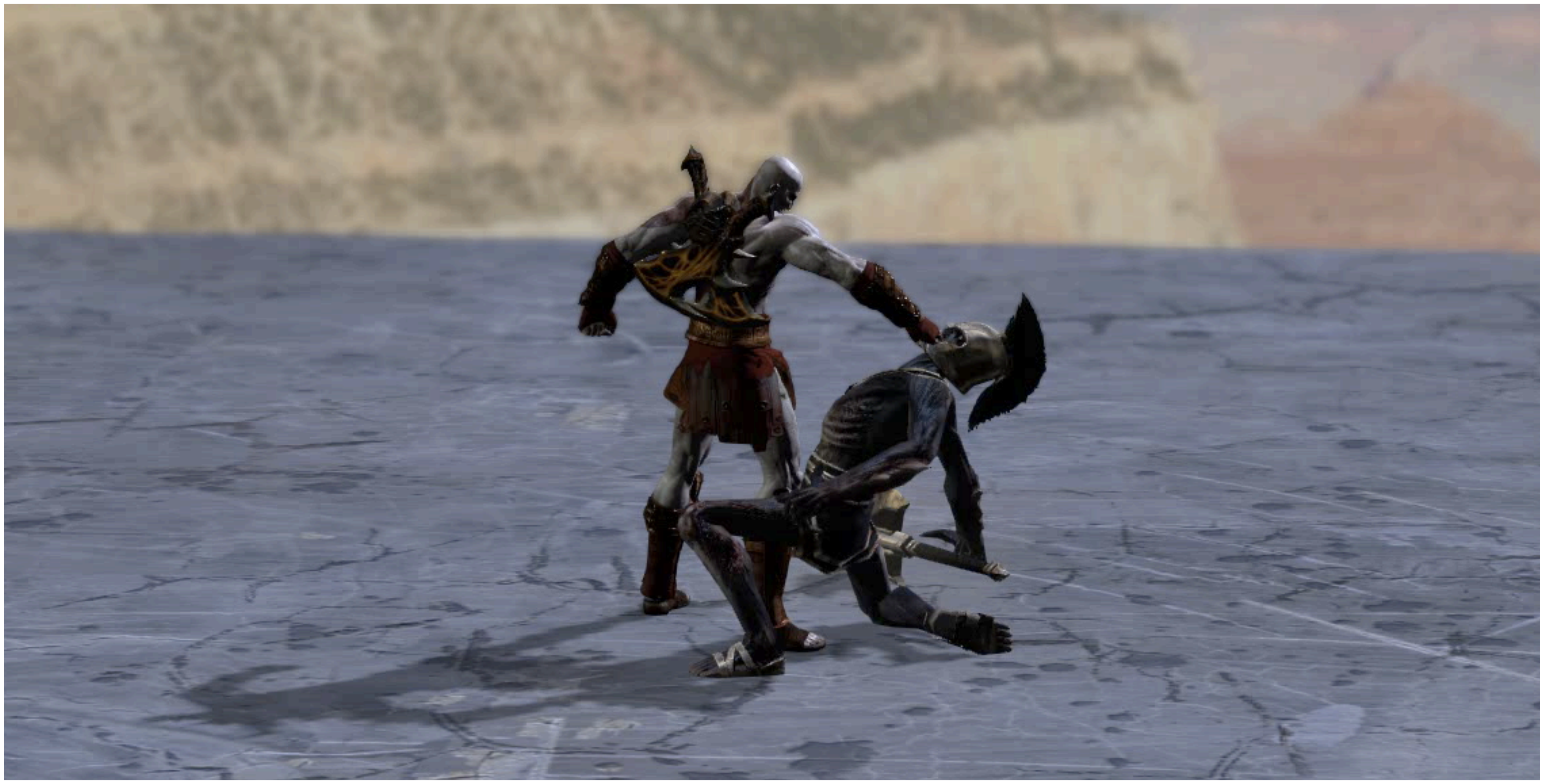
So having generated and constrained Offset, Orientation, and Distance, we now have our updated camera position and orientation. Finally we convert Distance back into blending format, and we're ready to send it to the blender.

# Dynamics

- Animated Camera
- Dynamic Camera
- Combat Camera

Then there's the combat camera.

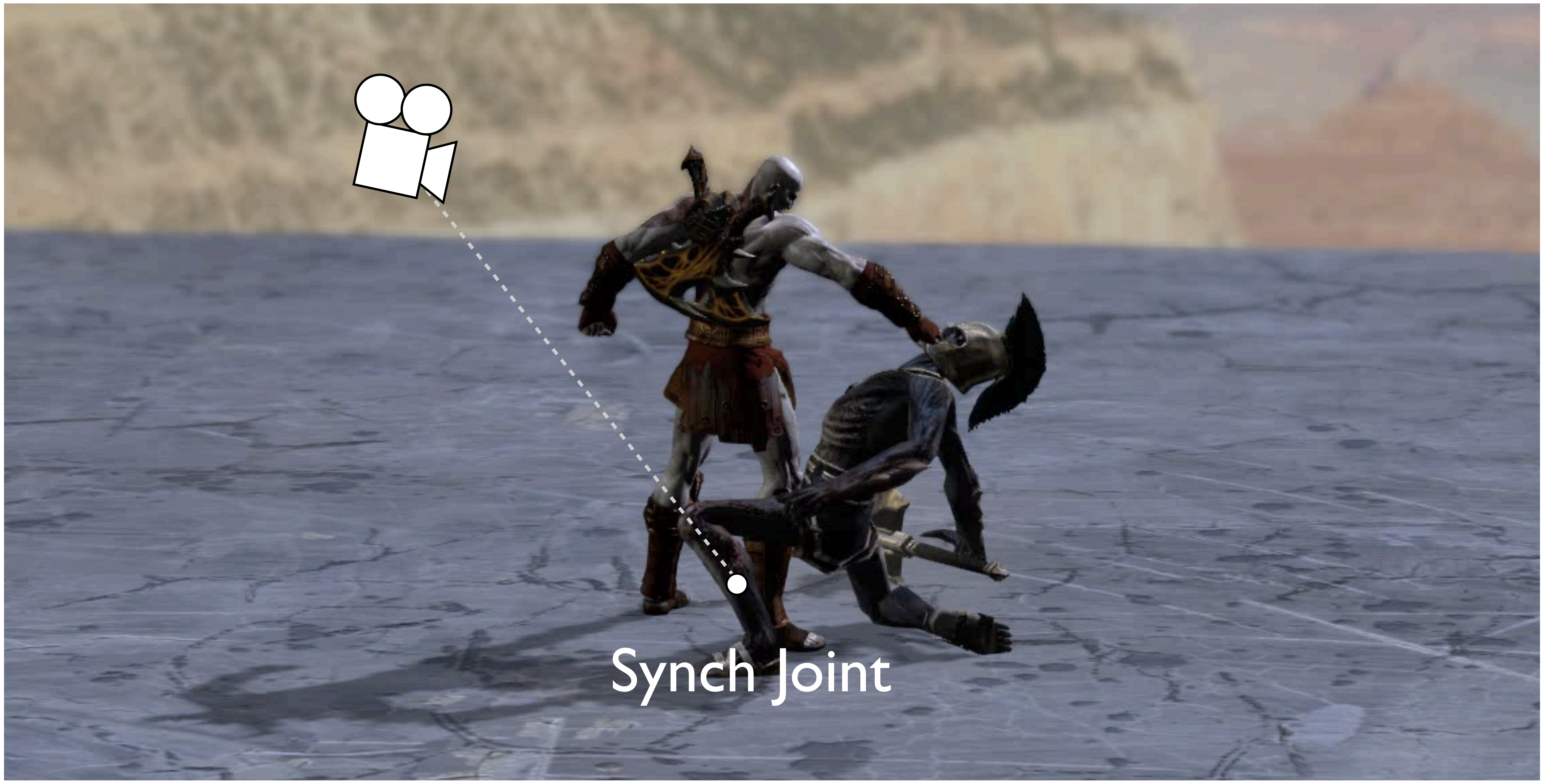
## Combat Camera



So here's our hero doing the thing he loves most, getting a little hands on time with a dear friend. When this happens, we like to take more direct control of the camera, to get the best shot of the action.

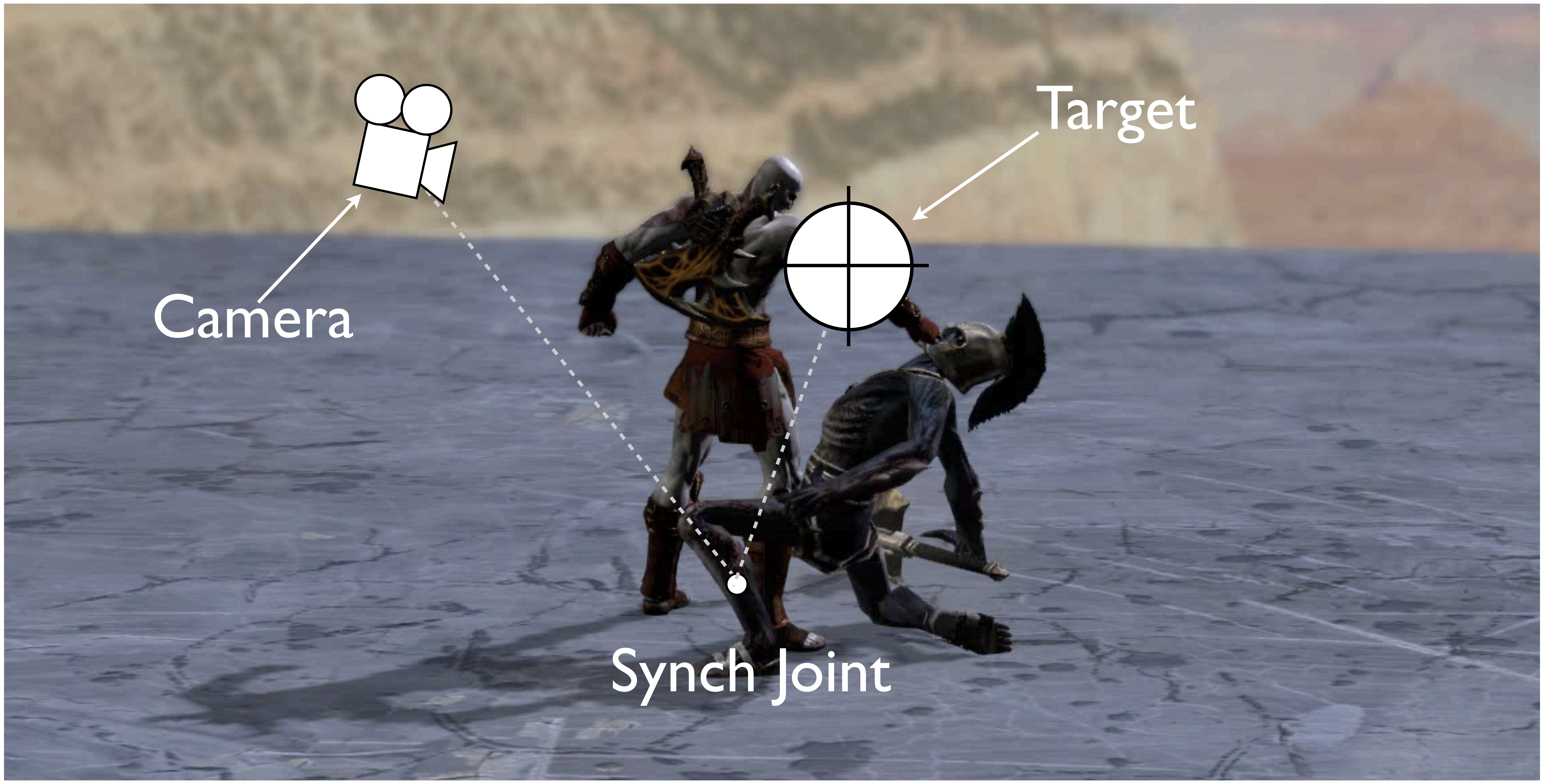
Now these events can happen anywhere in the environment, so obviously we can't author these cameras in the world.

# Combat Camera : Rig



Instead we animate them relative to the characters involved. So there’s effectively three characters in this scene. Our hero, his friend, and the camera rig. All three are synched by moving them such that a joint in each, called the synch joint, matches position and orientation. The animations they’re playing are also synchronised, and all run off of the same clock.

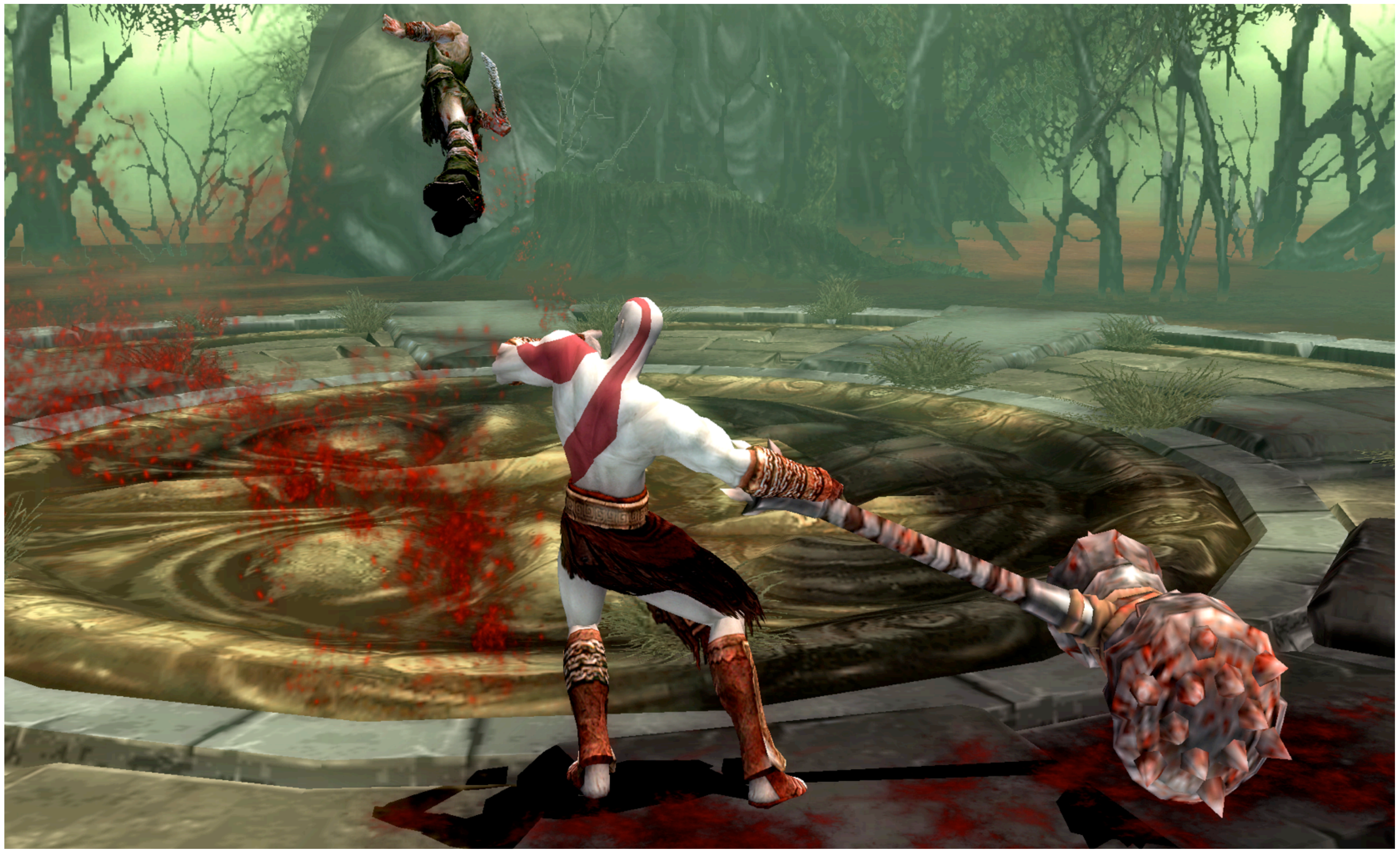
# Combat Camera : Rig



The combat rig is a creature with three joints. There's the synch joint at the root, and attached to that are a Camera and a Target, each on their own joints.

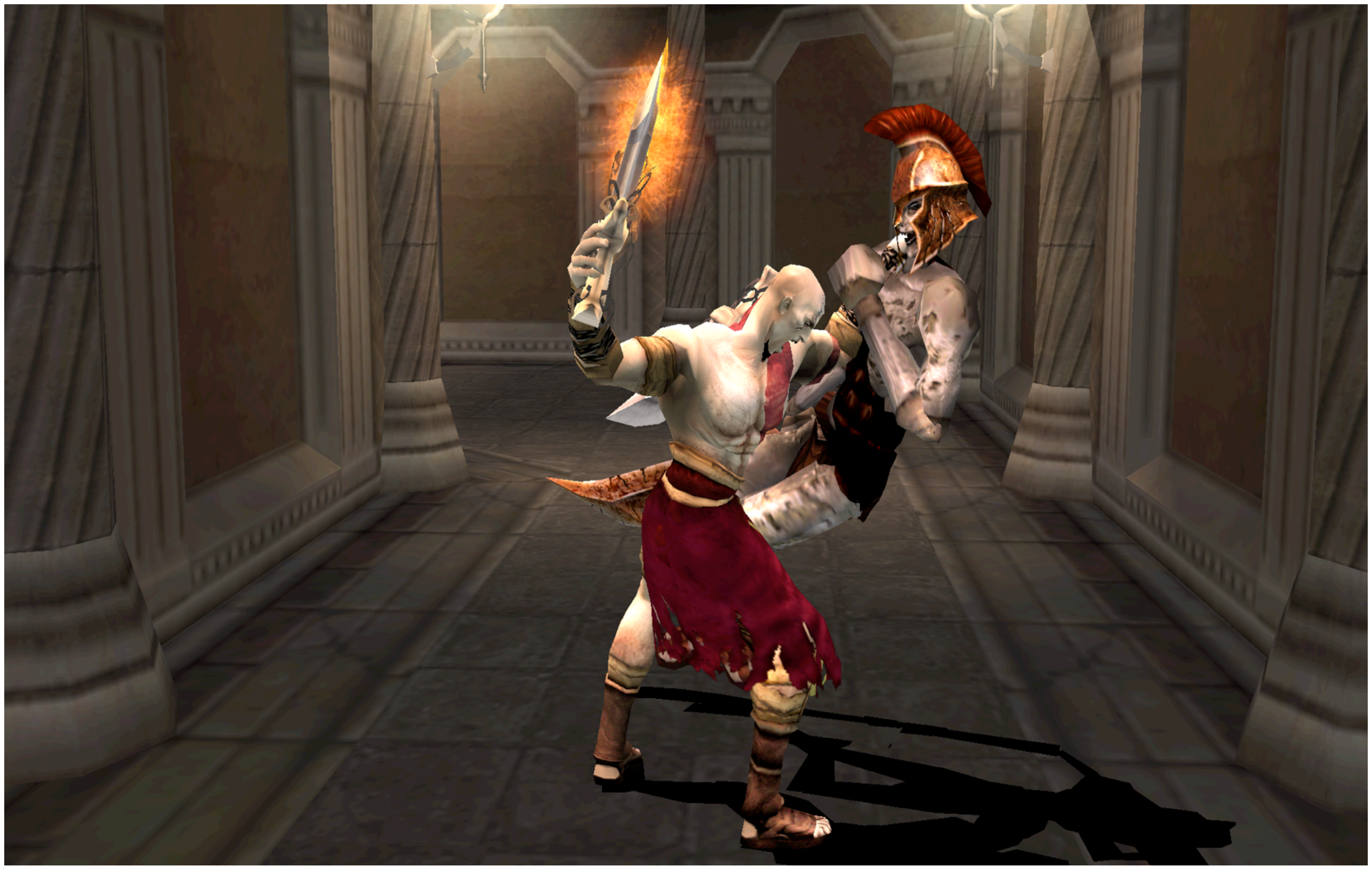
So why do we have a Camera, and a Target?

## Combat Camera : Lock



So here’s the thing, if we know we’re in an open arena, with plenty of clearance, and we’re confident that the camera animation played from any position isn’t going to go flying through the environment, then we’re OK. And sometimes we are.

## Combat Camera : Lock



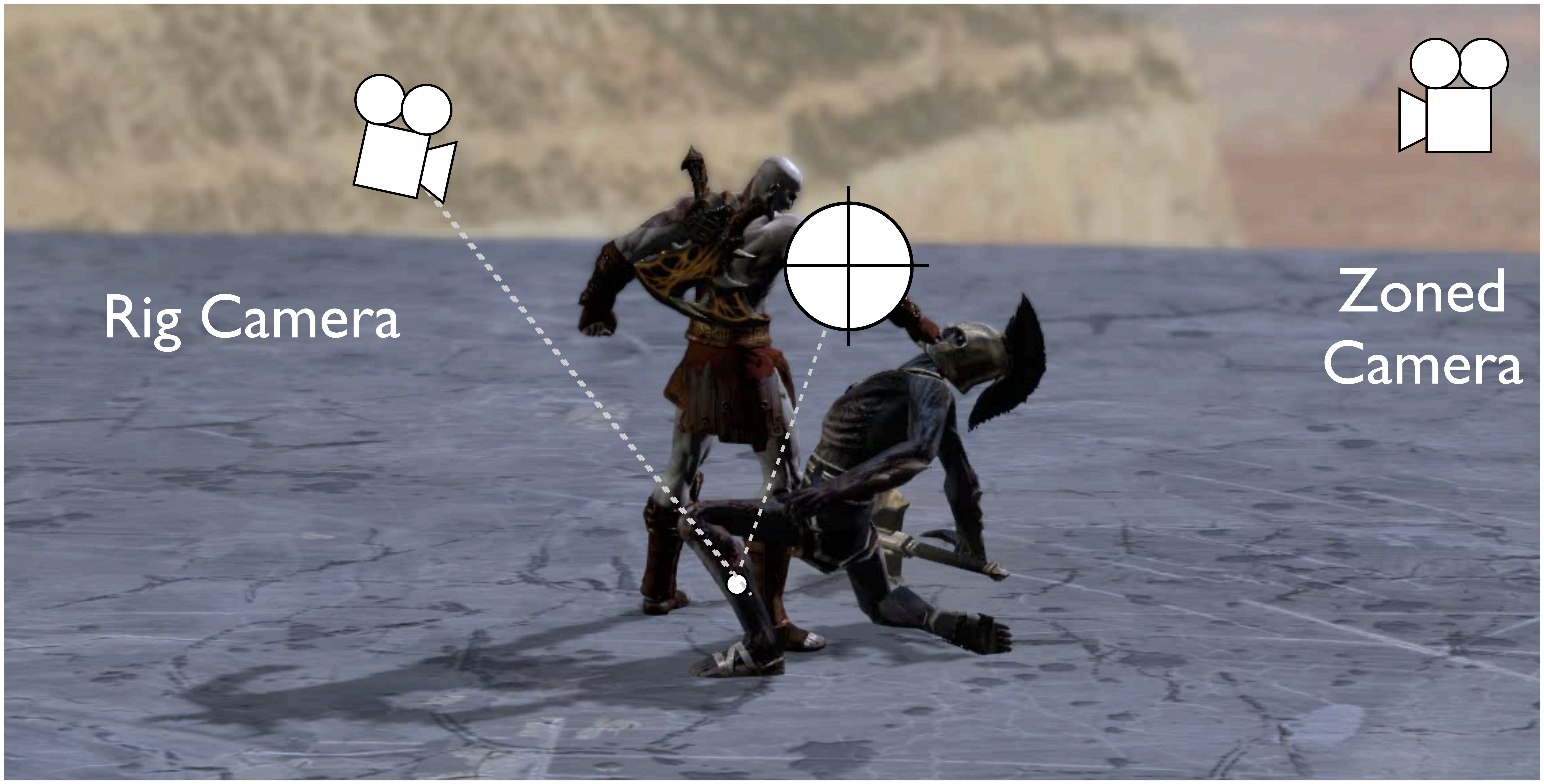
Except of course, in most environments we're not, and there's a very good chance that the camera is going to end up in the middle of a column, or behind a wall, and that's not good. We made a decision not to do any collision on the camera. It's very difficult to do camera collision well, you get all sorts of jolts to the camera.

## Combat Camera : Lock

- Position taken from underlying environmental camera
- Rotate camera to point at target
- Angle of view calculated to get same framing as rig camera would have had

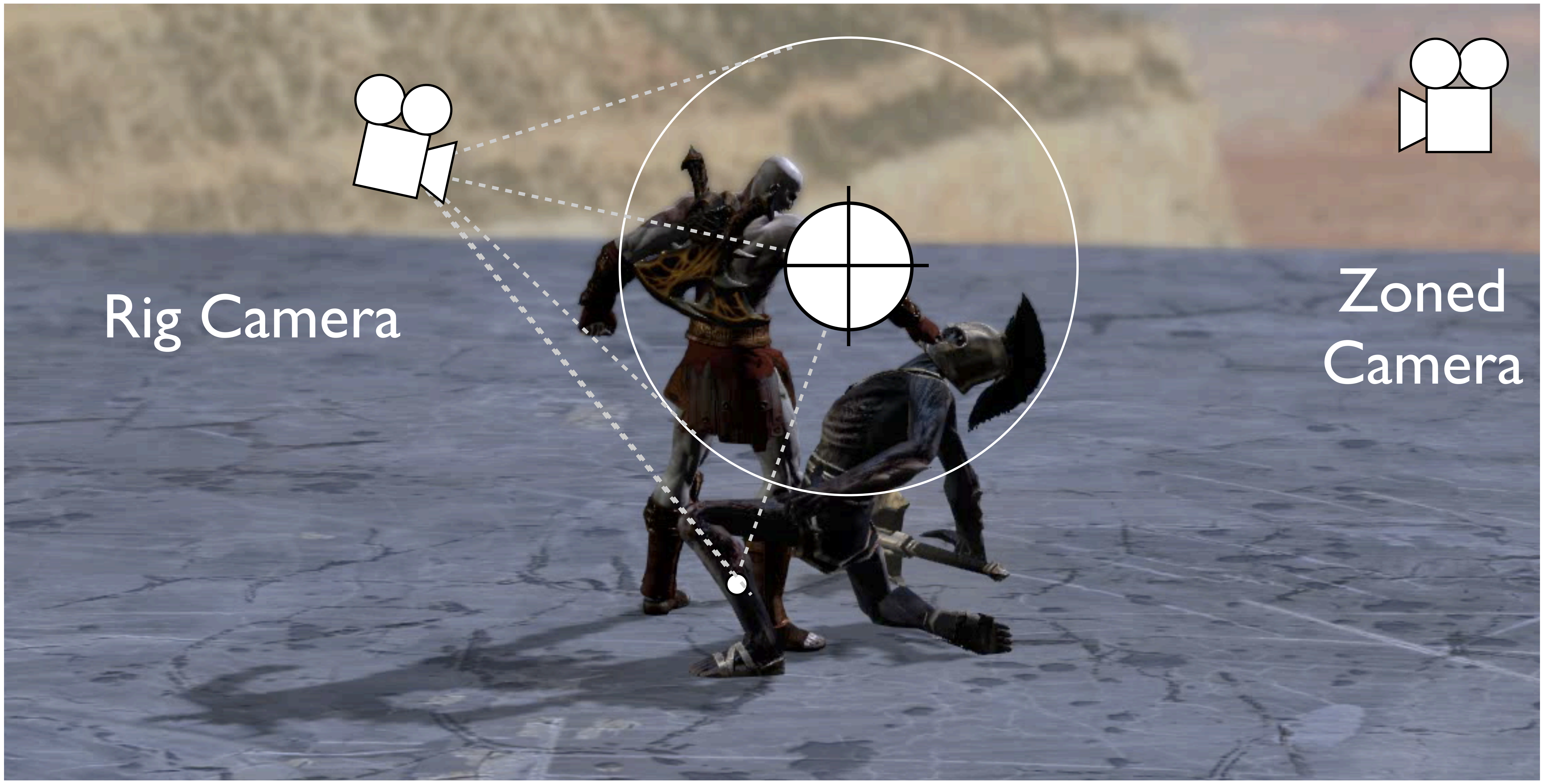
So what we do, when we're not confident of the environment is use the position of the environmental camera we're blending the combat camera over. That camera will have been designed not to intersect the environment, so we know the position is safe.
We then take the position of the rig target, and rotate the camera to face it.
Then we set fov such the the target appears at the same size it would have done if we were using the camera on the rig directly.

# Combat Camera : Lock



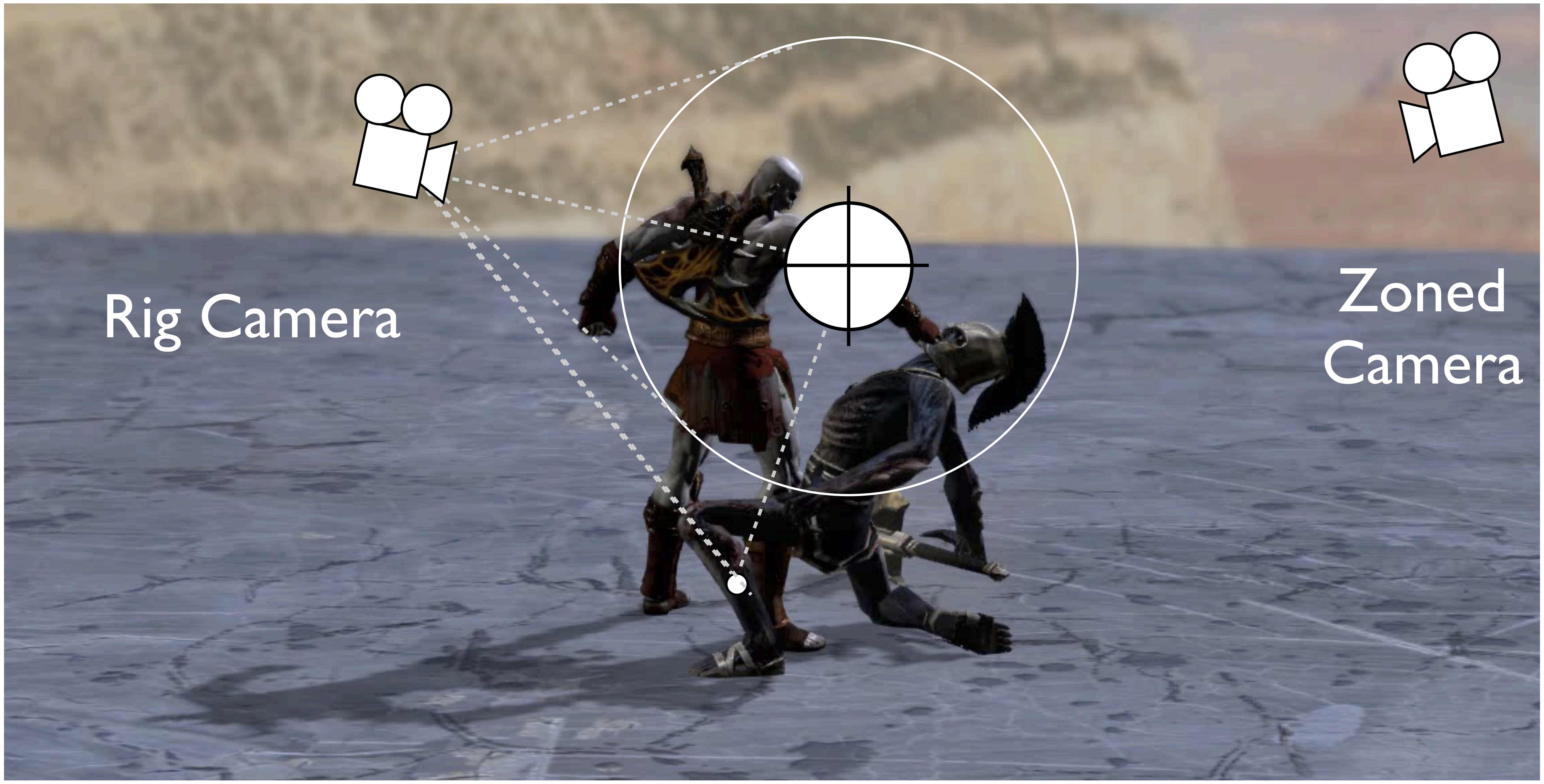
You can visualise this by imagining a sphere that encompasses the area of interest. It’s defined by the target point, the camera, and the fov.
*
We then make the zoned camera frame that sphere by rotating it to point at the center,
*
and bringing it’s fov in to fit the sphere

# Combat Camera : Lock



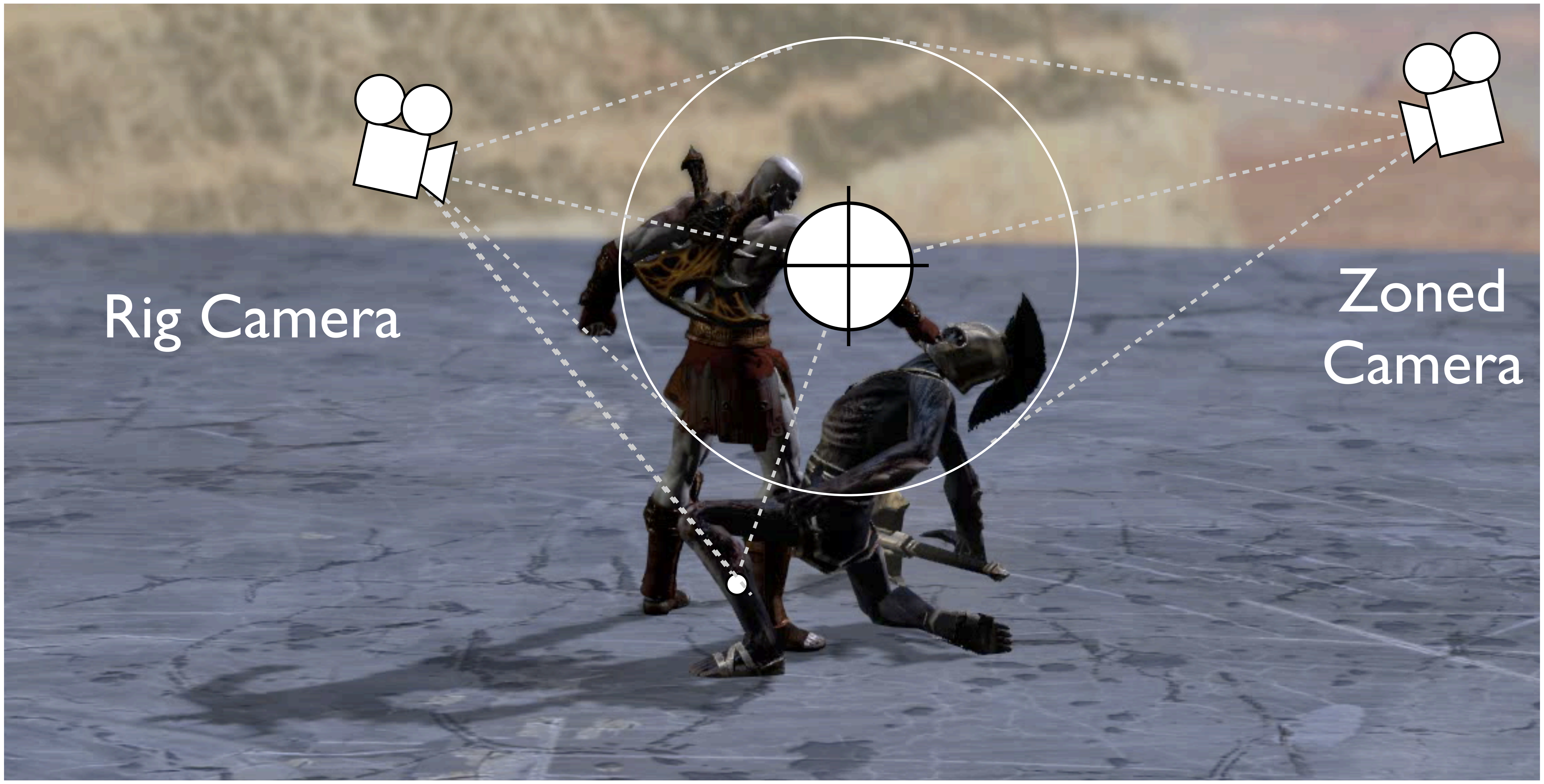
You can visualise this by imagining a sphere that encompasses the area of interest. It’s defined by the target point, the camera, and the fov.
*
We then make the zoned camera frame that sphere by rotating it to point at the center,
*
and bringing it’s fov in to fit the sphere

# Combat Camera : Lock



You can visualise this by imagining a sphere that encompasses the area of interest. It's defined by the target point, the camera, and the fov.
*
We then make the zoned camera frame that sphere by rotating it to point at the center,
*
and bringing it's fov in to fit the sphere

# Combat Camera : Lock



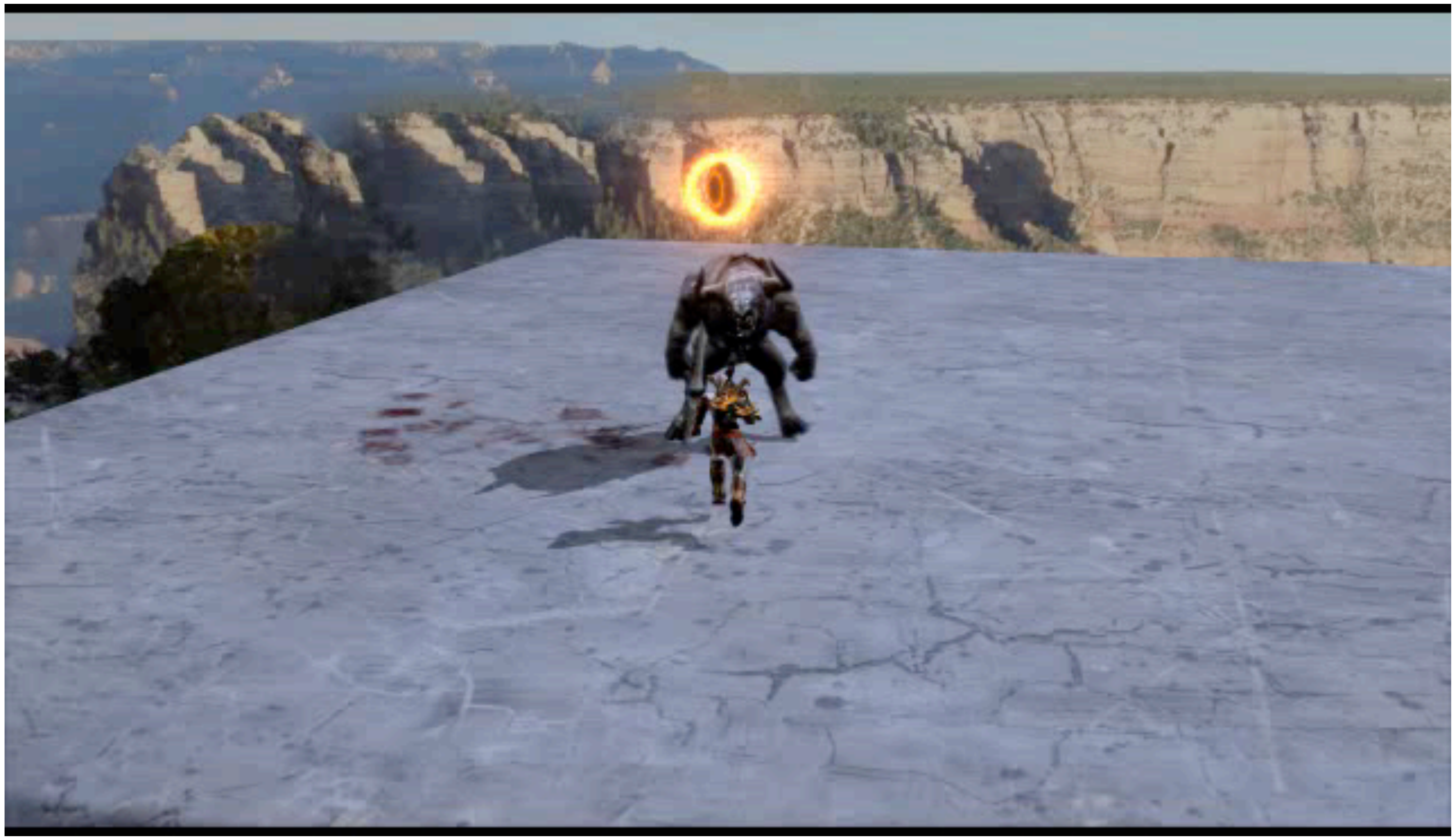
You can visualise this by imagining a sphere that encompasses the area of interest. It’s defined by the target point, the camera, and the fov.
*
We then make the zoned camera frame that sphere by rotating it to point at the center,
*
and bringing it’s fov in to fit the sphere

## Combat Camera : Rotation

- Rotate creature (and thus rig) such that environmental camera has same yaw as rig camera.
- Animate multiple cameras from different angles.
- Pick closest to minimise initial rotation.

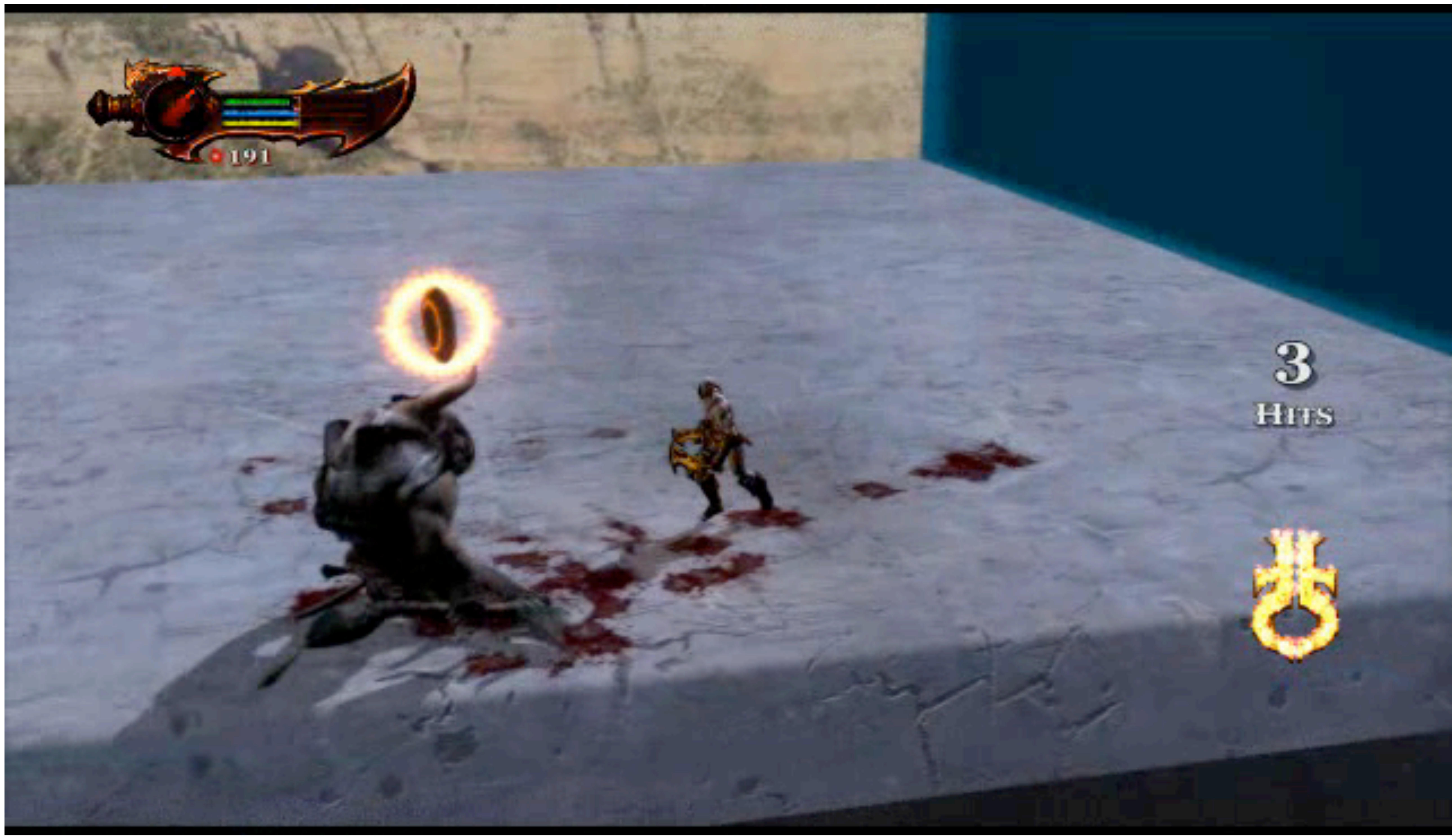
On top of that, we get some of the positional information back, by rotating the target creatures around the up axis, to match the camera animation.
We tween this initial rotation in over time, which can look odd, especially for large angles.
So in order to minimise this initial rotation, we author cameras from multiple angles, and pick the closest one.
Generally they author four, at rough compass points. Sometimes merging them into one, under animation control.

## Combat Camera : Rotation



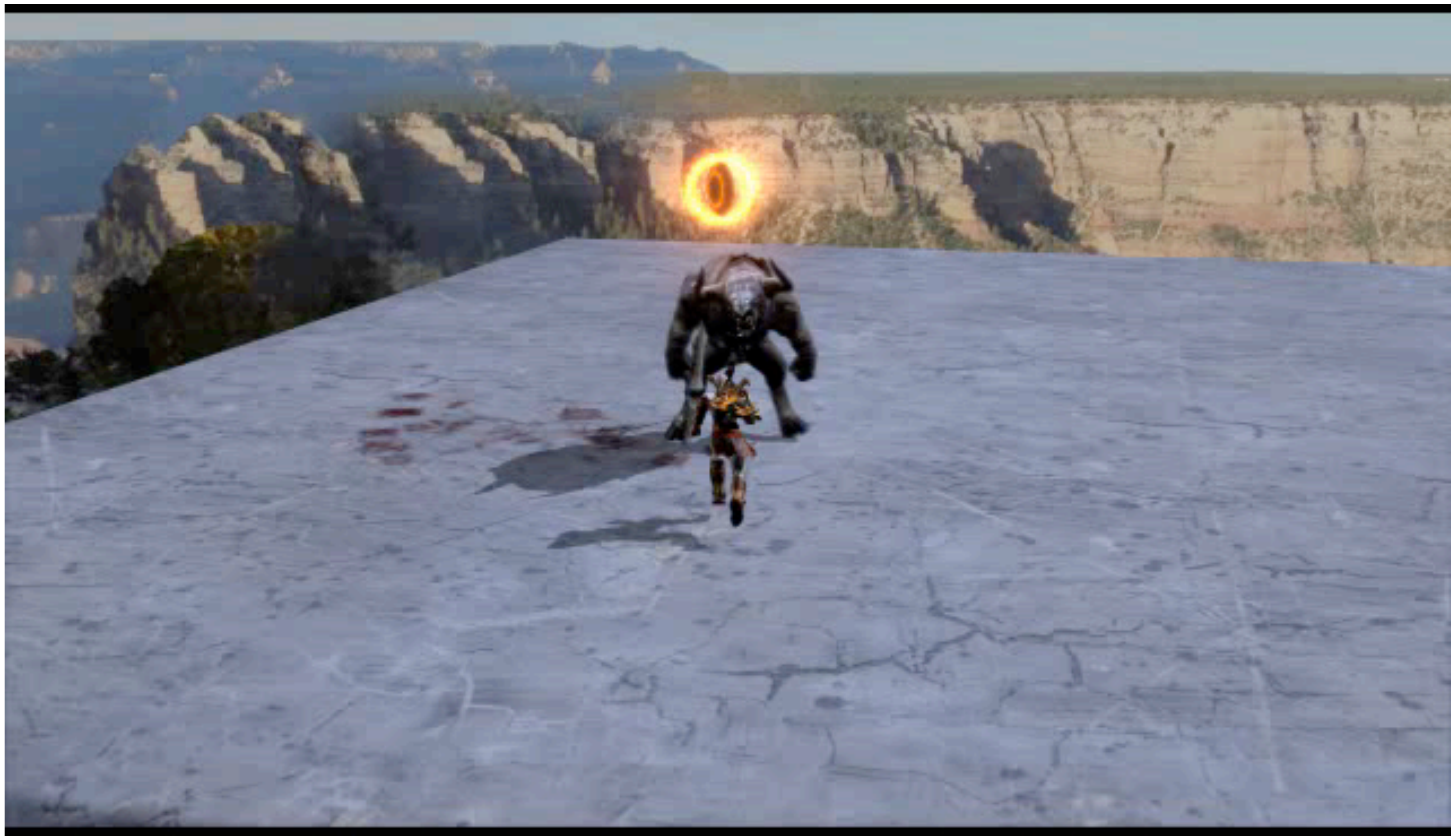
So here’s an example. When we tackle the minotaur head on, you can’t see any rotation.

# Combat Camera : Rotation



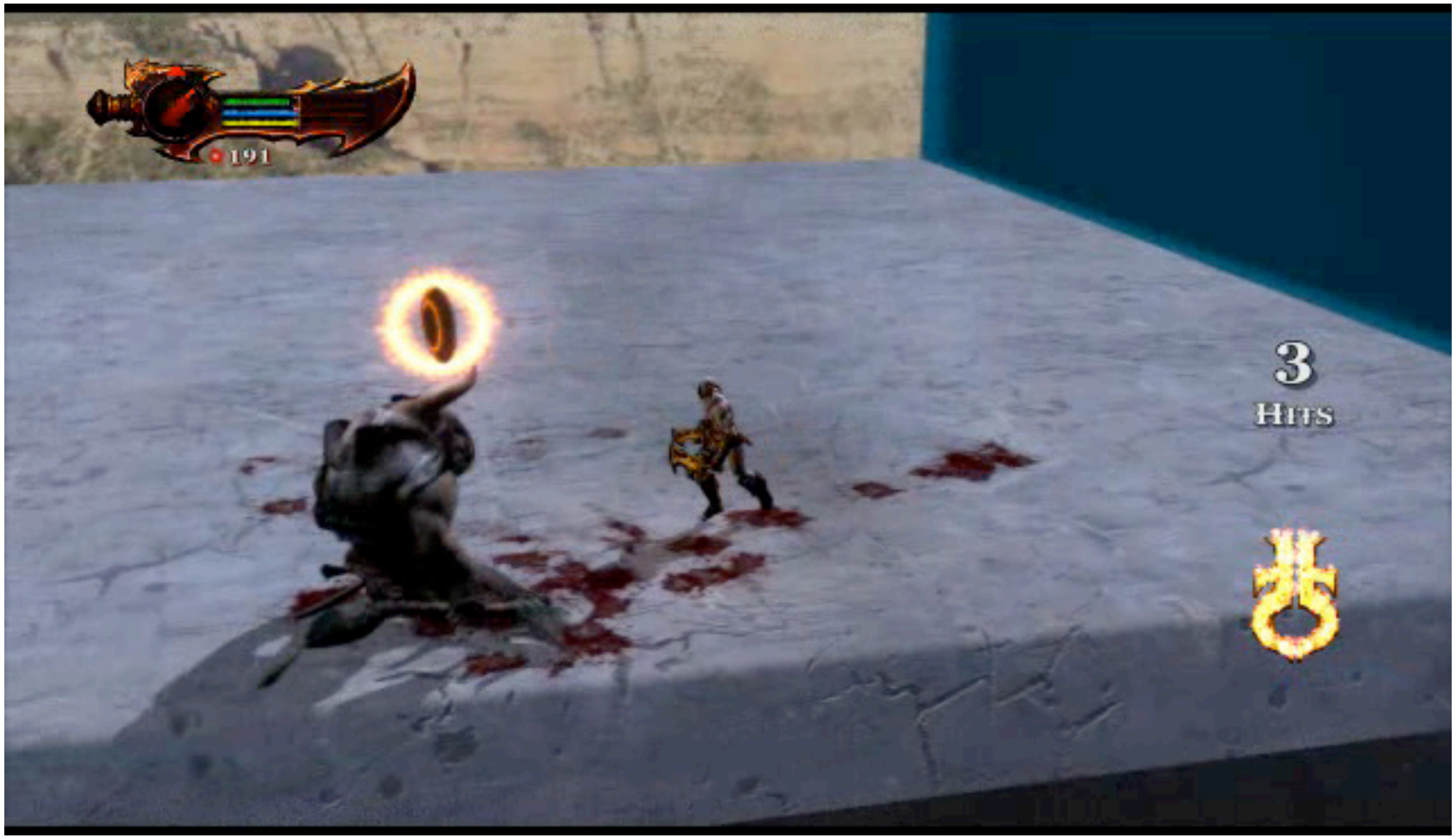
But when we tackle him from the side, you can see the rotation between the head slam, and the minigame where you’re twisting the horns.

## Combat Camera : Rotation



OK, one more time, from head on.

## Combat Camera : Rotation



And from the side

Now you might be able to spot some foot sliding in there, but the animators have hidden it by having Kratos take some steps at that point in the move.

## Combat Camera

- Rig is a creature synched to creature the triggered it
- Animation synched to target creature
- Made up of synch joint, camera, and target
- Fast version has no rig, and uses hero target
- Optionally take position from underlying camera
- Optionally rotate creatures from camera yaw animation

So that's the combat camera. There's also a minimal version that doesn't use a rig, but instead uses the hero's target, and a width defined in the triggering action.

Again, both the creature rotation and position override are optional features. Often a boss is constrained to a specific area, so if the designers are confident that there will always be enough space around a character, they can turn position override off, and move the camera however they want. An unexpected side effect of this, was that many of our cutscenes in 3 were actually authored as combat cameras synched to one of the protagonists.

## Overview

| | |
|---|---|
| • Selection | Environment, Combat, Scripting, Filtering |
| • Blending | Blend Tree, Weights, Modes, Parameters |
| • Dynamics | Animated, Dynamic, Combat |
| • Targeting | Hero, Creatures, Damping, Weighting, Prioritisation |

So that's selection, blending, and dynamics. Now lets move on to Targeting, or how we decide what we're actually looking at.

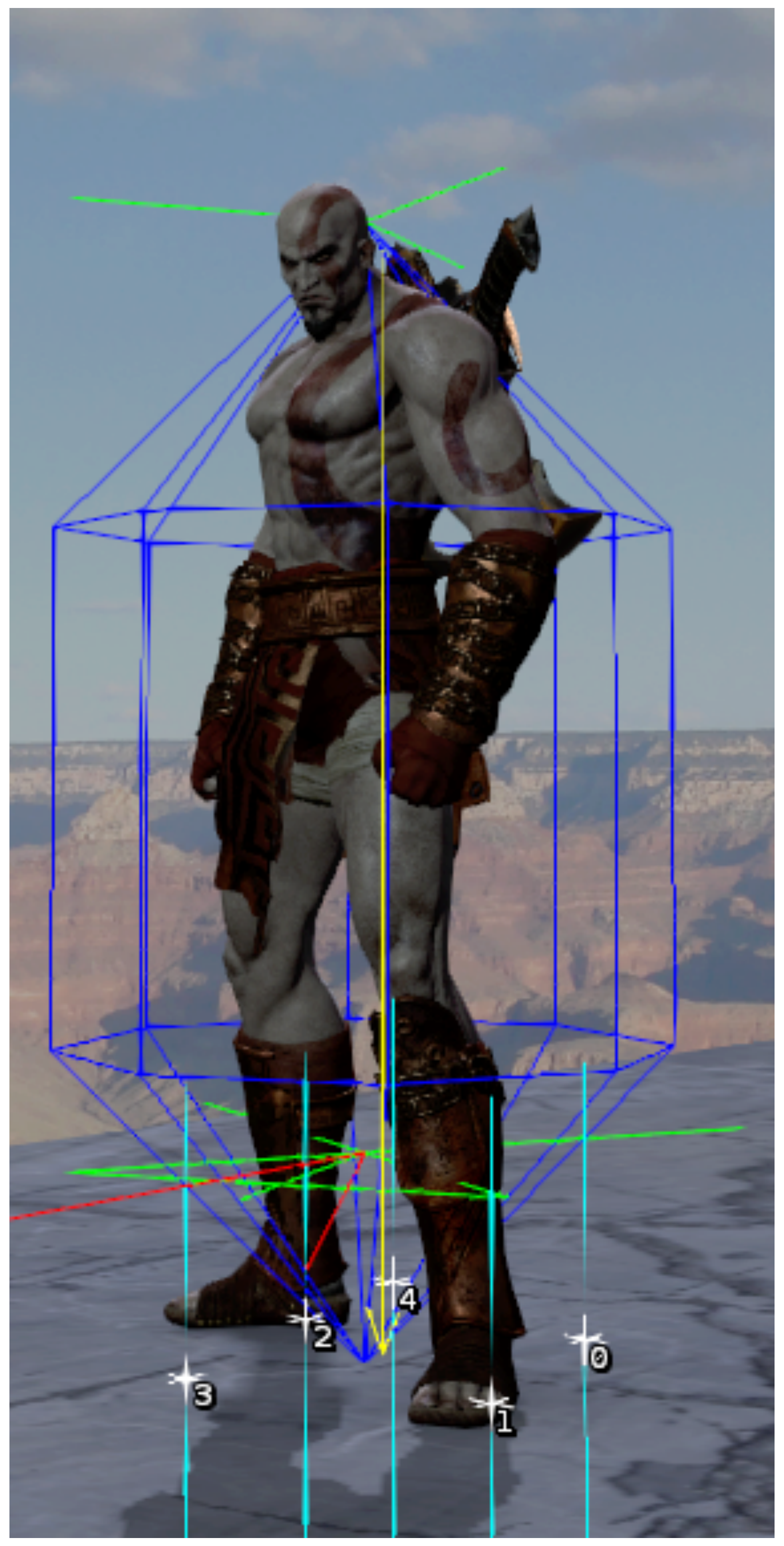
## Targeting : Hero

- Logical position on ground, between feet
- Collision capsule extends 2.25m up to head
- Target is a single point 1.5m above base
- No damping on hero target specifically
- All damping comes from player logic

So here’s our hero, the players avatar, puppet of the Gods, and all round moody bloke.

## Targeting : Hero

- Logical position on ground, between feet
- Collision capsule extends 2.25m up to head
- Target is a single point 1.5m above base
- No damping on hero target specifically
- All damping comes from player logic

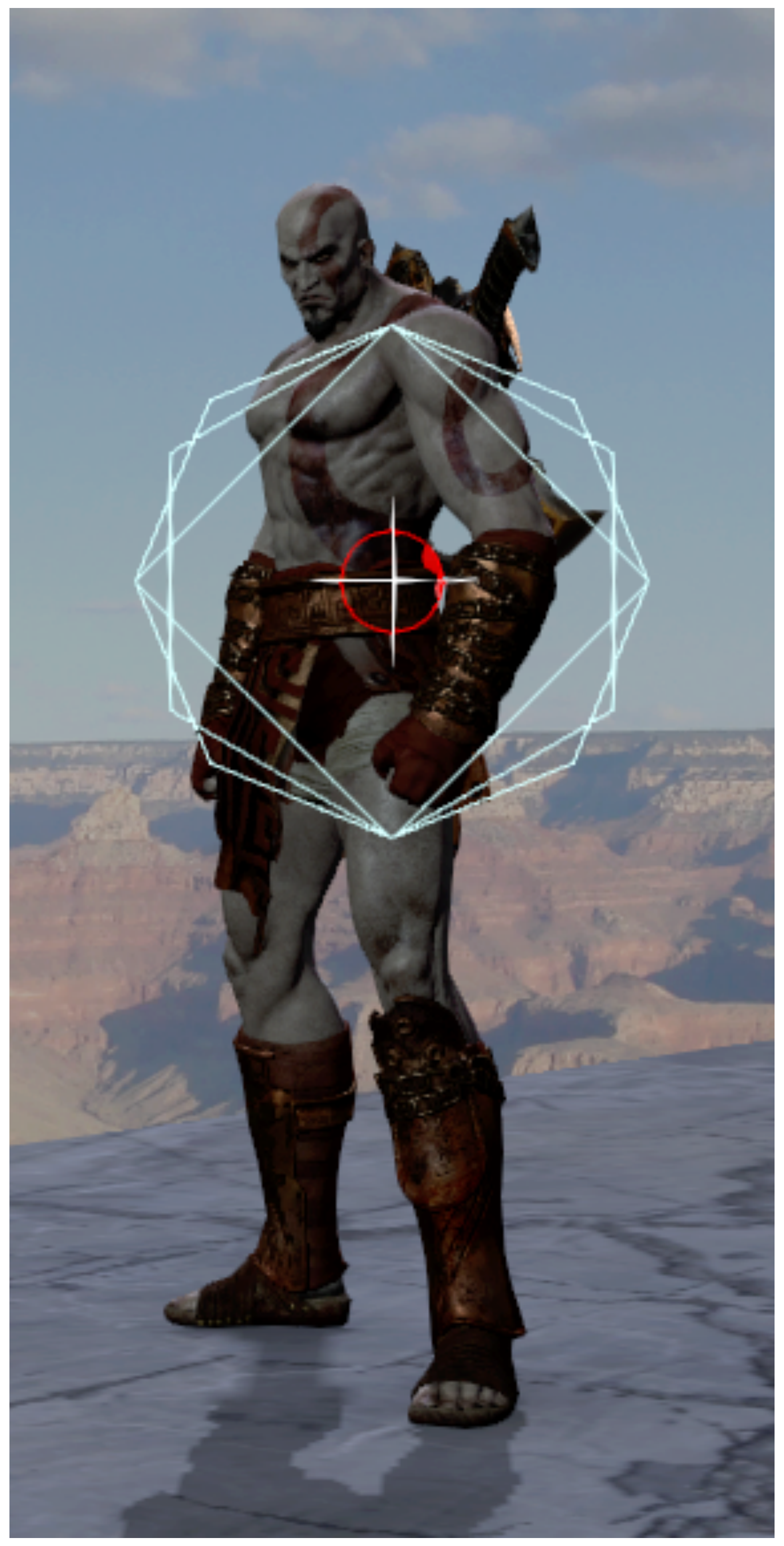


As far as the game is concerned, he starts at a logical position between his feet, and extends 2.25m straight up to the top of his head.

There's a collision capsule that stretches between these two points.

## Targeting : Hero

- Logical position on ground, between feet
- Collision capsule extends 2.25m up to head
- Target is a single point 1.5m above base
- No damping on hero target specifically
- All damping comes from player logic



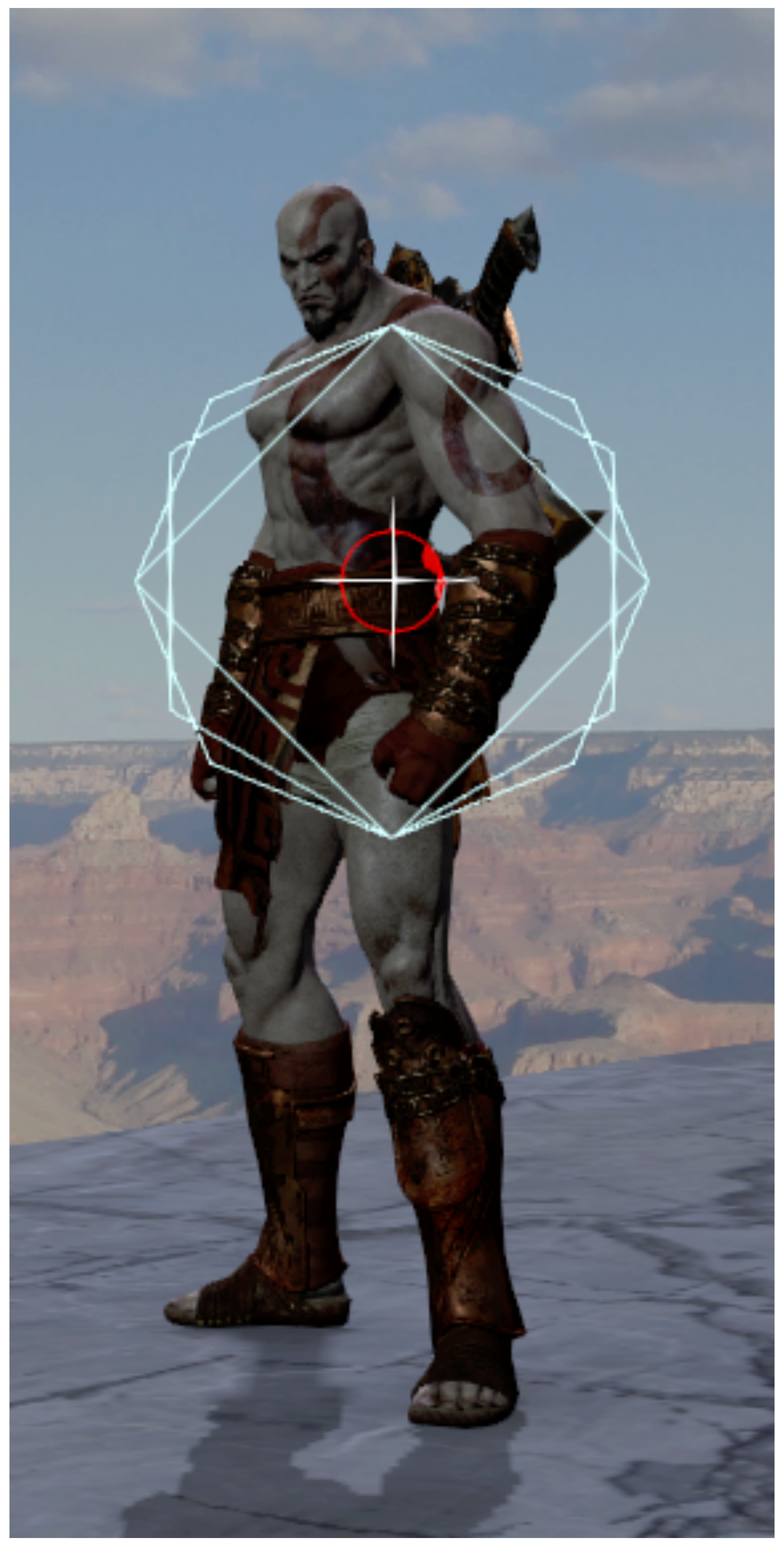
And about 1.5m up that capsule, at gut height, there’s the camera target.

There’s no explicit damping on the hero target. All the damping comes from the player logic.

This ensures that the camera is always tight on the hero, and he doesn’t lag or lead the desired framing too much.

## Targeting : Hero

- Jump Correction
- Optional, per camera
- Ignores the effect of jumping
- Lands higher that he started, tween up
- Falls past corrected height, resume tracking

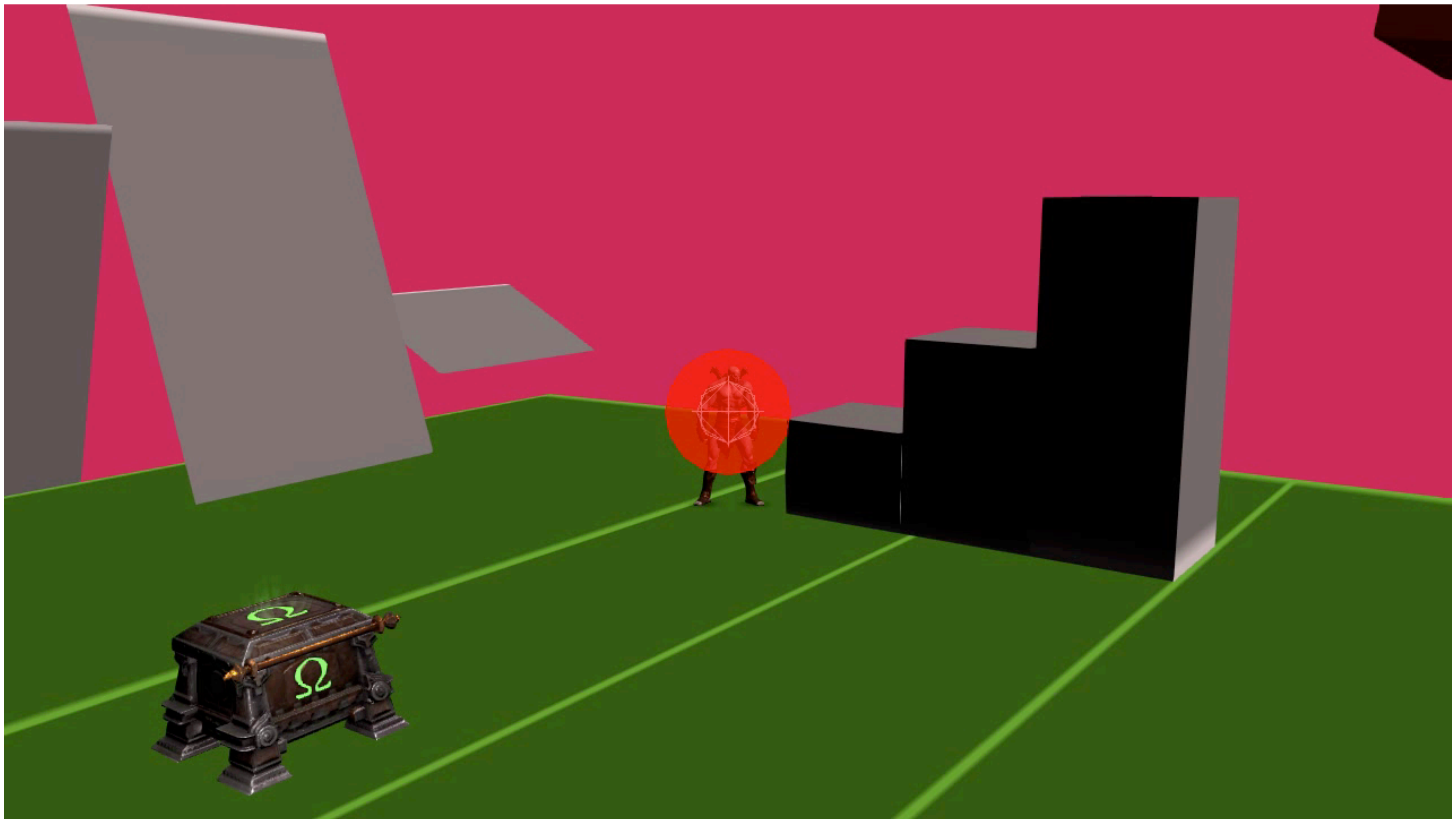


Now Kratos jumps around a lot, and sometime we don’t want that to drag the camera up and down. So there’s an optional system called Jump Correction that can be enabled on a per-camera basis that allows us to ignore the effect of jumping.

If he lands on a higher surface than he started on, we tween the target up to meet him.

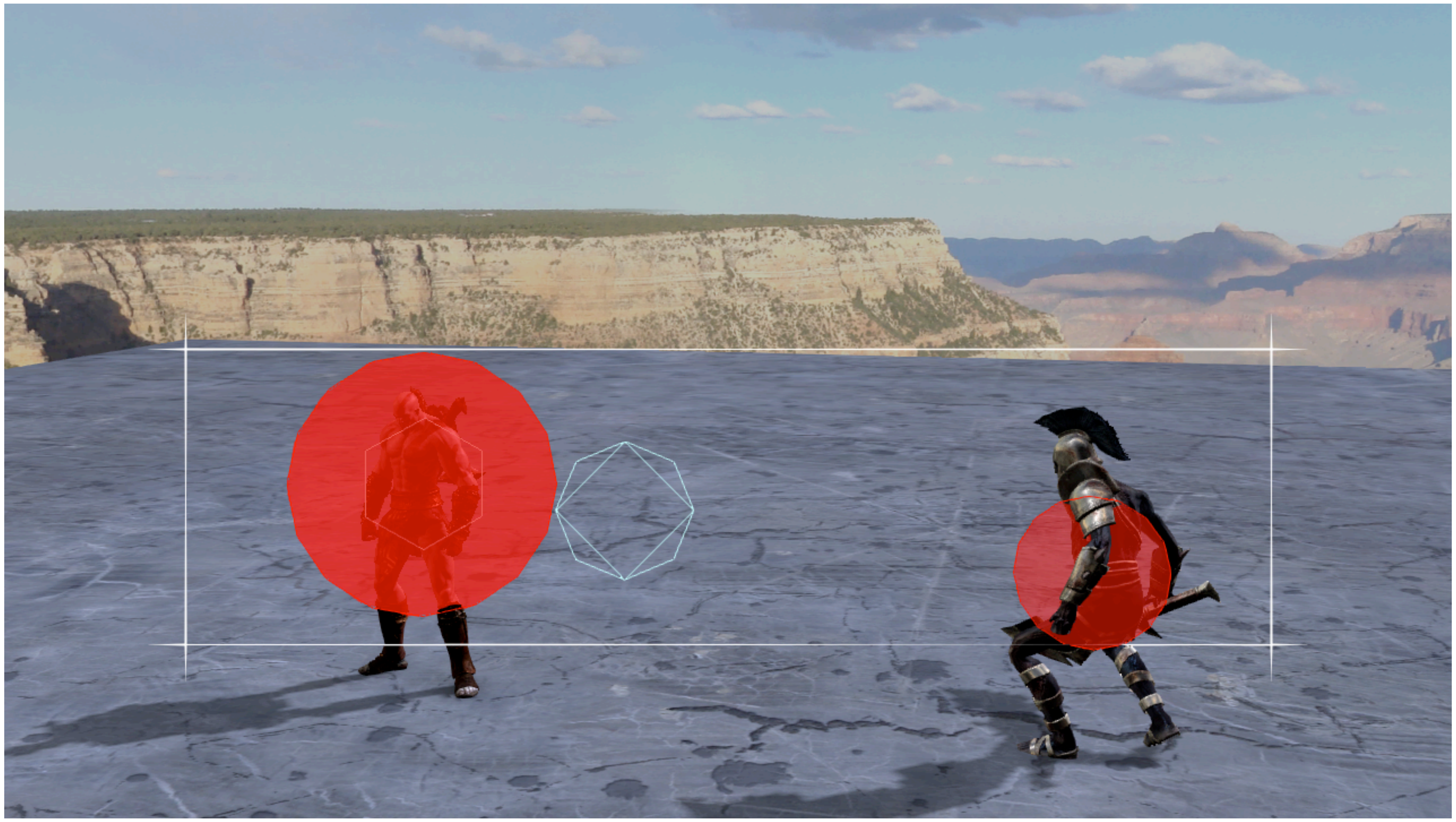
And if he falls below the starting height, we resume tracking his vertical position.

## Targeting : Hero



So lets look at that in action.

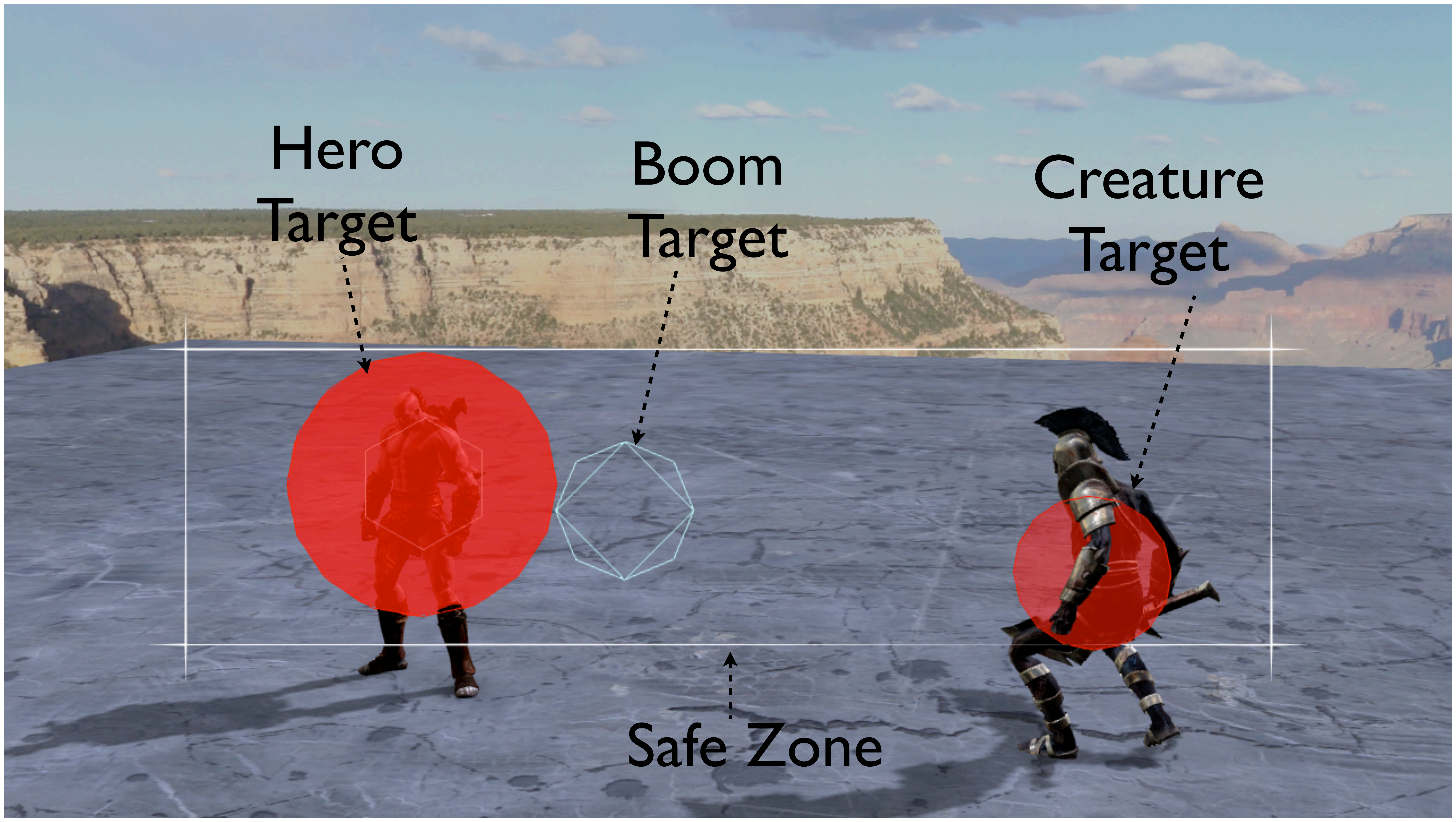
## Targeting : Creatures



And that’s pretty much all we did for targeting for the first God of War. So for the sequel, we naturally had to improve on this. And the obvious thing to do was to support more than one target. Ideally to have a target on everything of interest, and somehow take them all into account when constructing a camera.

* Now this next sections going to have a few videos set in the test level. So I should explain the debug info you can see here. The big red sphere on the left over Kratos is the Hero target, the smaller one on the right over the Grunt, is a creature target. The diamond just to the left of centre is the Boom Target, this is the point we use for calculating the Boom for the dynamic camera. And the big white rectangle is the safe zone we use for the framing constraint.

# Targeting : Creatures

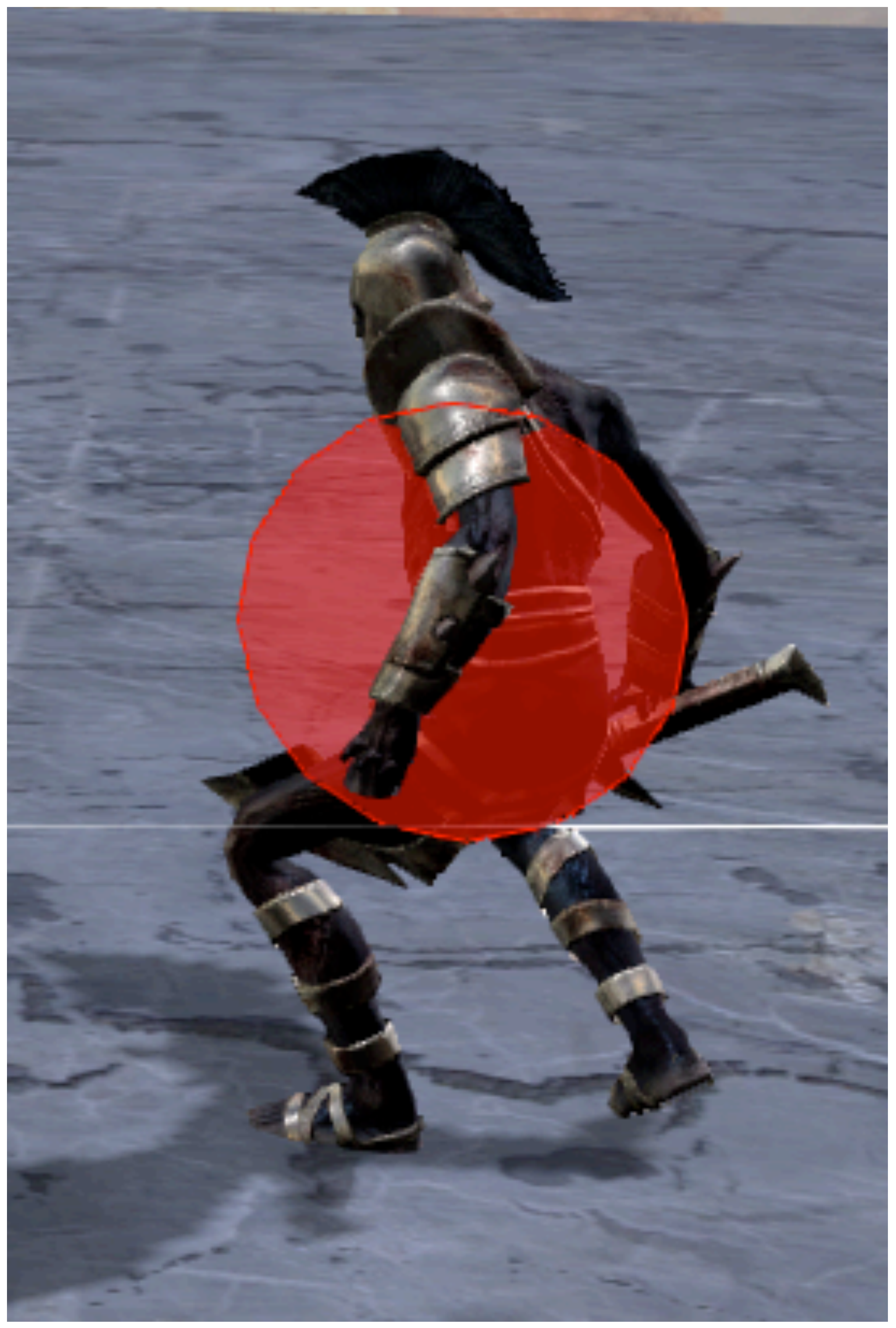


And that's pretty much all we did for targeting for the first God of War. So for the sequel, we naturally had to improve on this. And the obvious thing to do was to support more than one target. Ideally to have a target on everything of interest, and somehow take them all into account when constructing a camera.

* Now this next sections going to have a few videos set in the test level. So I should explain the debug info you can see here. The big red sphere on the left over Kratos is the Hero target, the smaller one on the right over the Grunt, is a creature target. The diamond just to the left of centre is the Boom Target, this is the point we use for calculating the Boom for the dynamic camera. And the big white rectangle is the safe zone we use for the framing constraint.

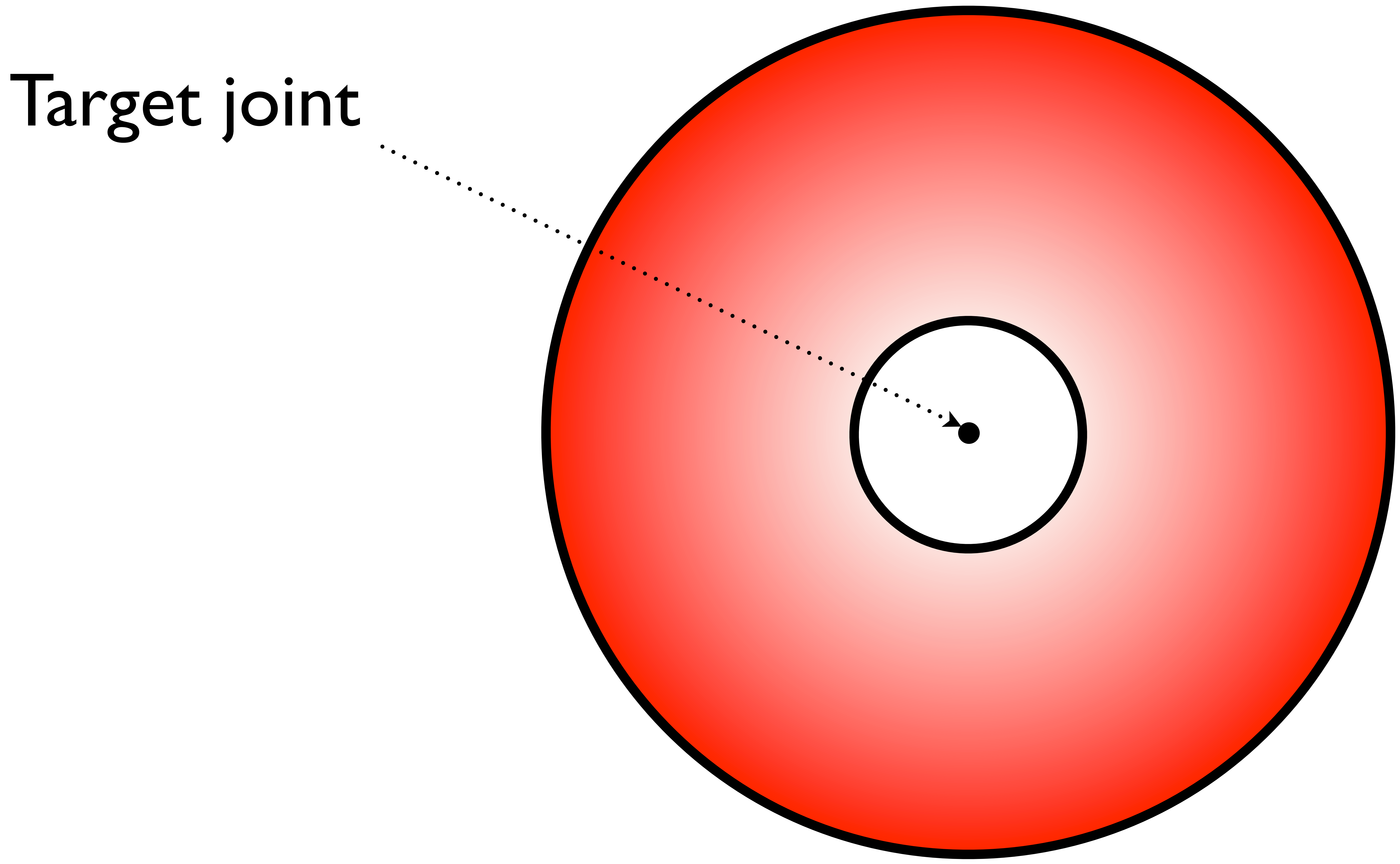
## Targeting : Creatures

- Is a sphere, not just a point
- Weight and priority define influence
- Can be attached to a joint
- Needs to be damped



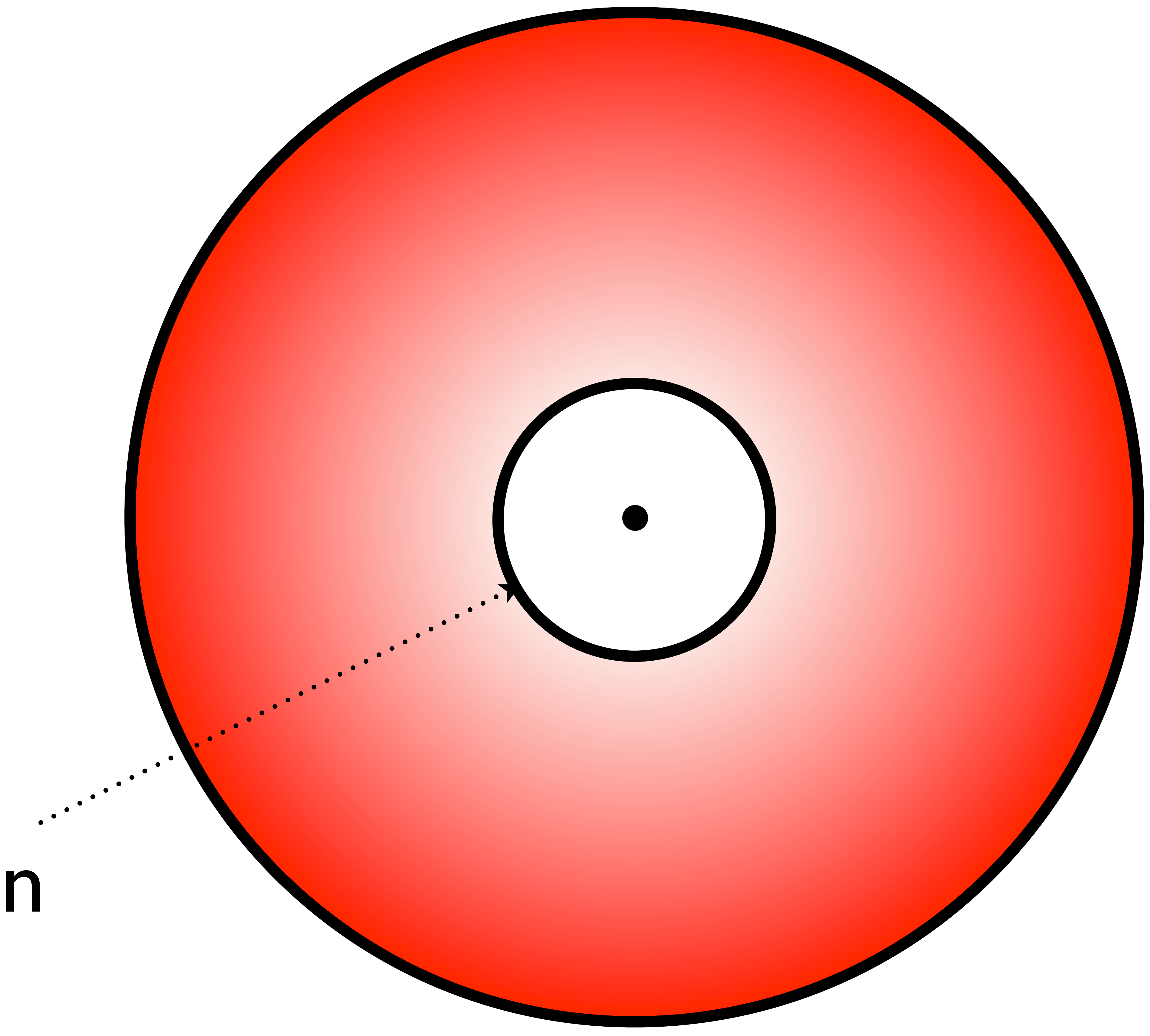
So how is a creature target different from the Hero target? Well it's a sphere, rather than just point, and it has a weight and priority that we'll use to define it's influence.
It can also be attached to a joint, which means we'll need to damp it. Recall that for the Hero target we have no damping, as we rely on the player logic to move the capsule smoothly. Animation joints make no such promises, and make movements entirely at the animators whim. Creating enormous potential for undesired motion to be added to any camera that is using that target.
So we need to filter out small movement, so that an idling creature doesn't set the camera oscillating. But we also need to make sure we track large movements, and that we transition between the two states smoothly.

## Targeting : Damping



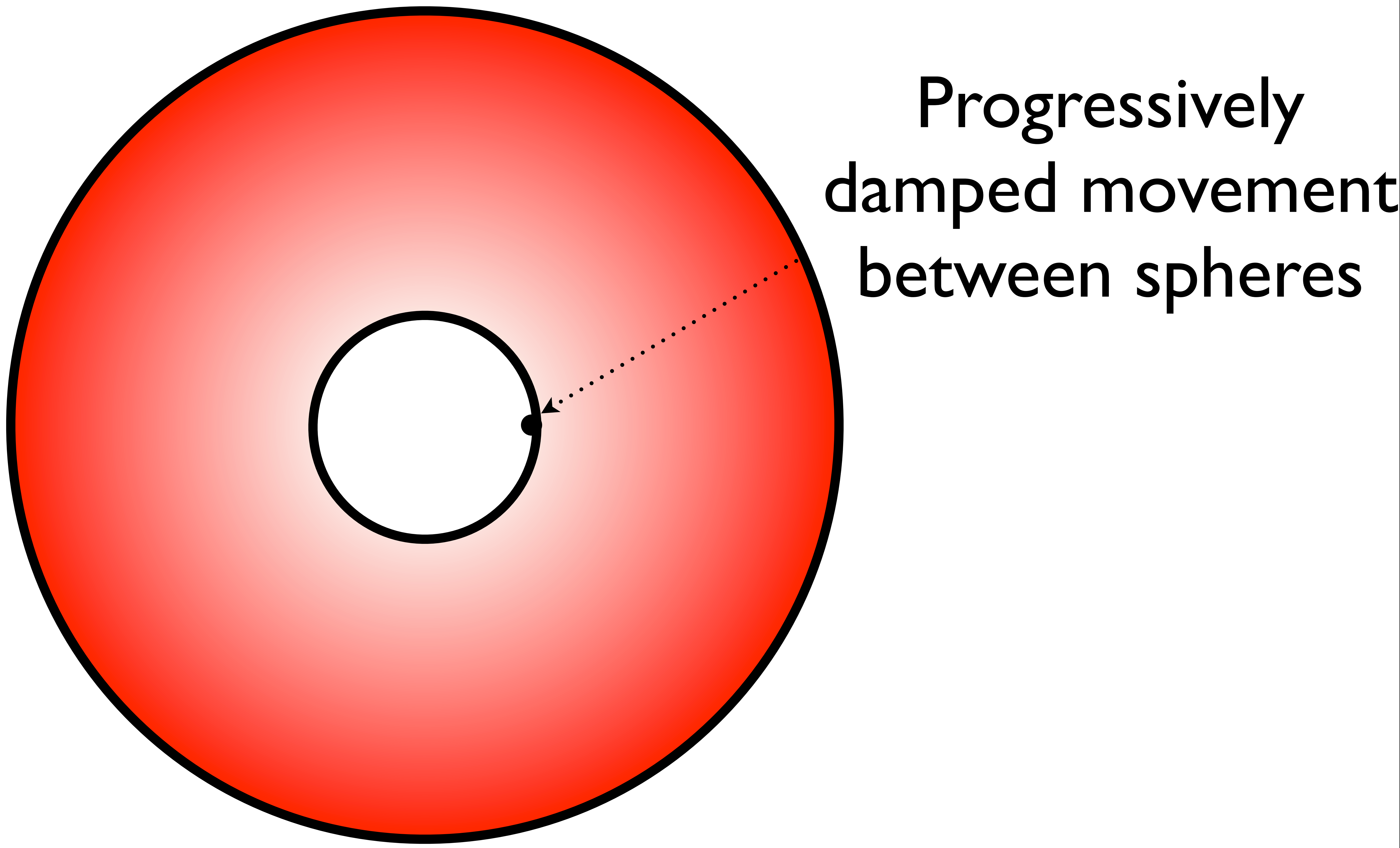
We solve this by wrapping the joint in two spheres.
* There's free movement within the inner sphere.
* Progressively damped movement between the spheres, that eventually returns the joint to the inner sphere.
* And undamped movement at the outer sphere. If the joint has reached, or moved outside of the outer sphere, the target moves instantly to keep the joint inside. This guarantees the target never become detached from the creature.

## Targeting : Damping



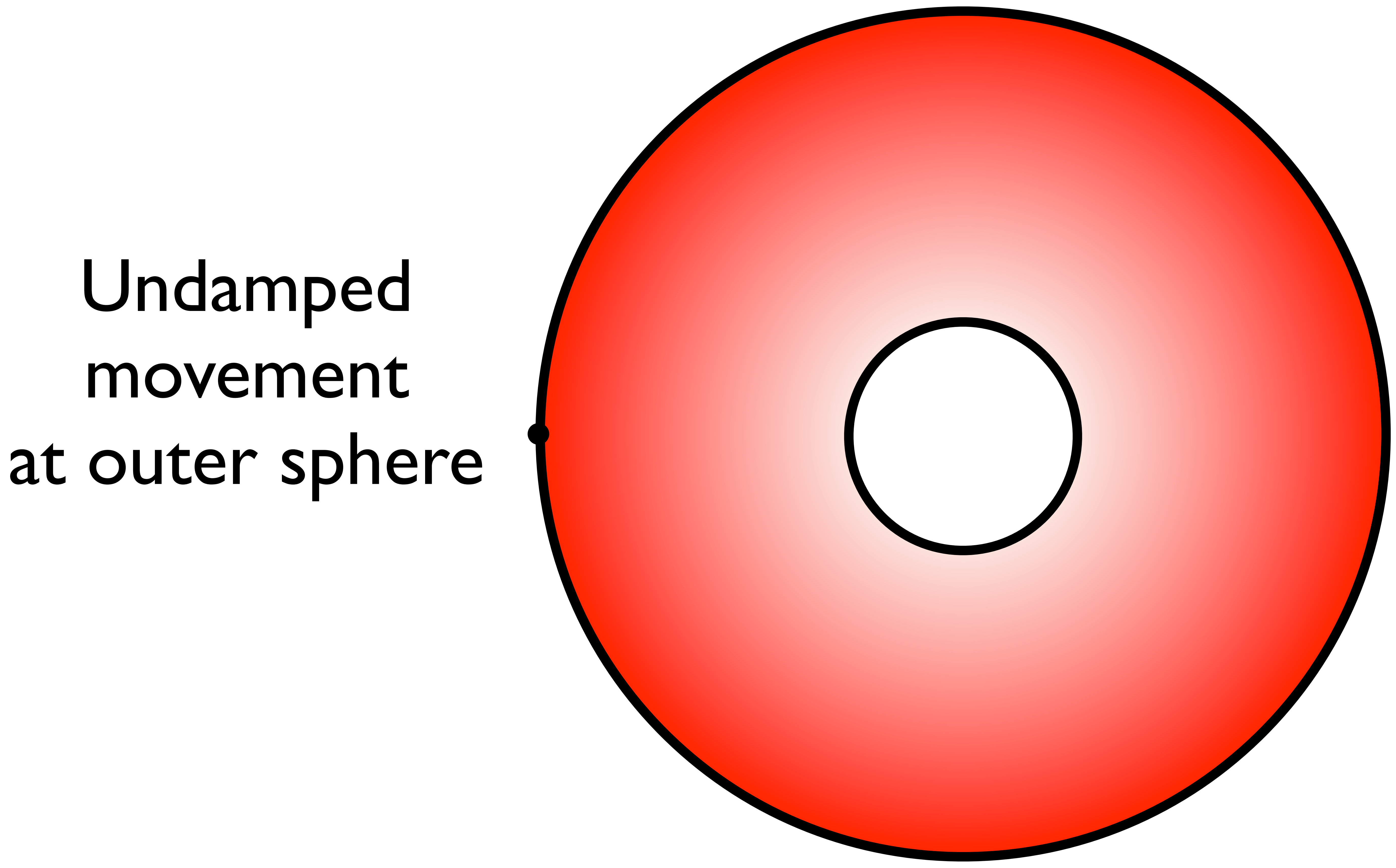
We solve this by wrapping the joint in two spheres.
* There's free movement within the inner sphere.
* Progressively damped movement between the spheres, that eventually returns the joint to the inner sphere.
* And undamped movement at the outer sphere. If the joint has reached, or moved outside of the outer sphere, the target moves instantly to keep the joint inside. This guarantees the target never become detached from the creature.

## Targeting : Damping



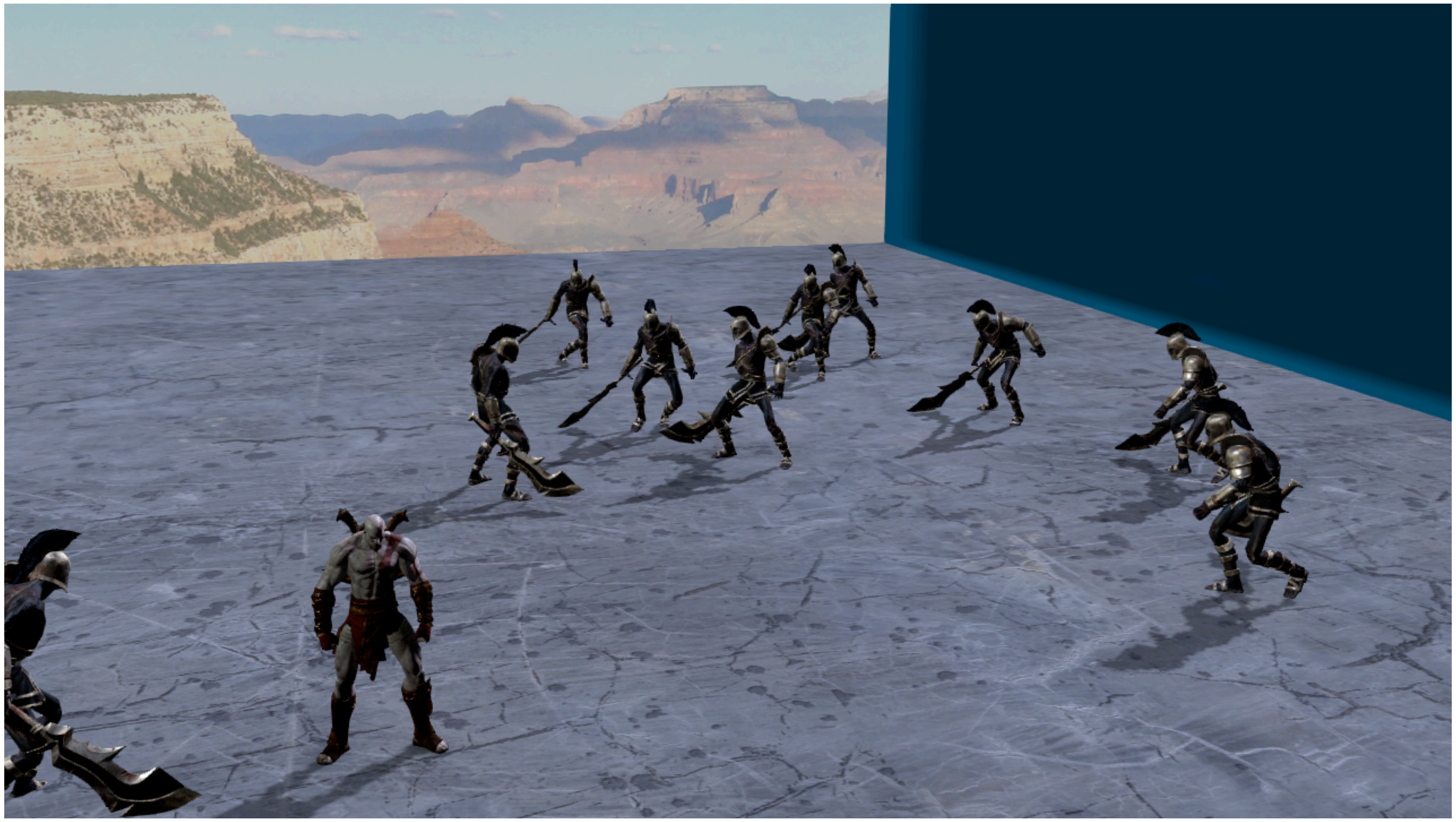
We solve this by wrapping the joint in two spheres.
* There's free movement within the inner sphere.
* Progressively damped movement between the spheres, that eventually returns the joint to the inner sphere.
* And undamped movement at the outer sphere. If the joint has reached, or moved outside of the outer sphere, the target moves instantly to keep the joint inside. This guarantees the target never become detached from the creature.

## Targeting : Damping



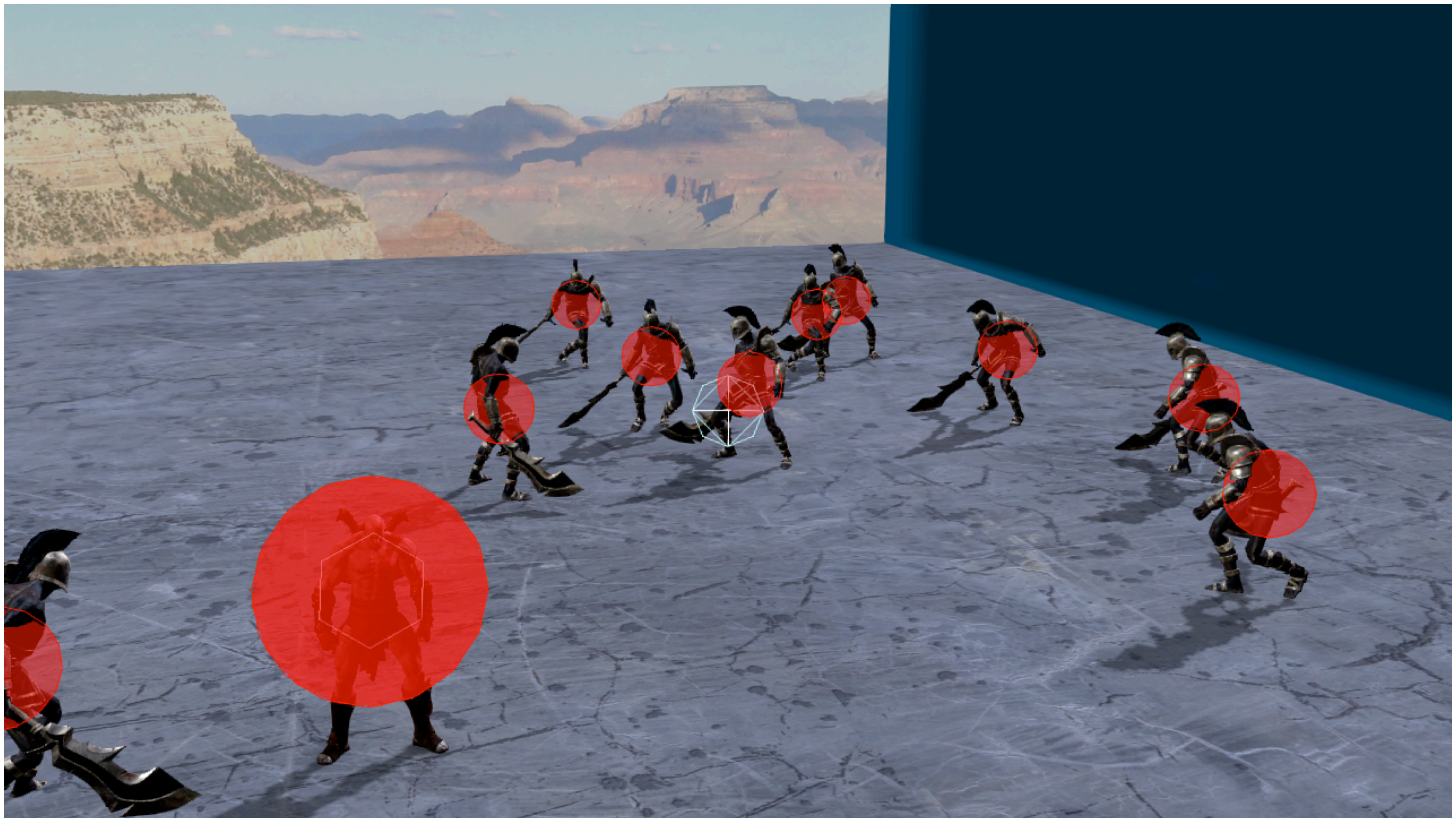
We solve this by wrapping the joint in two spheres.
* There’s free movement within the inner sphere.
* Progressively damped movement between the spheres, that eventually returns the joint to the inner sphere.
* And undamped movement at the outer sphere. If the joint has reached, or moved outside of the outer sphere, the target moves instantly to keep the joint inside. This guarantees the target never become detached from the creature.

## Targeting : Weighted Average



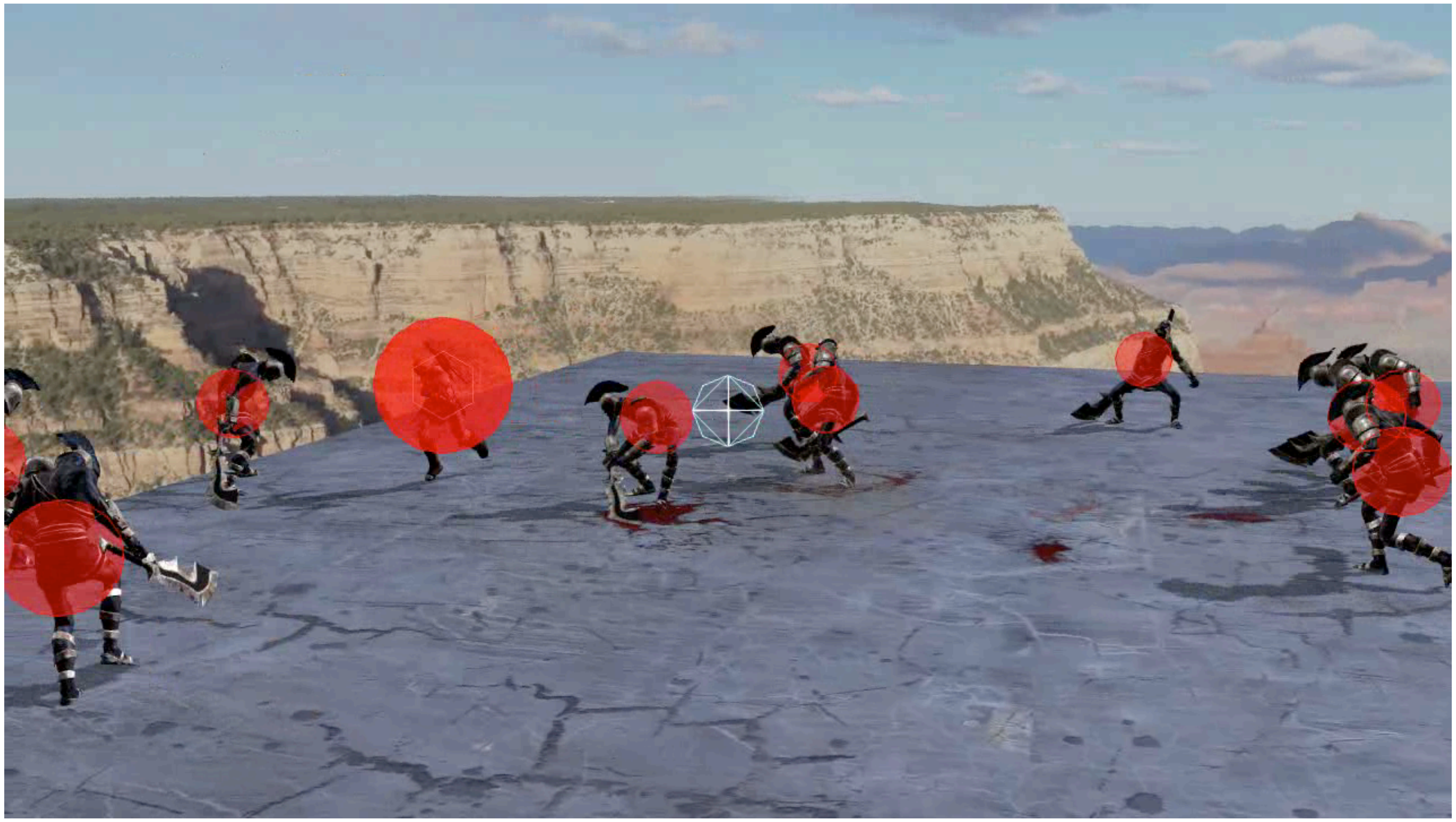
So back to Kratos, and he’s got quite the crowd with him now. Lets look at that from the camera systems perspective

## Targeting : Weighted Average



So now we put targets on every creature in the fight, the easiest thing to do is just average them to produce a boom target we can feed into the dynamic camera.
This helps, and since Kratos is normally somewhere near the middle of the fight, it doesn't usually move him off screen.

## Targeting : Weighted Average



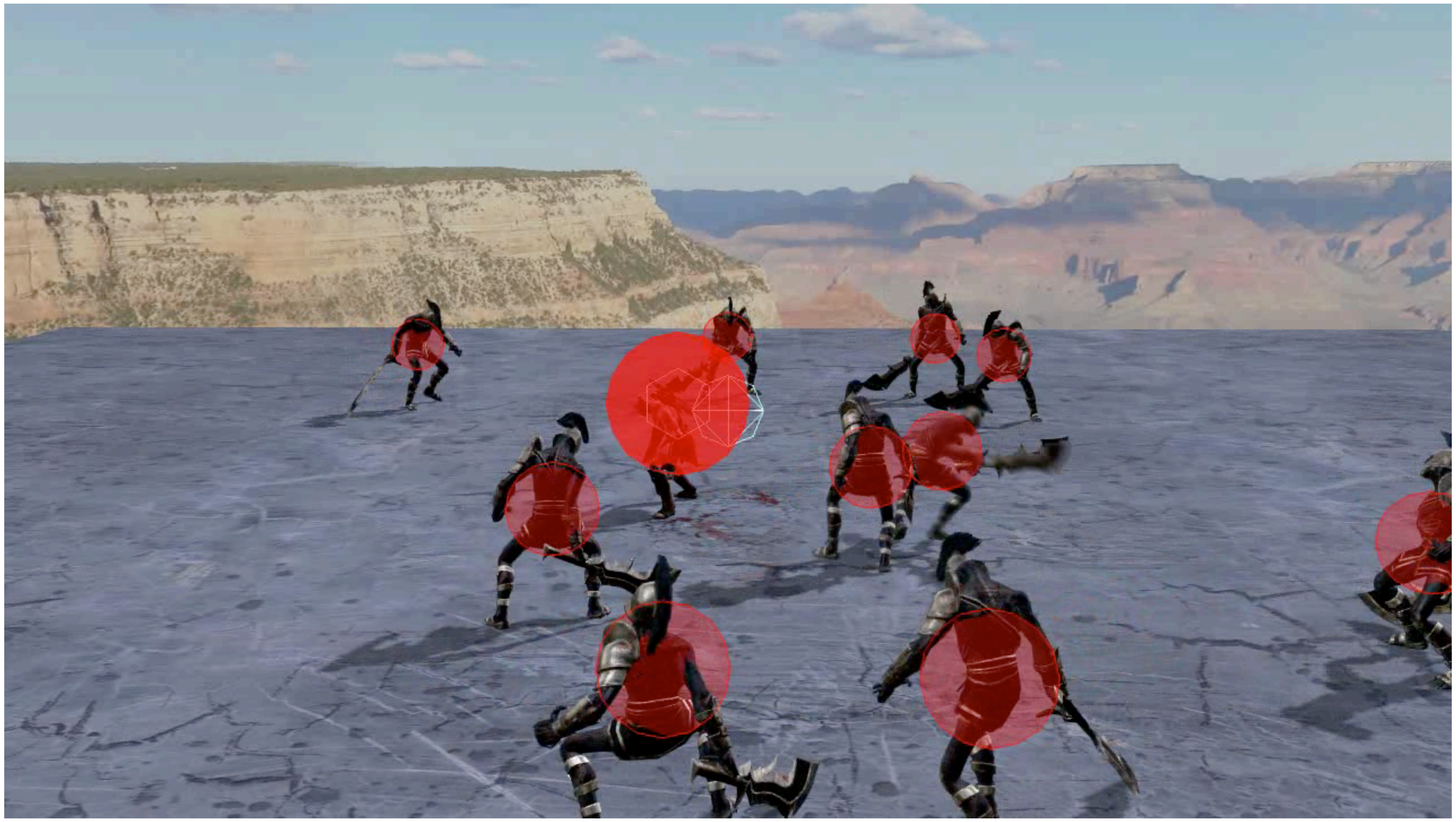
But there's no guarantee that it doesn't. If he's running away from a slow moving creature, then we could quite easily end up with both of them off screen. So what we do, is add a weight to each target, and generate the boom target from a weighted average.

## Targeting : Weighted Average

- Sum of all weighted positions divided by sum of all weights.
- Target weight is product of base, distance, and activation
  - Base weight - importance of target
  - Distance weight - fade out over distance from Hero
  - Activation weight - fade in and out and birth and death

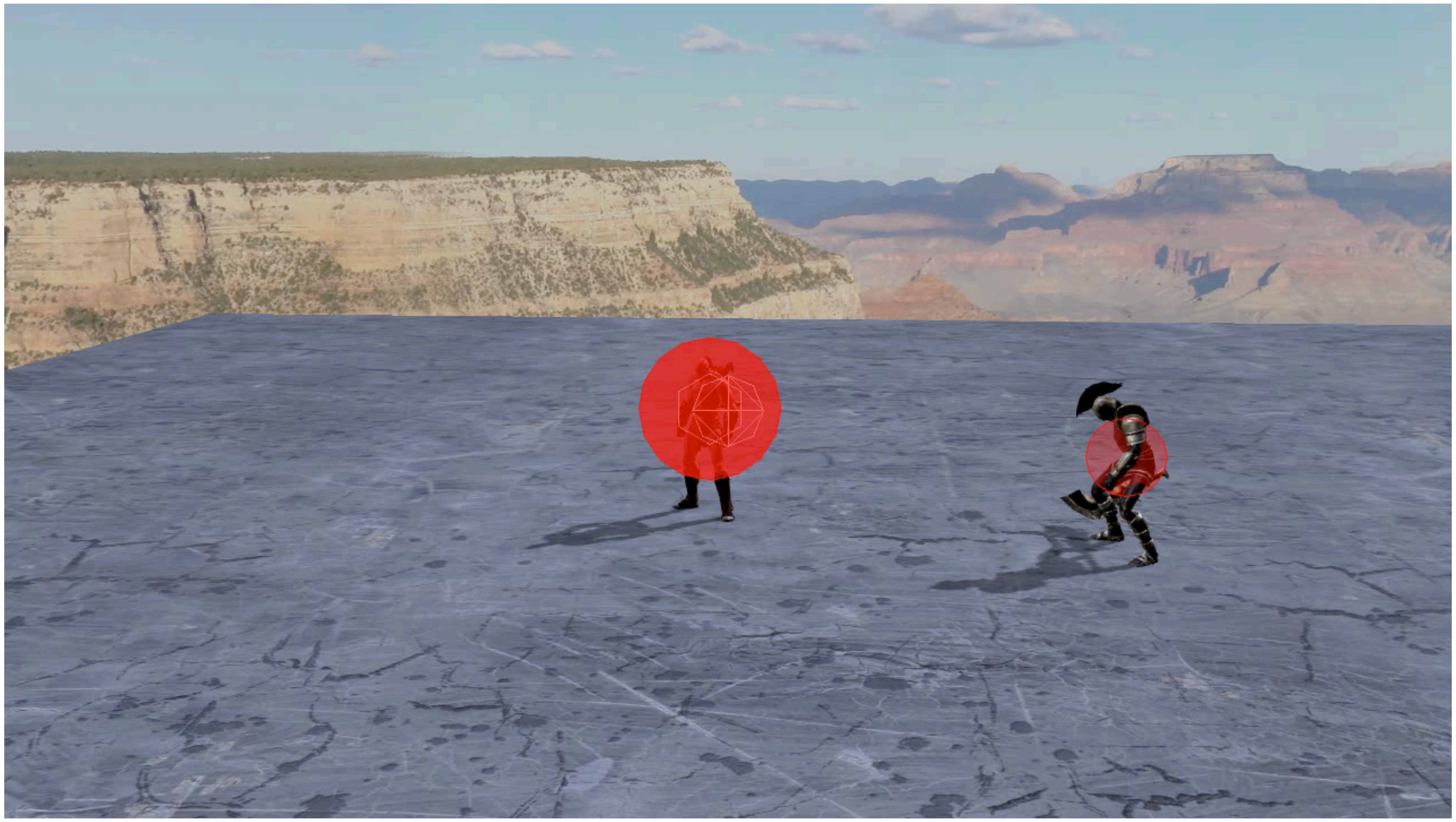
So the boom target is now the sum of all the target positions multiplied by their weights, and divided by the sum of all weights.
Each targets weight is the product of three contributing weights, each from 0..1
The Base weight, which is how important the target is.
The Distance weight, which fades out between a minimum and maximum distance from the Hero.
And the Activation weight, which we use to smoothly add and remove targets. Usually when a creature is spawned or killed.

## Targeting : Weighted Average



So lets see that in action. Now for a fight with lot of creatures, this works pretty well.

## Targeting : Weighted Average



But by the time we’re down to one creature, well, it’s not really getting in as close as it could.

## Targeting : Weighted Average

- Problems:
  - Ignores size of fight
  - Doesn’t get camera in as close as it could go
  - Doesn’t help when threats spread out
- Really want a solution that takes each target into consideration

The problem is that this technique makes no distinction between a fight that’s close in, or has spread out.

This makes it almost impossible for the camera designers to control the framing of the fight.

The flaw is that in the averaging process, it loses all the extra targeting information. We really want a solution that takes each target into account, but keeps the most important target in shot at all times.

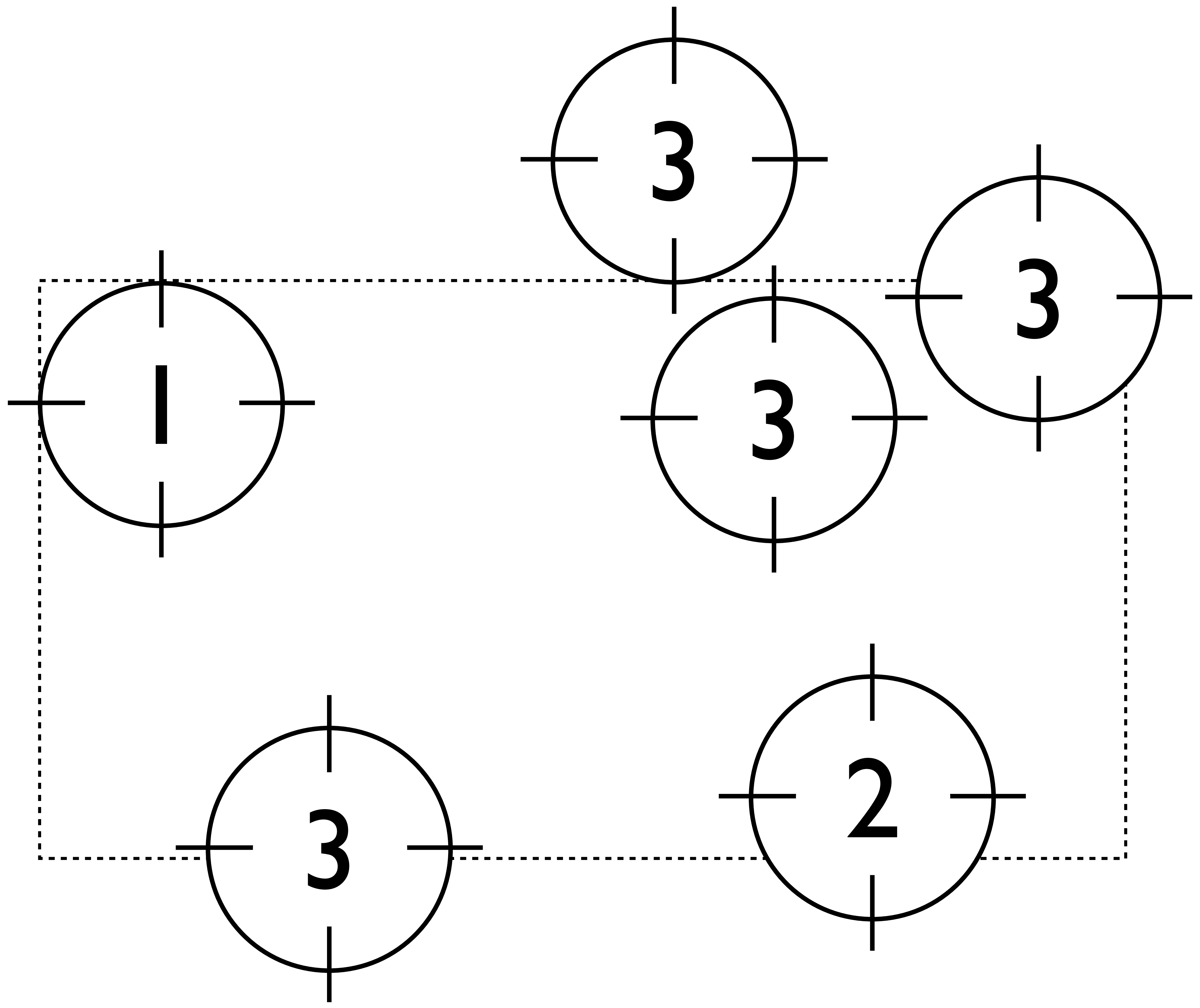
## Targeting : Prioritised Framing

- Promise that highest priority target remains in frame
- Frame lower priority targets as best possible
- New framing and distance constraints to handle multiple targets
- Iterate over priority levels, starting at lowest
- Constraining highest priority last, ensures it is in frame
- Move camera along Boom to best frame targets

So, for the second pass at dealing with multiple targets, the goal was to promise that the highest priority target would remain in frame. And to make the best effort to frame the lower priority targets.
The new algorithm expands upon the framing and distance constraints. Instead of clamping a single point, it tracks the camera to best frame the extents of multiple target spheres.
It does this for each active priority level, starting from the lowest and ending at the highest. That way we guarantee that the highest priority target is in frame.
It then pulls back or pushes forward to best frame all targets.

## Targeting : Prioritised Framing

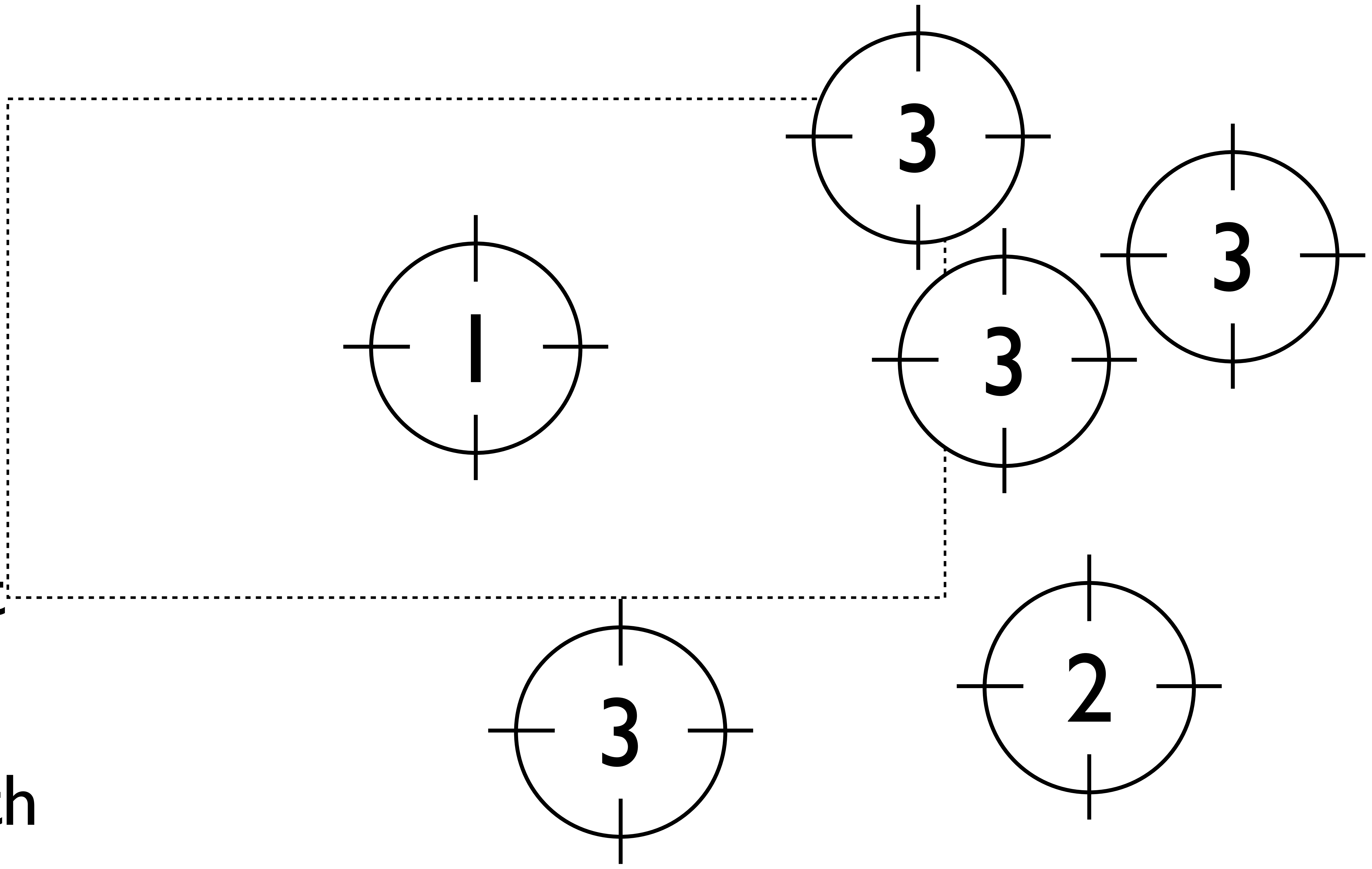
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here's the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he's on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we're done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

## Targeting : Prioritised Framing

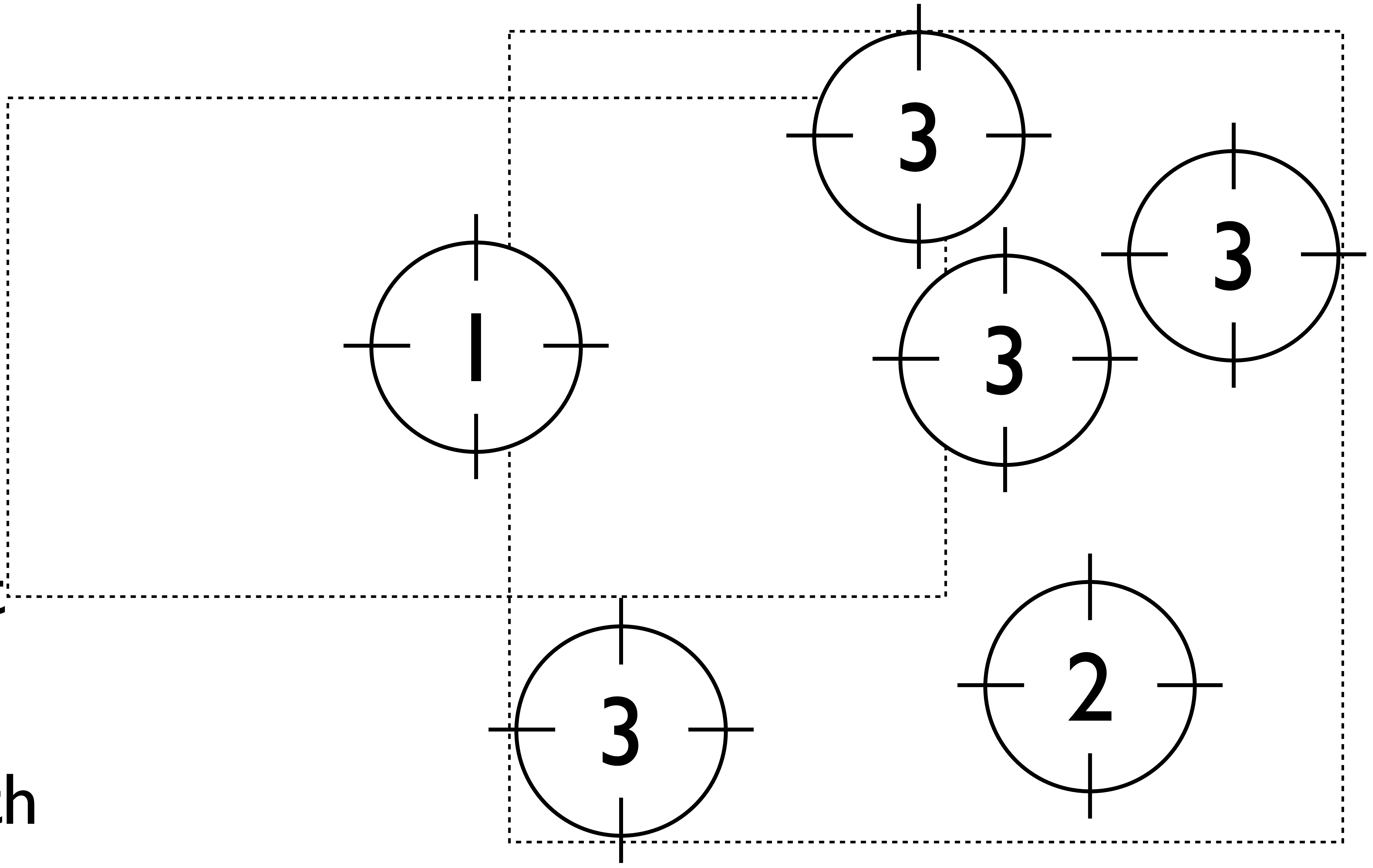
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here’s the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he’s on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we’re done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

## Targeting : Prioritised Framing

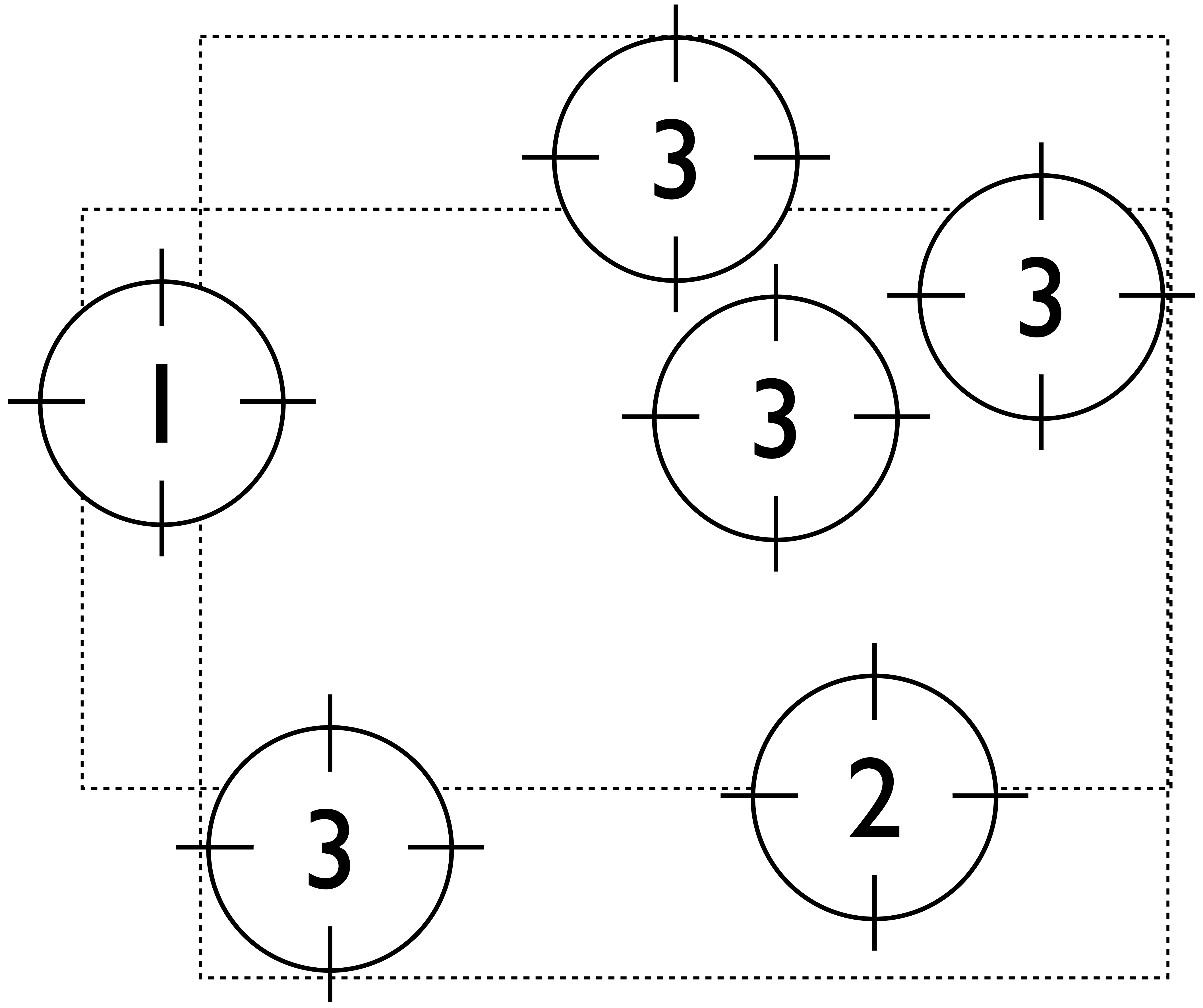
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here’s the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he’s on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we’re done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

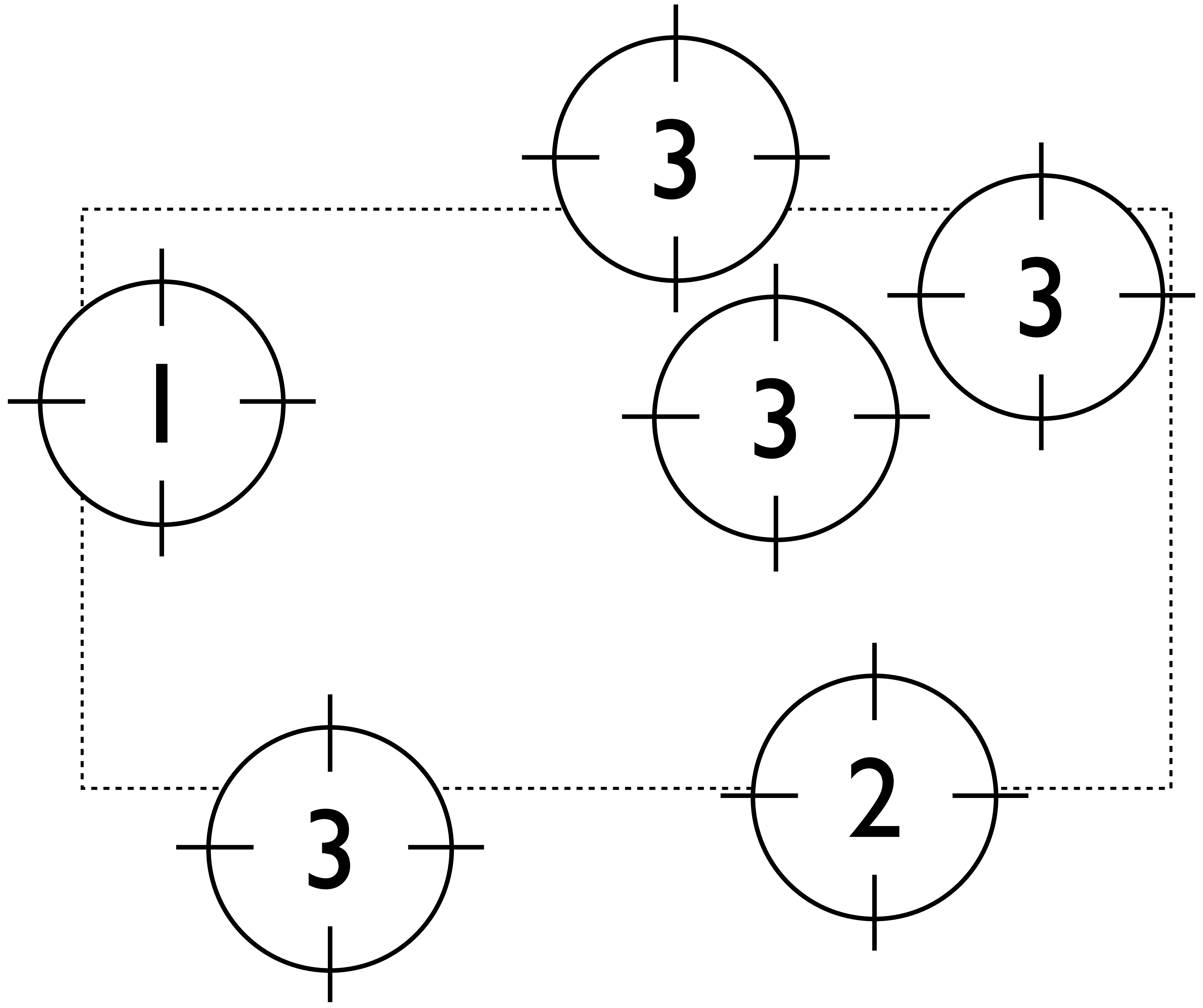
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here's the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he's on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we're done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

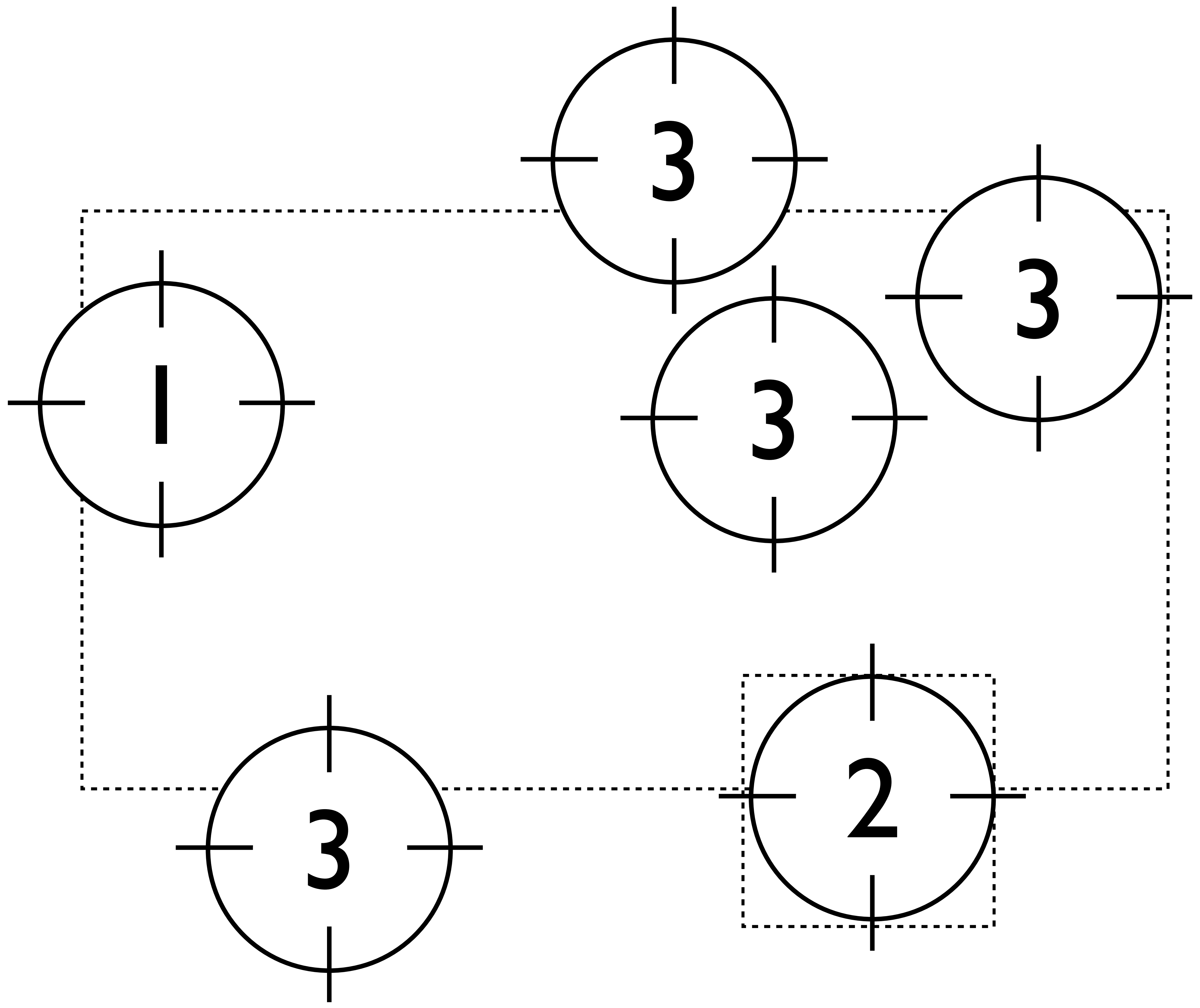
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here's the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he's on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we're done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

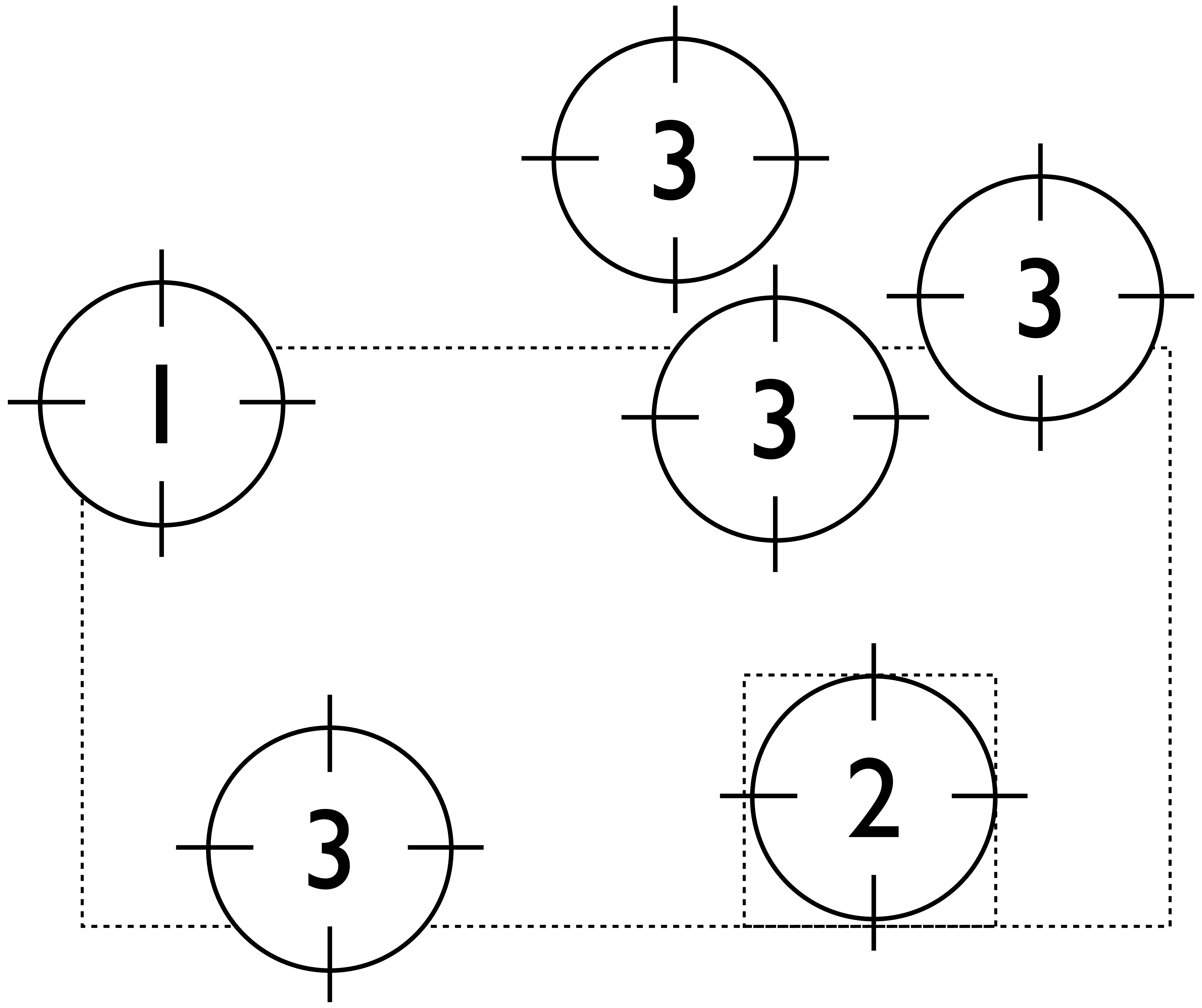
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here's the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he's on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we're done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

## Targeting : Prioritised Framing

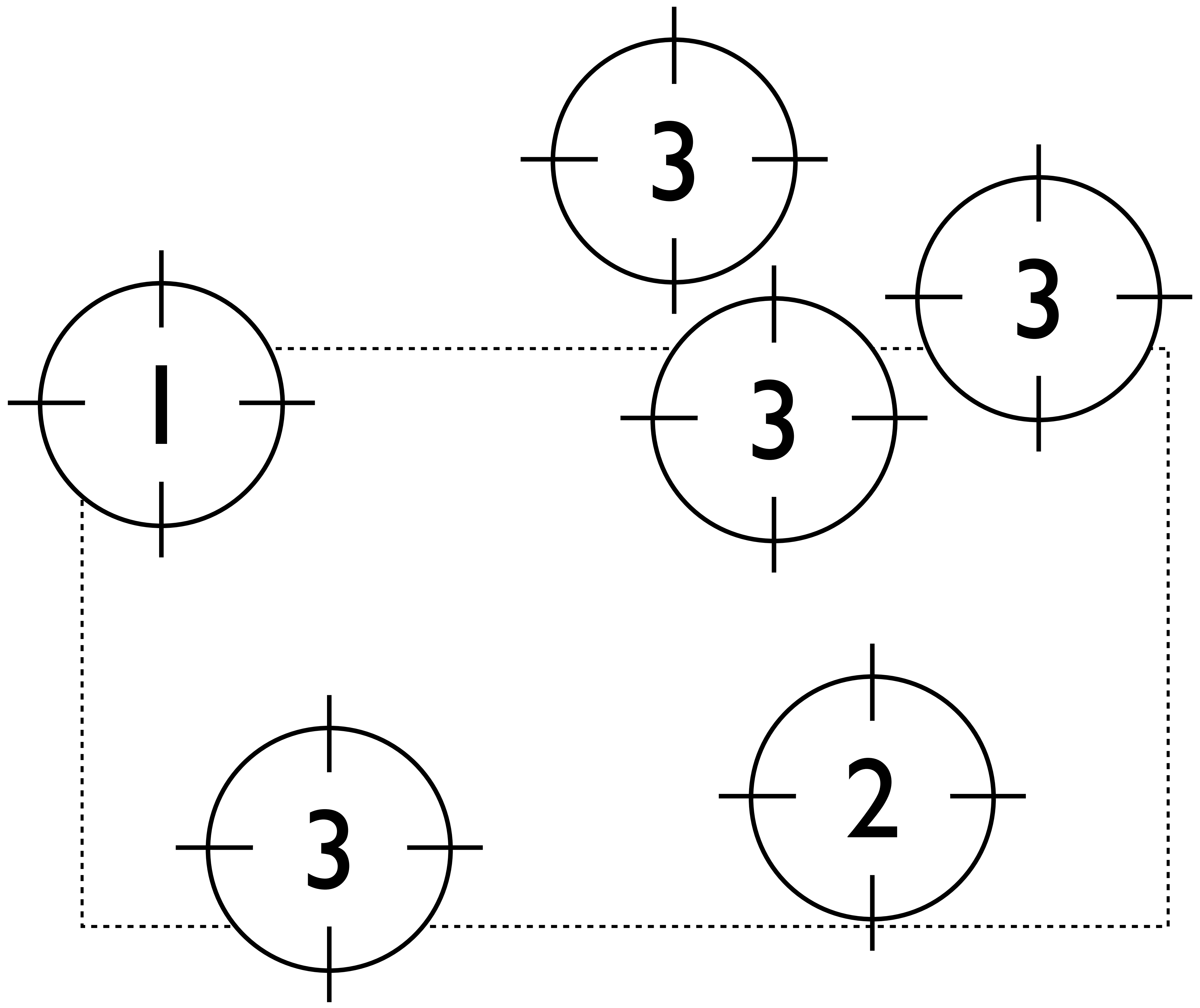
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here’s the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he’s on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we’re done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

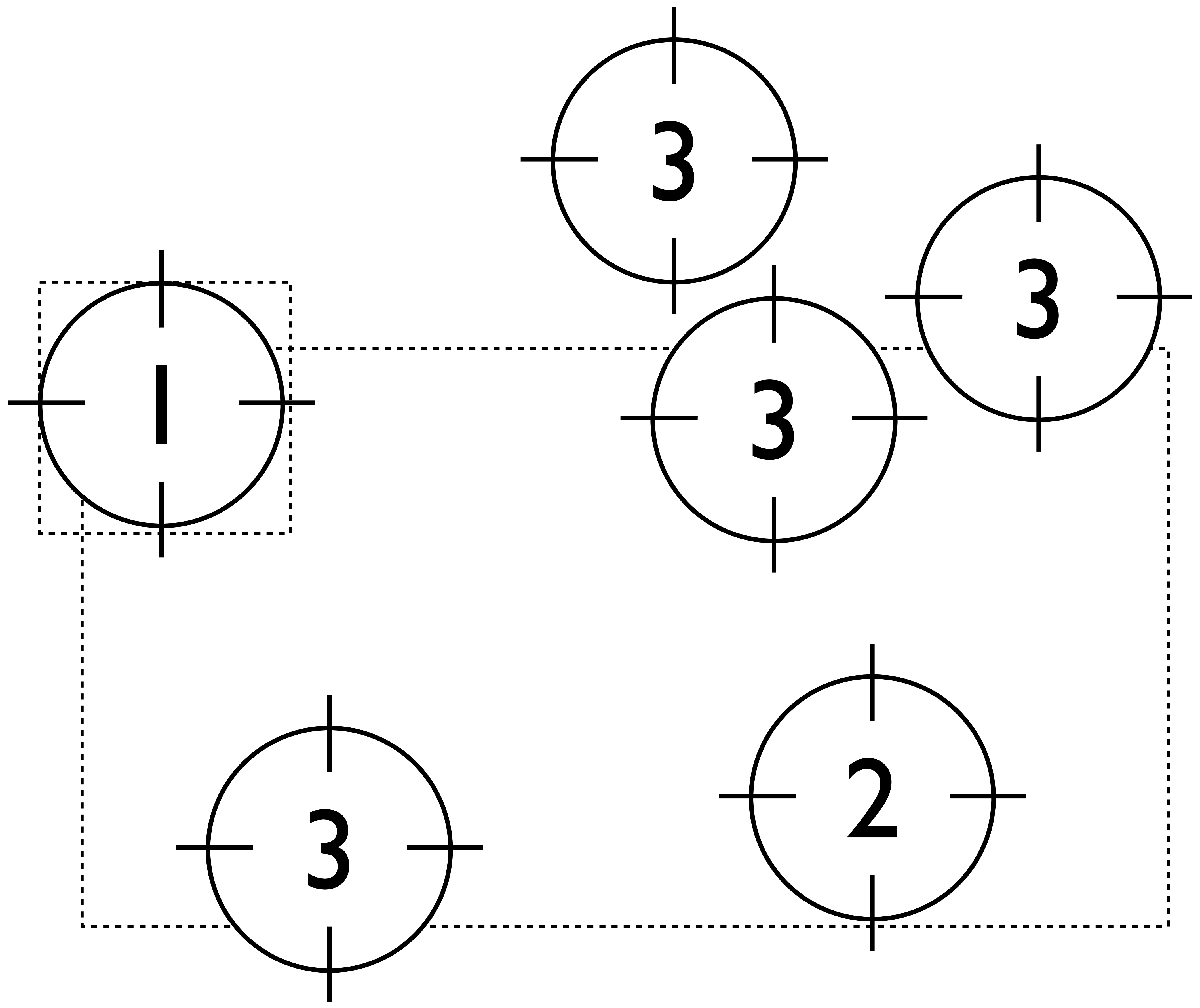
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here’s the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he’s on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we’re done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

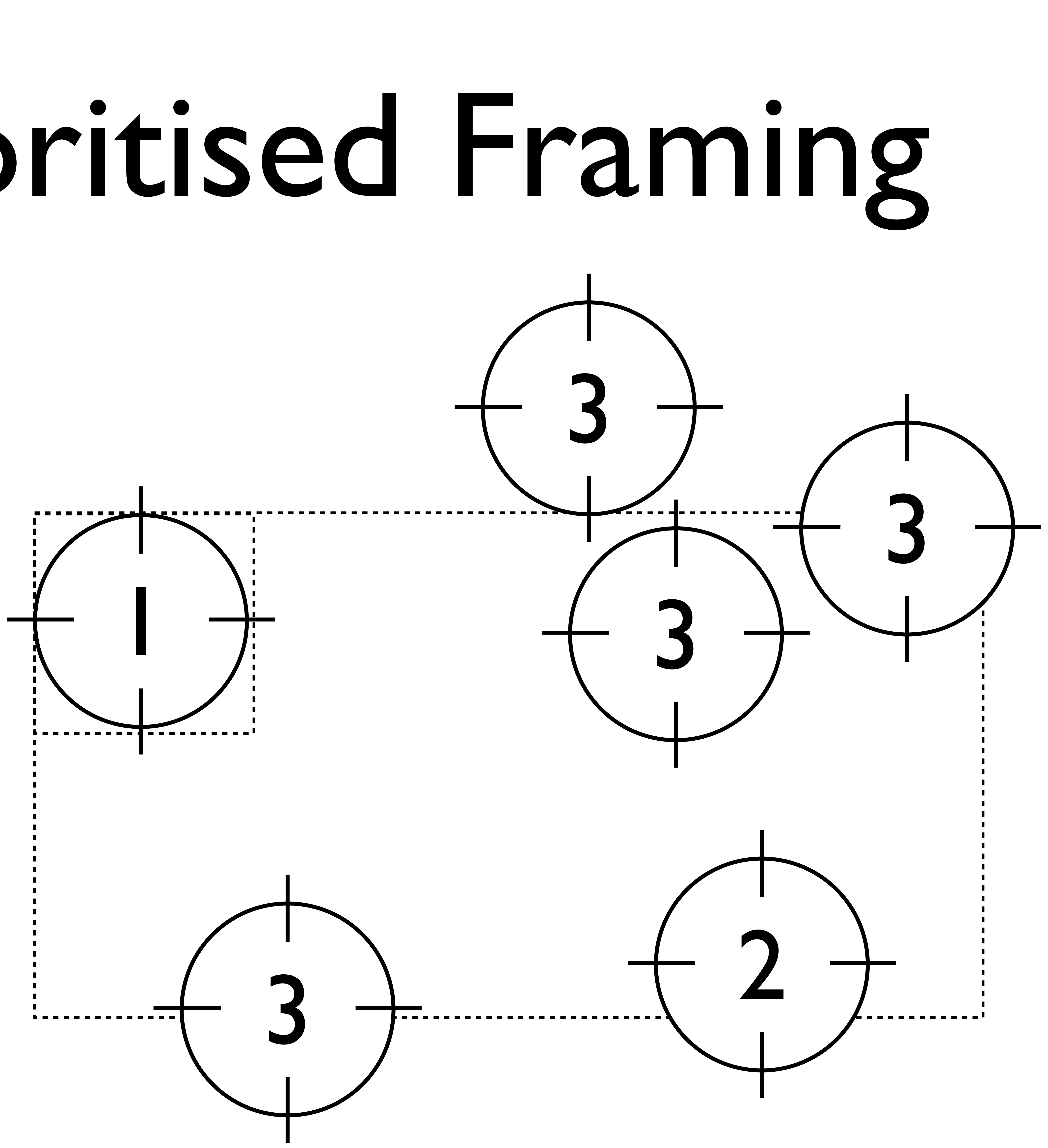
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here's the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he's on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we're done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

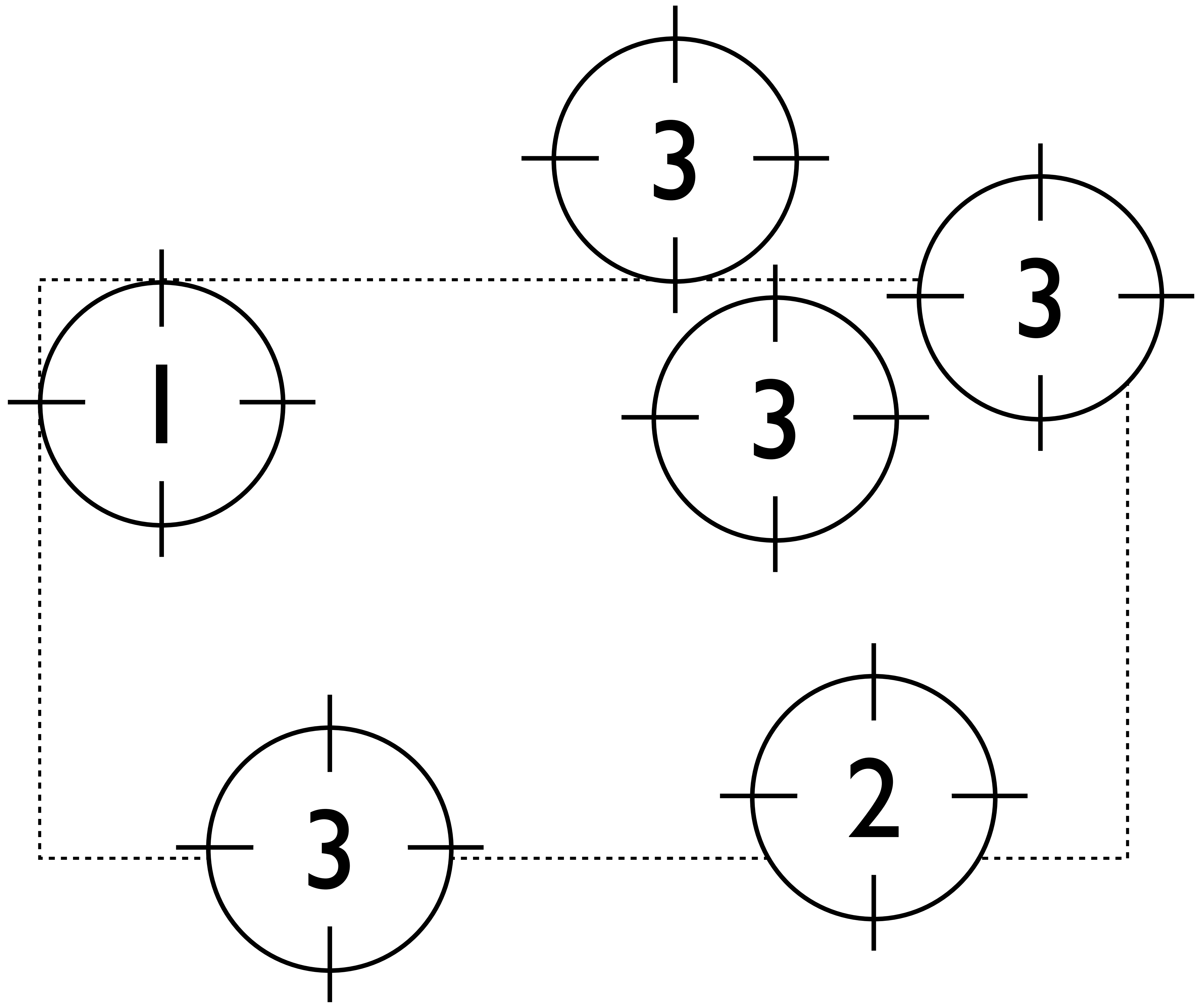
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here's the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he's on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we're done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

# Targeting : Prioritised Framing

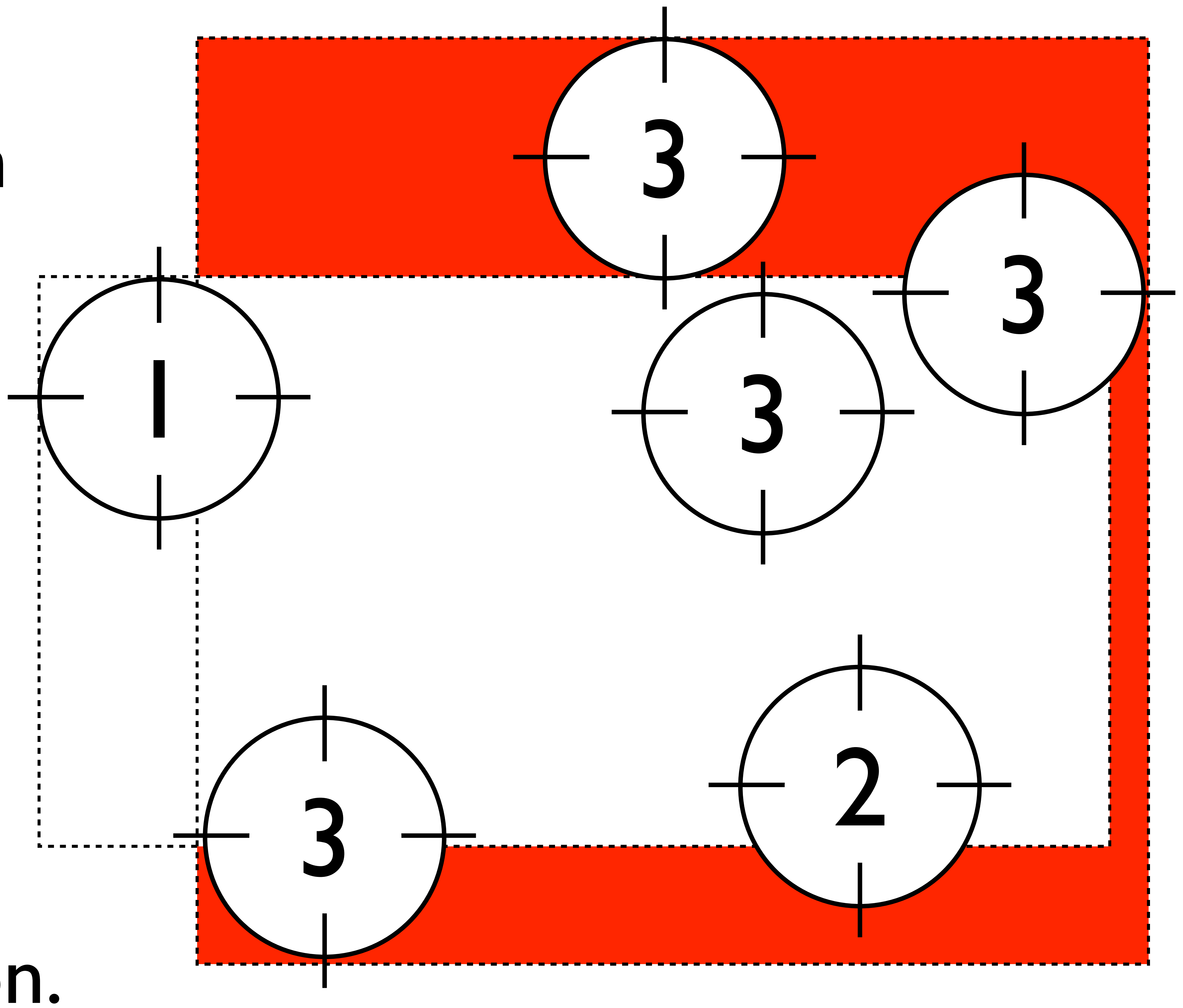
1. Centre on Hero
2. Calculate extents of lowest priority level
3. Minimally track to frame extents
4. Repeat 2 & 3 for next highest priority level
5. Apply delta to Azimuth and Elevation



So here’s the algorithm. This is all done in the spherical space we used for the original safe zone framing constraint.
*First we centre the frame on the hero, this is optional, but it lets you have the large safe zone without Kratos rattling around in it when he’s on his own.
*Then we calculate the extents of the lowest priority level, *and move the frame the smallest amount required to encompass as much of it as we can.
*Repeat at the next highest priority level until done.
*******
When we’re done, we have the new frame. Well, what we actually have is the amount the frame moved, and we apply that delta to the the azumith and elevation parameters that determine the position of the boom target on screen.

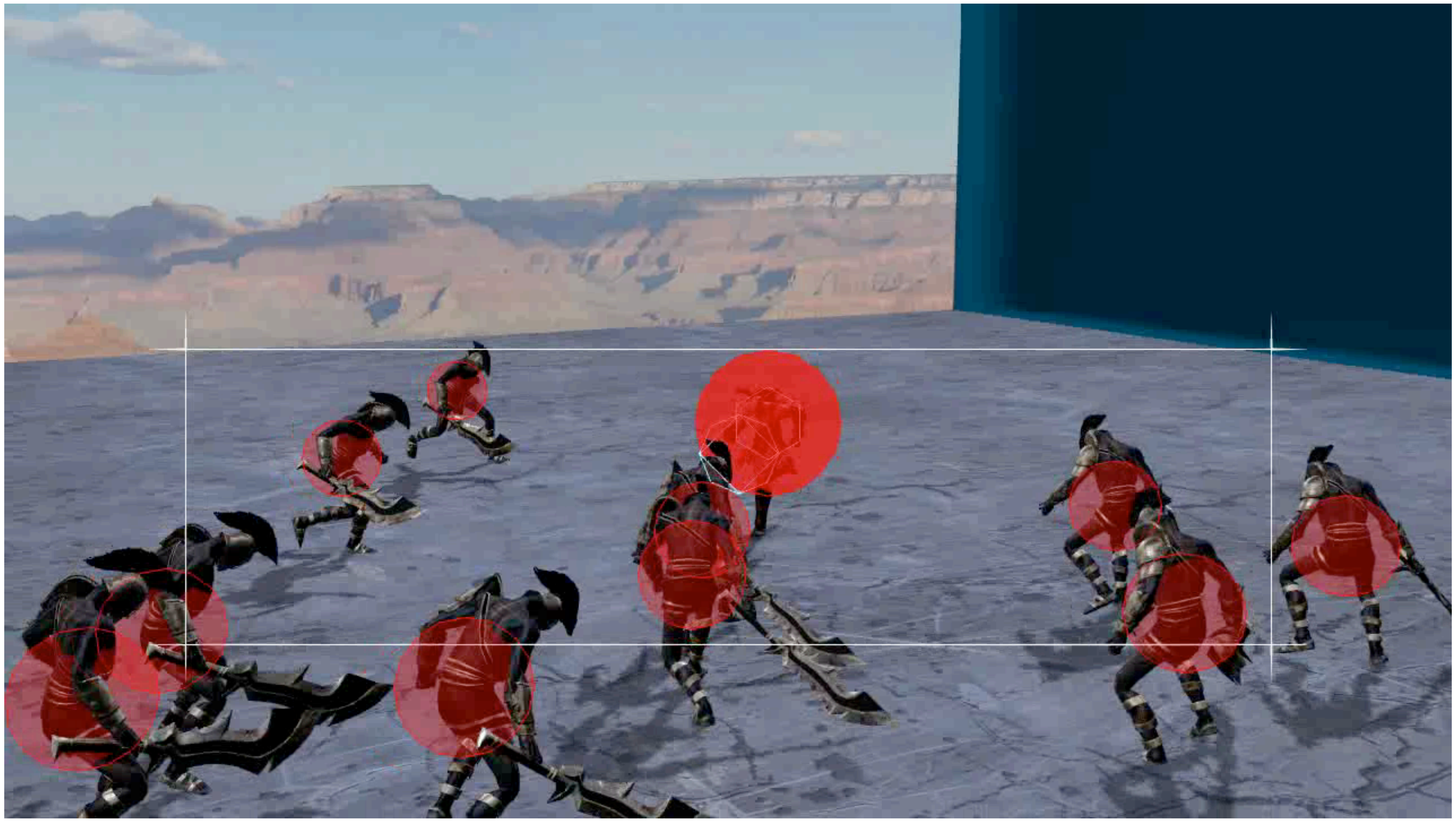
## Targeting : Prioritised Framing

- Use weight minimisation function on distance to minimise extents outside Safe Zone
- Framing weight pushes back to maintain framing
- Distance weight pulls camera towards characters
- No damping! Danger of oscillation.



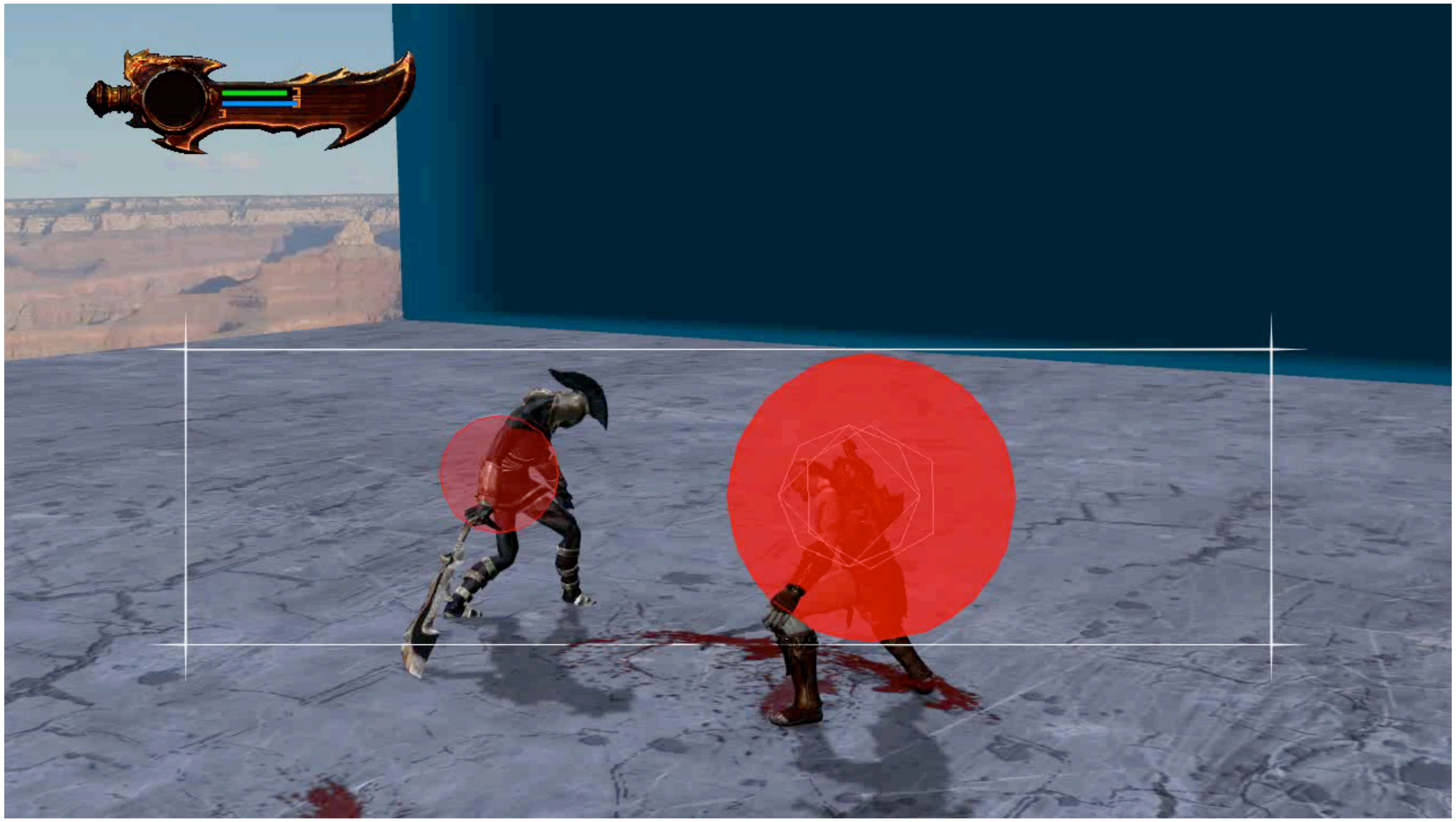
We also replace the boom ratio element of the distance constraint.
Instead we calculate the initial position of the camera on the boom, by using the same weight minimisation function we used to determine the position of the dolly on the rail.
This time we use the area of the target extents that remain outside the safe zone as a weight pulling the camera back towards the dolly. While we use distance from the boom target, as a weight pushing the camera forwards towards the action.
It's worth noting that we try to avoid damping this distance constraint. As there's the distinct danger of it interfering with the damping on the dolly. Which in turn can lead to unwelcome circular oscillations.

# Targeting : Prioritised Framing



So lets have a look at that in action. The large fight still looks good.

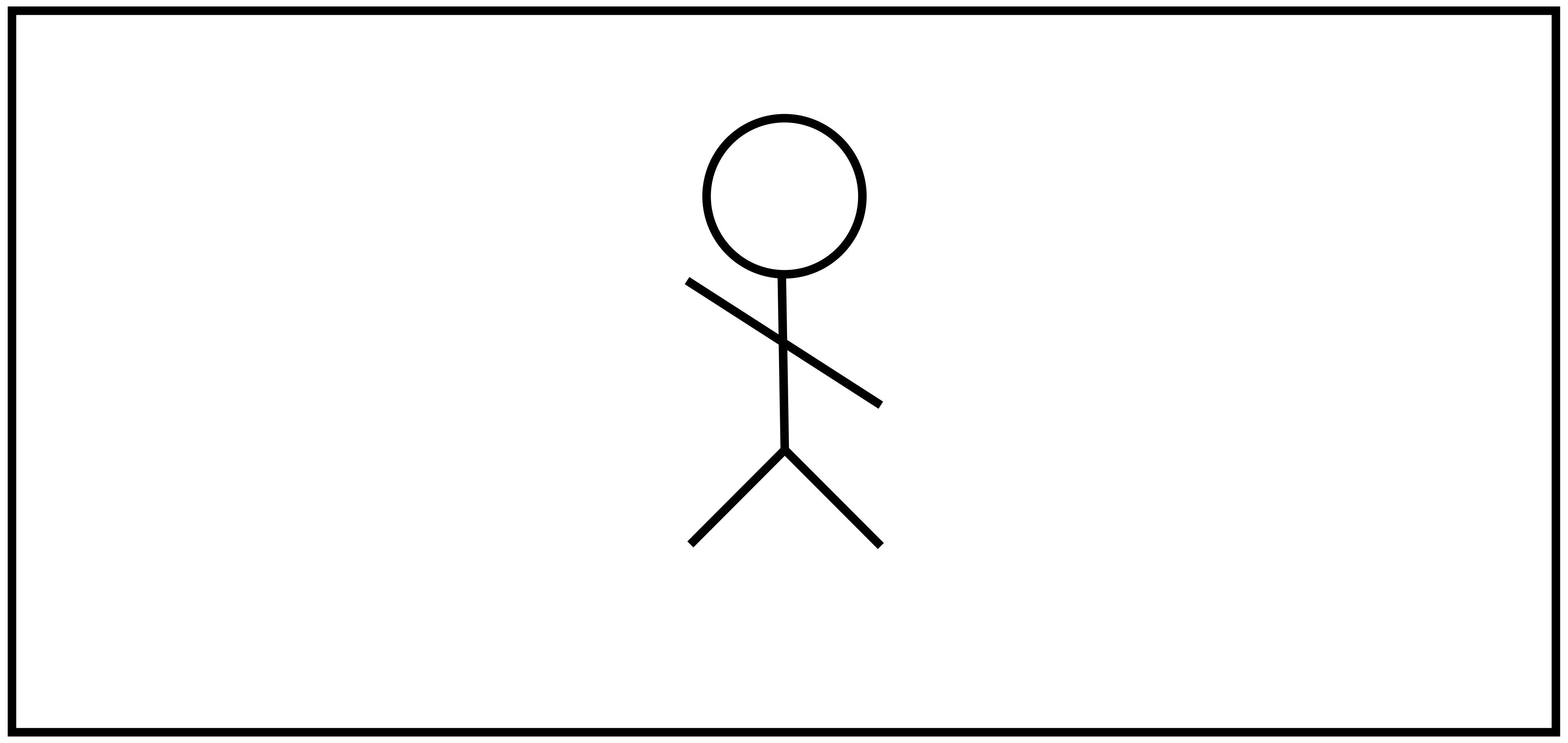
# Targeting : Prioritised Framing



But the single enemy fight is much improved, and the camera can get much closer.

## Targeting : Prioritised Framing

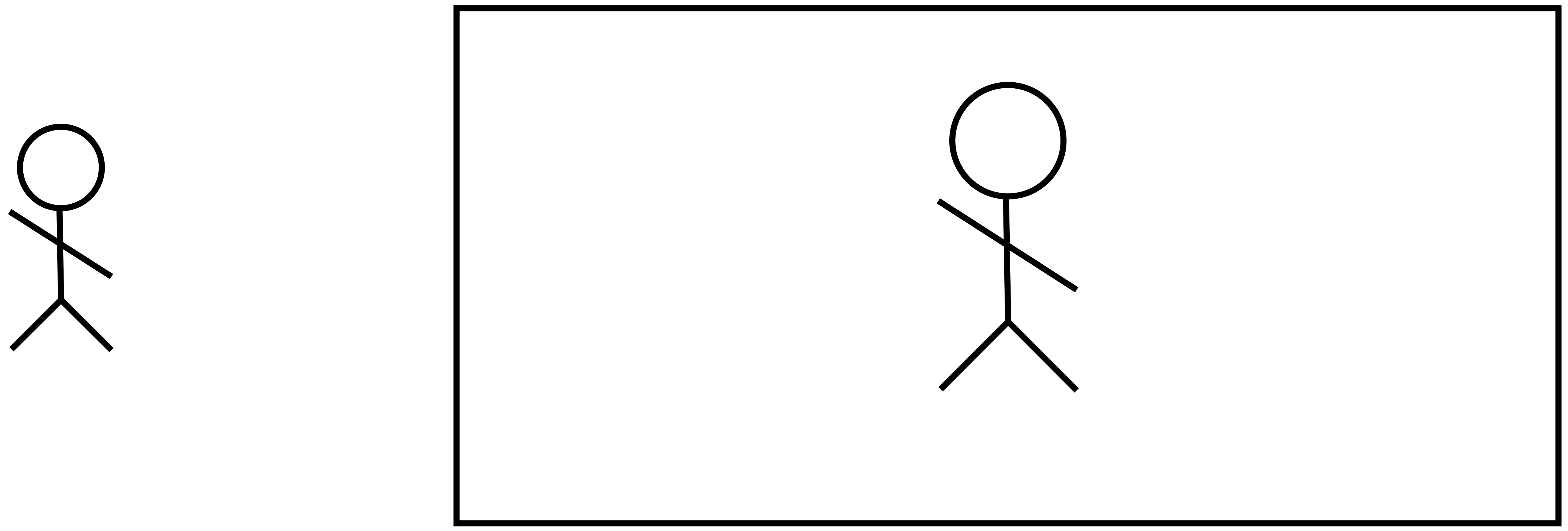
Problem: Birth and death of targets at bounds causes pops



Now that's pretty cool, but we're not done yet. There's a couple of problems. The first of which is, how do we deal with birth and death? Unlike the weighted average algorithm, this algorithm has no concept of weight, so when a character is spawned outside of the frame...
*
We get an unsightly pop.

## Targeting : Prioritised Framing

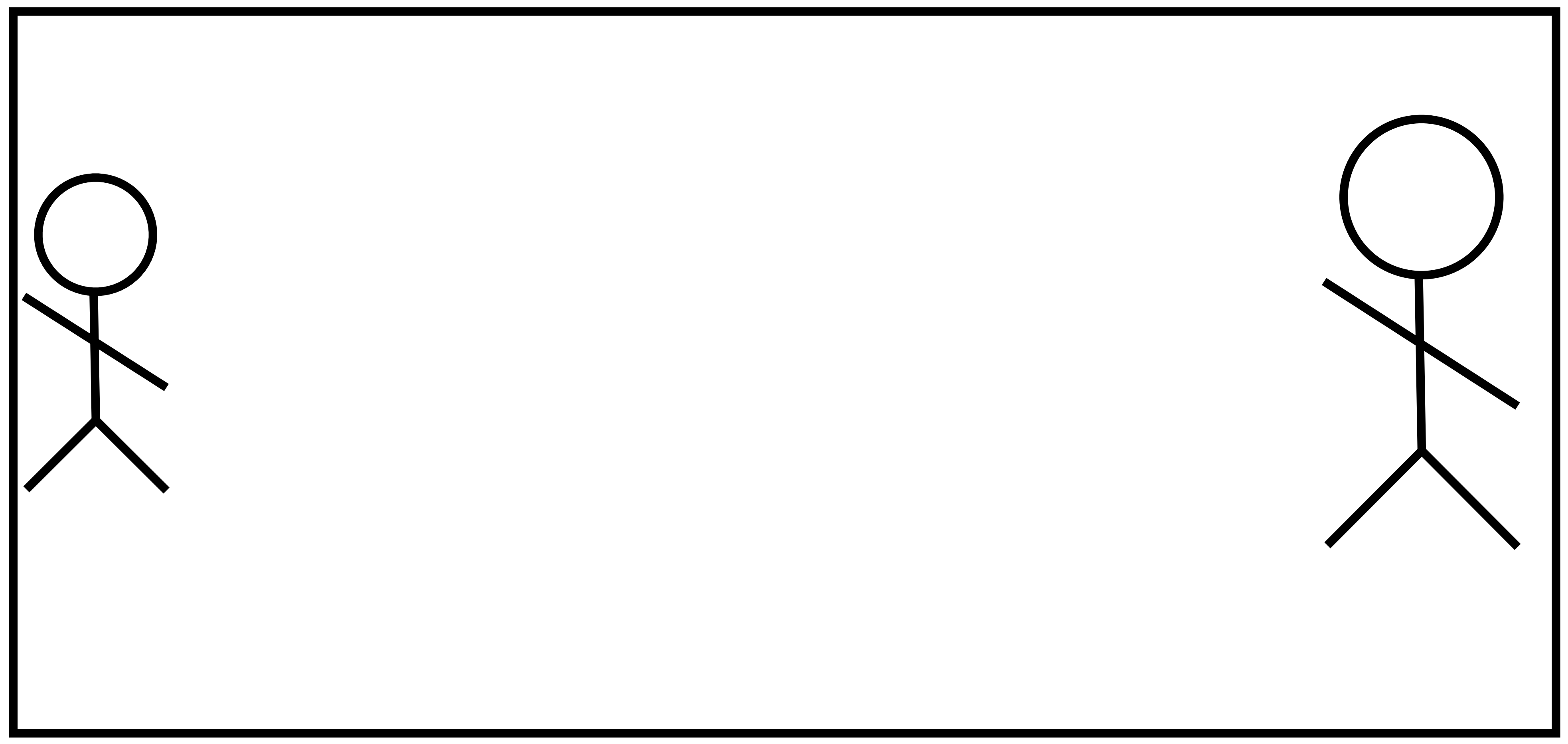
Problem: Birth and death of targets at bounds causes pops



Now that's pretty cool, but we're not done yet. There's a couple of problems. The first of which is, how do we deal with birth and death? Unlike the weighted average algorithm, this algorithm has no concept of weight, so when a character is spawned outside of the frame...
*
We get an unsightly pop.

## Targeting : Prioritised Framing

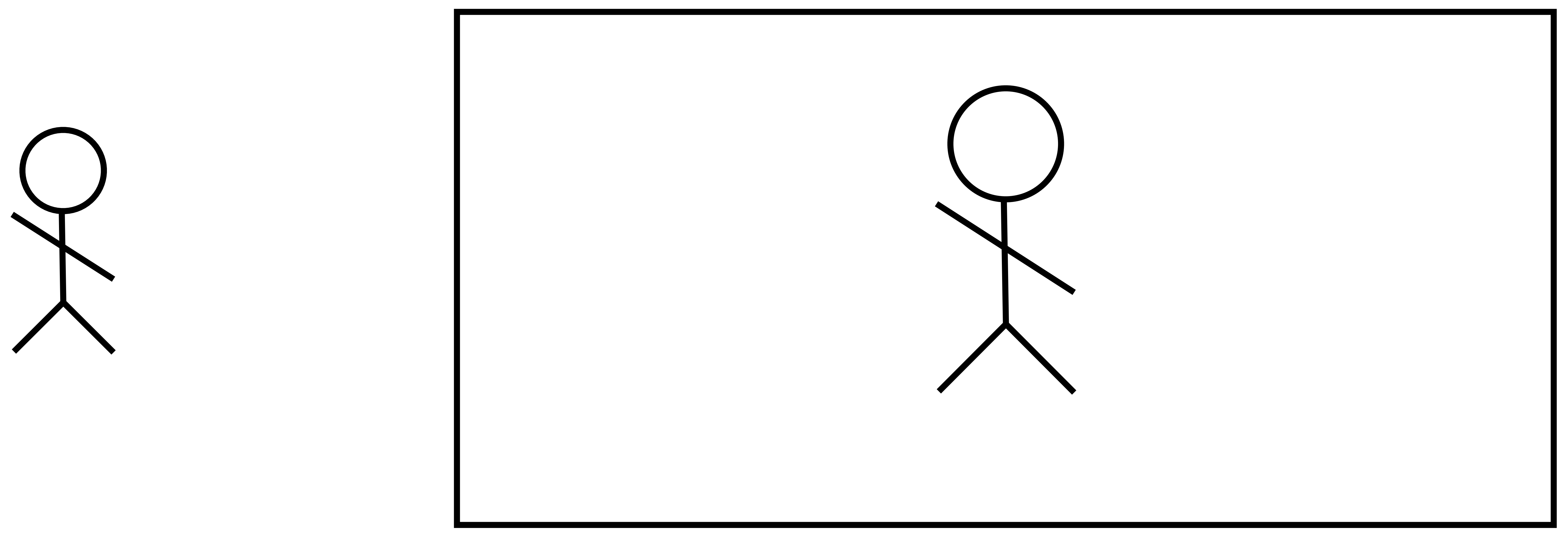
Problem: Birth and death of targets at bounds causes pops



Now that's pretty cool, but we're not done yet. There's a couple of problems. The first of which is, how do we deal with birth and death? Unlike the weighted average algorithm, this algorithm has no concept of weight, so when a character is spawned outside of the frame...
*
We get an unsightly pop.

# Targeting : Prioritised Framing

Solution : Tween targets in from highest priority target



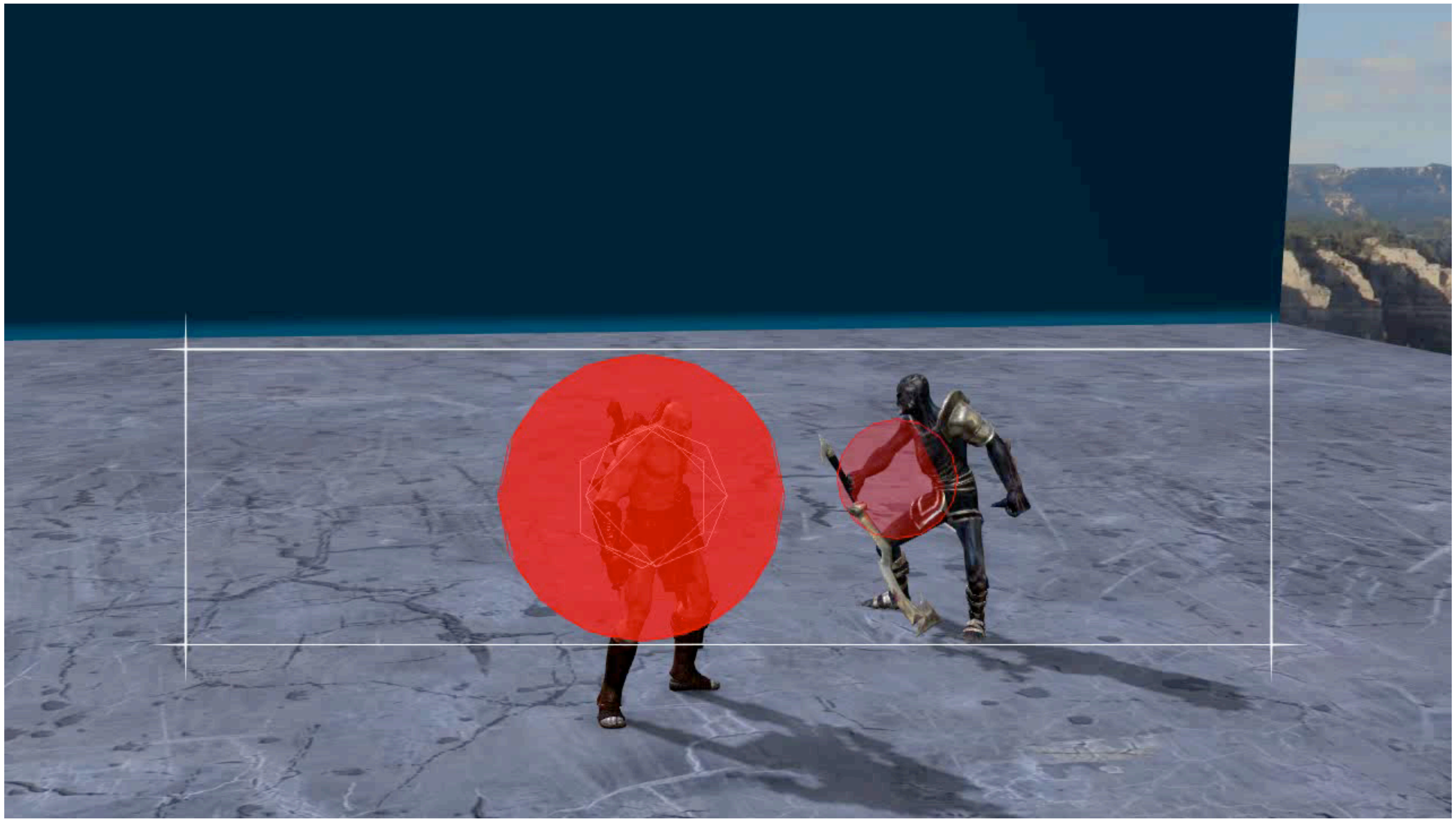
So the solution is to add weights to the system, and use them to tween the targets in and out.
*We tween a target’s logical position between it’s physical position and the position of the Hero. We never remove the hero target, and we’ve explicitly guaranteed that it’s going to be in frame, so we can hide the birth and death of the other target behind it without causing a pop.

# Targeting : Prioritised Framing

Solution : Tween targets in from highest priority target

So the solution is to add weights to the system, and use them to tween the targets in and out.
*We tween a target's logical position between it's physical position and the position of the Hero. We never remove the hero target, and we've explicitly guaranteed that it's going to be in frame, so we can hide the birth and death of the other target behind it without causing a pop.

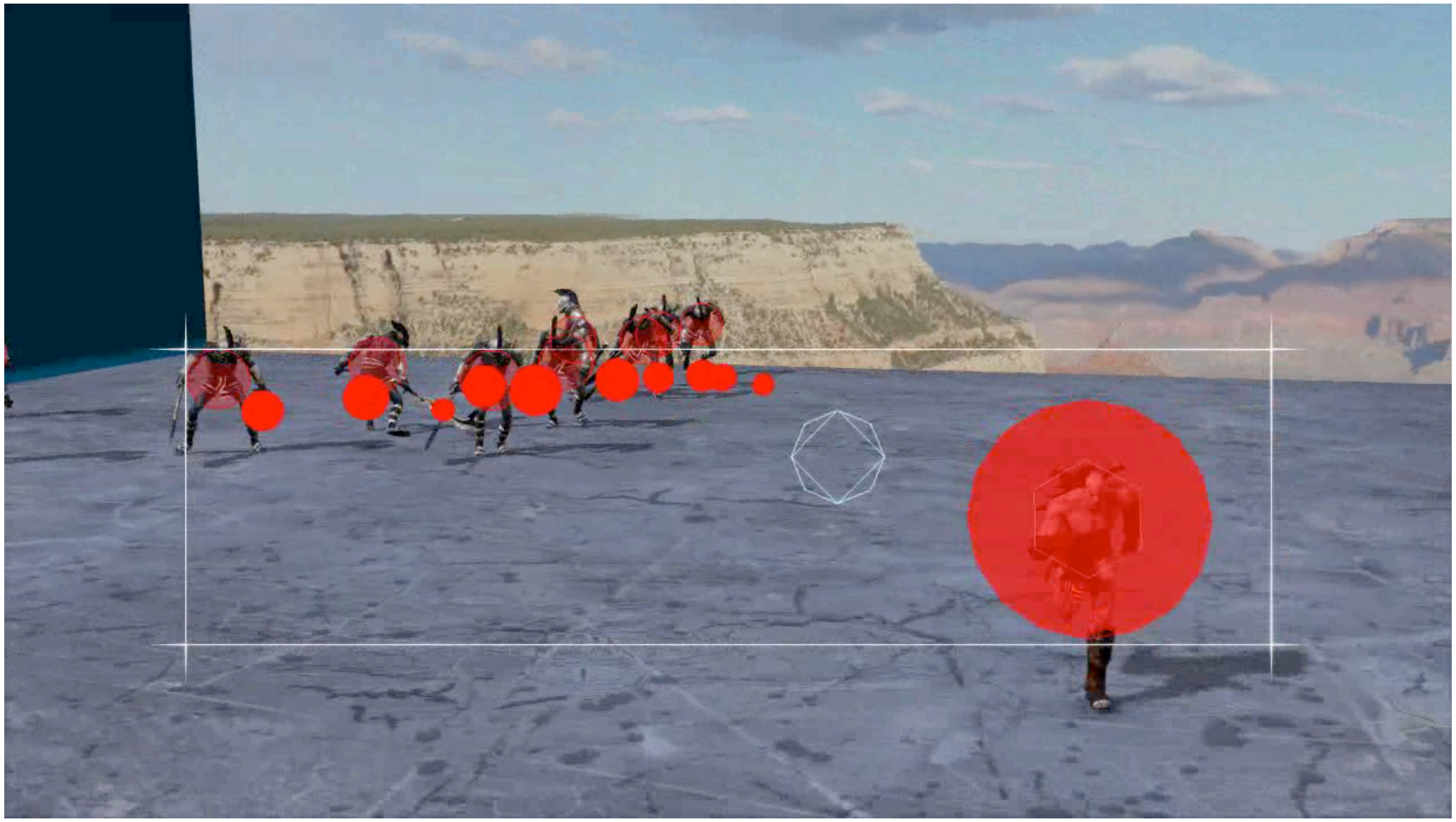
## Targeting : Prioritised Framing



So you see that solid red sphere moving from Kratos to the Grunt after it spawns? That’s the logical target tweening into it’s final position. You can see how it tracks the camera smoothly to frame the new creature.

Note that we tween the size as well as the position. Just in case the target is larger than Kratos.

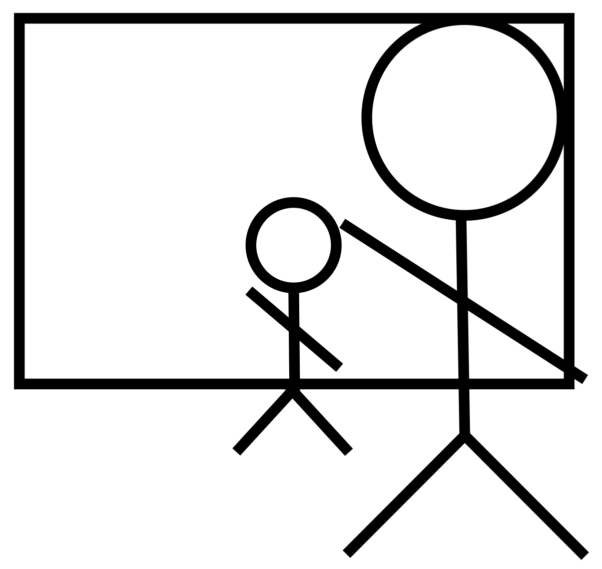
# Targeting : Prioritised Framing

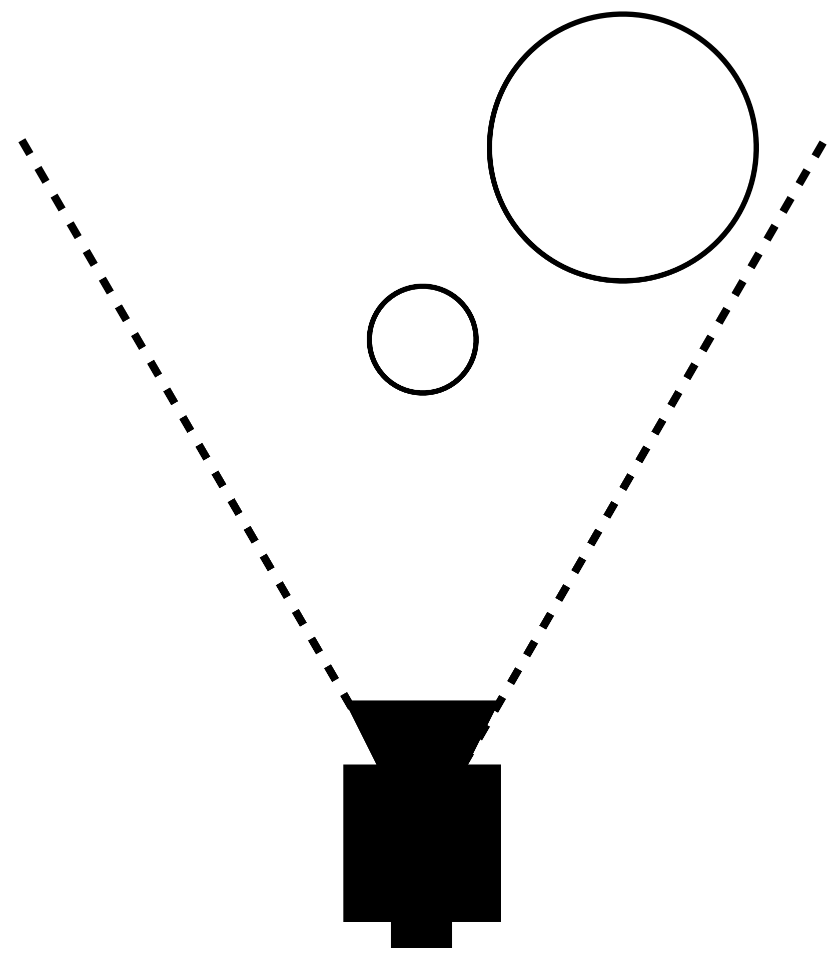


So because we can now weight a target, we can also use the distance weights. Again, the solid red blobs are the weighted targets.

## Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



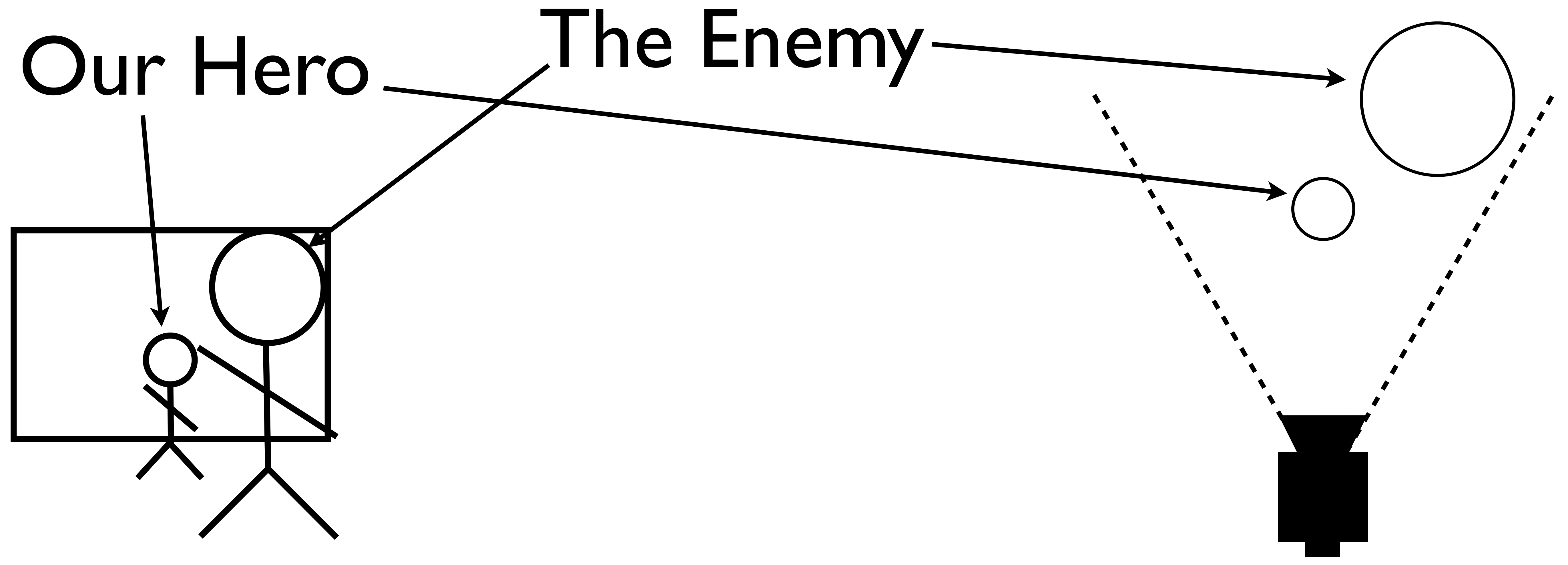


The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



Spherical View

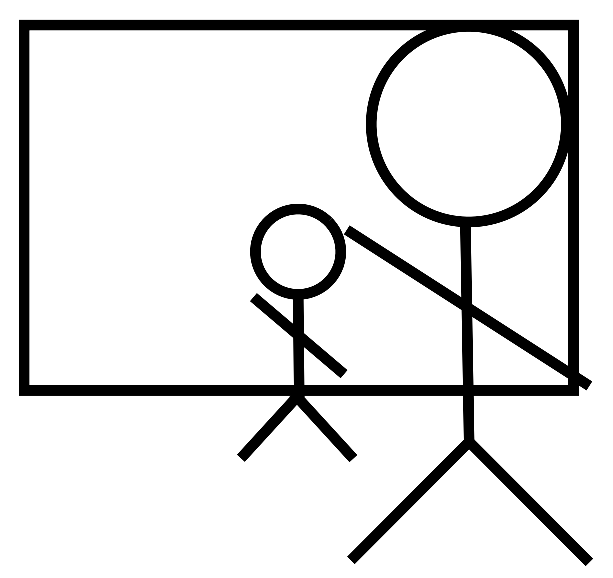
Top Down View

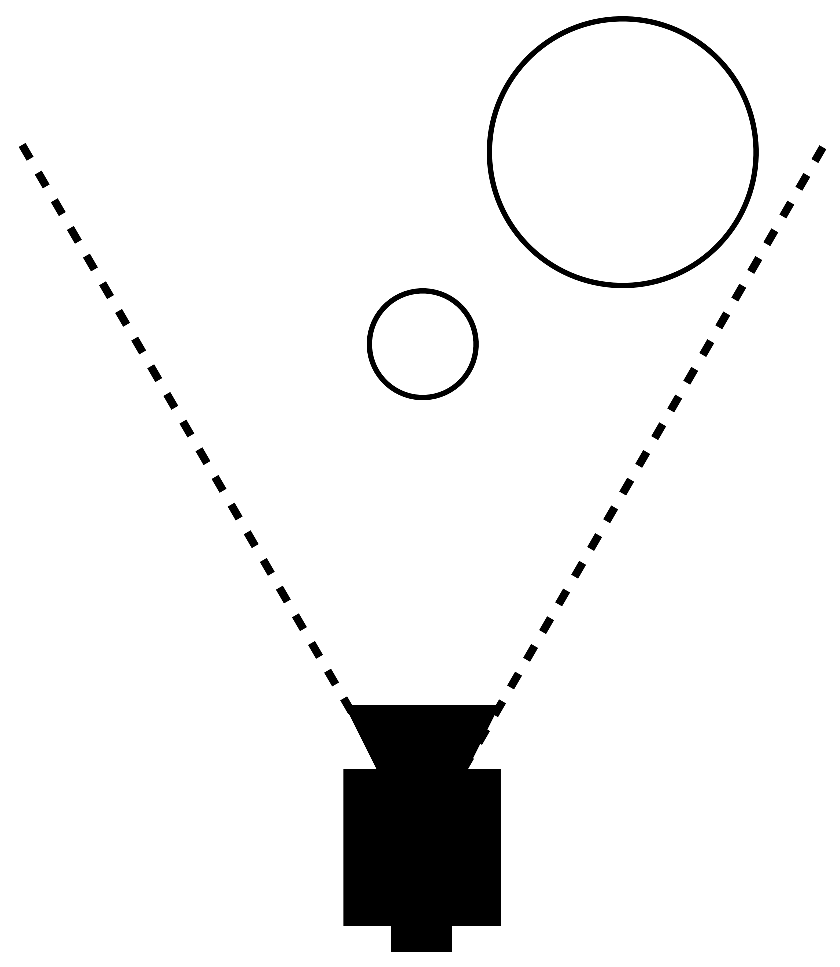
The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera


Spherical View

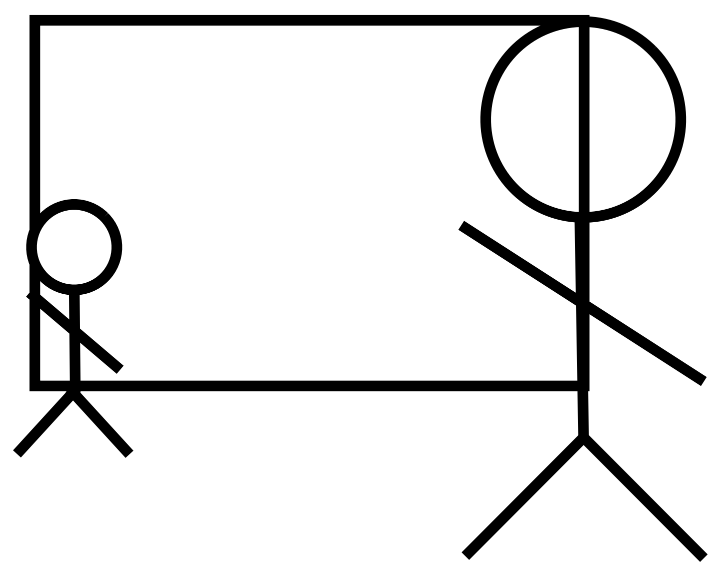

Top Down View

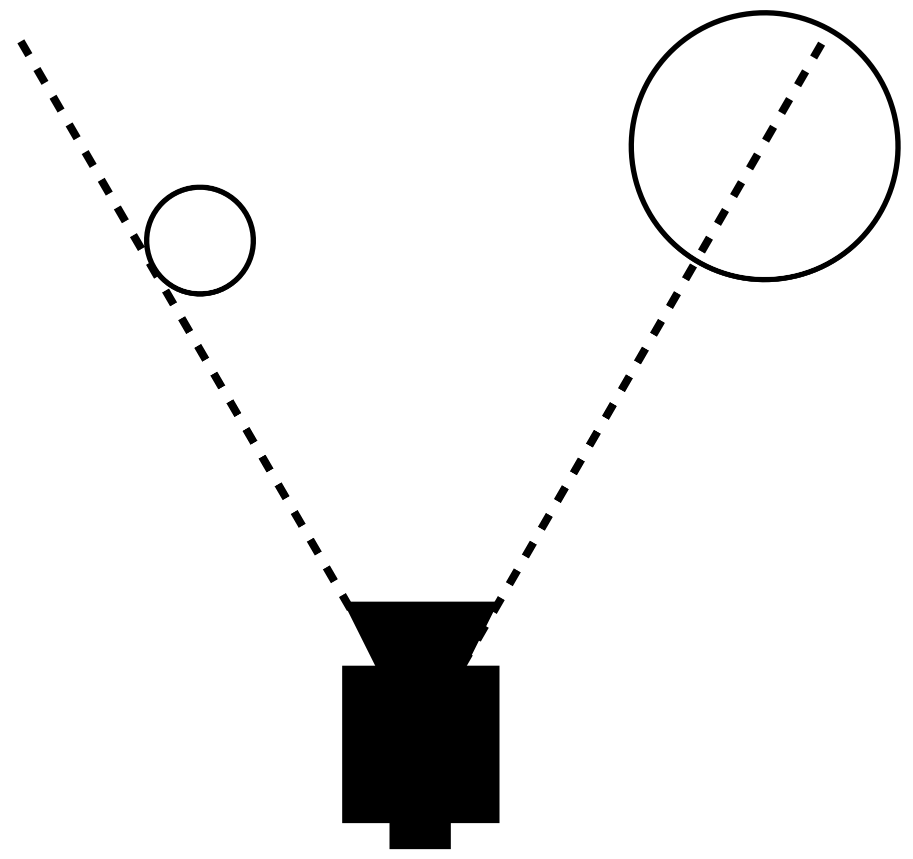
The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera





Spherical View

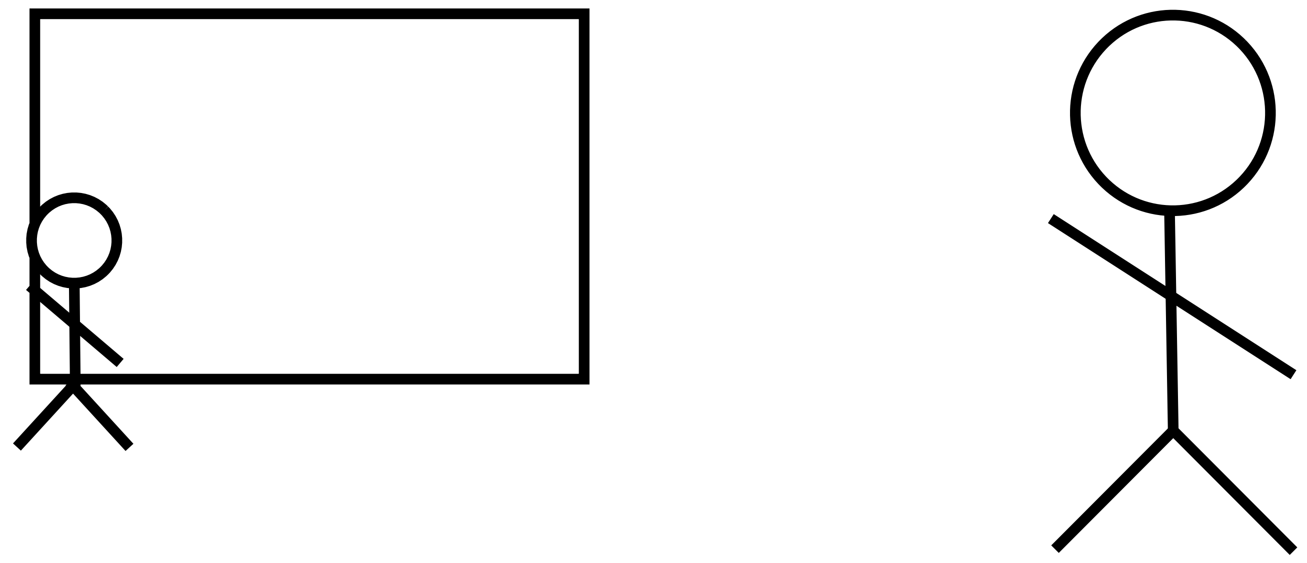
Top Down View

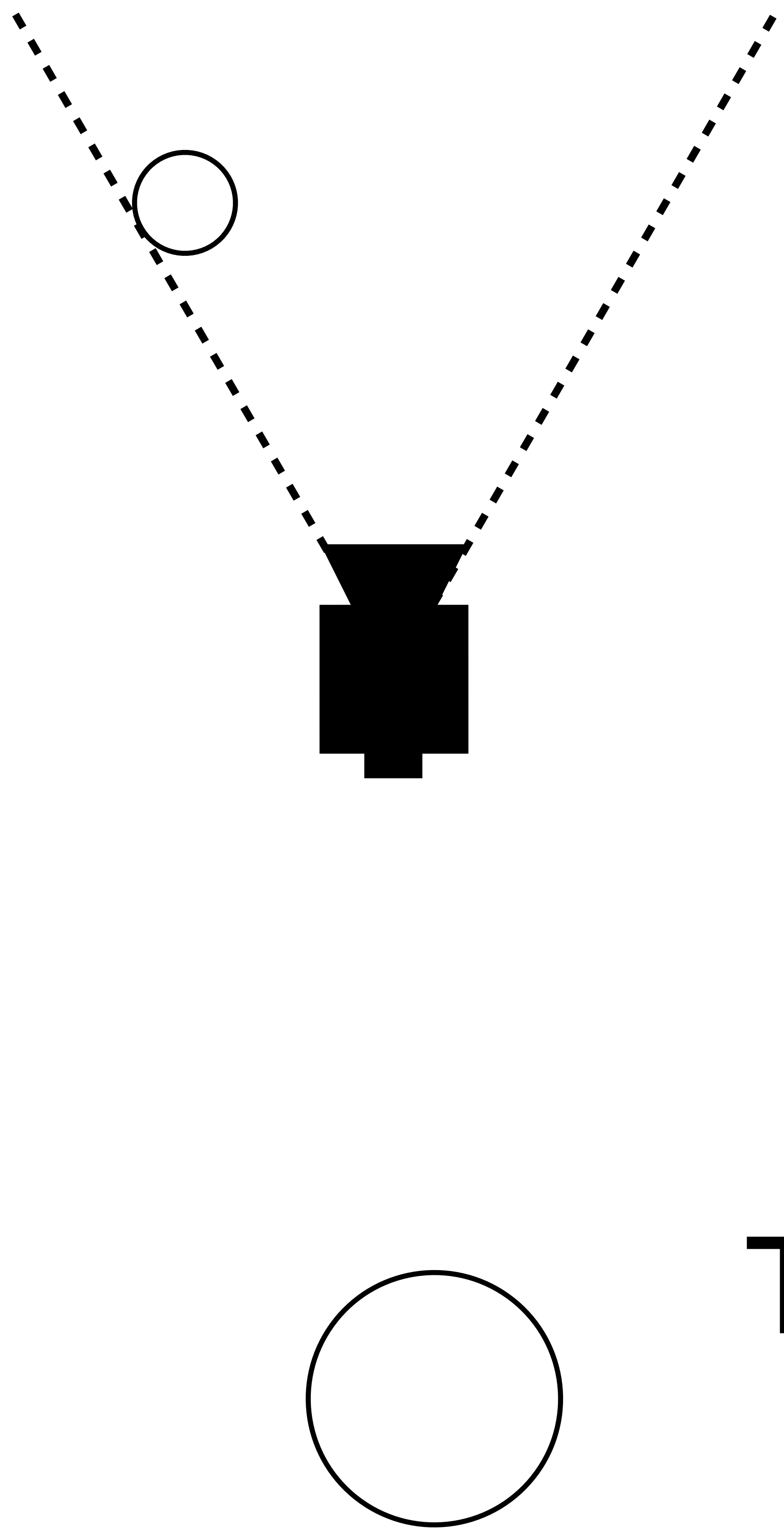
The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

## Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



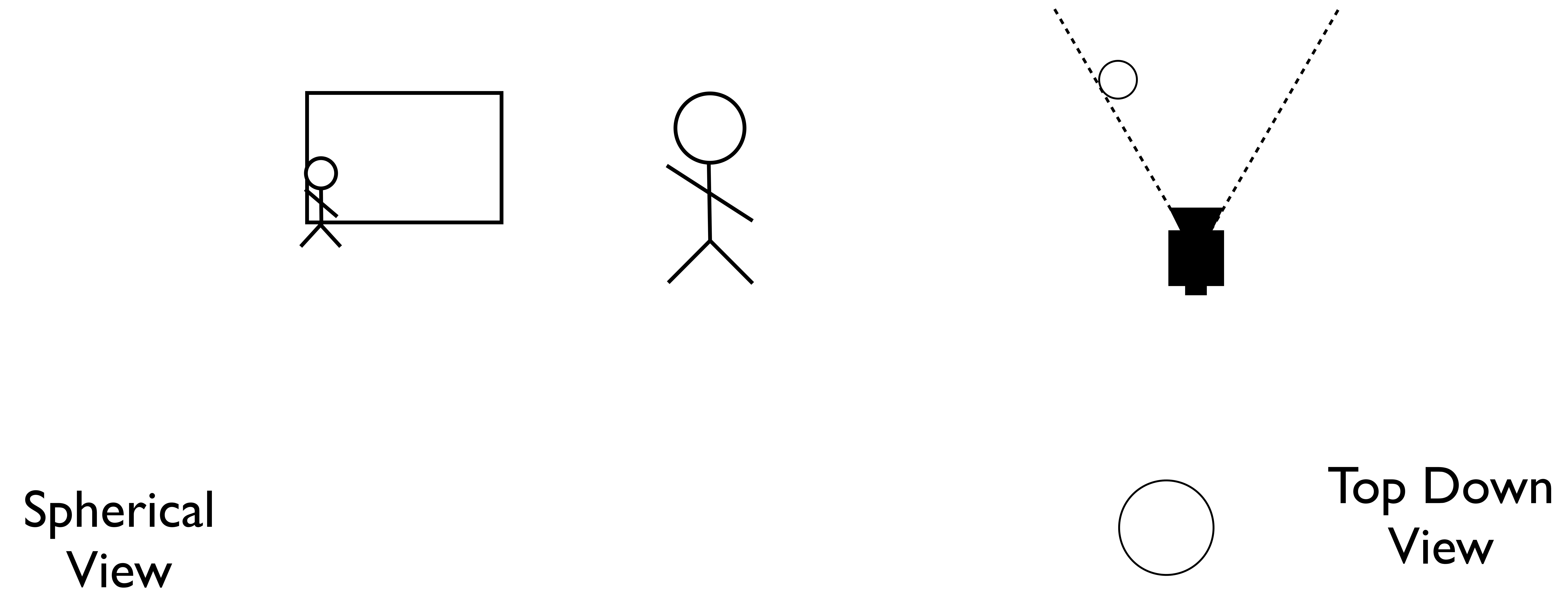


The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

## Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera

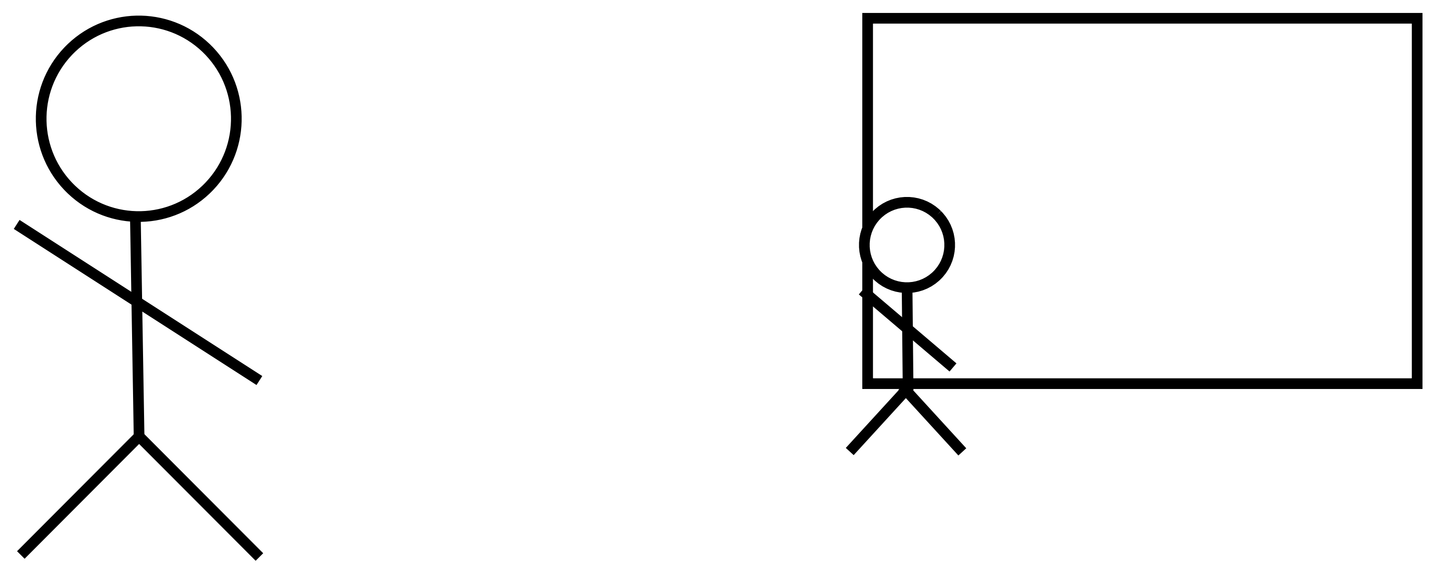


The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

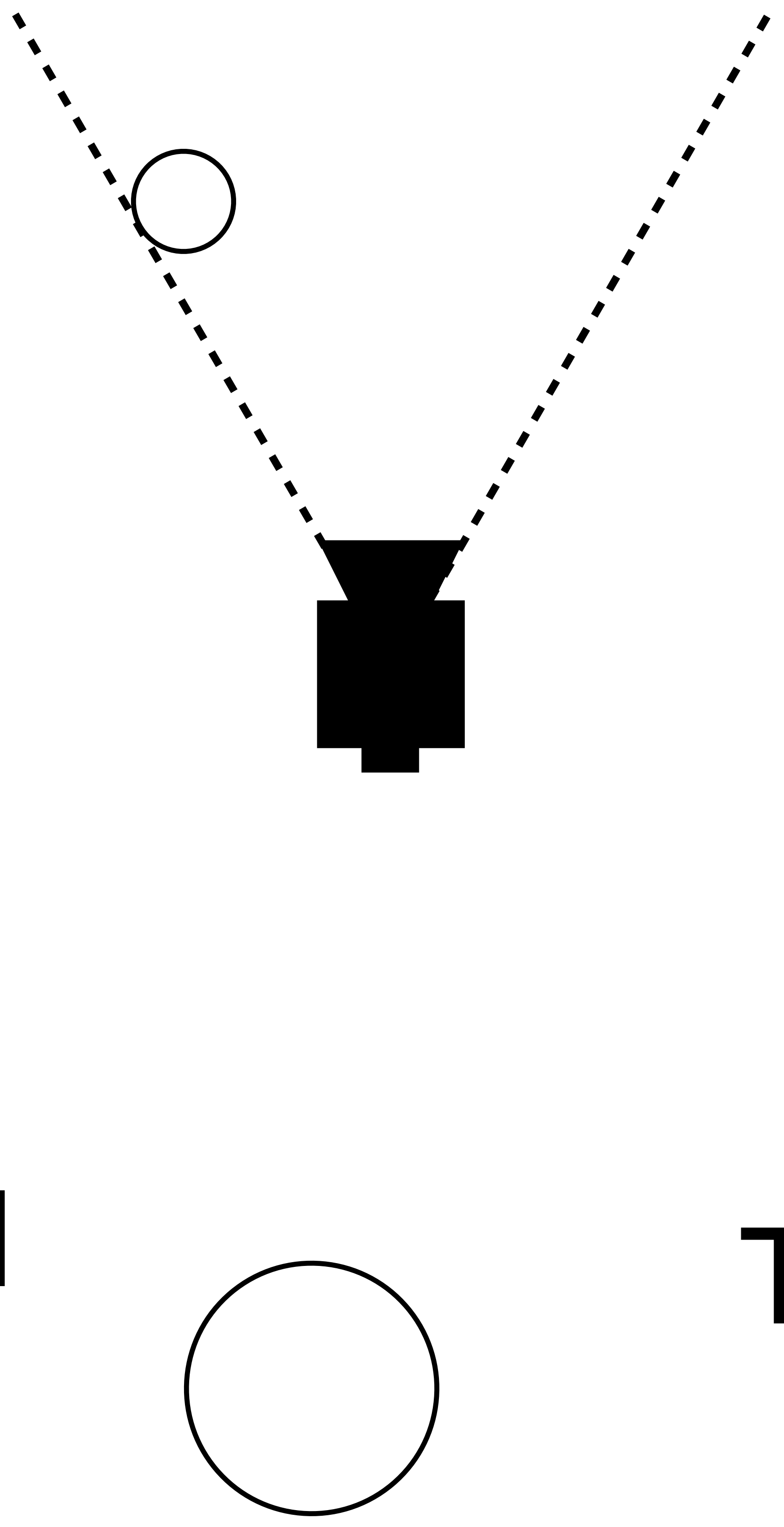
# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



Spherical View

Enemy has passed behind camera, warping it to the opposite side of the spherical view



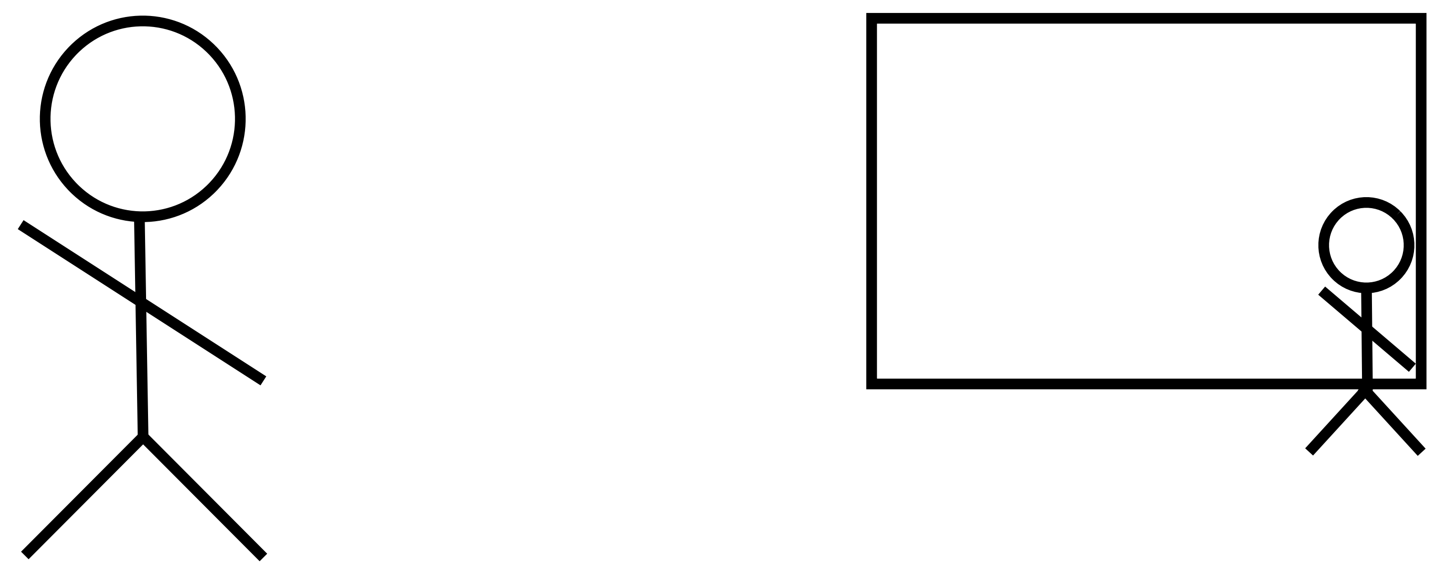
Top Down View

The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

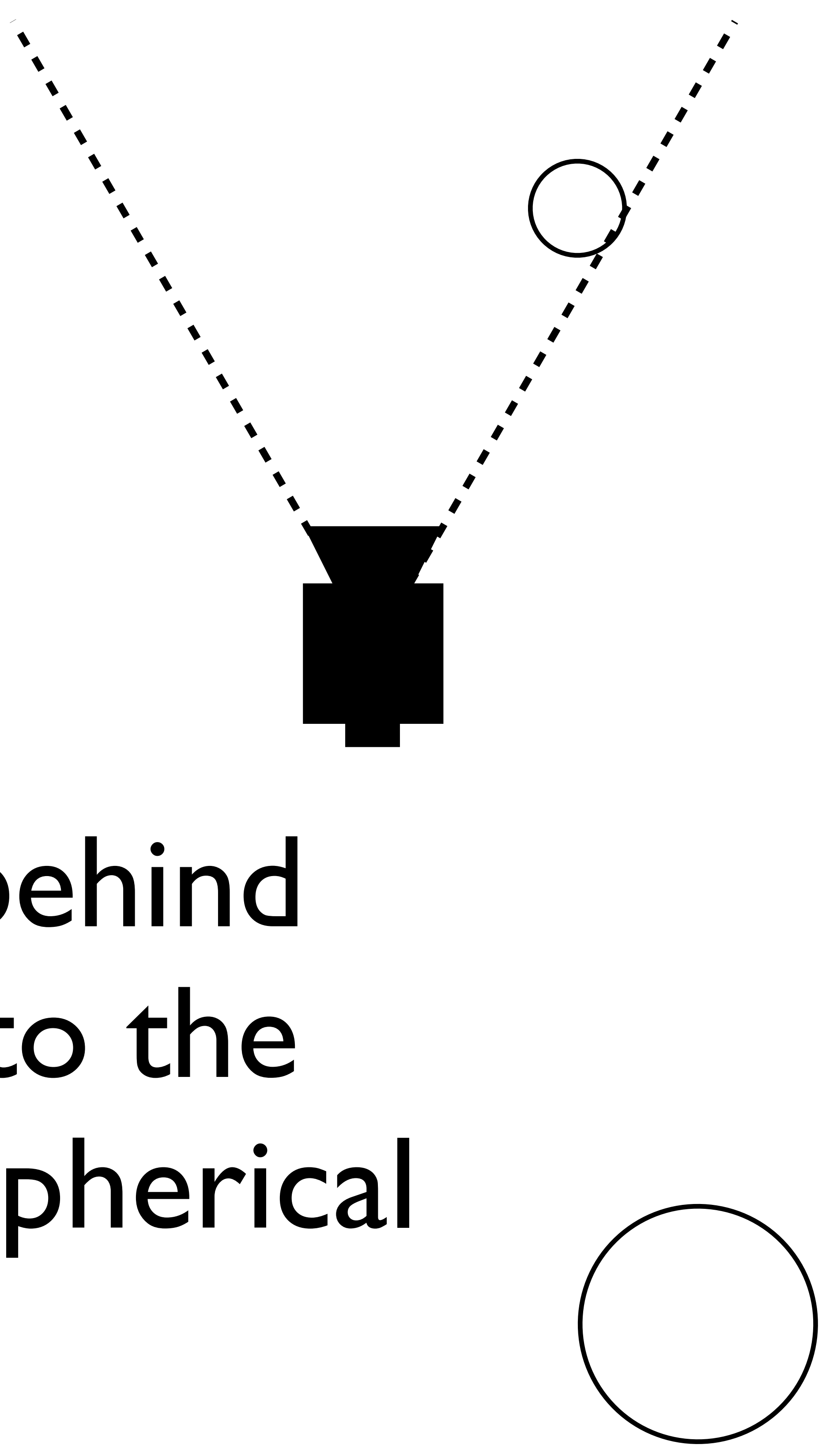
# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



Spherical View

Enemy has passed behind camera, warping it to the opposite side of the spherical view



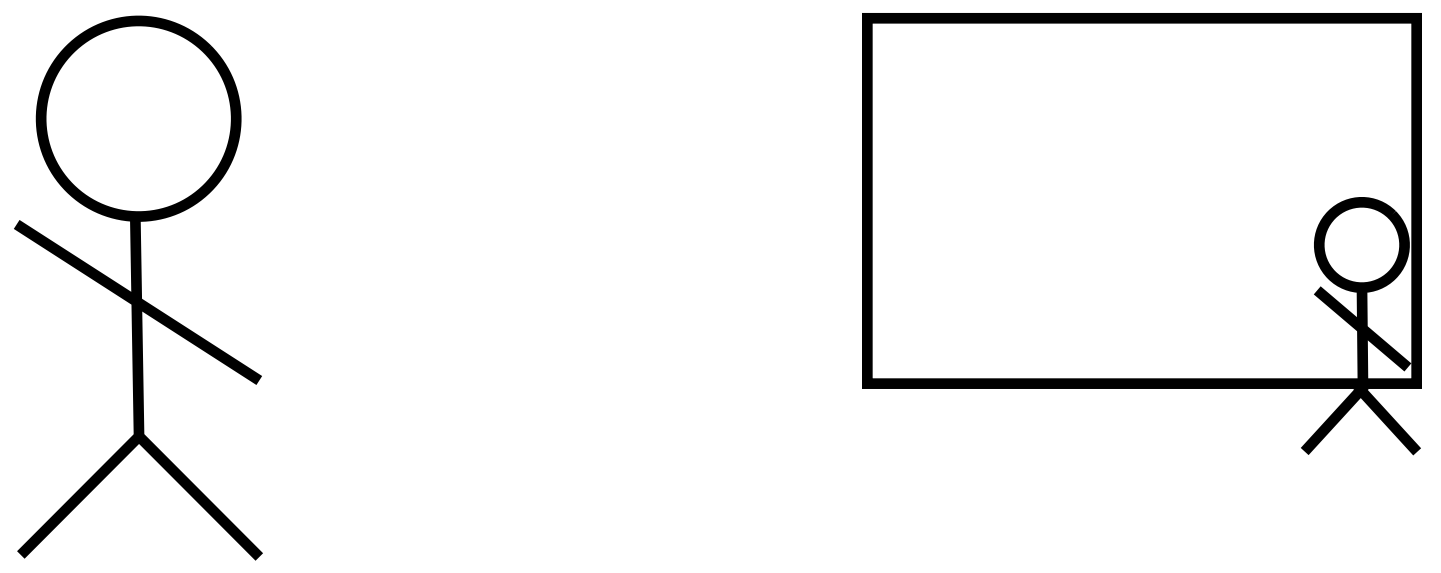
Top Down View

The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

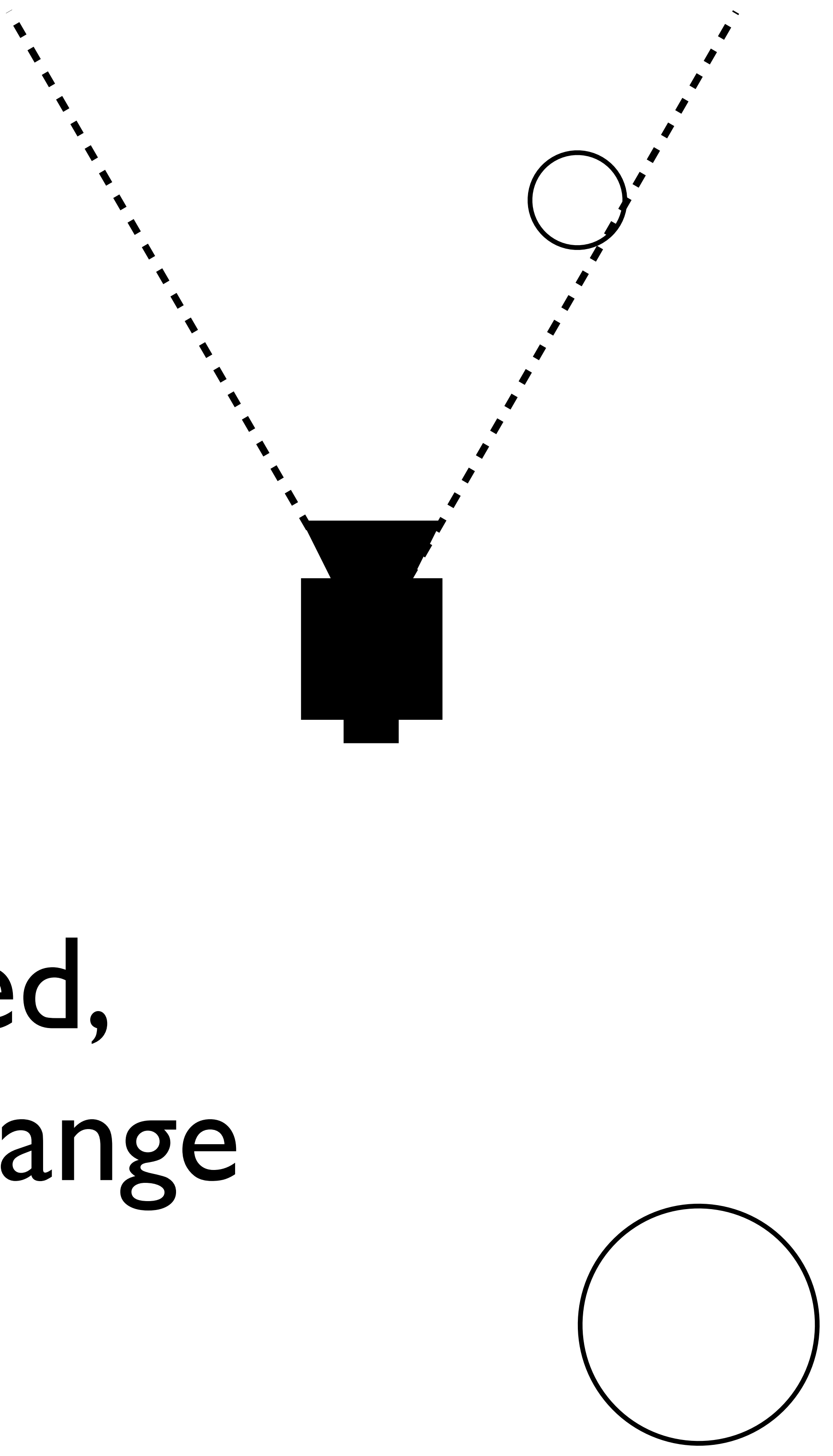
Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



Camera has moved,
causing enemy to change
sides again.

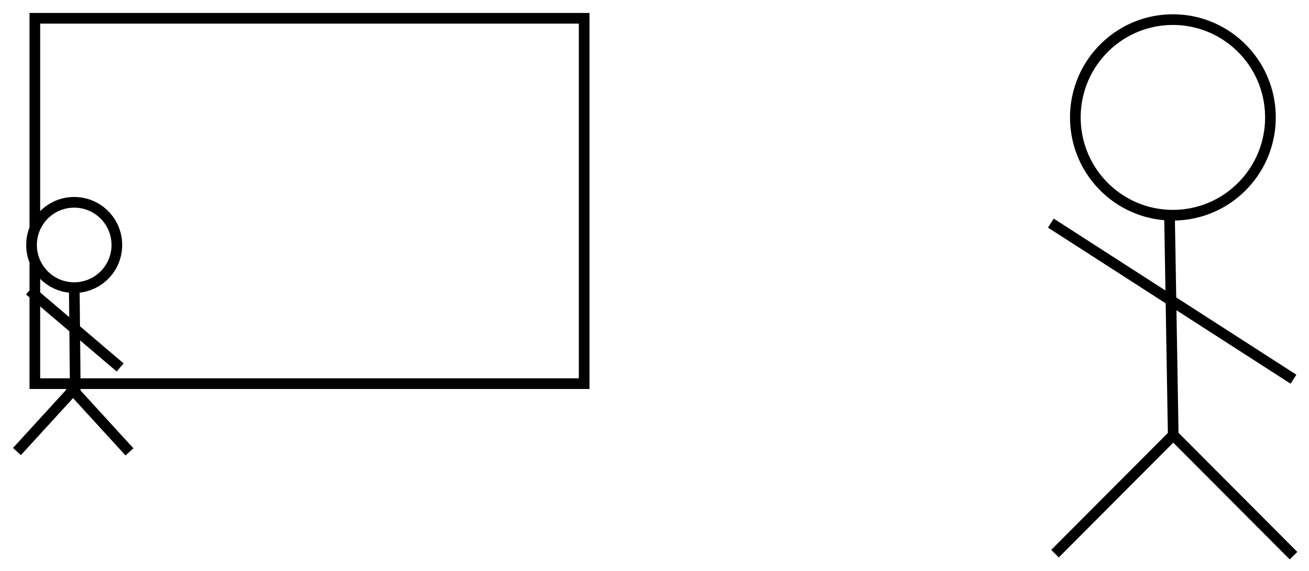


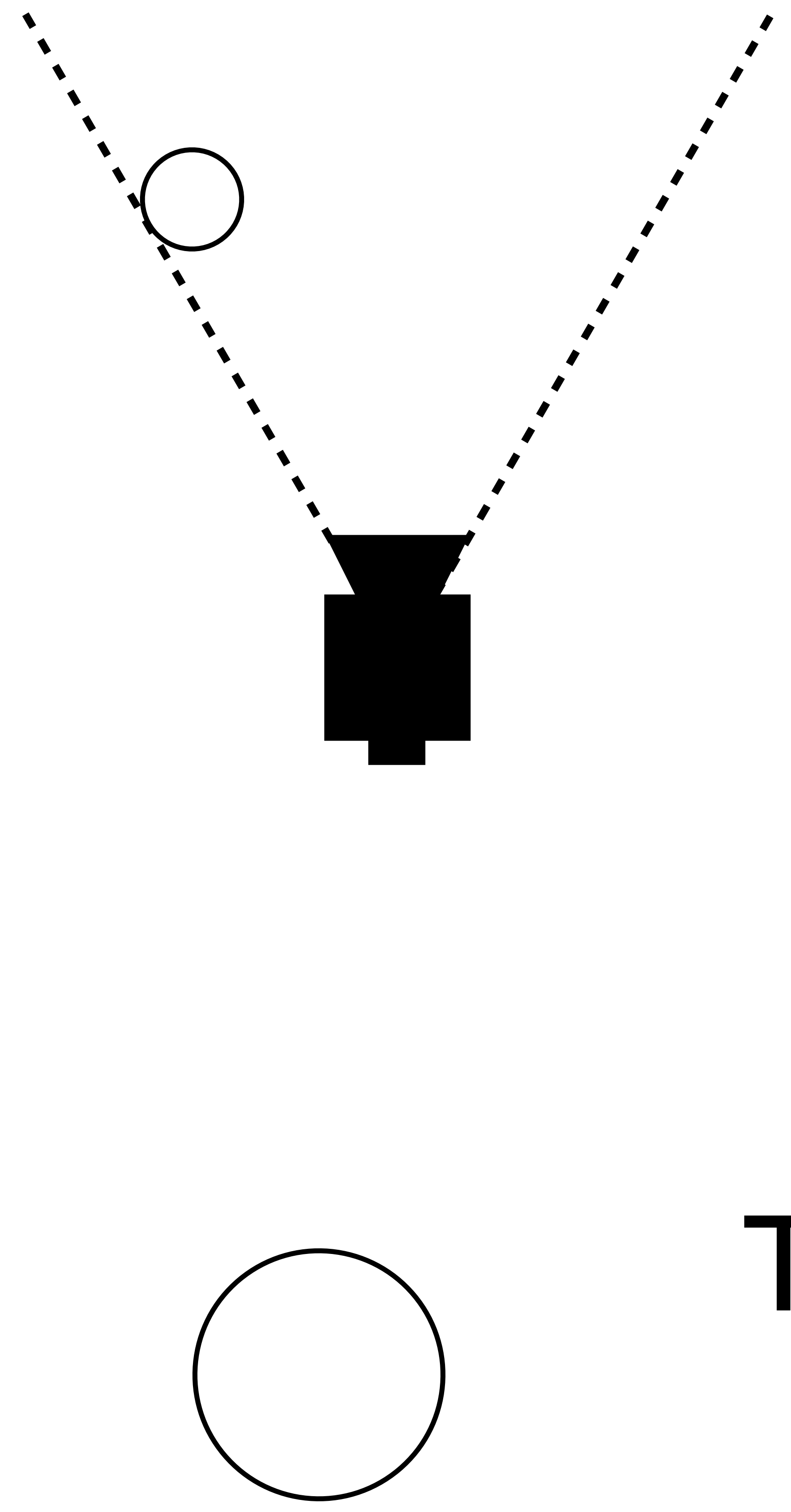
The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera





Camera has moved, causing enemy to change sides again.

Spherical View

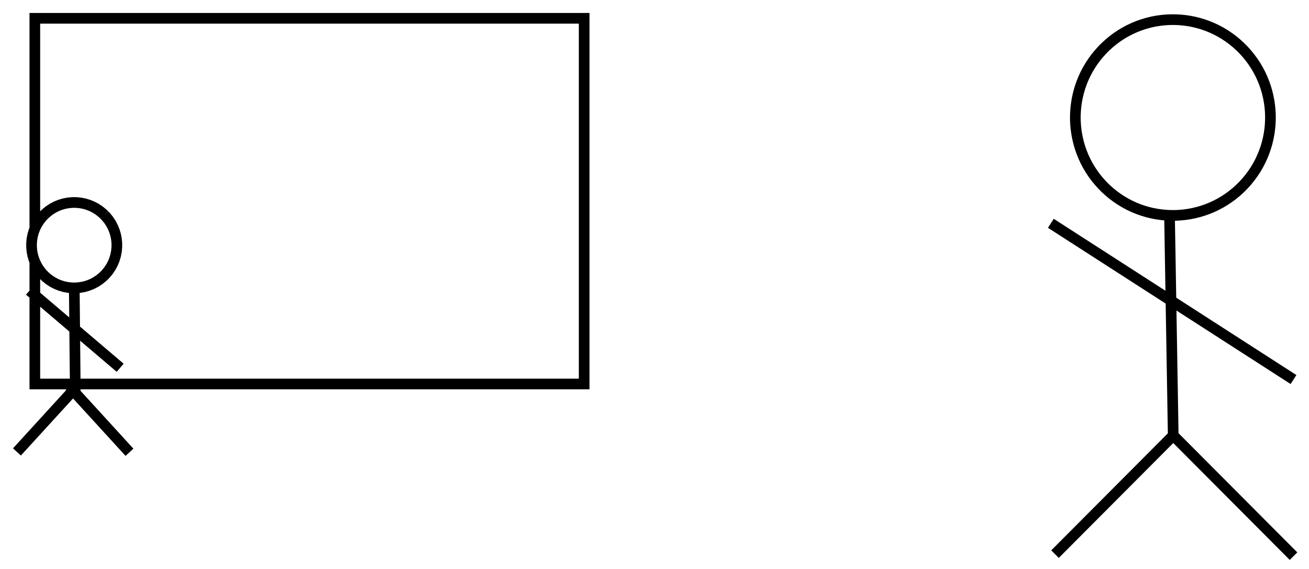
Top Down View

The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

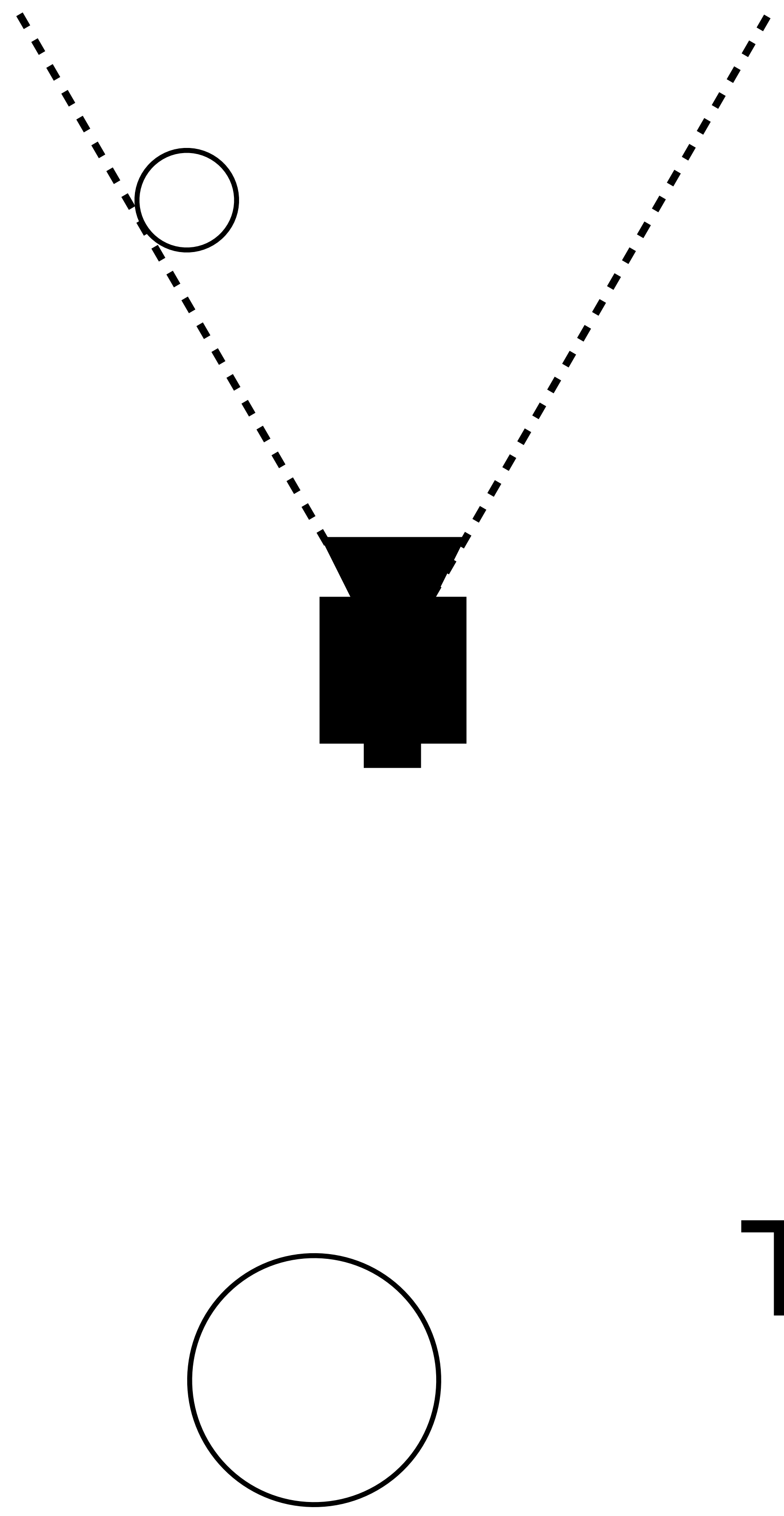
Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera



Camera is now oscillating between two states on alternate frames.

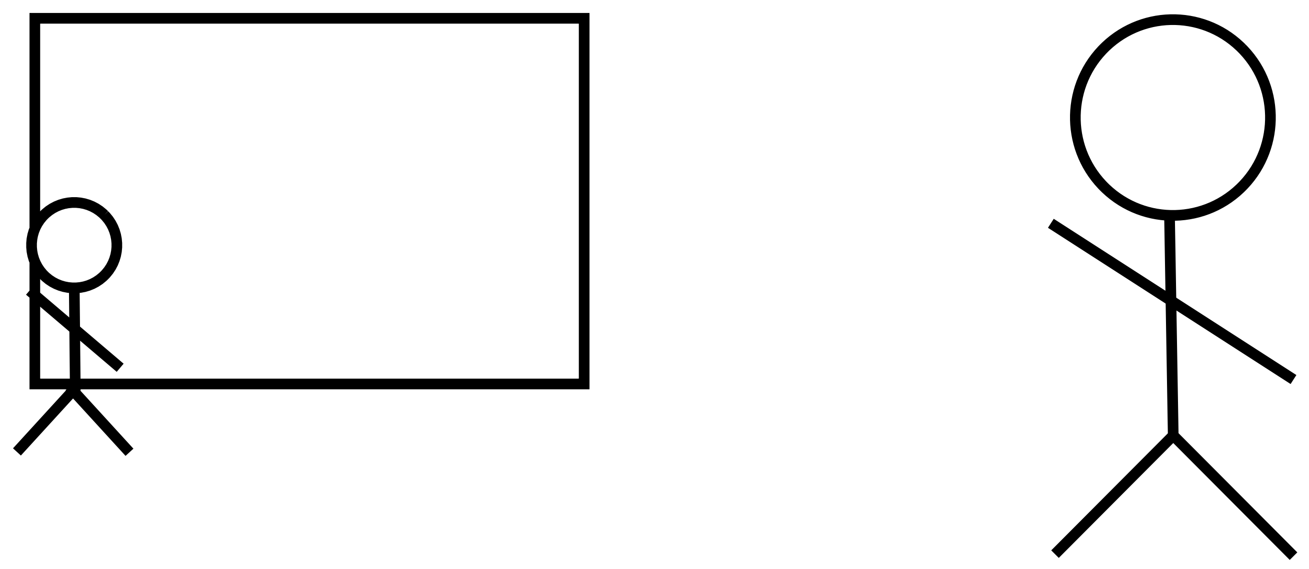


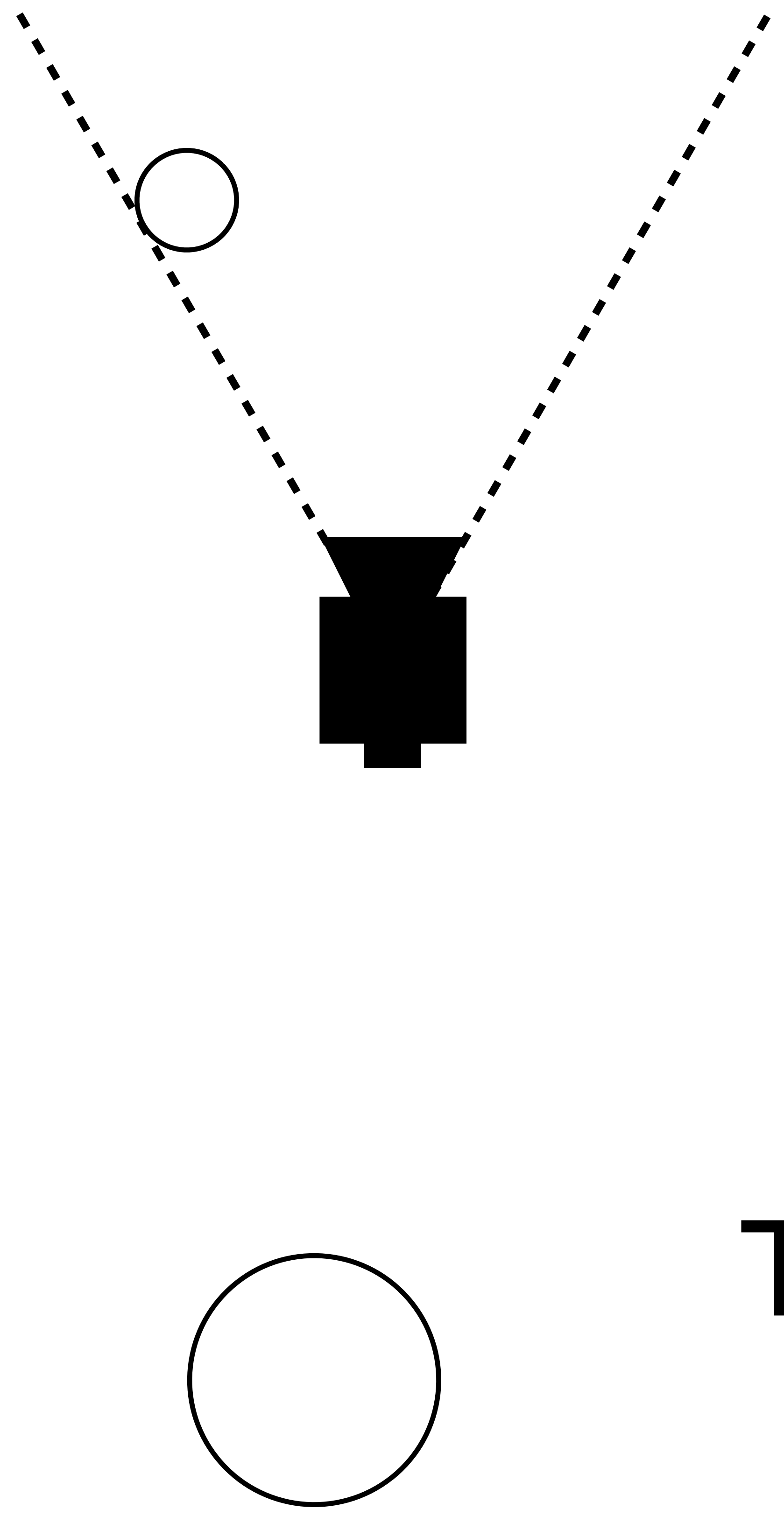
The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to -180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

# Targeting : Prioritised Framing

Problem : The spherical approximation breaks down behind the camera





Camera is now oscillating between two states on alternate frames.

This is not good!

Spherical View

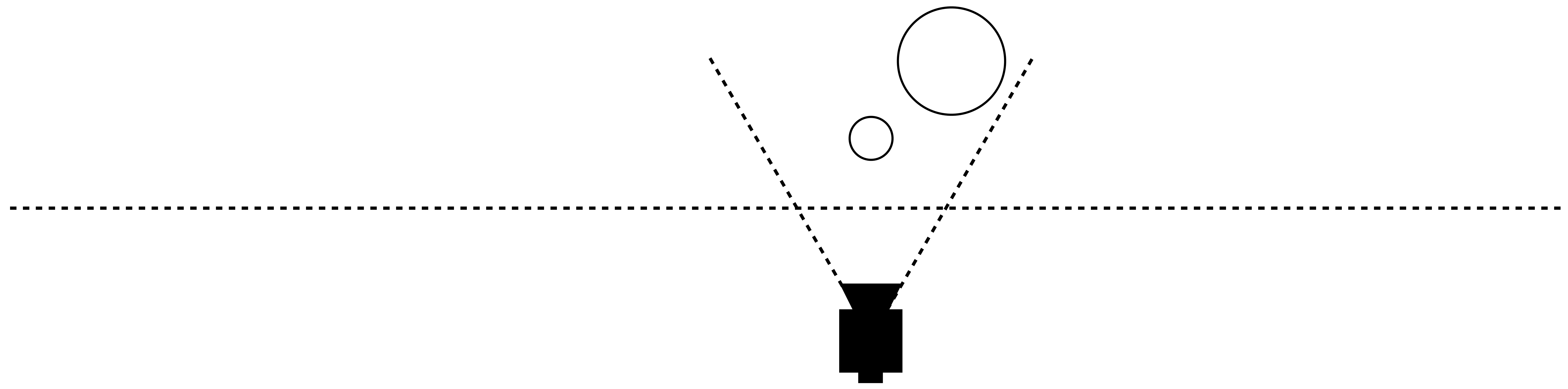
Top Down View

The second problem is little more tricky. Remember we're operating in spherical space here. There's a discontinuity directly behind the camera where +180 wraps to −180. Watch this. The target crosses behind the camera, and the camera pops to the other side of the player. Ouch. Worse, next frame it pops back again. Uh oh.

Previously this hasn't been a problem, because we've effectively guaranteed that the target (singular) never goes back there while the constraint is running. But now we've broken that promise. Or rather, we want to break it, because it doesn't make sense anymore.

## Targeting : Prioritised Framing

Solution : Apply minimum distance to target constraint, to targets

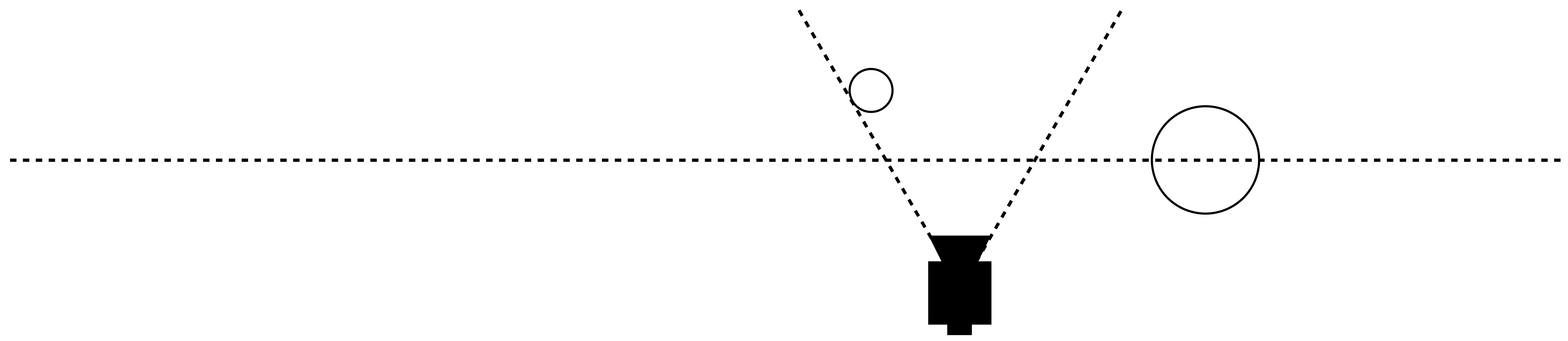


So how do we fix it?
*Well the solution is pretty similar to the weighting solution.
*We fix the logical position of the target, so that it never goes behind the camera. We clamp it to the plane parallel to the camera plane, defined by the minimum distance to target constraint.

Yes, much better.

## Targeting : Prioritised Framing

Solution : Apply minimum distance to target constraint, to targets

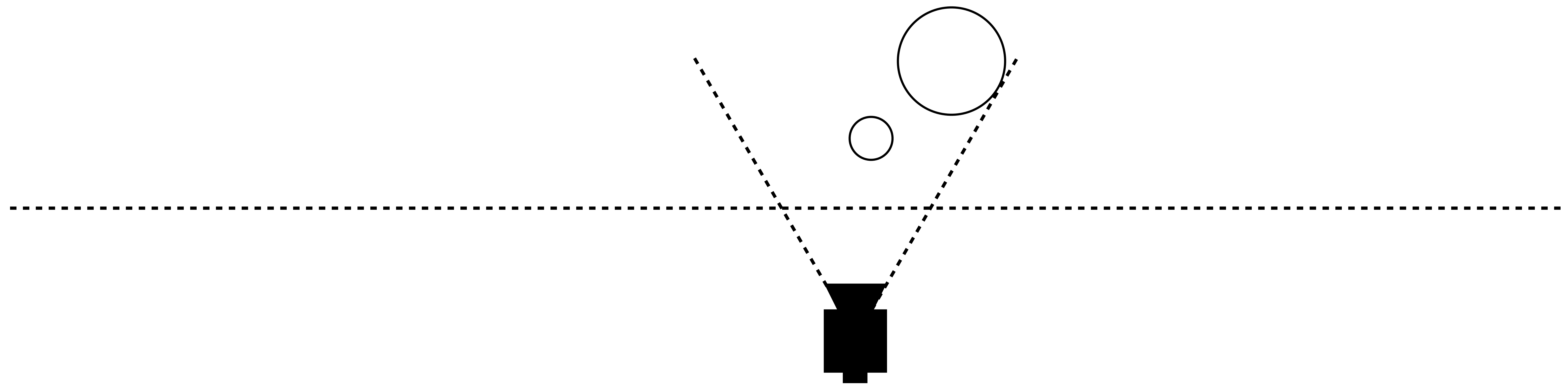


So how do we fix it?
*Well the solution is pretty similar to the weighting solution.
*We fix the logical position of the target, so that it never goes behind the camera. We clamp it to the plane parallel to the camera plane, defined by the minimum distance to target constraint.

Yes, much better.

## Targeting : Prioritised Framing

Solution : Apply minimum distance to target constraint, to targets

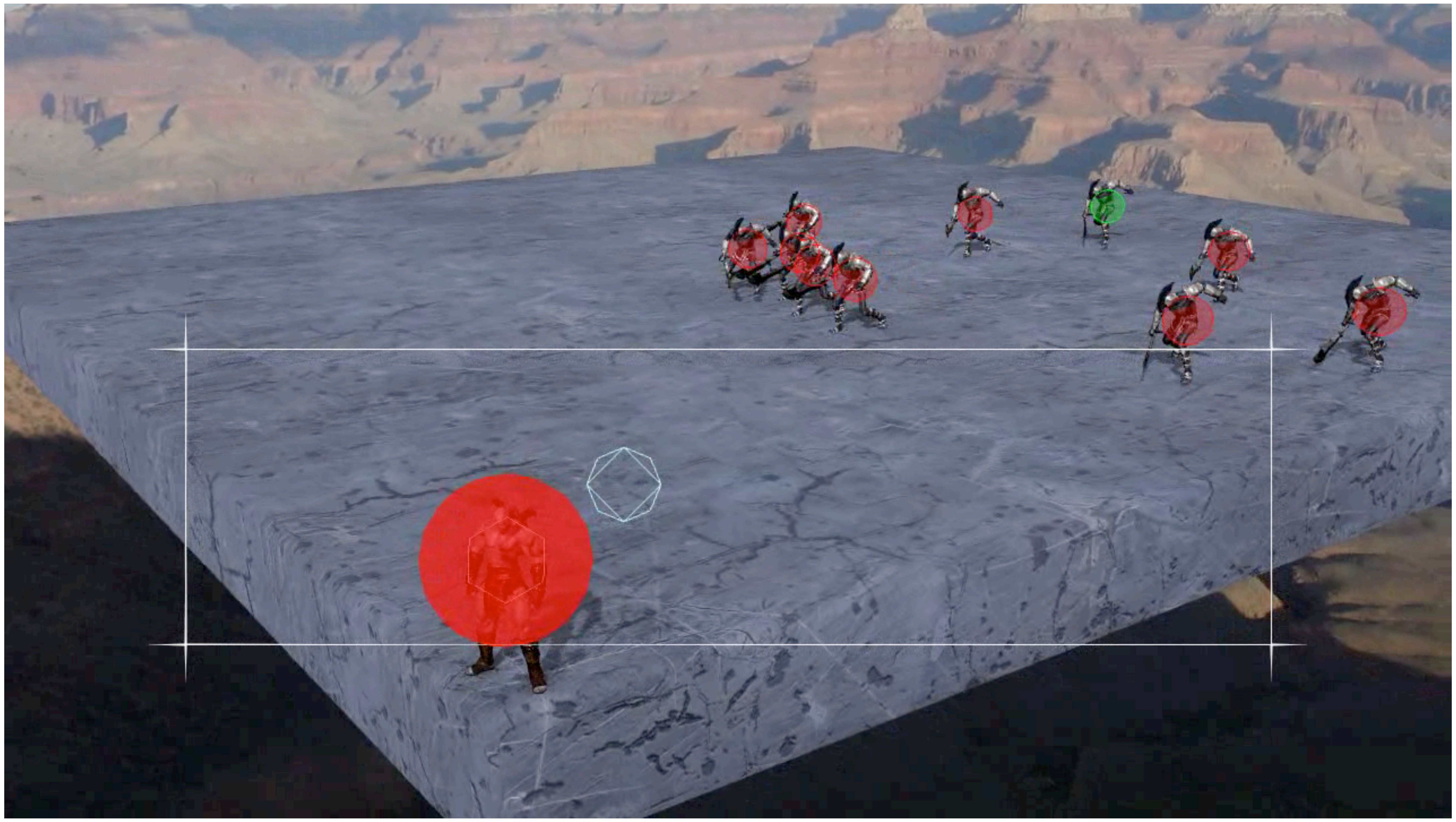


Much better!

So how do we fix it?
*Well the solution is pretty similar to the weighting solution.
*We fix the logical position of the target, so that it never goes behind the camera. We clamp it to the plane parallel to the camera plane, defined by the minimum distance to target constraint.

Yes, much better.

## Targeting : Prioritised Framing



So lets have a look at that in action. In this case, the solid red spheres only kick in when the constraint is applied.

And that’s pretty much it.

This came together fairly late in the development of God of War 2, but it worked pretty well, so we used it on all the human scale boss fights in the game.

In God of War 3 it’s used in almost every fight in the game.

# Overview

| | |
|---|---|
| • Selection | Environment, Combat, Scripting, Filtering |
| • Blending | Blend Tree, Weights, Modes, Parameters |
| • Dynamics | Animated, Dynamic, Combat |
| • Targeting | Hero, Ares, Damping, Weighting, Prioritisation |

So there we go, that’s pretty much everything. I don’t think I left anything out this time.

Ah crap.

# Thank You

- Any Questions?

email: phil_wilkins@playstation.sony.com

I’d like to thank the camera designers, all of whom have been invaluable in helping me develop this system.
Jessica Brunelle for the original God of War
Mark Simon and Steven Peterson for God of War 2
who were joined by Matt Fallows, and Josh Harrison for God of War 3
Tom Miller who wrote the original combat camera
the rest of the code team past and present,
and everyone else on the team
everyone who bought the games
and you for listening to me talk today

Any questions?