"An ideal virtual camera system, regardless of genre, is notable by the *lack* of attention given to it by the viewer"

From Introduction to:
M. Haigh-Hutchinson,
Real-Time Cameras: A Guide for Game Designers and Developers,
Morgan Kaufmann 2009.

Note: if you don't notice camera, then it is working well!

Show God of War clip (with sound off to minimize distraction)

Designing and Implementing a Dynamic Camera System

Phil Wilkins
Sony Playstation Entertainment
GDC '08
[edited by C. Rich]

Phil did camera work for God of War.

Implementing something from this lecture in your game counts towards an “A”.

Objectives

- Flexible
- Designer driven
- Smooth
- Not require player intervention
- No collision

We are *not* talking about a simple, automatic “following camera” as comes with Unity.

Designer-driven---same division of labor as we discussed on Friday for AI, i.e., designer provides CONSTRAINTS and system works
automatically within those constraints.

No collision with the environment. By which I mean that it is up to the designer to constrain the camera such that it doesn’t go through walls.

Whenever I’ve tried resolving camera collision with the environment in the past, it’s always introduced pops, or it gets hung up on geometry.

Collision geometry is designed to constrain the player, not the camera.

Overview

- Zoning
- Dynamics
- Blending
- Rails

Zoning deals with the use of a spatial database to select cameras,
Dynamics is the calculation of a single dynamic camera
Blending is where we smooth out the transitions between cameras
Rails deals with constraining the camera to a path

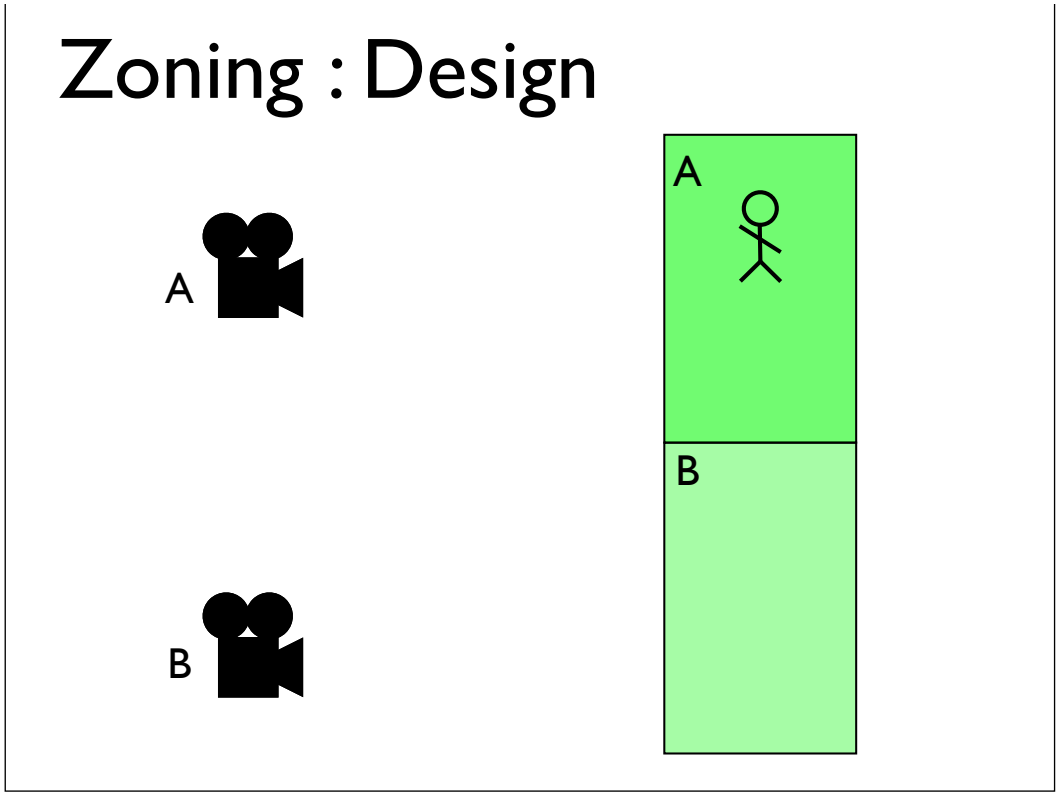
Any one of these techniques counts as optional tech (though you probably
want to use two of them together).

Overview

- Zoning
- Dynamics
- Blending
- Rails

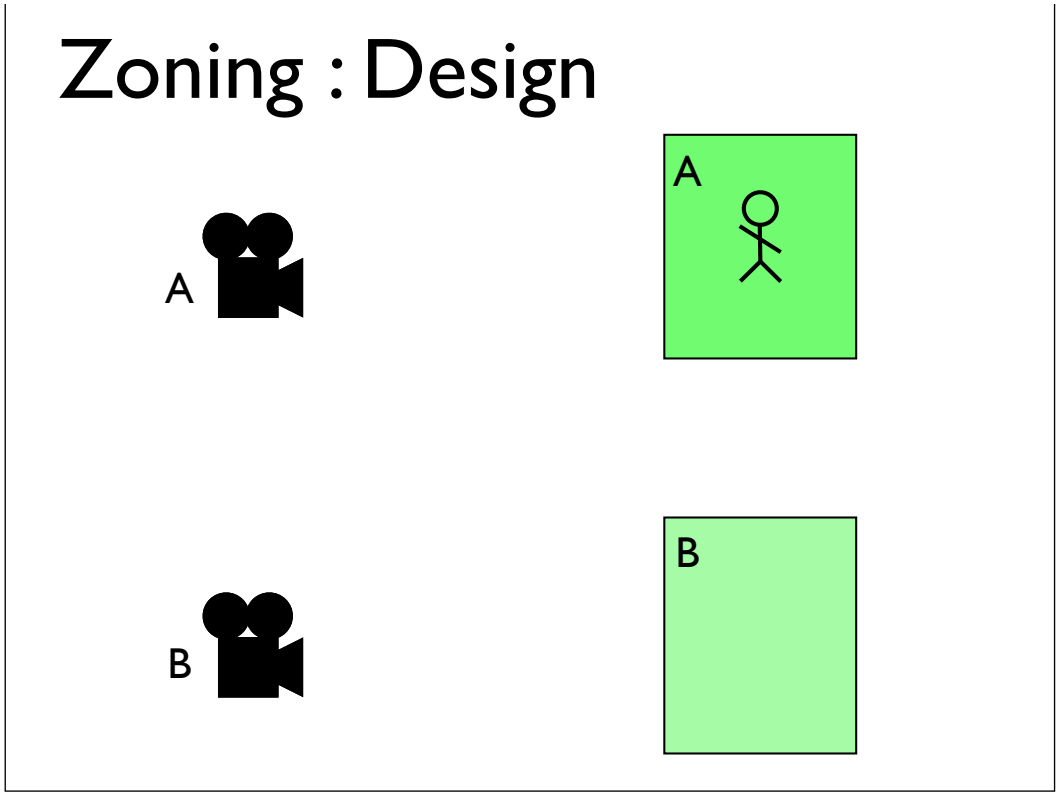
Zoning : Objectives

- Stationary Cameras
- Chosen by Player Position



The game has a spatial database of zones. To start with, let's assume that the zones are a TILING of all space.

Each referencing one or more cameras, although for the moment we'll just deal with the case where a zone references a single camera.

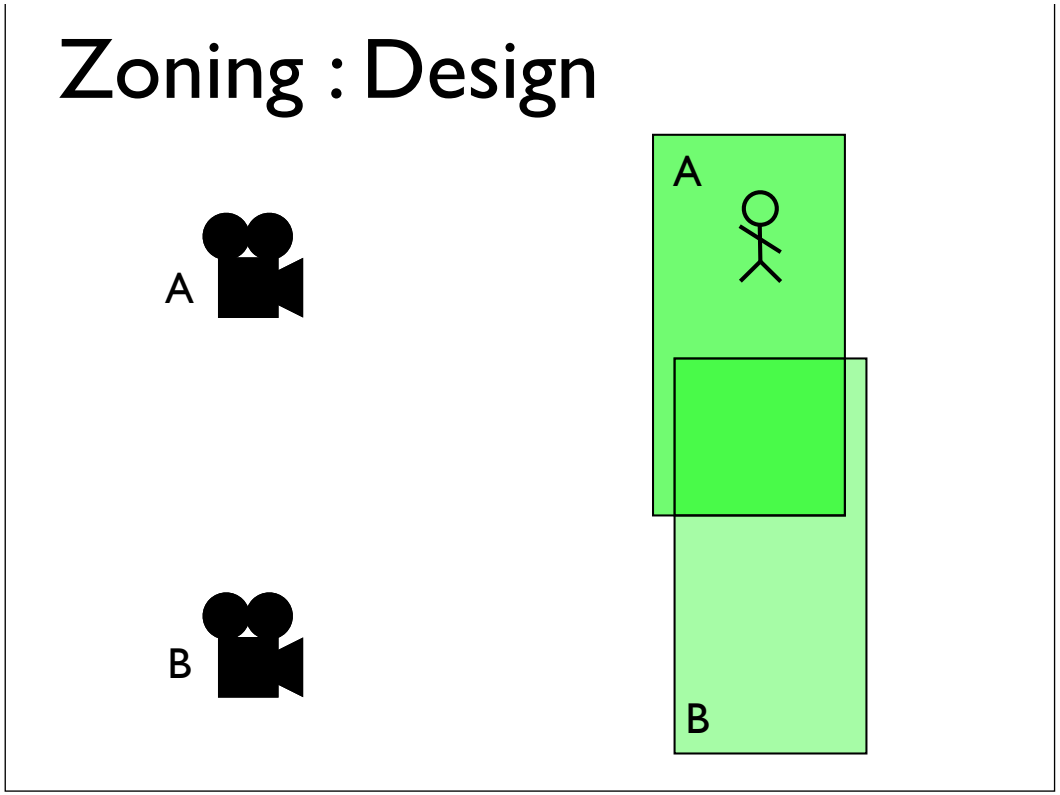


Notice zones do not touch here.

When the player moves INTO a zone that references a camera that isn't the currently active one, we activate that camera.

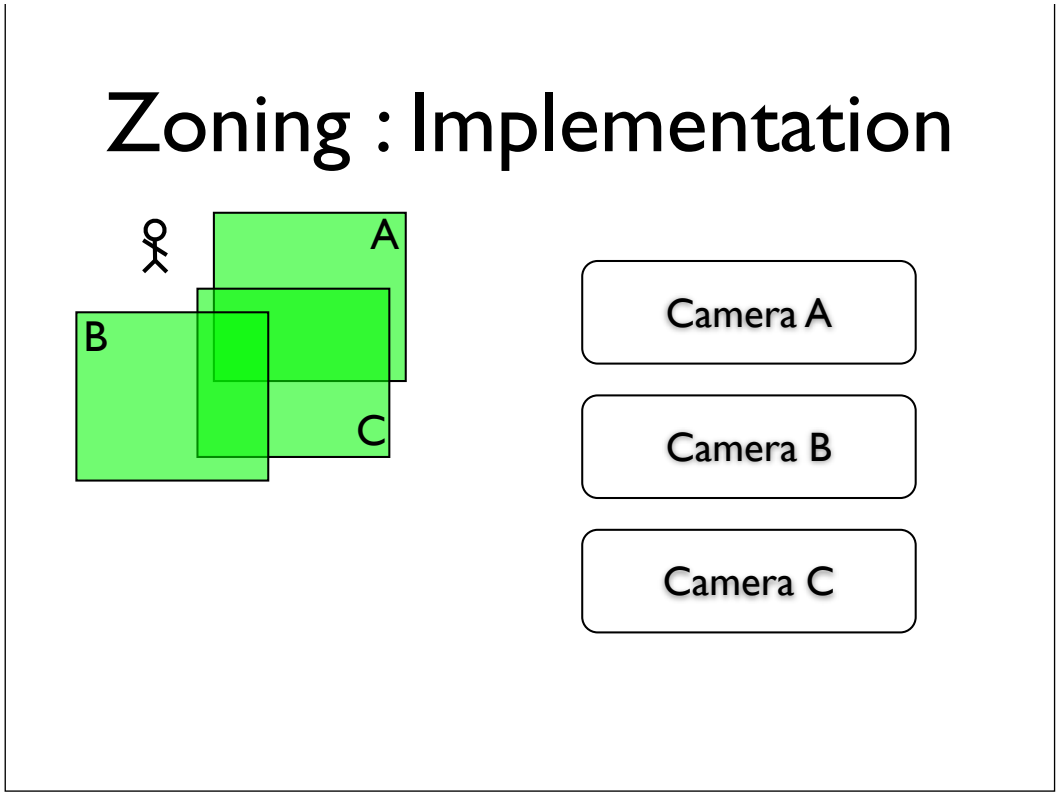
This allows us to loosen up the boundaries between cameras.

This also adds hysteresis (stickiness--why? Because camera in middle gap depends on why direction you came from) to our system, by playing with the boundaries of the zones.



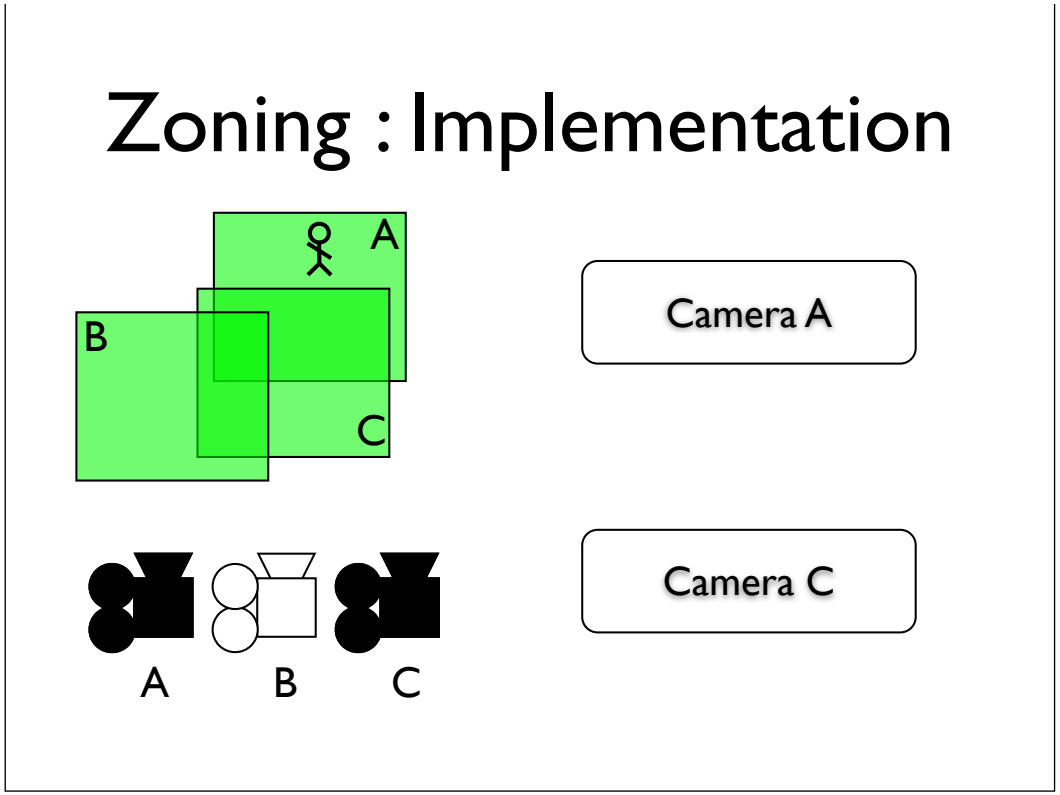
Alternatively we can overlap the zones.

In which case, when we enter the overlap space, we always change to the new camera.



So at runtime, each frame, we query the spatial database of zones, and get back a collection of camera references.

Since we can’t, or don’t want to make too many assumptions about the zone database, we treat the results as essentially unordered.



A naive implementation would just compare the results from the query, against the currently active camera, and if there's a difference, swap to the new camera. This works fine, unless your zones overlap.

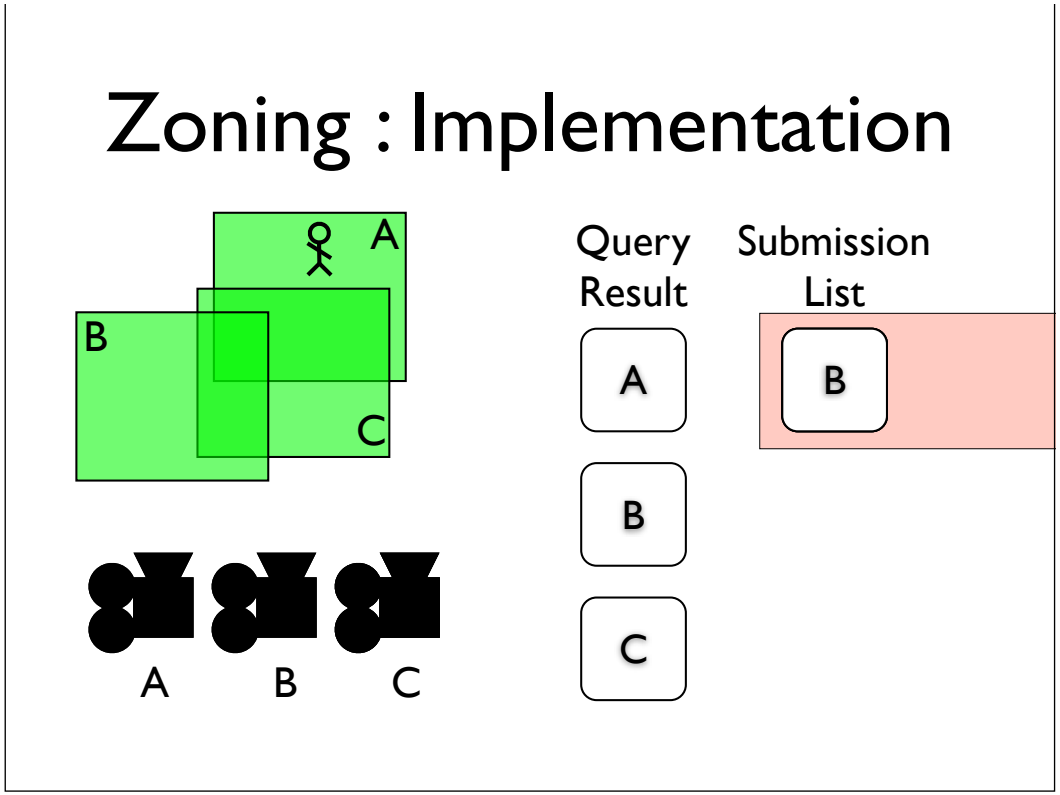
In this example, when we move from zone A into the overlap between zones A and C, we start off right, by switching to camera C,

but next frame, we're still in the overlap, still getting A and C back from the query, but now C is the active camera, and A looks like it's new, so we swap to A. Until next frame, when... well you get the idea, we're going to alternate between the two, which really isn't what we want.

So we need to keep track of some more information than just a bag of cameras...

Zoning : Implementation

- Submission List
  - List of all cameras that were submitted last frame.
  - Used to distinguish newly submitted cameras from old ones
  - New cameras inserted at top
  - Effectively sorted by age

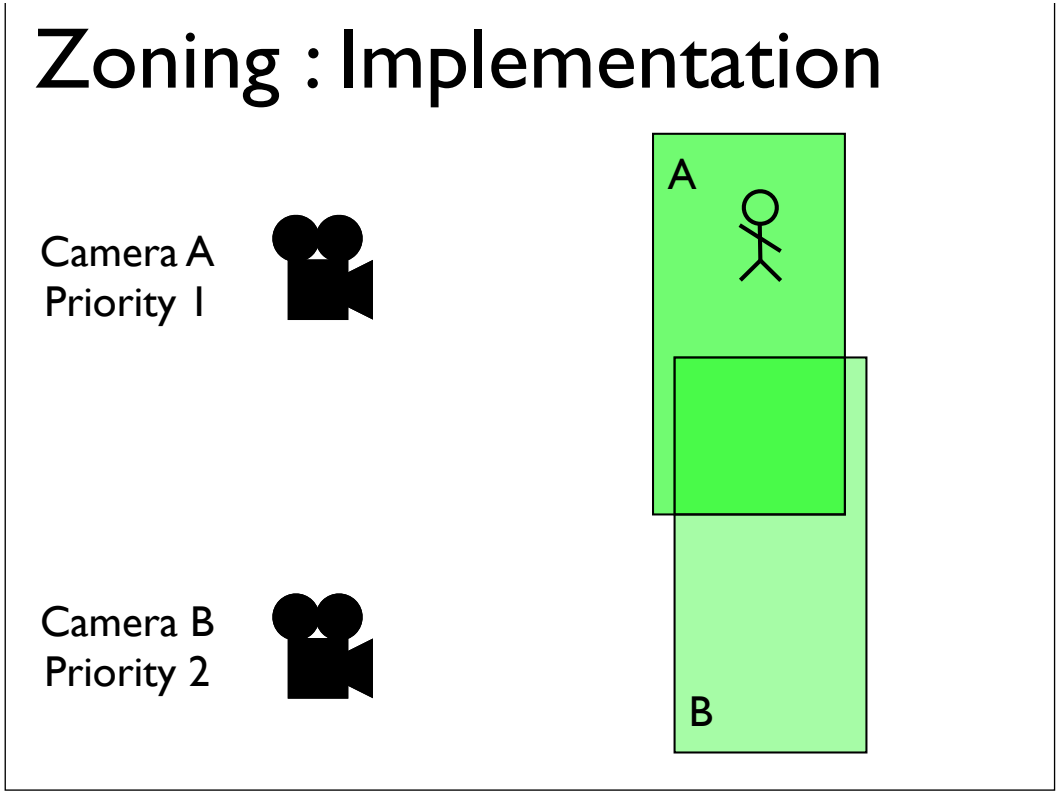


* Camera A is not in the submission list, so we'll add it.
* The top item in the submission list has changed, so we'll switch to that camera.
* When we move here, we get cameras A and C back from the query.
* Camera A is already in the submission list, but camera C isn't, so we add it to the top of the list.
* The top entry has changed, so we start that camera, camera C.

Next frame, we get cameras A and C back from the query again, but both are already in the submission list, so we don't need to change camera.

* Now we move here, to the overlap between all three zones, and we get all three cameras in the query.
* Cameras A and C are already in the list, but camera B isn't, so we add it at the top.
* The top entry has changed, so we start that camera, camera B.
* When we do the next move, camera A no longer appears in the query results,
* so we remove from the submission list, but the top item doesn't change, so the camera stays the same
* finally we move out of zone B, camera B disappears from the Query results,
* and we remove camera B from the Submission list.
* the camera at the top has changed, so we start that camera

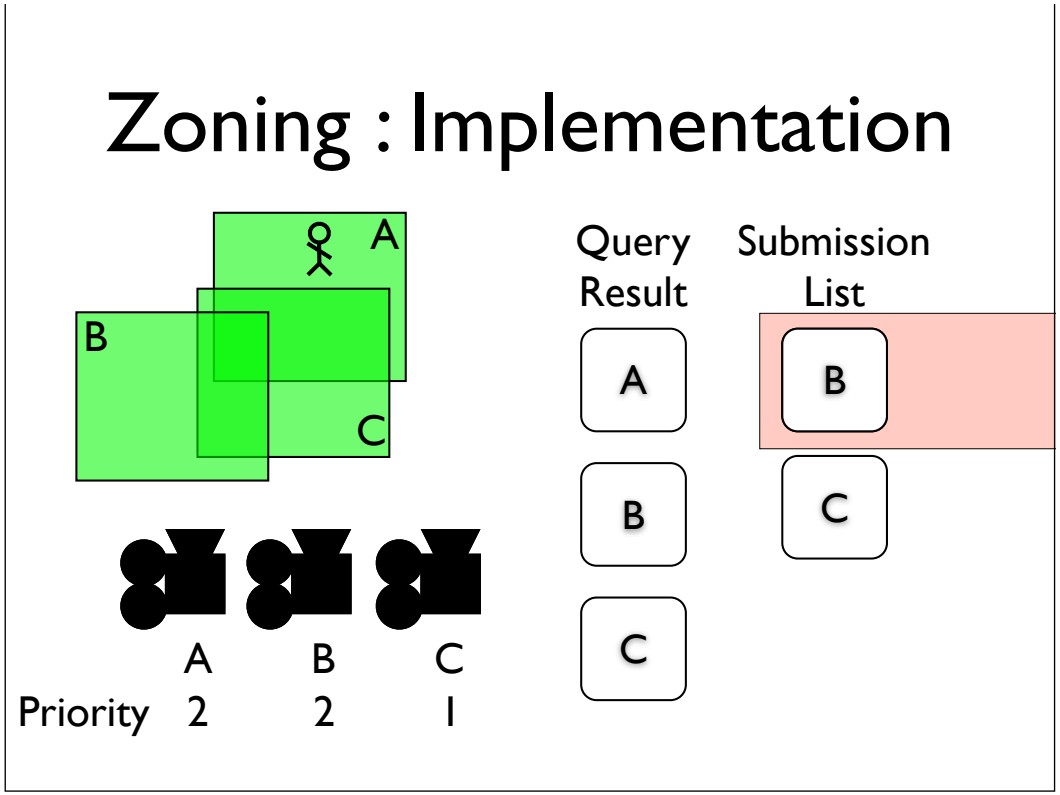
Zoning : Implementation



Another wrinkle for finer designer control: Camera priorities (2 is higher)

Higher priorities always override lower ones.

So in this example, we can see that, whenever the player is in zone B, camera B is active, because when we’re in the overlap, it has a higher priority.



Submission List with Priorities:
* Just like before, we start in zone A,
* and end up starting camera A
* but when we move into zone C
* we insert camera C into the submission list below camera A. **This is because in order to respect priorities, we maintain the submission list in priority order**. So now this time, the top hasn't changed, and we don't change camera.
* when move into zone B
* we insert camera B above camera A, because it's of equal priority, and between cameras of equal priority, we want to retain the behaviour we had before we introduced priorities.
* so now we have a new camera at the top, so we change to that camera
* moving out of zone A results in the same behaviour we saw in the last example, as priorities have no effect on removing entries from the submission list

Zoning Implementation

- Submission List
  - Insert and delete entries to match query results
  - Unless query result was empty
  - Sorted by priority
  - Then by age
  - Top entry is active camera

SUMMARY:

the submission list contains the current set of cameras up for consideration

insert and delete entries to match the current query results, assuming we got any

If the query was empty, then we hold the previous frames submissions (never want NO camera!)

Sorted by priority, then age, or rather, by how recently the camera was submitted

Which results in the newest, highest priority, camera sitting at the top of the list.

Overview

- Zoning
- Dynamics
- Blending
- Rails

Overview

- Zoning
- Dynamics
- Blending
- Rails

Dynamics : Objectives

Control following three properties of Player Avatar as it appears on the screen:

- Position
- Size
- Angle

So when we talk about dynamics, we're really talking about moving and orienting the camera to control certain display characteristics of the player avatar AS IT APPEARS ON THE SCREEN.

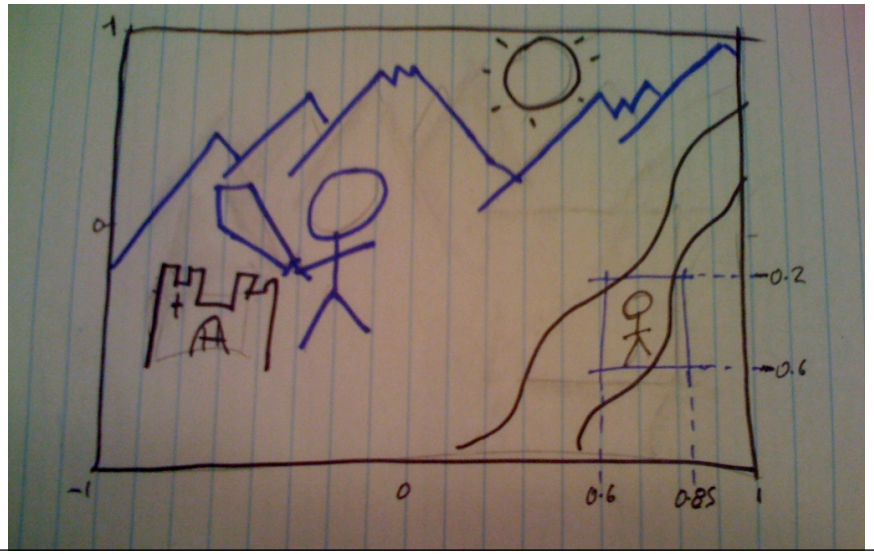
[GO TO BOARD TO ILLUSTRATE]

The players position on the screen

The angle that we are looking at them from.

and their size, which is a function of their distance from the camera, and the Field of View

Dynamics : Design



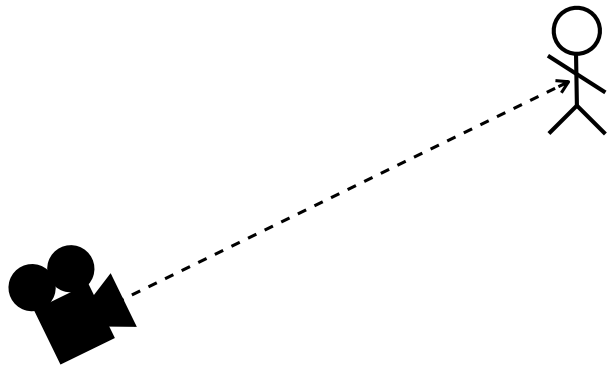
In order to control the players position **on the screen**, we define an area of the screen within which it is “safe” for him to move. Safe to move without having to move the camera to keep him in that zone.

We define a rectangular space on the screen, known as the Safe Zone. (If we want the player to always be at a particular position, we can shrink the boundaries down to that point.)

This is represented to the designer as a pair of ***resolution independent*** co-ordinates

...and at runtime we can overlay the safe zone on the real time display

Dynamics : Design

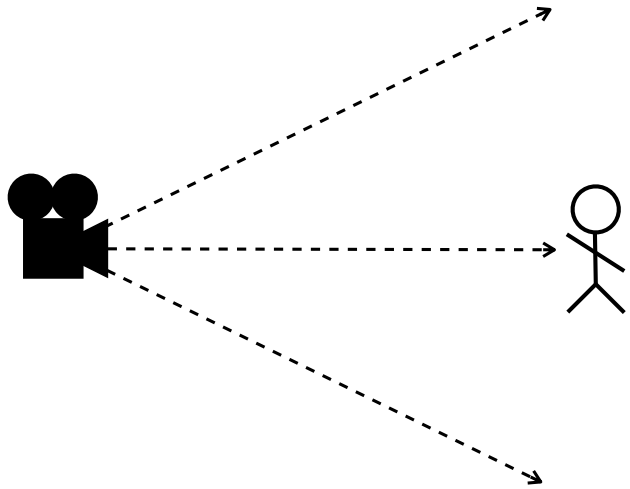


Both the position of the player on the screen and the angle at which we view him depend on the **angle** between the camera and the player.

We can specify the angle that we're viewing the player from as a fixed value.

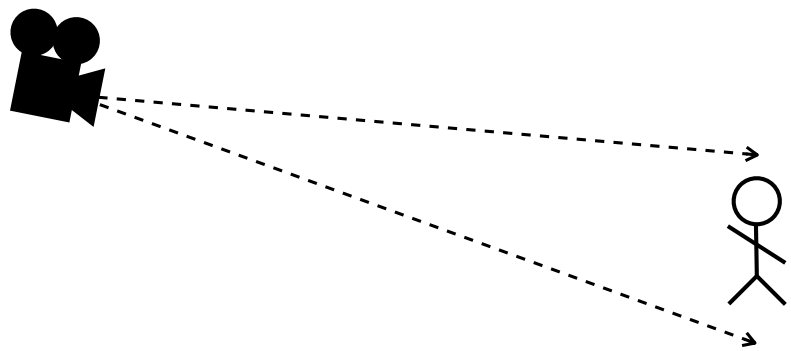
But notice camera moves around a lot (background moves), which can be disconcerting...

Dynamics : Design



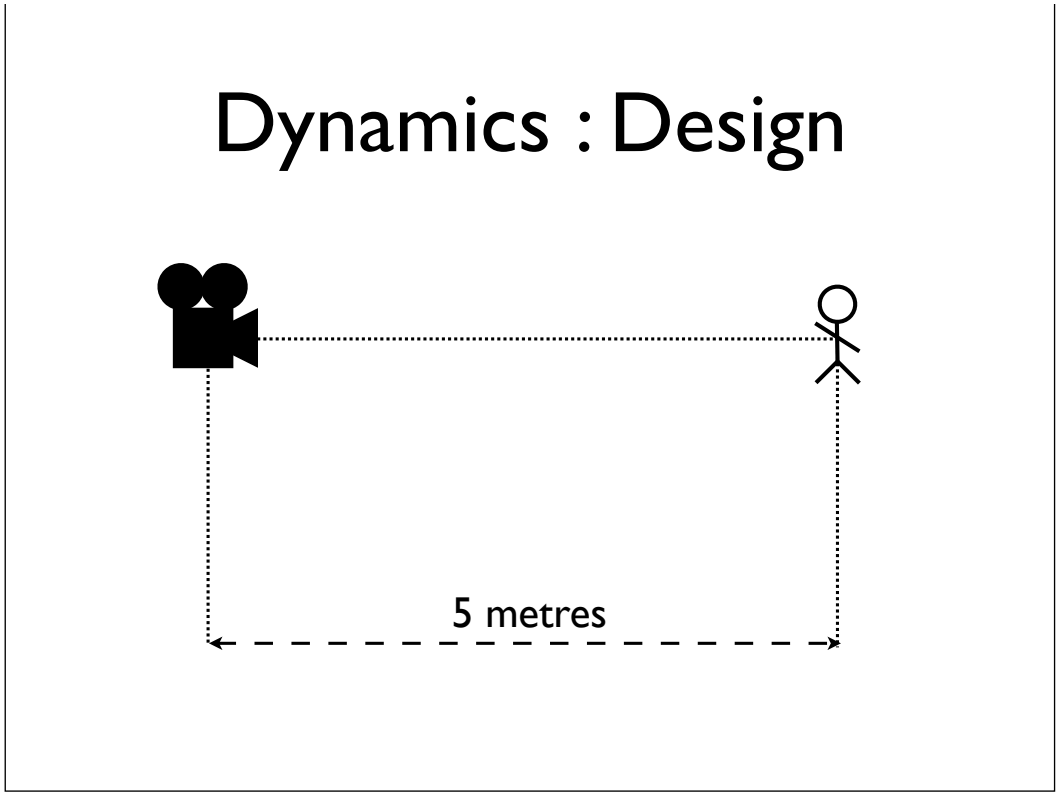
Slightly more flexibly (camera & therefore background doesn’t move so much), is to calculate it relative to a fixed position in space.

Dynamics : Design



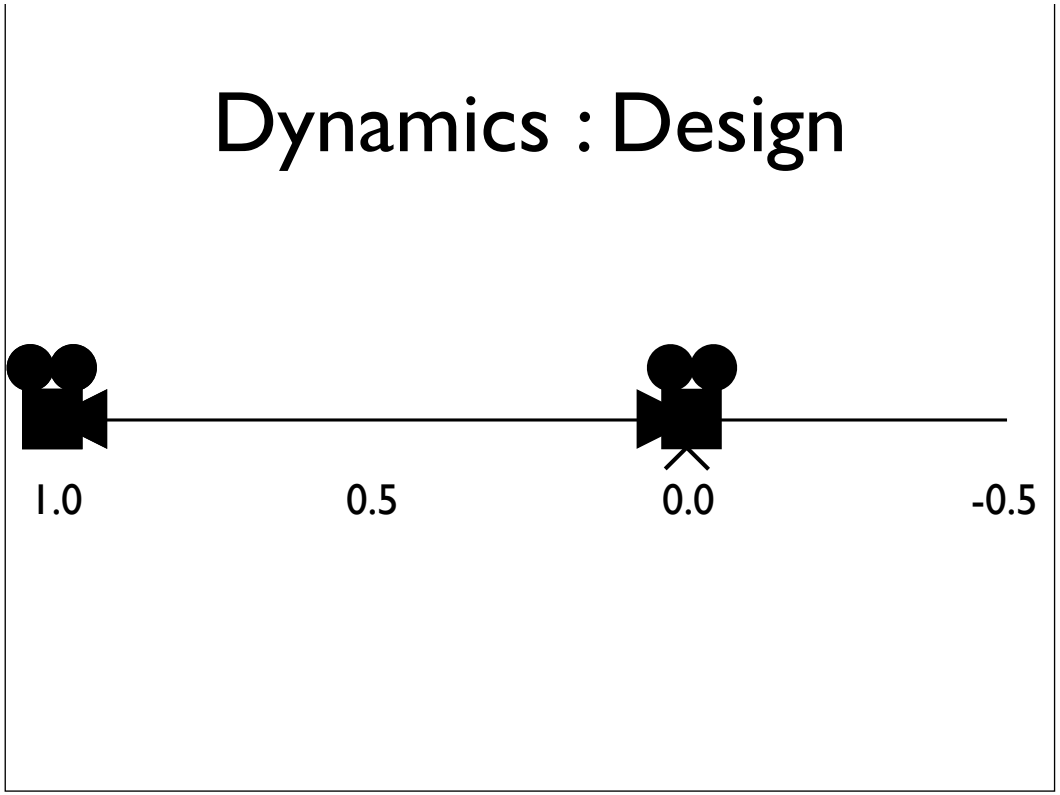
...and we can constrain it to within a fixed range before it moves.

plus or minus a fixed amount defined in the cameras attributes.



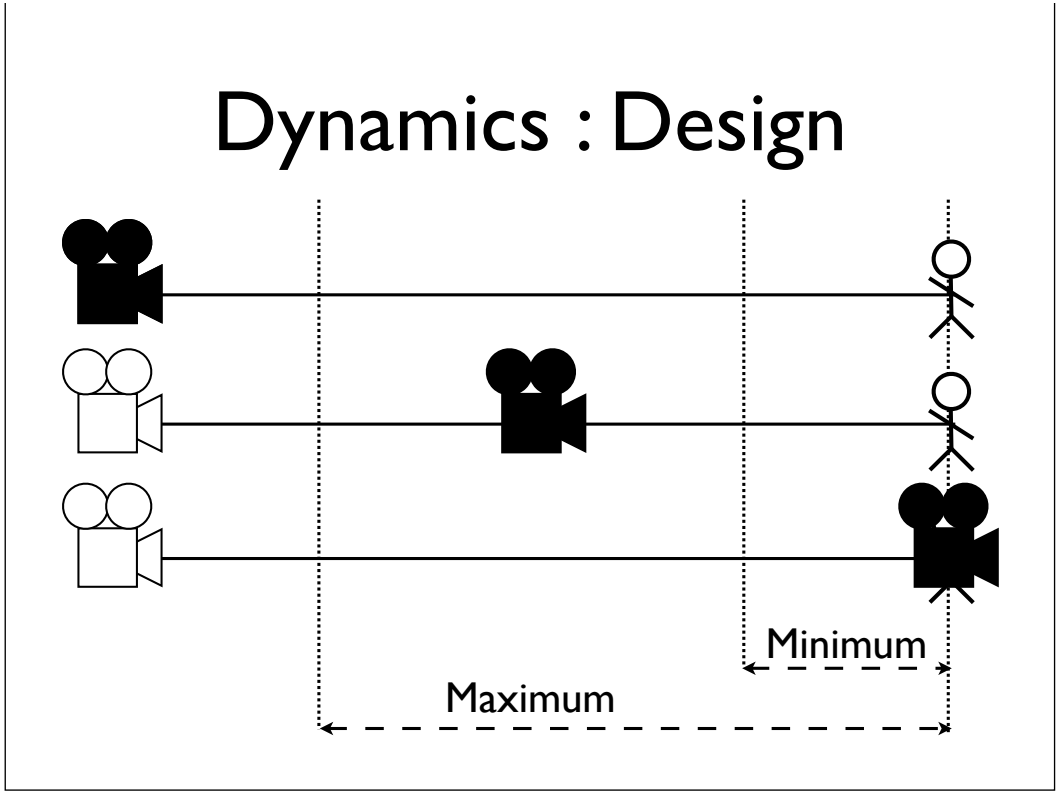
Finally we control the **size** of the player on screen, by controlling his distance to it.

The simplest way of specifying this, is to fix it to a set value.



Or we can specify it as a proportion of the distance from the camera node to the player

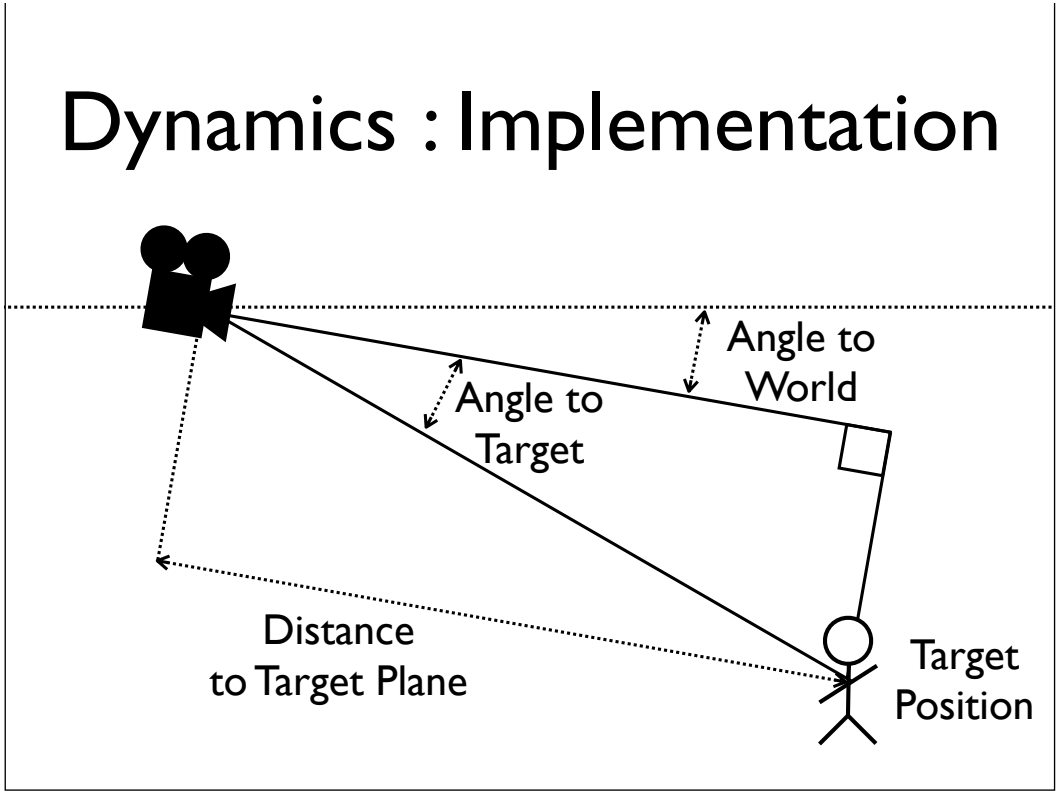
With negative values being behind the player, and in those cases, we automatically turn the camera around, to look back at the player



We allow the designer to set a range of valid distances for the camera.

Ensuring that it **never gets too far from, or too close** to, the player.

(Similarly to the safe zone, we can collapse these min/max constraints to represent a fixed distance.)



SUMMARY: So that's how we let the designer control the three defining properties of the camera
* the position of the player on screen
* the angle we're looking at him from, or rather, the orientation of the camera
* and his size, or rather, the distance from the camera to the plane of the target, perpendicular to the look vector

internally we calculate, constrain, and store these as
* the angle from the look vector of the camera, to the target. This is a 2d diagram, but in 3d this is a pair of angles, from the horizontal, and the vertical, in camera space. We use the angles, so that we can easily represent the cases where the target is behind the camera. It also makes it easier to apply the safe zone constraints, as these are internally represented as a pair of angular ranges in the same space.
* the angle of the camera to the world, again, 2d diagram, consider that dotted line to be zero degrees. In 3d we use an euler, because they're easy to manipulate and visualise.
* and the distance to the target plane

* we package these up with the position of the target, and we have enough information to calculate the actual position and orientation of the camera in the world

Dynamics : Implementation

- Calculate Angle from Camera to Target
- Constrain Angle from Camera to Target
- Calculate Angle from Camera to World
- Constrain Angle from Camera to World
- Calculate Distance from Camera to Target Plane
- Constrain Distance from Camera to Target Plane

Overview

- Zoning
- Dynamics
- Blending
- Rails

Overview

- Zoning
- Dynamics
- Blending
- Rails

Smoothing transitions between cameras (i.e., not jump cuts).

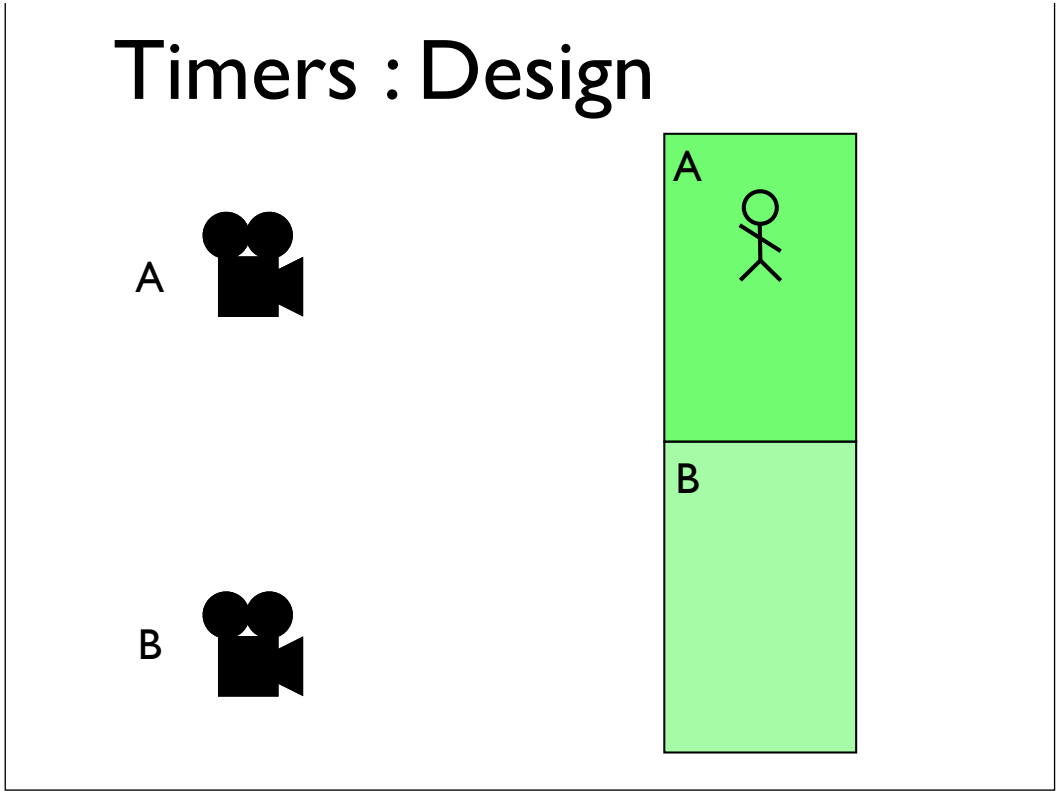
Blending : Overview

- Timers
- Ease
- Blend Space

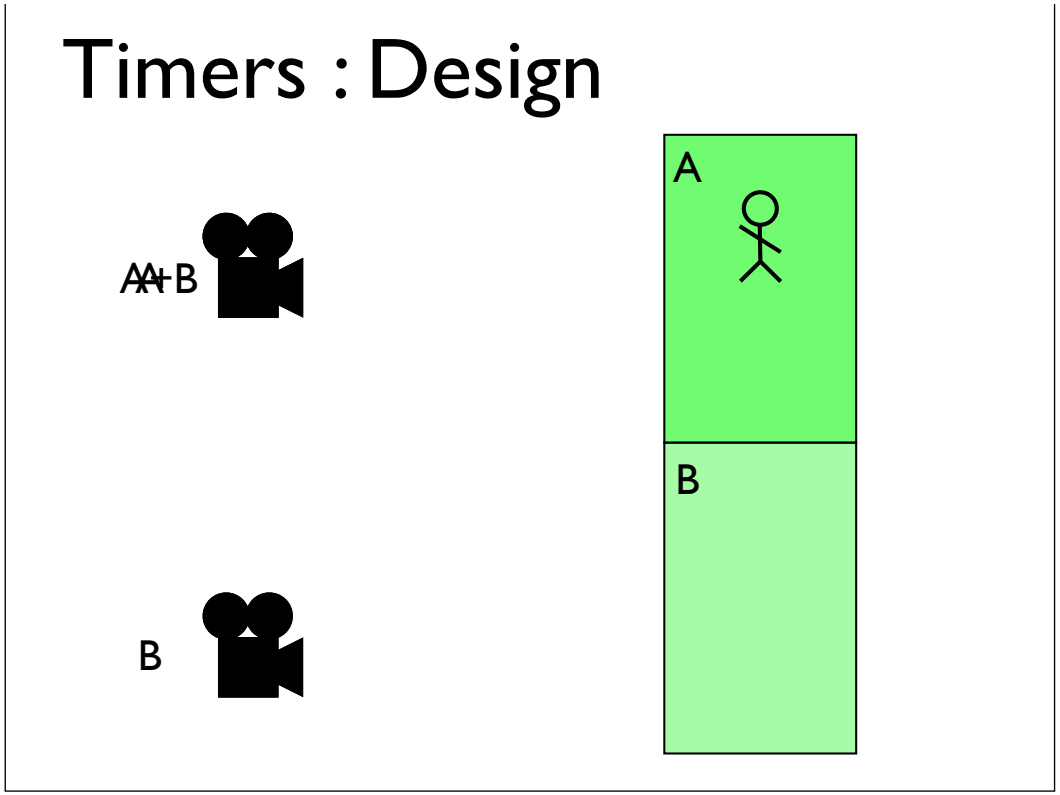
Timers, which track and update each blend

Ease, which controls the smoothness of a blend

and Blend Space, where I'll define what a blend between two cameras actually does.



when we start a new camera, we don’t cut to it, but blend into it over a fixed period of time.



And when I say blend, I don't mean combining the pixels from both camera!

I mean **creating a third camera from varying proportions** of two other cameras.

So when we start the second camera, what actually happens is that a phantom third camera moves from the first camera to the second. It's position and orientation determined by a blend of the two cameras, driven by a timer.

When we move into a zone that references a new camera, as well as starting that new camera, we also start a timer for it.

Timers : Implementation

- Timer List
  - Entry is a camera fading in
  - Camera can have multiple timers in list
  - FIFO
  - New timers inserted at the top
  - When a timer completes, all timers below it are removed

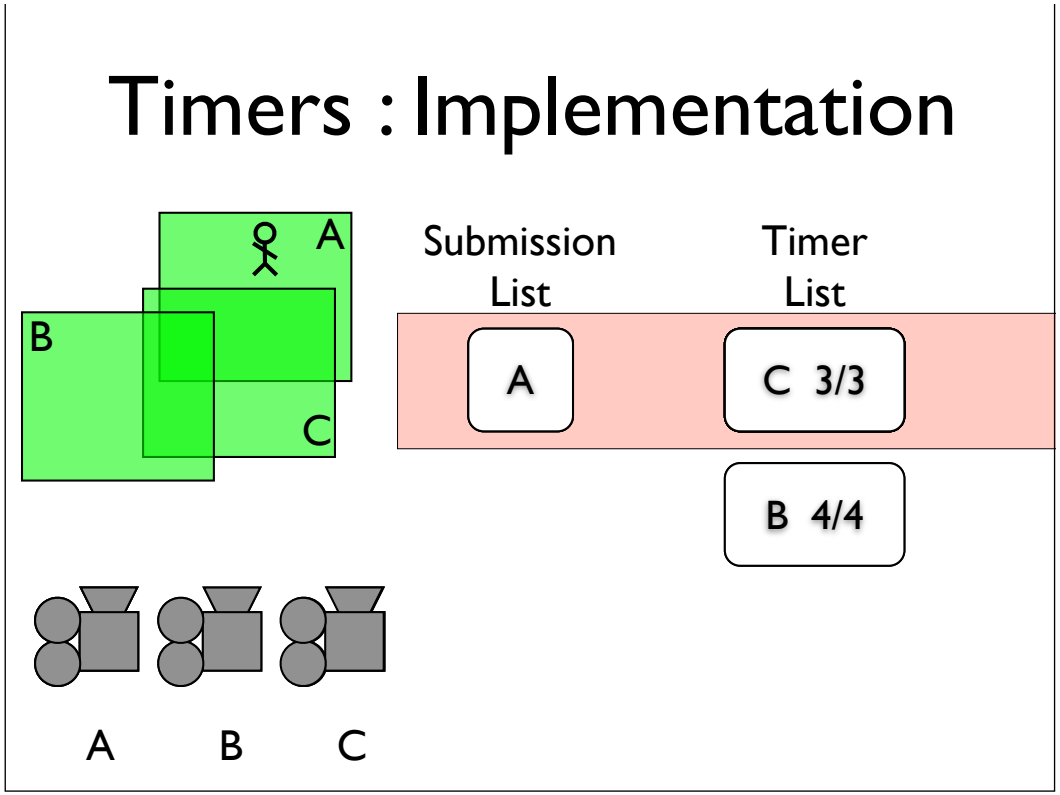
Now if the player is moving between zones, faster than their cameras fade in, then it's entirely possible that we'll be running multiple timers, simultaneously, so we need to store these timers in a list.

In fact, you may be fading back into a camera you're already fading out of. In these cases you may be tempted to try and reverse a running timer. I initially tried this, but couldn't get it to work smoothly, as you need to correct all the timers between the two instances.

Instead I decided to let each timer play out, and to track new timers separately. New cameras fade in, and by fading in, reduce the contribution of the cameras below them in the list.

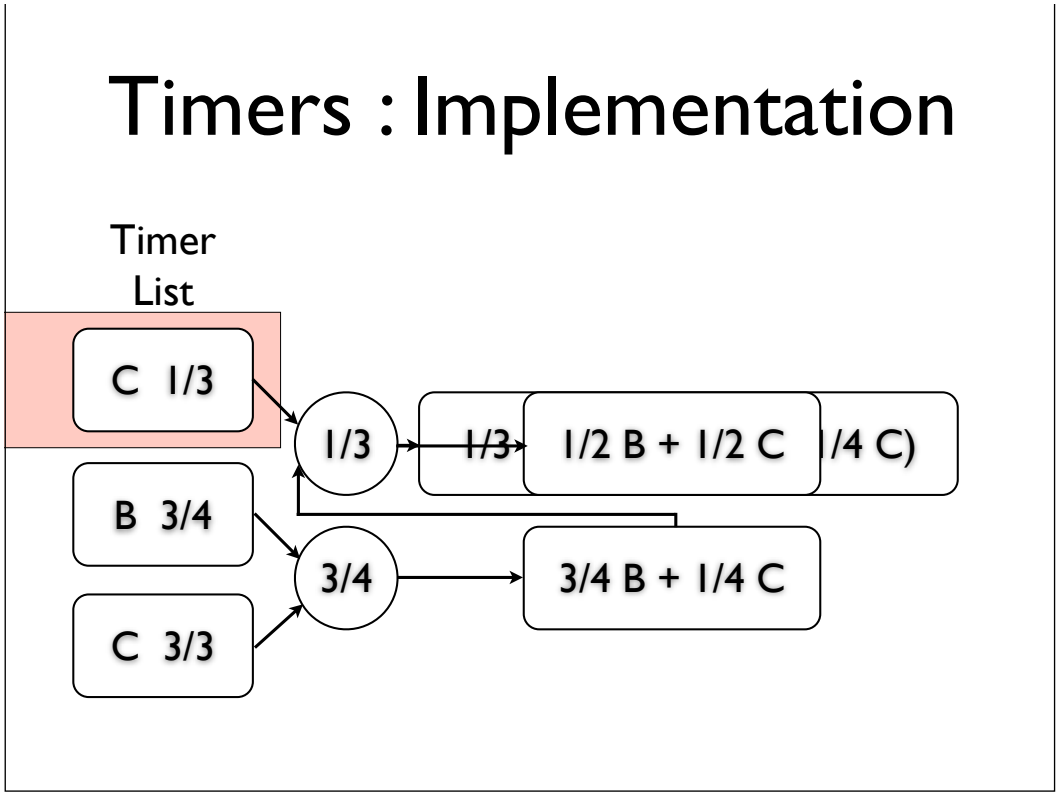
So cameras don't fade out by themselves, they fade out as a result of a new camera fading in,

The timer list is a FIFO. New timers get inserted at the top, and timers below completed timers fall out of the bottom.



Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

zone a -> top of submission list -> starts new timer, timer list is empty, so it starts immediately
zone c -> top of submission list -> starts new timer, camera is now a blend of a and c, hence grey
zone b -> increment old timers, start new timer, camera is now a blend of a,b and c
out of zone a -> camera c's timer has completed, drop camera a
out of zone b -> c at top of submission list again -> start new timer, note that c is in the timer list twice
wait a second -> b's timer completes, drop the entries below it
wait another second -> c's timer completes, drop the entries below it, and we're back with c at 100%

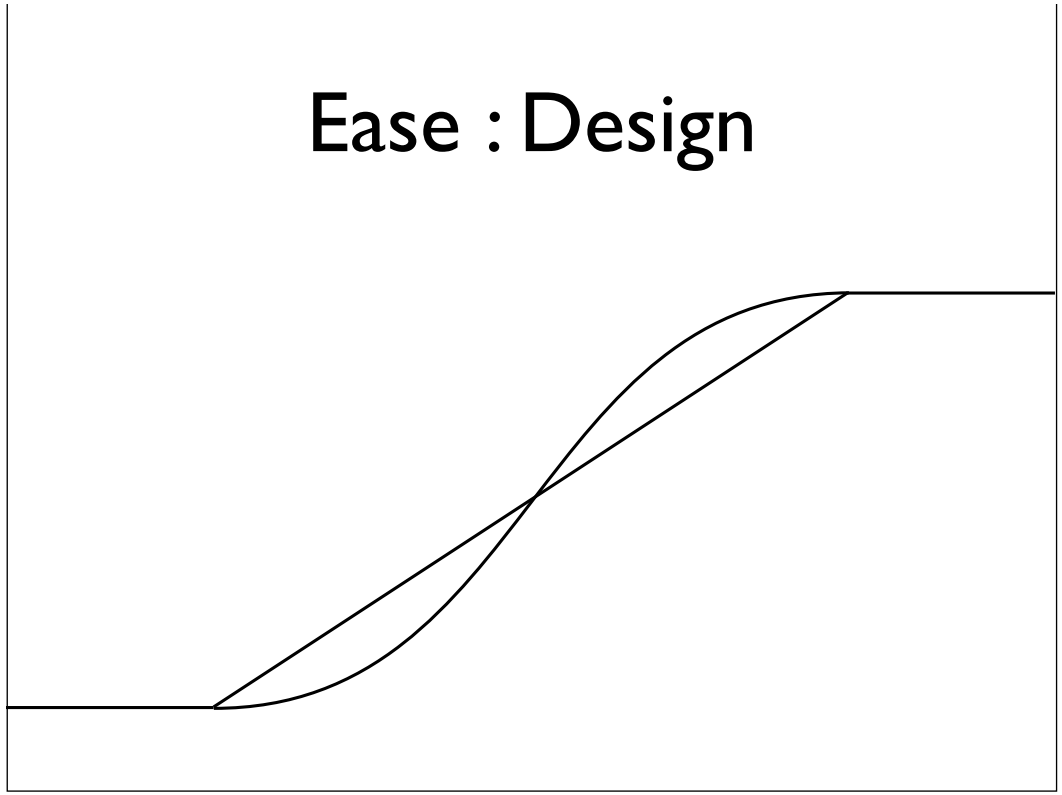


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. I this case we have two entries for camera c, because we moved out of it's zone, and back into it before the camera in-between, camera b, finished fading in.

We start with the oldest camera, and blend the next one in, using camera b's timer to define the proportion of camera b to use. So in this case, it's 3 seconds in, out of a total of 4, so that's three quarters of camera b, leaving one quarter for camera c.

Next we take the result, and blend that with the next newest camera, the top camera c,
using, the new cameras timer. It's 1 second in out of 3, so 1/3 of camera c, and 2/3 of the previous blend
which ultimately works out to be 1/2 camera b and 1/2 camera c



The trouble with using the timers raw, is that you get these **simple linear blends**. You can see the **sharp corners** here, and when you use them to blend cameras, you can see the jerk as it starts to move, and again when it stops. While sometimes this is desirable, mostly it's just ugly.

What we want, is to add what animators call "ease".

To do this, we feed the linear blend, into a spline.

Ease : Implementation

- Hermite Spline
- Fixed endpoints at 0 & 1
- Controllable tangents
- ease = 1 - tangent
- Ease in and out

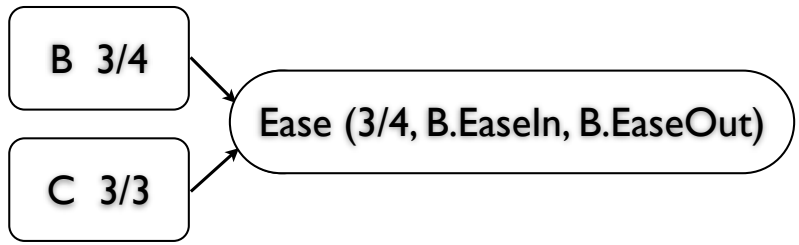
Specifically a Hermite spline, which lets you control the position and tangent of the endpoints of the curve.

We fix the endpoints at 0 and 1, and map the tangents into a range that makes sense as an ‘ease’ control.

With 1 representing full ease, 0 no ease, or linear, and -1 giving us negative ease, for those special times when you need a really harsh blend.

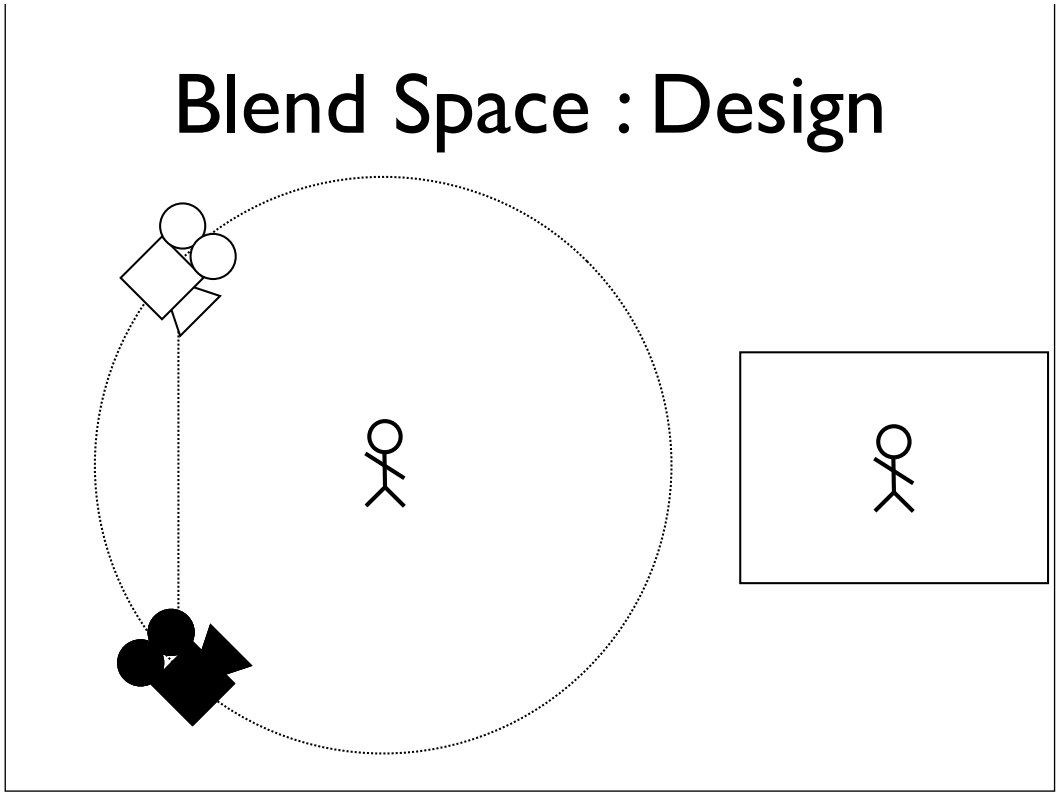
We allow the designer to control ease in and out separately.

Ease : Implementation



We apply ease when we calculate the blend factor between two cameras.

We take the timer, and feed it into the ease function, which just evaluates the hermite, taking ease in and out values from the new camera.



There's a hidden problem in what we have described so far for blending.

???

If we blend **positions** (not talking about proportions here) along a straight line, we get "blooming" effect.

Solution: Blending positions along a circular arc of fixed distance from player.

Overview

- Zoning
- Dynamics
- Blending
- Rails

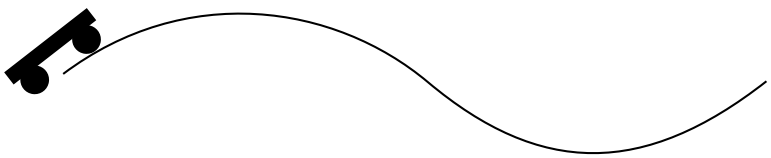
Overview

- Zoning
- Dynamics
- Blending
- Rails

Rails : Objectives

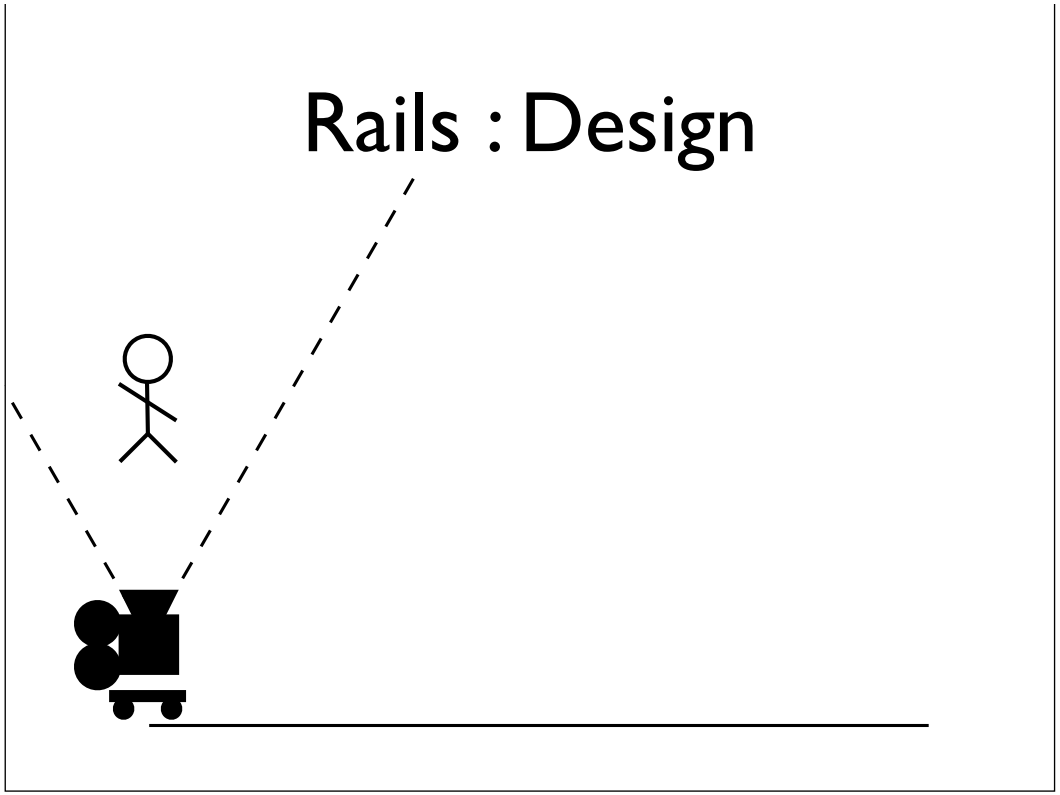
A generalization of this “camera track” idea is to borrow an idea from the film industry. One of their solutions to this problem, is to construct rails, and put the camera on a little cart, known as a Dolly, that rides on the rails.

Rails : Design



Now for us, the rail is spline, specifically a NURB (constructed in Maya).

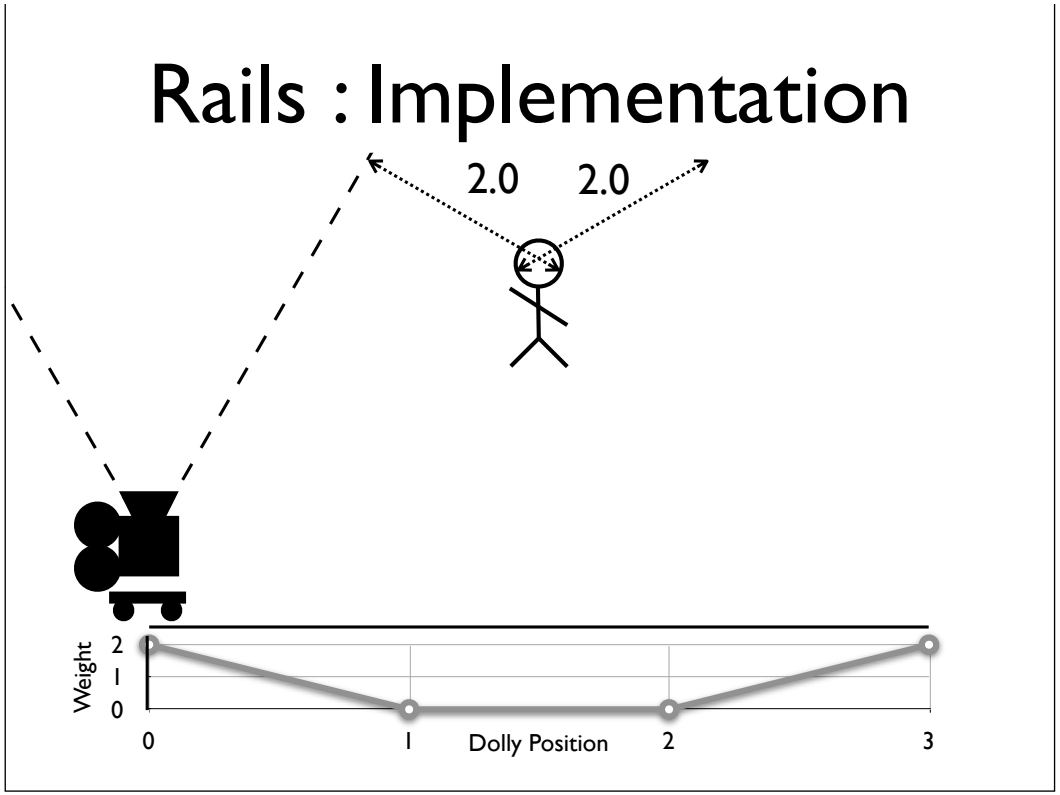
And the Dolly is a point on that spline, represented by the **parametric value** of the spline at that point.



So we have rail, we have a dolly on that rail, and we have the camera sitting on the dolly.

What we want to do, is only move the dolly by enough to keep the player within the constraints defined by the camera.

The player is free to move within the constraints, but when he tries to move outside them, the dolly moves to compensate as best it can.



So how do we do that.

Well remember that the Dolly is actually just a point on the spline represented by a single parameter.

We use the constraints to calculate a weight at a given point on the spline.

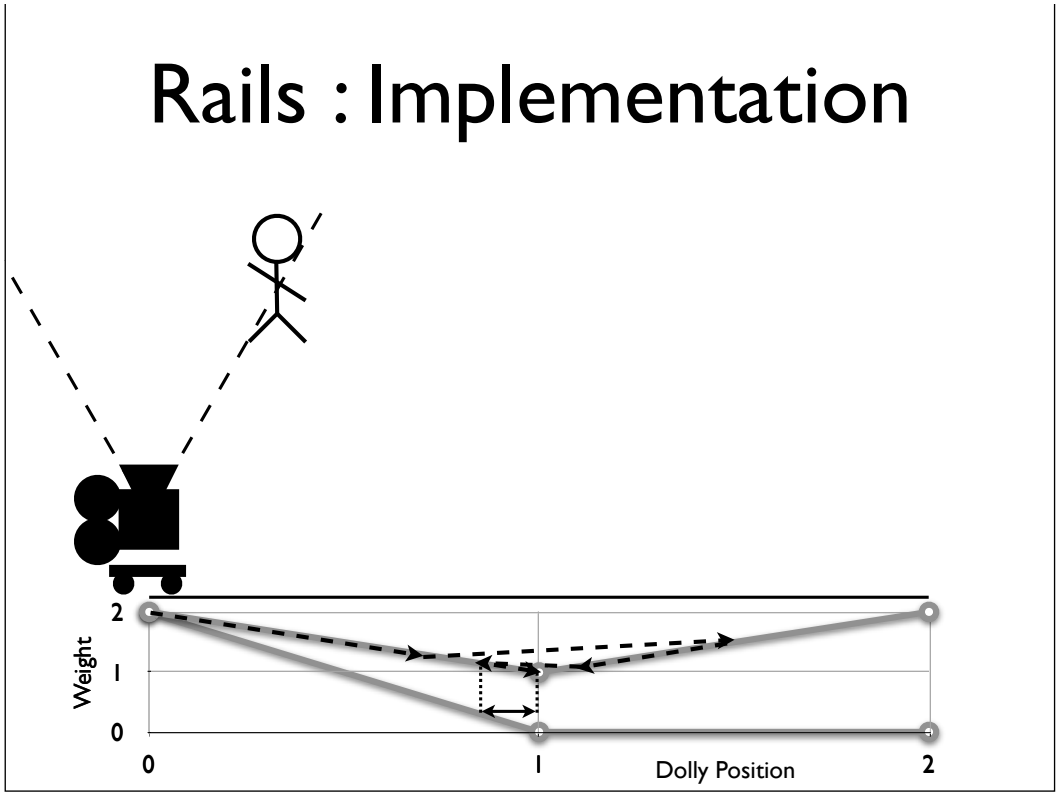
Here we see that the player is 2 units outside of the constraint, so the value of the weighting function at this point, zero, is 2.

If we move the dolly to position 1, the player is just inside the constraints, and so the weight is zero.

Between positions 1 and 2, the value remains zero, as the player remains inside both constraints.

And as we move to position 3, the player is now 2 units outside the other constraint, and the weight is 2 again.

So, armed with this weighting function, what we're actually trying to do, is find the **nearest minima** on it, to the previous position of the dolly.



In this example, the player moves outside of the cameras constraints. Which gives us these weights.
In order to help us find the nearest minima, we add the distance from the Dolly's initial position to the weighting function. ...and now the weights look like this.

Now there are a number of ways to locate the minima, but this is how we do it (classic **hill-climbing)**

We take a guess as to which direction the player has moved, and take an experimental step in that direction.
If the weight at the new position is lower, then we try another step.
If the weight is higher, then we turn around slow down, and go back.
If as a result of slowing down, our next move is below a certain threshold, then we stop.

The smaller the threshold, the smaller the potential error, the smoother the camera, but also, the more times you'll evaluate the weight.

Note: This method does NOT require calculating the derivative of the weighting function.

Rails : Implementation

- Additional Weights
  - Distance from Player to Dolly
  - Angle from Tangent of Rail at Dolly
  - Amount Boss obscures Player
  - Number of minor characters out of frame

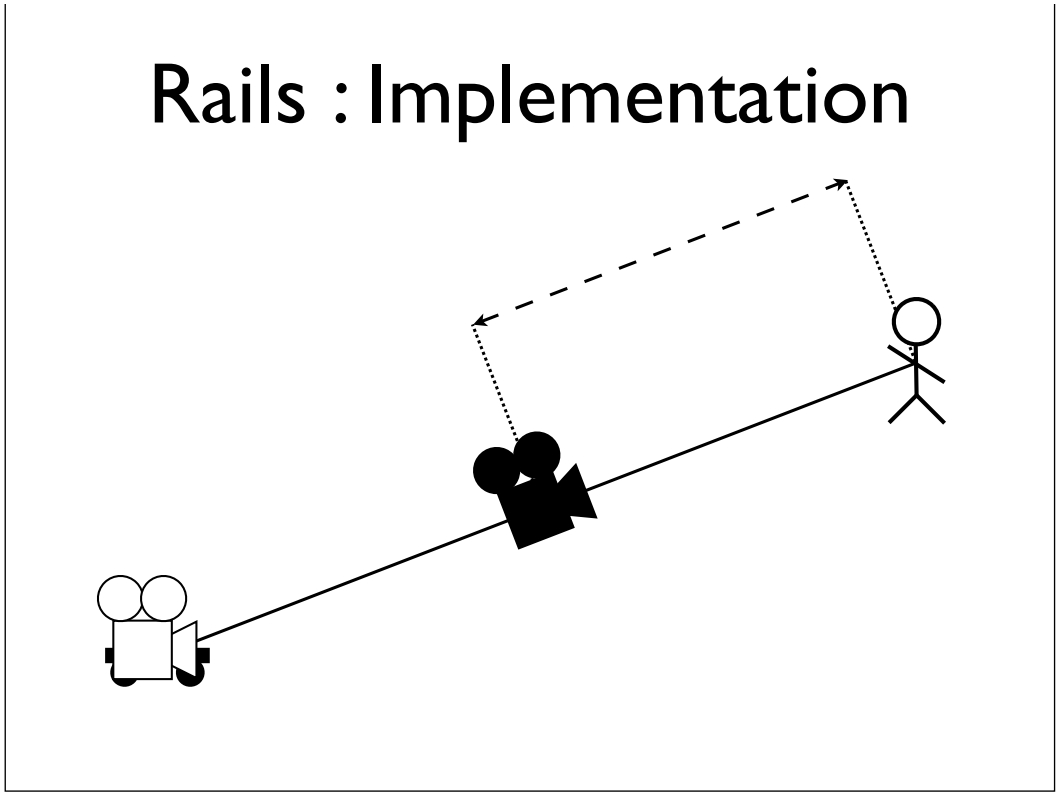
Which means it's relatively easy to experiment with different weighting functions.
For example, the distance from the dolly to the player, is fairly simple, and gives you the classic, drag or push the camera down the corridor, shot..

But the angle from the tangent of the rail at the dolly, which is the weight we would use for the tracking shot in the example, is a bit more complex.

Much more interesting cinematic weights:

For certain special cases, like Bosses, we can use weights to deal with the boss obscuring the player.

Probably the most complex weight we use, is related to the **number of minor characters that are out of frame**, and the amount they're out by.



So, having calculated the position of our dolly, we **combine** it with the camera dynamics we described earlier. Only instead of calculating distance and angle from the position and orientation of the camera node [in Maya], we derive them from the position and orientation of the dolly (where we consider the tangent of the rail at the dolly, to be the direction the dolly is pointing).

In fact you can consider the camera node, to be a dolly on a zero length rail.

Overview

- Zoning
- Dynamics
- Blending
- Rails

SUMMARY: In professional quality (AAA) camera control, there is a lot more involved than simple following cameras!

Other Stuff

- Dealing with multiple targets
- Target definition, and calculation
- Dealing with static and animated cameras
- Overriding cameras at arbitrary points to focus on dynamic areas of interest
- Framing fights, using multiple targets
- Damping
- Fragility of rotational blends
- Physical post effects like shake and sway

Not going to talk about.

End by looking again at God of War clip and trying to see where various techniques we talked about are used.