

Designing and Implementing a Dynamic Camera System

Phil Wilkins

Objectives

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

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
- Fields

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
and in Fields I'll present a more advanced way of controlling
Blending

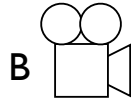
Overview

- Zoning
- Dynamics
- Blending
- Rails
- Fields

Zoning : Objectives

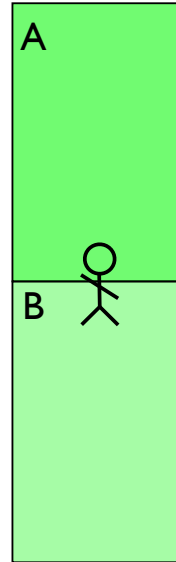
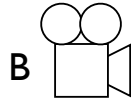
- Stationary Cameras
- Chosen by Player Position
- Alone in the Dark

Zoning : Design



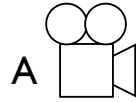
The game has a spatial database of zones. Each referencing one or more cameras, although for the moment we'll just deal with the case where a zone references a single camera.

Zoning : Design



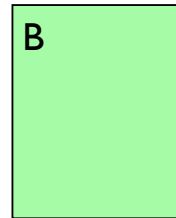
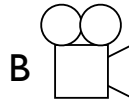
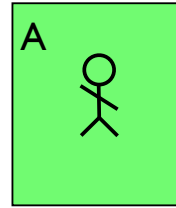
The game has a spatial database of zones. Each referencing one or more cameras, although for the moment we'll just deal with the case where a zone references a single camera.

Zoning : Design



The game has a spatial database of zones. Each referencing one or more cameras, although for the moment we'll just deal with the case where a zone references a single camera.

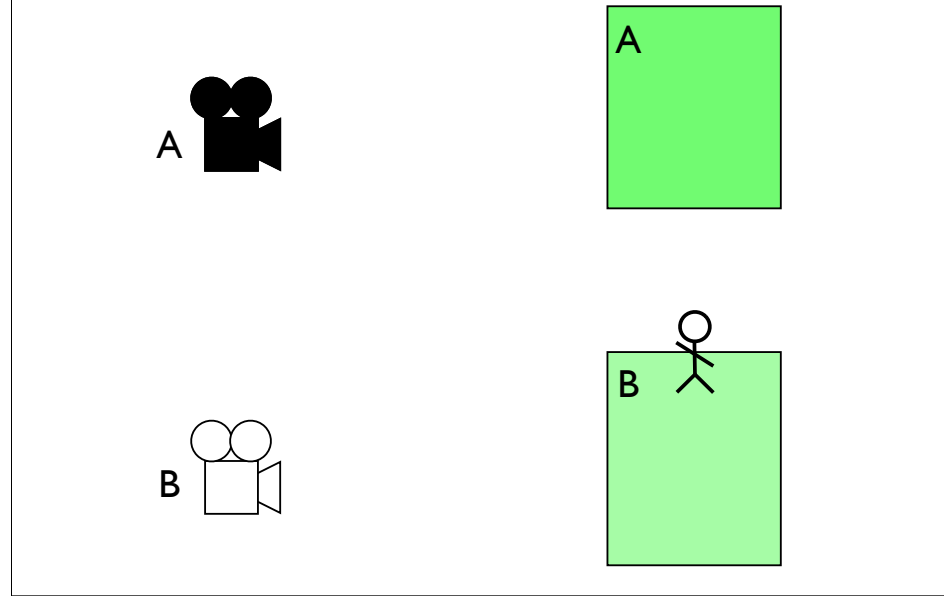
Zoning : Design



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. To add hysteresis to our system, by playing with the boundaries of the zones.

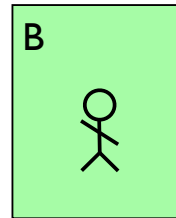
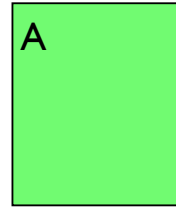
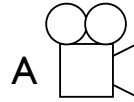
Zoning : Design



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. To add hysteresis to our system, by playing with the boundaries of the zones.

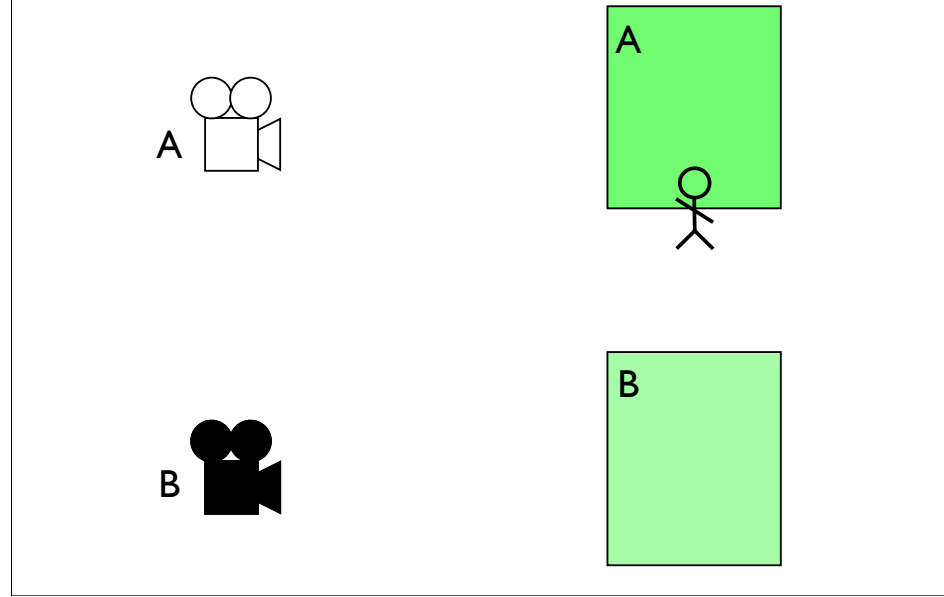
Zoning : Design



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. To add hysteresis to our system, by playing with the boundaries of the zones.

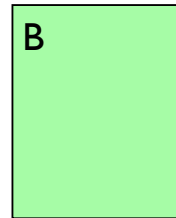
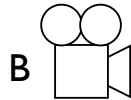
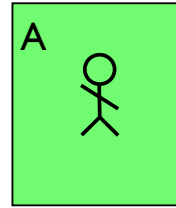
Zoning : Design



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. To add hysteresis to our system, by playing with the boundaries of the zones.

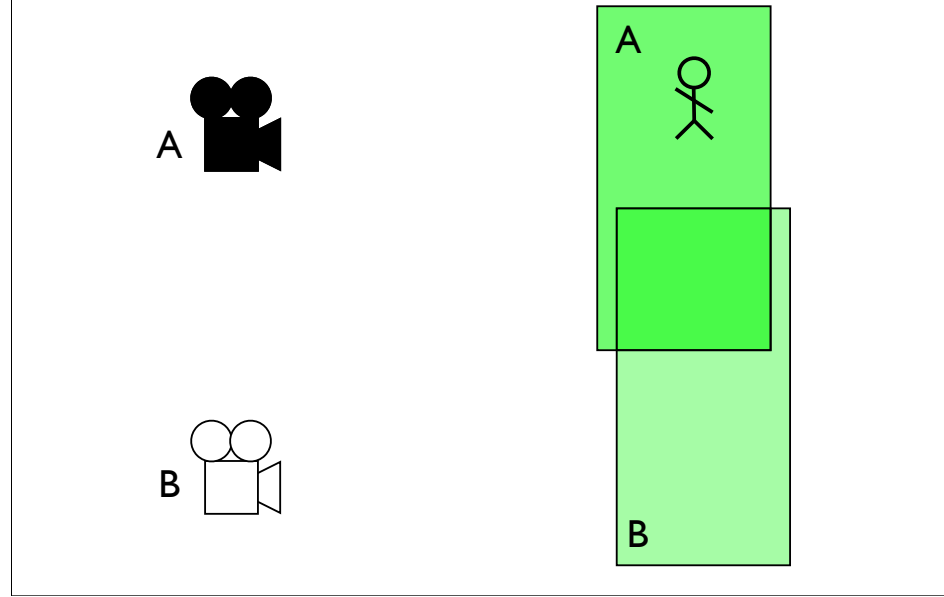
Zoning : Design



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

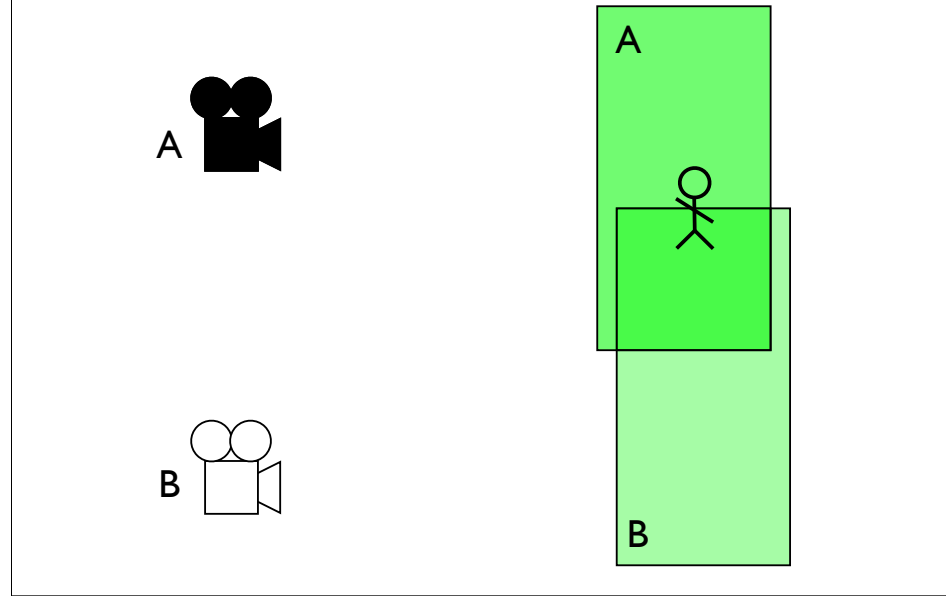
This allows us to loosen up the boundaries between cameras. To add hysteresis to our system, by playing with the boundaries of the zones.

Zoning : Design



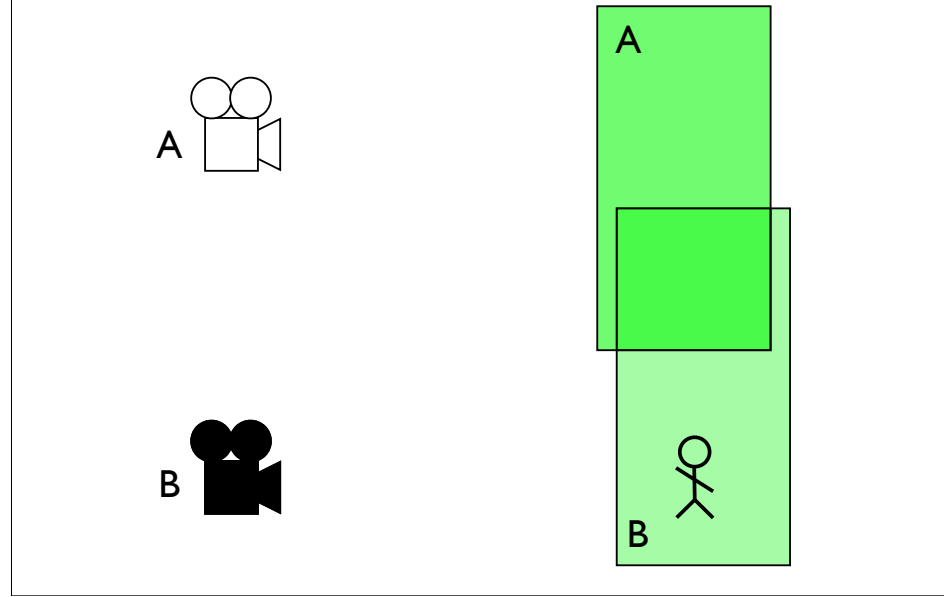
Alternatively we can overlap the zones. In which case, when we enter the overlap space, we always change to the new camera.

Zoning : Design



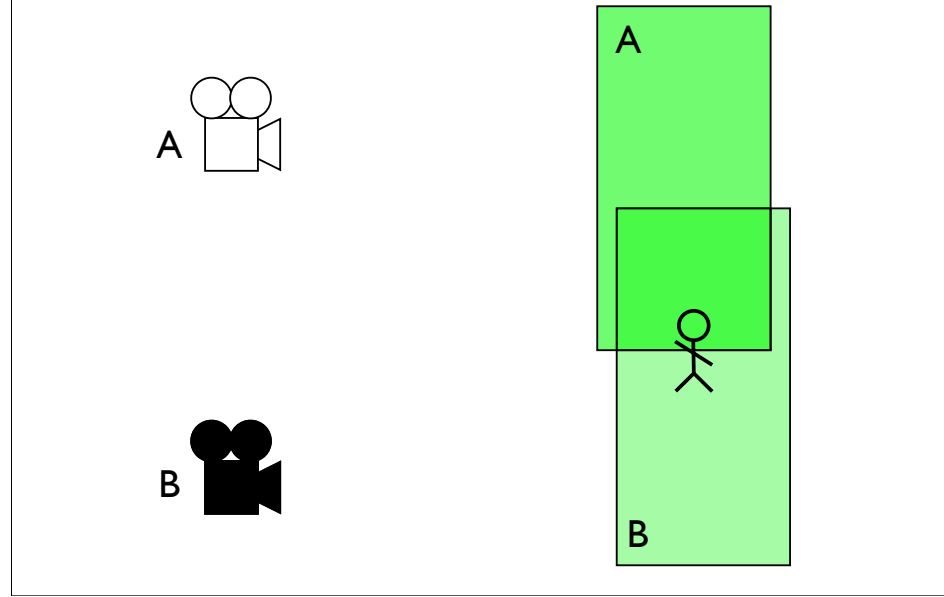
Alternatively we can overlap the zones. In which case, when we enter the overlap space, we always change to the new camera.

Zoning : Design



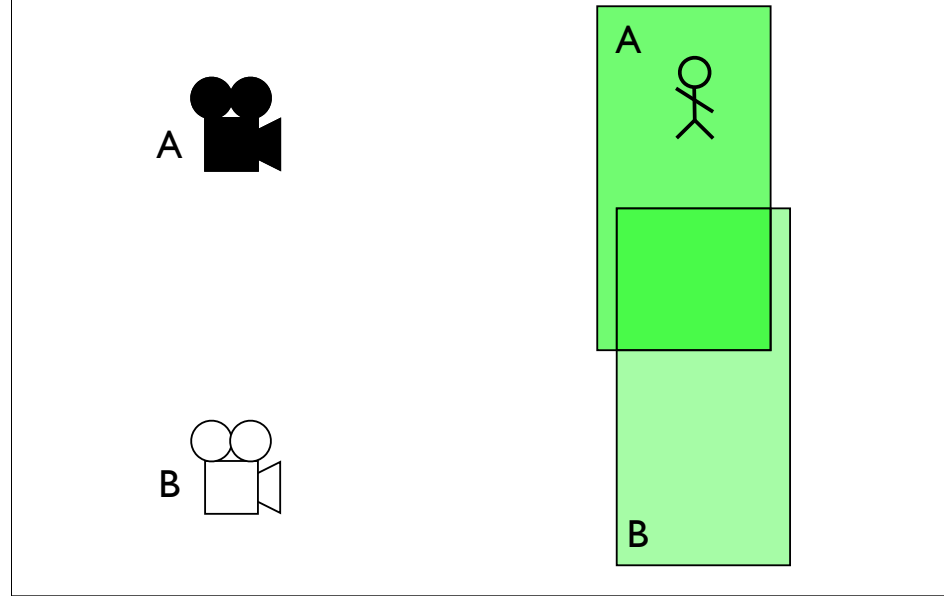
Alternatively we can overlap the zones. In which case, when we enter the overlap space, we always change to the new camera.

Zoning : Design



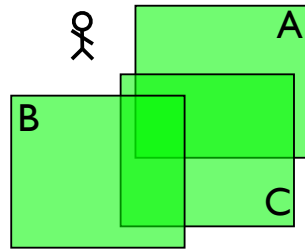
Alternatively we can overlap the zones. In which case, when we enter the overlap space, we always change to the new camera.

Zoning : Design



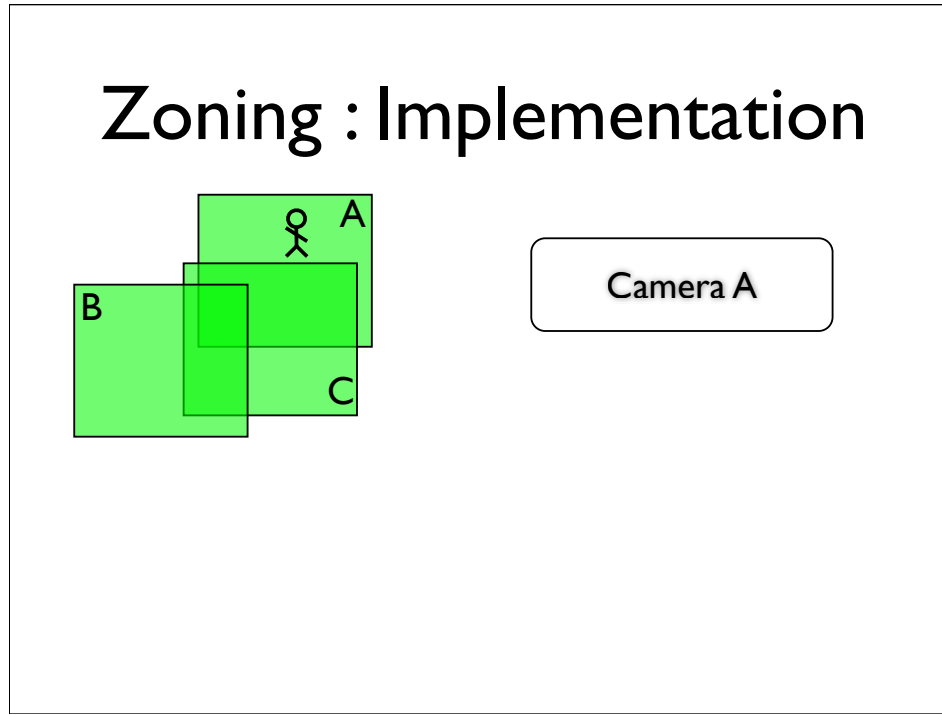
Alternatively we can overlap the zones. In which case, when we enter the overlap space, we always change to the new camera.

Zoning : Implementation



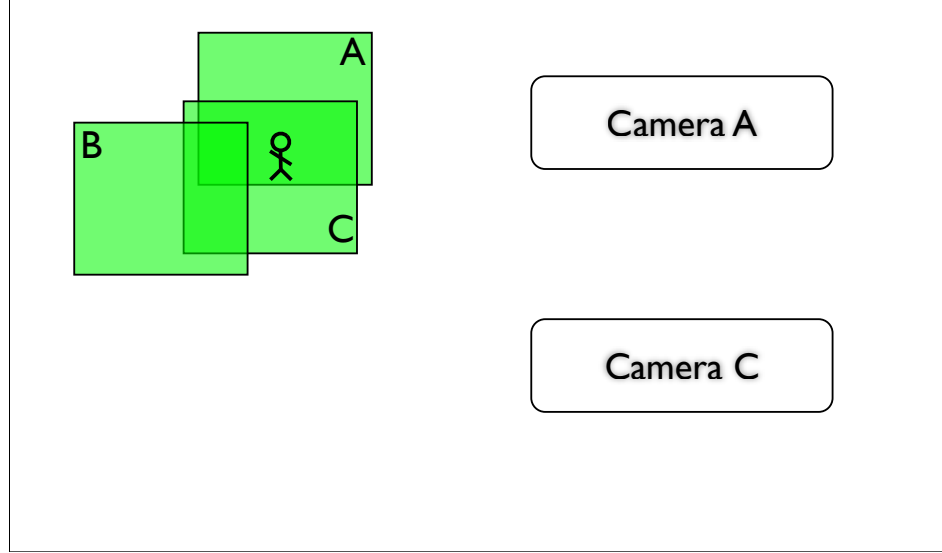
So at runtime, each frame, we query the spatial database of zones, and get back a collection of camera references. Since we can't, or don't want to make too many assumptions about the zone database, we treat the results as essentially unordered.

Zoning : Implementation



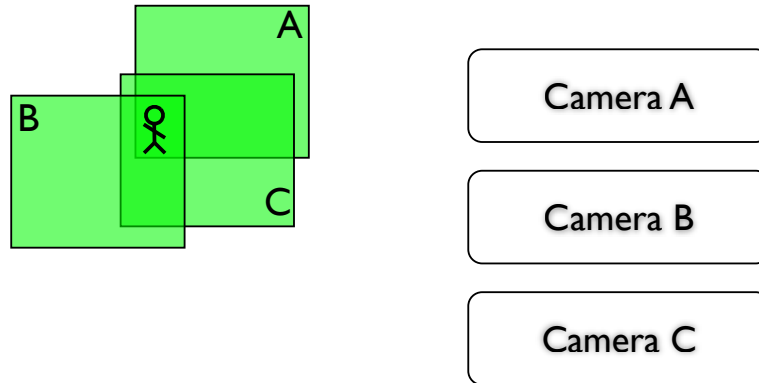
So at runtime, each frame, we query the spatial database of zones, and get back a collection of camera references. Since we can't, or don't want to make too many assumptions about the zone database, we treat the results as essentially unordered.

Zoning : Implementation



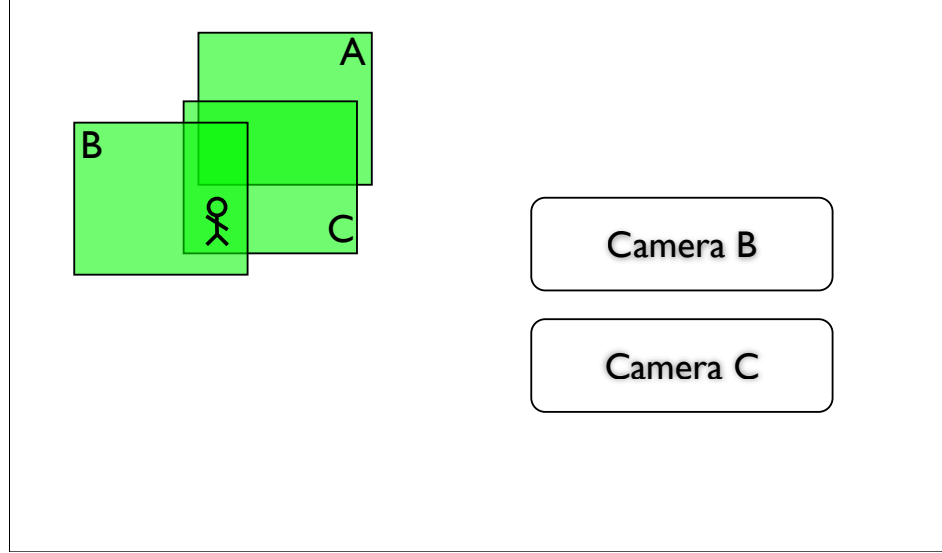
So at runtime, each frame, we query the spatial database of zones, and get back a collection of camera references. Since we can't, or don't want to make too many assumptions about the zone database, we treat the results as essentially unordered.

Zoning : Implementation



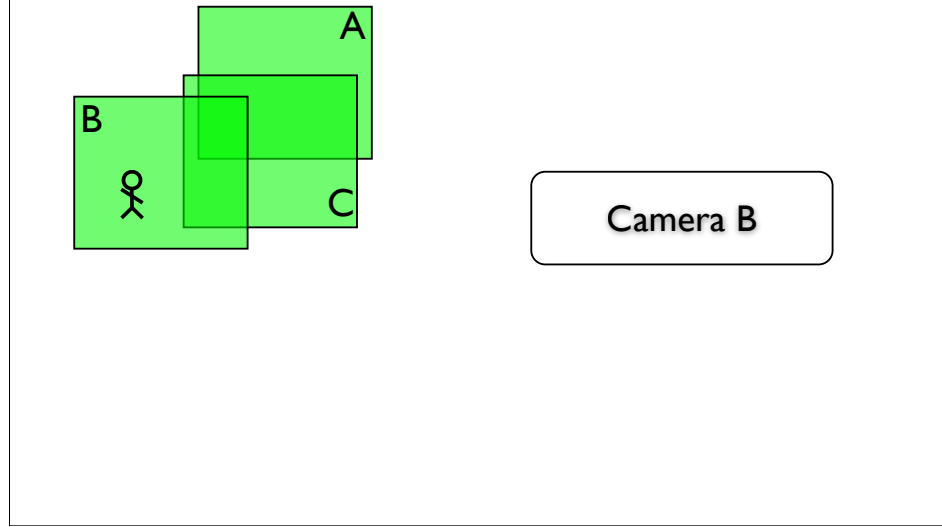
So at runtime, each frame, we query the spatial database of zones, and get back a collection of camera references. Since we can't, or don't want to make too many assumptions about the zone database, we treat the results as essentially unordered.

Zoning : Implementation



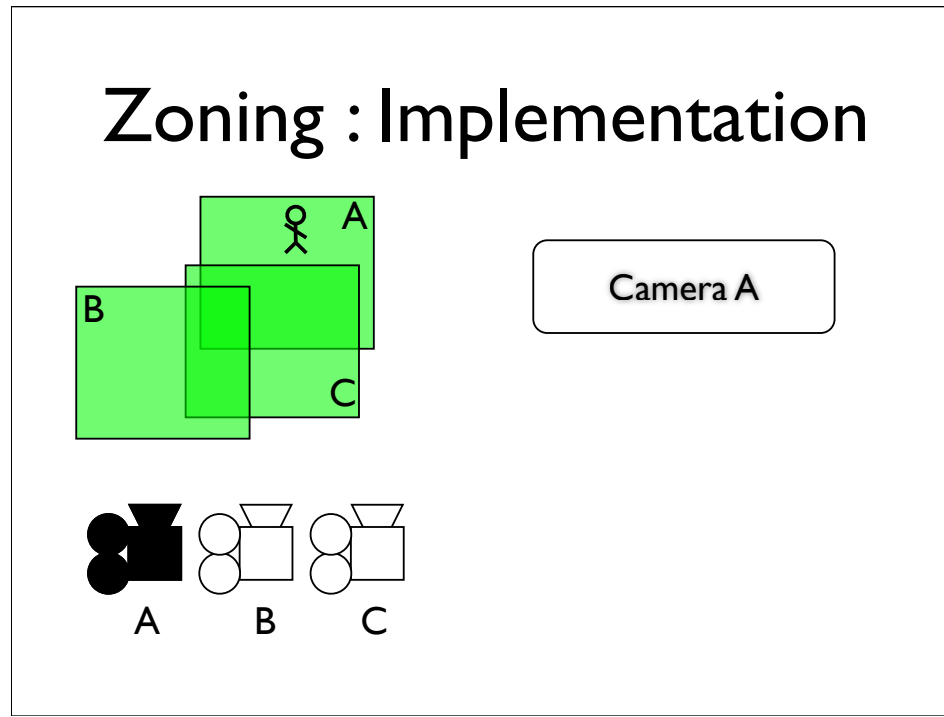
So at runtime, each frame, we query the spatial database of zones, and get back a collection of camera references. Since we can't, or don't want to make too many assumptions about the zone database, we treat the results as essentially unordered.

Zoning : Implementation



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.

Zoning : Implementation

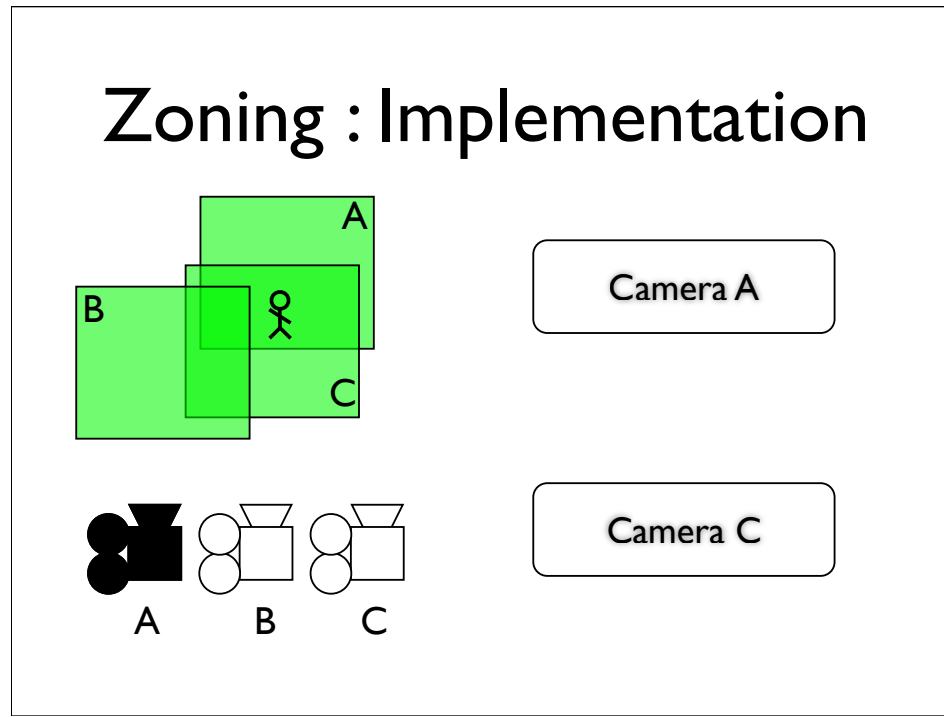


A naive implementation would just compare the results from the query, against the currently active camera, and if there's a difference, swap to the new camera. This works fine, unless your zones overlap.

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

Zoning : Implementation

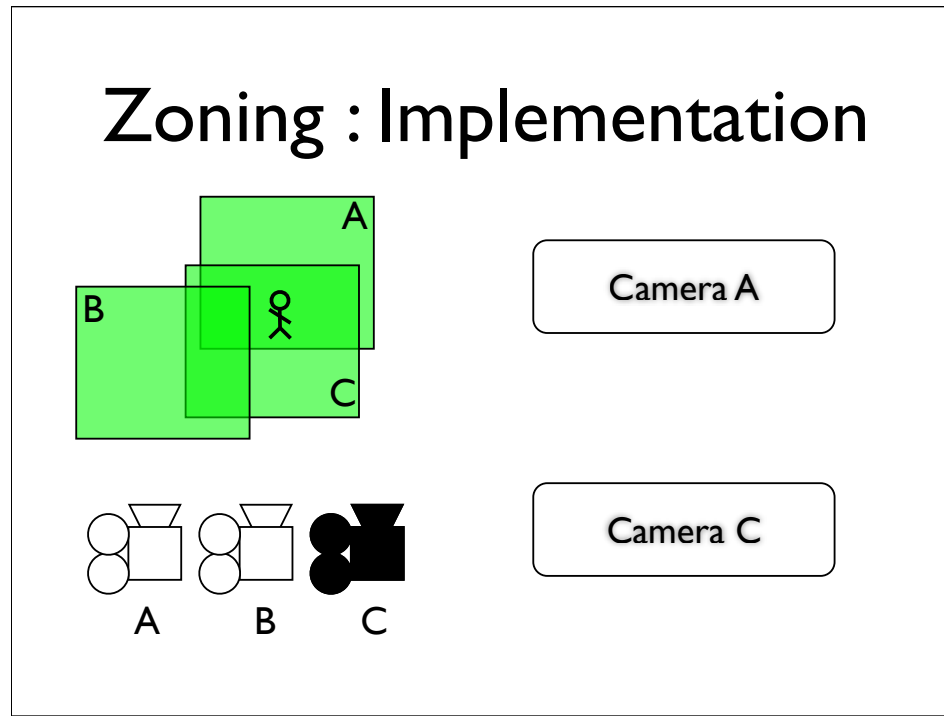


A naive implementation would just compare the results from the query, against the currently active camera, and if there's a difference, swap to the new camera. This works fine, unless your zones overlap.

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

Zoning : Implementation

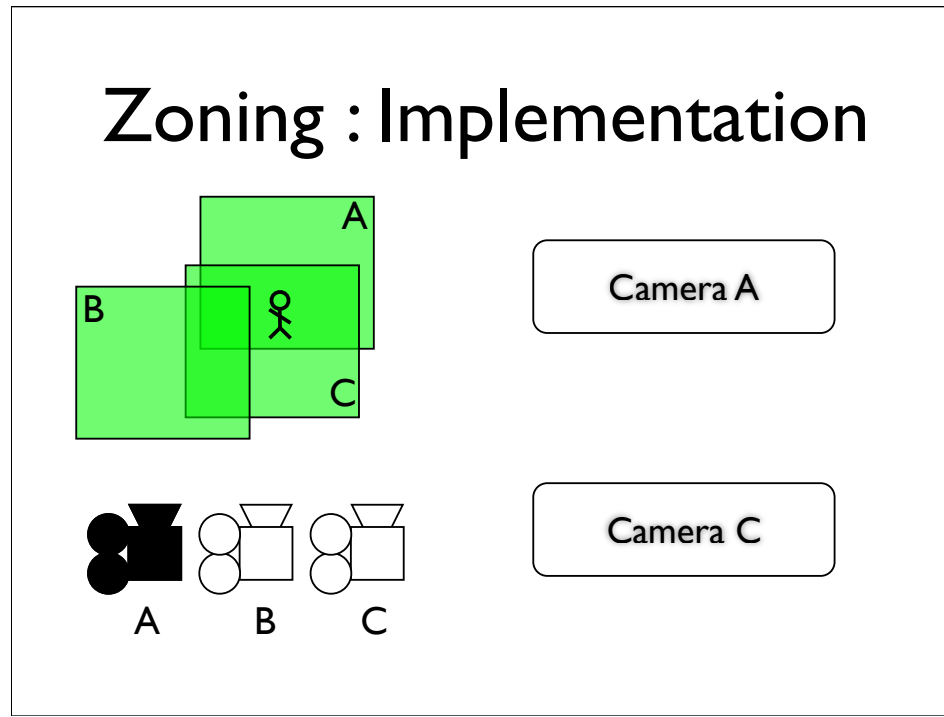


A naive implementation would just compare the results from the query, against the currently active camera, and if there's a difference, swap to the new camera. This works fine, unless your zones overlap.

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

Zoning : Implementation



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.

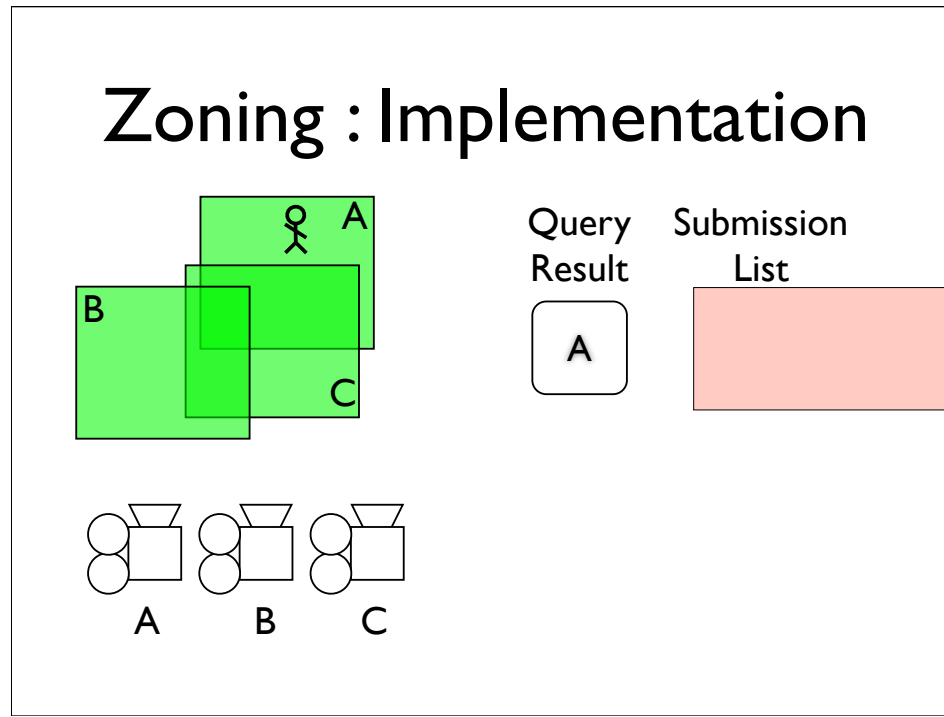
In this example, when we move from zone A into the overlap between zones A and C, we start off right, by switching to camera C,

but next frame, we're still in the overlap, still getting A and C

Zoning : Implementation

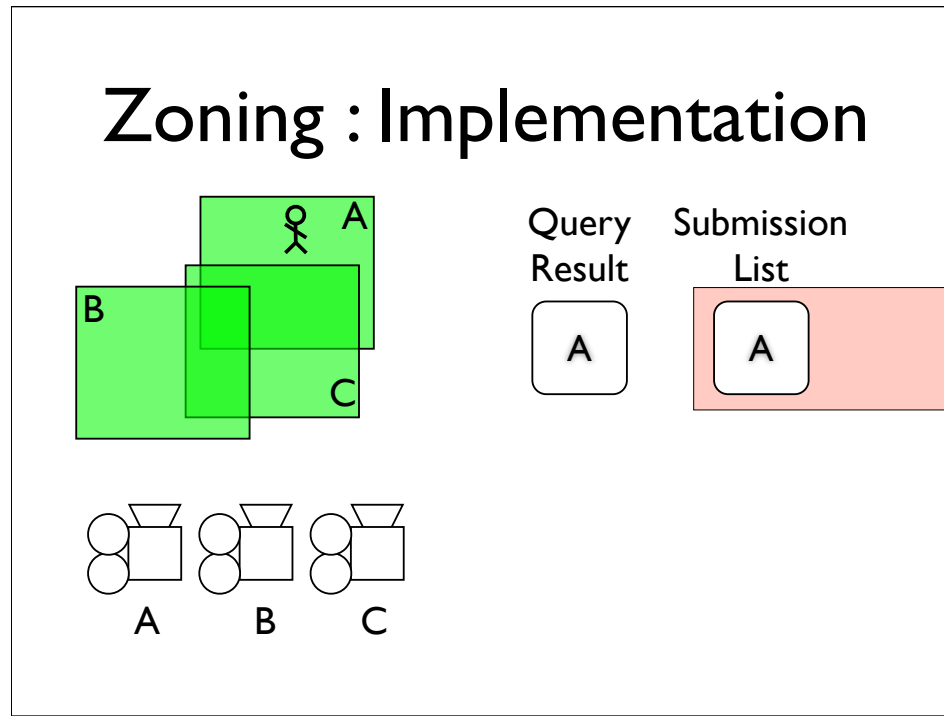
- Submission List
 - List of all cameras that were submitted last frame.
 - Used to distinguish newly submitted cameras from old ones
 - New cameras inserted at top
 - Effectively sorted by age

Zoning : Implementation



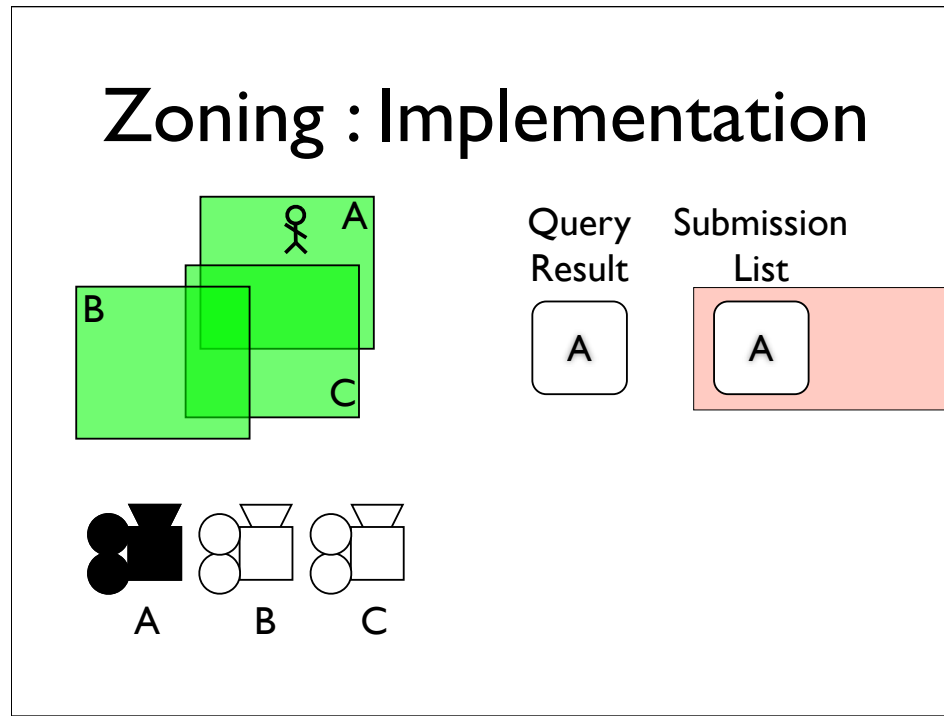
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



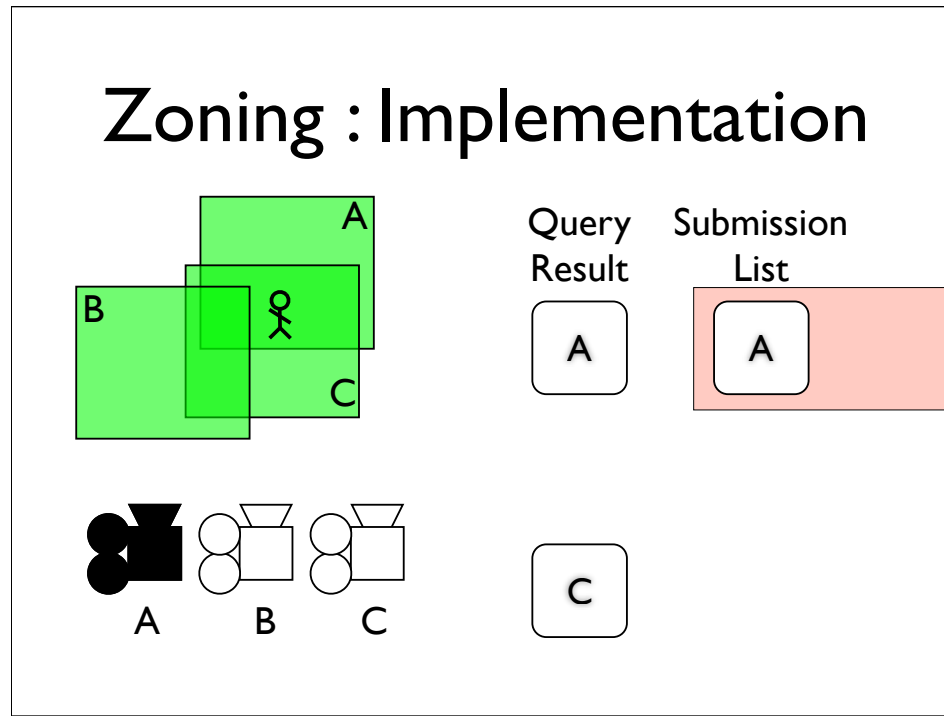
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



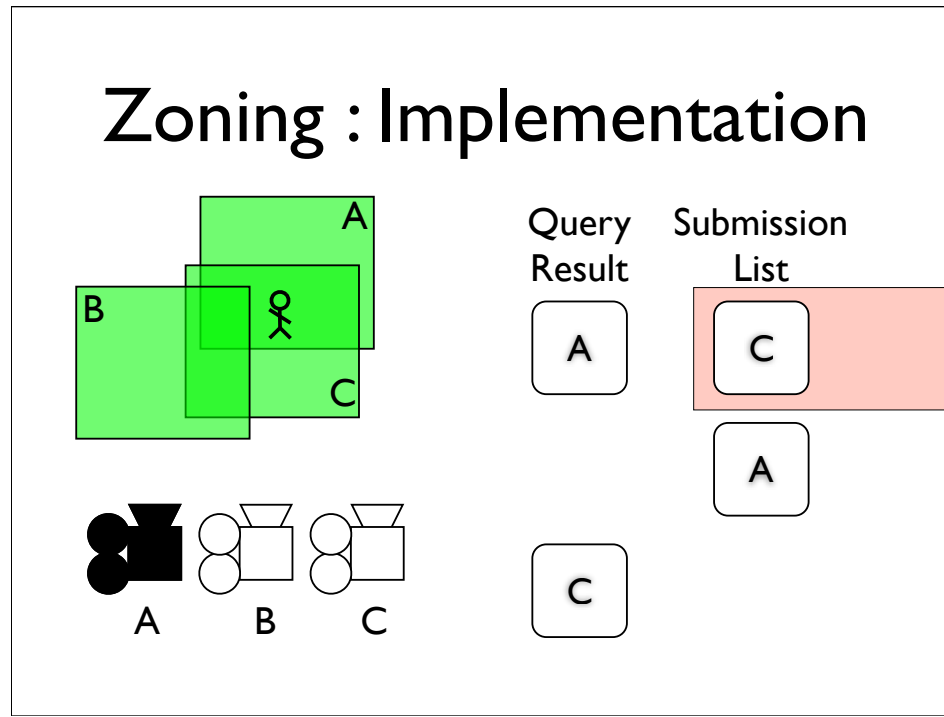
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



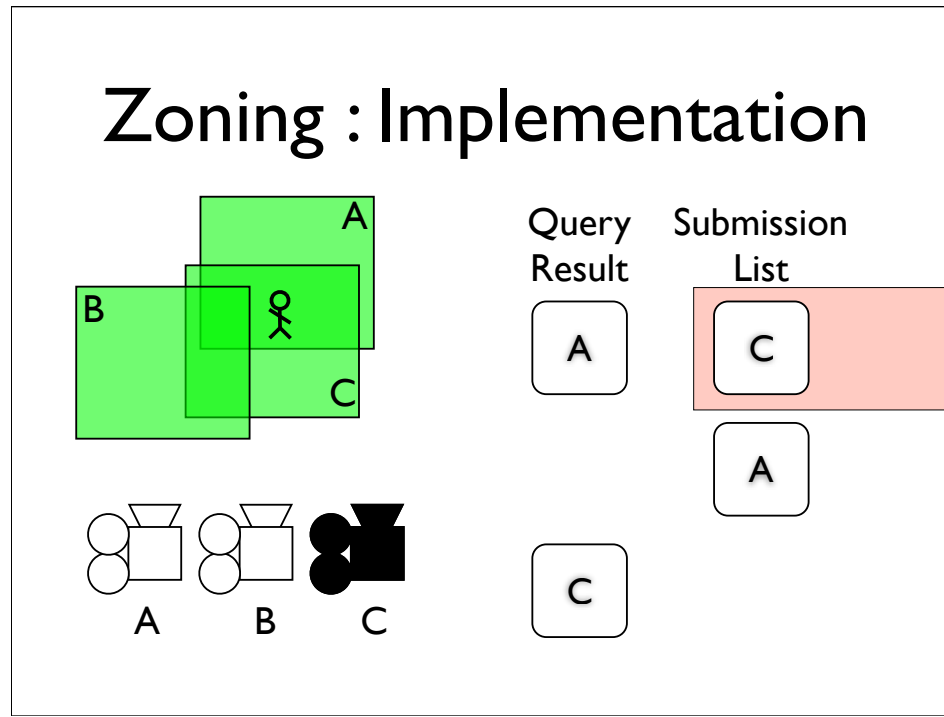
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



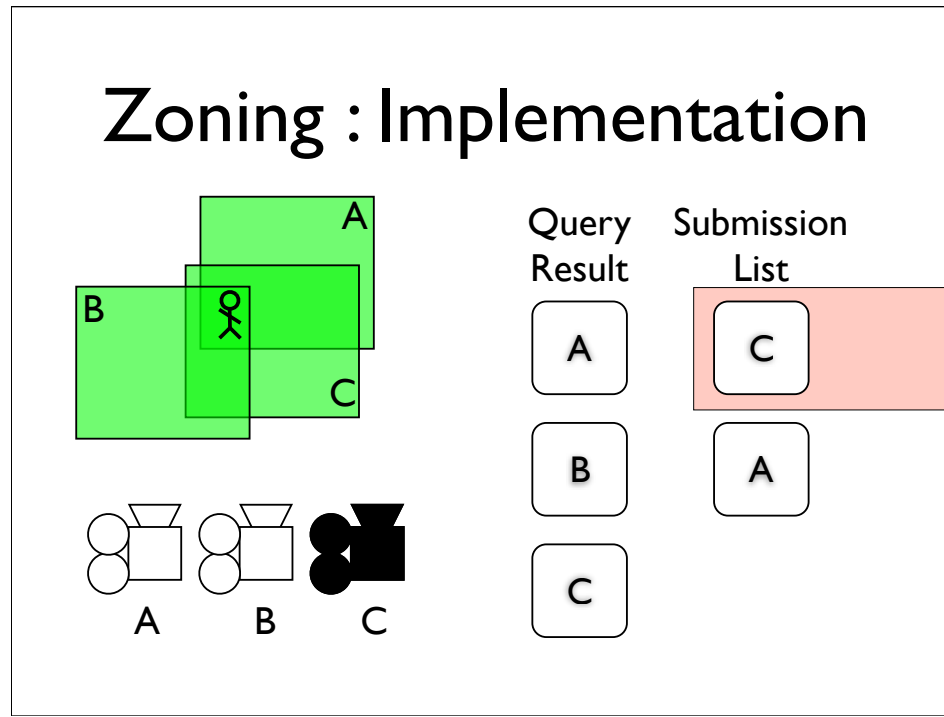
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



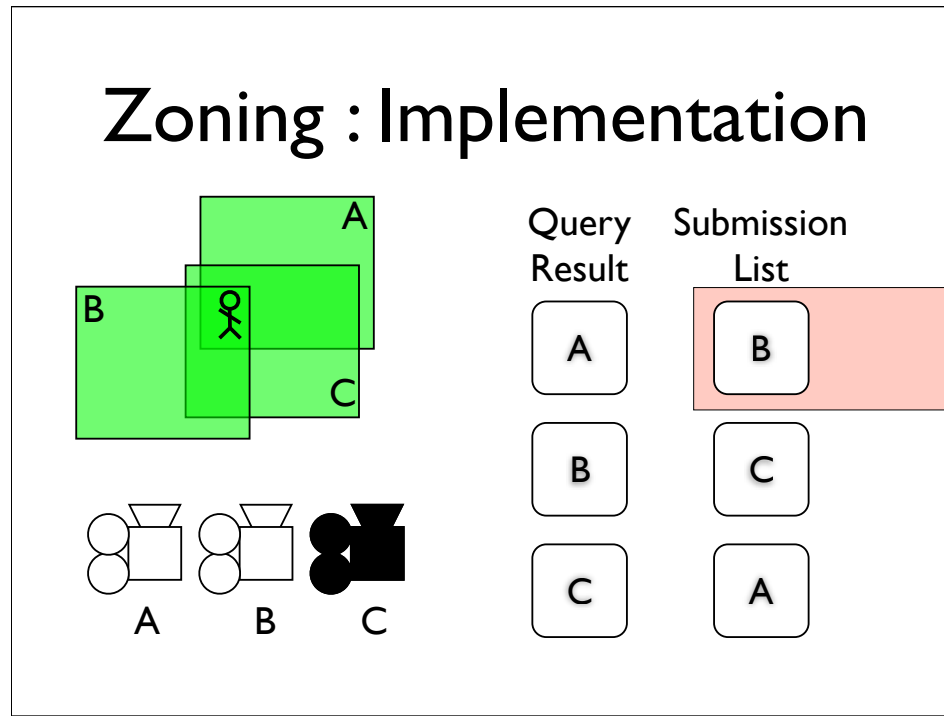
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



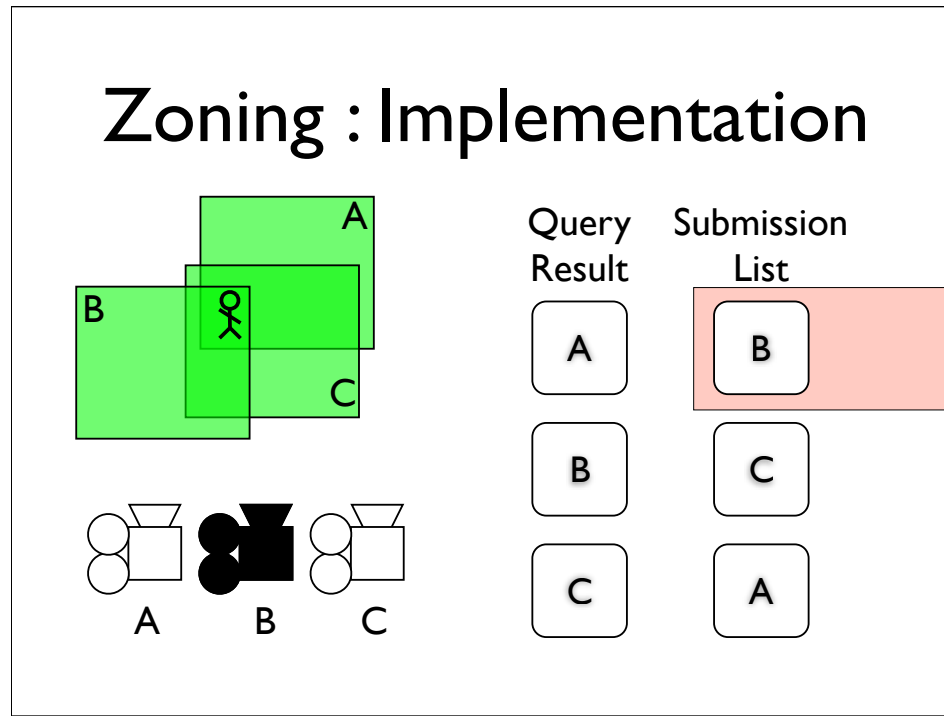
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



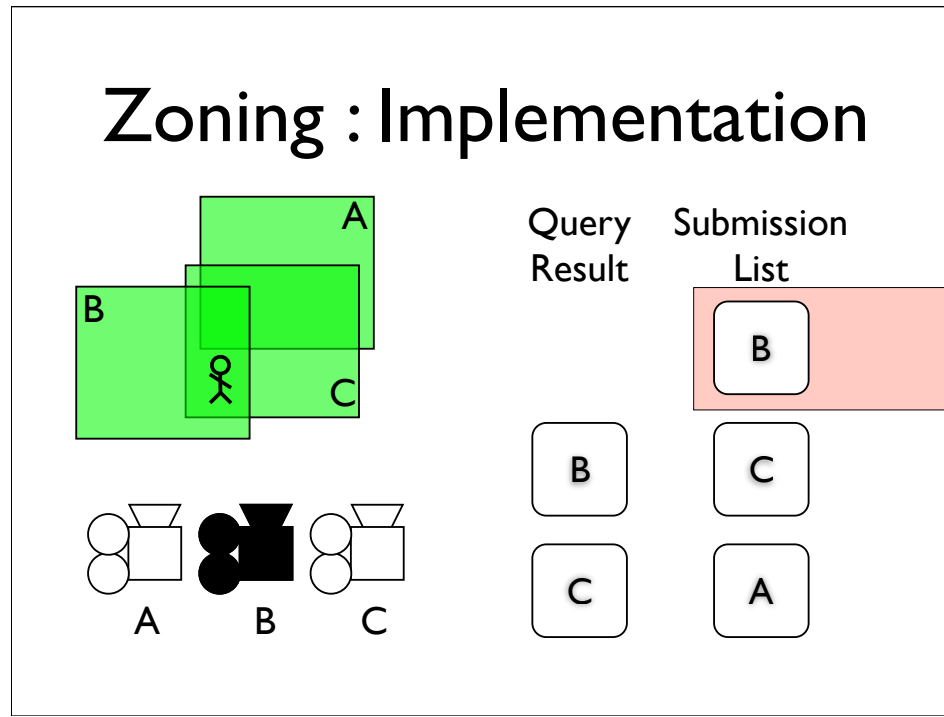
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



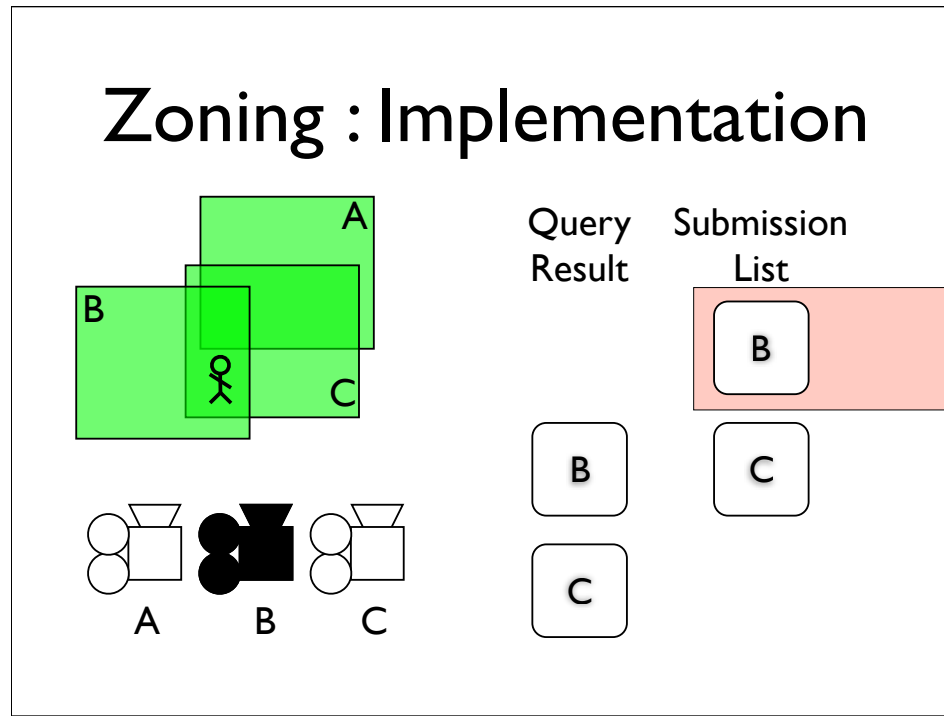
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



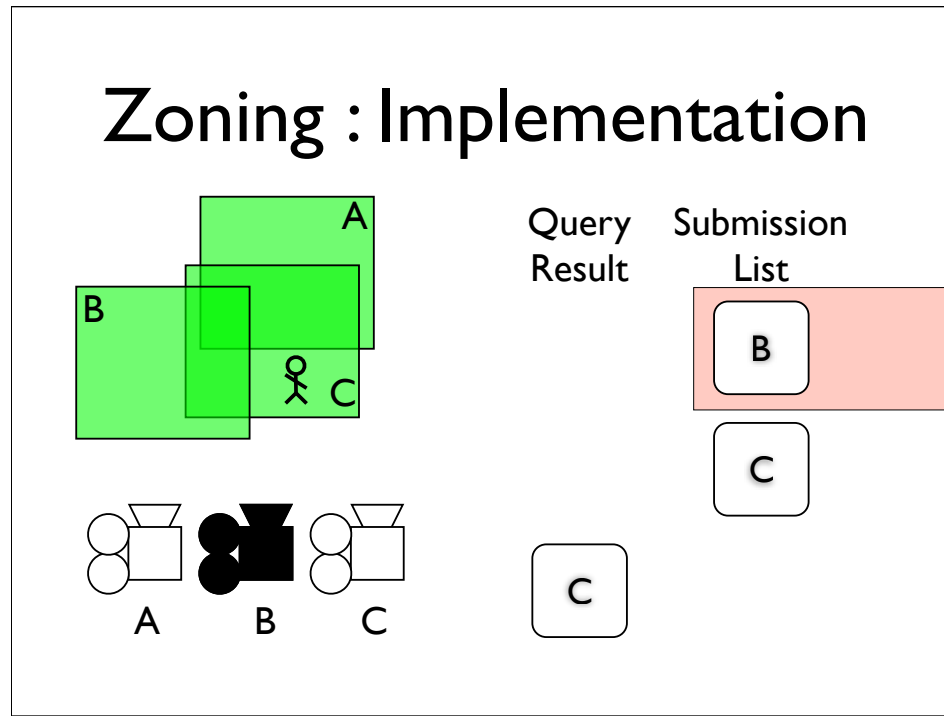
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



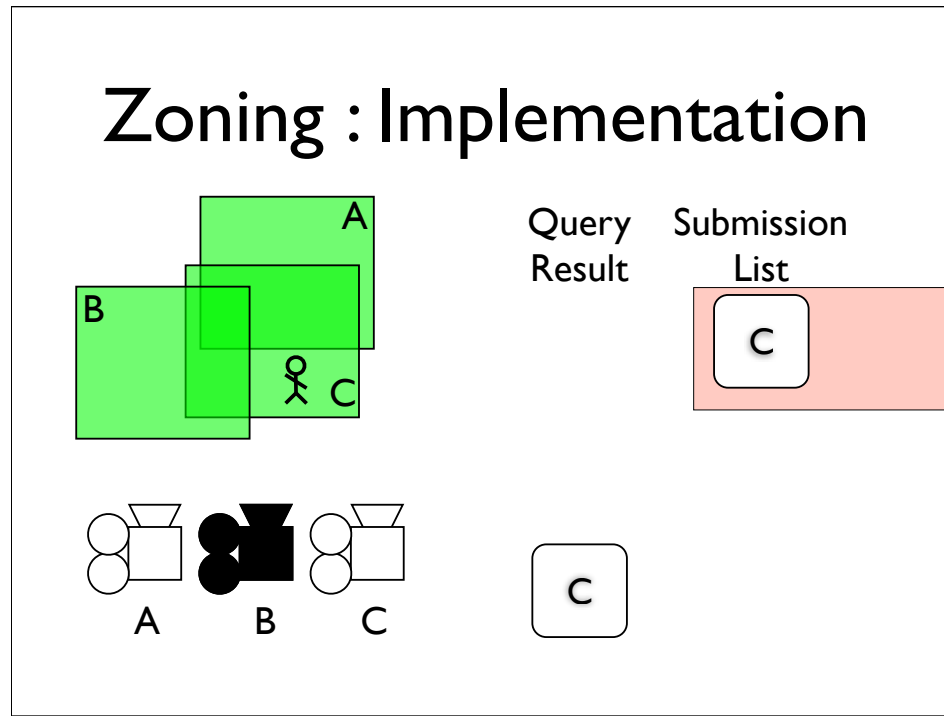
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



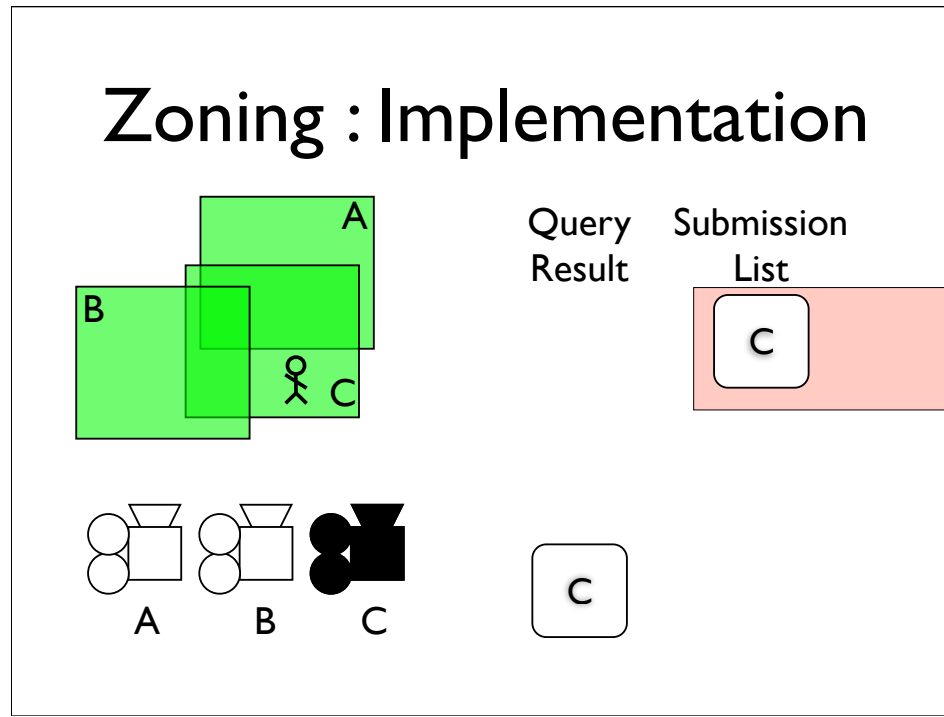
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



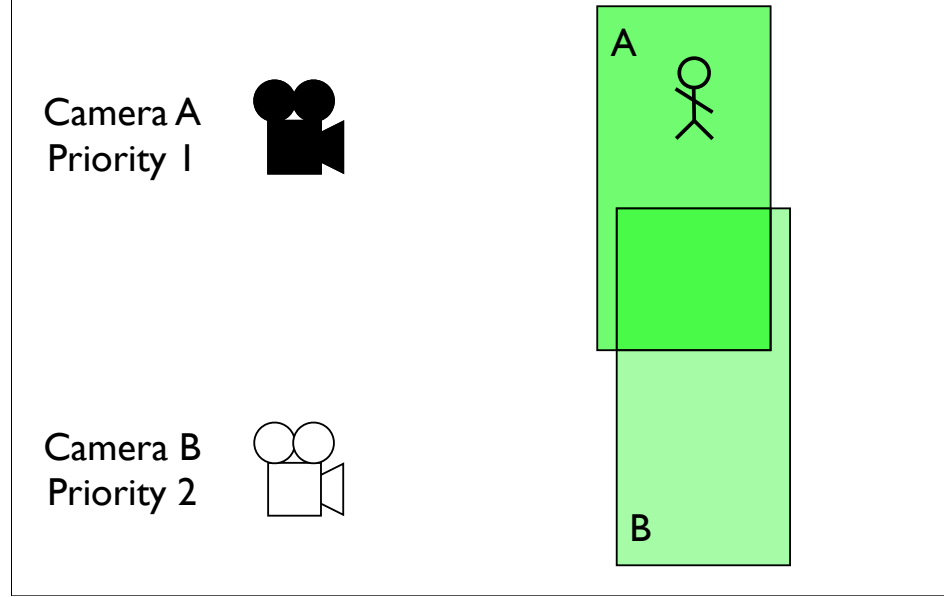
- * Camera A is not in the submission list, so we'll add it.
- * The top item in the submission list has changed, so we'll switch to that camera.
- * When we move here, we get cameras A and C back from the query.
- * Camera A is already in the submission list, but camera C isn't, so we add it to the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

Zoning : Implementation



- * 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

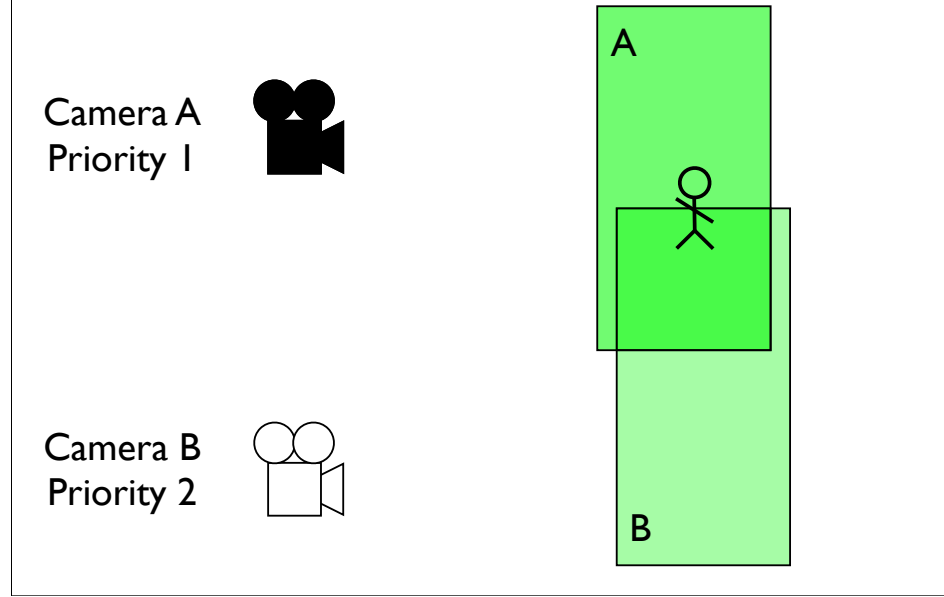
Zoning : Implementation



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.

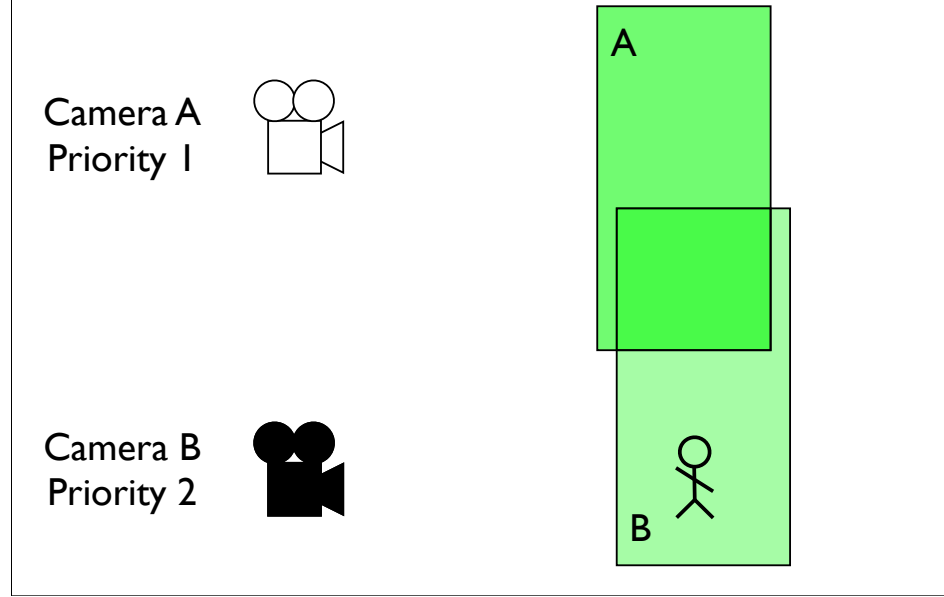
Zoning : Implementation



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.

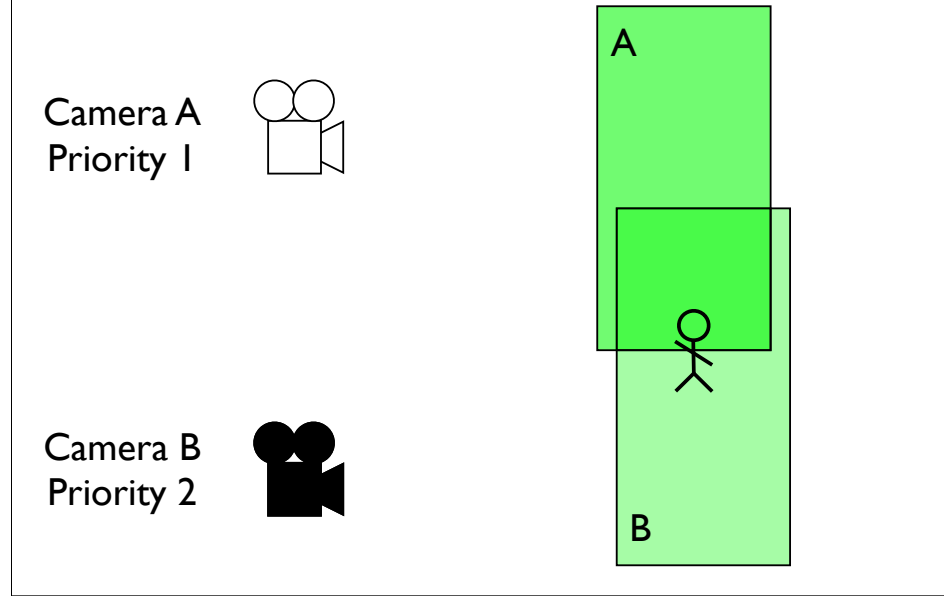
Zoning : Implementation



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.

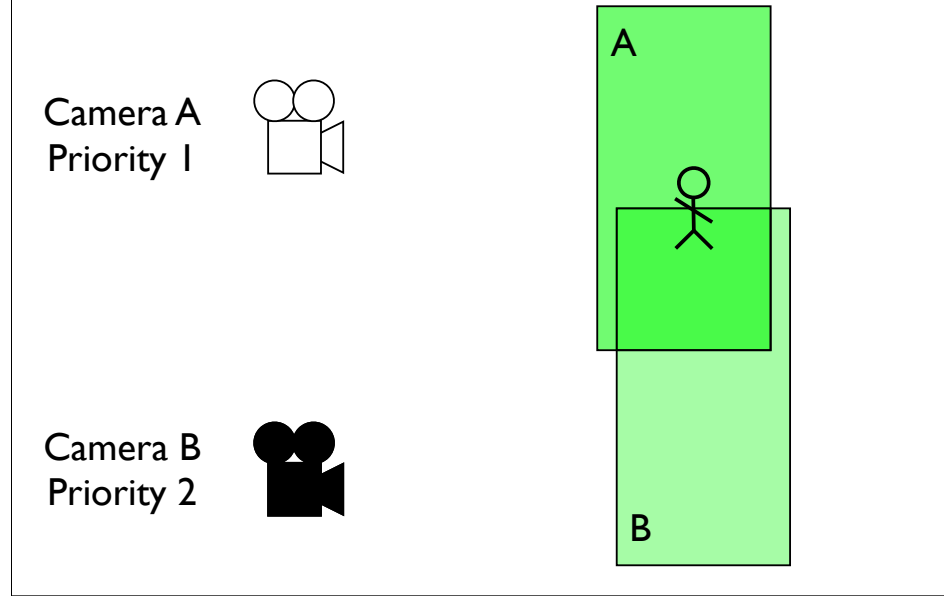
Zoning : Implementation



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.

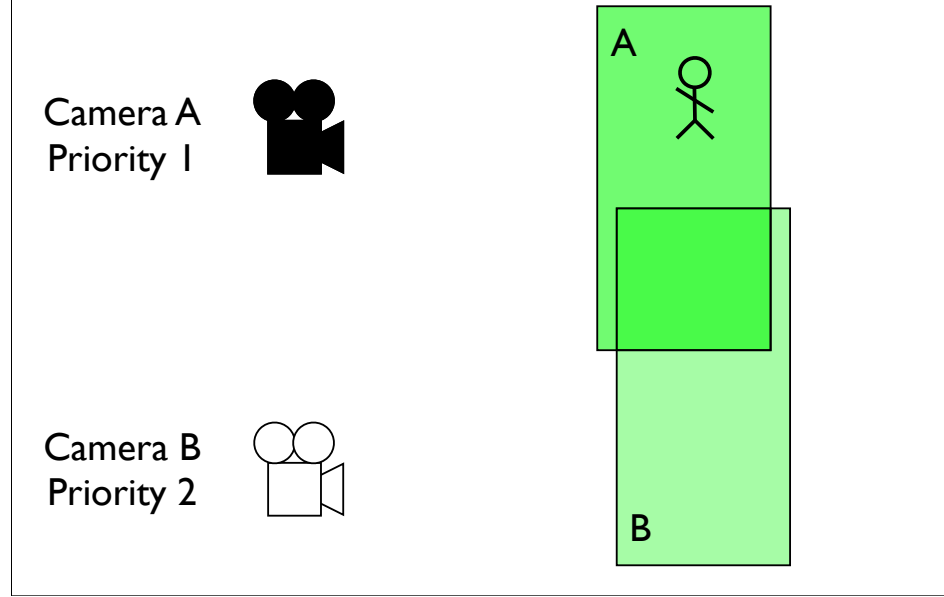
Zoning : Implementation



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.

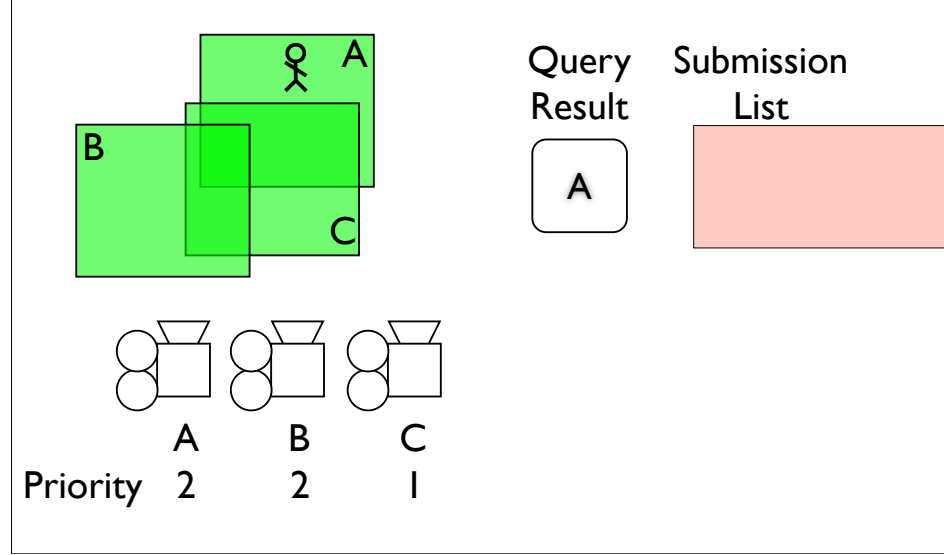
Zoning : Implementation



Higher priorities always override lower ones.

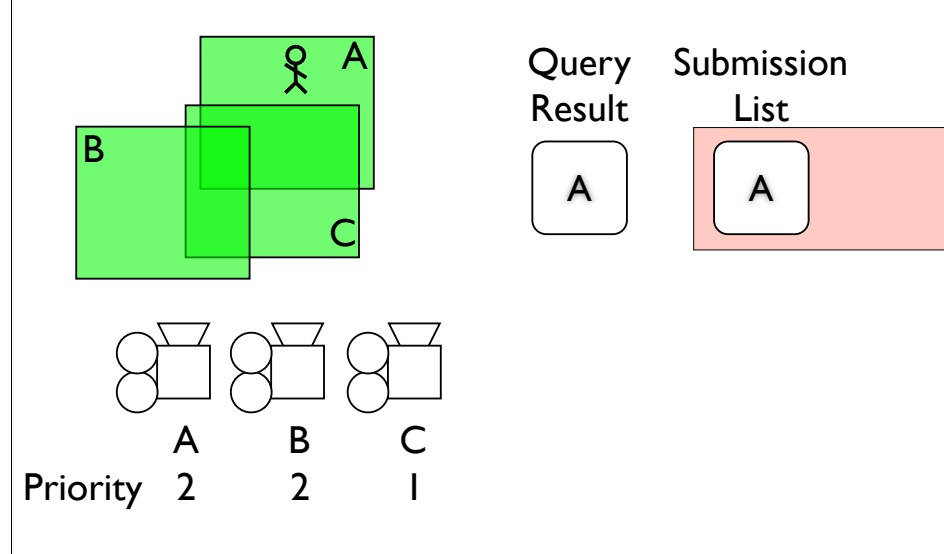
So in this example, we can see that, whenever the player is in zone B, camera B is active, because when we're in the overlap, it has a higher priority.

Zoning : Implementation



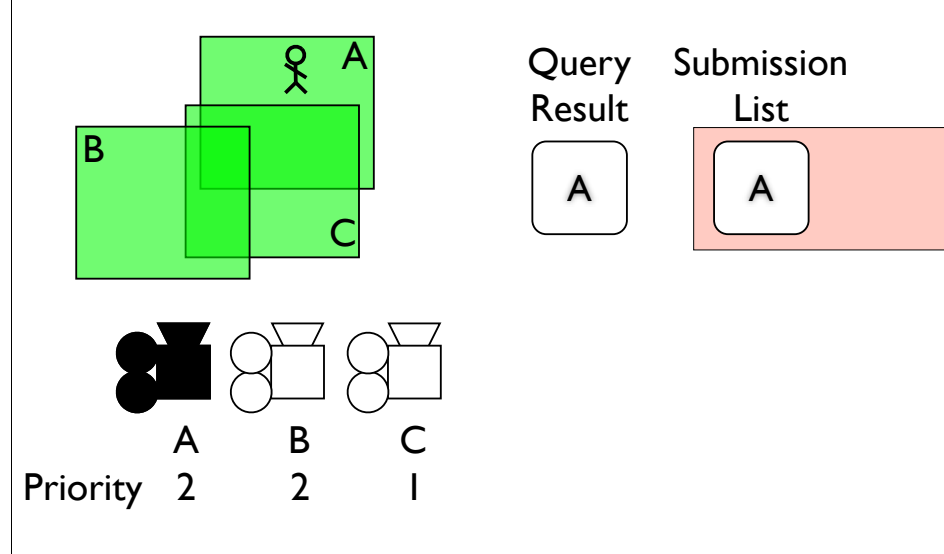
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



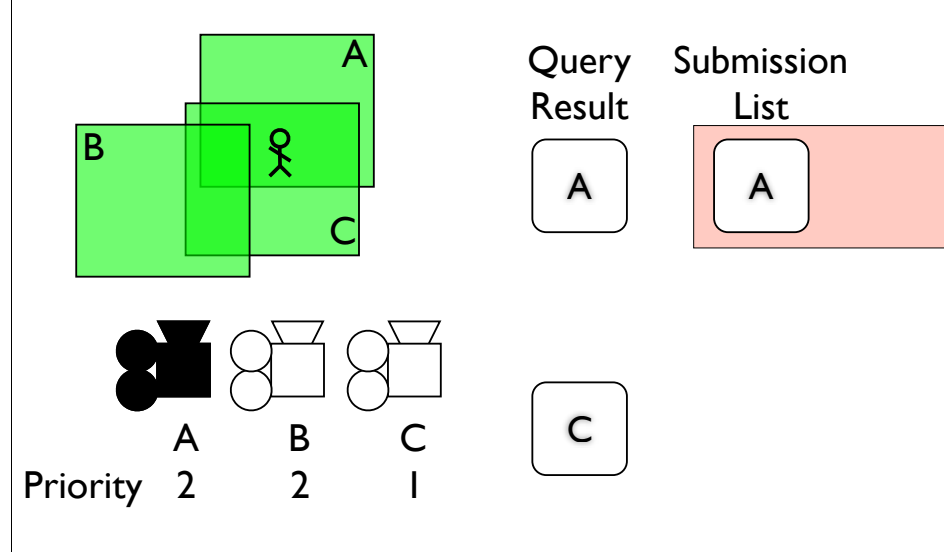
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



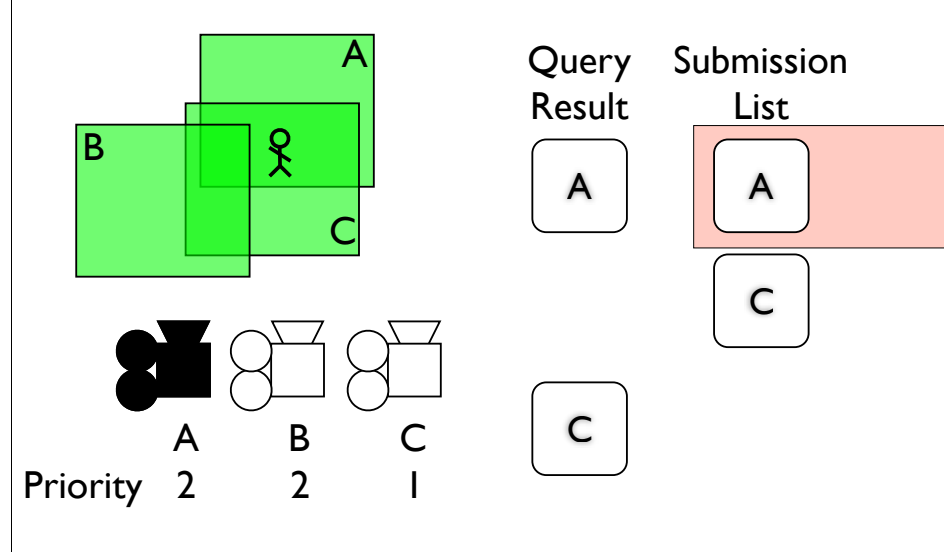
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



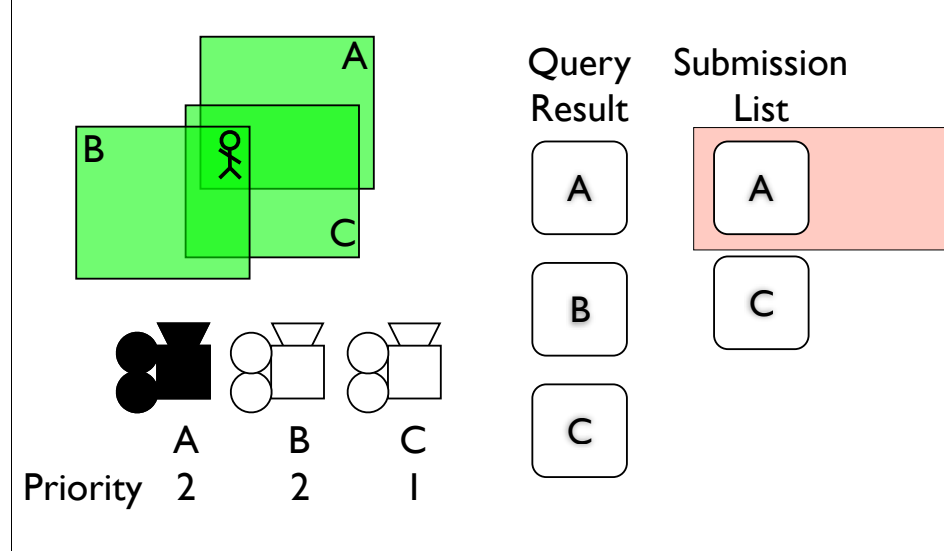
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



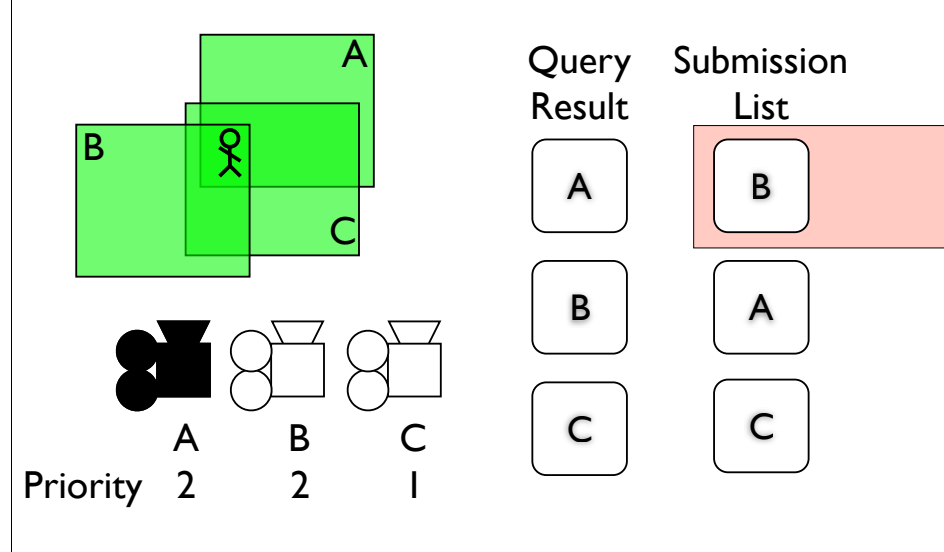
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



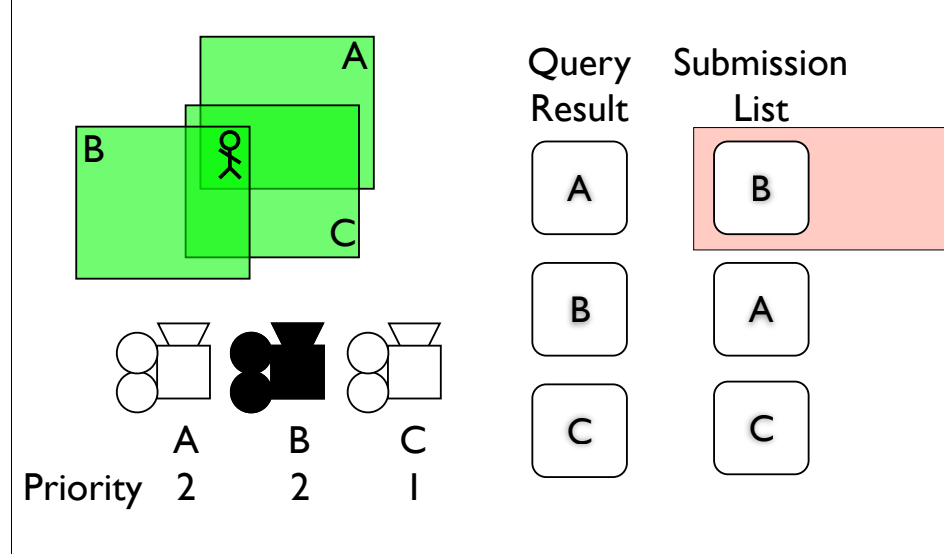
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



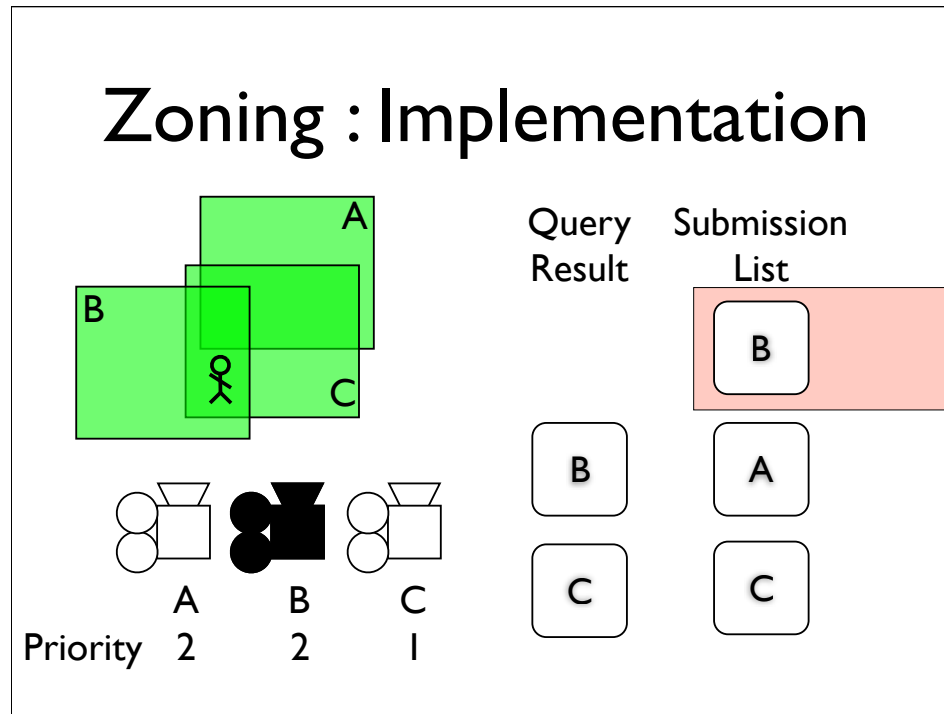
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



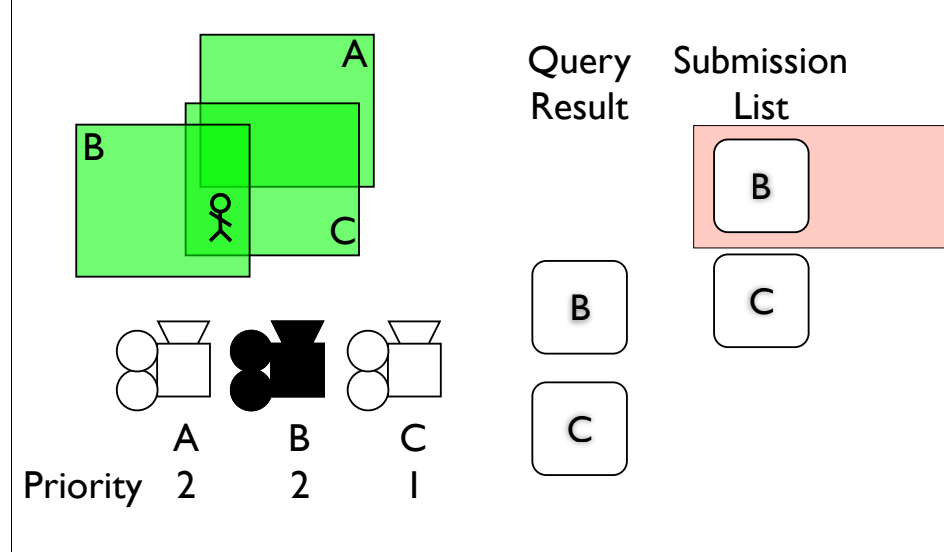
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



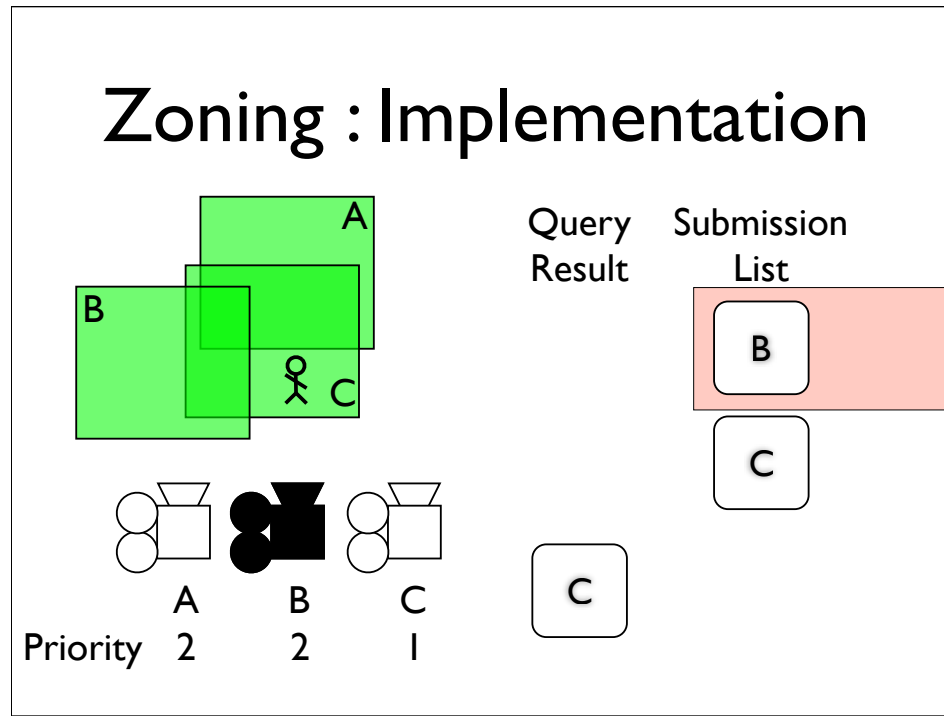
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



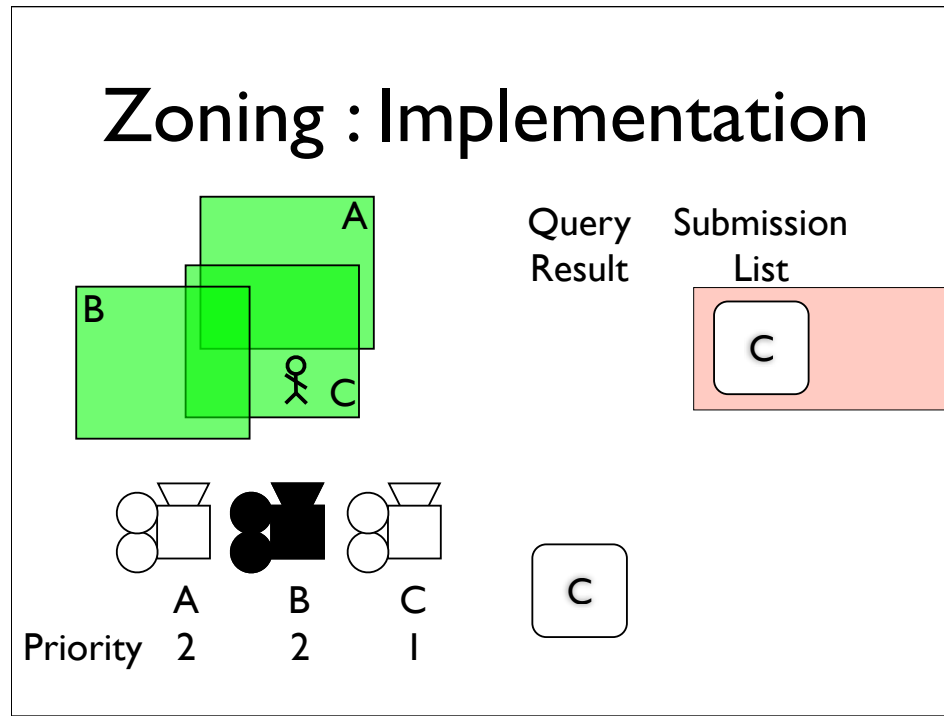
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



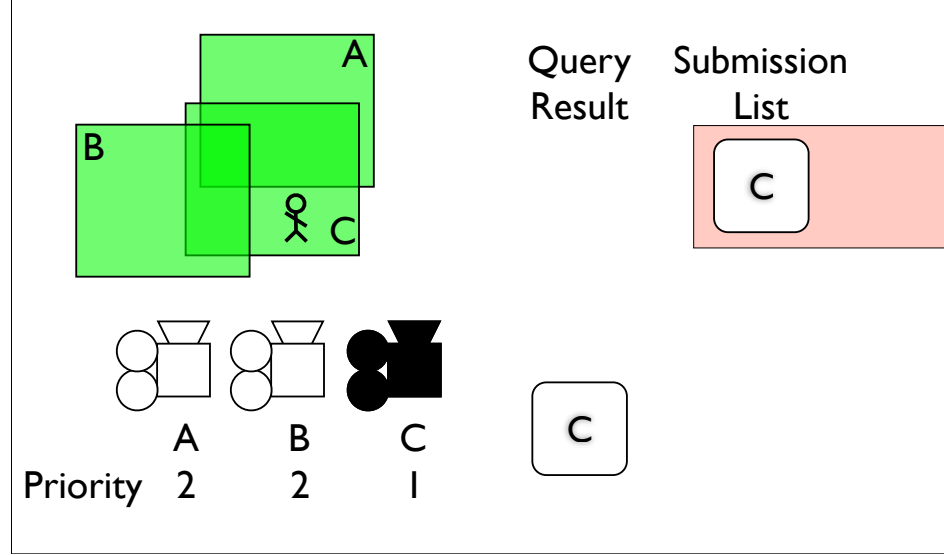
- * Just like before, we start in zone A,
- * and end up starting camera A
- * but when we move into zone C
- * we insert camera C into the submission list below camera A. This is because in order to respect priorities, we maintain the submission list in priority order. So now this time, the top hasn't 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

Zoning : Implementation



- * 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

Zoning : Implementation



- * 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

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

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

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

Overview

- Zoning
- Dynamics
- Blending
- Rails
- Fields

Overview

- Zoning
- **Dynamics**
- Blending
- Rails
- Fields

Dynamics : Objectives

- Control the display of the Player
 - Position
 - Angle
 - Size

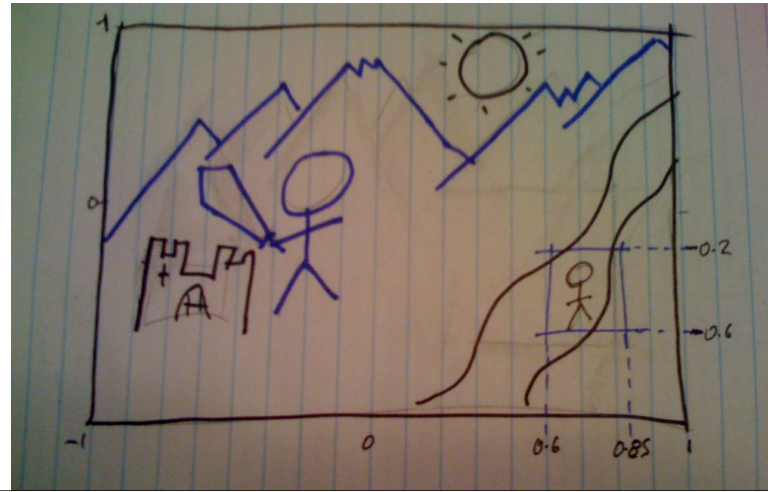
So when we talk about dynamics, we're really talking about moving and orienting the camera to control certain display characteristics of the player.

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

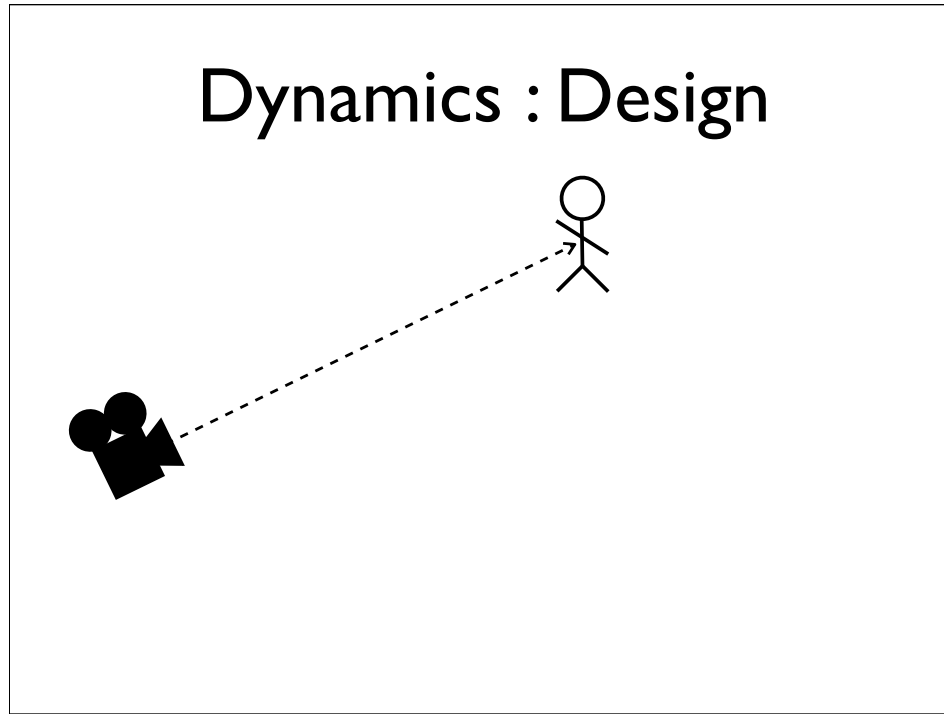


In order to control the player's position on the screen, we define an area of it, within which it is safe for him to move. Safe to move without having to move the camera to keep him in that zone.

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.

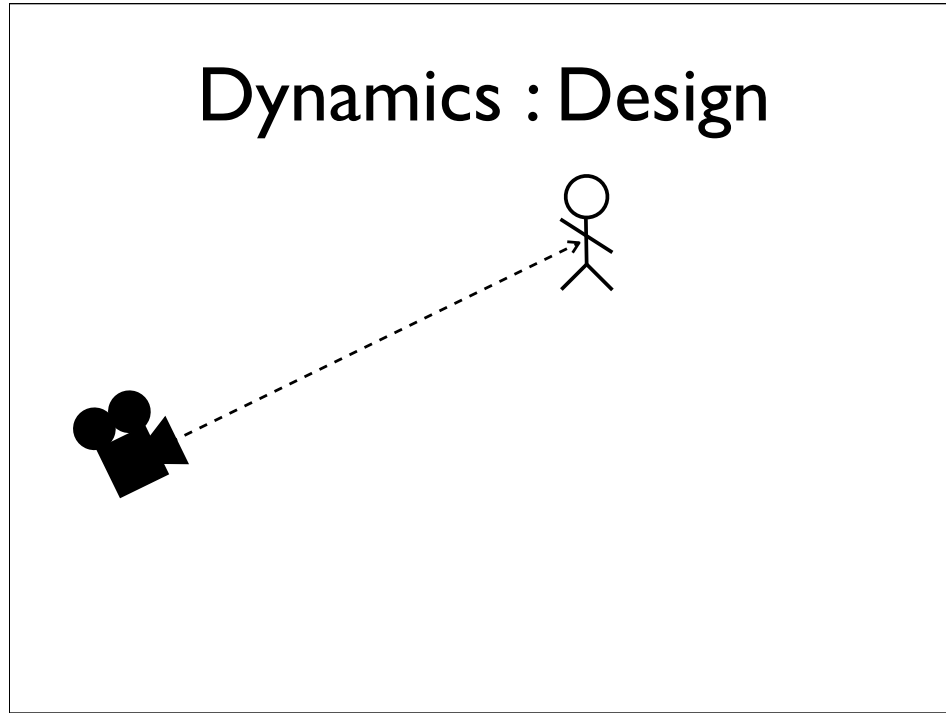
This is represented to the designer as a pair of resolution independent co-ordinates

Dynamics : Design



Next we calculate the angle. We can specify the angle that we're viewing the player from as a fixed value. In which case we use the orientation of the camera in Maya to define that value.

Dynamics : Design



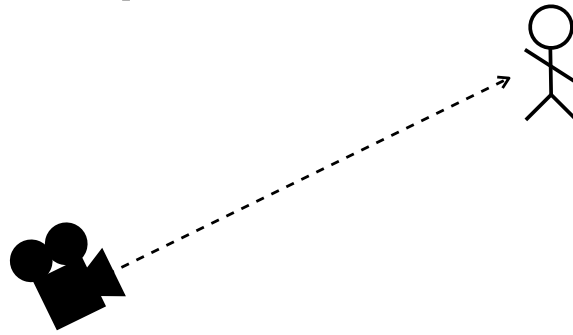
Next we calculate the angle. We can specify the angle that we're viewing the player from as a fixed value. In which case we use the orientation of the camera in Maya to define that value.

Dynamics : Design



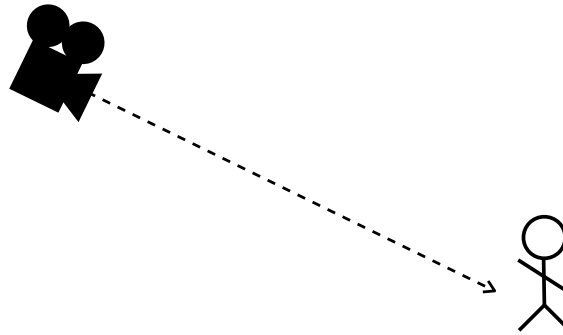
Slightly more interesting, is to calculate it relative to a fixed position in space. Now because each dynamic camera is still defined by a camera in Maya, we already have a convenient fixed position. That of the camera node in Maya.

Dynamics : Design



Slightly more interesting, is to calculate it relative to a fixed position in space. Now because each dynamic camera is still defined by a camera in Maya, we already have a convenient fixed position. That of the camera node in Maya.

Dynamics : Design



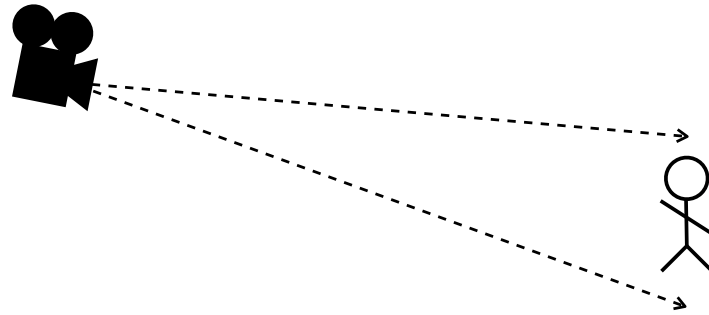
Slightly more interesting, is to calculate it relative to a fixed position in space. Now because each dynamic camera is still defined by a camera in Maya, we already have a convenient fixed position. That of the camera node in Maya.

Dynamics : Design



Slightly more interesting, is to calculate it relative to a fixed position in space. Now because each dynamic camera is still defined by a camera in Maya, we already have a convenient fixed position. That of the camera node in Maya.

Dynamics : Design

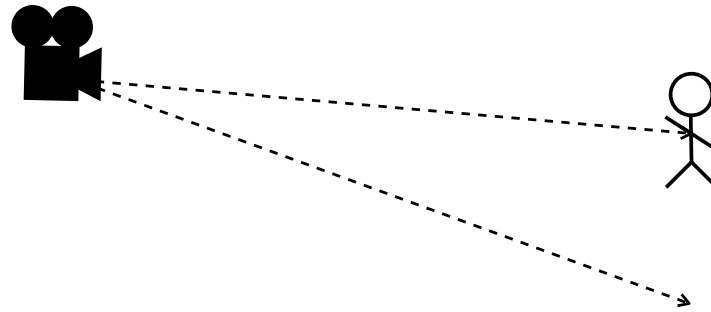


...and we can constrain it to within a fixed range.

Now to specify that range, we again, use the orientation of the camera node in Maya,

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

Dynamics : Design

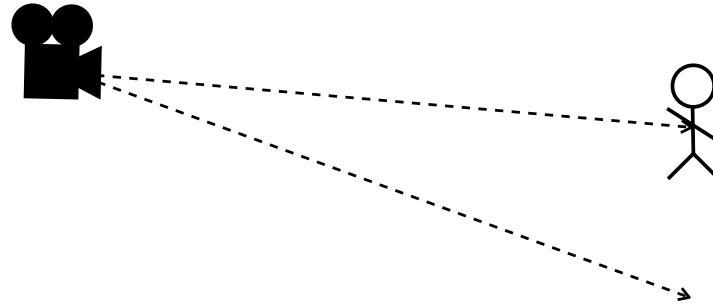


...and we can constrain it to within a fixed range.

Now to specify that range, we again, use the orientation of the camera node in Maya,

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

Dynamics : Design

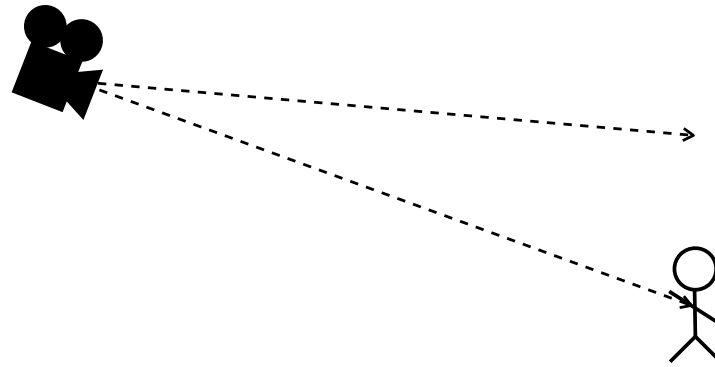


...and we can constrain it to within a fixed range.

Now to specify that range, we again, use the orientation of the camera node in Maya,

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

Dynamics : Design

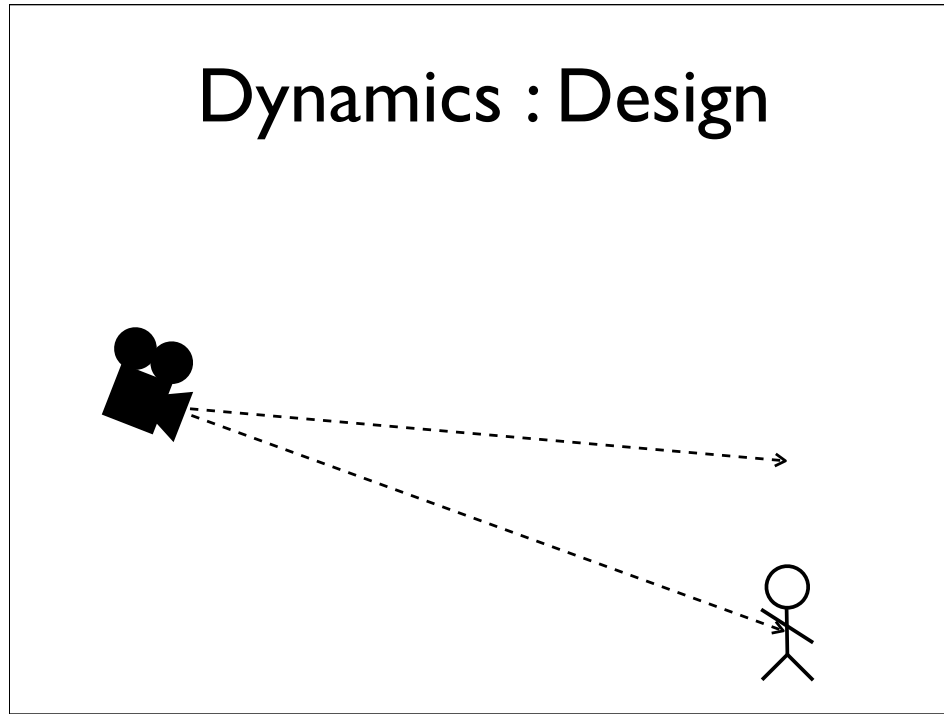


...and we can constrain it to within a fixed range.

Now to specify that range, we again, use the orientation of the camera node in Maya,

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

Dynamics : Design

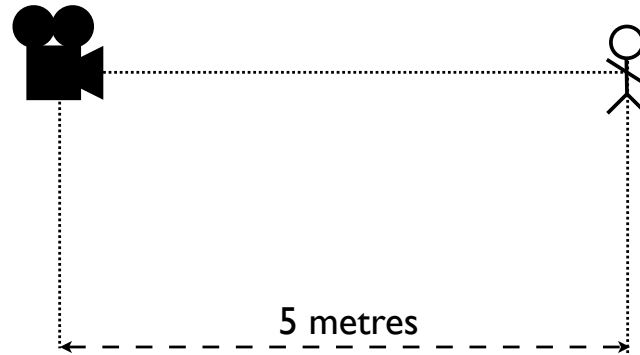


...and we can constrain it to within a fixed range.

Now to specify that range, we again, use the orientation of the camera node in Maya,

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

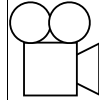
Dynamics : Design



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.

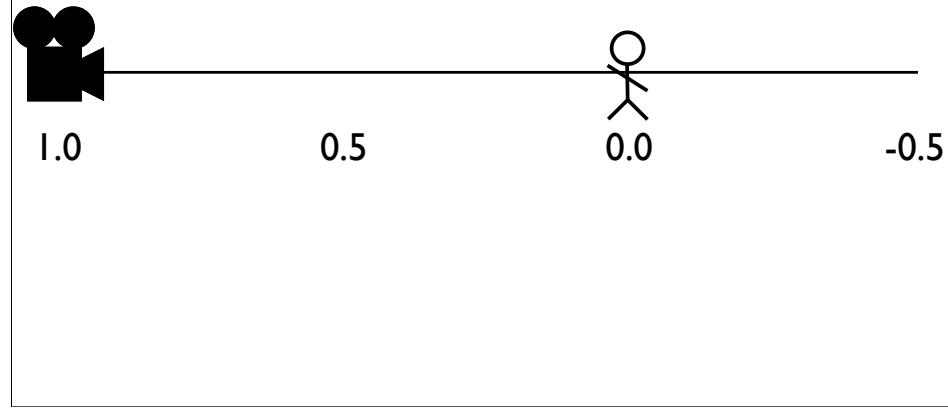
Dynamics : Design



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

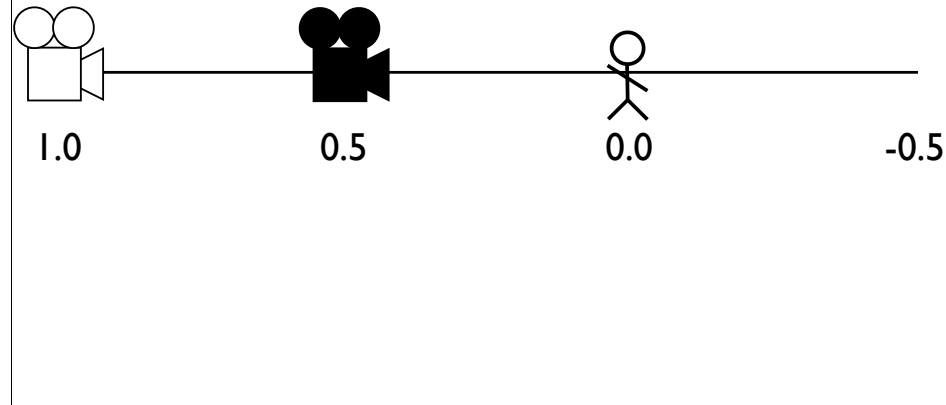
Dynamics : Design



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

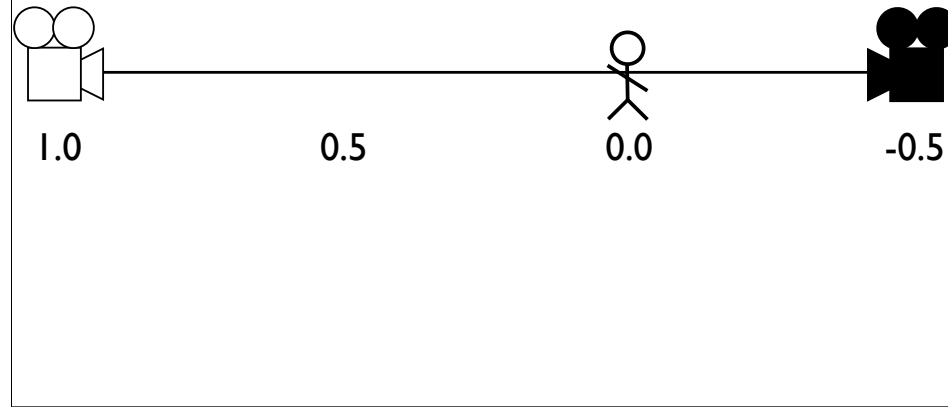
Dynamics : Design



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

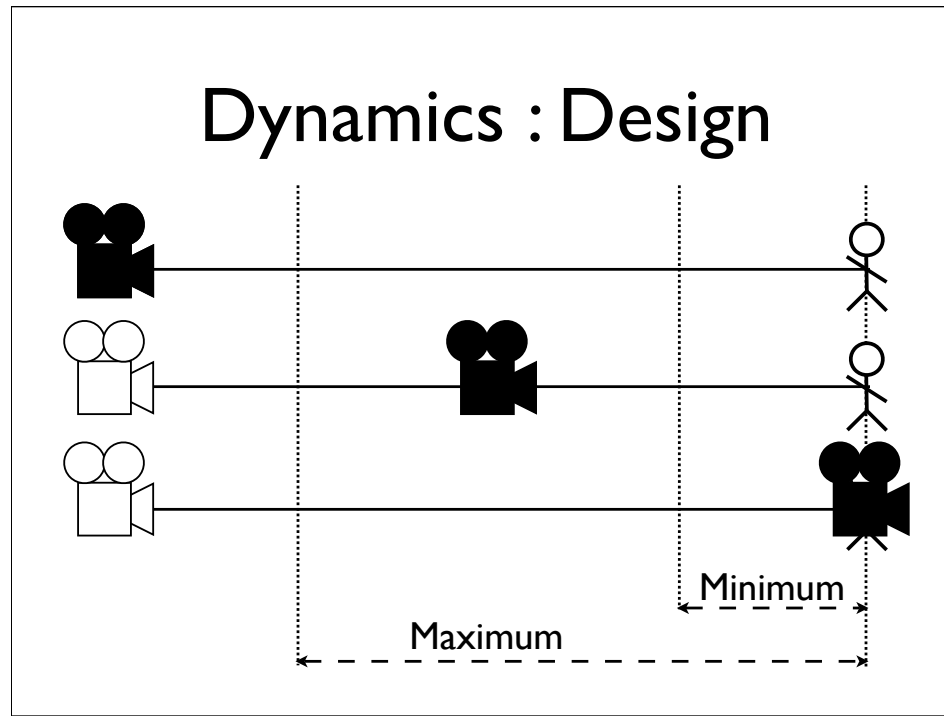
With negative values being behind the player, and in those cases, we automatically turn the camera around, to look back at the player

Dynamics : Design



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

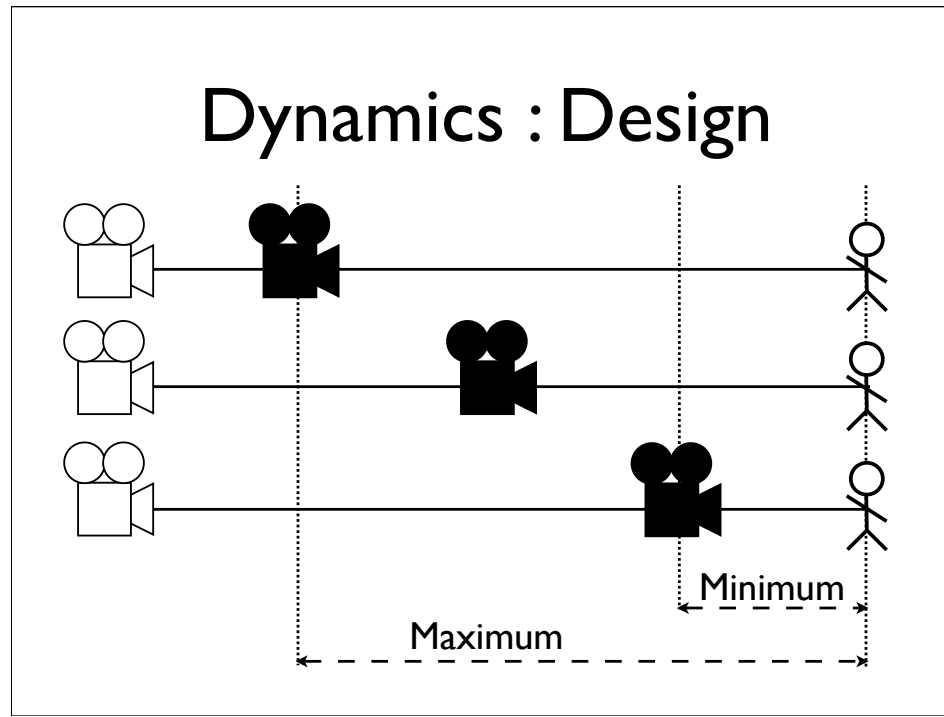
With negative values being behind the player, and in those cases, we automatically turn the camera around, to look back at the player



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 constraints to represent a fixed distance.



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 constraints to represent a fixed distance.

Dynamics : Implementation



So that's how we let the designer control the three defining properties of the camera

the position of the player on screen

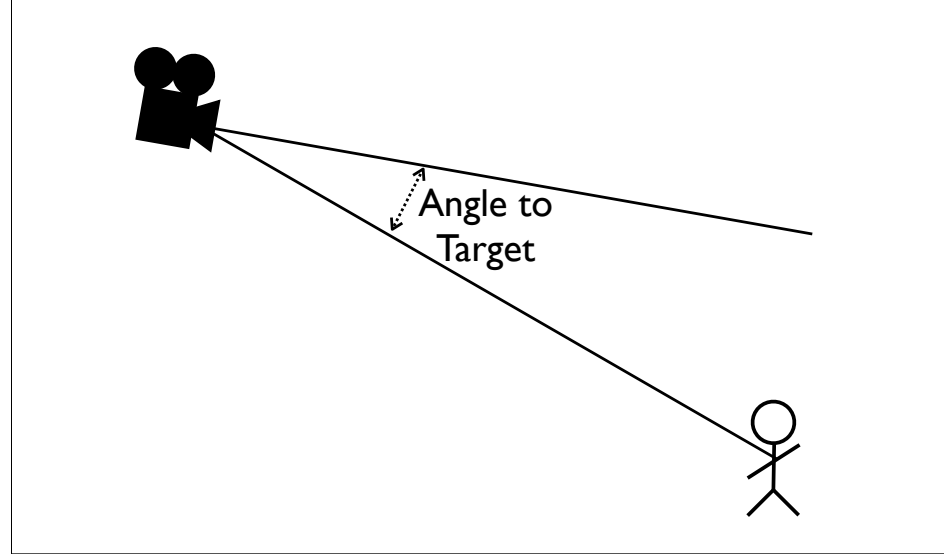
the angle we're looking at him from, or rather, the orientation of the camera

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

Dynamics : Implementation



So that's how we let the designer control the three defining properties of the camera

the position of the player on screen

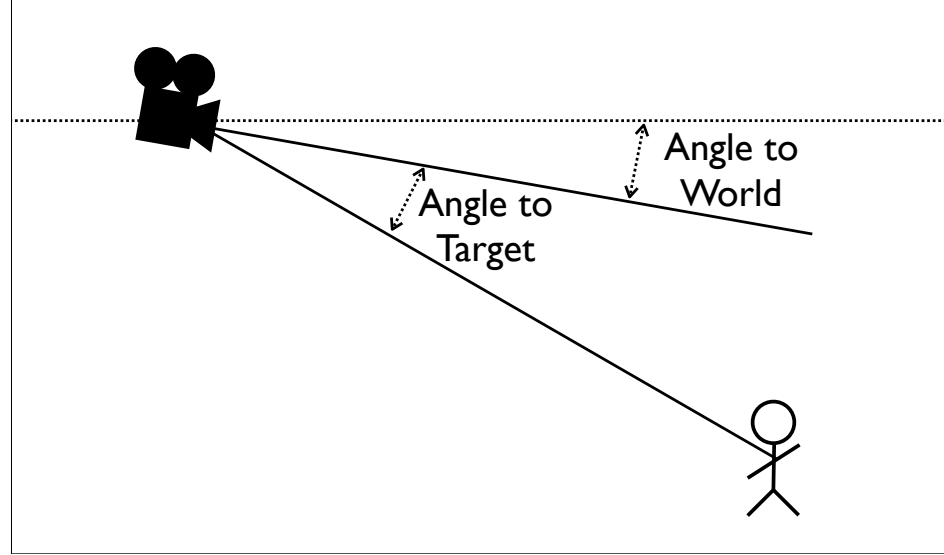
the angle we're looking at him from, or rather, the orientation of the camera

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

Dynamics : Implementation



So that's how we let the designer control the three defining properties of the camera

the position of the player on screen

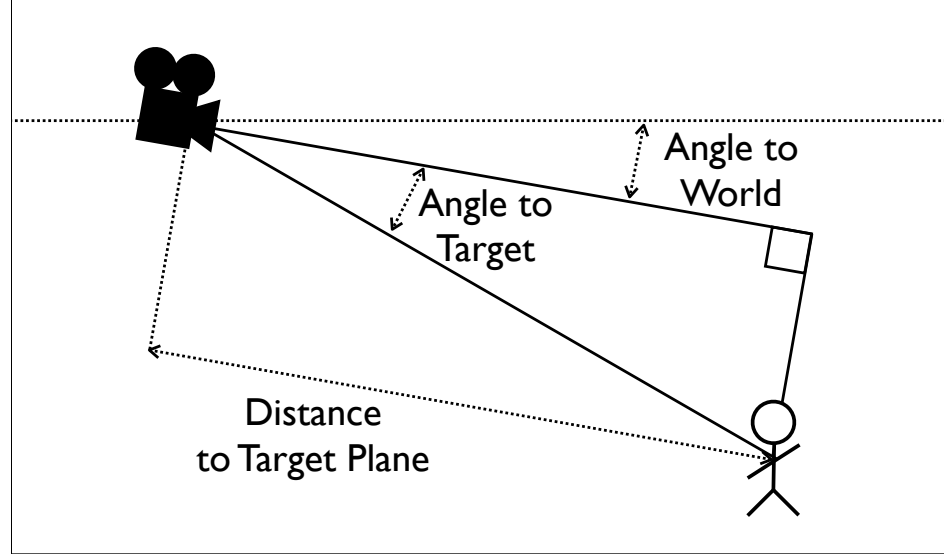
the angle we're looking at him from, or rather, the orientation of the camera

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

Dynamics : Implementation



So that's how we let the designer control the three defining properties of the camera

the position of the player on screen

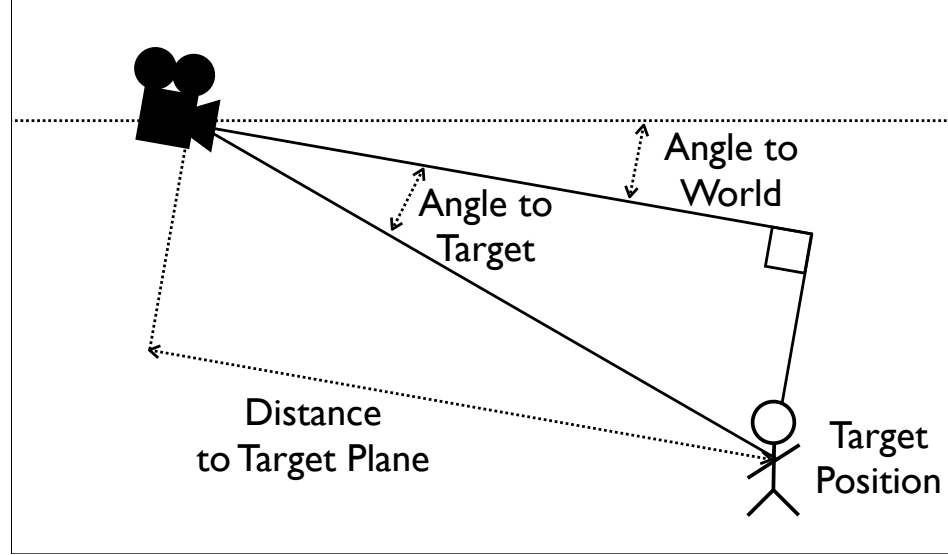
the angle we're looking at him from, or rather, the orientation of the camera

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

Dynamics : Implementation



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

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
- Fields

Overview

- Zoning
- Dynamics
- **Blending**
- Rails
- Fields

Blending : Overview

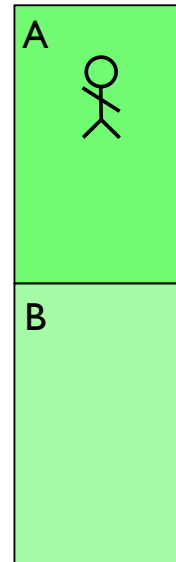
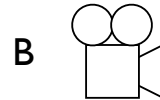
- Timers
- Ease
- Blend Space

Timers, which track and update each blend

Ease, which controls the smoothness of a blend

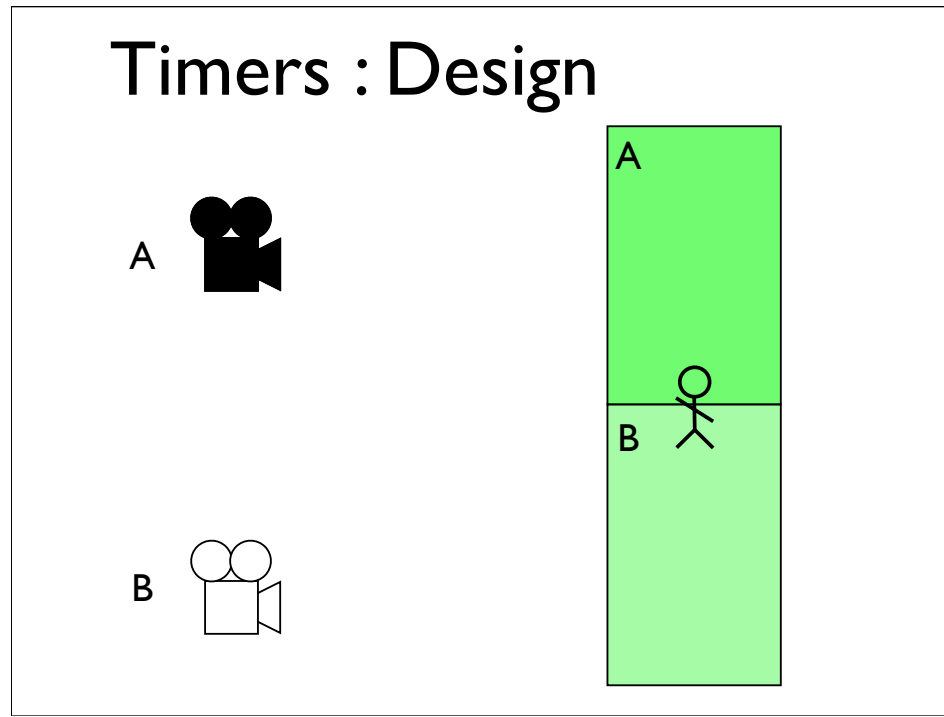
and Blend Space, where I'll define what a blend between two cameras actually does.

Timers : Design



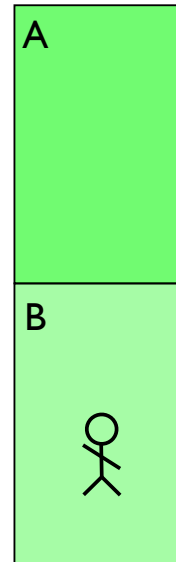
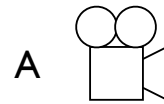
when we start a new camera, we don't cut to it, but blend into it over a fixed period of time.

Timers : Design



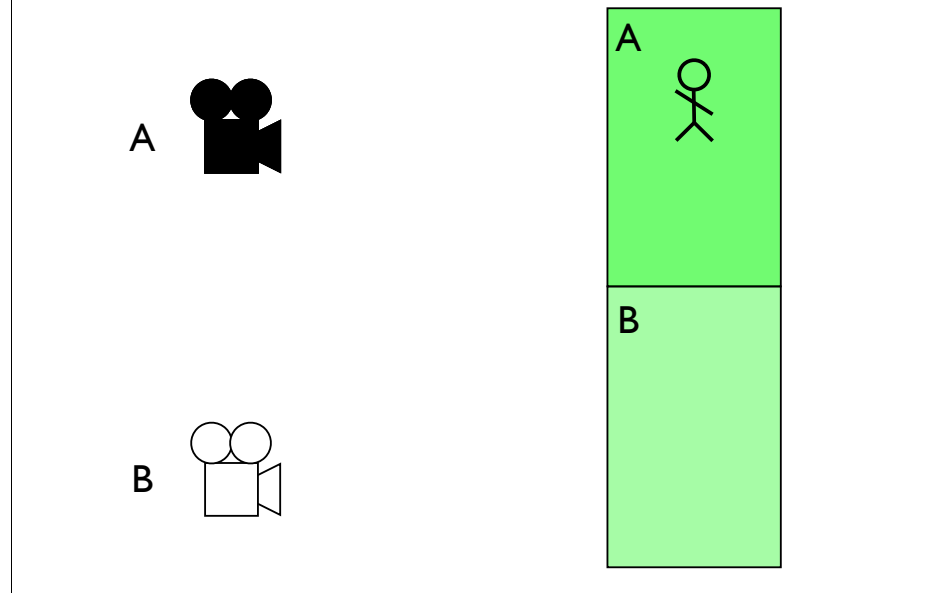
when we start a new camera, we don't cut to it, but blend into it over a fixed period of time.

Timers : Design



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

Timers : Design

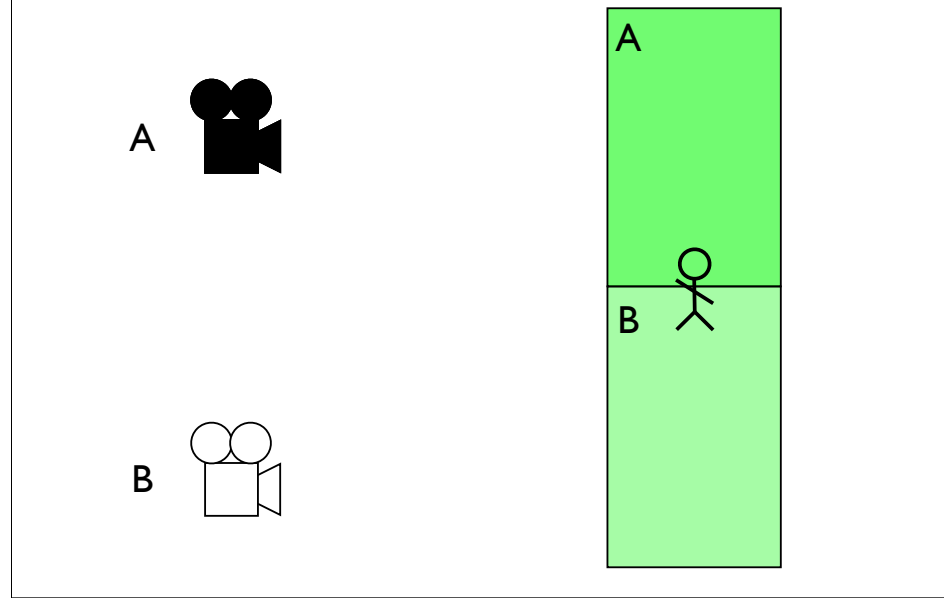


And when I say blend, I mean creating a third camera from varying proportions of two other cameras.

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 : Design

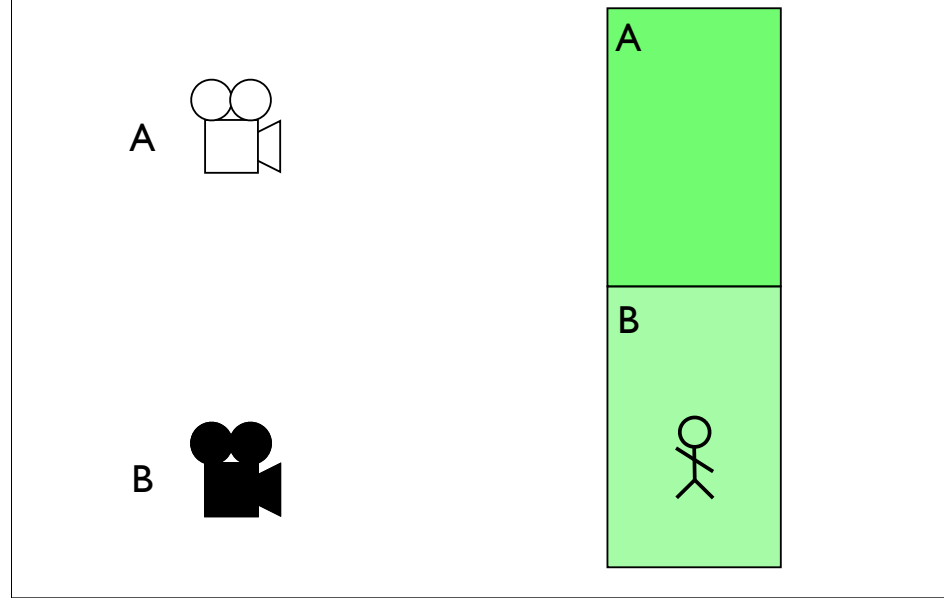


And when I say blend, I mean creating a third camera from varying proportions of two other cameras.

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 : Design



And when I say blend, 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.

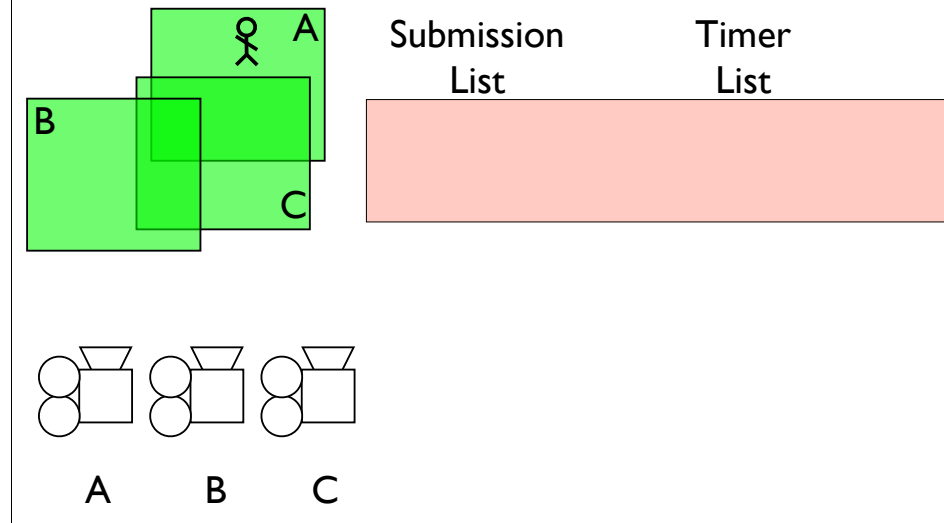
Timers : Implementation

- 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.

Timers : Implementation



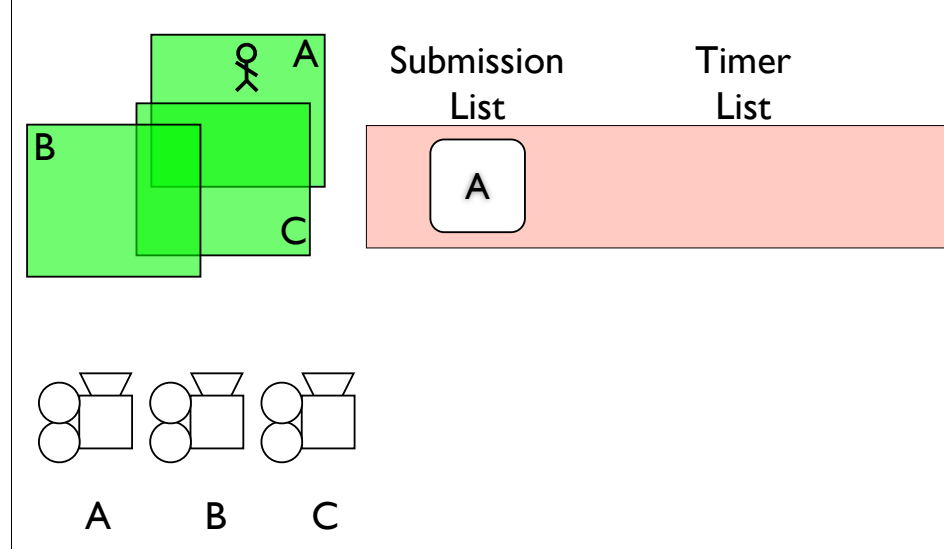
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



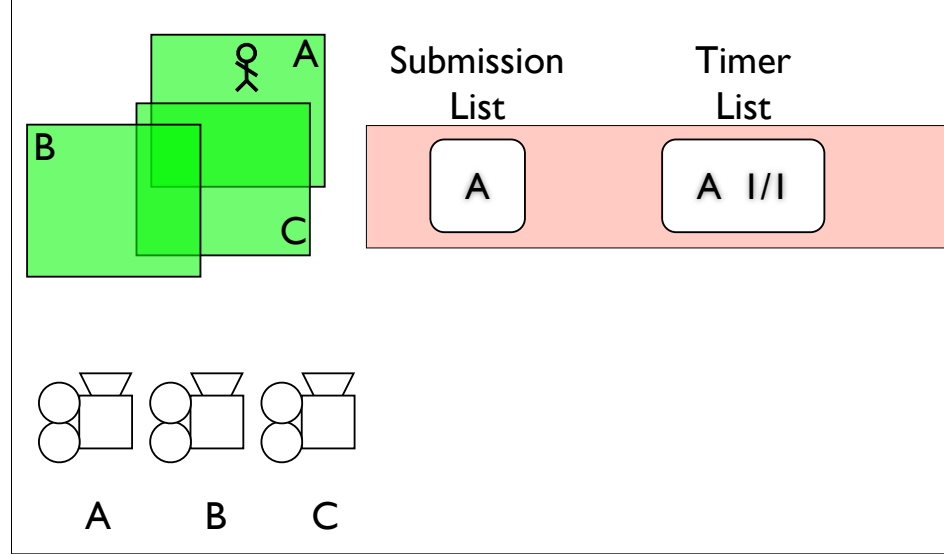
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



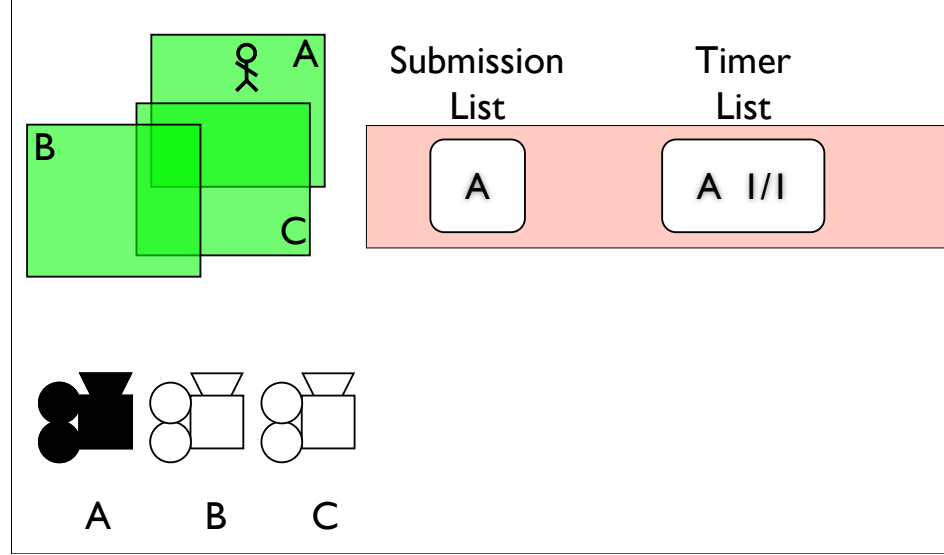
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



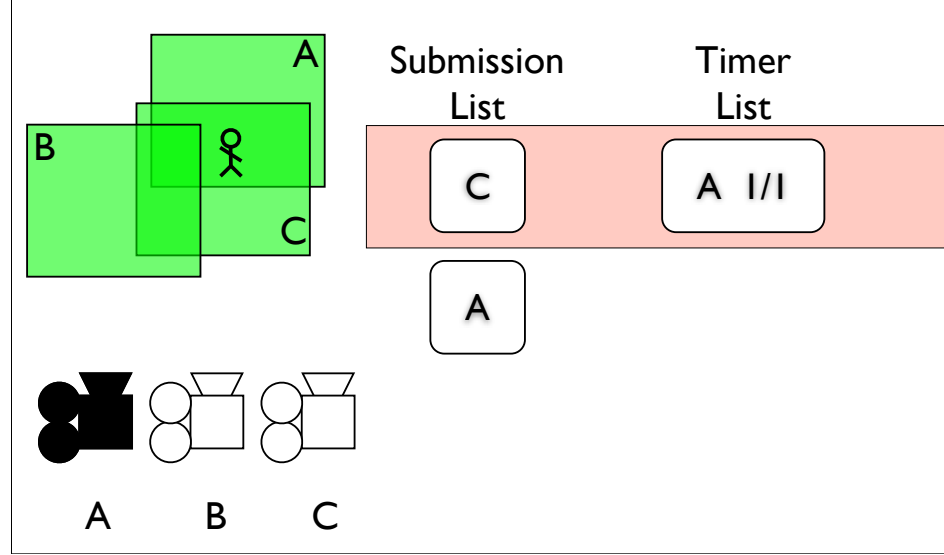
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

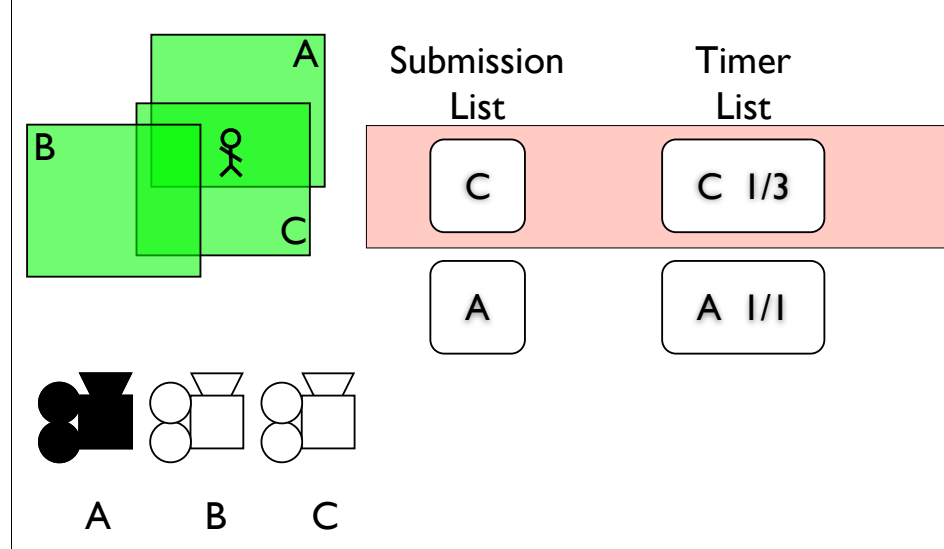
Timers : Implementation



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 and b

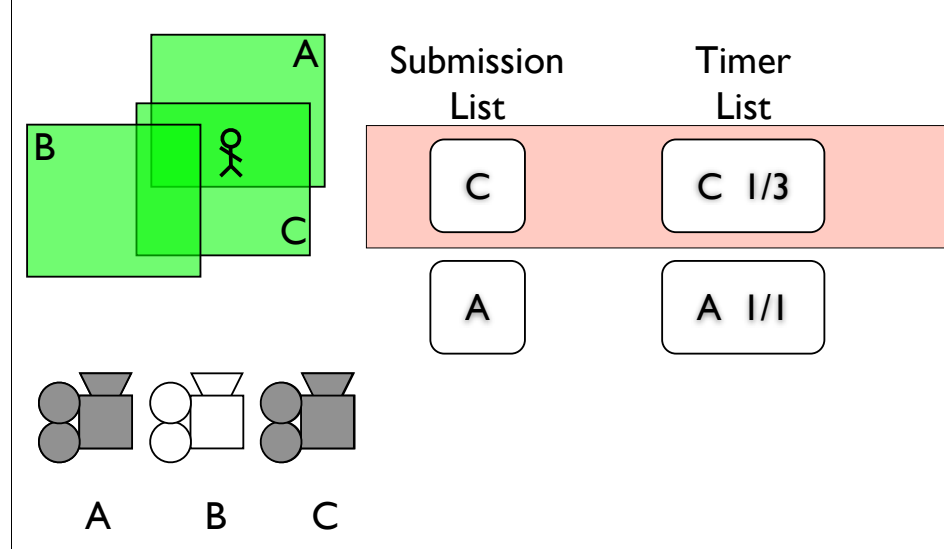
Timers : Implementation



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 b and c

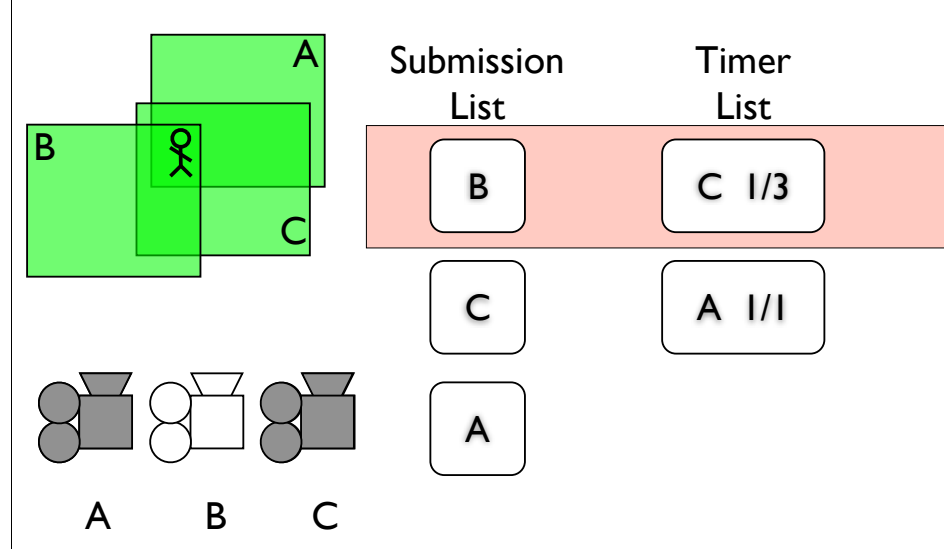
Timers : Implementation



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 b and c

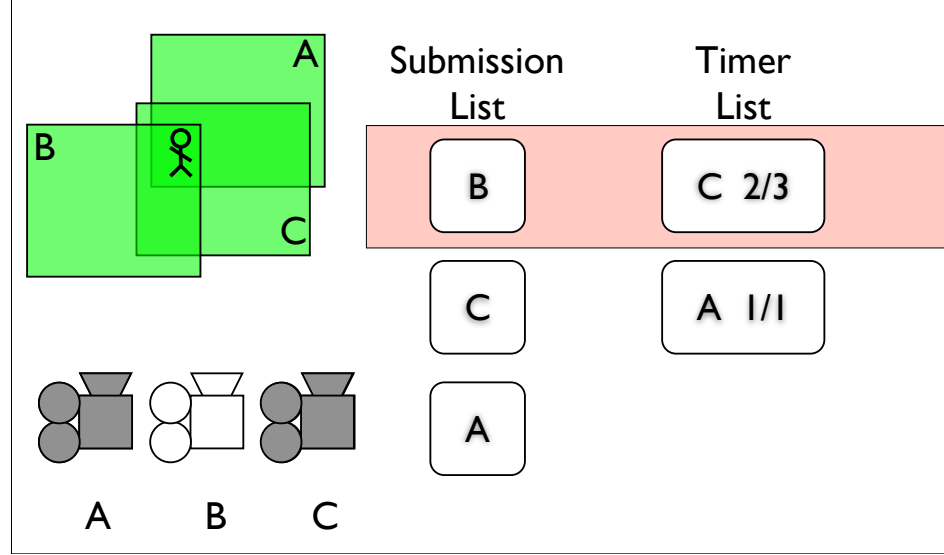
Timers : Implementation



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 b and c

Timers : Implementation



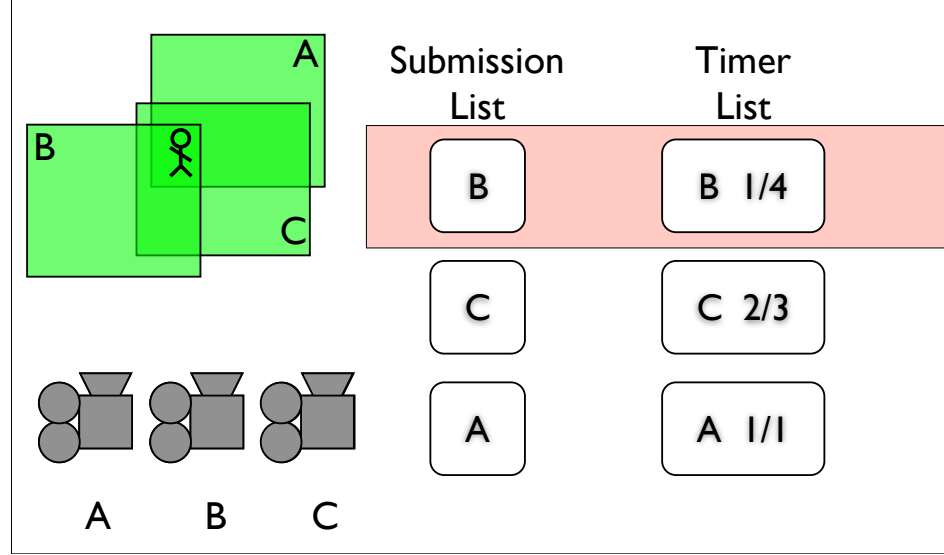
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



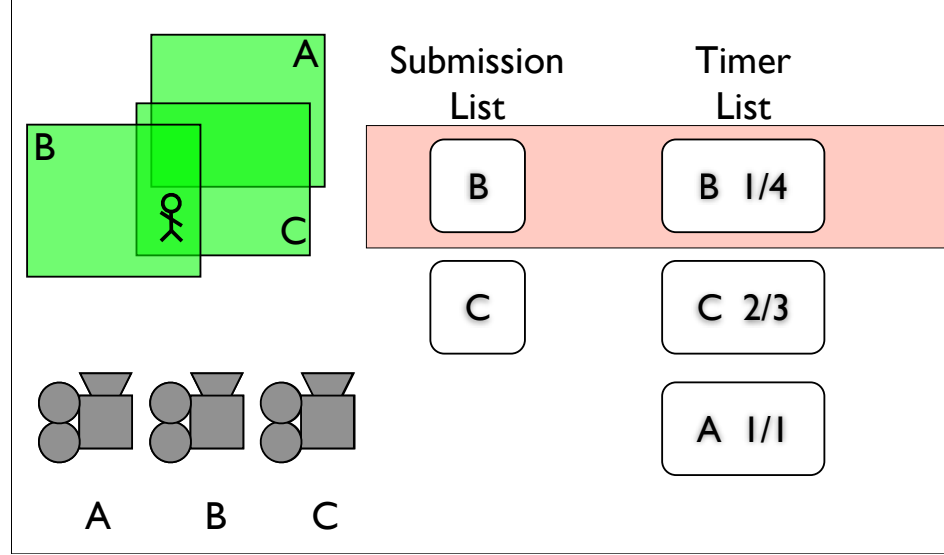
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



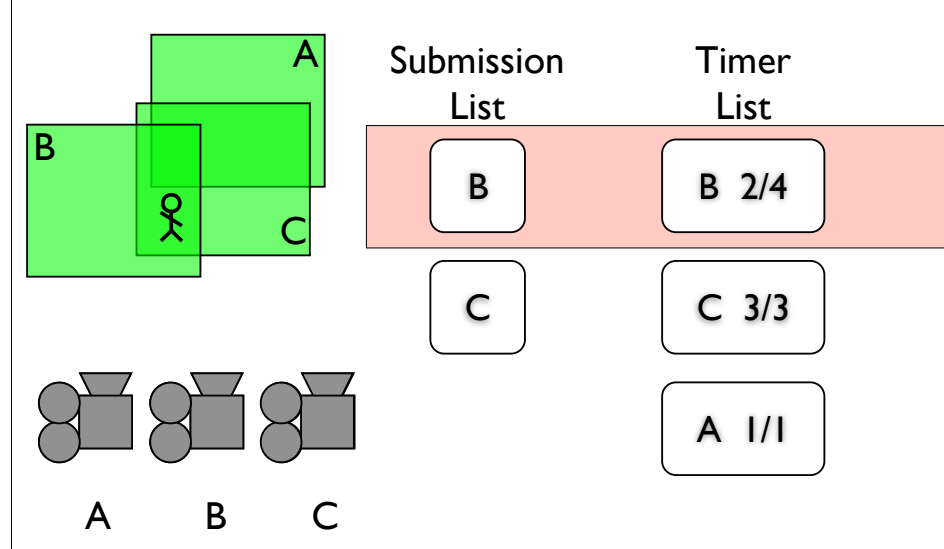
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



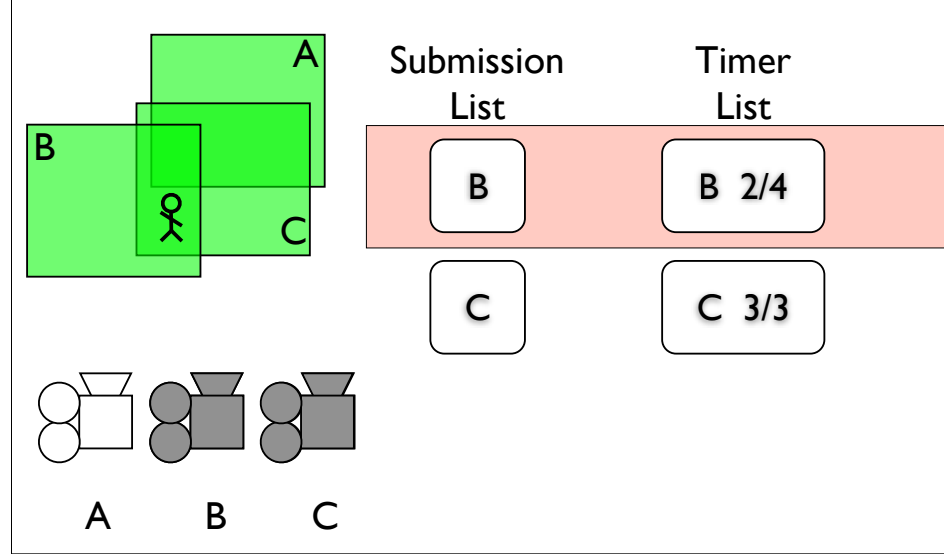
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

Timers : Implementation



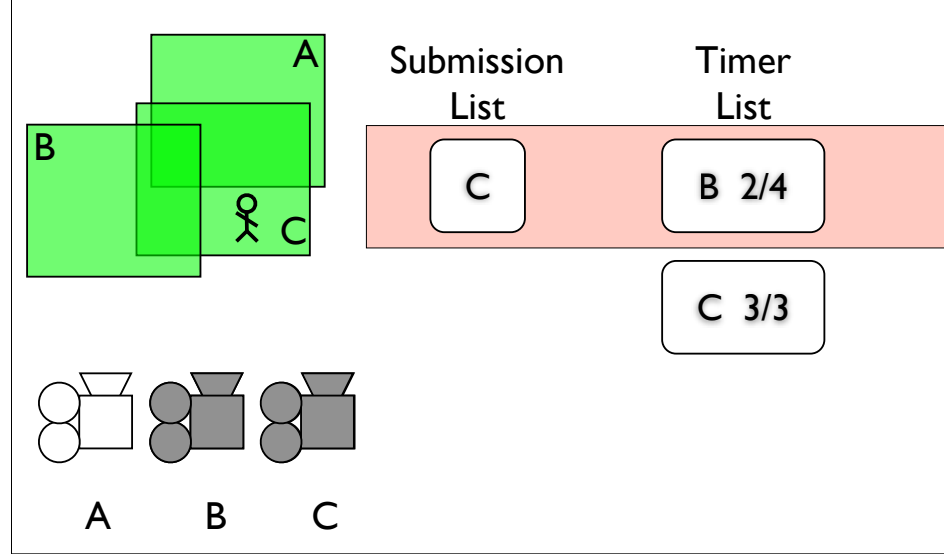
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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

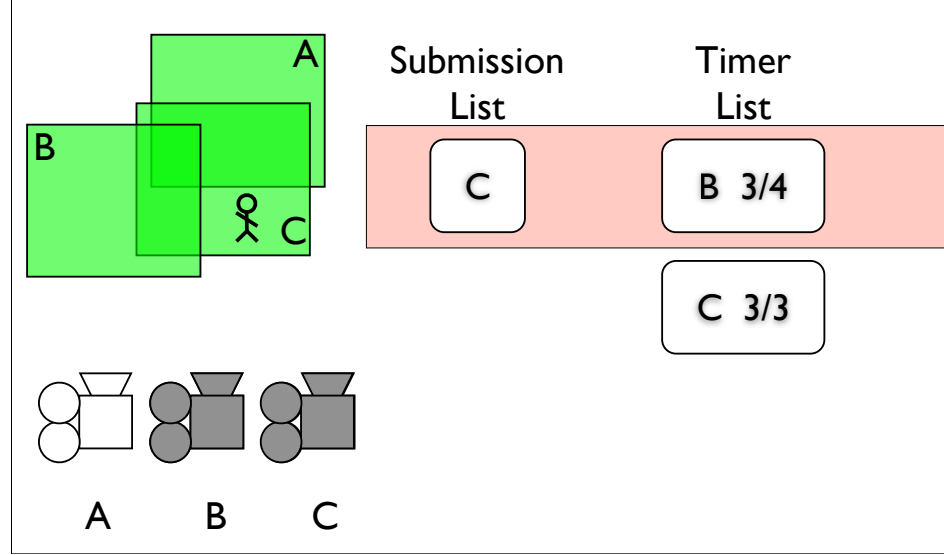
Timers : Implementation



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 b and c

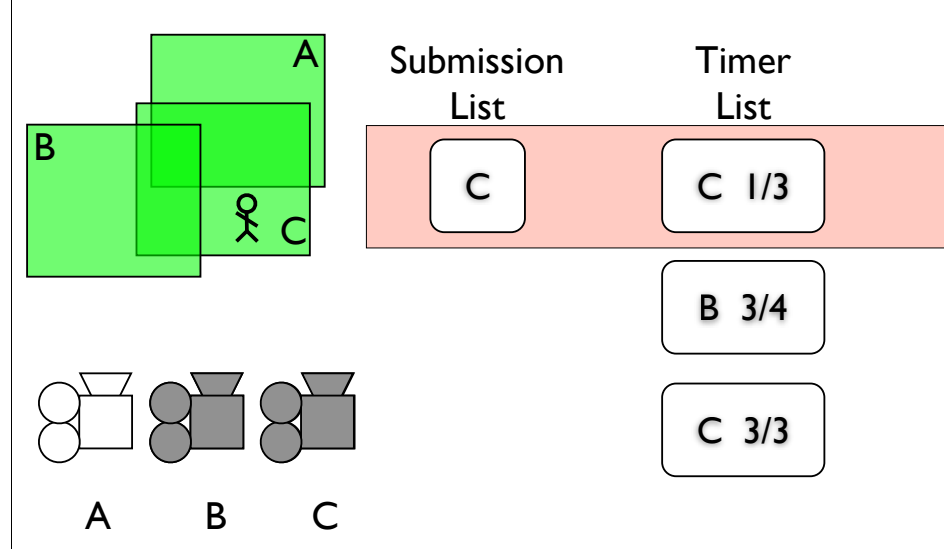
Timers : Implementation



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 b and c

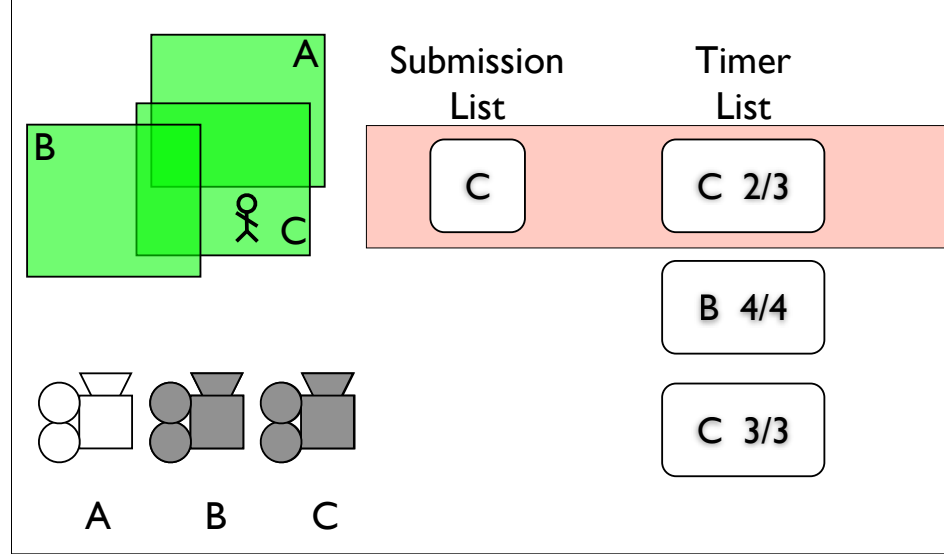
Timers : Implementation



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 b and c

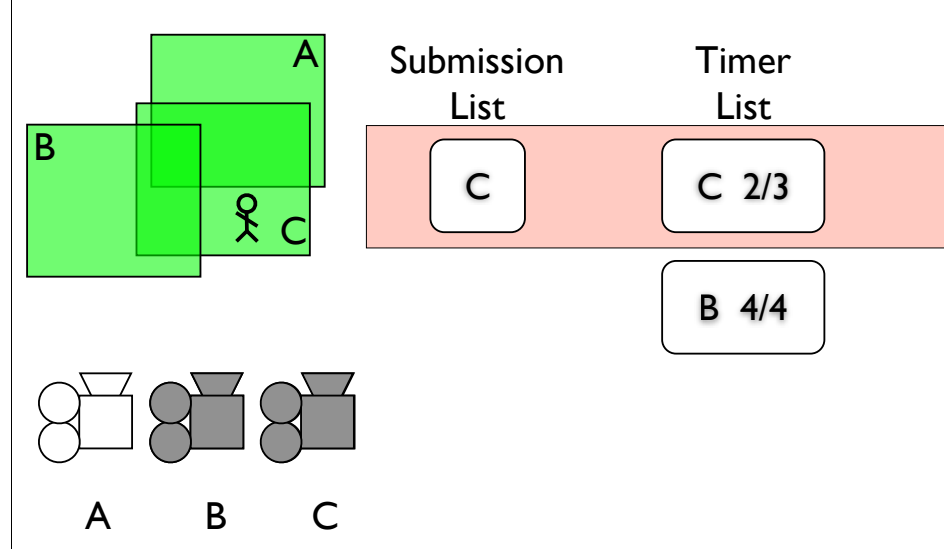
Timers : Implementation



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 b and c

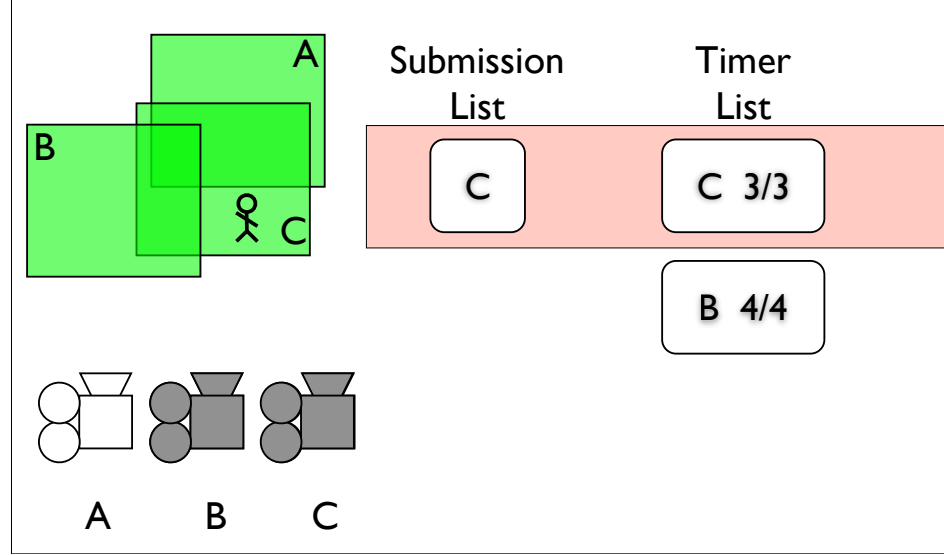
Timers : Implementation



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 b and c

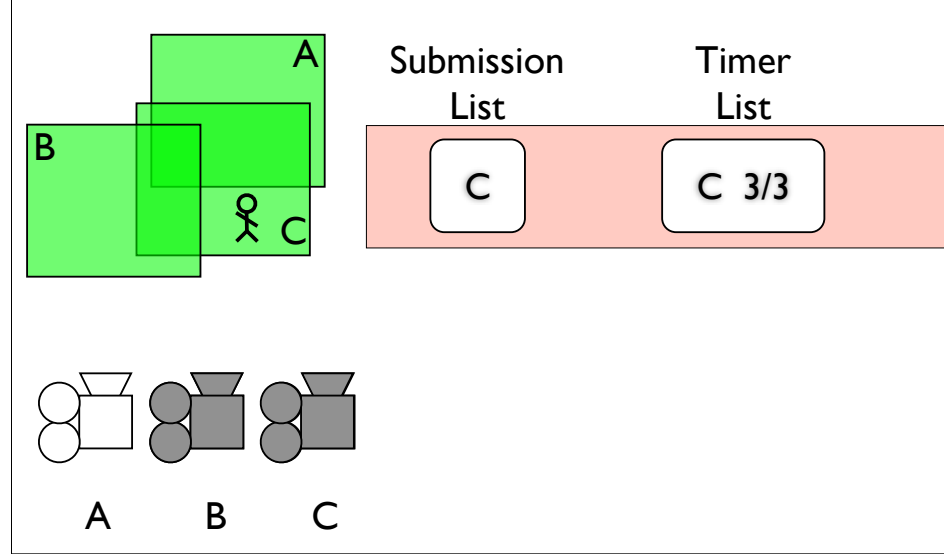
Timers : Implementation



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 b and c

Timers : Implementation



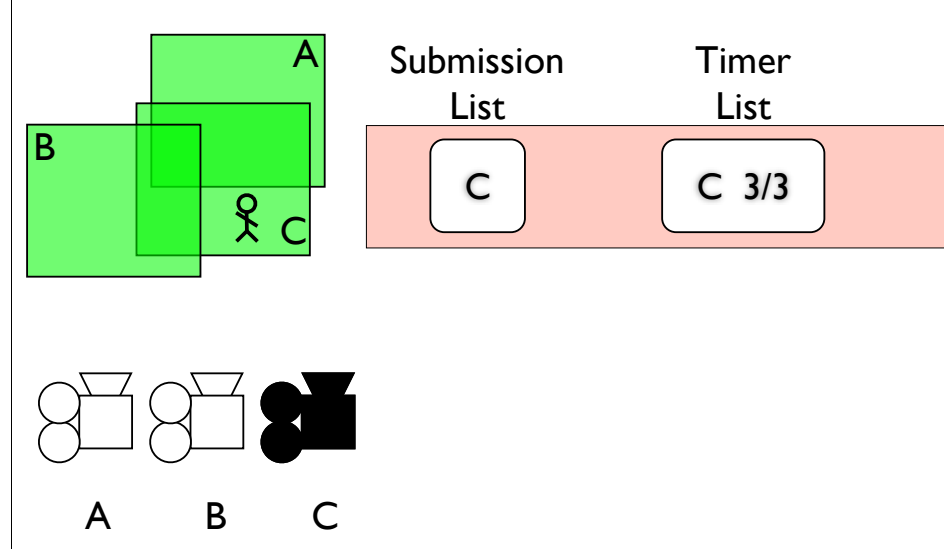
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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 b and c

Timers : Implementation



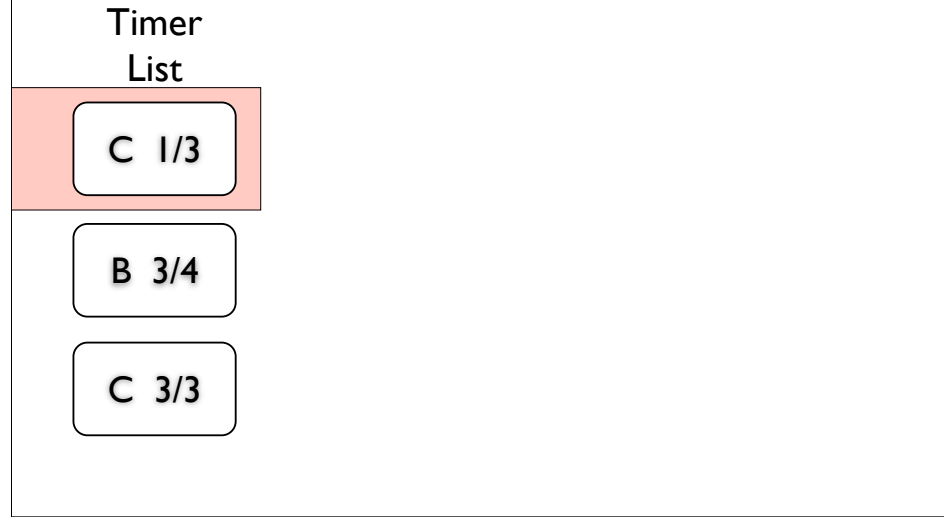
Note the red bar at the top of the lists, this is because the top of the submission list, and the top of the timer list should be the same camera. This is the active camera. The camera that we would cut to if we weren't fading it in.

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 b and c

Timers : Implementation

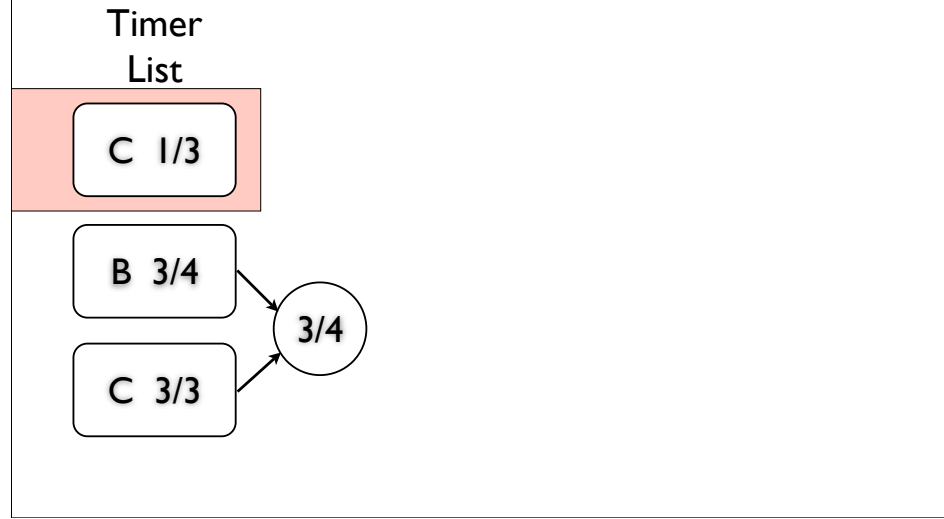


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. In 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 with 2 cameras, the first timer is 1/4, and the second

Timers : Implementation

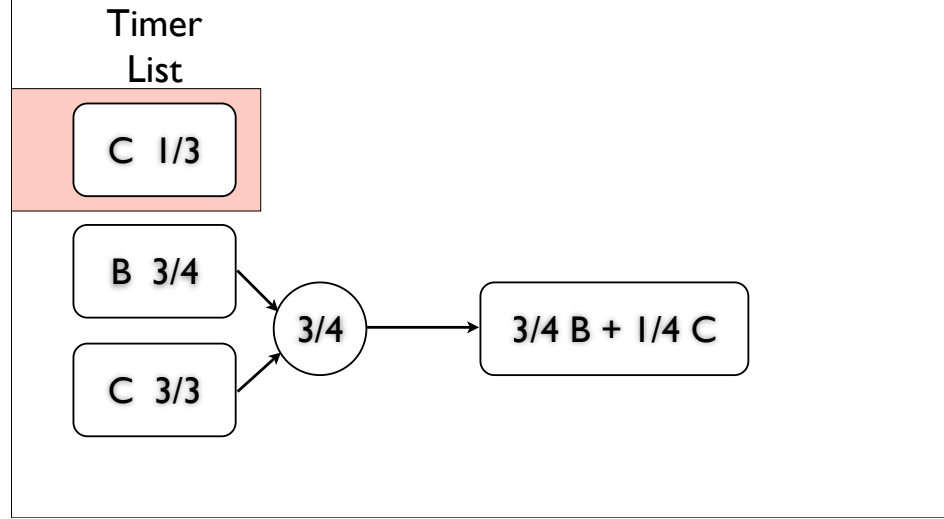


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. In 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 with 3/4 camera b and 1/4 camera c.

Timers : Implementation

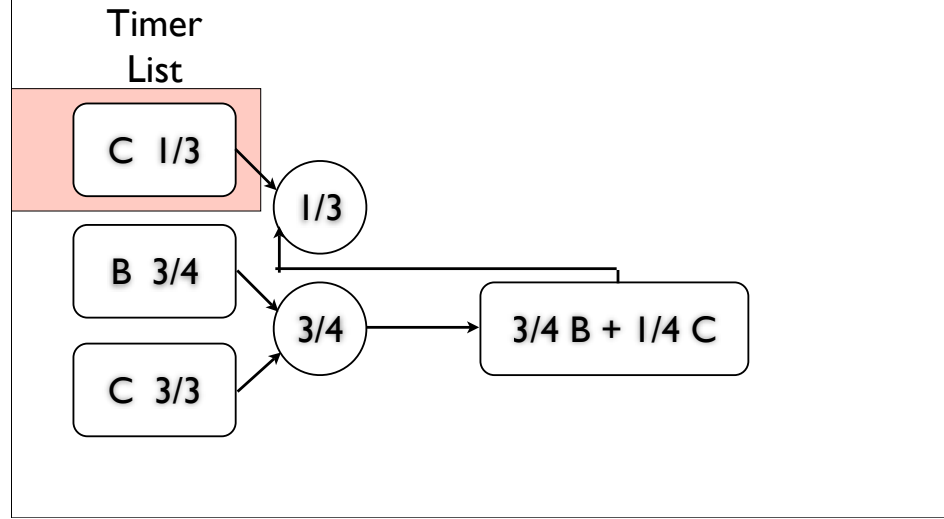


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. In 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 with 3/4 of camera b and 1/4 of camera c, that's

Timers : Implementation

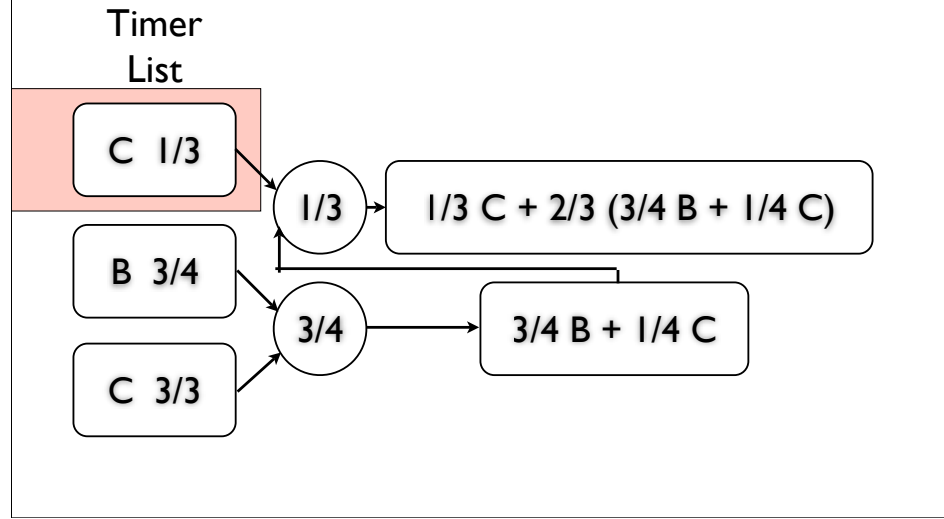


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. In 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 with 3/4 camera b and 1/4 camera c, the blend is $3/4 B + 1/4 C$.

Timers : Implementation

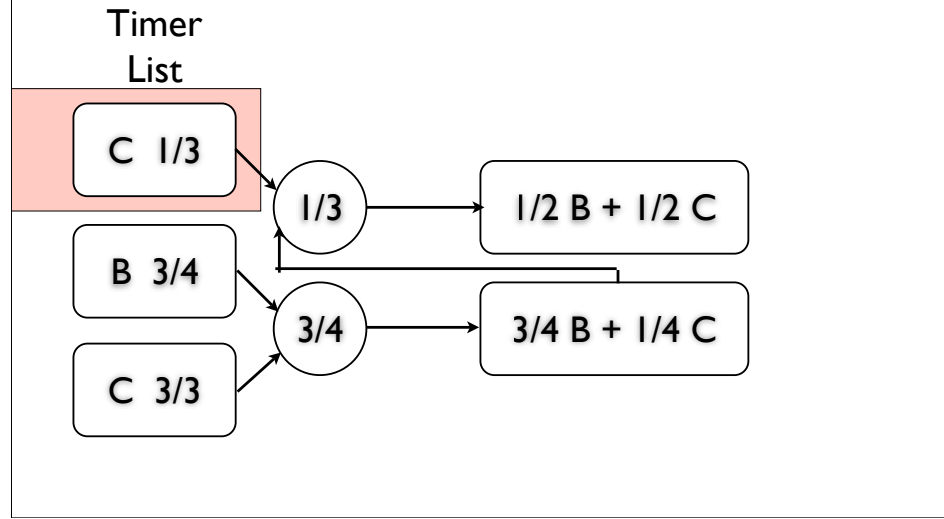


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. In 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

Timers : Implementation

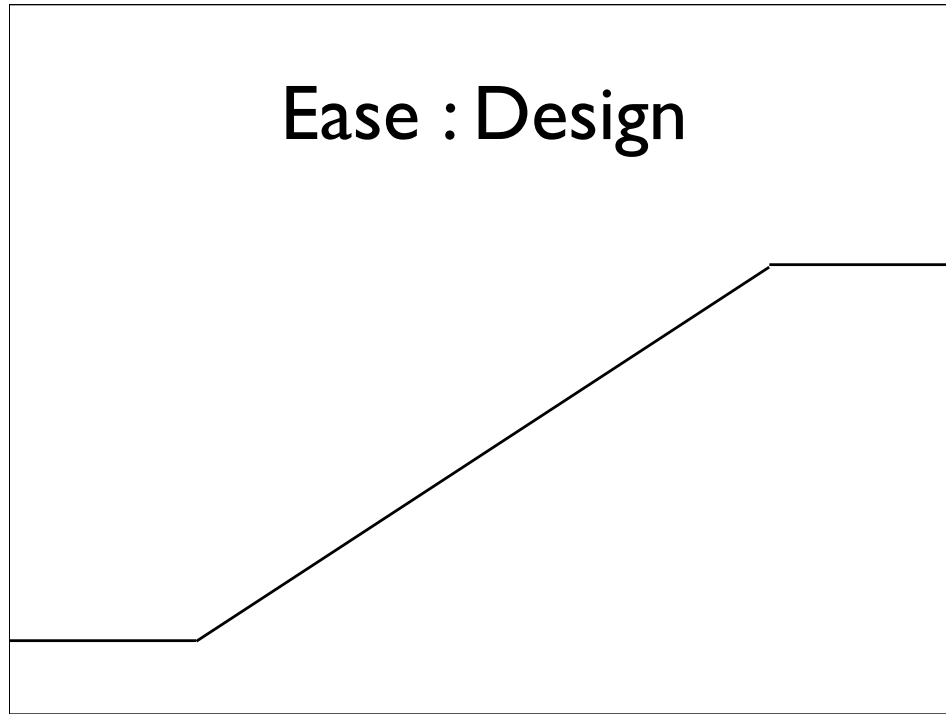


So that's how we maintain the timers, but how does this affect the actual blend of cameras we want?

Lets look at one of those blends in detail. In 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

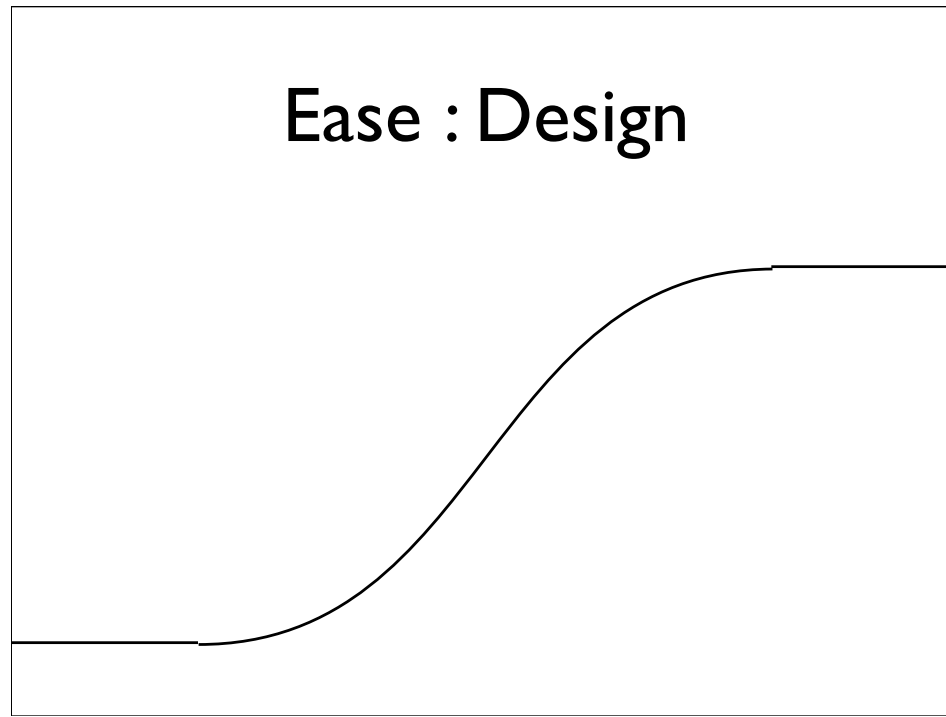
Ease : Design



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.



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

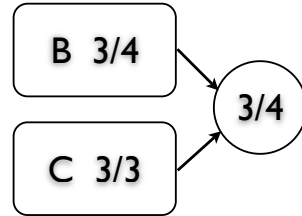
- Hermite Spline
- Fixed endpoints at 0 & 1
- Controllable tangents
- $\text{ease} = 1 - \text{tangent}$
- Ease in and out

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.

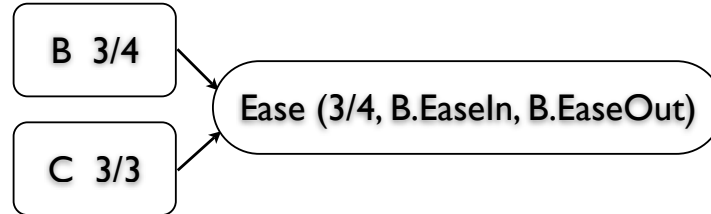
Ease : Implementation



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.

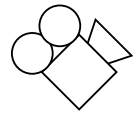
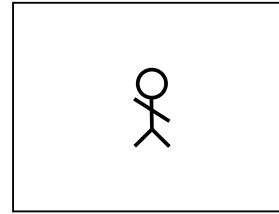
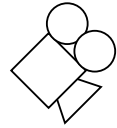
Ease : Implementation



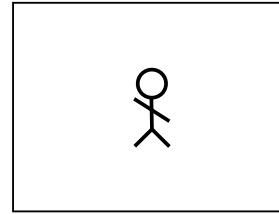
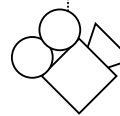
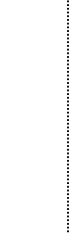
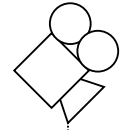
We apply ease when we calculate the blend factor between two cameras.

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.

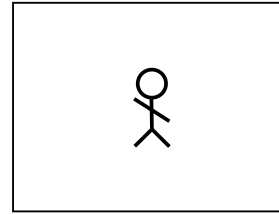
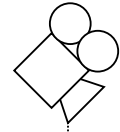
Blend Space : Design



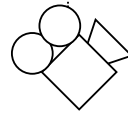
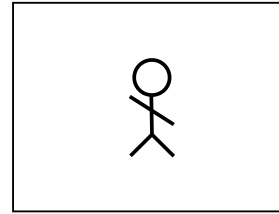
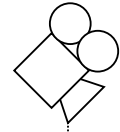
Blend Space : Design



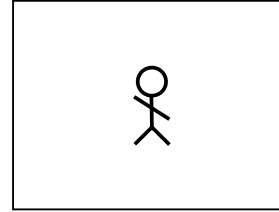
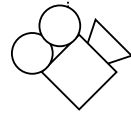
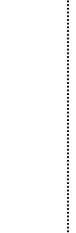
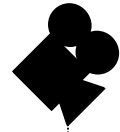
Blend Space : Design



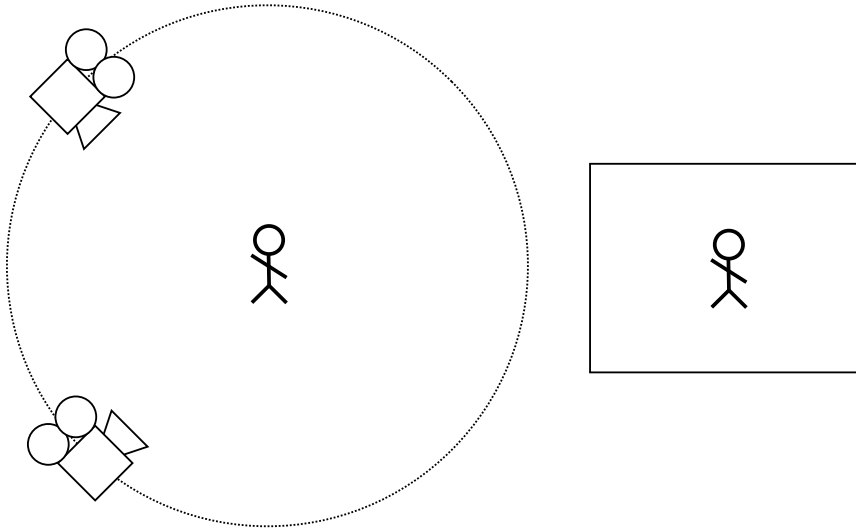
Blend Space : Design



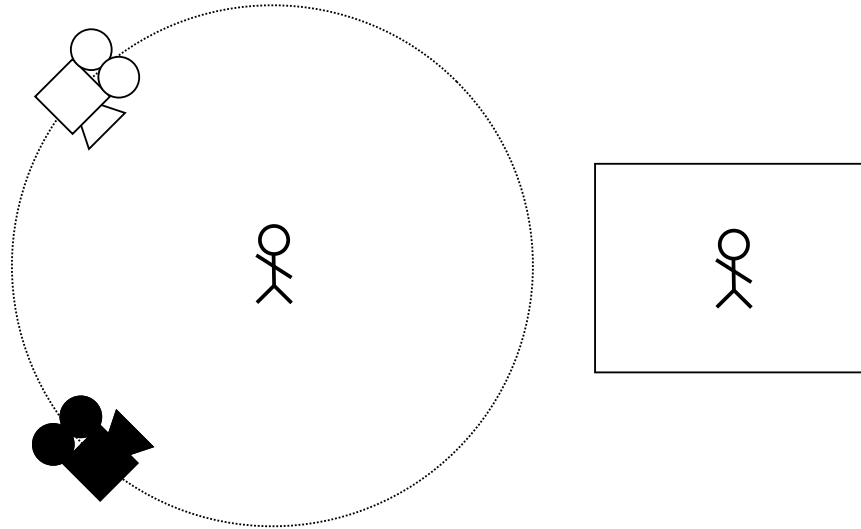
Blend Space : Design



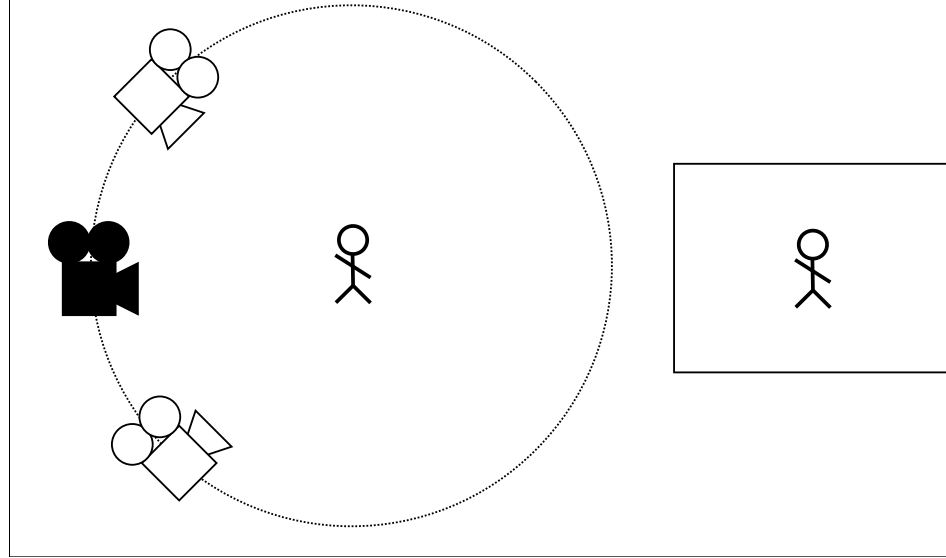
Blend Space : Design



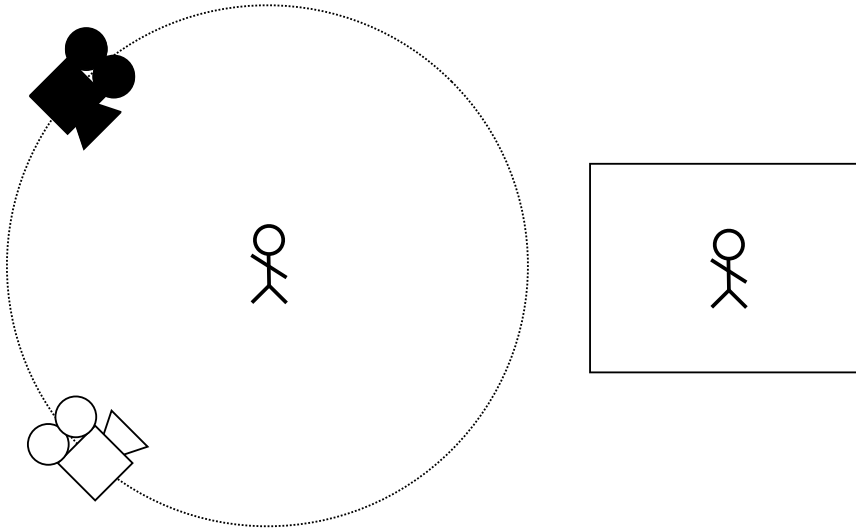
Blend Space : Design



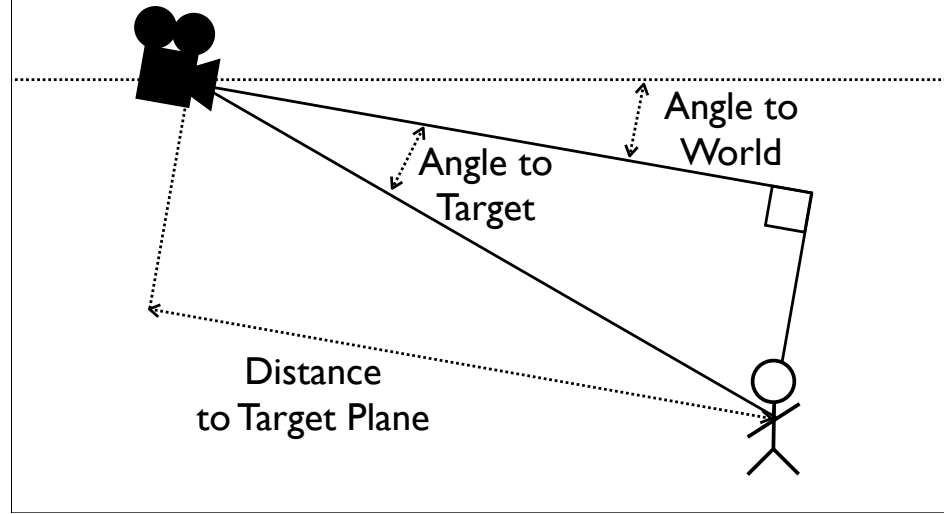
Blend Space : Design



Blend Space : Design



Blend Space : Implementation



In fact, what we do, is to blend the values we calculated during the dynamics phase. The position, size, and orientation of the player in screen space. Represented by the angle to the world, the angle to the target, and the distance to the target plane.

Overview

- Zoning
- Dynamics
- **Blending**
- Rails
- Fields

Overview

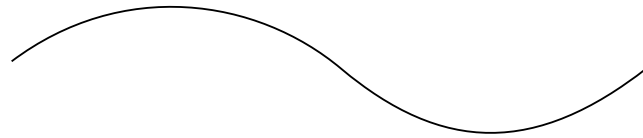
- Zoning
- Dynamics
- Blending
- **Rails**
- Fields

Rails : Objectives



So in order to deal with this, we 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.

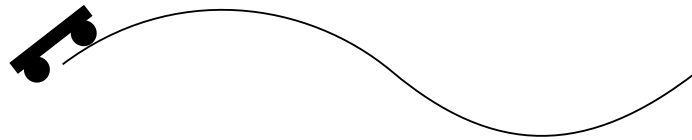
Rails : Design



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.

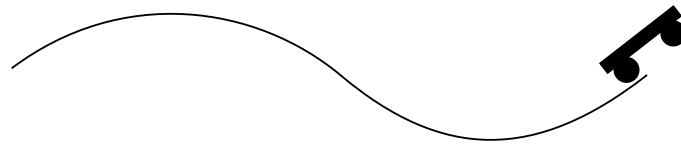
Rails : Design



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

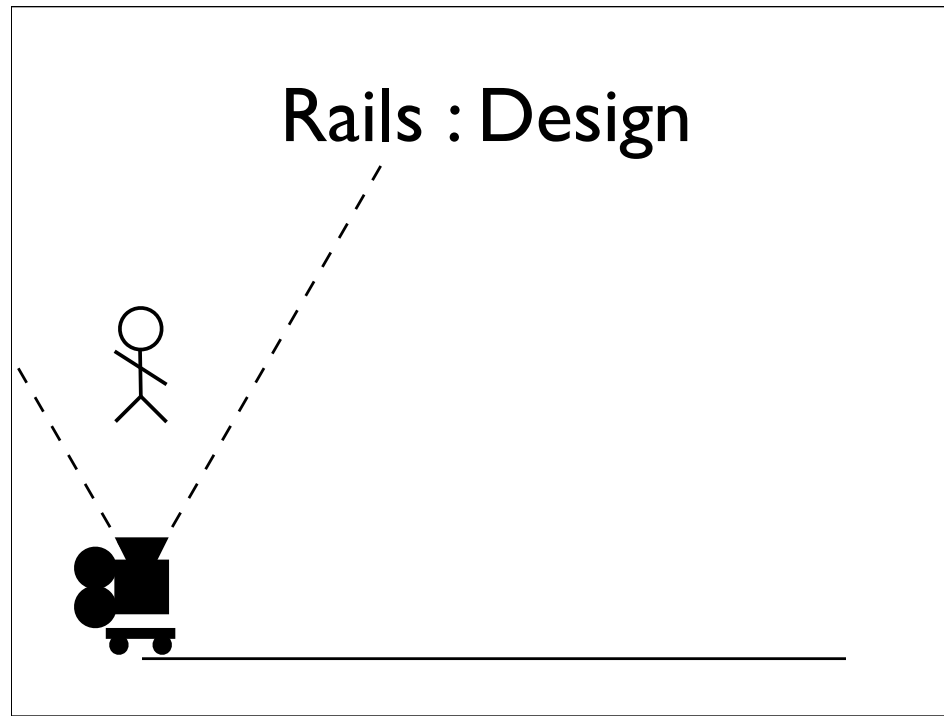
And the Dolly is a point on that spline, represented by the parametric value of the spline at that point.

Rails : Design



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

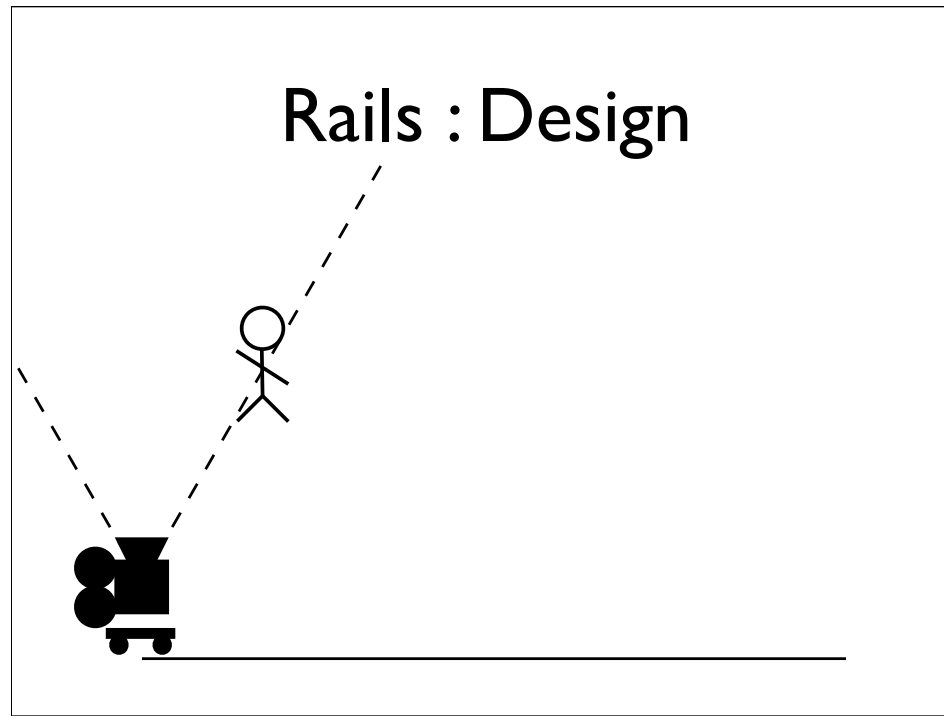
And the Dolly is a point on that spline, represented by the parametric value of the spline at that point.



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.

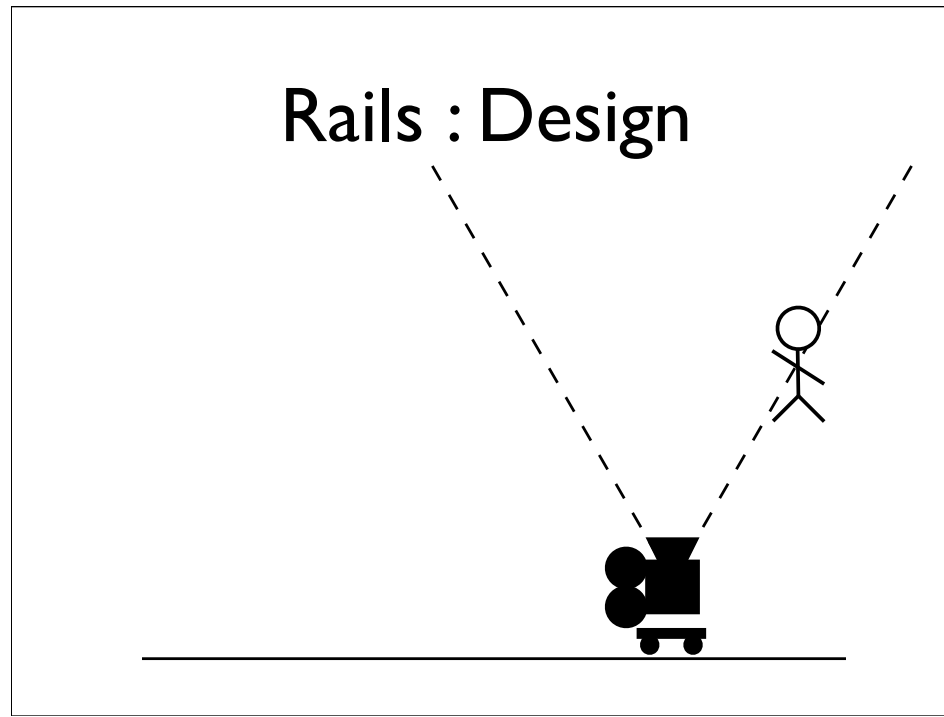
The player is free to move within the constraints, but when he tries to move outside them, the dolly moves to compensate as best it can.



So 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.

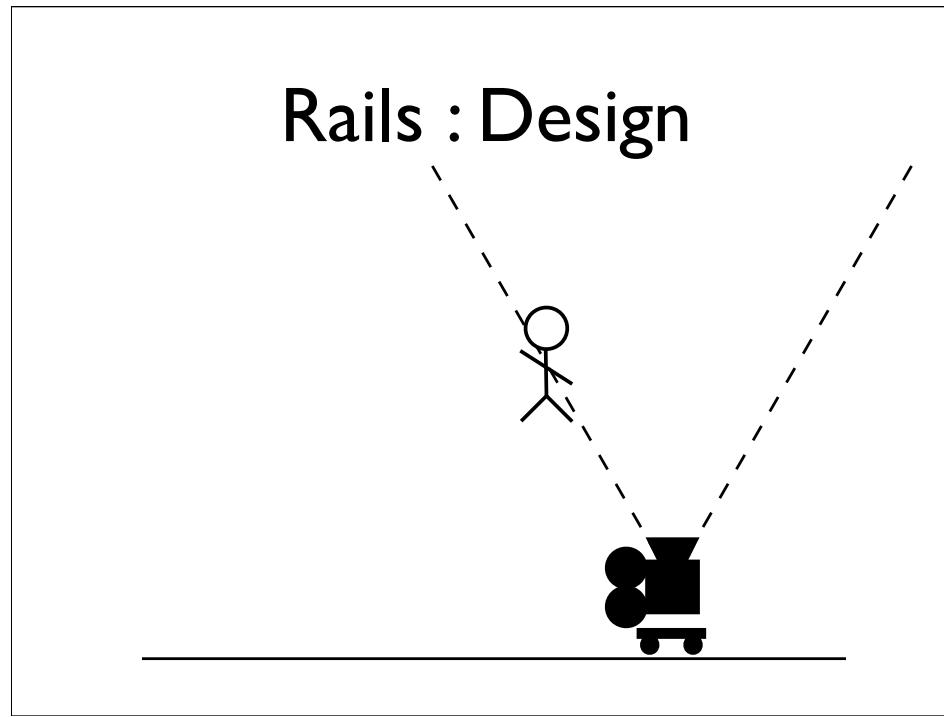
The player is free to move within the constraints, but when he tries to move outside them, the dolly moves to compensate as best it can.



So 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.

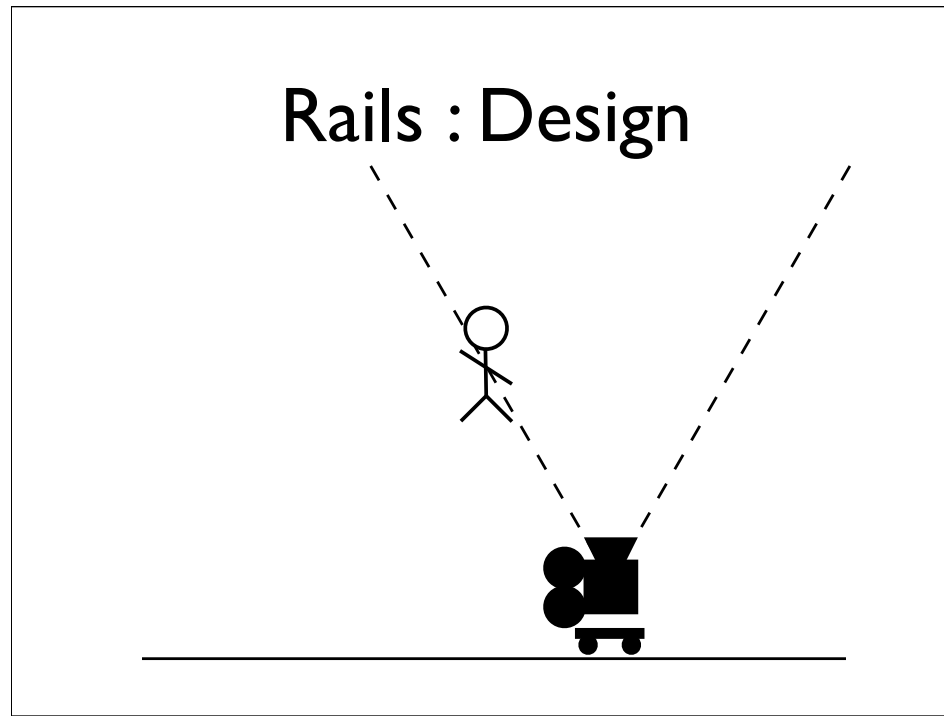
The player is free to move within the constraints, but when he tries to move outside them, the dolly moves to compensate as best it can.



So 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.

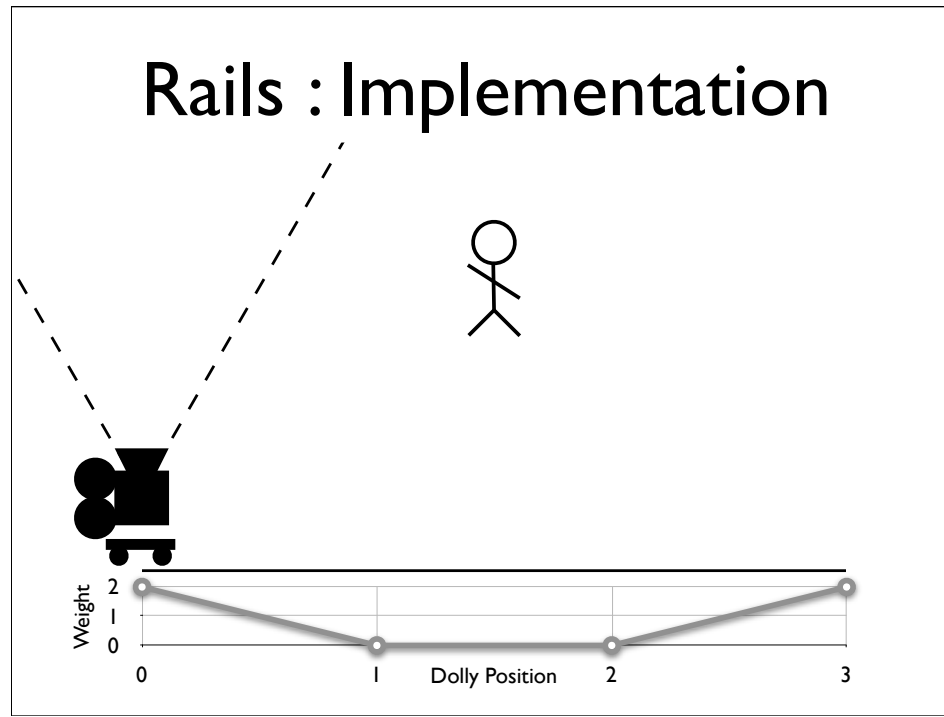
The player is free to move within the constraints, but when he tries to move outside them, the dolly moves to compensate as best it can.



So 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.

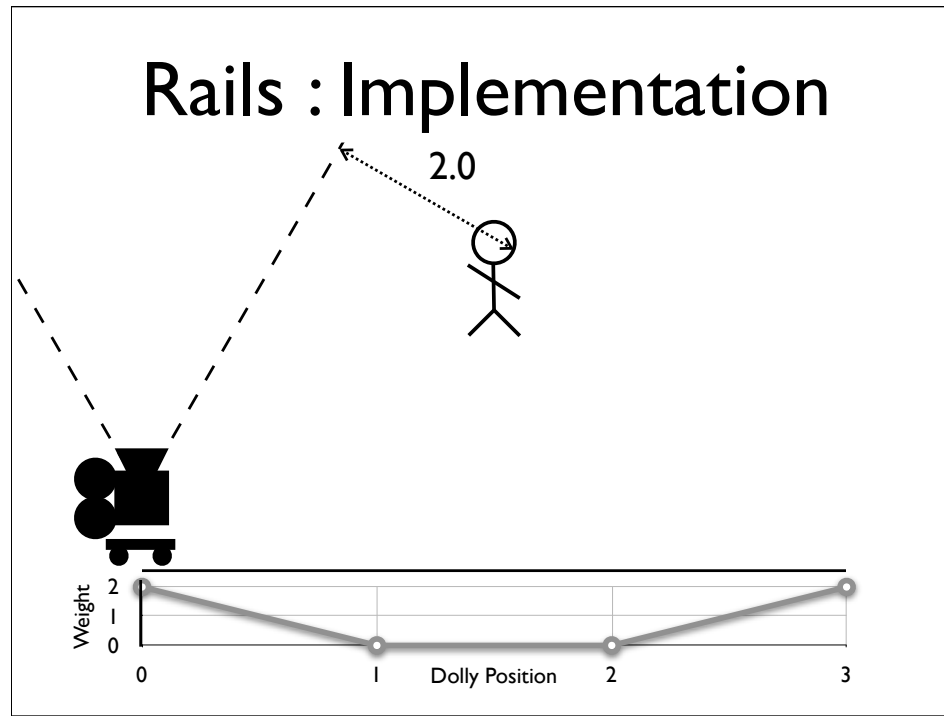


So how do we do that.

We'll 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 10 units outside of the constraint, so the value of the weighting function at this point, zero, is 10.

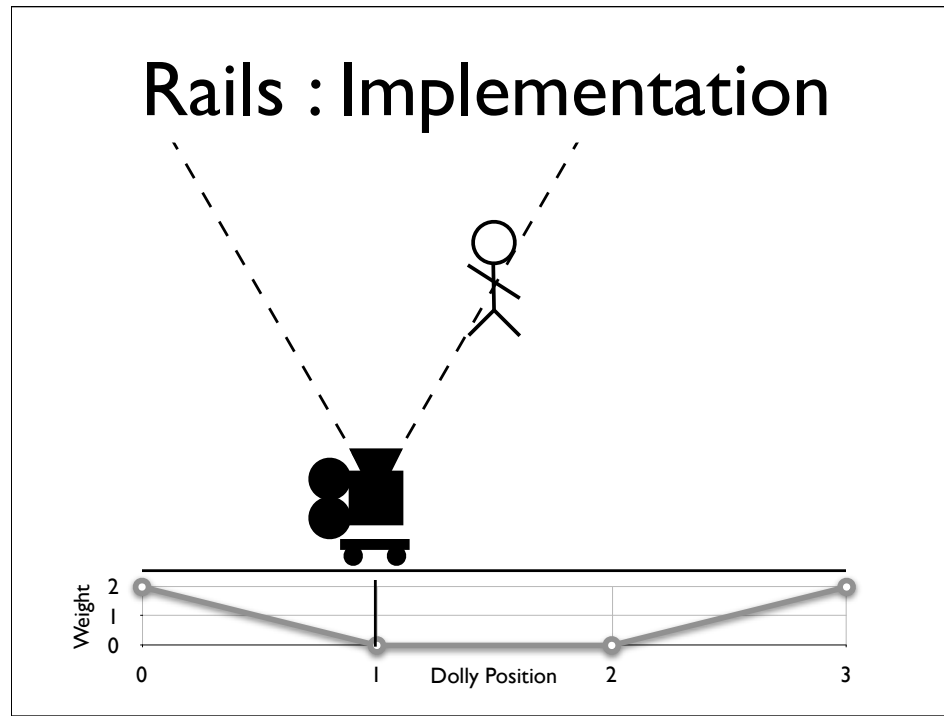


So how do we do that.

We'll 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 10 units outside of the constraint, so the value of the weighting function at this point, zero, is 10.

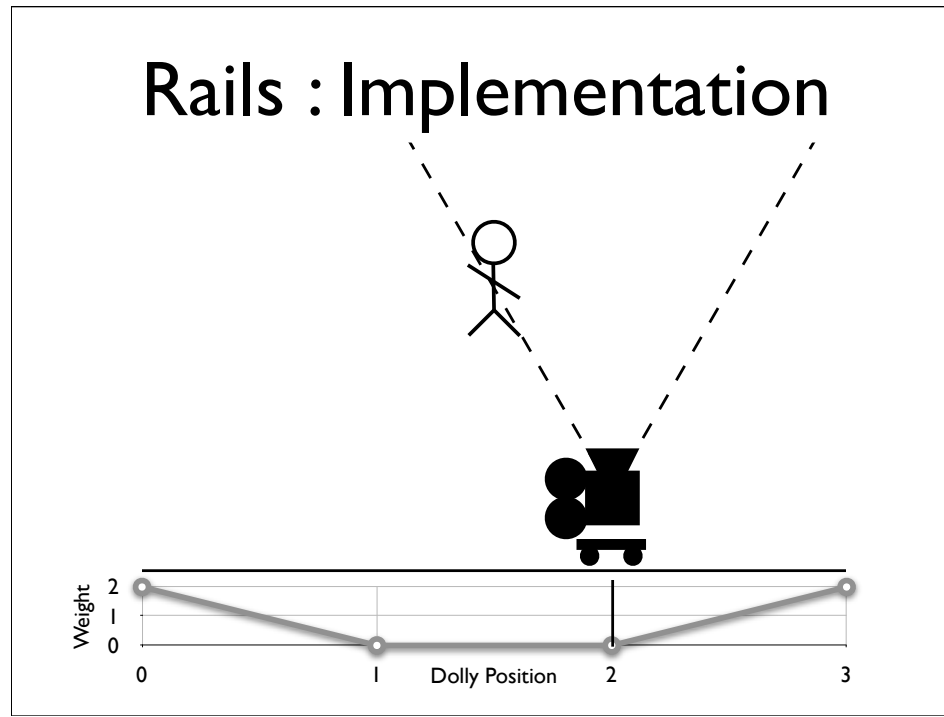


So how do we do that.

We'll 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 10 units outside of the constraint, so the value of the weighting function at this point, zero, is 10.



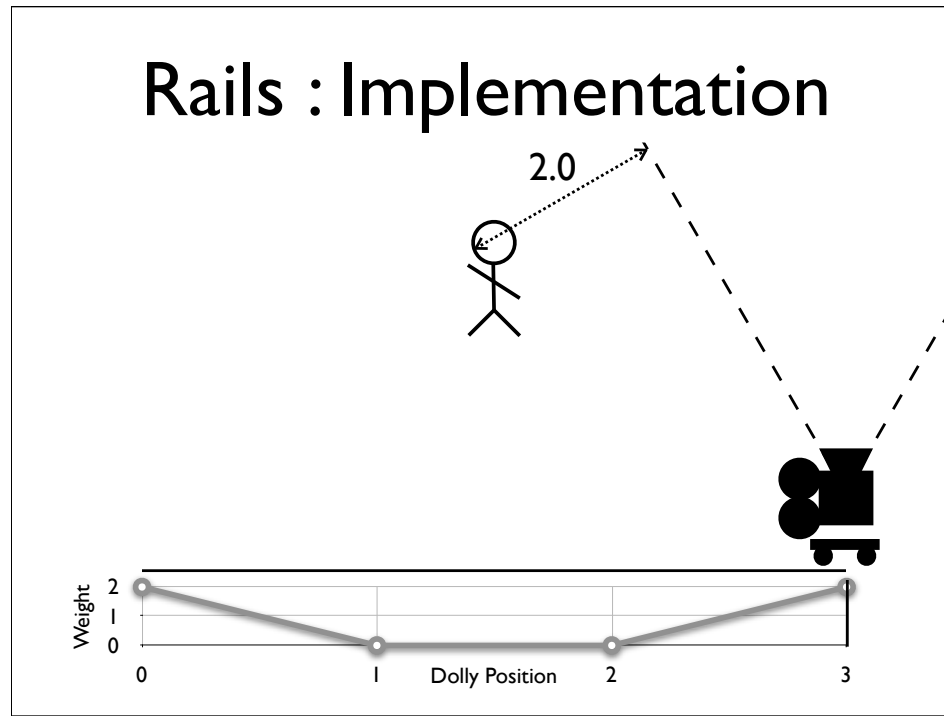
So how do we do that.

We'll 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 10 units outside of the constraint, so the value of the weighting function at this point, zero, is 10.

Rails : Implementation



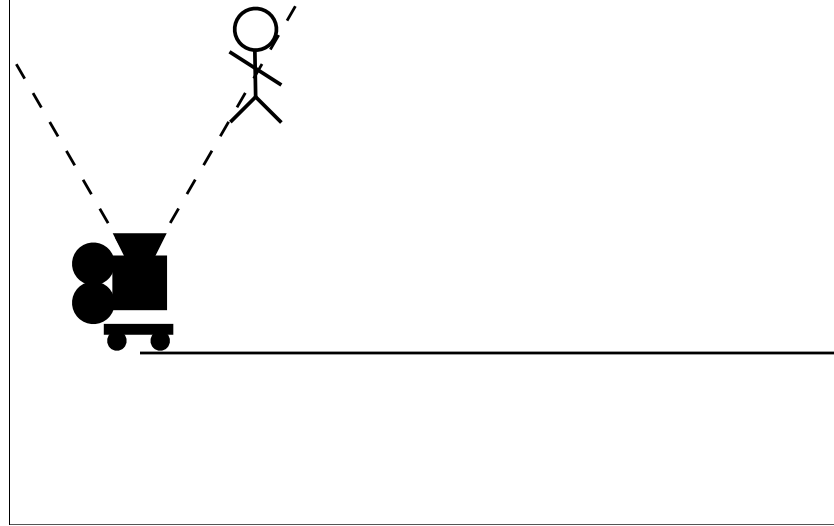
So how do we do that.

We'll 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 10 units outside of the constraint, so the value of the weighting function at this point, zero, is 10.

Rails : Implementation



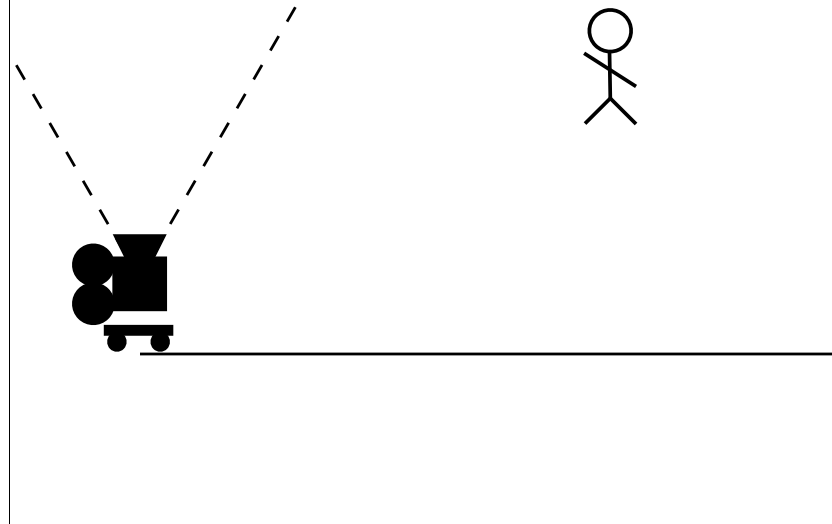
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



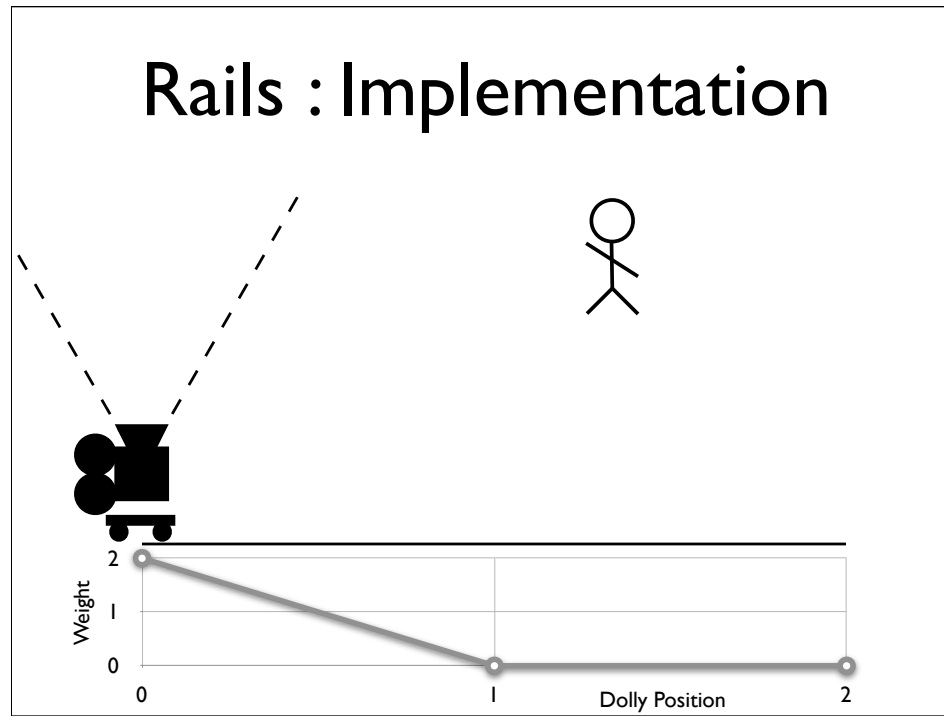
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



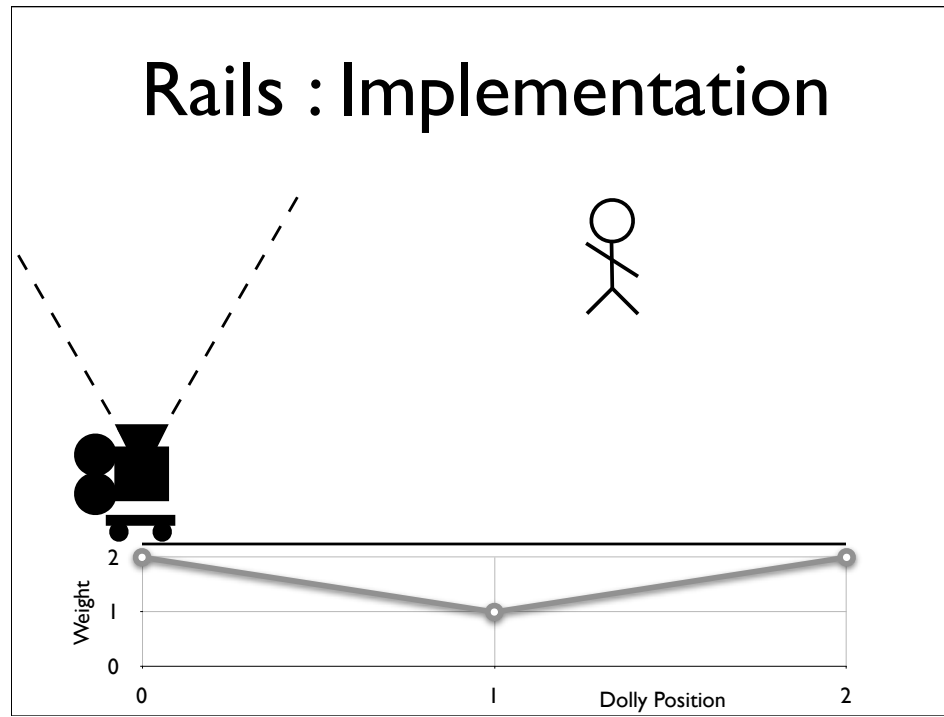
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



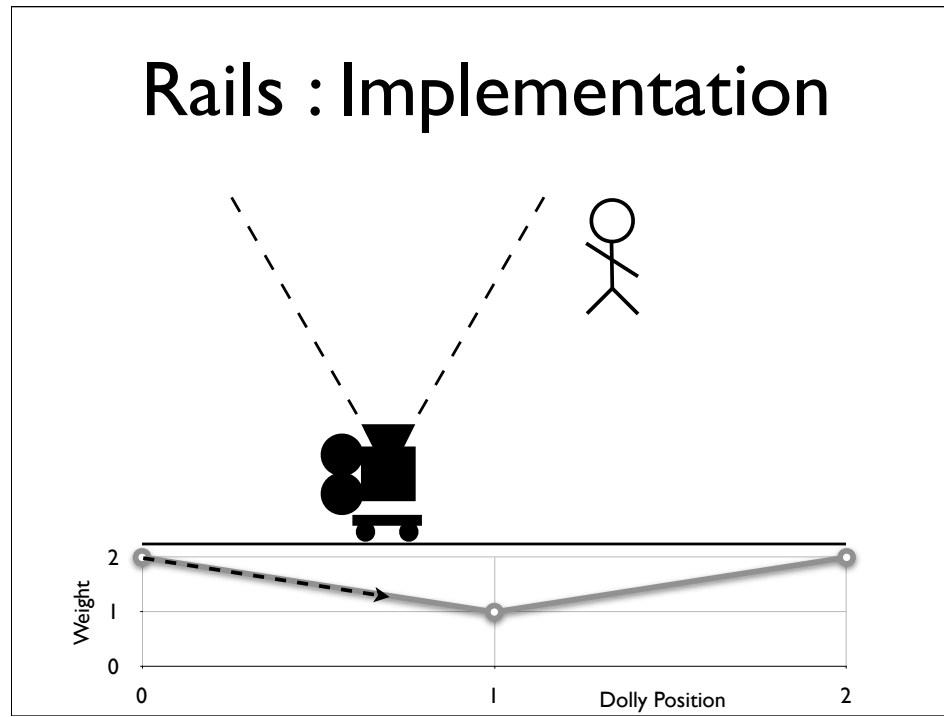
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



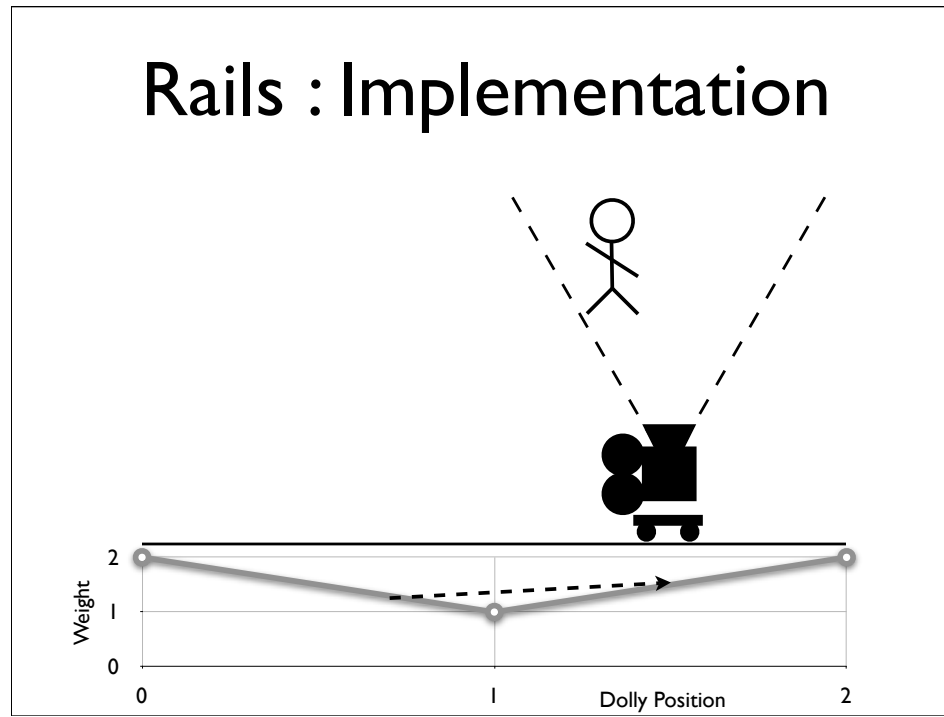
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



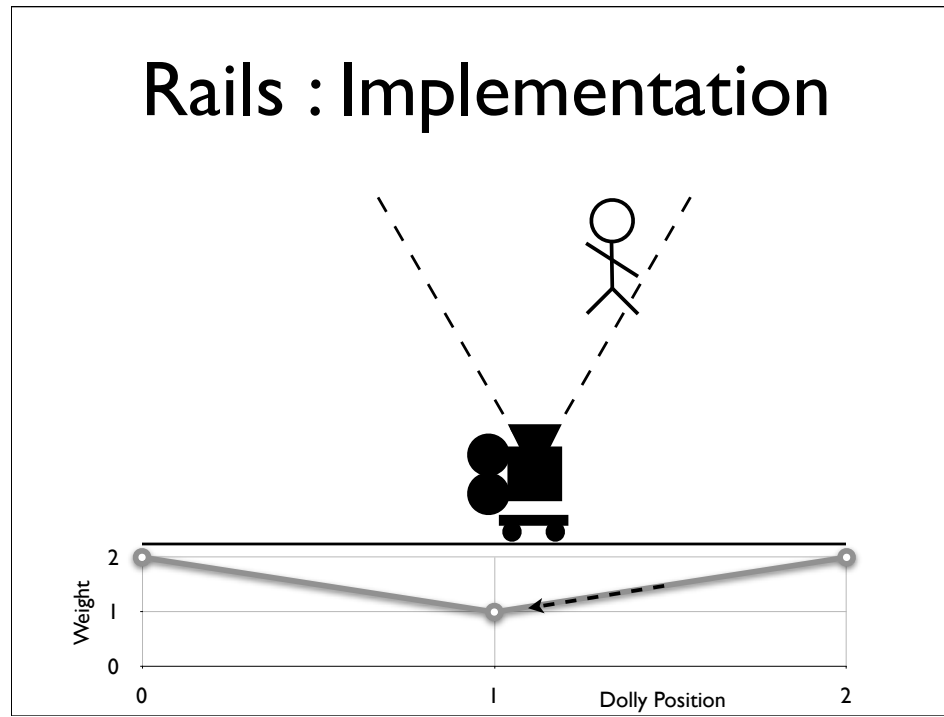
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



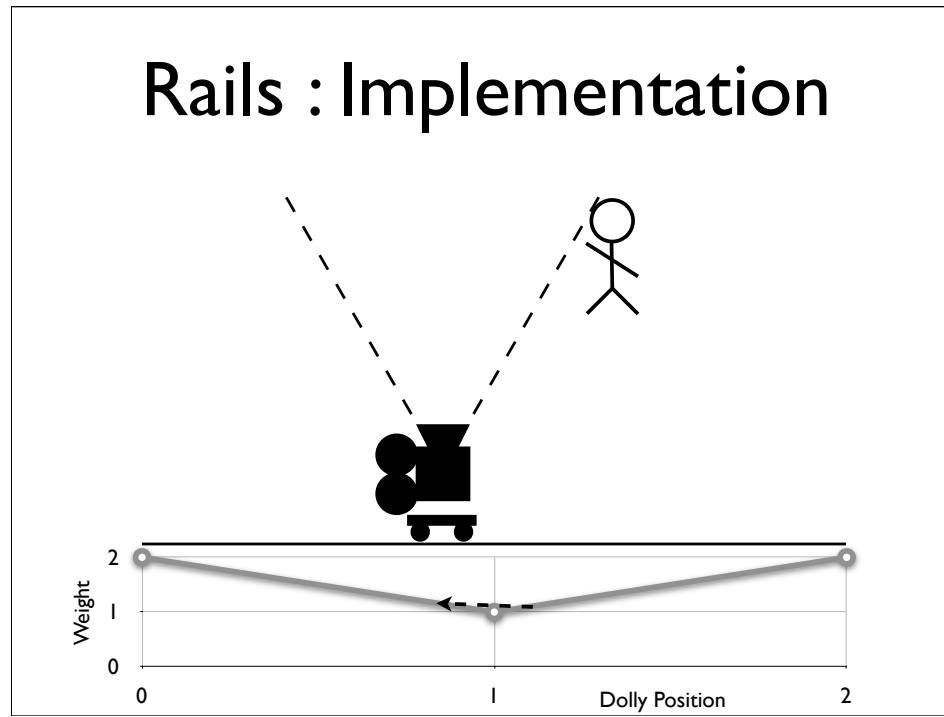
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



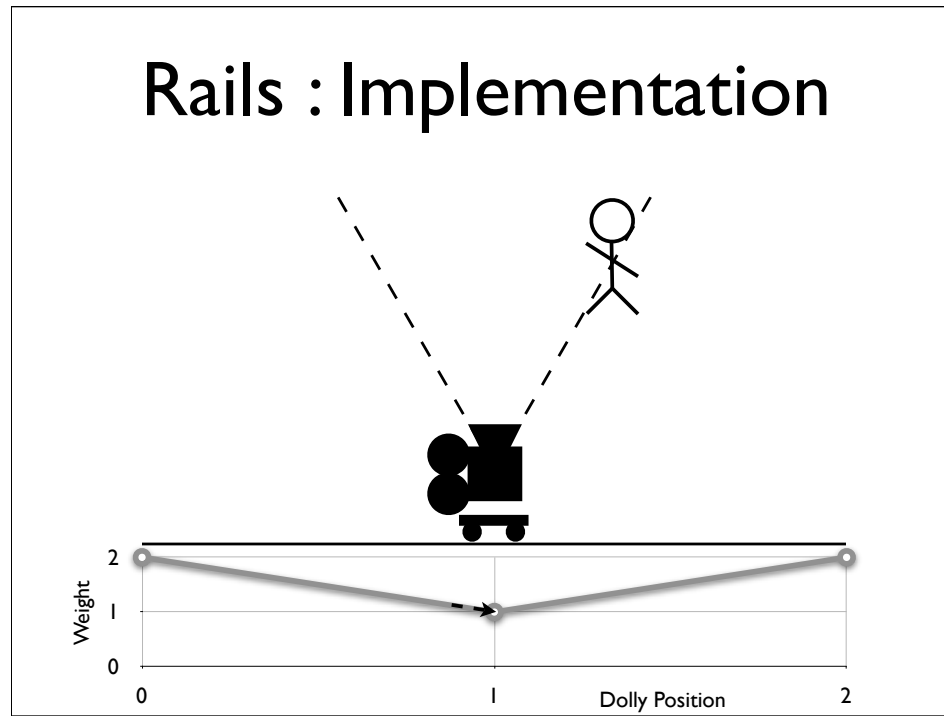
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



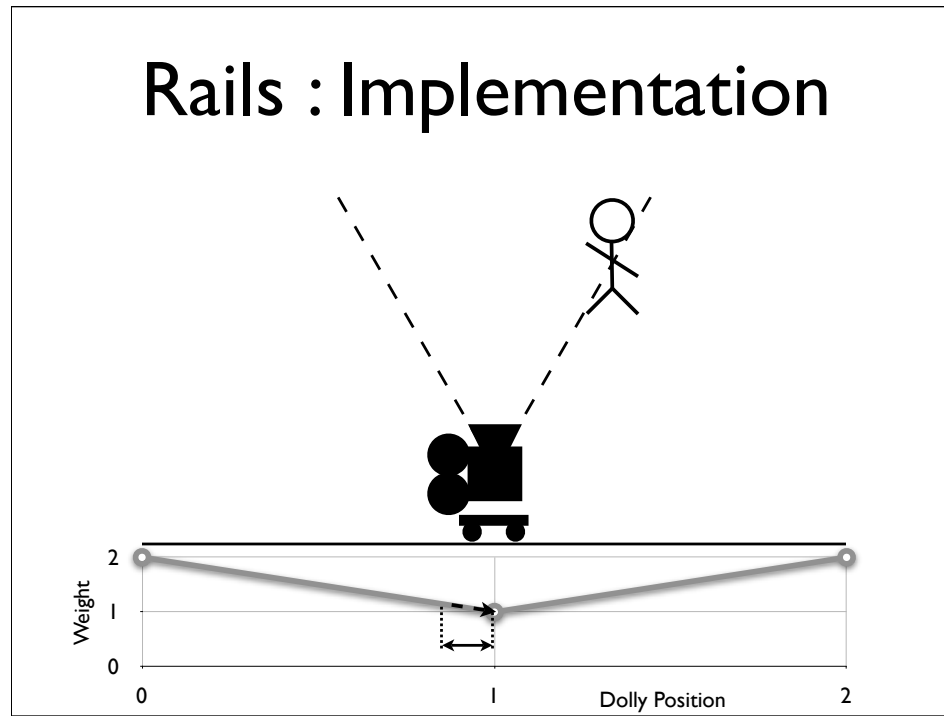
In this example, the player moves outside of the camera's constraints. Which gives us these weights.

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.

We take a guess as to which direction the player has moved, and

Rails : Implementation



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.

We take a guess as to which direction the player has moved, and

Rails : Implementation

1. Add distance from last frame to weight function
2. Guess a move based on previous frame
3. If weight decreases, keep going
4. If weight increases, slow down, turn around
5. Repeat 3 & 4 until moved less than threshold

So to summarise

we start by adding the distance from the previous frames dolly position to the weighting function
we then guess a first move, based on how far, and which direction we moved last frame
if the weight at the new position decreases, we keep going
otherwise we slow down and turn around
until we've moved less than our threshold value

Rails : Implementation

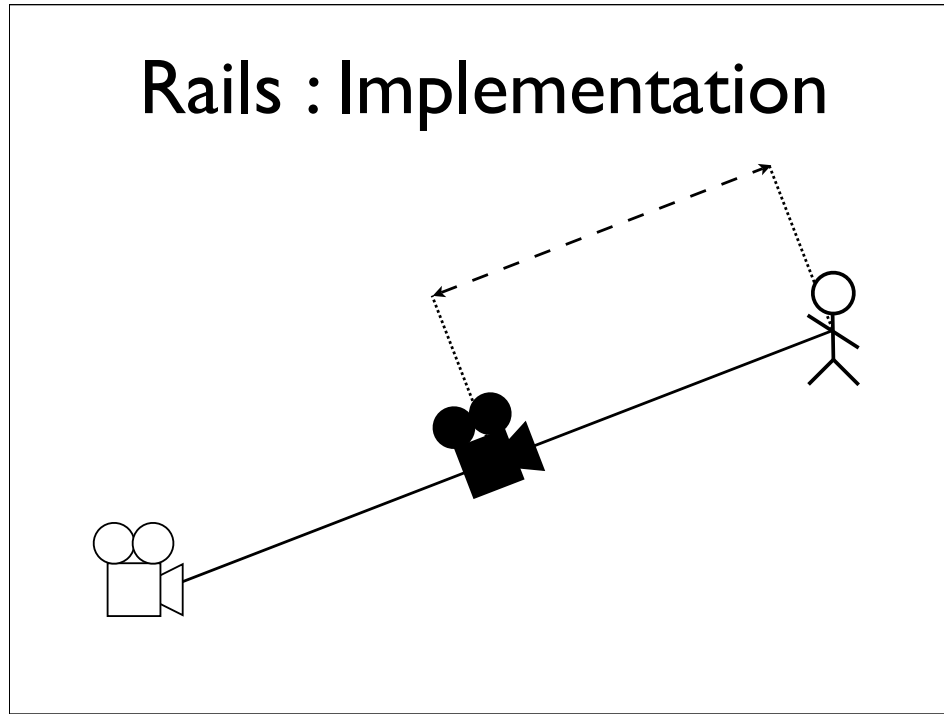
- Additional Weights
 - Distance from Player to Dolly
 - Angle from Tangent of Rail at Dolly
 - Amount Boss obscures Player
 - Number of minor characters out of frame

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.

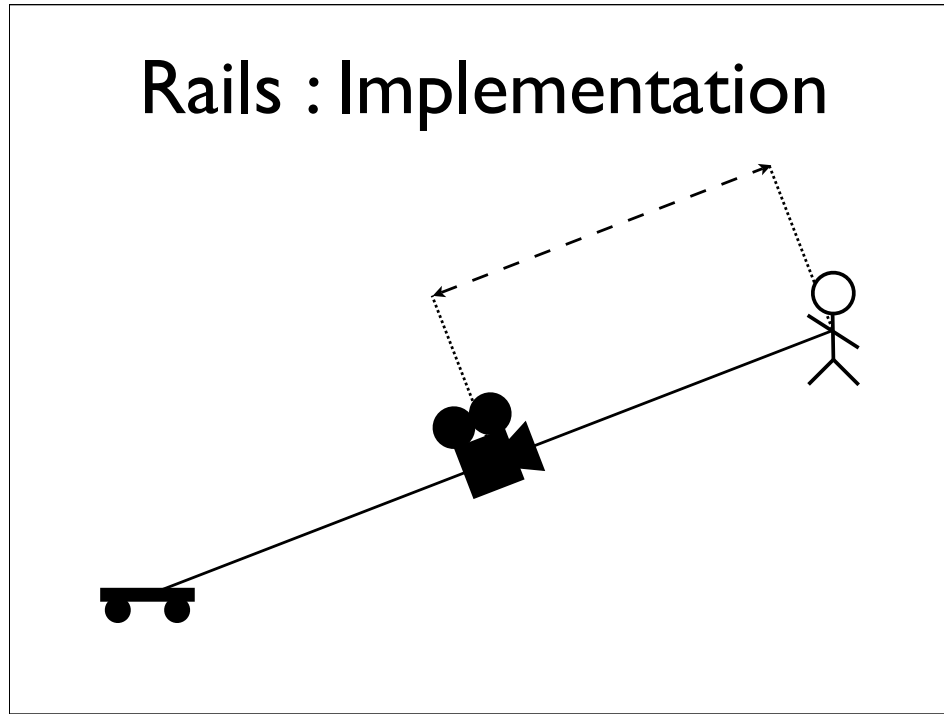
Rails : Implementation



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.

Rails : Implementation



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.

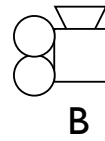
Overview

- Zoning
- Dynamics
- Blending
- **Rails**
- Fields

Overview

- Zoning
- Dynamics
- Blending
- Rails
- **Fields**

Fields : Design



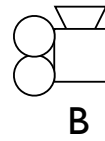
Blending with timers is all well and good, but sometime we wish we could control the progression of a blend, from the position of the player.

So if they stop halfway, the blend stops too.

and if they reverse direction, so does the blend

Note that the zone in the middle here, references two cameras, and has a direction.

Fields : Design



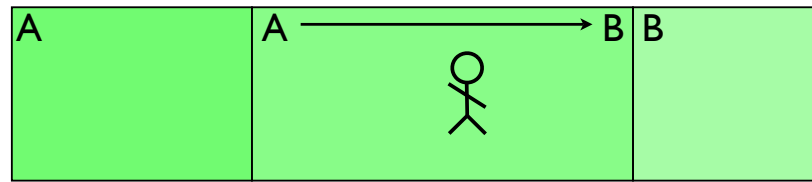
Blending with timers is all well and good, but sometime we wish we could control the progression of a blend, from the position of the player.

So if they stop halfway, the blend stops too.

and if they reverse direction, so does the blend

Note that the zone in the middle here, references two cameras, and has a direction.

Fields : Design



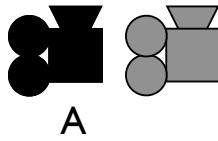
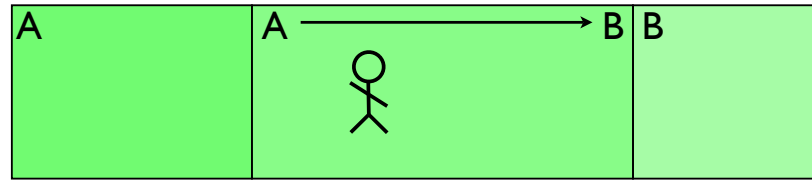
Blending with timers is all well and good, but sometime we wish we could control the progression of a blend, from the position of the player.

So if they stop halfway, the blend stops too.

and if they reverse direction, so does the blend

Note that the zone in the middle here, references two cameras, and has a direction.

Fields : Design



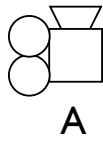
Blending with timers is all well and good, but sometime we wish we could control the progression of a blend, from the position of the player.

So if they stop halfway, the blend stops too.

and if they reverse direction, so does the blend

Note that the zone in the middle here, references two cameras, and has a direction.

Fields : Design



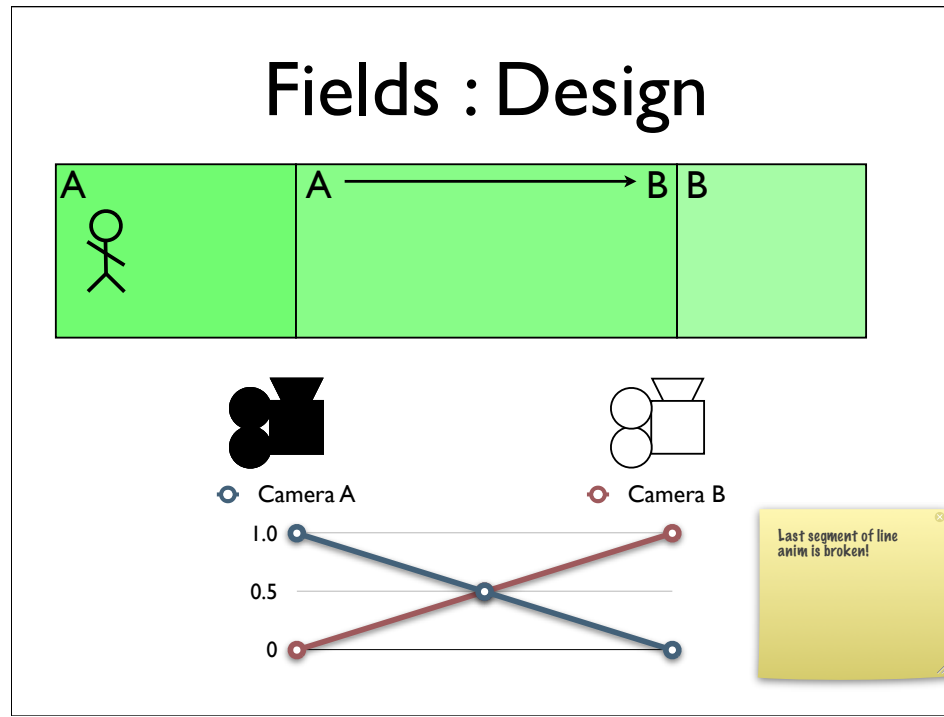
Blending with timers is all well and good, but sometime we wish we could control the progression of a blend, from the position of the player.

So if they stop halfway, the blend stops too.

and if they reverse direction, so does the blend

Note that the zone in the middle here, references two cameras, and has a direction.

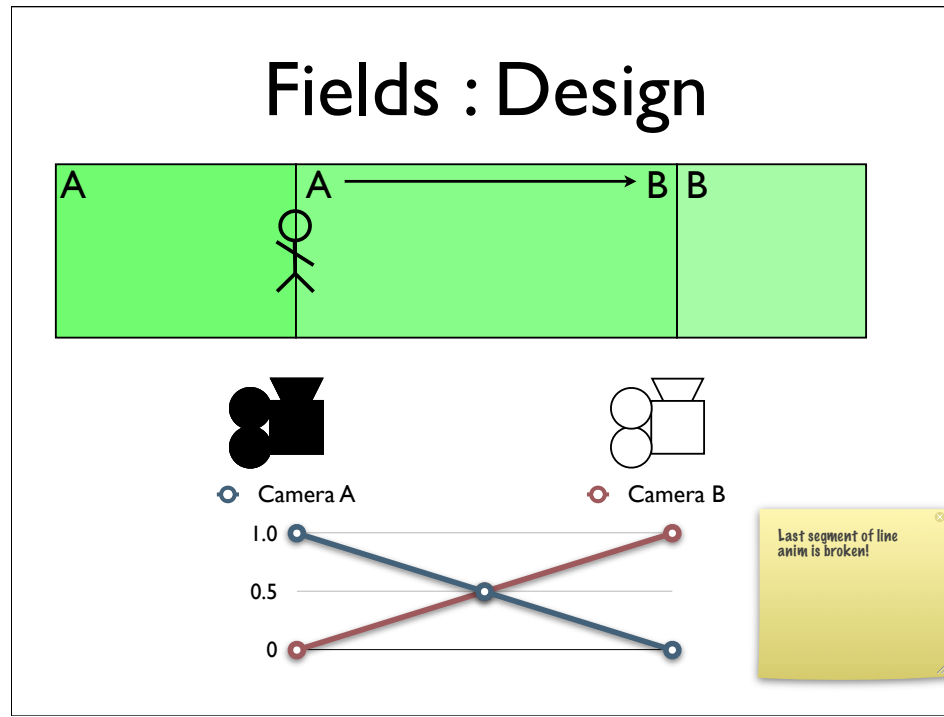
Fields : Design



In fact what we want is to be able to convert a position within a zone, into a value. A field value. In this definition, a field is just a value that is determined from a position. Like a magnetic field, has a particular strength at a particular position.

So as we cross this zone in the middle, we want the field value to derived from the position within the zone, and then to turn that field value, into a blend weight for the cameras in question.

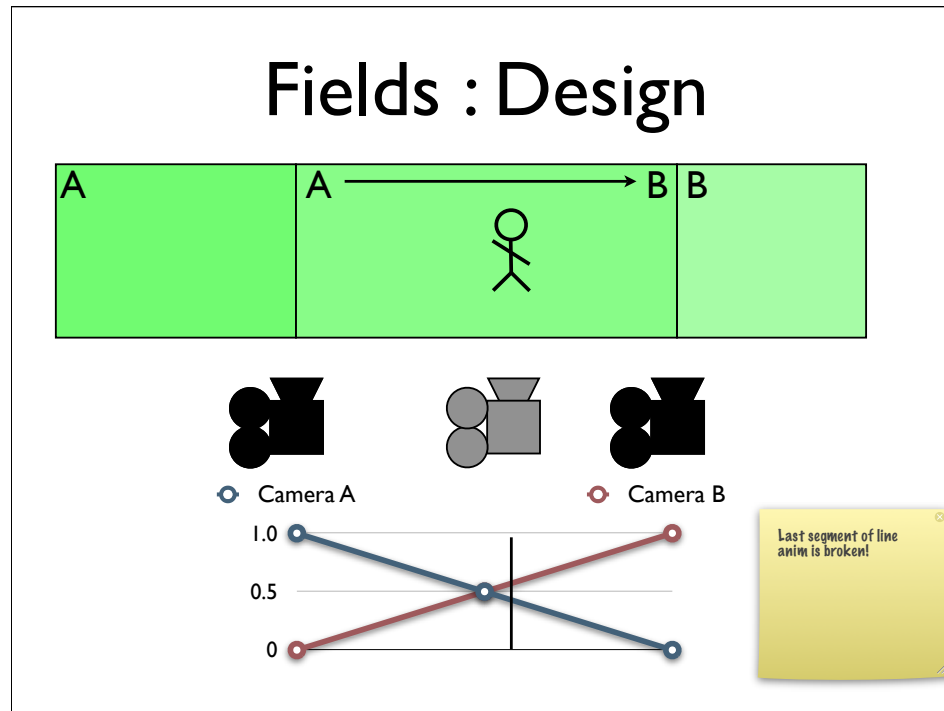
Fields : Design



In fact what we want is to be able to convert a position within a zone, into a value. A field value. In this definition, a field is just a value that is determined from a position. Like a magnetic field, has a particular strength at a particular position.

So as we cross this zone in the middle, we want the field value to derived from the position within the zone, and then to turn that field value, into a blend weight for the cameras in question.

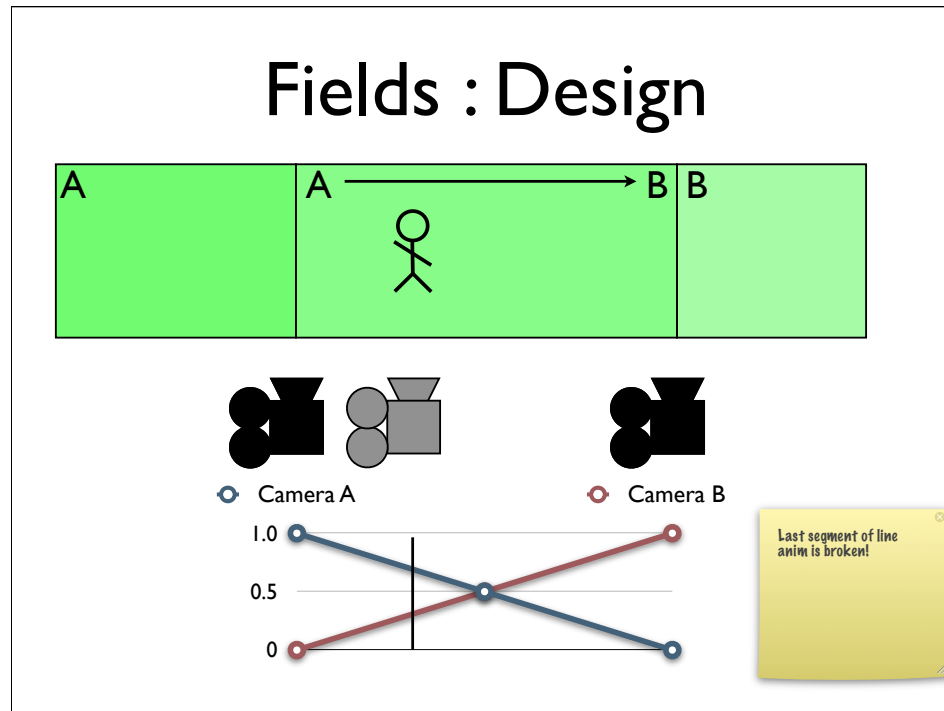
Fields : Design



In fact what we want is to be able to convert a position within a zone, into a value. A field value. In this definition, a field is just a value that is determined from a position. Like a magnetic field, has a particular strength at a particular position.

So as we cross this zone in the middle, we want the field value to derived from the position within the zone, and then to turn that field value, into a blend weight for the cameras in question.

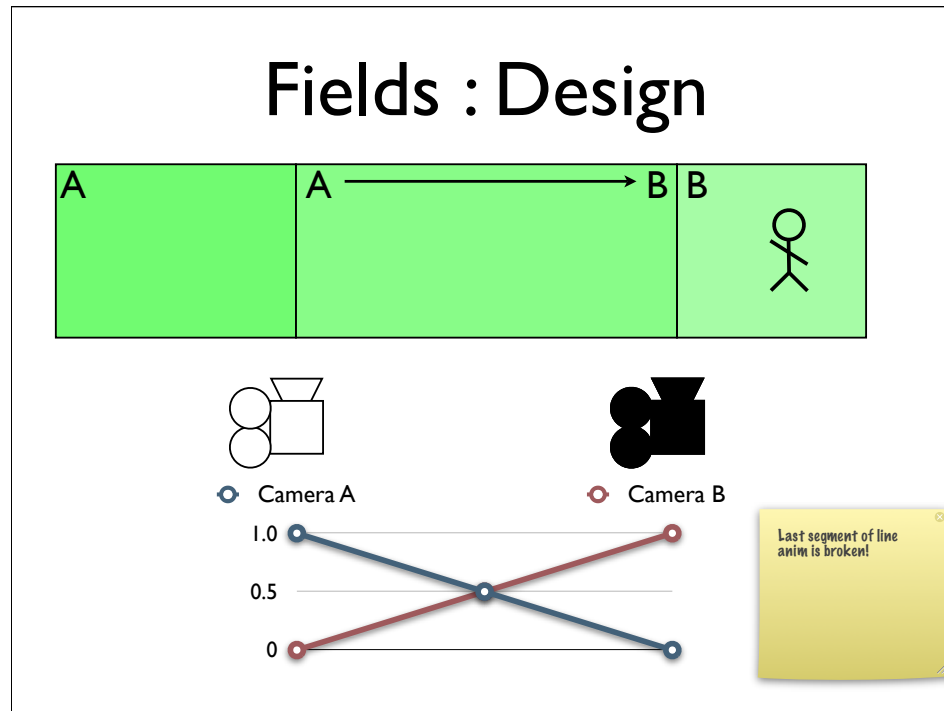
Fields : Design



In fact what we want is to be able to convert a position within a zone, into a value. A field value. In this definition, a field is just a value that is determined from a position. Like a magnetic field, has a particular strength at a particular position.

So as we cross this zone in the middle, we want the field value to derived from the position within the zone, and then to turn that field value, into a blend weight for the cameras in question.

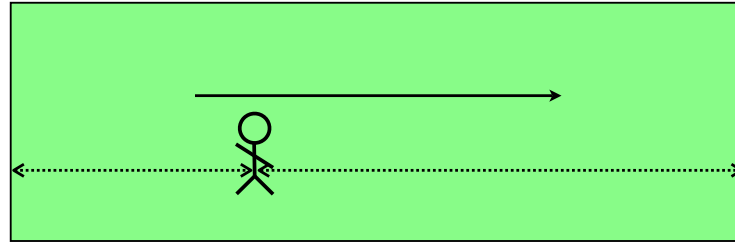
Fields : Design



In fact what we want is to be able to convert a position within a zone, into a value. A field value. In this definition, a field is just a value that is determined from a position. Like a magnetic field, has a particular strength at a particular position.

So as we cross this zone in the middle, we want the field value to derived from the position within the zone, and then to turn that field value, into a blend weight for the cameras in question.

Fields : Implementation



$$\frac{\text{Dot}(\text{plane}, \text{position})}{\text{Dot}(\text{plane}, \text{direction})}$$

plane (A,B,C,D) where
 $Ax+By+Cz+D = 0$
position (x,y,z,1) direction (x,y,z,0)

Well the first thing we need, is to turn the position within a zone, into a field value that we can use to weight the blend.

The direction of field is defined by a vector associated with the zone.

Because all our zones are convex, a line through the player, parallel to that vector, will intersect two planes of the zone.

calculate the distance from the player to the intersection point on

Fields : Implementation

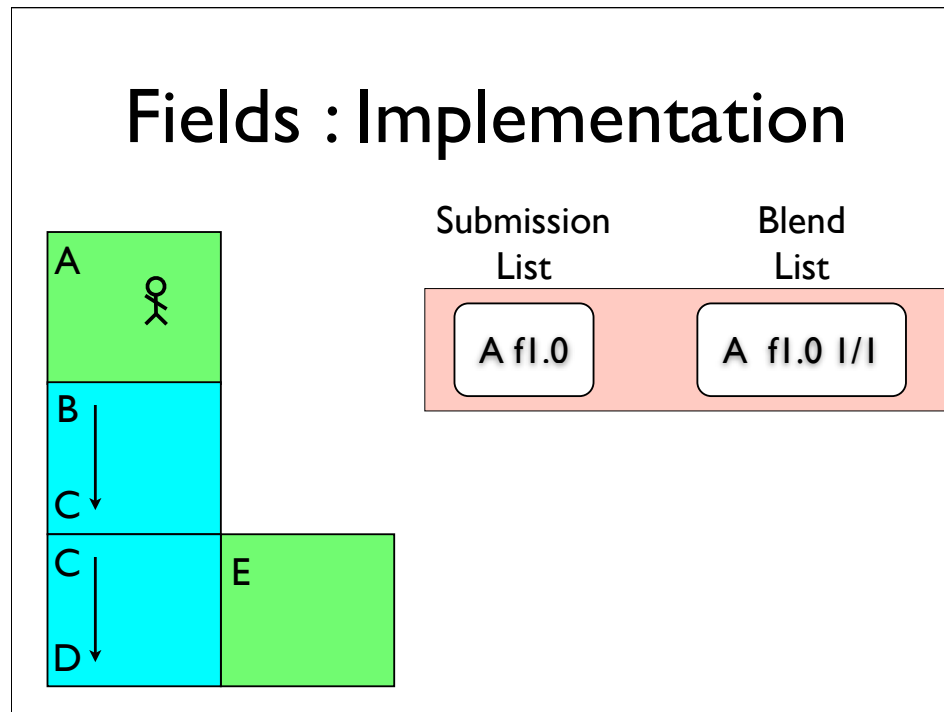
- Field values are returned from the query and copied into Submission List
- Timer list becomes a Blend list
- Blend list tracks Timers and Field values
- Field blended cameras are flagged and only blend with other field blended cameras at the same priority level
- Field values are frozen if all members of the group drop out of the query results

So having calculated a field value, it gets returned as part of the query, and copied into the entry in the submission list, and from there into any entries for that camera in the Timer list.

The timer list no longer just contains timers, it now contains field values as well, and so we rename it the Blend list, as now contains a list of blending primitives.

I should note here that field blended cameras only blend with other field blended cameras. The group of field blended cameras will be...

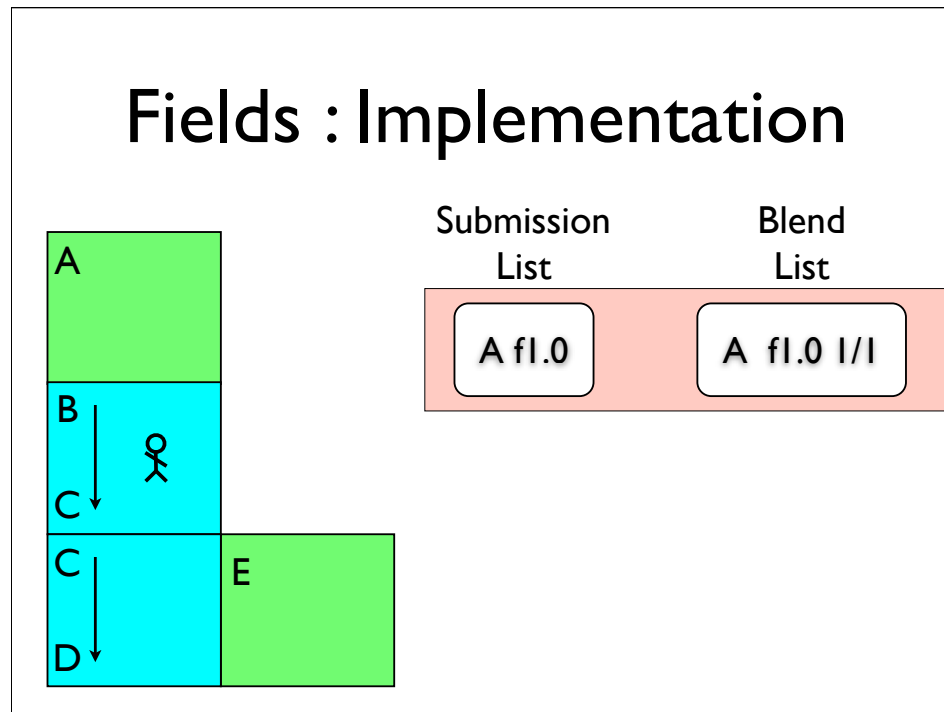
Fields : Implementation



So lets see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

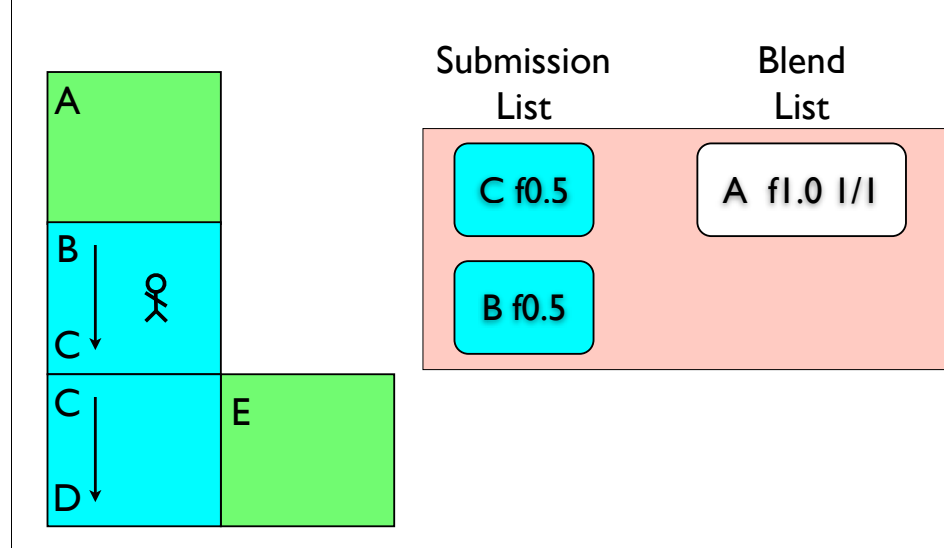
Fields : Implementation



So lets see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

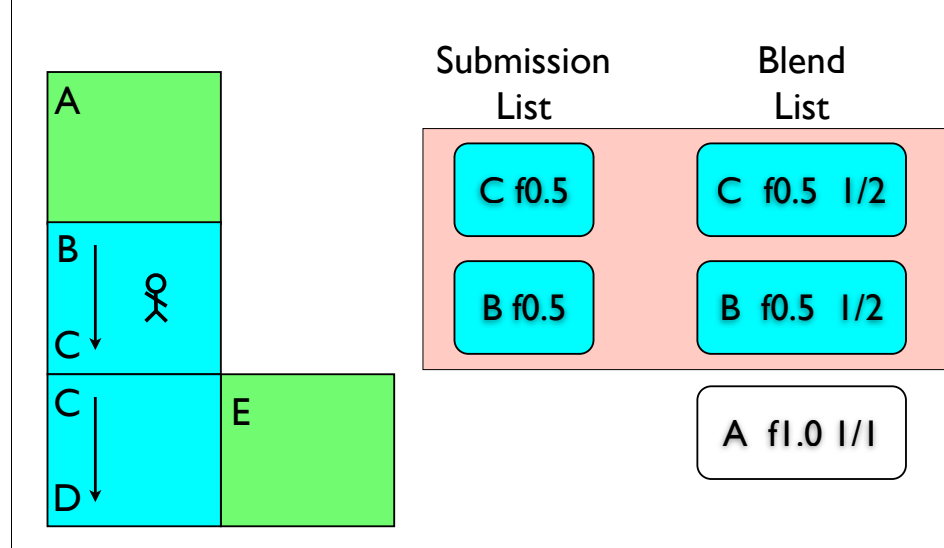
Fields : Implementation



So lets see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

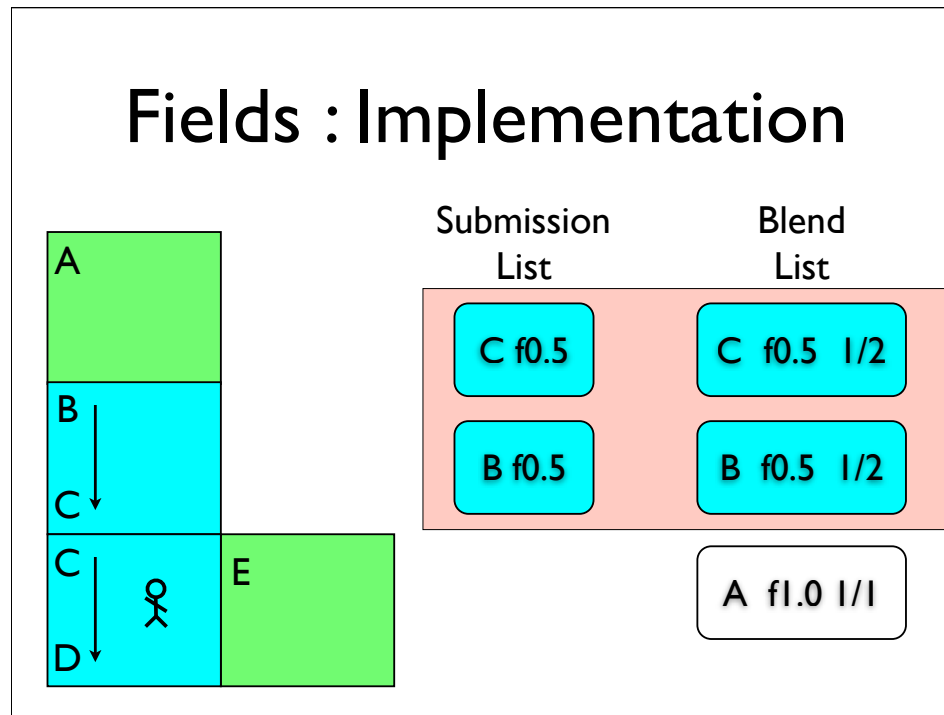
Fields : Implementation



So let's see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

* move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

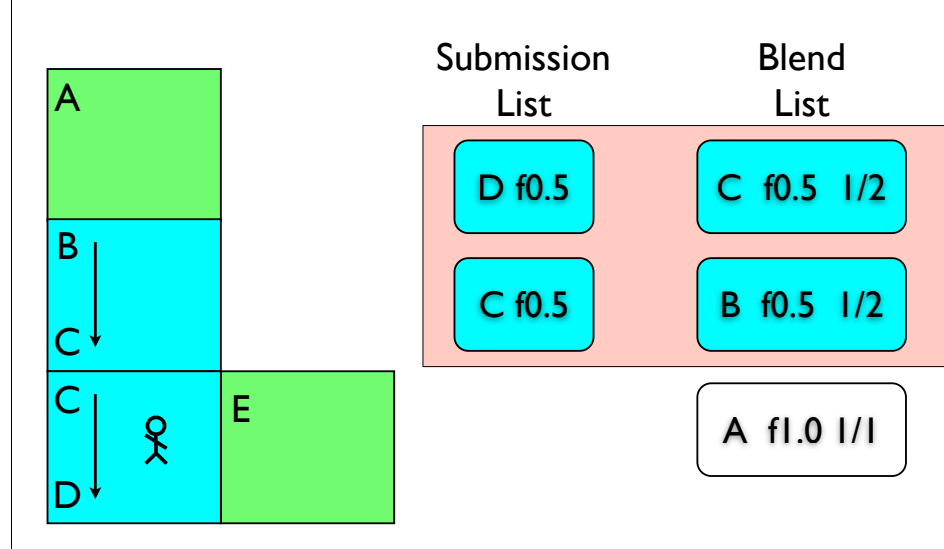
Fields : Implementation



So let's see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. Note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

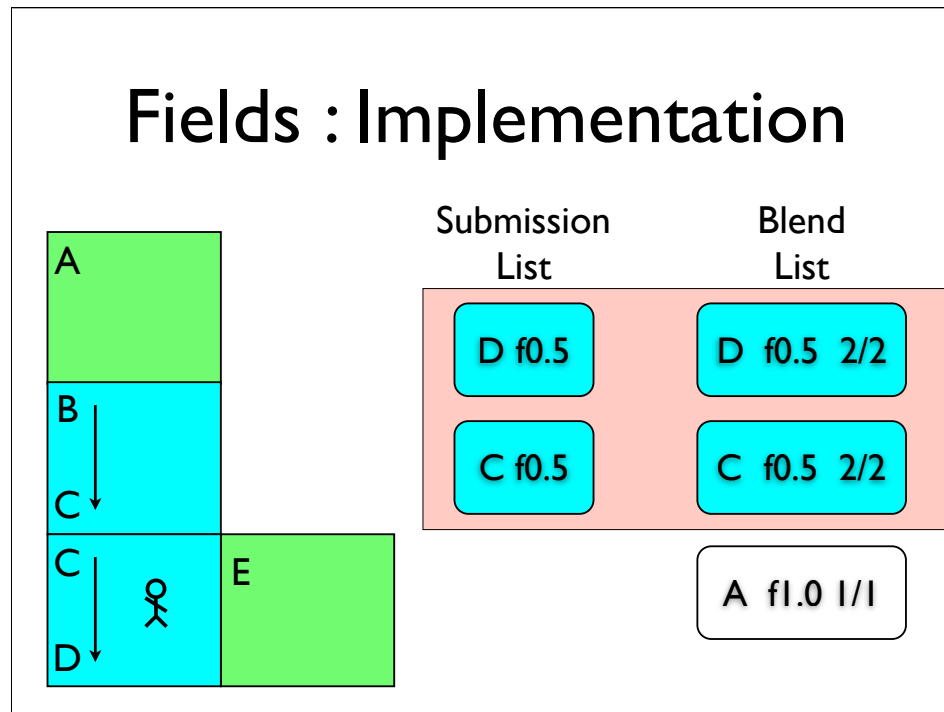
Fields : Implementation



So let's see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

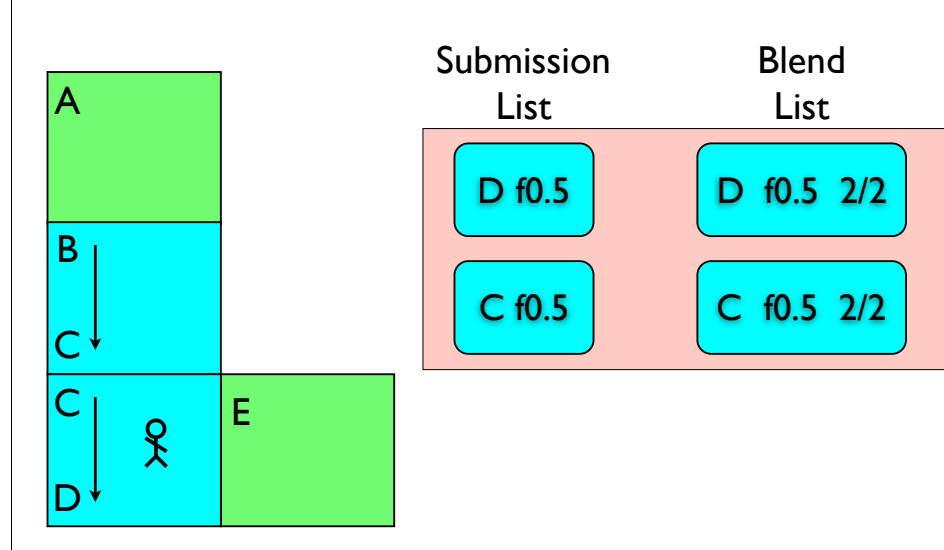
Fields : Implementation



So lets see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

* move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

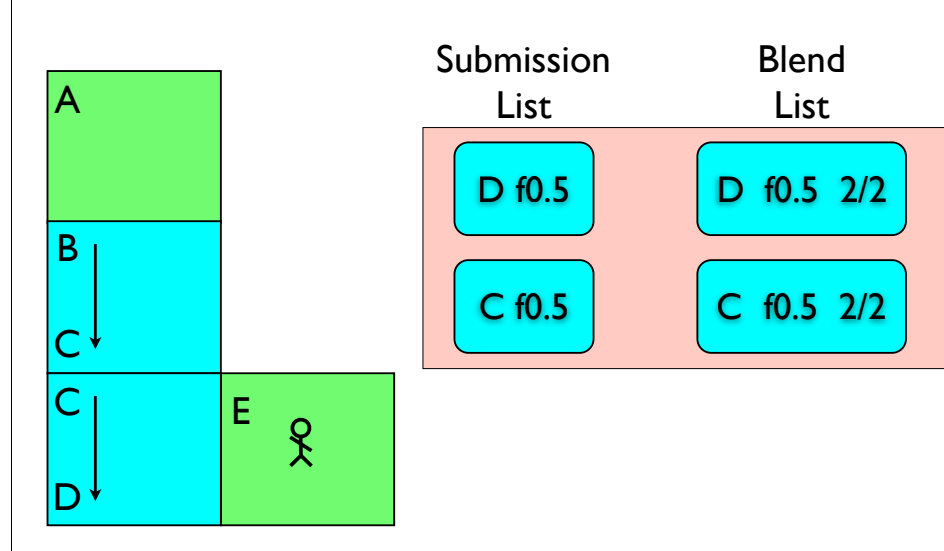
Fields : Implementation



So lets see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

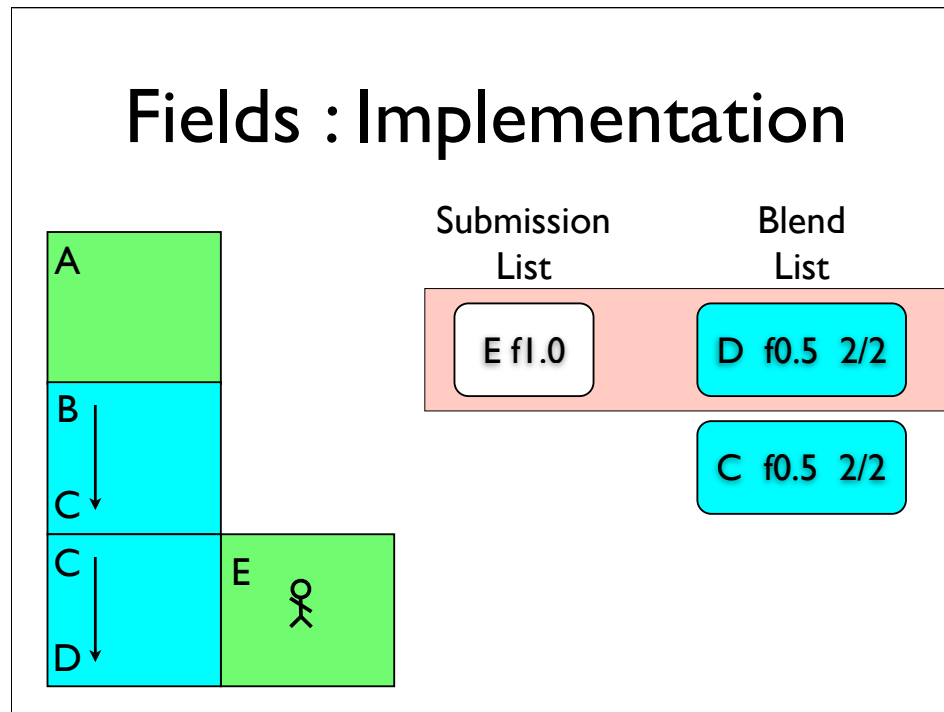
Fields : Implementation



So let's see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. Note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

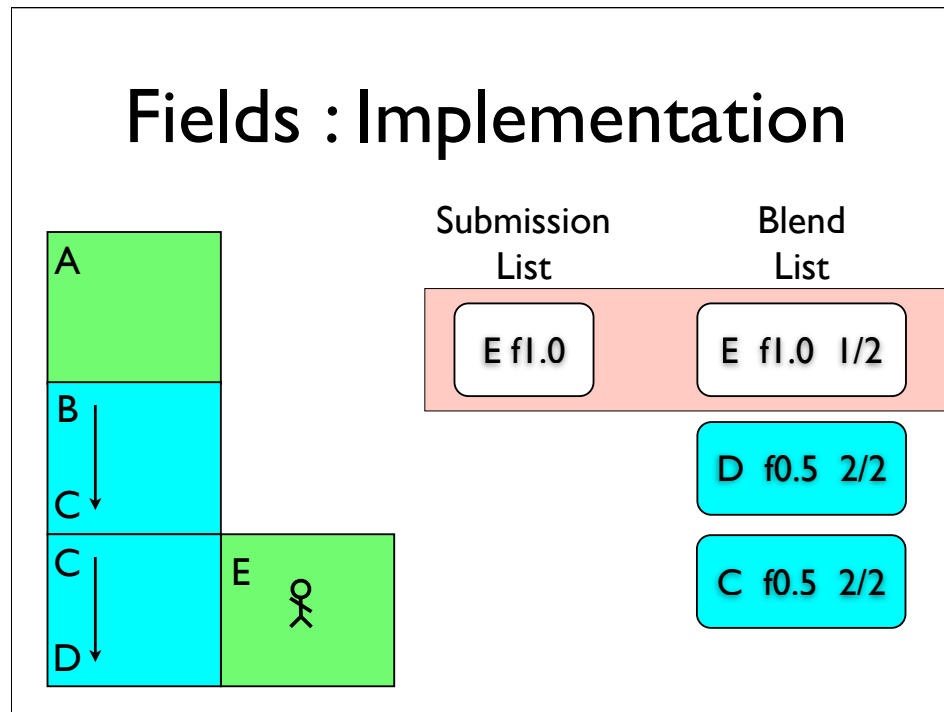
Fields : Implementation



So let's see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. Note that we've copied this field value into the blend primitive

- * move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

Fields : Implementation



So lets see how that works in an example. Here our field blended cameras are coloured blue. We start in zone a, a is in the submission list, and we give it a nominal field value of 1, this is just to make the maths for the blending a little simpler. note that we've copied this field value into the blend primitive

* move into the field blended zone for b and c, and b and c appear at the top of the submission list. Note that the red area, indicating the currently active cameras has grown. That's because unlike before where we only had one camera nominally active at any one time, we can now have many.

Fields : Implementation

E f1.0 1/2

D f0.6 2/2

C f0.3 2/2

B f0.6 2/2

So, lets have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights

- *and sum them within their group so that we can

- *normalise them

- *we then blend between timers, remember that field blended groups share timers

- *and multiply the groups weight back into the individual weights

Fields : Implementation

E f1.0 1/2	1.0
D f0.6 2/2	0.6
C f0.3 2/2	0.8
B f0.6 2/2	0.6

So, lets have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights
- *and sum them within their group so that we can
- *normalise them
- *we then blend between timers, remember that field blended groups share timers
- *and multiply the groups weight back into the individual weights

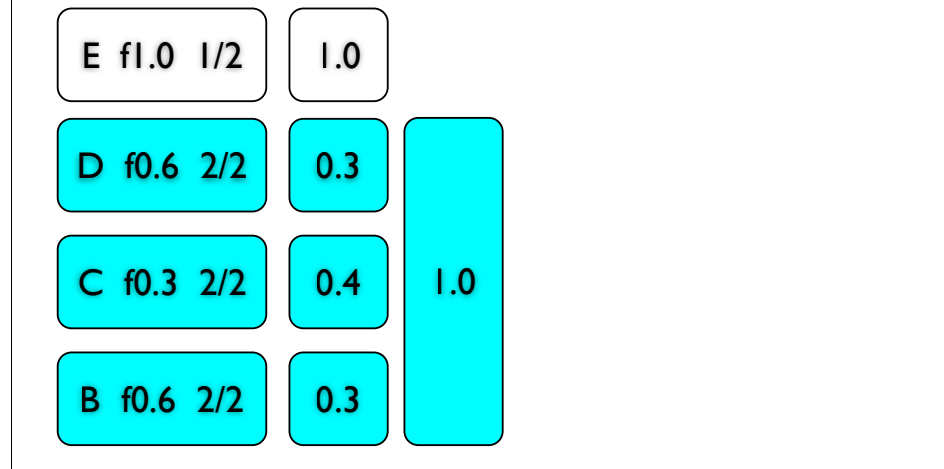
Fields : Implementation

E f1.0 1/2	1.0	
D f0.6 2/2	0.6	2.0
C f0.3 2/2	0.8	
B f0.6 2/2	0.6	

So, lets have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights
- *and sum them within their group so that we can
- *normalise them
- *we then blend between timers, remember that field blended groups share timers
- *and multiply the groups weight back into the individual weights

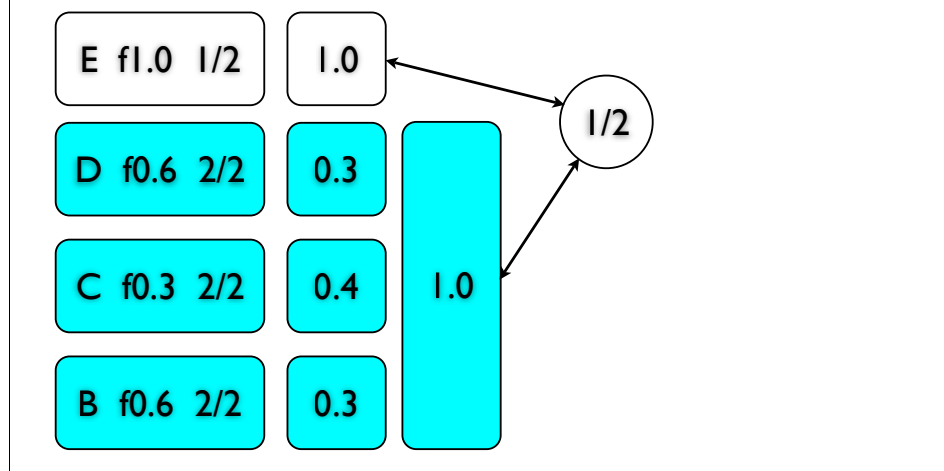
Fields : Implementation



So, let's have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights
- *and sum them within their group so that we can
- *normalise them
- *we then blend between timers, remember that field blended groups share timers
- *and multiply the groups weight back into the individual weights

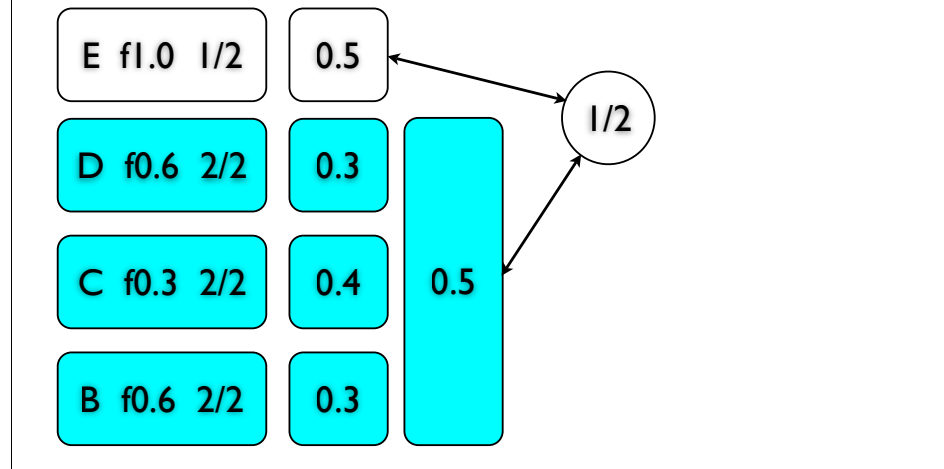
Fields : Implementation



So, let's have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights
- *and sum them within their group so that we can
- *normalise them
- *we then blend between timers, remember that field blended groups share timers
- *and multiply the groups weight back into the individual weights

Fields : Implementation



So, let's have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights
- *and sum them within their group so that we can
- *normalise them
- *we then blend between timers, remember that field blended groups share timers
- *and multiply the groups weight back into the individual weights

Fields : Implementation

E f1.0 1/2	0.5
D f0.6 2/2	0.15
C f0.3 2/2	0.2
B f0.6 2/2	0.15

So, lets have a look at an example blend list, and see how we calculate the overall weights

- *first we copy our field values into a set of weights
- *and sum them within their group so that we can
- *normalise them
- *we then blend between timers, remember that field blended groups share timers
- *and multiply the groups weight back into the individual weights

Fields : Implementation

E f1.0 1/2	0.5
D f0.6 2/2	0.15
C f0.3 2/2	0.2
B f0.6 2/2	0.15

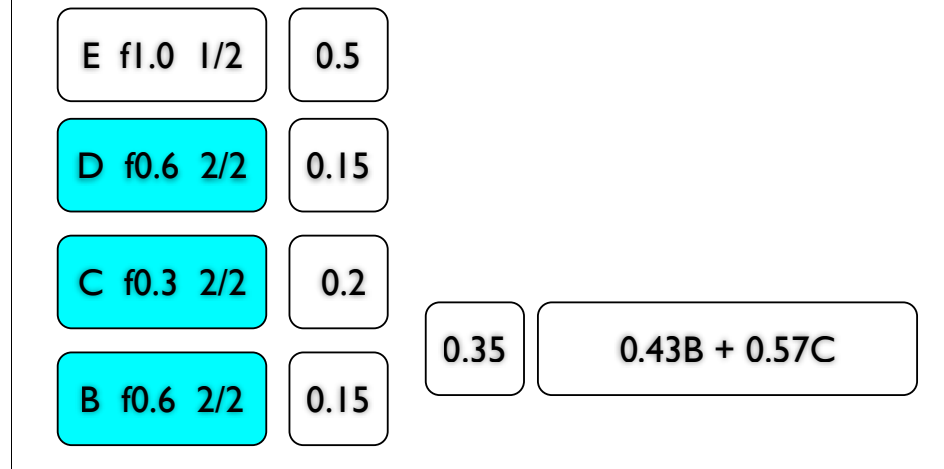
Having calculated our list of weights, we iterate along it blending in pairs as before

We calculate each weight in the pair, as a proportion of their sum, and give the result the sum as it's weight

We then blend this with next entry in the blend list

Until we've blended the last entry. So that's pretty much the same as for timed blends, except that we pre-calculate the weights, and

Fields : Implementation



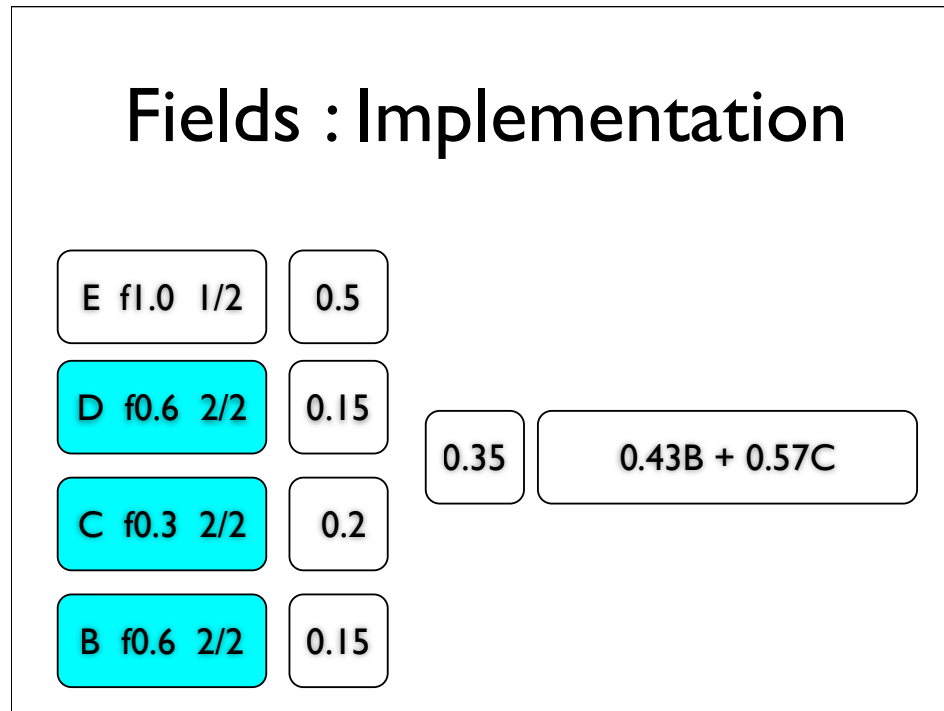
Having calculated our list of weights, we iterate along it blending in pairs as before

We calculate each weight in the pair, as a proportion of their sum, and give the result the sum as it's weight

We then blend this with next entry in the blend list

Until we've blended the last entry. So that's pretty much the same as for timed blends, except that we pre-calculate the weights, and

Fields : Implementation



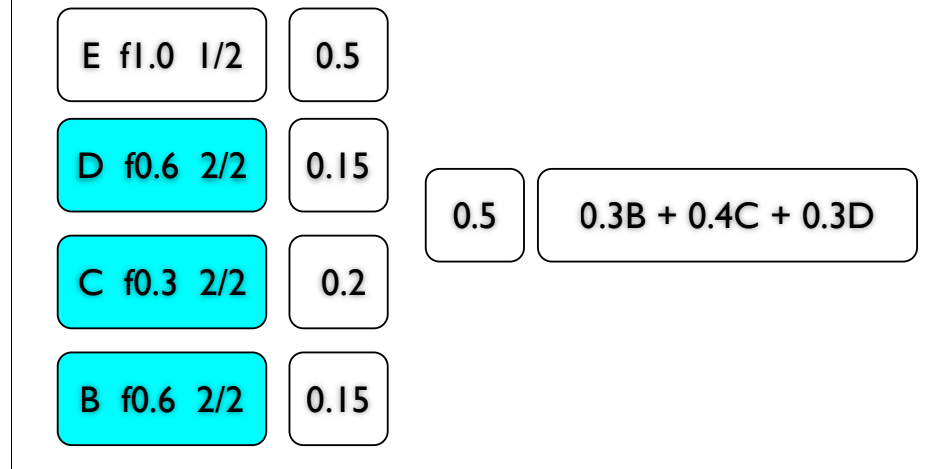
Having calculated our list of weights, we iterate along it blending in pairs as before

We calculate each weight in the pair, as a proportion of their sum, and give the result the sum as its weight

We then blend this with next entry in the blend list

Until we've blended the last entry. So that's pretty much the same as for timed blends, except that we pre-calculate the weights, and

Fields : Implementation



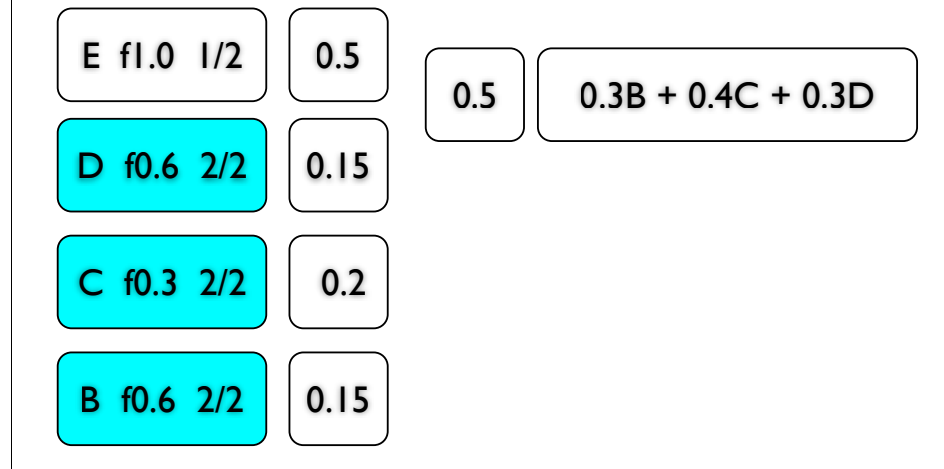
Having calculated our list of weights, we iterate along it blending in pairs as before

We calculate each weight in the pair, as a proportion of their sum, and give the result the sum as it's weight

We then blend this with next entry in the blend list

Until we've blended the last entry. So that's pretty much the same as for timed blends, except that we pre-calculate the weights, and

Fields : Implementation



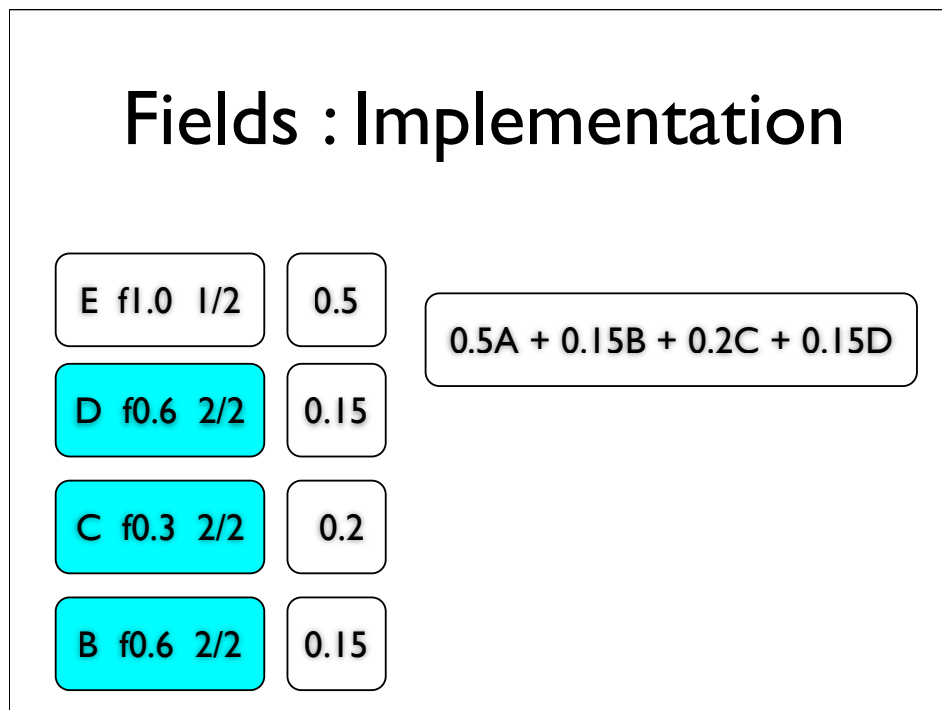
Having calculated our list of weights, we iterate along it blending in pairs as before

We calculate each weight in the pair, as a proportion of their sum, and give the result the sum as it's weight

We then blend this with next entry in the blend list

Until we've blended the last entry. So that's pretty much the same as for timed blends, except that we pre-calculate the weights, and

Fields : Implementation



Having calculated our list of weights, we iterate along it blending in pairs as before

We calculate each weight in the pair, as a proportion of their sum, and give the result the sum as it's weight

We then blend this with next entry in the blend list

Until we've blended the last entry. So that's pretty much the same as for timed blends, except that we pre-calculate the weights, and

Overview

- Zoning
- Dynamics
- Blending
- Rails
- **Fields**

Overview

- Zoning
- Dynamics
- Blending
- Rails
- Fields

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