Game Developers Conference®

February 28 - March 4, 2011
Moscone Center, San Francisco

www.GDConf.com

UBM TechWeb

GDC

DOING MORE WITH  LESS

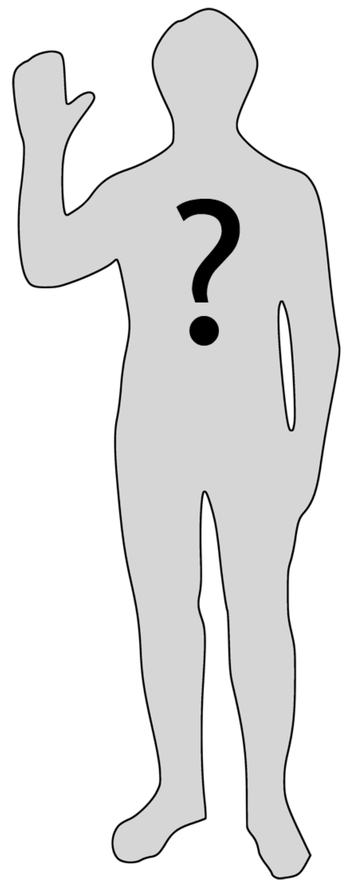
Animating NPCs in Uncharted

John Bellomy

AI & Animation Programmer

Animation System

- Emphasis on NPCs
- Beginning with Uncharted 1
- Problems Encountered
- Solutions Developed for U2



Overview talk here

animation system, emphasis on npcs
story of uncharted 1 to uncharted 2
problems encountered
solutions

Animation System

Animation System

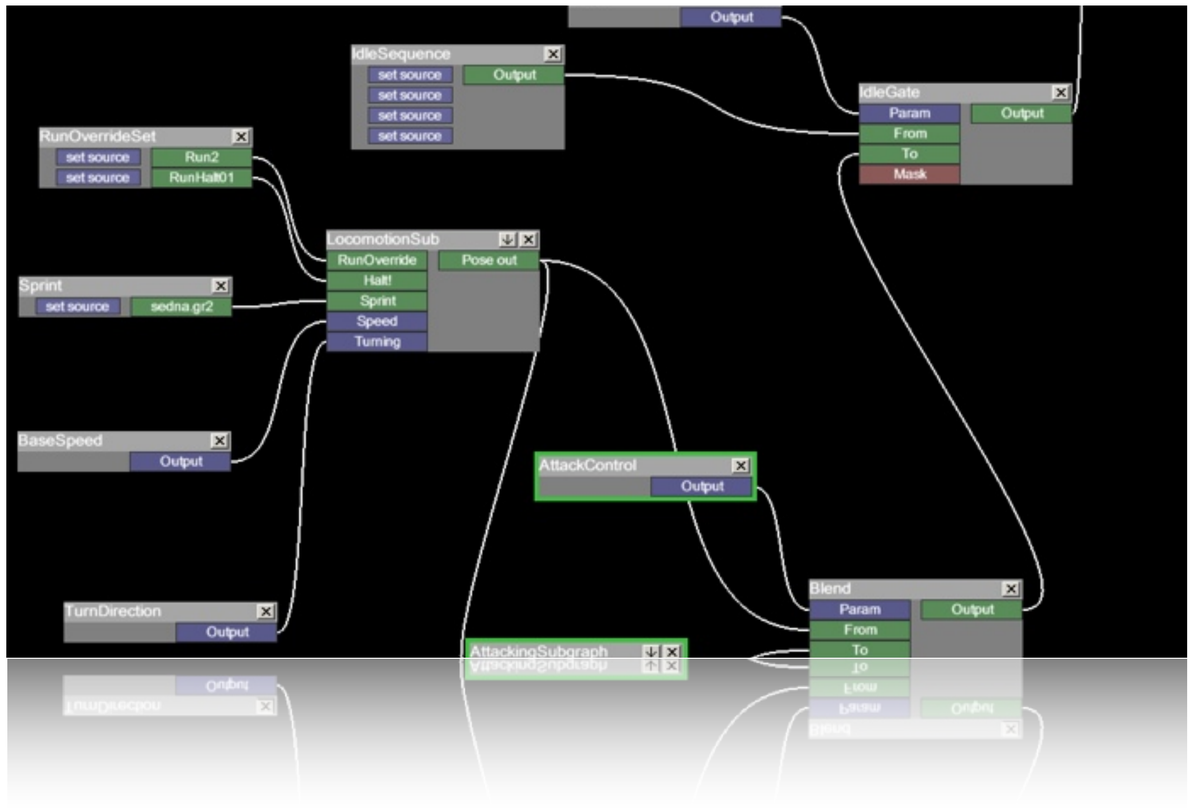
- Robust
- Flexible

Animation System

Animation System

- Fast
- Minimal Memory Cost

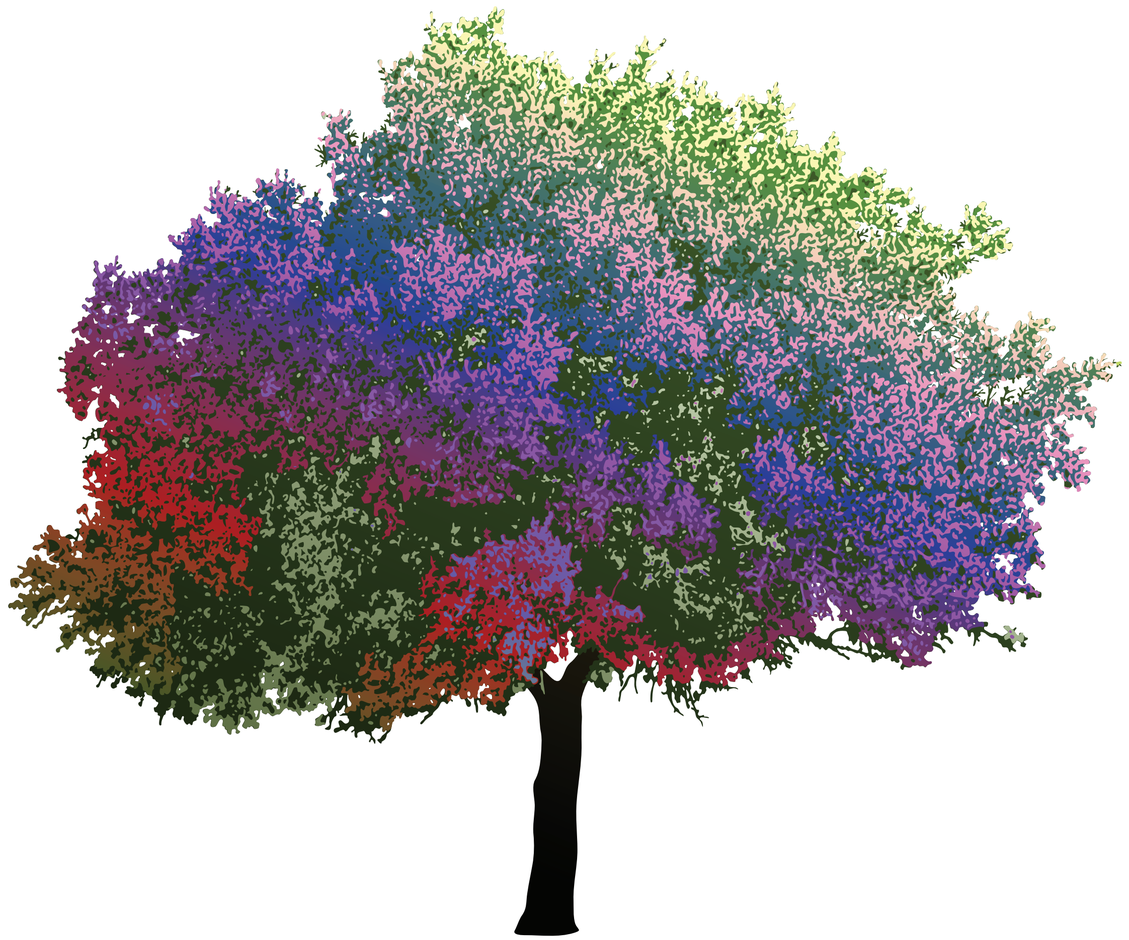
Node Based



Show screenshot of a node editor

One common approach is to have one large node graph controlling the animations and blends for a character

Node Based



This can be thought of as one big tree

State Graph Based

a single tree per behavior

Lets talk about Uncharted 1.

It was a from-scratch engine created for the PS3.

Christian Gyrling previous talks

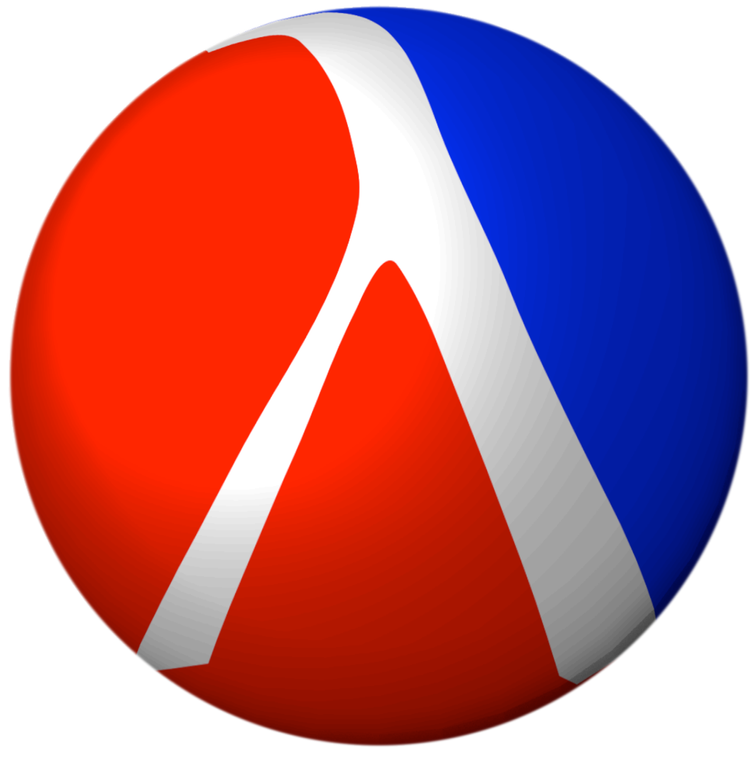
Most game engines & animation software packages out there will include a well developed node-based animation editor and build their runtimes to work with this data. We did not have any such editor.

What we did have...

Was a good familiarity with Lisp

DC

- Based on PLT Scheme
- Dual Roles:
  - Data Generation
  - Runtime Scripting
- Compiles into bin file



Limit DC example context to animation system

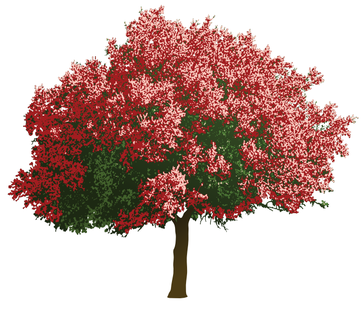
Uncharted States

A character would have a module which contained all it’s states

Uncharted States

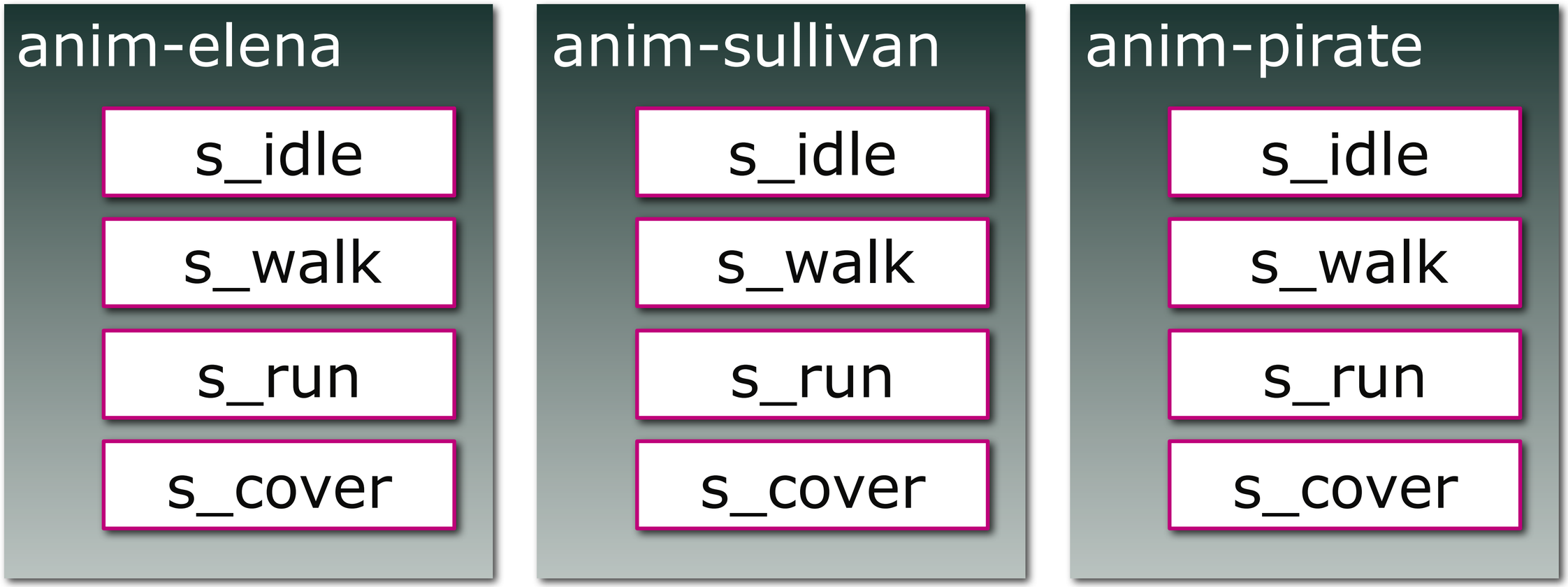
anim-elena

- s_idle
- s_walk
- s_run
- s_cover



A character would have a module which contained all it’s states

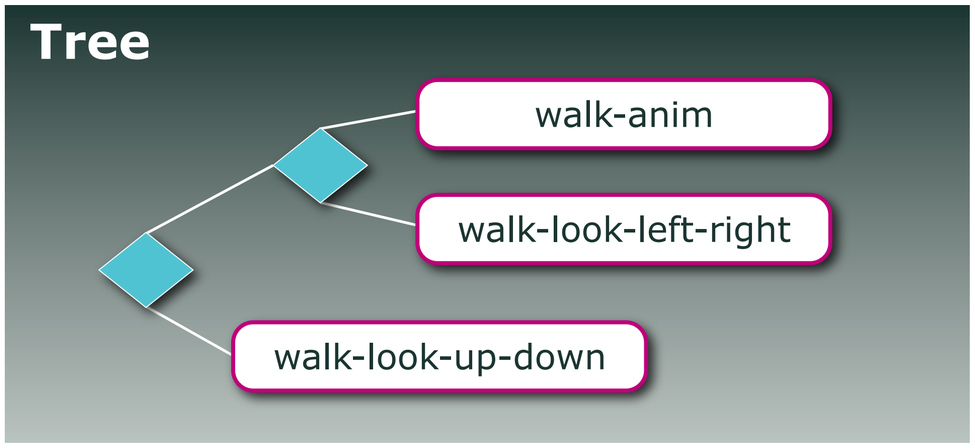
Uncharted States



Each character had their own module to describe their animation state graph

Animation State

s_walk



Transitions

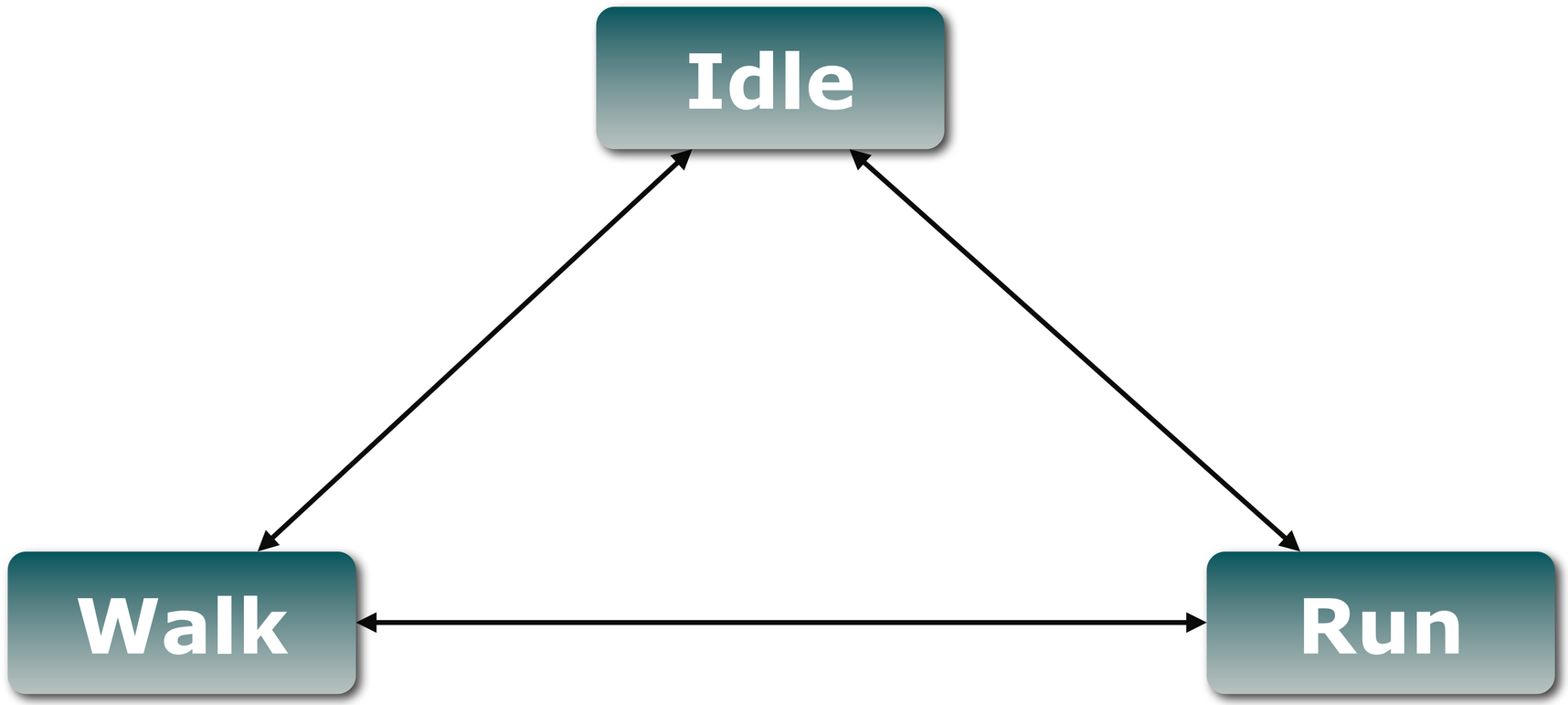
idle ⟶ s_idle

run ⟶ s_run

Script Funcs

λ

Transitions



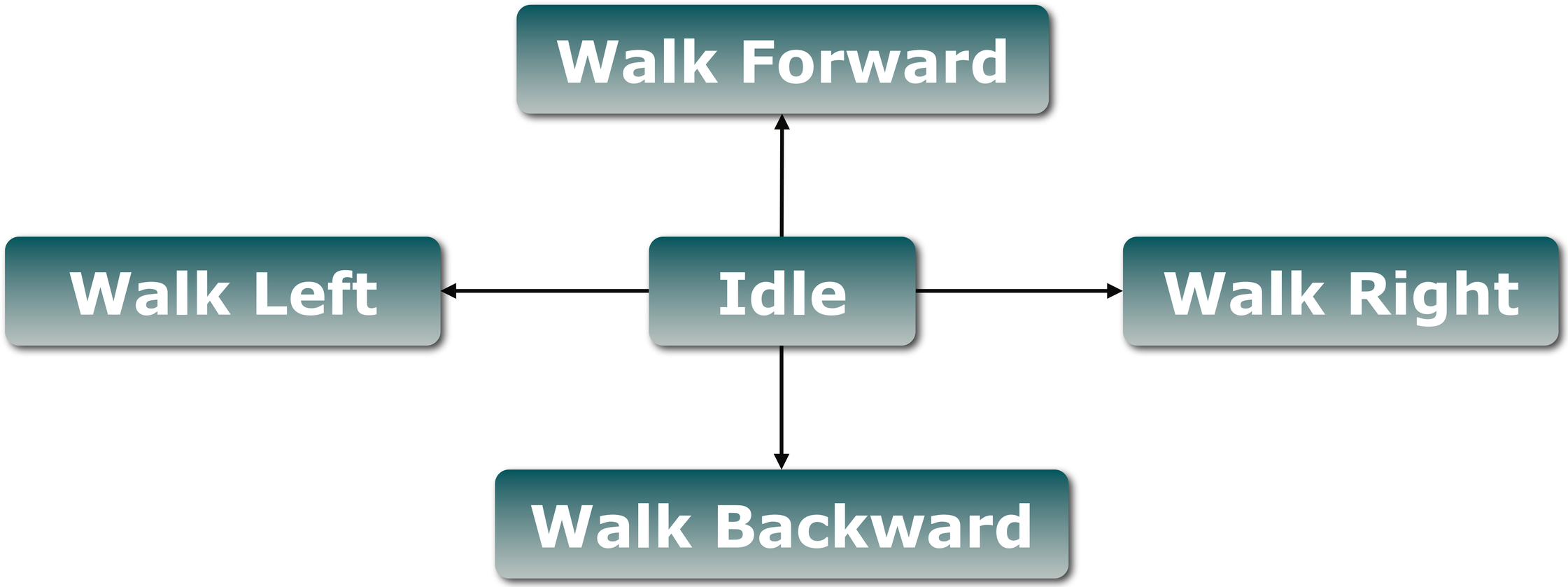
States are connected via transition links.

Transitions describe both the behavior of the blend (i.e. blend time, blend curve, blending motion versus blending animation)

Transitions

- Blend Time
- Curve Type
- Animation & Motion Blending Controls
- Optional Conditions

Transitions

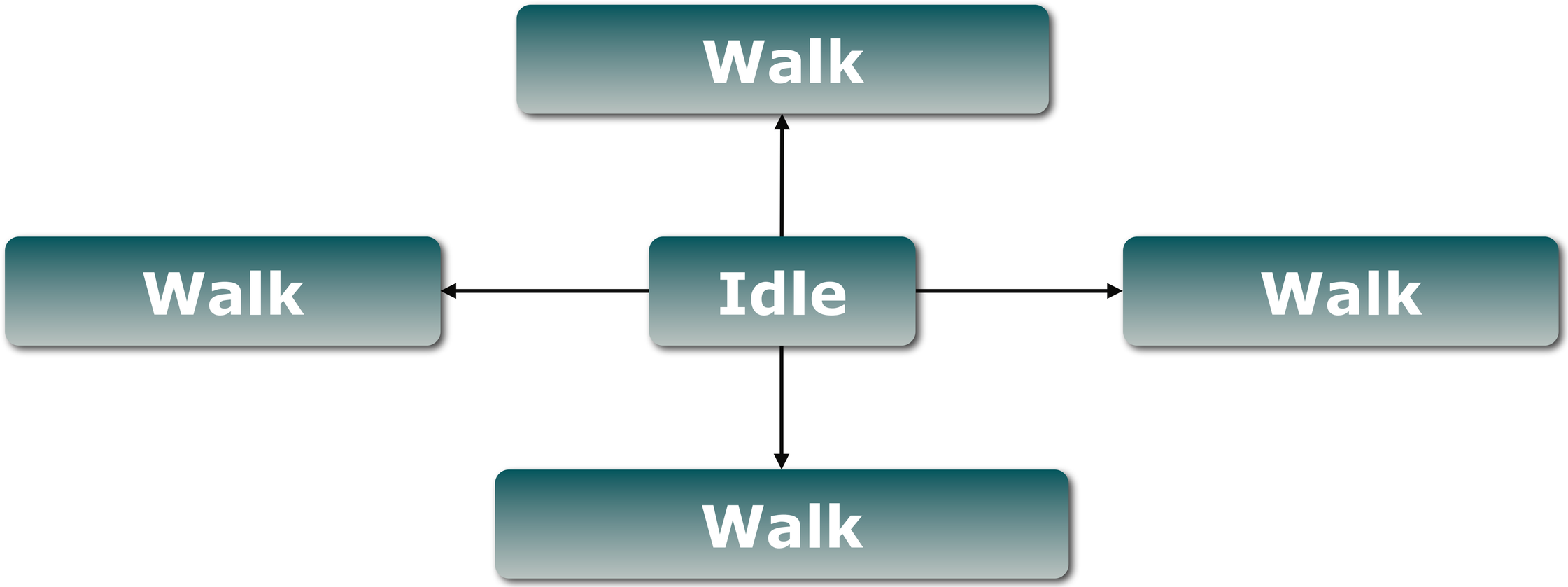


A major benefit about this approach is the use of named transitions between states.

Instead of taking transitions by requesting the desired state explicitly, each transition is named and thusly requested by. The calling code doesn't even need to know about what state it's currently in, let alone how the transition will behave (we have various helper functions to query for transition availability).

This means a transitions behavior (how long a blend, if it went to an intermediary state, etc) could be defined independently of it's functionality (this is a 'walk' transition).

Transitions



A major benefit about this approach is the use of named transitions between states.

Instead of taking transitions by requesting the desired state explicitly, each transition is named and thusly requested by. The calling code doesn't even need to know about what state it's currently in, let alone how the transition will behave (we have various helper functions to query for transition availability).

This means a transitions behavior (how long a blend, if it went to an intermediary state, etc) could be defined independently of it's functionality (this is a 'walk' transition).

Transitions

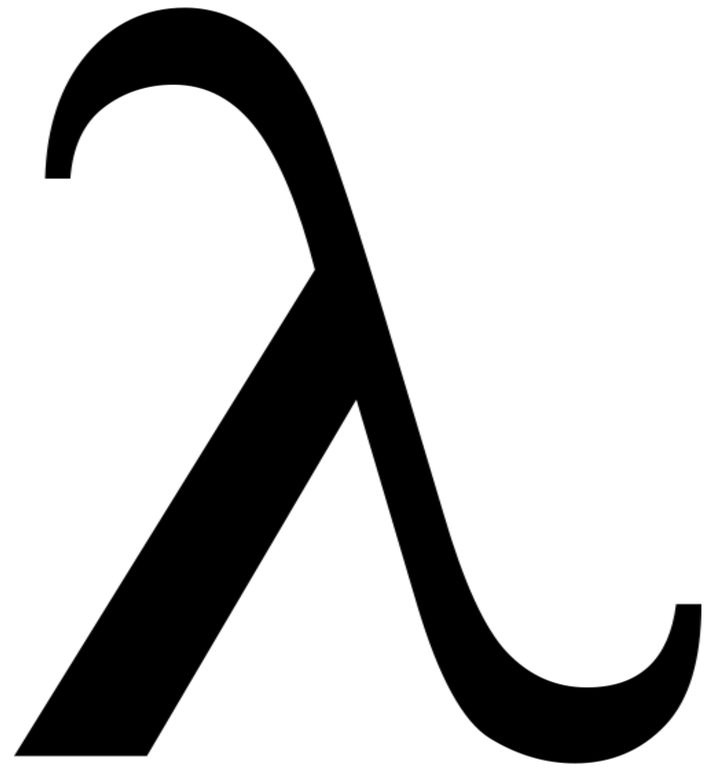
Often we will have multiple transitions with the same name with varying conditions based on the desired character behavior.

This is an example of the locomotion logic requesting the walk transition and the correct directional transition being taken automatically. The higher level logic didn't need to deduce a walk-left or walk-180 transition was needed, it just said 'walk'

Animation State

```
(define-state s_walk
    :tree (blend
            (blend
              (anim "walk-anim")
              (anim “look--left-right“))
          (anim “look--up-down“))      :transitions (
                                          (transition 'idle    's_idle)
                                          (transition 'run     's_run)
                                          (transition 'sprint 's_sprint))
                                    )
```

Anim script funcs



- One benefit of authoring our animation states in scheme was that it was easy to inline snippets DC Script (our runtime scripting language)
- These were defined as scheme lambdas
- We would use these “animation funcs” various things, building in some initial flexibility
  - E.g. an animation node could use it to repurpose the time axis of an animation

Animation State

```
(define-state s_walk
   :tree (blend
           (blend
             (anim "walk-anim")
             (anim “look--left-right“))
          (anim “look--up-down“))

   :phase-func (npc-phase-func (* 1.5 phase))

   :transitions (
      (transition 'idle    's_idle)
      (transition 'run     's_run)
      (transition 'sprint 's_sprint))
)
```

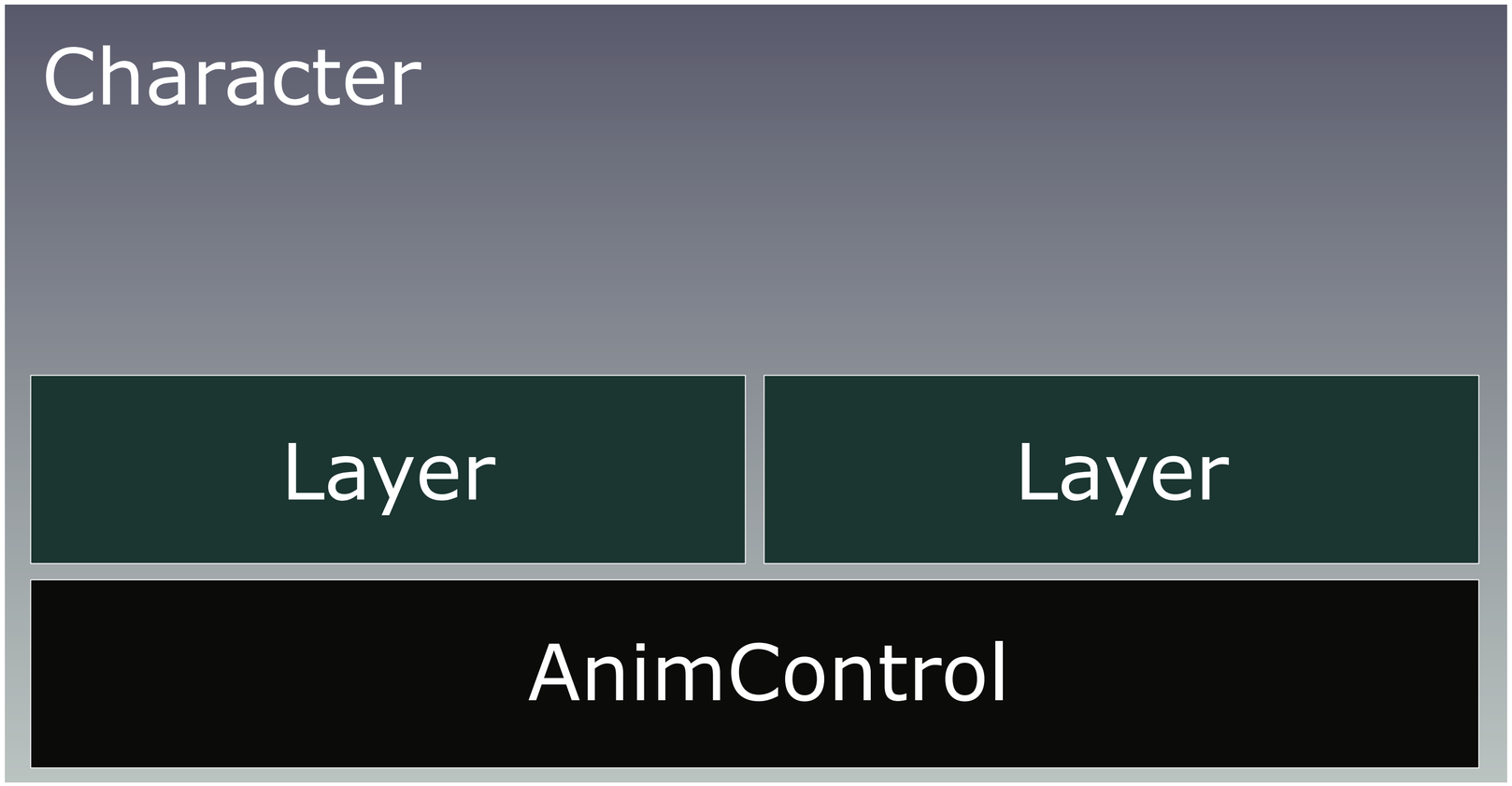
Runtime

Runtime

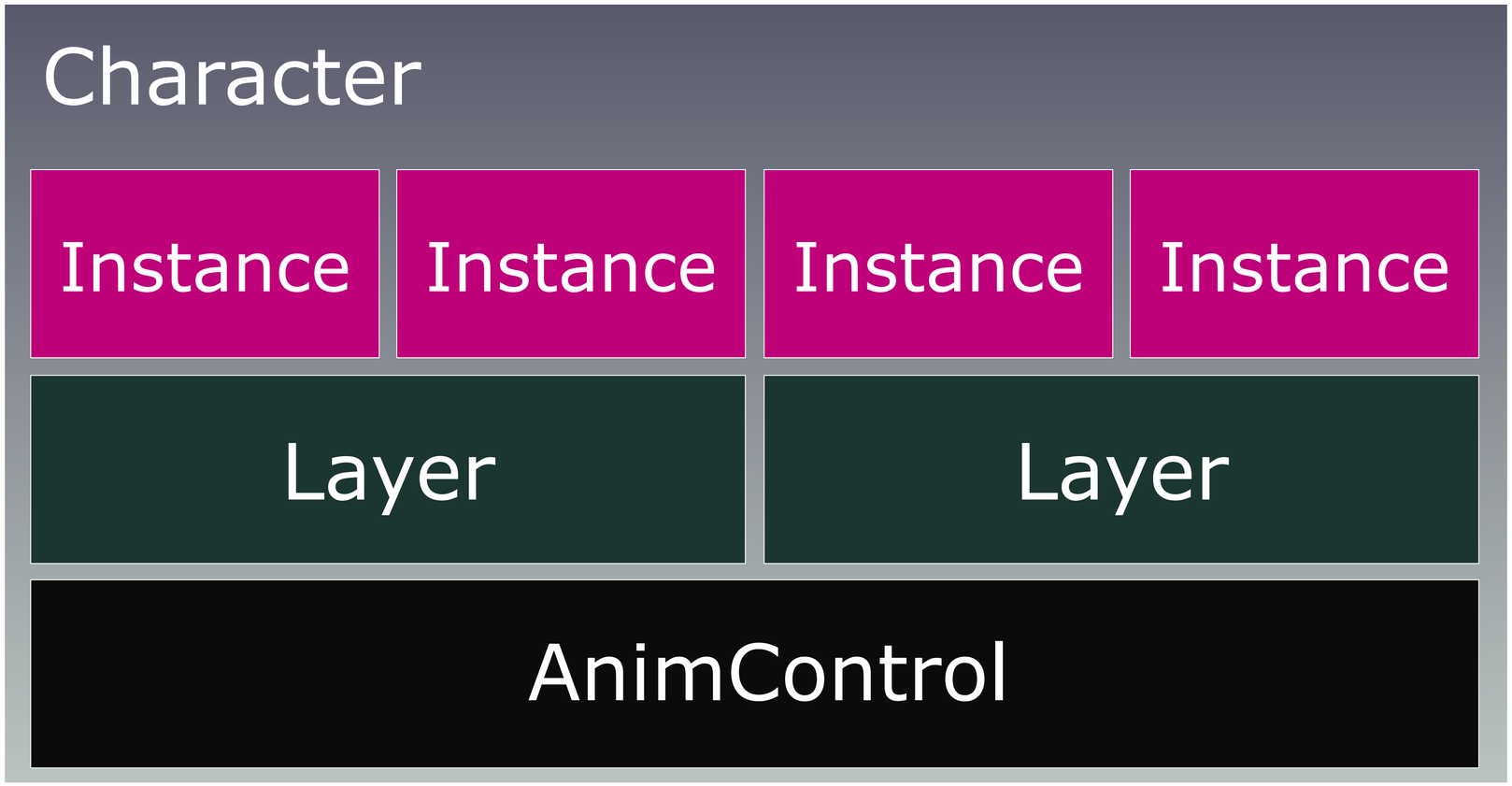
Character

AnimControl

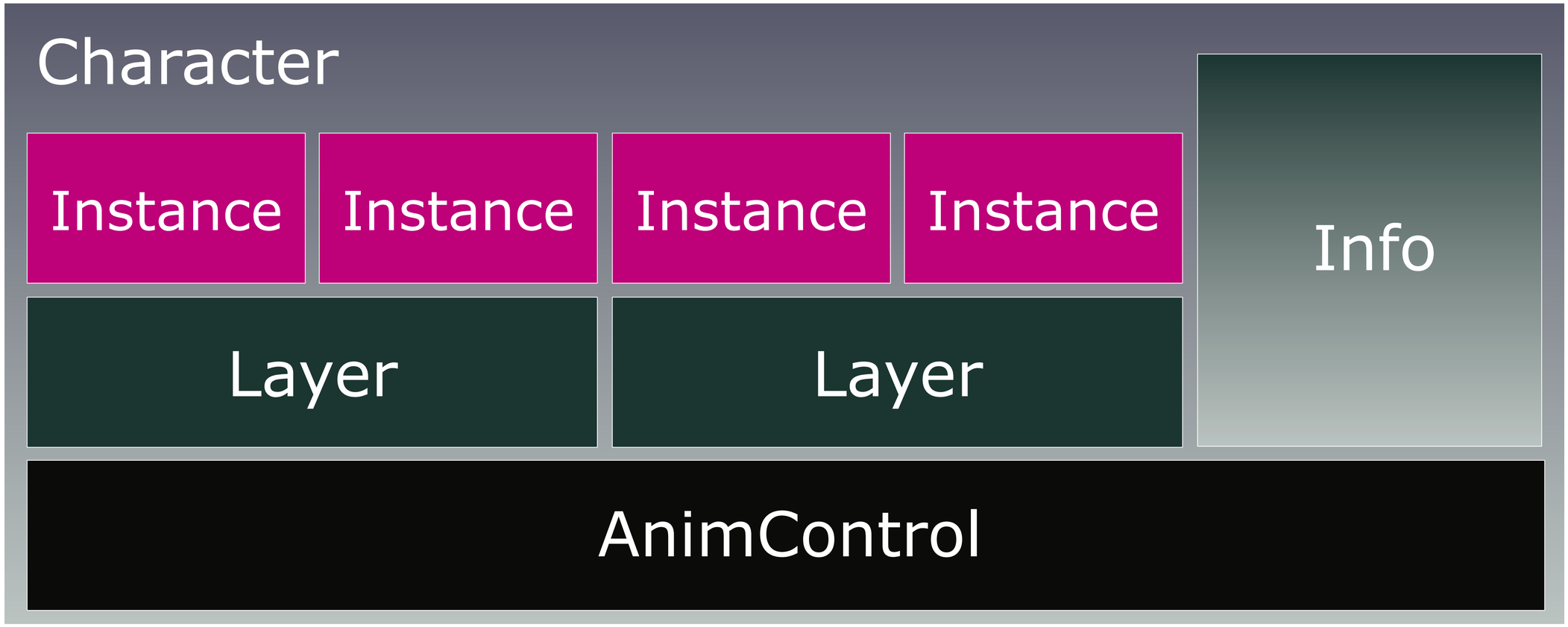
Runtime



Runtime


Character
Instance
Instance
Instance
Instance
Layer
Layer
AnimControl

Runtime


Character
Instance
Instance
Instance
Instance
Info
Layer
Layer
AnimControl

Runtime

Runtime

Info

Aim Blend Factors

Movement Speed

Facing Direction

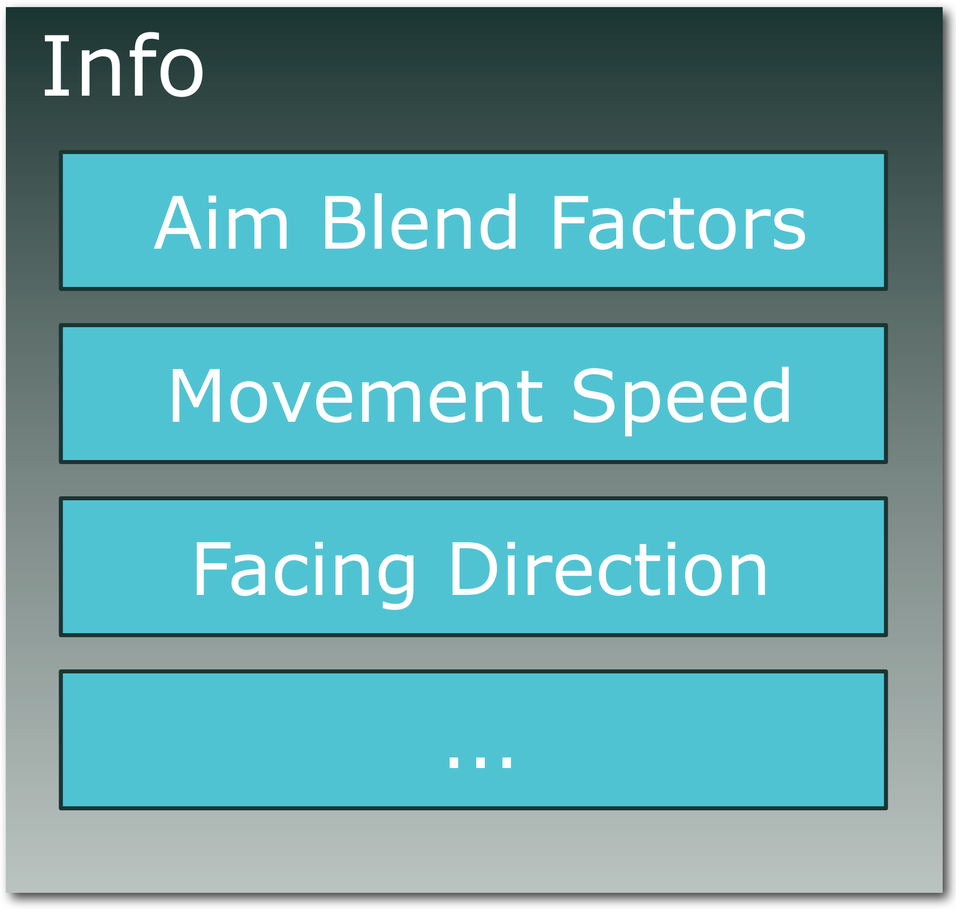
...

The info structure provides context

The animation system cannot read game data, only what is provided via the info structure

Animation world is isolated, and the info structure is the ambassador

Runtime

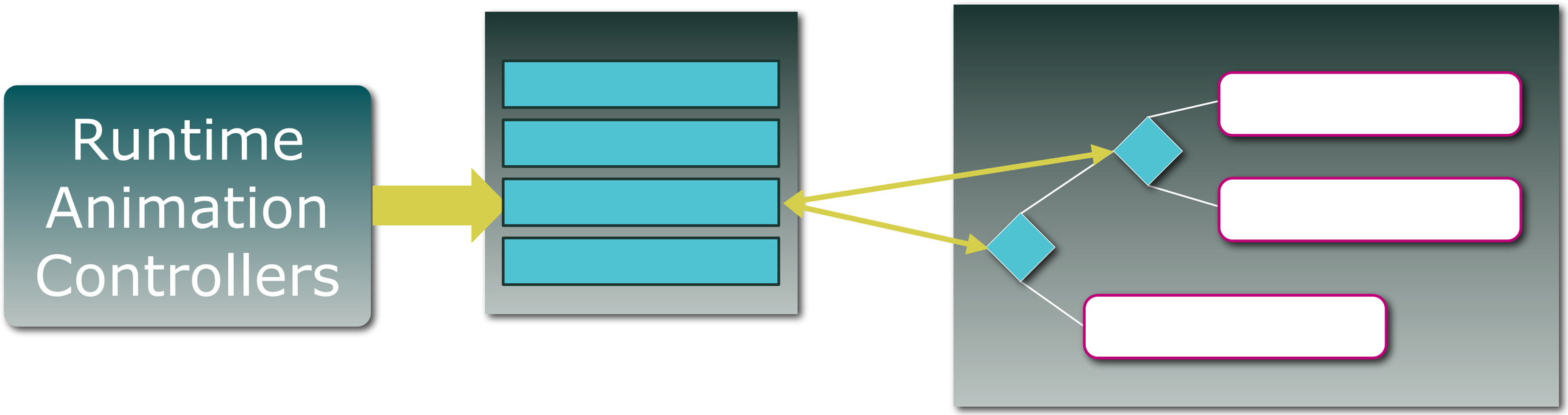


The info structure provides context

The animation system cannot read game data, only what is provided via the info structure

Animation world is isolated, and the info structure is the ambassador

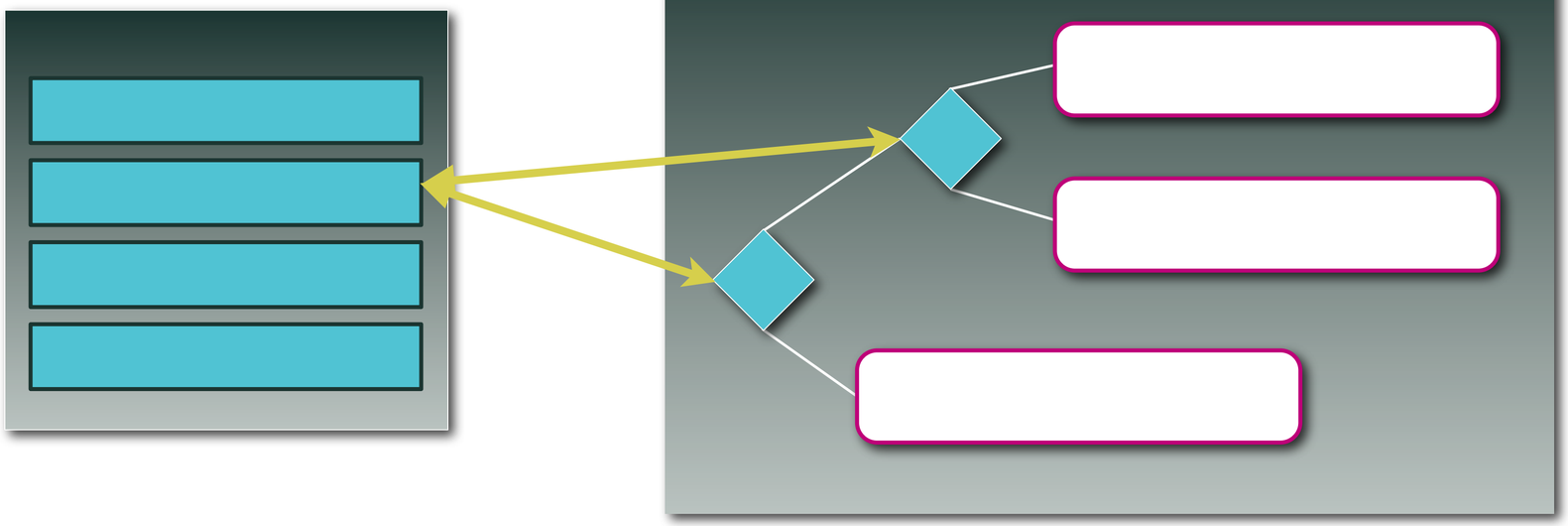
Runtime

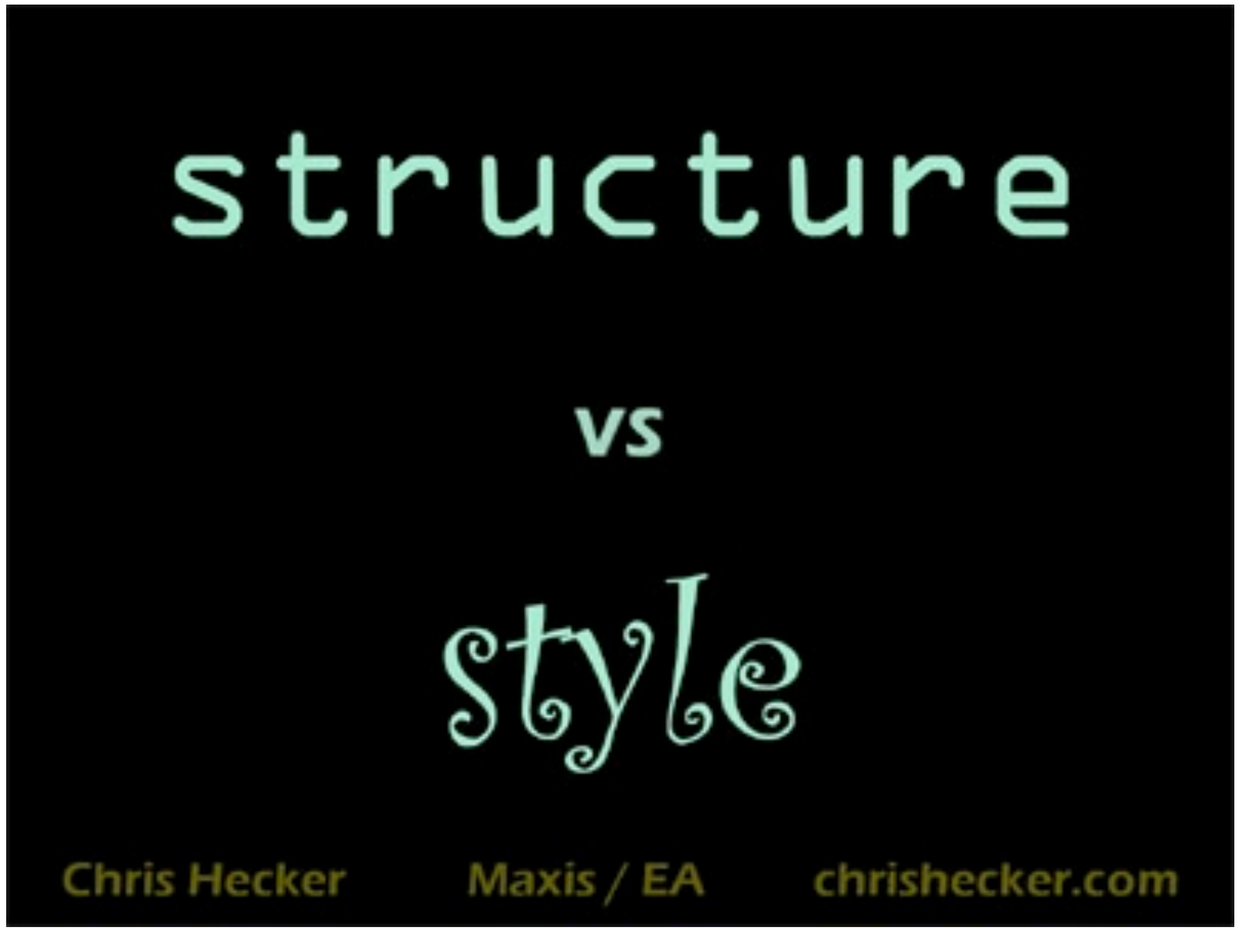


For doing things like aiming and look ats, we would combine script funcs and info structures. Info structures became a way for the higher level AI logic to influence the animation tree without having to know specific implementation details of the currently active tree.

In the beginning we tried to get all our flexibility and features through script funcs and info structures, but it became too cumbersome and clunky. Adding a feature meant changes in multiple places and there was no real good way to source the data on the runtime side.

Info structures





http://chrishecker.com/Structure_vs_Style

- Separating Structure and Style allows you to do both more efficiently
-

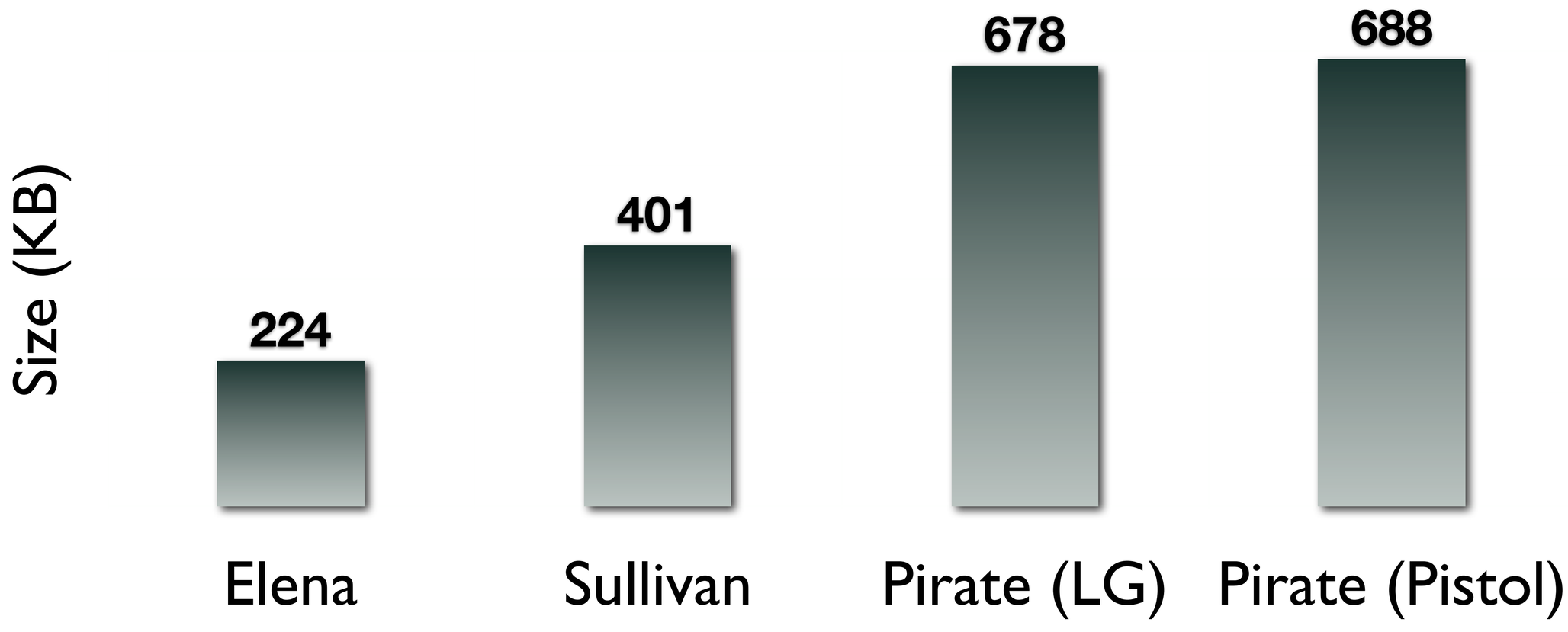
Moving Forward

-More characters
-More variations across character types
-More variations within character types

UDF Anim Graph Memory Usage

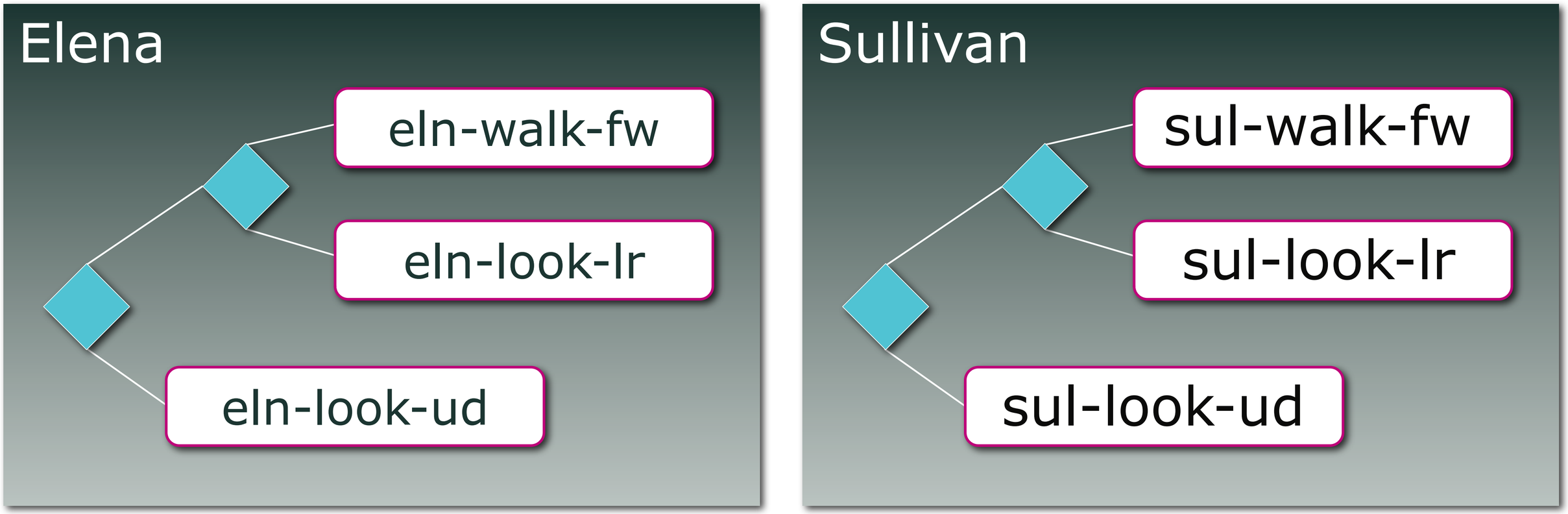
Jumping back to the end of Uncharted 1, our first priority was to get that crazy memory usage down.

UDF Anim Graph Memory Usage



Jumping back to the end of Uncharted 1, our first priority was to get that crazy memory usage down.

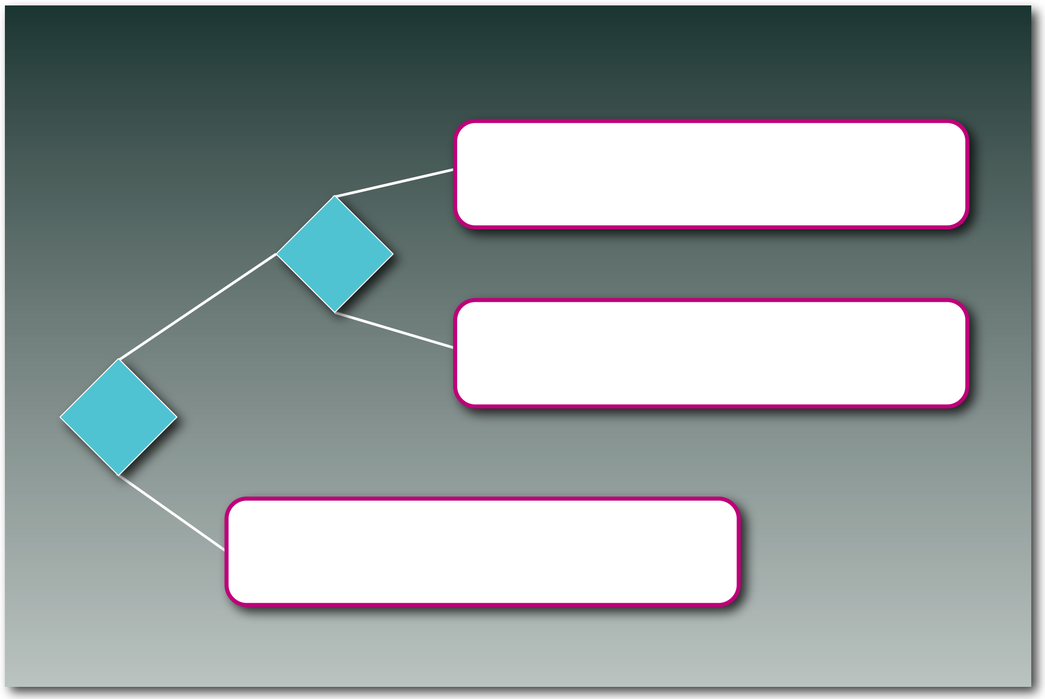
Duplicate Trees



When we looked at all the different animation state graphs and trees there wasn't that much difference between them. With some cajoling we could make every character use the same state graph and tree structure.

Of course each of the previous bin files referenced different animation names so the solution because obvious: introduce an animation name abstraction system and thus our first real foray into separating style and structure.

Anim Sets



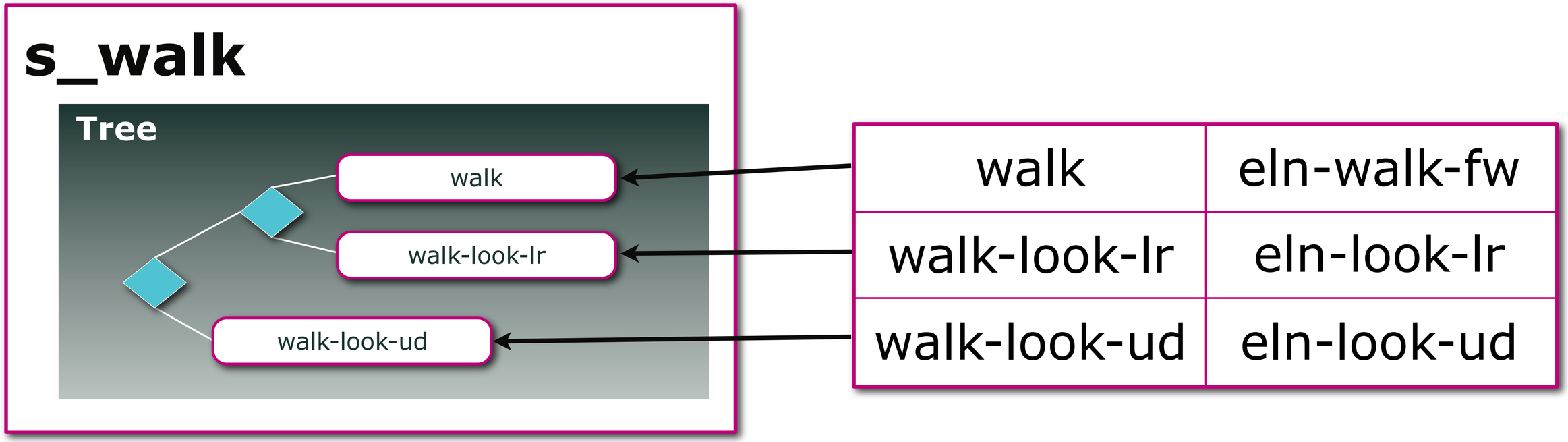
Sullivan
sul-walk-fw
sul-look-lr
sul-look-ud

Elena
eln-walk-fw
eln-look-lr
eln-look-ud

We want to extract out unique information

We use unique keys to pair ‘virtual’ animations

Anim Sets



At its core anim-sets are runtime translation of animation names. This is very simple but powerful.

We would define our state graph and tree in one as normal, but then create a second file that contained only anim sets for a particular character.

This is good because despite our shorthand macros to define animation states and trees it’s still not good for an animator to try and parse to make edits to animation names.

Anim Sets

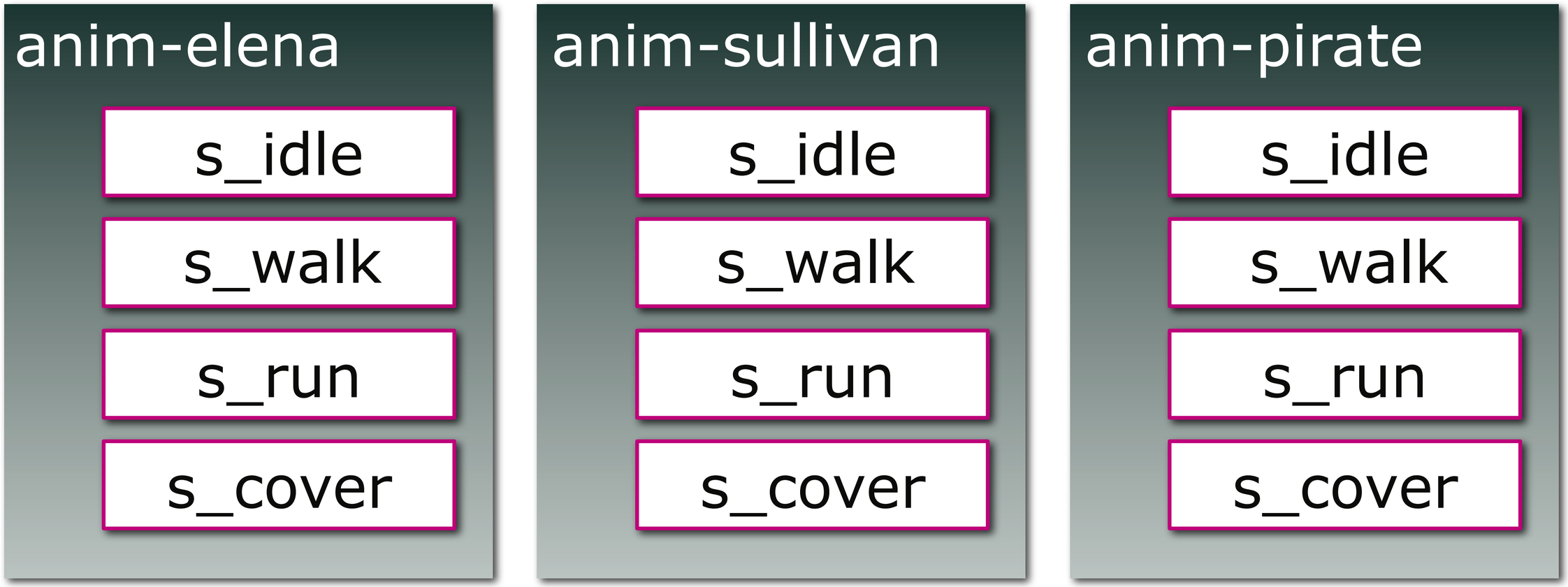
walk	eln-walk-fw
walk-look-lr	eln-look-lr
walk-look-ud	eln-look-ud

At its core anim-sets are runtime translation of animation names. This is very simple but powerful.

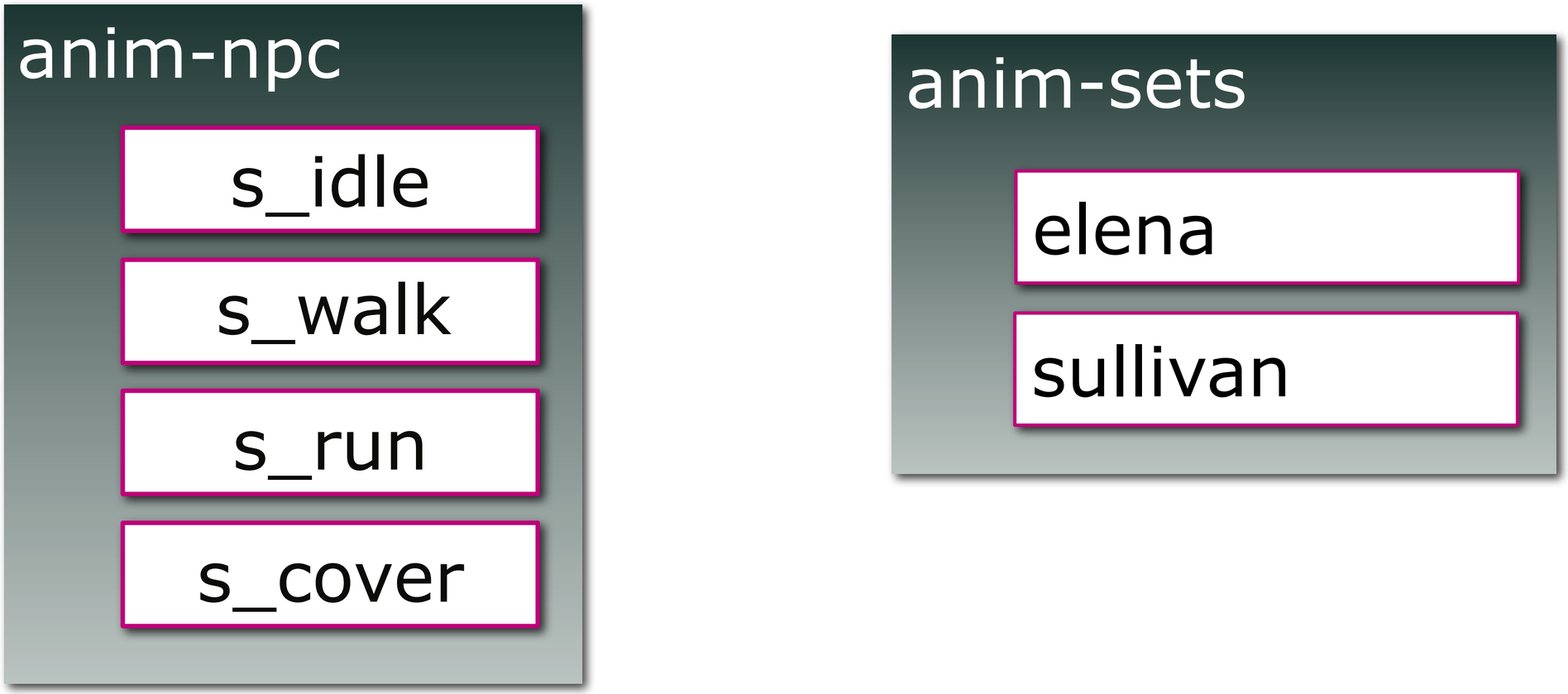
We would define our state graph and tree in one as normal, but then create a second file that contained only anim sets for a particular character.

This is good because despite our shorthand macros to define animation states and trees it’s still not good for an animator to try and parse to make edits to animation names.

Uncharted States



Anim Sets Split



Anim Set File

```
(define-anim-set *longgun-soldier-medium-anim-set*

  ...

  ( combat-run                    <- sol-med-lg-gunout-run-d-fw )
  ( combat-run-look-left-right    <- sol-med-lg-gunout-run-d-fw-look-left-right )
  ( combat-run-look-up-down       <- sol-med-lg-gunout-run-d-fw-look-up-down )

  ...

)
```


An anim-set file is much easier to parse mentally, even in scheme. With source animation names on the left and translated on the right.

Anim Set File

```
(define-anim-set *longgun-soldier-medium-anim-set*

  ...

  ( combat-run                  <- sol-med-lg-gunout-run-d-fw )
  ( combat-run-look-left-right  <- sol-med-lg-gunout-run-d-fw-look-left-right )
  ( combat-run-look-up-down     <- sol-med-lg-gunout-run-d-fw-look-up-down )

  ...

)
```


An anim-set file is much easier to parse mentally, even in scheme. With source animation names on the left and translated on the right.

Anim Set File

```
(define-anim-set *longgun-soldier-medium-anim-set*

  ...

  ( combat-run                  <- sol-med-lg-gunout-run-d-fw )
  ( combat-run-look-left-right  <- sol-med-lg-gunout-run-d-fw-look-left-right )
  ( combat-run-look-up-down     <- sol-med-lg-gunout-run-d-fw-look-up-down )

  ...

)
```


An anim-set file is much easier to parse mentally, even in scheme. With source animation names on the left and translated on the right.

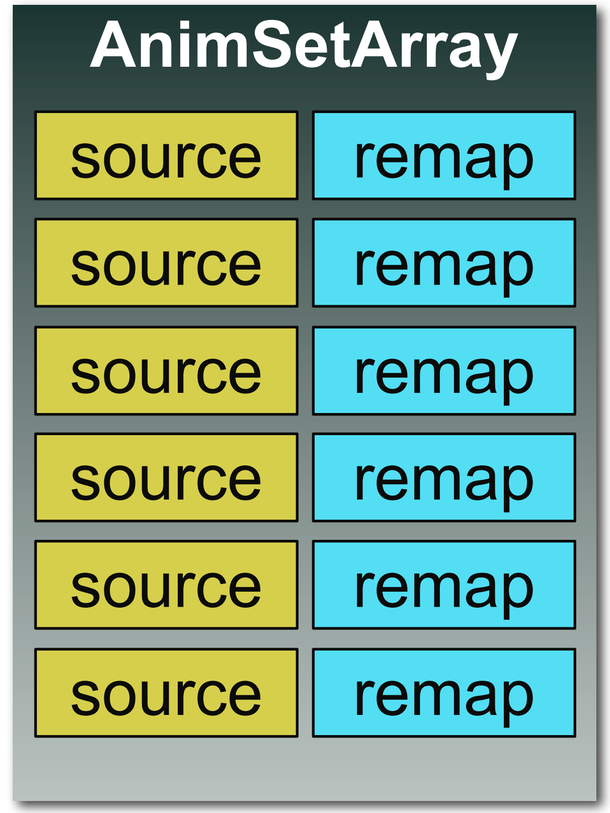
Anim Sets

- Simple structure…

```
struct AnimSetEntry
{
    StringId m_sourceId;
    StringId m_remapId;
};

struct AnimSet
{
    const AnimSetEntry* m_animSetArray;
    I32 m_animSetArrayCount;
};
```

- Binary search queries



Anim set data is quite simple. A single entry is a key-value pair of string hashes (what we call StringIds).

Collectively the array of entries is sorted by key.

This results in an equally simple binary search function for querying

Memory Gains

UNCHARTED: DRAKE'S FORTUNE

anim-elena
anim-sullivan
anim-pirate
...

2.7 MB

UNCHARTED 2: AMONG THIEVES™

anim-npc
anim-sets

347 KB

In Uncharted 1 we had individual animation bin files (containing all the states, trees, etc.) which in all consumed 2.27 MEGS of memory to have loaded.

Moving to Uncharted 2 we used only one animation bin file but multiple individual anim set files.

By simply removing the redundant animation states we saw a drastic drop in memory consumption: ~347KB to have animation data for all characters!

Memory Gains

anim-npc — 289 KB

anim-sets — 58 KB

347 KB

Production Gains

- Files simple enough for artists to edit
  - Decoupled programmer from workflow
- Supports dynamic reloading
- Fast iteration

Problems for Programmers

- One Tree To Rule Them All
  - Fixed behaviors across character types
  - Had to build for the most expensive case
  - Changing trees hugely impactful on animators

- There are problems though…

- Using one animation bin file for all characters constrains all to the same state graph & tree structure
  - The cajoling we did before meant giving up flexibility in our animation trees across different character types

- Since we had one state/tree definition list for all character types, creating a character became filling out a master list, which grew to a significant size- cumbersome!
- Animators were often required to make animations for slots not needed for the desired look
  - For example a villager in the background might not need the same tree fidelity as Chloe, but they were the same.

Problems for Animators

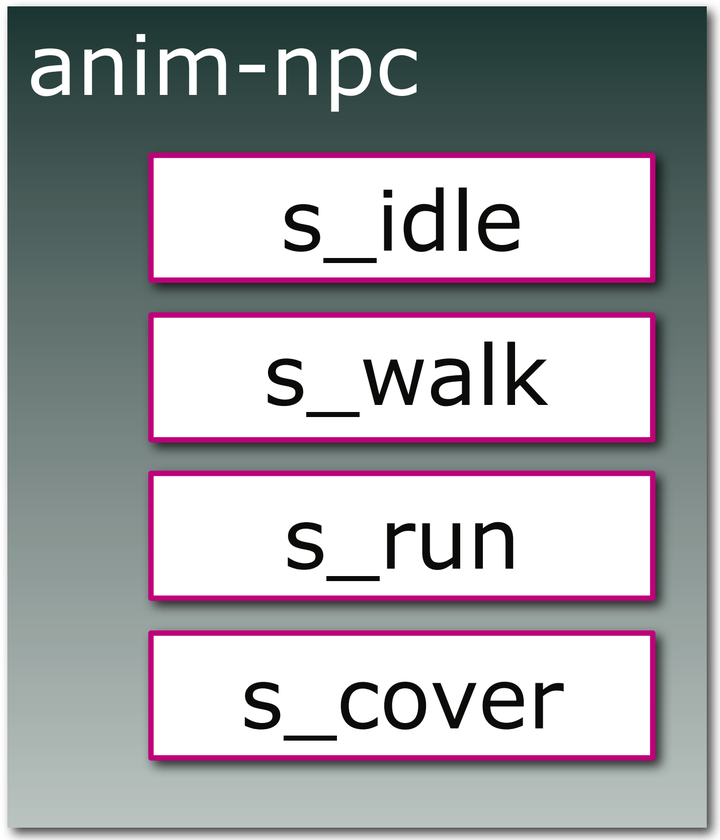
- Creating a character became “Fill this list”
  - Became a big list
- Characters with small variations disproportionality time consuming

- There are problems though…

- Using one animation bin file for all characters constrains all to the same state graph & tree structure
  - The cajoling we did before meant giving up flexibility in our animation trees across different character types

- Since we had one state/tree definition list for all character types, creating a character became filling out a master list, which grew to a significant size- cumbersome!
- Animators were often required to make animations for slots not needed for the desired look
  - For example a villager in the background might not need the same tree fidelity as Chloe, but they were the same.

Tree Variations

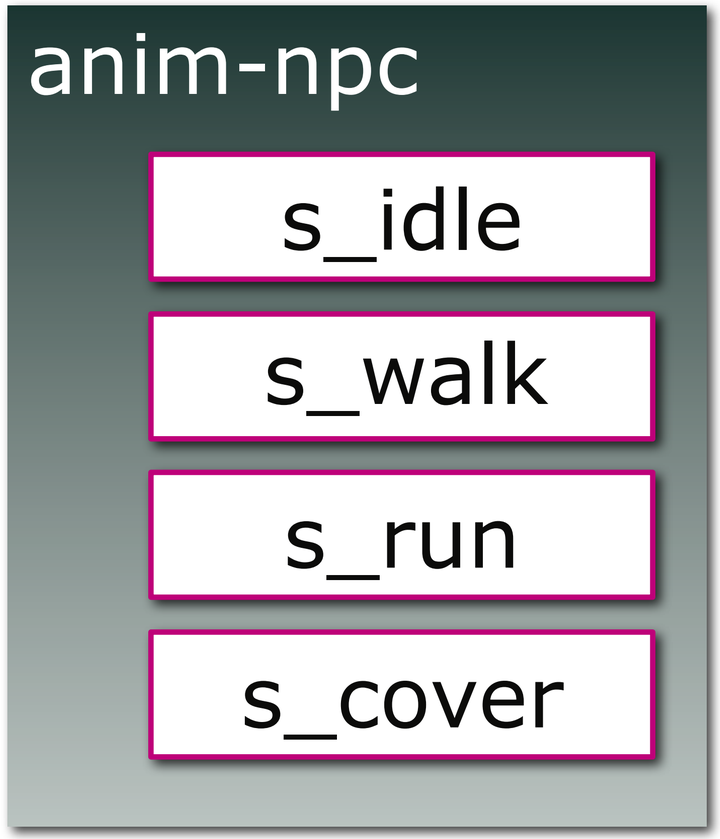


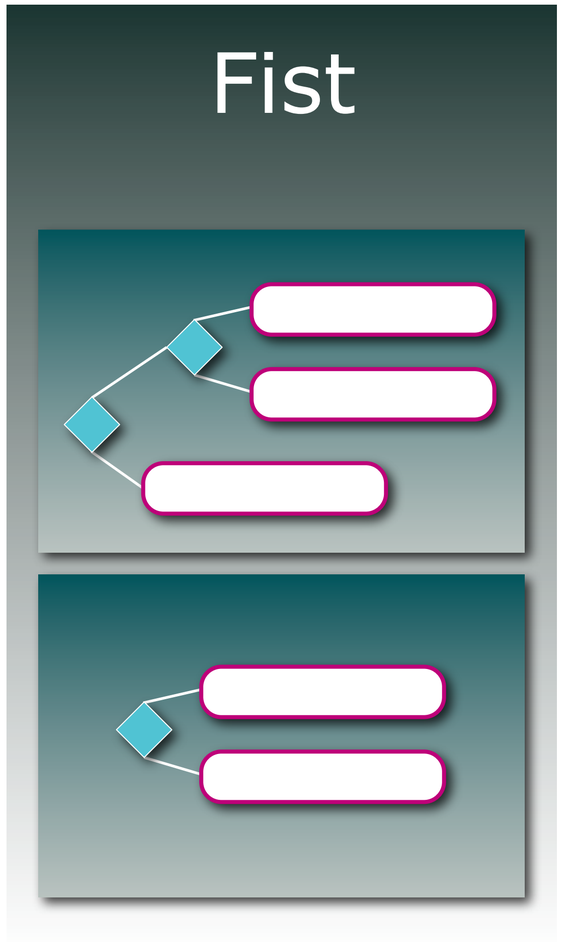
We needed to be able to change the trees themselves per character

While a fist soldier might need a less complicated animation tree

a pistol char would be more demanding

Tree Variations



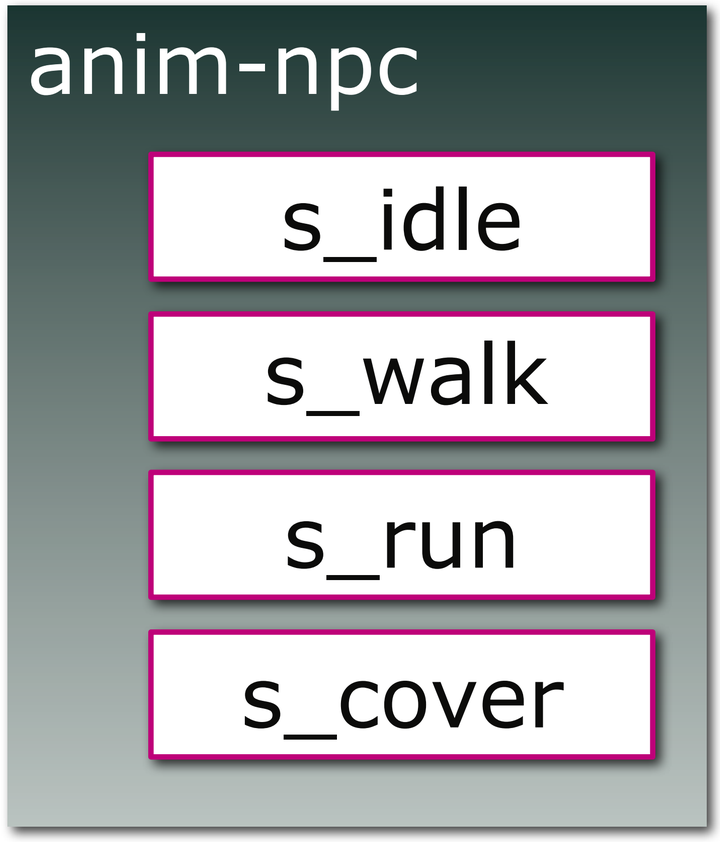


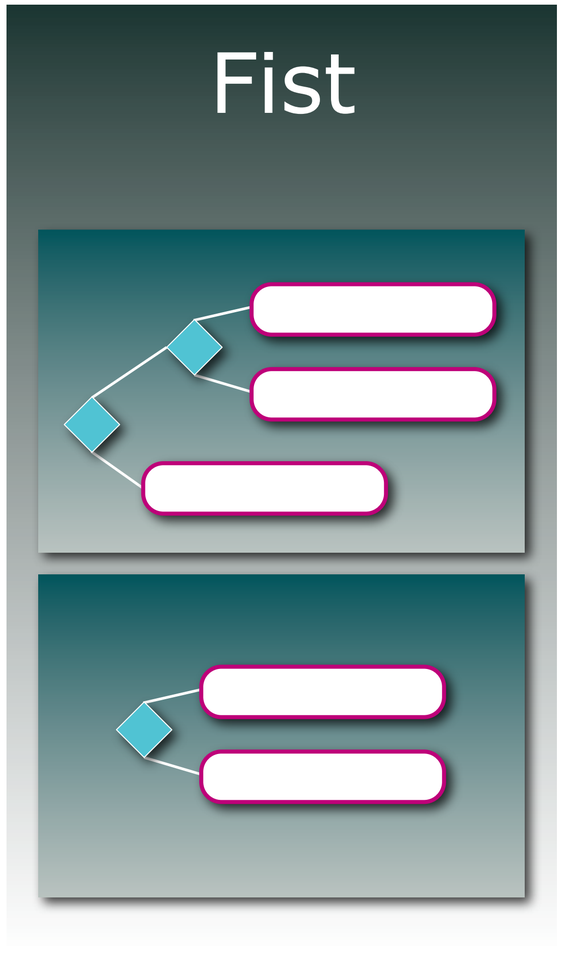
We needed to be able to change the trees themselves per character

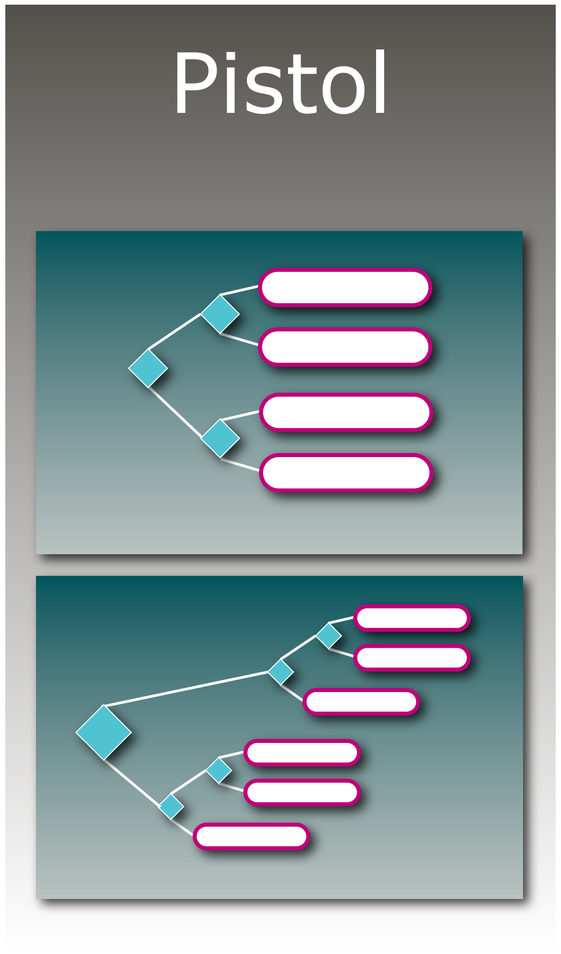
While a fist soldier might need a less complicated animation tree

a pistol char would be more demanding

Tree Variations







We needed to be able to change the trees themselves per character

While a fist soldier might need a less complicated animation tree

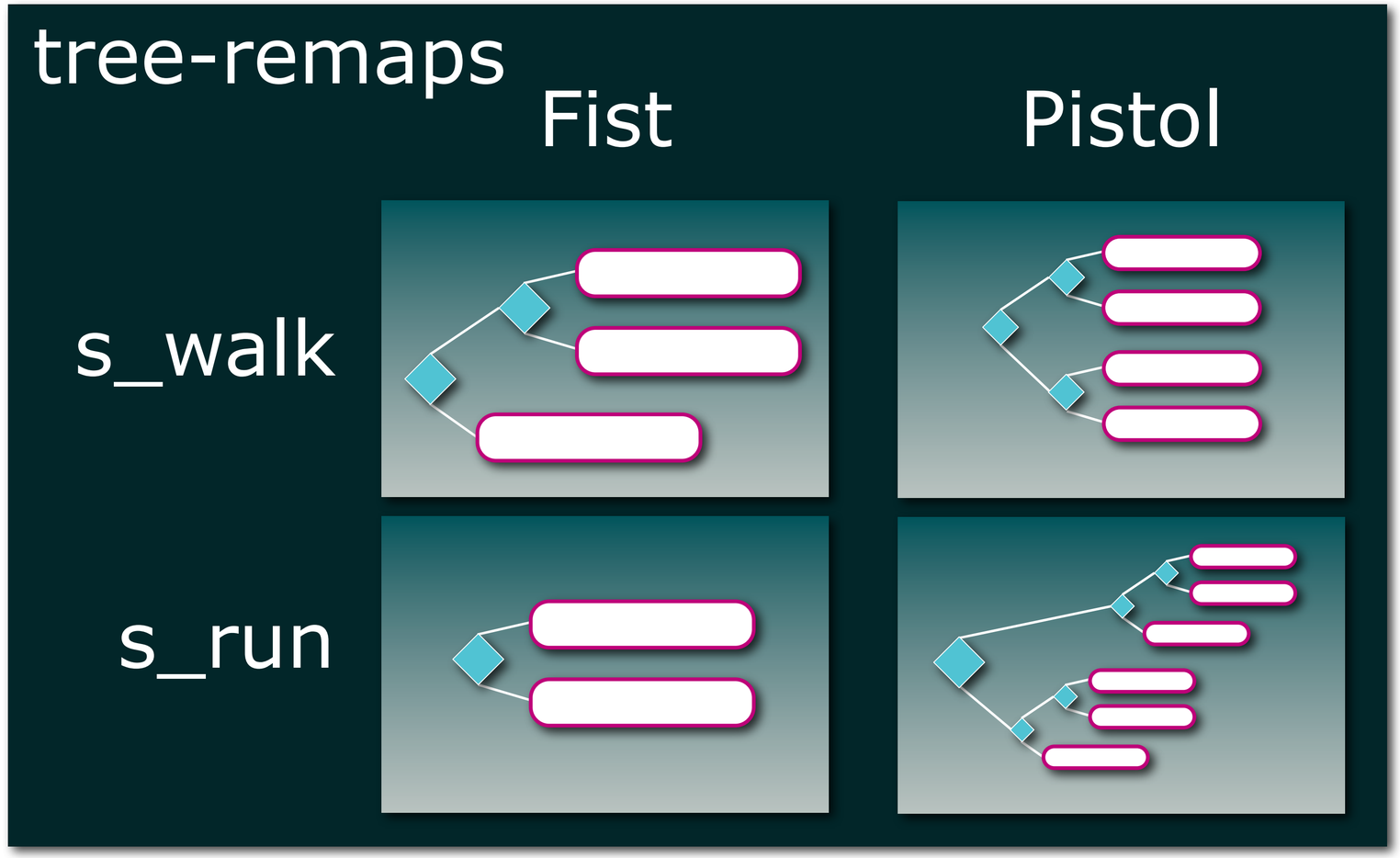
a pistol char would be more demanding

Tree Remaps

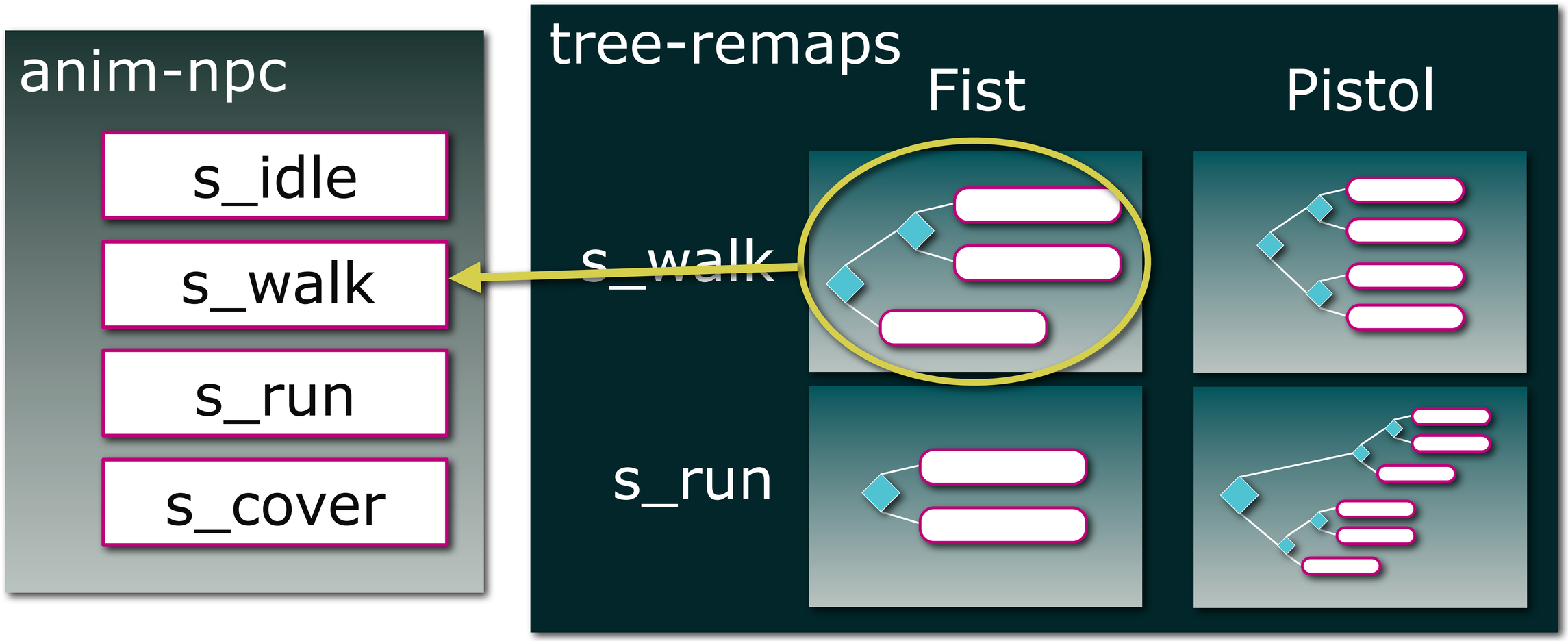
anim-npc

- s_idle
- s_walk
- s_run
- s_cover

tree-remaps



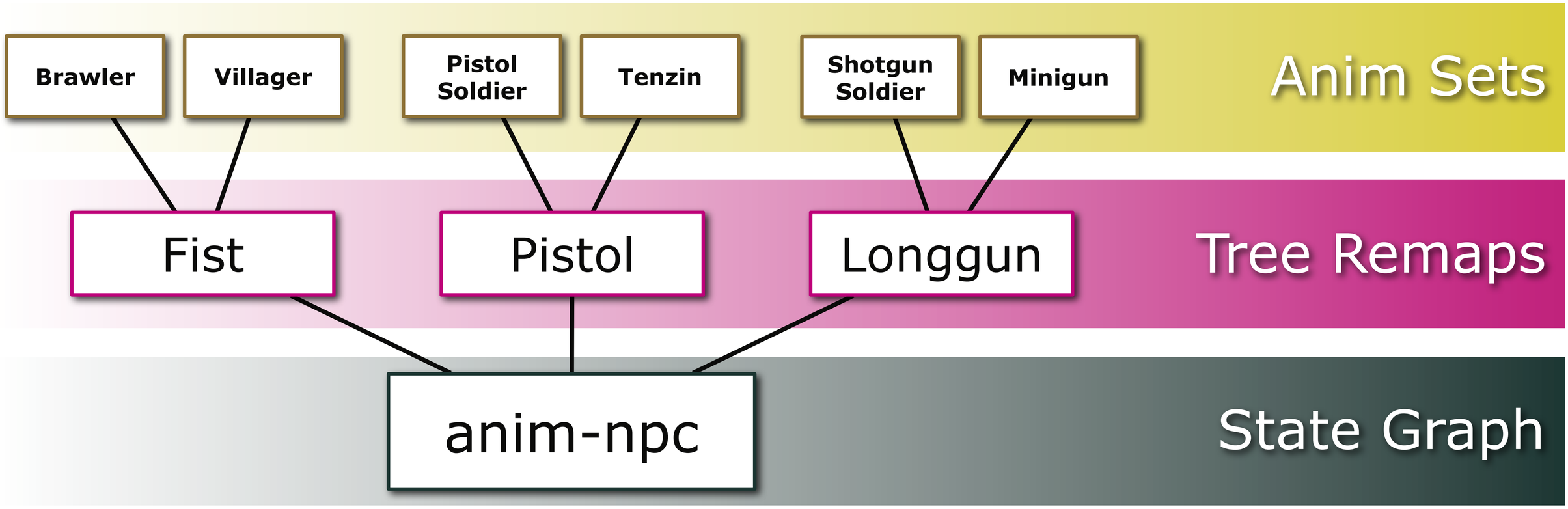
Tree Remaps


anim-npc
s_idle
s_walk
s_run
s_cover
tree-remaps
Fist
Pistol
s_walk
s_run

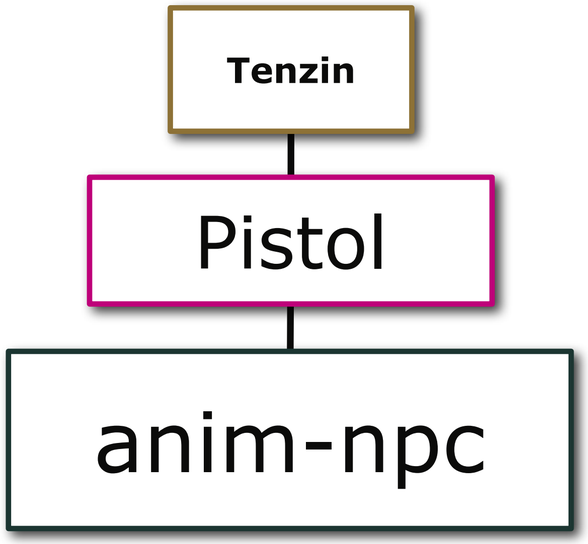
TREE-REMAPS

With tree remaps we can now have a lot more variety for a single animation state (in this case ‘idle’), be it a simple looping idle animation, a base animation with a look at tree, or a complicated aim-look tree

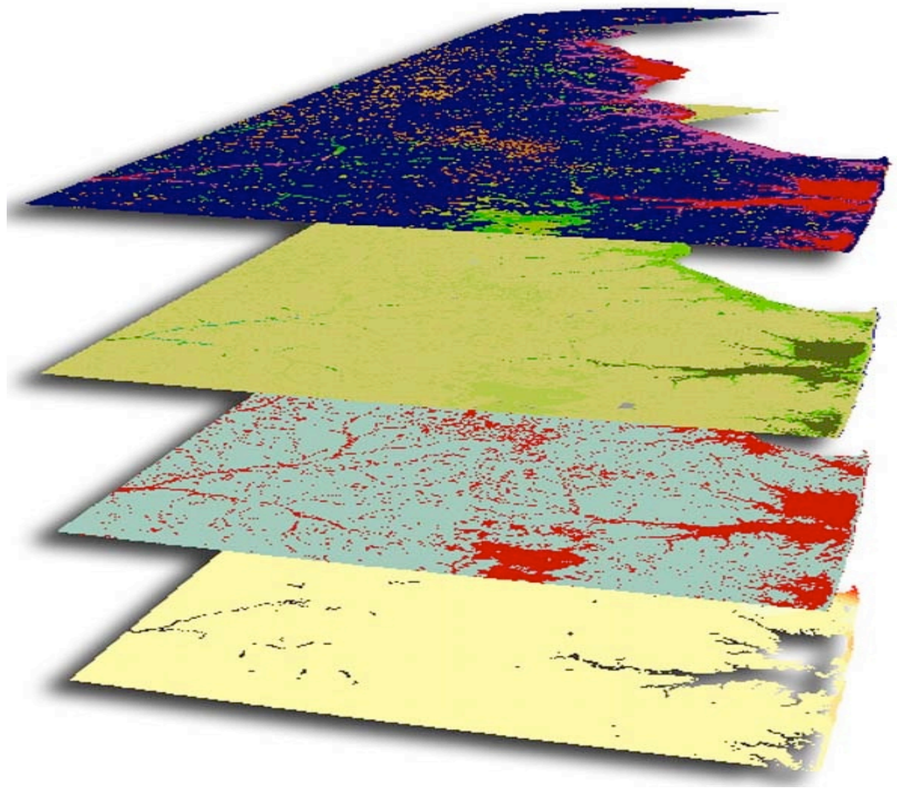
Data Source Hierarchy



Data Source Hierarchy

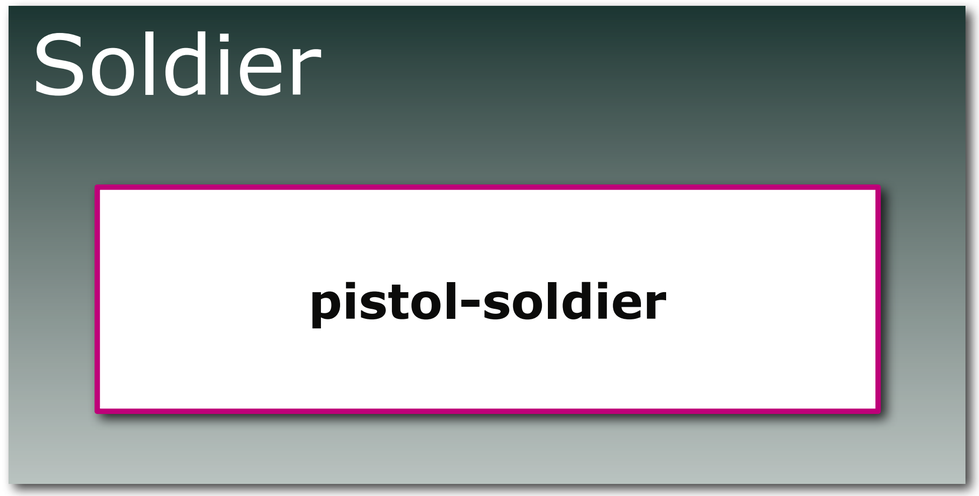


Layered Remaps

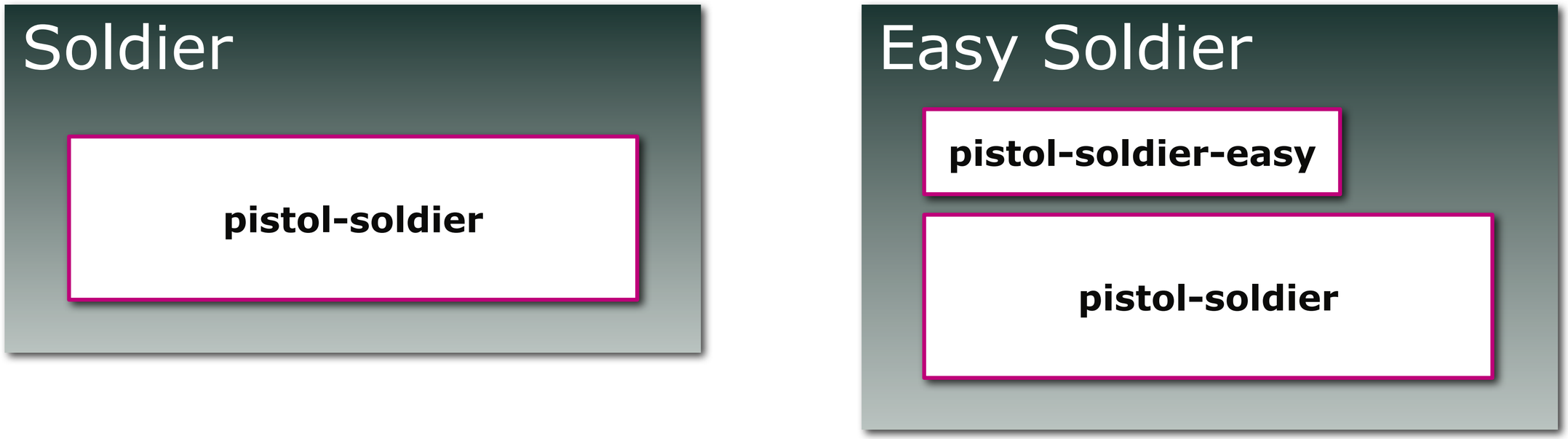


Because we are resolving both animation trees and animation names in our runtime we can do some interesting things: namely layering.

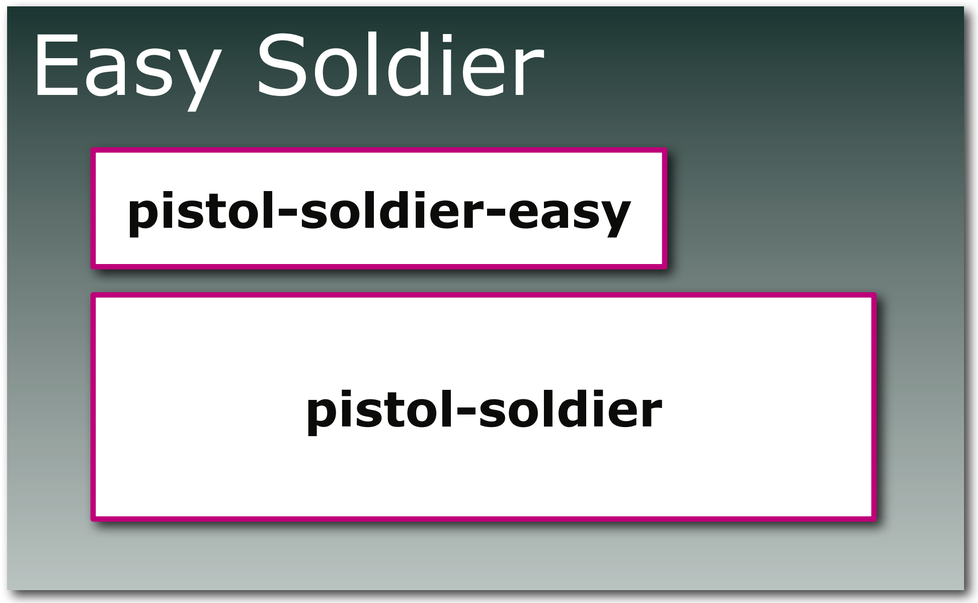
Layered Remaps



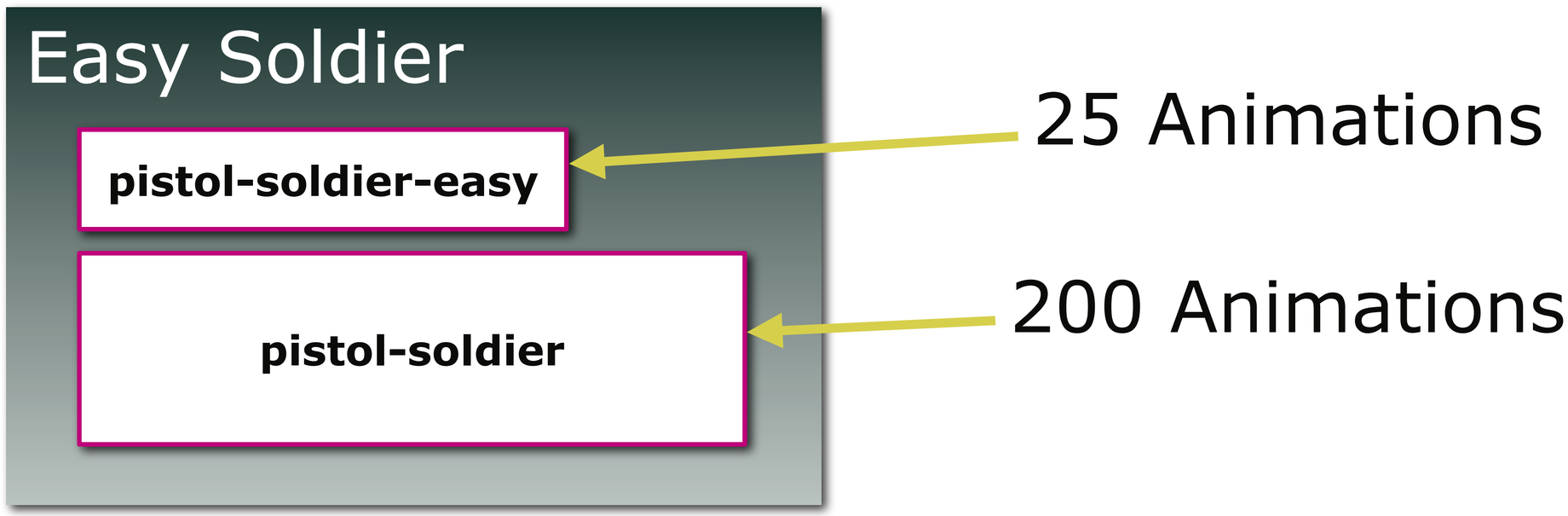
Layered Remaps



Layered Remaps



Layered Remaps



Layered Anim Sets

This lets us define characters in terms of deltas.

we can compose a character from multiple pieces and layering lets us achieve that

Layered Anim Sets

the animators took well to the layering approach because artists respond well to explanations of “it works like photoshop”

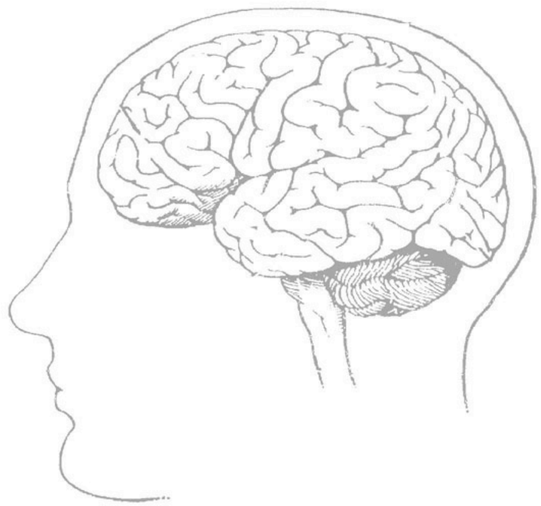
Remap sources

Deciding which remaps we pushed could potentially come a multitude of sources:

- AI logic could add or remove them to match behavior changes
- Level scripts could do so to follow high level gameplay flow
- Designers could attach them to entity spawners in the level

Remap sources

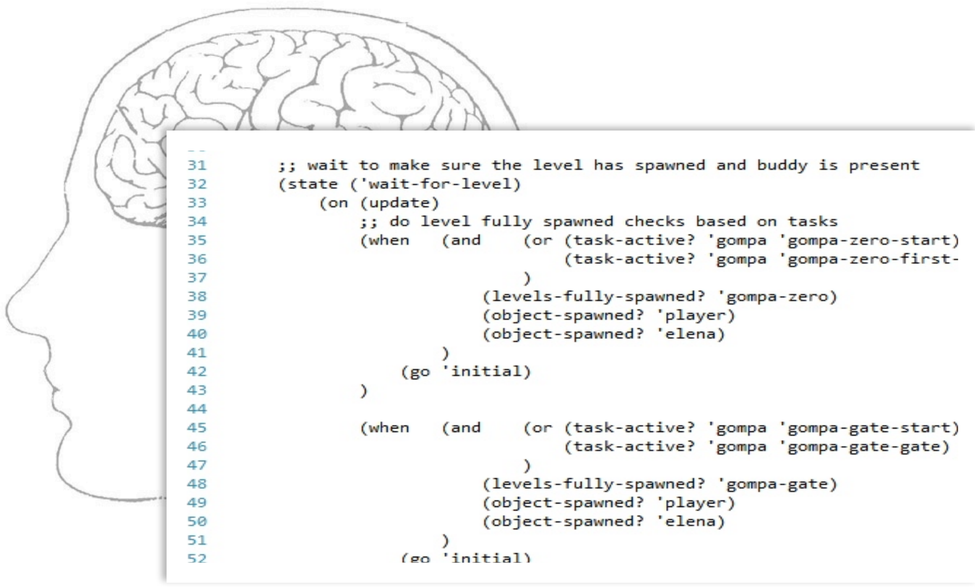
- AI



Deciding which remaps we pushed could potentially come a multitude of sources:
- AI logic could add or remove them to match behavior changes
- Level scripts could do so to follow high level gameplay flow
- Designers could attach them to entity spawners in the level

Remap sources

- AI
- Level Scripts

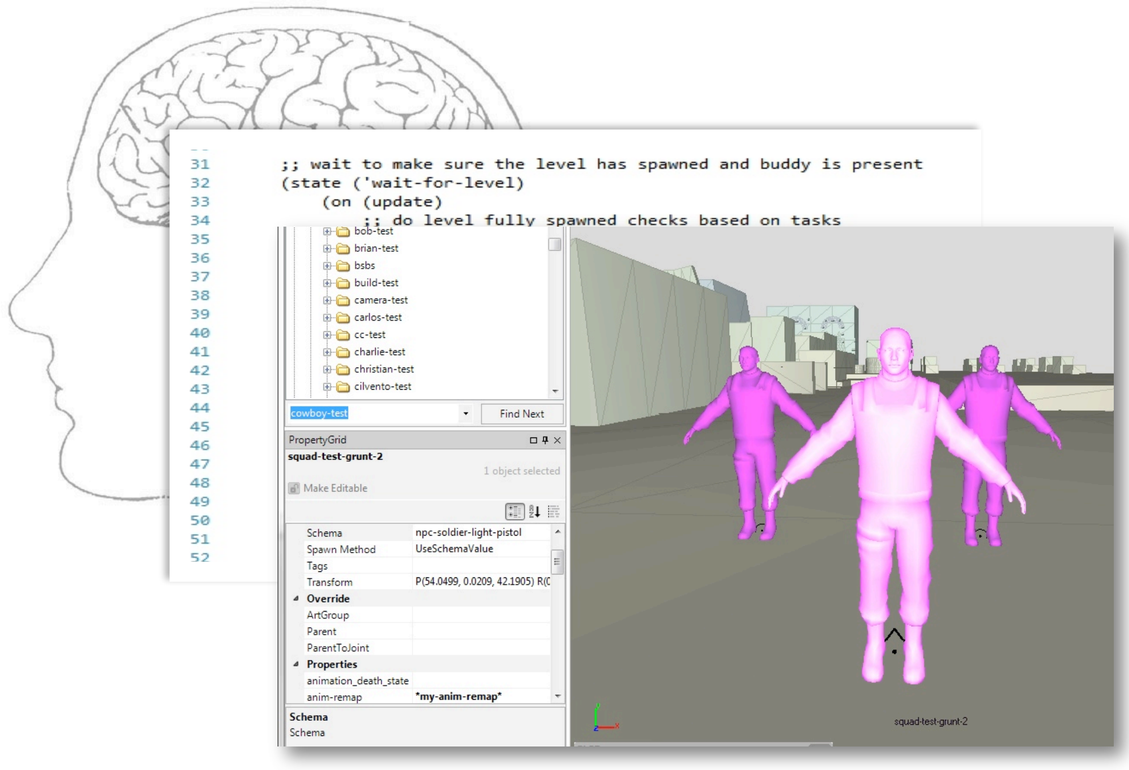


```
31      ;; wait to make sure the level has spawned and buddy is present
32      (state ('wait-for-level)
33          (on (update)
34              ;; do level fully spawned checks based on tasks
35              (when   (and    (or (task-active? 'gompa 'gompa-zero-start)
36                                  (task-active? 'gompa 'gompa-zero-first-
37                              )
38                          (levels-fully-spawned? 'gompa-zero)
39                          (object-spawned? 'player)
40                          (object-spawned? 'elena)
41                      )
42                  (go 'initial)
43              )
44
45              (when   (and    (or (task-active? 'gompa 'gompa-gate-start)
46                                  (task-active? 'gompa 'gompa-gate-gate)
47                              )
48                          (levels-fully-spawned? 'gompa-gate)
49                          (object-spawned? 'player)
50                          (object-spawned? 'elena)
51                      )
52                  (go 'initial)
```


Deciding which remaps we pushed could potentially come a multitude of sources:
- AI logic could add or remove them to match behavior changes
- Level scripts could do so to follow high level gameplay flow
- Designers could attach them to entity spawners in the level

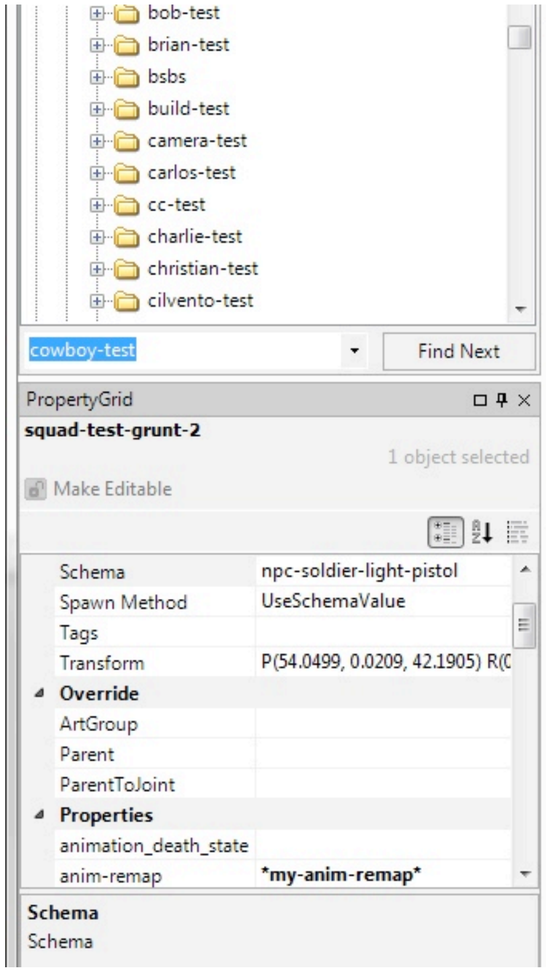
Remap sources

- AI
- Level Scripts
- Spawner Property



Deciding which remaps we pushed could potentially come a multitude of sources:
- AI logic could add or remove them to match behavior changes
- Level scripts could do so to follow high level gameplay flow
- Designers could attach them to entity spawners in the level

Remap sources



Being able to add remaps on spawners ended up being a particularly big win for us because it made for really easy “spot-fixing” of characters as needed for levels.

An animator could create a small delta anim-set file to fixup the animations required for the situation created by the designer. The designer could then place it on the spawner in our level editor and see the result right away.

All without having to bother a programmer – epic win!

VARIATIONS

The same principle of defining characters with delta sets also allows us to tackle the problem of creating variations in a nice way.

Besides being able to configure variations of characters with delta animations, we could create variations within particular character types

Instance Variations

We created what we called “instance variations”

Instance variations were defined as a collection of anim-sets and a character would pick one of these collections at it’s birth and apply the set.

This works well because a walk animation might require different aim or look-at animations to go with it, and it’s important to vary them as a group as to not make the character look disjointed.

Instance Variations

A	B
walk-a	walk-b
walk-a-aim-lr	walk-b-aim-lr
walk-a-aim-ud	walk-b-aim-ud

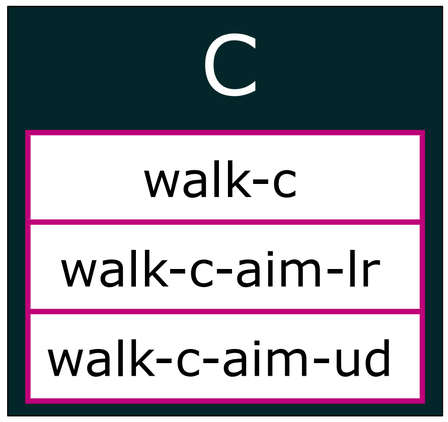
C	D
walk-c	walk-d
walk-c-aim-lr	walk-d-aim-lr
walk-c-aim-ud	walk-d-aim-ud

We created what we called “instance variations”

Instance variations were defined as a collection of anim-sets and a character would pick one of these collections at it’s birth and apply the set.

This works well because a walk animation might require different aim or look-at animations to go with it, and it’s important to vary them as a group as to not make the character look disjointed.

Instance Variations



We created what we called “instance variations”

Instance variations were defined as a collection of anim-sets and a character would pick one of these collections at it’s birth and apply the set.

This works well because a walk animation might require different aim or look-at animations to go with it, and it’s important to vary them as a group as to not make the character look disjointed.

Costs

With so much of an animation state in potential flux we needed a way to work and animate a character reliably and to avoid paying unnecessary re-resolve costs.

We needed to resolve once and store it for later. Our initial solution was to pad our DC animation state and animation node structures with fields for resolved values; we would then make a copy of this structure and store it per active animation state instance.

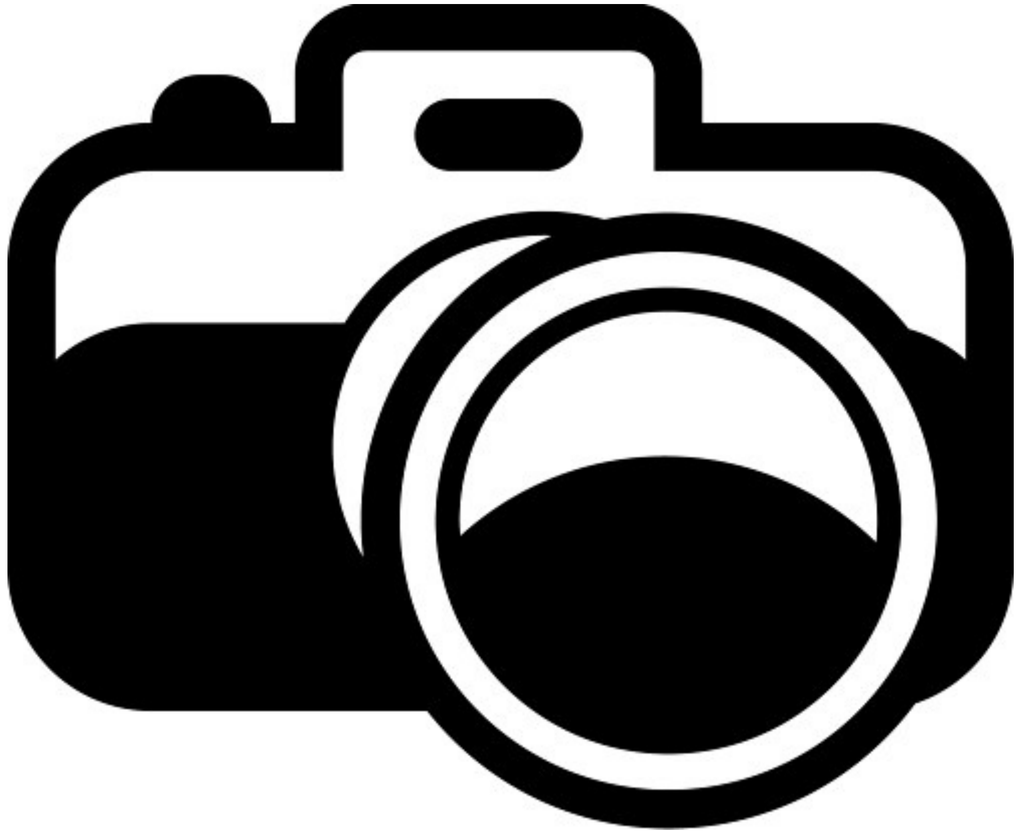
This is obviously wasteful for both increasing the size of our bin files and duplicating constant data in memory!

Costs

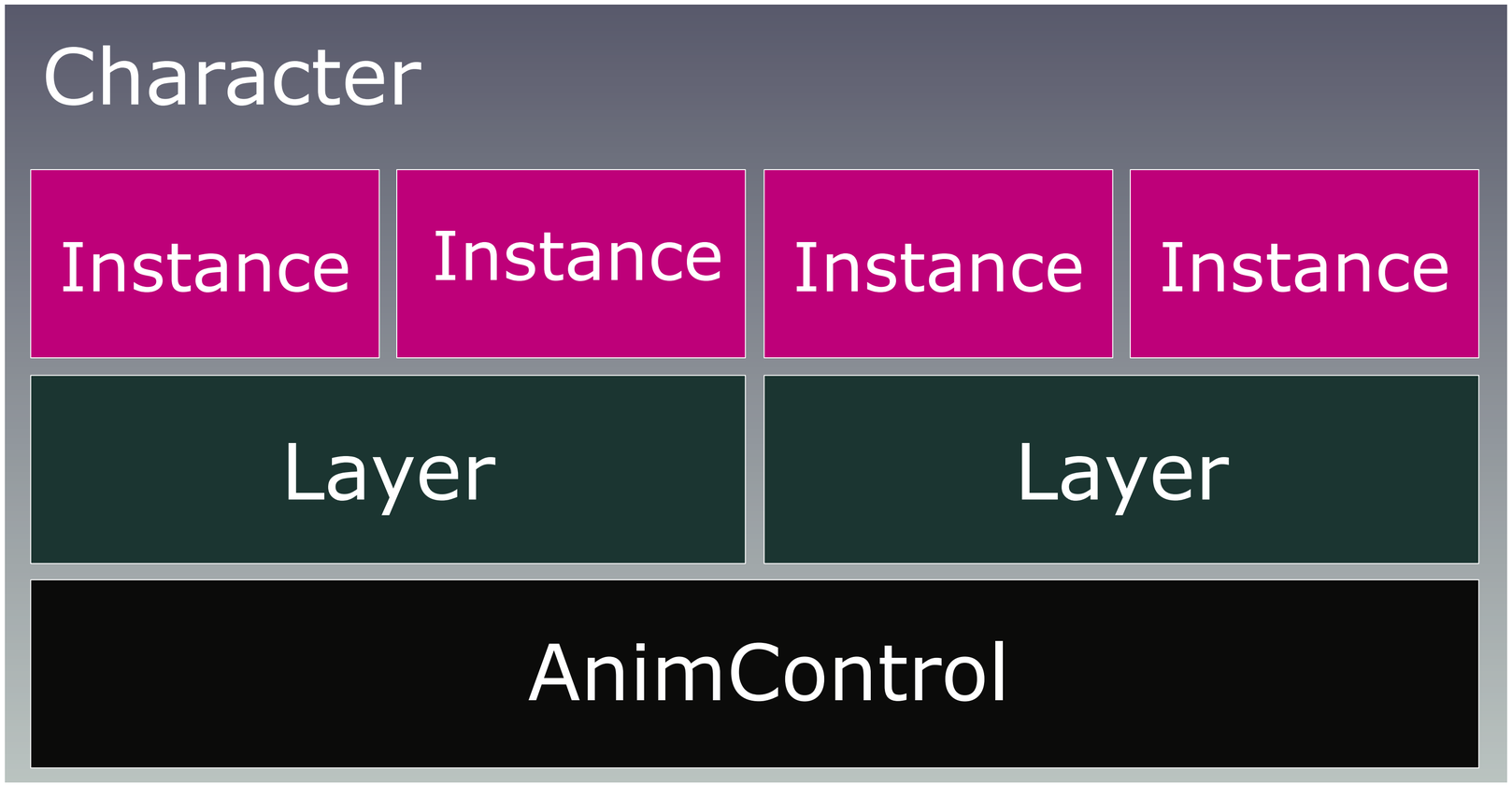
- Many Translation Layers
- Re-evaluation Expensive
- Time Coherency Extremely Problematic

Solution

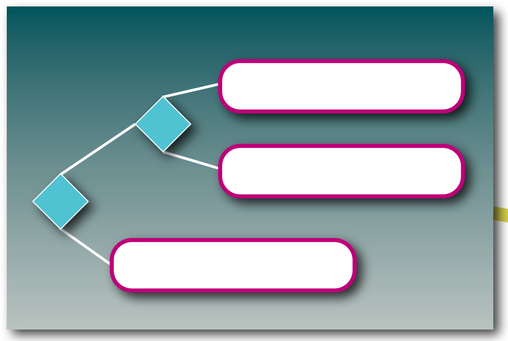
Capture
Instantaneous
Character State

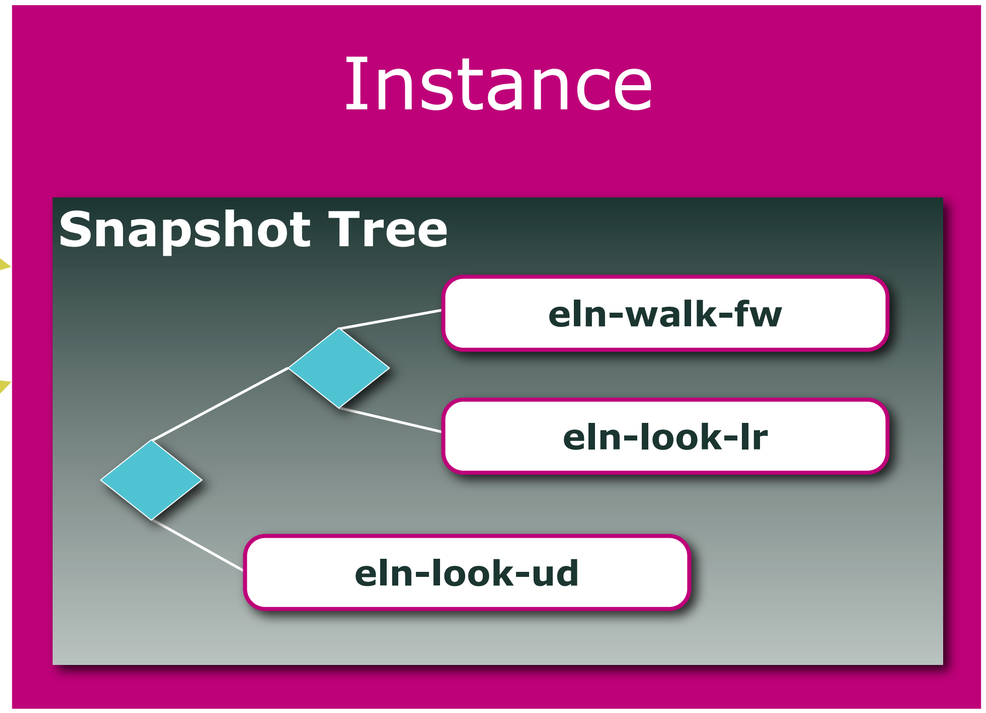


Runtime


Character
Instance
Instance
Instance
Instance
Layer
Layer
AnimControl

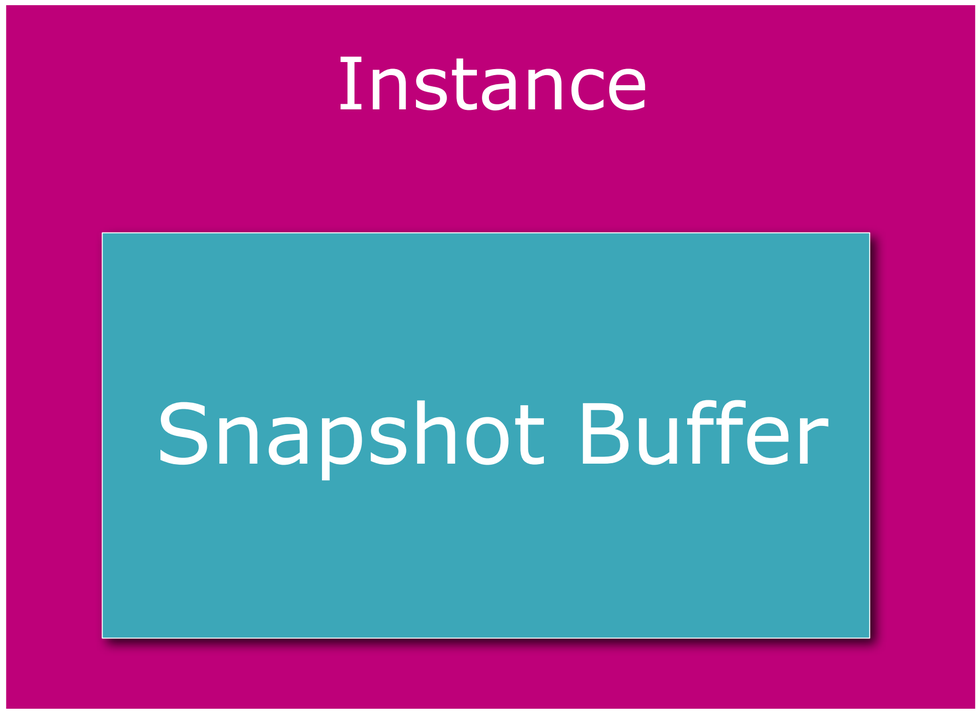
SNAPSHOTTING





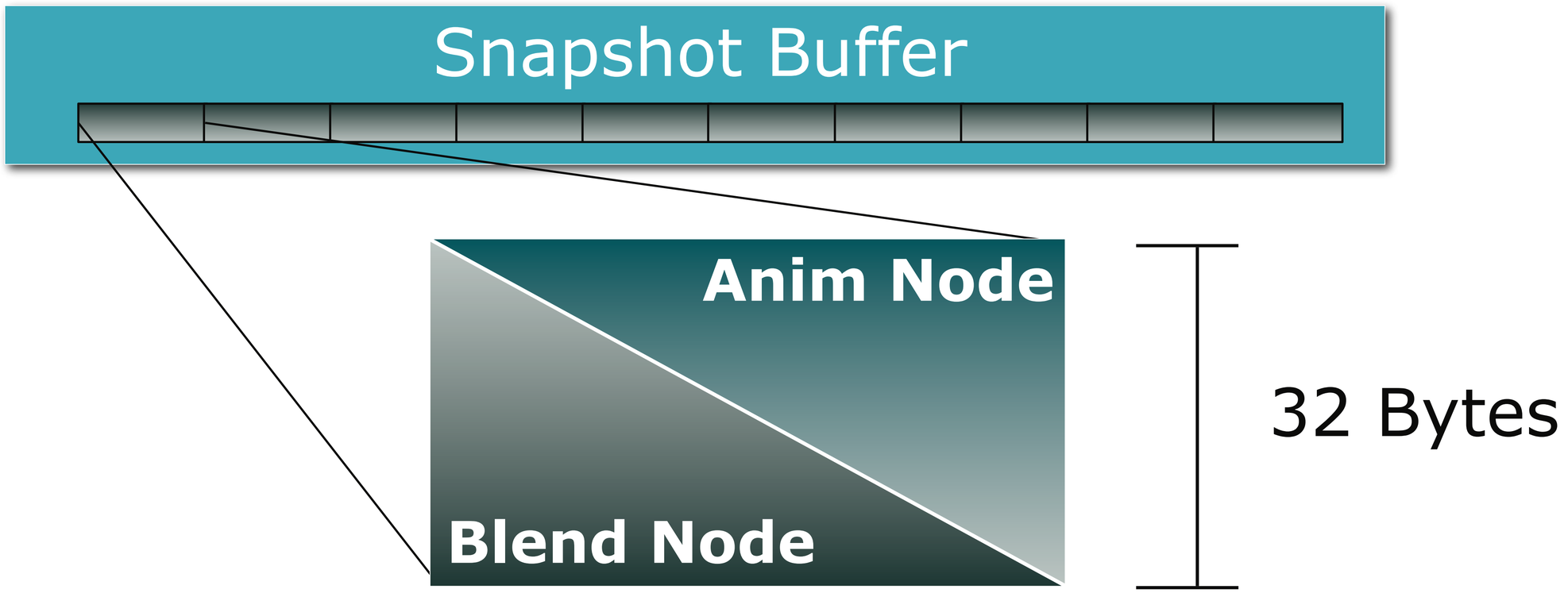
When we construct an animation state instance, we record all translated animation names, trees and any other dynamically resolved data.

Snapshotting



Instance doesn't translates on entry
Stores translated information

SNAPSHOTS



When we construct a snapshot tree we fill out a pre-allocated buffer with snapshot “nodes”

(A node is just a union of the snapshot blend and snapshot animation structures)

The snapshotting itself is simply accomplished by walking the DC tree, computing all the translated values, and storing them.

Constructing a new animation state instance takes ~100 microseconds spent constructing our snapshot. This is done any time a character changes animation states.

Snapshot Nodes

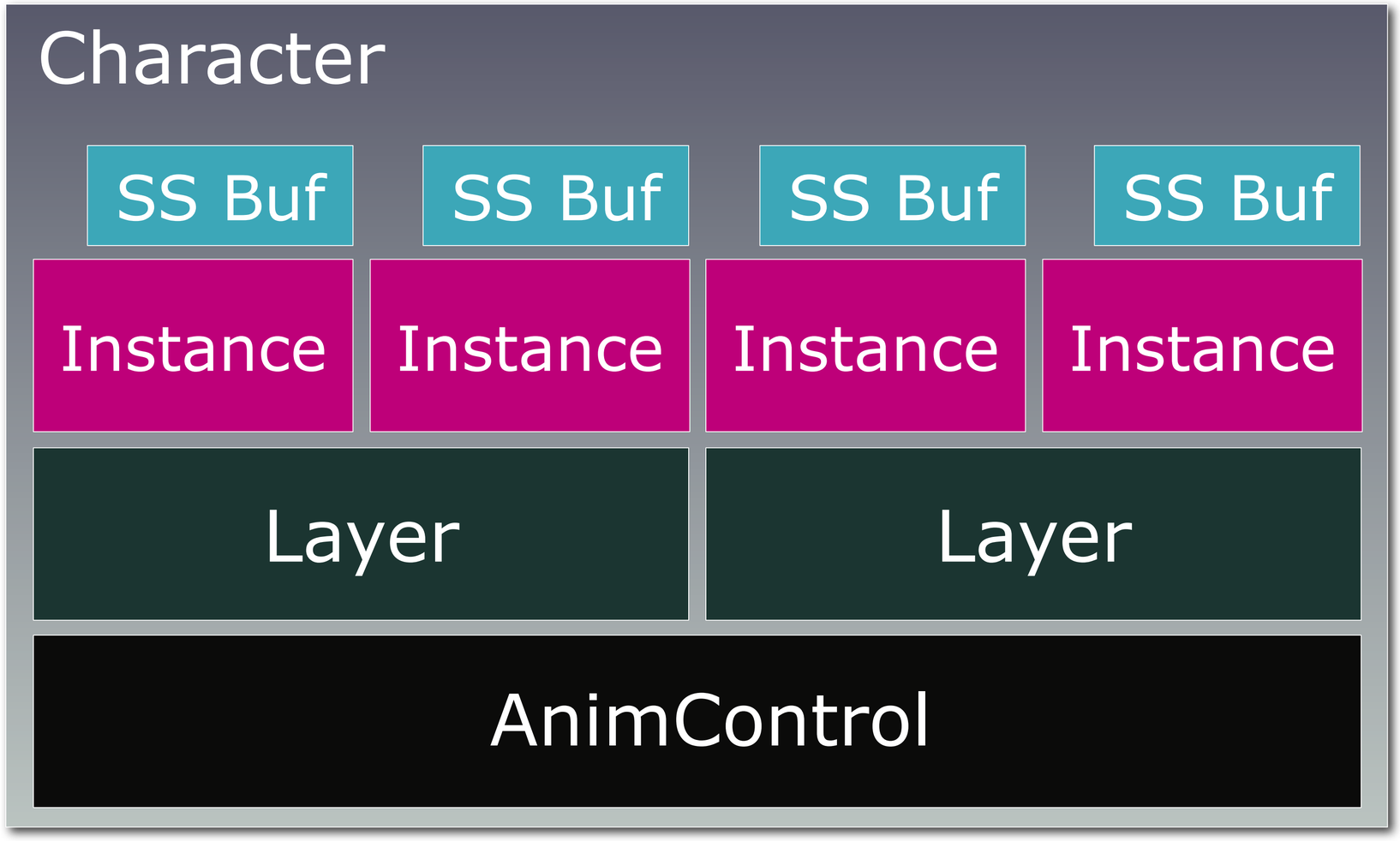
Anim Node

- Anim Name
- Skeleton
- Art Resource

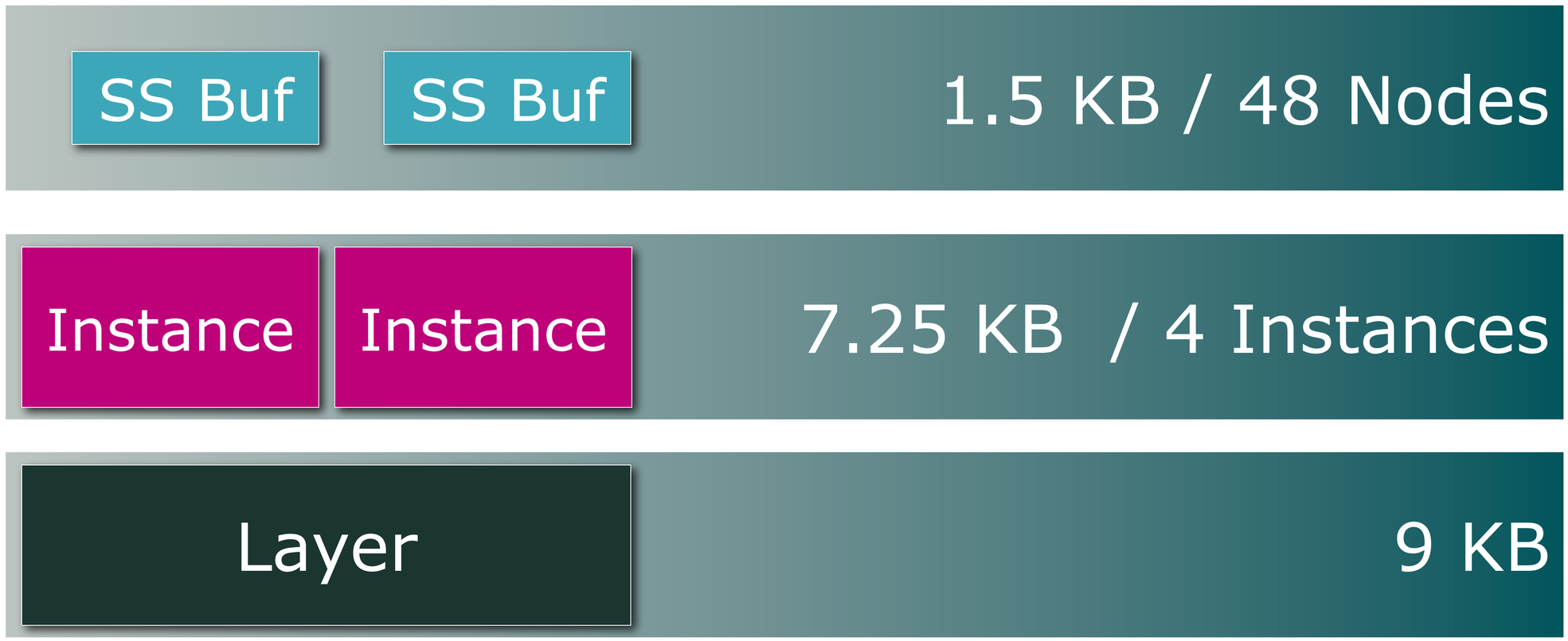
Blend Node

- Blend Factor
- Tree Breadth
- Child Indices

Snapshot Memory



Snapshot Memory



How much memory do we spend on snapshotting?

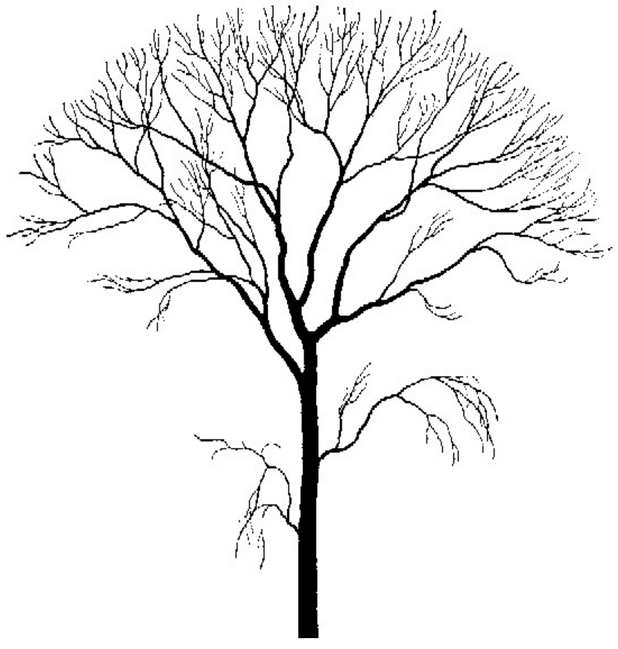
An NPC will typically contain only one AnimStateLayer (that is, an animation layer that uses our DC state graph system to construct blends). The total cost of allocating one of these layers weighs in at a little over nine KB.

For each state layer we allocate a certain number of AnimStateInstances for the instance pool. This will be the maximum number of states we can be blending together at a single point in time.

For NPC's this number is 4, and a single instance weighs in at around 1.8KB, bringing the total allocation cost for instances to 7.25 KB for the entire layer.

Within a single instance we allocate 1.5KB for an array of 48 snapshot nodes (32 bytes each). This is the buffer needed for a single instance to completely snapshot a state. For

SNAPSHOTTING

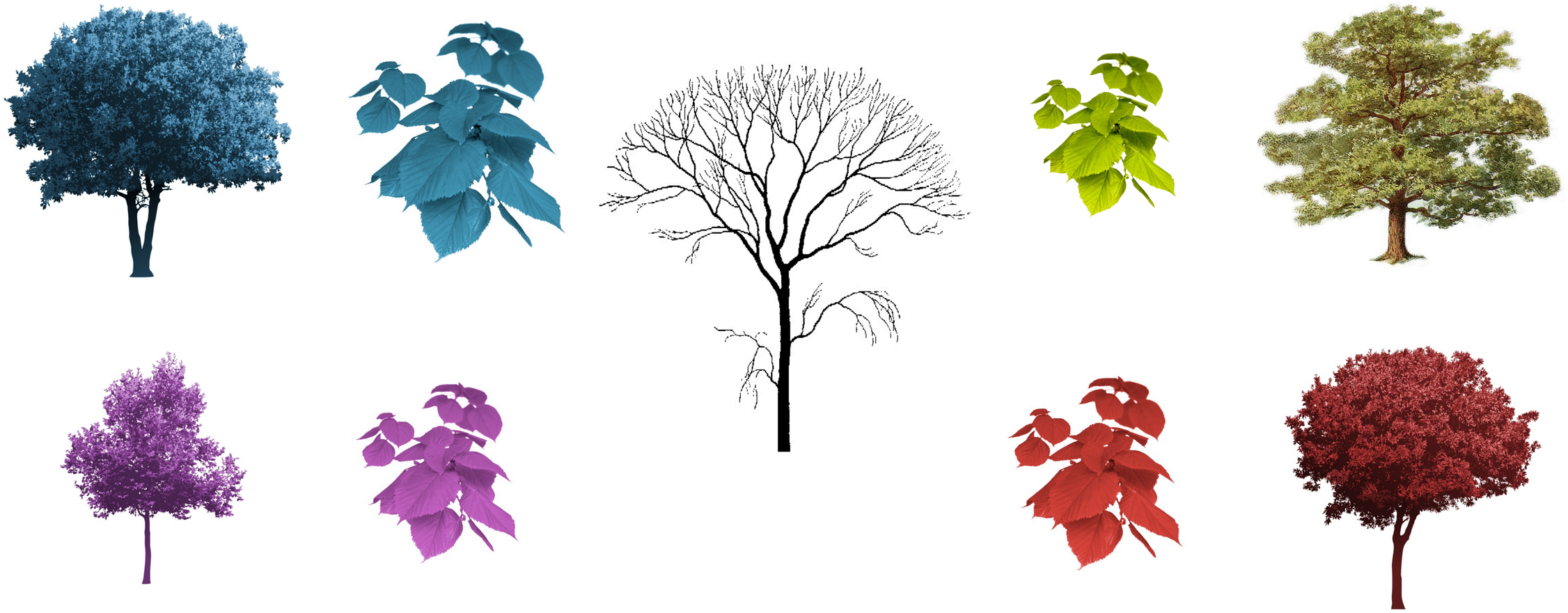


Base Anim State

Anim Sets Tree Remaps

Snapshot

SNAPSHOTTING



PROBLEMS

- Loss of Clarity
  - No quick way to see how a state will animate
- Best compensated with runtime debugging aids

PROBLEMS

- Could no longer evaluate arbitrary animation states
  - Predicting movement for things like jumps
- Fixed by ability to create arbitrary snapshots
  - Typically on the stack with a stack buffer

Evolution

Evolution

Fill in Animations

Construct Trees

Add Fidelity

Create Variety

Build State Graph

Evolution

Structure

Style

Fill in Animations

Construct Trees

Add Fidelity

Create Variety

Build State Graph

Evolution

Structure

Style

Construct Trees

Add Fidelity

Create Variety

Build State Graph

Fill in Animations

Evolution

Structure

Style

Construct Trees

Fill in Animations

Add Fidelity

Create Variety

Build State Graph

Evolution

Structure

Style

Construct Trees

Fill in Animations

Add Fidelity

Create Variety

Build State Graph

Evolution

Structure

Style

Construct Trees

Build State Graph

Fill in Animations

Add Fidelity

Create Variety

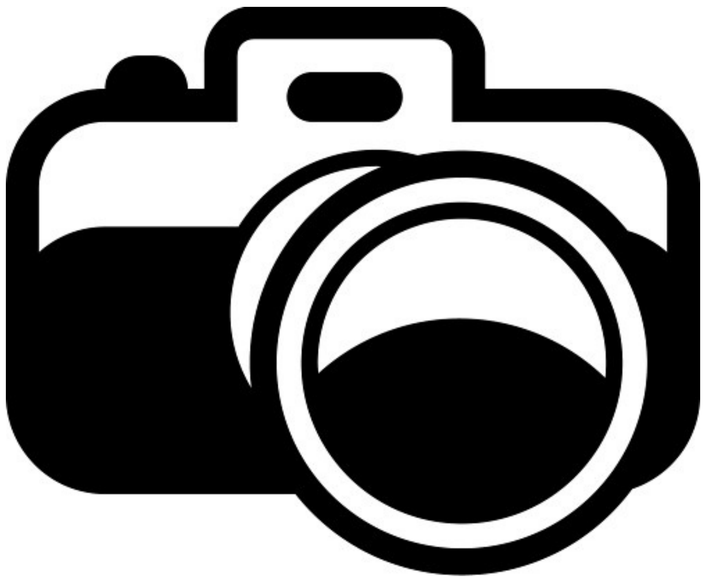
Evolution

Structure

Style

Construct Trees

Build State Graph



Fill in Animations

Add Fidelity

Create Variety

Whats Next

- Better Tools
- Transitions
- Get it running on the SPUs!

QUESTIONS

john_bellomy@naughtydog.com

P.S.: We’re Hiring ! jobs@naughtydog.com