CS 4204 Computer Graphics

Animation Transition Blending and State Machine

Yong Cao

Virginia Tech

Reference: UC San Diego, CSE 169 Animation course slides. http://graphics.ucsd.edu/courses/cse169_w06/

Blending & Sequencing

- *Now that we understand how a character skeleton works and how to manipulate animation data, we can edit and play back simple animation*
- *The subject of blending and sequencing encompasses a higher level of animation playback, involving constructing the final pose out of a combination of various inputs*

Blending & Sequencing

- *Most areas of computer animation have been pioneered by the research and special effects industries*
- *Blending and sequencing, however, is one area where video games have made a lot of real progress in this area towards achieving interactively controllable and AI characters in complex environments…*
- *The special effects industry is using some game related technology more and more (battle scenes in Lord of the Rings…)*

Animation Playback

Poses

- *A pose is an array of values that maps to a skeleton*
- *If the skeleton contains only simple independent DOFs, the pose can just be an array of floats*
- *If the skeleton contains quaternions or other complex coupled DOFs, they may require special handling by higher level code*
- *Therefore, for generality, we will assume that a pose contains both an array of M≥0 floats and an additional array of N≥0 quaternions*

$$\Phi = \left[\phi_0 \cdots \phi_{M-1} \quad \mathbf{q}_0 \cdots \mathbf{q}_{N-1}\right]$$

Animation Clip

- *Remember that the AnimationClip stores an array of channels for a particular animation (or it could store the data as an array of poses…)*
- *This should be treated as constant data, especially in situations where multiple animating characters may simultaneously need to access the animation (at different time values)*
- *For playback, animation is accessed as a pose. Evaluation requires looping through each channel.*

```
class AnimationClip {
        void Evaluate(float time,Pose &p);
}
```

Animation Player

- *We need something that 'plays' an animation. We will call it an animation player*
- *At it's simplest, an animation player would store a* **AnimationClip*, Skeleton*,** *and a float time*
- *As an active component, it would require some sort of Update() function*
- *This update would increment the time, evaluate the animation, and then pose the rig*
- *However, for reasons we will see later, we will leave out the* **Skeleton*** *and just have the player generate and output a Pose*

Animation Player

```
class AnimationPlayer {
        float Time;
        AnimationClip *Anim;
        Pose P;
public:
        void SetClip(AnimationClip &clip);
        const Pose &GetPose();
        void Update();
};
```

Animation Player

- ***A simple player just needs to increment the Time and access the new pose once per frame***
- ***The first question that comes up though, is what to do when it gets to the end of the animation clip?***
  - Loop back to start
  - Hold on last frame
  - Deactivate itself… (return 0 pose?)
  - Send a message…

Animation Player

Some features we may want to add for a more versatile animation player include:

- Variable playback rate
- Play backwards (& deal with hitting the beginning)
- Pause

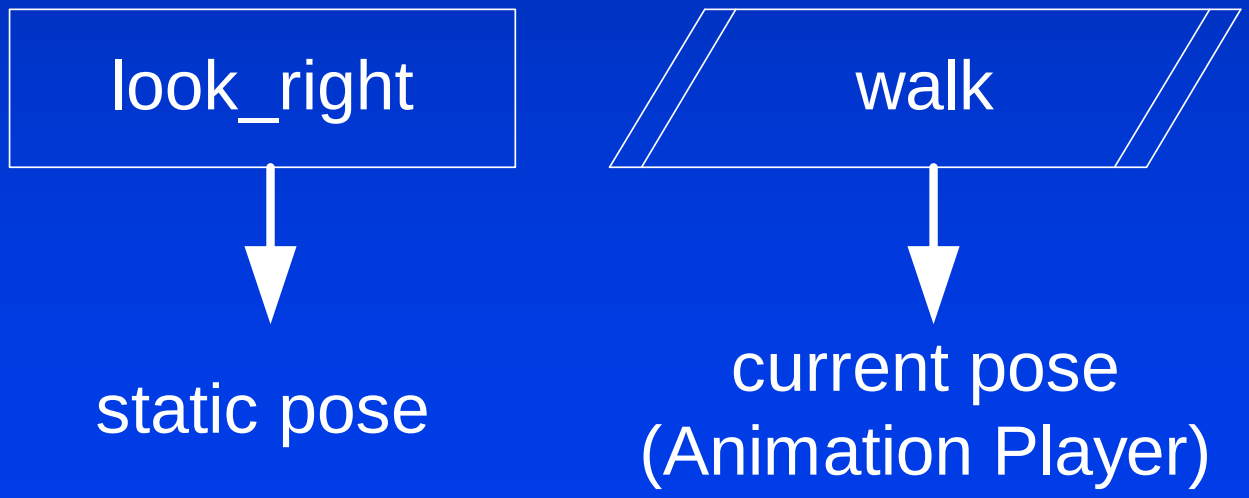
It's kinda like a DVD player…

Animation Player

- *The animation player is a basic component of an animation blending & sequencing system*
- *Many of these might ultimately be combined to get the final blended pose. This is why we only want it to output a pose*
- *By the way, remember the issue of sequential access for keyframes? The animation player should ultimately be responsible for tracking the current keyframe array (although the details could be pushed down to a specific class for dealing with that)*

Animation Player

As we will use players and static poses as basic components in our blending discussion, we will make a notation for them:

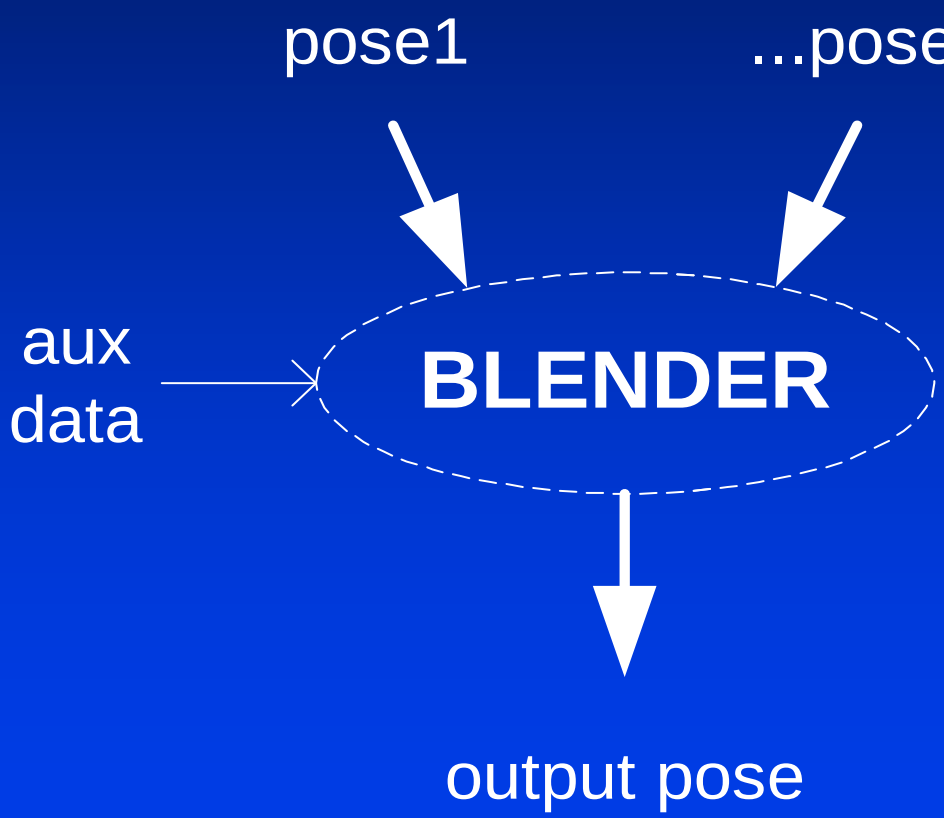


Animation Blending

Blending Overview

- *We can define blending operations that affect poses*
- *A blend operation takes one or more poses as input and generates one pose as output*
- *In addition, it may take some auxiliary data as input (control parameters, etc.)*

Generic Blend Operation



Cross Dissolve

- *Perhaps the most common and useful pose blend operation is the 'cross dissolve'*
- *Also known as: Lerp (linear interpolation), blend, dissolve…*
- *The cross dissolve blender takes two poses as input and an additional float as the blend factor (0…1)*

Cross Dissolve

- *The two poses are basically just interpolated*
- *The DOF values can use Lerp, but the quaternions should use the 'Slerp' operation (spherical linear interpolate)*

$$\phi' = Lerp(t, \phi_1, \phi_2) = (1-t)\phi_1 + t\phi_2$$

$$\mathbf{q}' = Slerp(t, \mathbf{q}_1, \mathbf{q}_2) = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{q}_1 + \frac{\sin(t\theta)}{\sin\theta}\mathbf{q}_2$$

Cross Dissolve: Handling Angles

- *If a DOF represents an angle, we may want to have the interpolation check for crossing the +180 / -180 boundary*

$$if\left(\phi_1-\phi_2>180°\right)\ \phi'=Lerp\left(t,\phi_1-360°,\phi_2\right)$$
$$else\ \ if\left(\phi_2-\phi_1>180°\right)\ \phi'=Lerp\left(t,\phi_1,\phi_2-360°\right)$$
$$else\ \ \phi'=Lerp\left(t,\phi_1,\phi_2\right)$$

- *Unfortunately, this complicates the concept of a DOF (and a pose) a bit more. Now we must also consider that some DOFs behave in different ways than others*

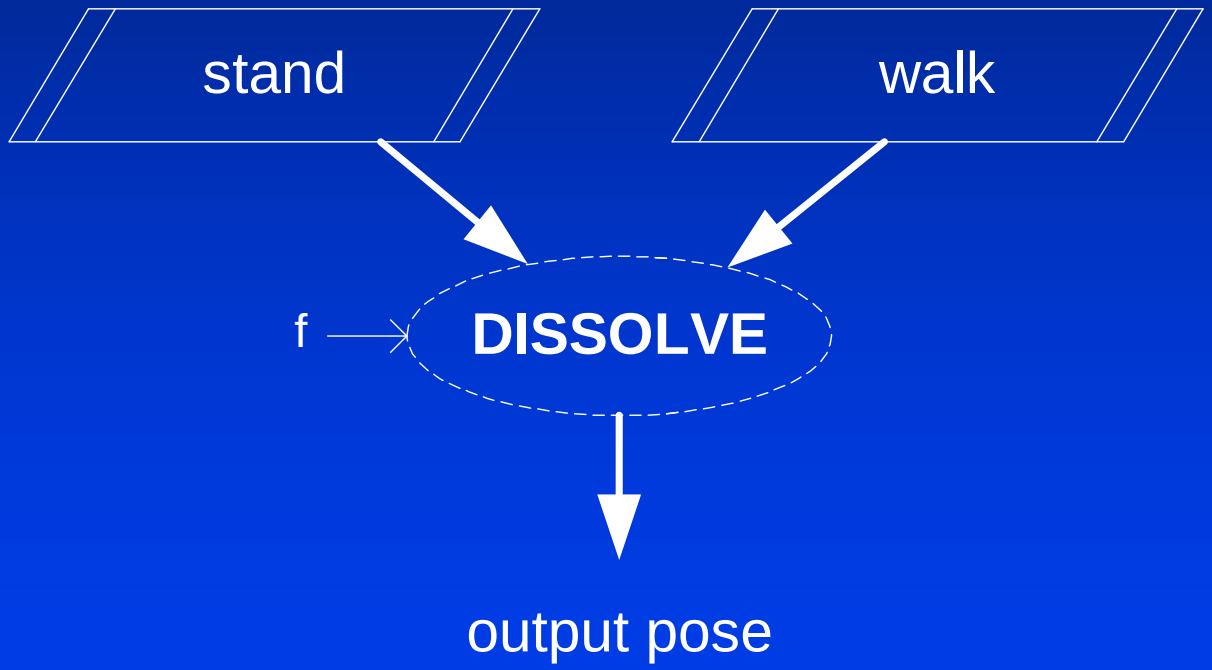
Cross Dissolve: Quaternions

Also, for quaternions, we may wish to force the interpolation to go the 'short way':

$$if\ (\mathbf{q}_1 \cdot \mathbf{q}_2 < 0)\ \mathbf{q}' = Slerp(t, -\mathbf{q}_1, \mathbf{q}_2)$$
$$else\ \ \mathbf{q}' = Slerp(t, \mathbf{q}_1, \mathbf{q}_2)$$

Cross Dissolve: Stand to Walk

Consider a situation where we want a character to blend from a stand animation to a walk animation

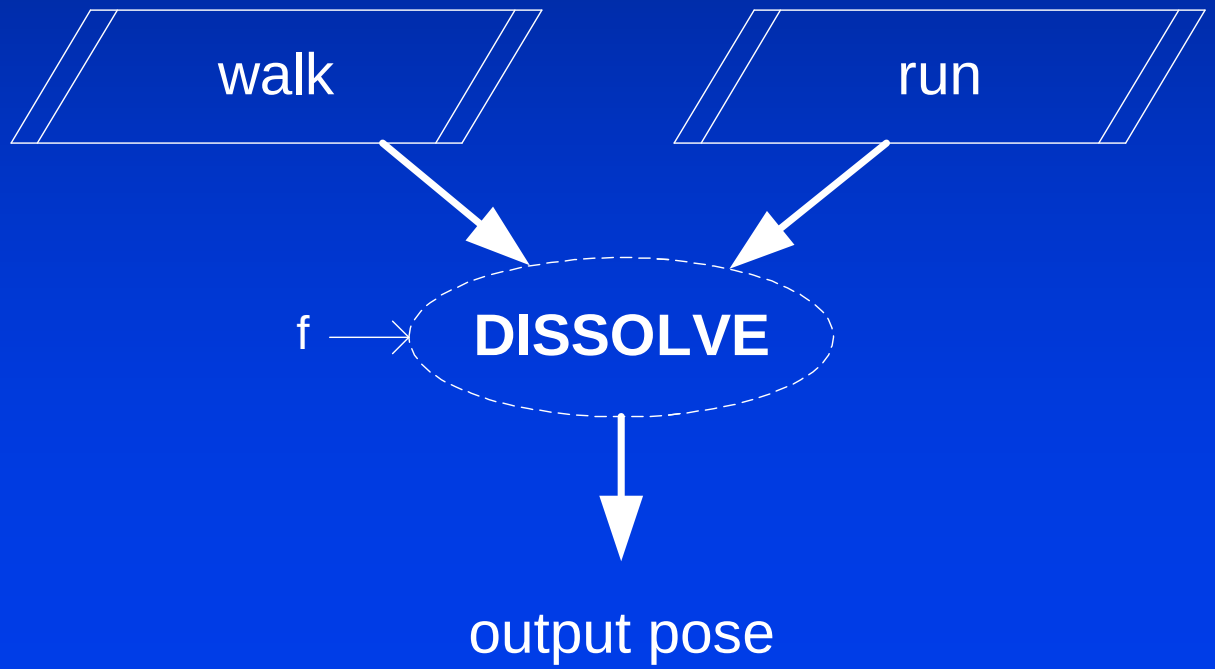


Cross Dissolve: Stand to Walk

- *We could have two independent animations playing (stand & walk) and then gradually ramp the ‘t’ value from 0 to 1*
- *If the transition is reasonably quick (say <0.5 second), it might look OK*
- *Note: this is just a simple example of a dissolve and not necessarily the best way to make a character start walking…*

Cross Dissolve: Walk to Run

Blending from a walk to a run requires some additional consideration…



Cross Dissolve: Walk to Run

- *Lets say that we have a walk and a run animation*
- *Each animation is meant to play as a loop and contains one full gait cycle*
- *They are set up so the character is essentially moving in place, as on a treadmill*
- *Let's assume that the duration of the walk animation is d_{walk} seconds and the run is d_{run} seconds*
- *Let's also assume that the velocity of the walk is v_{walk} and run is v_{run} (these represent the speed that the character is supposed to be moving forward, but keep in mind, the animation itself is in place)*

Cross Dissolve: Walk to Run

- *We want to make sure that the walk and run are in phase when we blend between them*
- *One could animate them in a consistent way so that the two clips both start at the same phase*
- *But, let's assume they aren't in sync…*
- *Instead, we'll just store an offset for each clip that marks some synchronization point (say at the time when the left foot hits the ground)*
- *We'll call these offsets o_{walk} and o_{run}*

Cross Dissolve: Walk to Run

- *Let's assume that f is our dissolve factor (0…1) where f=0 implies walking and f=1 implies running*
- *The resulting velocity that the character should move is simply:*
  - $v' = Lerp(f, v_{walk}, v_{run})$
- *To get the animations to stay in phase, however, we need to adjust the speeds that they are playing back*
- *This means that when we're halfway between walk and run, the walk will need to be sped up and the run will need to be slowed down*

Cross Dissolve: Walk to Run

- *As we are sure that we want the two to stay in phase, we can just lock them together*
- *For example, we will just say that if t_{walk} is the current time of the walk animation, then t_{run} should be:*

$$t_{run} = \mathrm{mod}\left(\left(t_{walk} - o_{walk} + d_{walk}\right)\frac{d_{run}}{d_{walk}} + o_{run}, d_{run} \right)$$

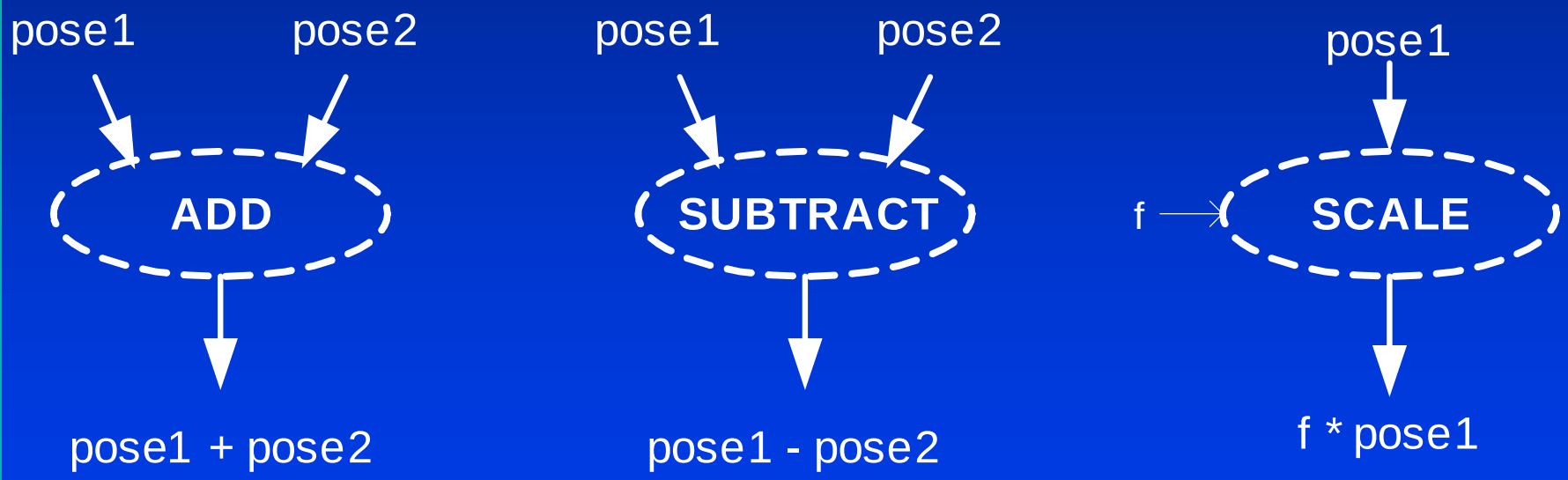
Cross Dissolve: Walk to Run

To speed up the walk animation appropriately, we will define a rate r_{walk} that the walk animation plays at (default would be 1.0)

$$r_{walk} = Lerp\left(f, 1.0, \frac{d_{walk}}{d_{run}} \right)$$

Basic Math Blend Operations

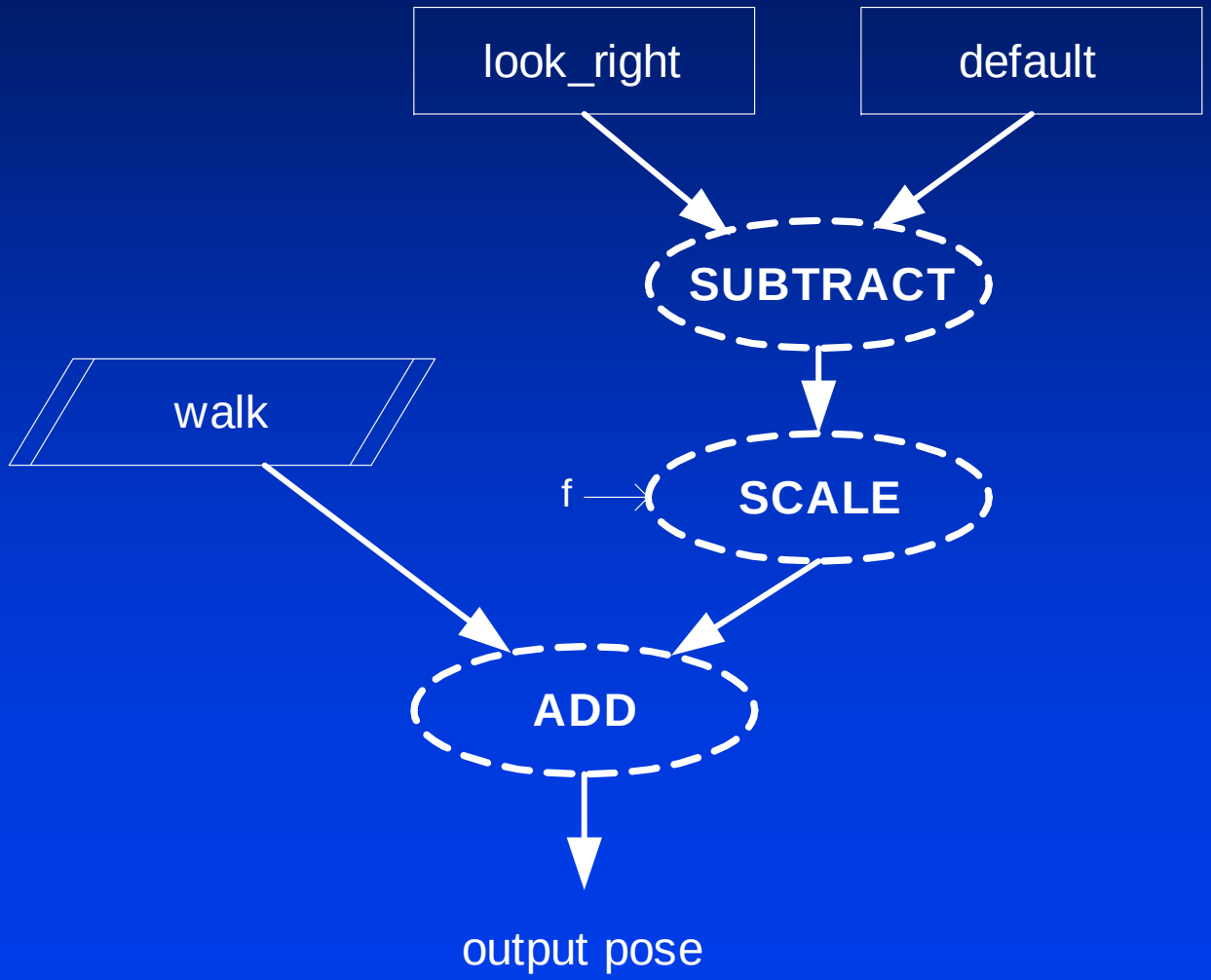
We can also define some blenders for basic math operations:



Basic Math Blend Operations

- *Its not always obvious how to define consistent behaviors between independent DOFs*
- *Quaternion addition and subtraction don't really give an expected result*
- *Addition of orientations implies that you start with the first orientation and then you do a rotation from there that corresponds to how the second orientation is rotated from neutral*

Math Operations: Body Turn

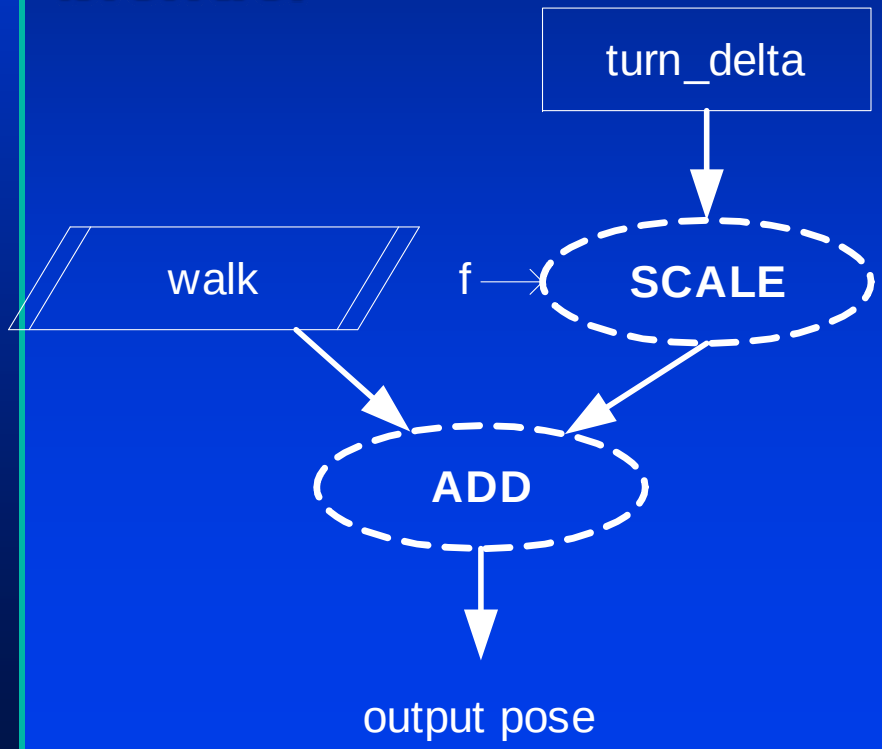


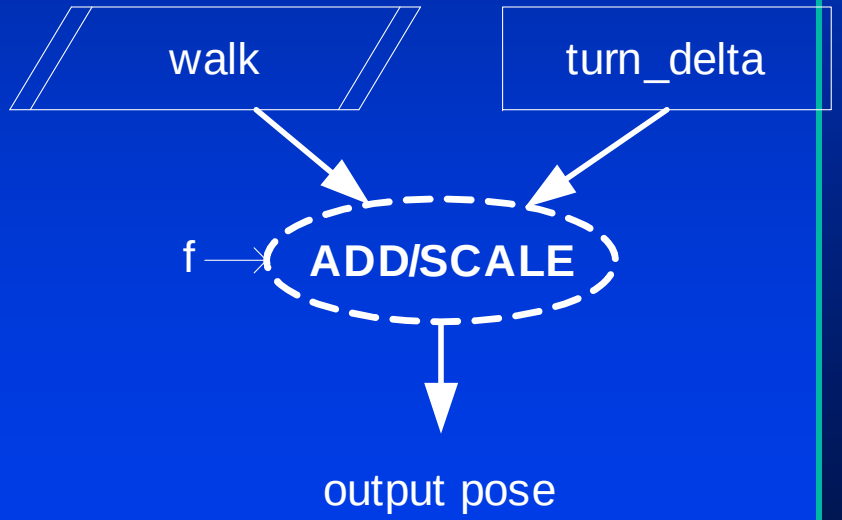
Body Turn

- *As an example of math blending operations, consider a character that walks and turns*
- *One approach to achieving this is to have an underlying walk animation and 'layer' (add) some body turn on top of it*
- *We make a static 'look_right' pose and a static 'default' pose*
- *The subtraction gives us the difference between look_right and default*
- *If we scale this and then add it on top of the underlying walk animation. The scale we use can be based on how hard the character is turning (-1…1)*

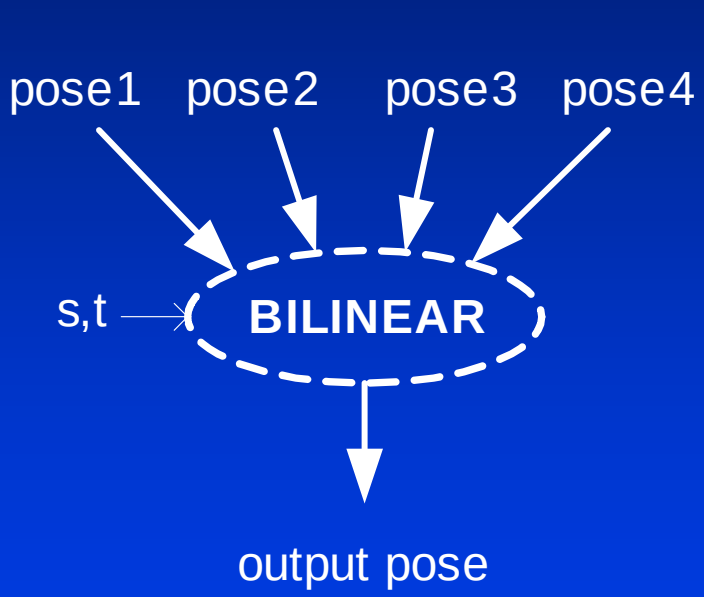
Body Turn

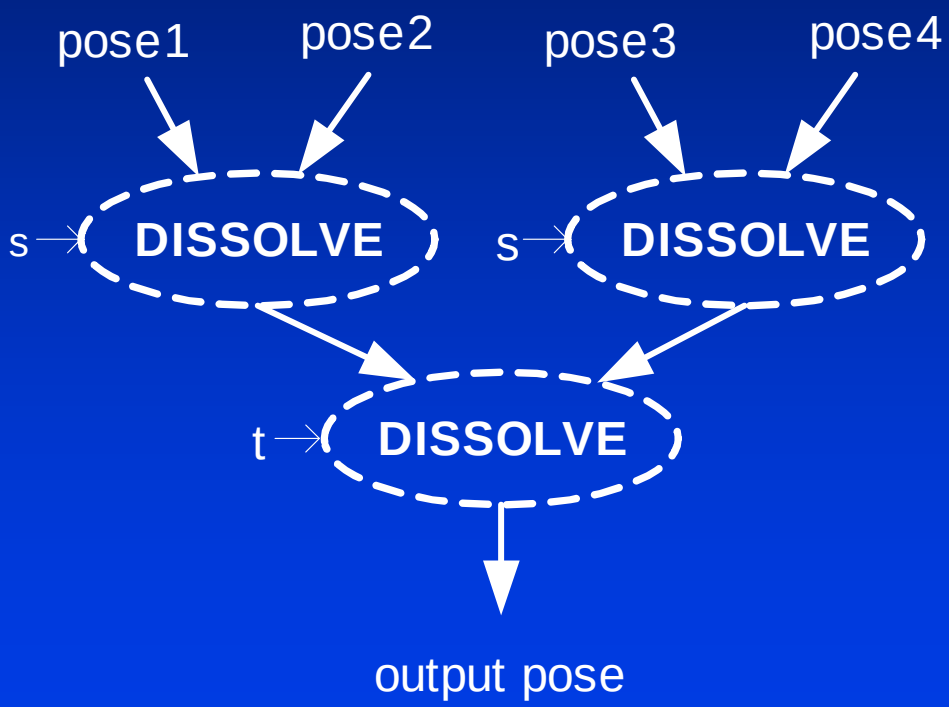
We can also speed this up by precomputing the subtraction and making a combined add/scale blender





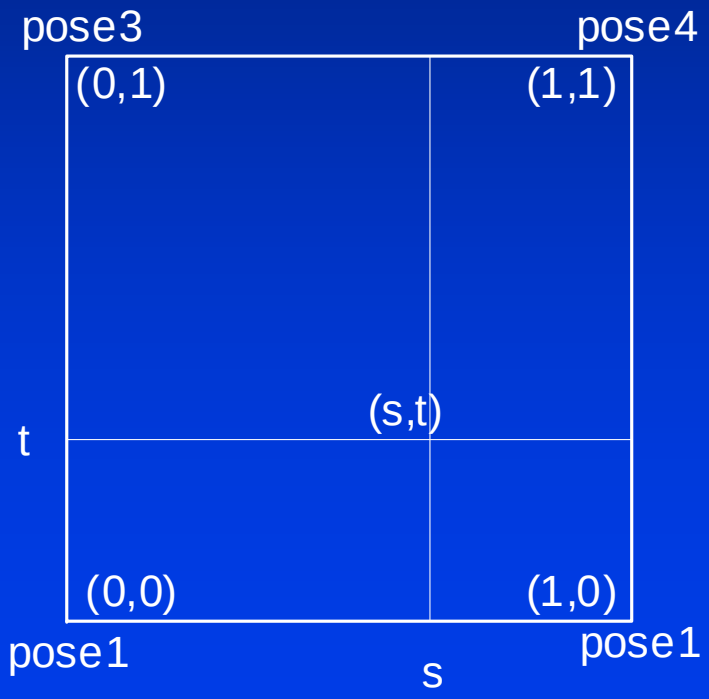
Bilinear Blend





Bilinear Blend

Bilinear blend is an extension to the cross dissolve that takes four input poses and two interpolation parameters s & t



Bilinear Blend

- *Bilinear (and trilinear…) blends can be useful for a wide range of applications*
- *As one example, consider a video game character who has to aim a weapon*
- *The character must be able to stand still and aim at any object within +/- 135 degrees to the side to side and +/- 45 degrees up and down*
- *An animator can supply key poses at 45 degree increments in both directions*
- *Then, for any desired angle, we can find the right four targets and do a bilinear blend*

Combine Blender

- *We can also have a blender that combines poses in different ways*
- *For example, we might want to treat the upper body separately from the lower body, or treat each limb separately, etc.*
- *We can use different blenders for each body section and then combine them into a final pose*
- *This also implies that we can use smaller pose vectors in each body section to save computations and memory*
- *The actual combine operation could just have lookup tables that map index values of the incoming poses to index values of the final pose*

Mirror Blender

- *Mirroring animations across the x=0 plane can be an effective way to save memory and complexity*
- *It requires that a character is symmetrical (or close enough…)*
- *Like the combine blender, mirroring requires some sort of table as input that describes how to mirror each DOF*
- *Different DOFs will need different treatment*

$$Translation: \; x' = -x, \; y' = y, \; z' = z$$

$$Rotation: \quad x' = x, \quad y' = -y, \; z' = -z$$

$$Quaternion: \mathbf{q}' = \begin{bmatrix} q_0 & q_1 & -q_2 & -q_3 \end{bmatrix}$$

- *Also, DOFs on the right need to be swapped with DOFs on the left*

Clamp Blender

- *DOF limits can be implemented as a blend operation*
- *This can be for performance, as it allows precise control over when (and if) DOF limits are used*
- *For example, consider that DOF limits should not be necessary when cross dissolving between two animations (assuming the animations are already within the legal limits)*
- *Use of add, subtract, and scale operations may require clamping for safety*

Multi-Track Blending

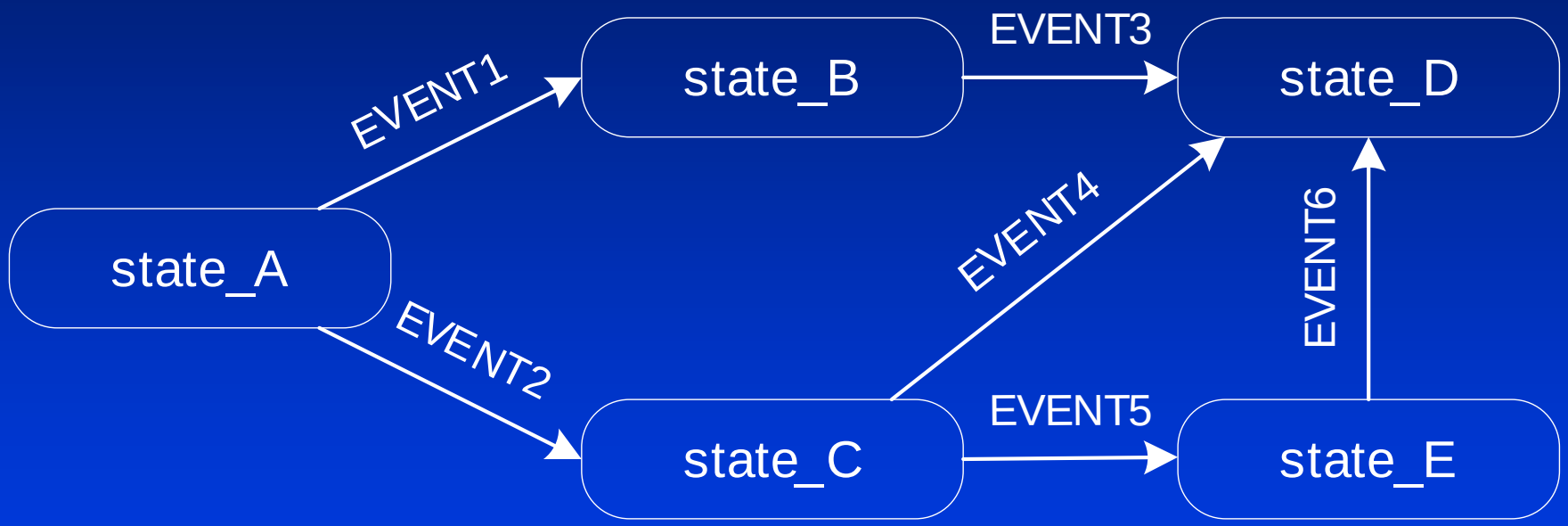
- *One can also think of an animation blending system as being similar to a multi-track audio (or video) editing system*
- *Different animations (or poses) can be placed in different ‘tracks’ and each track could have some additional controls and custom behaviors*

Animation State Machines

State Machines

- *Blending is great for combining a few motions, but it does not address the issue of sequencing different animations over time*
- *For this, we will use a state machine*
- *We will define the state machine as a connected graph of states and transitions*
- *At any time, exactly one of the states is the current state*
- *Transitions are assumed to happen instantaneously*

State Machines

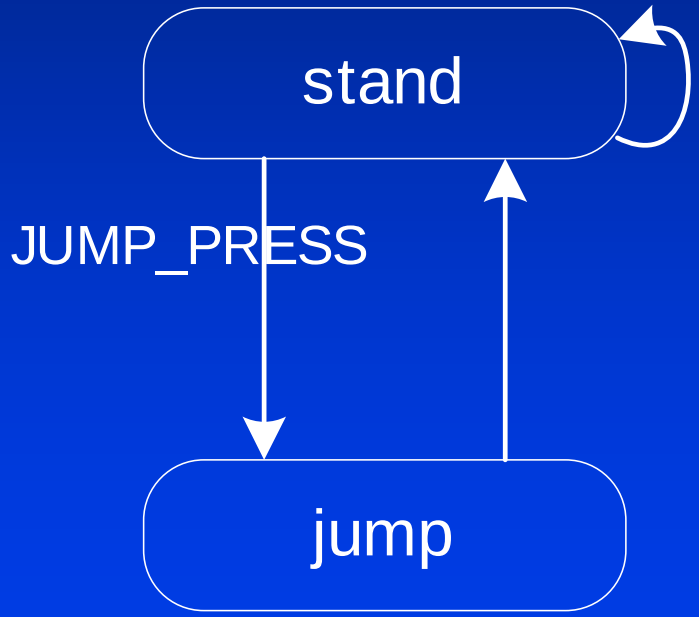


State Machines

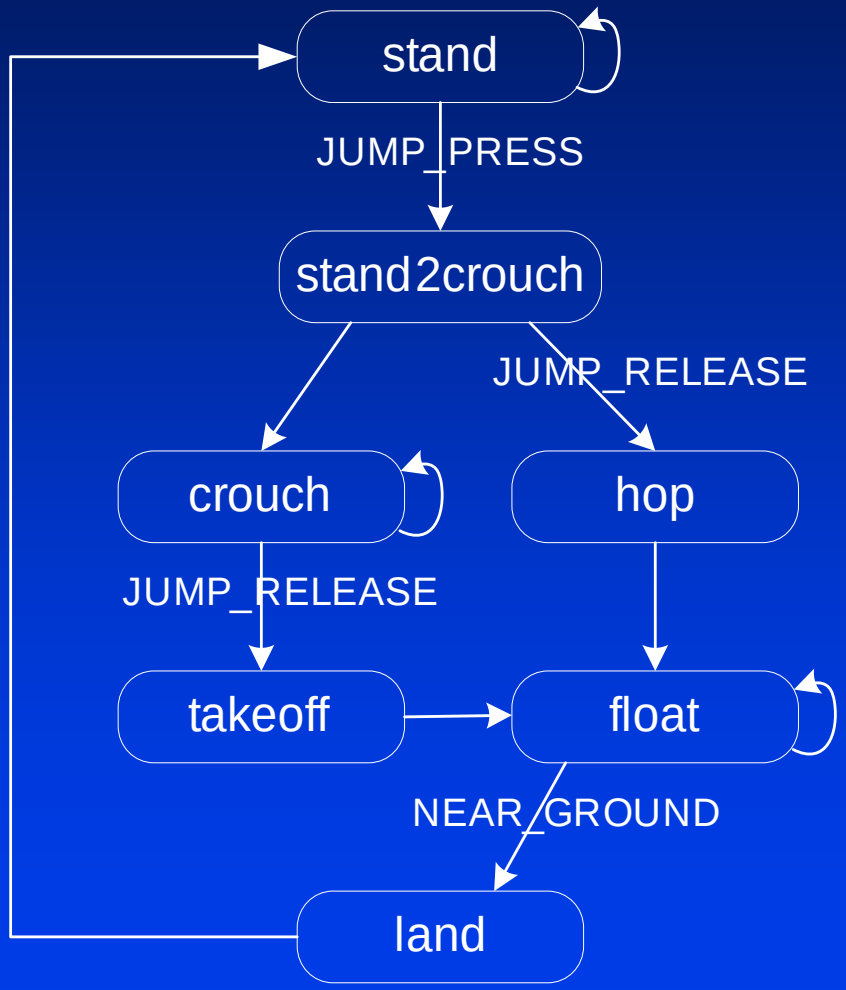
- *In the context of animation sequencing, we think of states as representing individual animation clips and transitions being triggered by some sort of event*
- *An event might come from some internal logic or some external input (button press…)*

Simple Jump State Machine

Consider a simple state machine where a character jumps upon receiving a JUMP_PRESS message



More Complex Jump



State Machine (Text Version)

```
stand {JUMP_PRESS          stand2crouch }
stand2crouch {
                  JUMP_RELEASE      hop
                  END               crouch }
crouch            {JUMP_RELEASE     takeoff }
takeoff           {END              float }
hop    {END                         float }
float  {NEAR_GROUND                 land }
land   {END                         stand }
```

State Machine Extensions

- *Global entry transitions*
- *Event masking*
- *Fancy states*
- *Modifiers*
- *Logic in states*
- *Combining blenders & state machines*
- *State machines within state machines*

Creating State Machines

- *Typing in text*
- *Graphical state machine editor*
- *Automated state machine generation (motion graphing)*

Character Mover

Character Mover

- *When we want an interactive character to move around through a complex environment, we need something to be responsible for the overall placement of the character*
- *We call this the character* mover
- *We can think of the mover as a matrix that positions the character's root*

Character Mover

- *Usually, we think of the mover matrix as being on the ground right below the character's center*
- *The mover sits perfectly still when the character isn't moving and generally moves at a smooth constant rate as the character walks*
- *The character's root translation would be animated relative to the mover*

Character Mover: Walking

- *Consider a walk animation where we have the character is moving at a rate of v meters/second*
- *The actual animation is animated as if on a treadmill (but the root may still have some translation (bobbing up/down back/forth, left/right))*
- *If the mover is moving at v meters/second though, the animation will look correct*

Character Mover

- *The mover might be coded up to do some simple accelerations, decelerations, turning, and collision detection with the ground and walls*
- *Depending on the speed that the mover is moving, we might select blend to an appropriate gait animation*

Character Mover

- *Sometimes, we want the character to do more complex moves, such as a dive roll to the right*
- *In this situation, we might want to explicitly animate what the mover should do*
- *This data can be written out with the animation and stored as additional channel data (3 translations, 3 rotations)*
- *These extra channels can be blended like any other channel, and then finally added to the mover when we pose the skeleton*