12 Separation of Concerns Architecture for AI and Animation

Bobby Anguelov

12.1 Introduction

There are two requirements for creating believable characters in today's games: the first is that characters need to make the correct decisions (artificial intelligence [AI]), and the second is that they need to look good when acting on those decisions (animation). With the heavy visual focus in today's games, it is fair to say that an AI system will live or die based on the quality of its animation system. Smart decisions won't matter much if the animation system can't execute them in a visually pleasing manner.

As we've improved the animation fidelity in our games, we've encountered a huge jump in the amount of content needed to achieve the required level of fidelity. This content refers to both the animation data, the data structures that reference the animation data, and the code required to control and drive those data structures. The biggest challenge facing us today is simply one of complexity, that is, how do we manage, leverage, and maintain of this new content in an efficient manner?

We feel that traditional techniques for managing this content have already reached their limits with the content volumes present in the last generation of games. Given the order of magnitude jump in memory between the last generation and the current-gen consoles, as well as the expectations of the audience, it is not unreasonable to expect a similar jump in the content volumes. As such, we need to take the time to evaluate and adjust our workflows and architecture to better deal with this increase in content and complexity.

In this chapter, we propose an architecture for managing the complexity of a modern animation system based on our experience developing for both last- and current-gen titles [Vehkala 13, Anguelov 13].

12.2 Animation Graphs

Before we discuss the higher-level architecture, it is worth giving a quick overview of modern-day animation systems. Animation graphs (animgraphs) are ubiquitous within the industry when it comes to describing the set of animations as well as the necessary chaining of these animations in performing in-game actions.

An animgraph is, in its simplest form, a directed acyclic graph wherein the leaf nodes are the animation sources (i.e., resulting in an animation pose) and the branch nodes are animation operations (i.e., pose modification such as blending). These sorts of animgraphs are commonly referred to as blend trees since they primarily describe the blends performed on a set of animations. Animation operations contained within a blend tree are usually driven through control parameters. For example, a simple blend between two animations will require a "blend weight" control parameter to specify the contribution of each animation to the final blended result. These control parameters are our primary means of controlling (or driving) our blend trees, the second mechanism being animation events.

Animation events are additional temporal hints that are annotated onto the animation sources when authored. They provide contextual information both about the animation itself and information needed by other systems. For example, in a walking animation, we might want to mark the periods in which the left or right foot is on the ground as well as when the game should trigger footstep sounds (i.e., contact period for each foot). Animation events are sampled from each animation source and then are bubbled up through the graph to the root. As these events bubble up through the graph, they can be also used by the branch nodes in their decision making, especially within state machine transitions. A simple blend tree describing forward locomotion for a character is shown in Figure 12.1, wherein we can control the direction and the speed of a character with the control parameters: "direction" and "speed."

In addition to blending, we also have the ability to select between two animations at branch nodes. For example, in Figure 12.1, we could replace the speed blend with a "select" node that will choose either the walk blend or the run blend based on the control parameter.

While a blend tree can perform all the necessary operations needed for a single action, it is extremely difficult to build a single blend tree to handle all the actions available to our characters. As such, we often wish to separate each action into its own blend tree and have some mechanism to switch between the actions. Often, these actions would have a predefined sequence as well as restrictions on which actions could

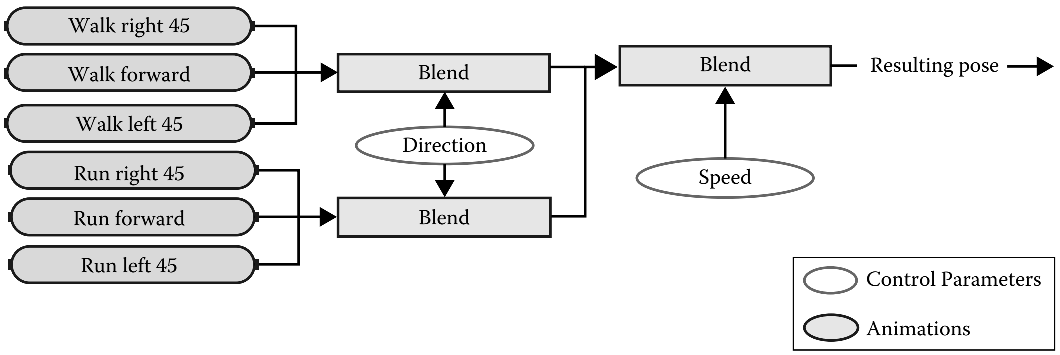

Figure 12.1

A simple blend tree describing forward locomotion.

be chained together, and so traditionally this switching mechanism between actions took the form of a state machine.

Within these state machines, the states would contain the blend trees, and the state transitions would result in blends from one blend tree to another. Each state transition would be based on a set of conditions which, once met, would allow the transition to occur. These conditions would also need to check animation-specific criteria like control parameter values, animation events, and time-based criteria like whether we reached the end of an animation. In addition, the states would also need to contain some logic for controlling and driving the blend trees (i.e., setting the control parameters values appropriately).

State machines were the final tool needed to allow us to combine all our individual actions in one system and so allow our characters to perform complex behaviors by chaining these actions together. Let's consider the simple example presented in Figure 12.1; since it only covered the forward arc of motion, we extend the direction blend to cover all directions, but we don't have any animation for when the character is idle and not moving. So we add this animation in another blend tree, which results in us needing a state machine to switch between the two blend trees. Now we realized the transitions between moving and idle don't look great, so we want to add some nice transition animations, which means that we need two more blend trees. Now we then realize that when stopping, it matters which foot of the character is planted, so we need two states to cover that and transitions that check the animation events in the walk animation. In the end, we end up with the state machine setup shown in Figure 12.2.

We can already see that for even the most basic setup, there is already a large degree of complexity present. Consider the fact that each blend tree might have its own set of control parameters that the code needs to be aware of and control, as well as all of the transition logic and state setup that needs to be created and driven. Now factor in that modern-day characters have dozens of available actions, each of which may require numerous blend trees as well as state logic for each and we now have a recipe for a complexity explosion, one which, unfortunately, has already occurred, and we now find ourselves trying to move forward through the fallout.

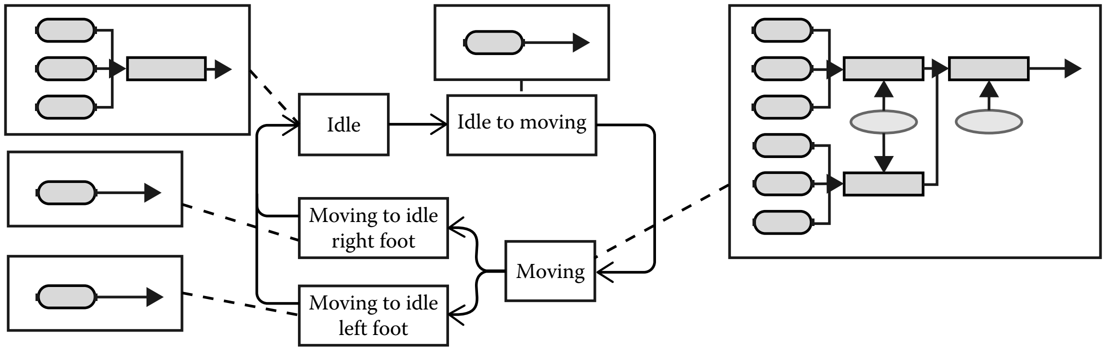

Figure 12.2

A simple animation state machine describing basic locomotion.

12.3 Complexity Explosion and the Problem of Scalability

To discuss the problem of scalability, we need to focus on the state machines described previously. Traditionally, a lot of developers would reuse the same state machine for driving both the animation and the gameplay state changes. This applies both to AI, where the state machine might take the form of a behavior tree or some other decision-making construct, as well as the player's state machine setup. We used the term state machine here, but this could be any sort of AI state change mechanic (behavior trees, planners, etc.). For simplicity's sake, from now onwards, we will use the term gameplay state machine to refer to any sort of high-level AI or player decision-making systems.

Reusing the high-level gameplay state machines for animation purposes is problematic for a variety of reasons, but the main issue is one of code/data dependency. As the blend graphs exist outside of the code base, they can be considered data and so are loaded at runtime as resources. With these blend graph resources, it is the responsibility of the game code to drive them by setting the necessary control parameters required by the graph. As such, the code needs to have explicit knowledge of these parameters and what they are used for. It is important to note that control parameters usually represent the animation-specific values (i.e., normalized blend values [0–1]), and so desired inputs need to be converted by the gameplay code into values that the animation system understands (e.g., the direction value in Figure 12.1 needs to be converted from degrees into a 0–1 blend value). In giving our code this explicit knowledge of the control parameter conversions, we've created a code/data dependency from our gameplay code to the blend tree resources meaning that whenever we change the blend trees, we need to change the code as well. The code/data dependency is pretty much unavoidable, but there is a lot we can do to push it as far away from gameplay code as possible, thereby reducing the risks resulting from it as well as allowing fast iterations.

The second biggest problem with reusing gameplay state machines is the asynchronous lifetimes of states, in that there isn't a one-to-one mapping between the gameplay states and the animation states. For example, consider a simple locomotion state from the gameplay's standpoint: a single state is usually enough to represent that a character is in motion, but on the animation side, we require a collection of states and transitions to actually achieve that motion. This usually means that we end up having animation only state machines embedded within the gameplay state machines, and over the course of a development cycle, the line between the two state machines becomes blurred. In fact, this is the main concern with the code/data dependency, since if we are required to make significant modifications to the blend trees, then the code needs to be adjusted as well and, unfortunately, this could end up affecting or even breaking the current gameplay since the two systems are so intertwined. Even worse, when animation and gameplay are so closely coupled, it can be tempting for a programmer to make direct use of information from the blend tree or make assumptions about the structure of the blend trees for gameplay decisions, which in pathological cases requires large portions of the gameplay code having to be rewritten when animation changes.

The example presented in Figure 12.2 is misleading: the idle state is a separate gameplay state. So if we were to create a simple gameplay state machine for a character with some additional abilities like jumping and climbing ladders, we might end up with a state machine setup similar to that in Figure 12.3.

There is already a significant degree of complexity in Figure 12.3, but even so, it doesn't show the full picture, as now the transitions between the gameplay states also need to contain and drive animation transitions. For example, when we transition between the idle and jump states, we somehow need to let the jump state know which animation state we are arriving from, so that we can choose the appropriate target animation state. Changing or adding animation transitions means we now need to modify the gameplay transitions in addition to all of the actual gameplay code and animation logic. As our system grows, maintenance and debugging starts to become a nightmare. The costs and risks associated with adding new states can be so high that it becomes nearly impossible to justify such change late in the development cycle. The best way to move forward and avoid this situation is to work toward loosening the couplings between the systems, and this is where a separation of concerns (SoC) architecture comes in.

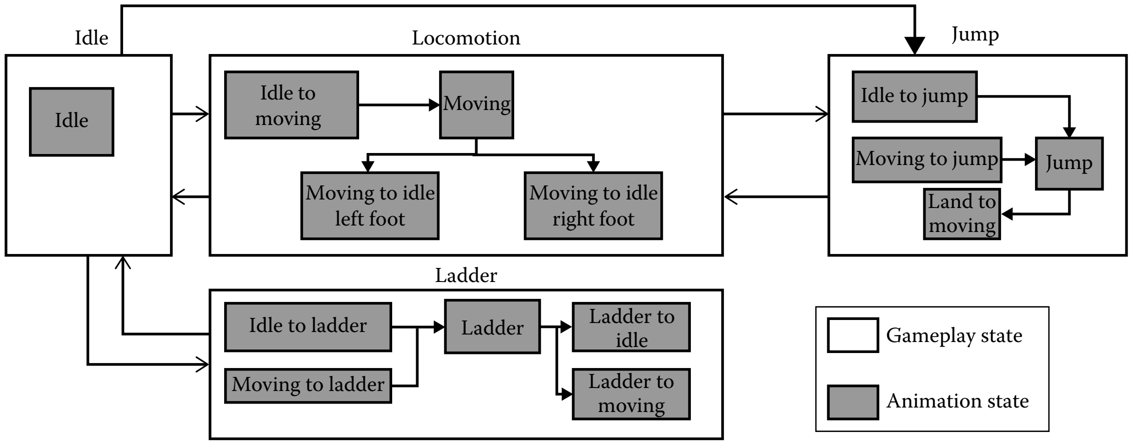

Figure 12.3

A combined gameplay/animation state machine for a simple character.

12.4 SoC

SoC is a principle that states that various systems that interact with one another should each have a singular purpose and thereby not have overlapping responsibilities [Greer 08]. It is already clear that this isn't the case for the state machine presented in Figure 12.3, as we have two distinct state machines and responsibilities intertwined together. So as a first step, we want to separate the animation state logic from the gameplay state logic. This is relatively easy, and, for many developers, this is already the case, in that their animation system supports animation state machines (e.g., *Morpheme*, *Mecanim*, *EmotionFX*). Unfortunately, the concept of animation state machines is not as common as one would imagine, and it was only in version 4 of both the Unity and Unreal engines that animation state machines were introduced. An animation state machine is simply a state machine at the animation system level, allowing us to define and transition between animation states as described in the previous sections. From this point on, we will use the term "animgraph" to refer to the combination of blend trees and state machines in a single graph.

Animation state machines can also be hierarchical in that the blend trees contained within a state can also contain additional state machines as leaf nodes. This allows for easy layering of animation results on top of one another but is not a feature that is available in all animation systems. For example, Natural Motion's *Morpheme* middleware is entirely built around the concept, while Unity's *Mecanim* only supports a single state machine at the root of an animgraph, but allows for the layering of multiple graphs on top of one another.

If we extract all the animation state machine logic from Figure 12.3, we end up with the state machine setup shown in Figure 12.4. As you can see, the animation state machine once separated out is still relatively complex but this complexity can be further simplified by making use of hierarchical state machines, with container states for each action (i.e., "jump" or "ladder"). The gameplay state machine is now free to only worry about gameplay transitions without having to deal with the animation transition, and it is now also possible, to some degree, to work on either system independently.

There is still a catch. While this initial separation goes a long way to help decouple the systems, we still have a coupling between the gameplay state machine and the animation system. We still need to have explicit knowledge of the control parameters

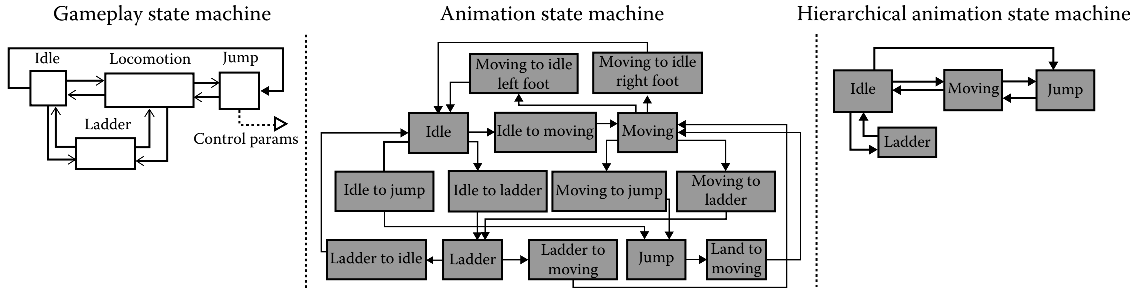

Figure 12.4

Separation of animation and gameplay state machines.

required to drive the state machine as well as explicit knowledge of the topology of the animgraph to be able to determine when states are active or when transitions have occurred/completed. This means that we still need a lot of code to drive and poll the animation system, and unfortunately this code still takes the form of some sort of state machine. It is surprising to note that only once we separated out the animation state machines from the gameplay state machines did we realize that we actually had three state machines that were intertwined: the gameplay state machine that controls character decision making, the animation state machine representing the states and possible transitions, and the animation driver state machine that acts as the interface between the gameplay and animation state machines. The fact that there was a hidden state machine goes a long way to highlight the danger posed by building monolithic systems.

Coming back to the animation driver state machine, its responsibilities are to inspect the animation state machine and provide that information back to the gameplay system. It is also responsible for converting from the desired gameplay control parameter values to values the animation system understands as well as triggering the appropriate animation state transitions when needed. In many cases, this driving code would also be responsible for any animation postprocessing required, that is, post rotation/translation of the animation displacement. As such, we still have a lot of different responsibilities within one system and we haven't really improved the maintainability or extensibility of our characters. To move forward, we need to pull all of this animation-specific code out of our gameplay systems, which will remove any remaining code/data dependencies we have between our animation data and gameplay code.

12.5 Separating Gameplay and Animation

To pull out the animation driving code, there are two things we can do, the first is relatively easy and goes a long way to simplify the gameplay code, while the second is significantly more invasive and time-consuming. So if you find yourself battling code/data dependencies toward the end of a project, the first technique might prove useful.

We mentioned that one of the key responsibilities of the animation driving code was to convert between high-level gameplay desires such as "move at 3.5 m/s while turning 53° to left" to the animation level control parameters, which might be something like "direction = 0.3 and speed = 0.24" (i.e., the blend weight values that will result in the required visual effect). To do the conversion between the gameplay values and the animation values, it is necessary for the gameplay code to have knowledge about the animations that are available, the blends that exist, what values drive what blends, etc. Basically, the driving code needs to have full knowledge of the blend tree just to convert a value from, for example, degrees to a blend weight. This means that if an animator modifies the blend tree, then the gameplay code might be invalidated and require code changes to restore the functionality. This means that any animgraph changes require both programmer and animation resources and a potentially significant delay before a build with both the code and data changes can be rolled out to the production team.

A simple way to get around this problem is to move all translation logic into the animgraph (i.e., directly feed in the higher-level gameplay values). Depending on your

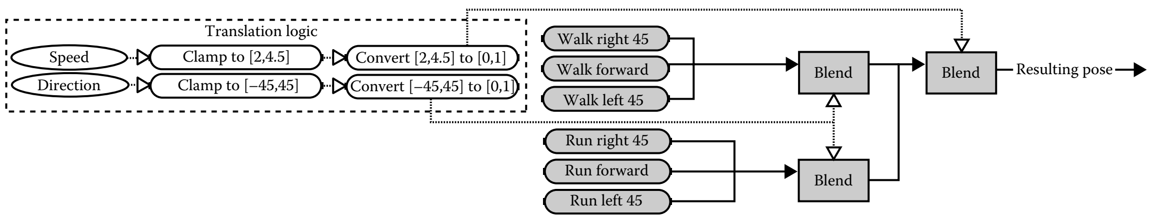

Figure 12.5

Moving control parameter translation logic to the animgraph.

animation system, this may or not be possible. For example, in Unreal 4, this is relatively trivial to do through the use of blueprints, while on Unity, there seems to be no way to perform math operations on control parameters within the graph. The benefits of moving the translation logic into the graph are twofold: first, gameplay code does not need any knowledge of the graph or the blends; all it needs to know is that it has to send a direction and speed values in a format it understands (i.e., degrees and m/s, respectively). In removing that dependency from the code and moving it into the animgraph, animators can now make drastic changes to the animgraphs without having to modify the gameplay code; in fact, they can even swap out entire graphs just as long as the inputs are the same, taking the setup shown in Figure 12.1 and moving all translation logic to the blend tree result in the setup shown in Figure 12.5.

In addition to the translation logic, we can also move a lot of other control parameter logic to the graph (e.g., dampening on input values so we get smooth blends to the new values instead of an instant reaction to a new value).

It is still important to note that gameplay code should probably be aware of the capabilities of the animation (i.e., roughly what the turning constraints are and what reasonable speeds for movement are, but it doesn't have to be explicit). In fact, it is usually gameplay that defines some of these constraints. Imagine that we have the setup in Figure 12.5 and gameplay decides we need the character to sprint; the gameplay team simply feeds in the faster velocity parameter and notifies the animation team. The animation team can now create and integrate new animations independently of the gameplay team. From a technical standpoint, moving the translation logic from the code to data removes one layer of coupling and brings us closer to the final SoC architecture we'd like.

The second thing we need to do to achieve the SoC architecture is to move all the animation driver state machine code from the gameplay state machine into a new layer that exists between the animation system and the gameplay code. In the classic AI agent architecture presented in [Russel 03], the authors separate an agent into three layers: sensory, decision making, and actuation. This is in itself an SoC design and one that we can directly apply to our situation. If we think of the gameplay state machine as the decision-making layer and the animation system as the final actuators, then we need an actuation layer to transmit the commands from the decision-making system to the actuators. This new layer is comprised of an animation controller and animation behaviors. Gameplay systems will directly interface with this new layer for any animation requests they have.

12.6 Animation Behaviors

An animation behavior is defined as a program that executes a specific set of actions are needed to realize a character action from a visual standpoint. As such, animation behaviors are purely concerned with the visual aspects of character actions, and they are not responsible for any gameplay state changes themselves. That is not to say they have no influence on gameplay, though. There is bidirectional flow of information between the gameplay systems and the animation behaviors, which will indirectly result in gameplay state changes, but these changes will not be performed by the behaviors themselves. In fact, we suggest the animation behaviors are layered below the gameplay systems (in your engine architecture), so that there is absolutely no way for the behaviors to even access the gameplay systems.

In describing animation behaviors, we feel it makes more sense to start at the animation system and slowly move back up to the gameplay system. As such, let's take a look at the example animgraph presented in Figure 12.6. We have a full-body animation state machine that contains all the full-body actions that our characters can perform. Within that state machine, we have a state called "locomotion," which, in turn, contains a state machine with the states necessary to perform locomotion. Each of these states represents additional blend trees/state machines.

Let's say that we wish to build a "move" animation behavior. For this behavior to function, it will need to have knowledge of the animgraph (especially the "locomotion" state machine), all the states contained within it, and the contents of each state. Once we've given the animation behavior all the necessary graph topological information, it will need to drive the state machine, which implies it requires knowledge about the needed control parameters and context thereof. With this knowledge, the animation behavior is ready to perform its task. This is achieved in three stages: "start," "execute," and "stop."

The "start" stage is responsible for ensuring that the animgraph is in a state in which the execute stage can proceed. For example, when starting to move from idle, we need to trigger the "idle to move" transition and wait for it to complete; only once that transition is complete and we are in the "move" state, we can move onto the "execute" stage. In the case of path following, we may also need to perform some pathfinding and path postprocessing here before the behavior can continue.

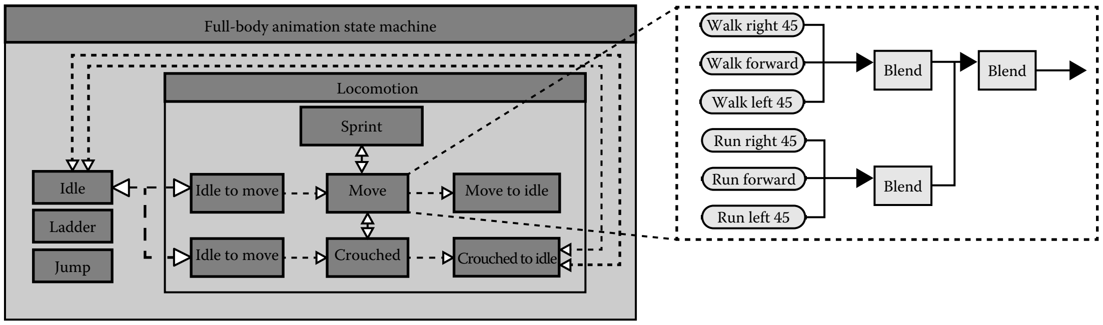

Figure 12.6

An example of a full-body animation state machine.

The "execute" stage is responsible for all the heavy lifting. It is responsible for driving the animgraph to generate the required visual result. In the context of locomotion, we would perform the path following simulation and set the direction and speed parameters needed to follow a given path. Once we detect that we have completed our task, we will transition over to the "stop" stage.

The "stop" stage is responsible for any cleanup we need to do as well as transitioning the animgraph to a neutral state from which other animation behaviors can continue. With our locomotion example, we would free the path and then trigger the "move to idle" transition in this stage and complete our behavior.

It is important to note that in the example given in Figure 12.6, the actual transition between "idle" and the "locomotion" state exists within the "full-body" state machine. This implies that both "idle" and "locomotion" need to know about the "full-body" state machine. Well, in fact, it turns out that all states in the "full-body" state machine need to know about it. This brings us to the concept of *animgraph views*. An animgraph view is an object that has knowledge of a specific portion of the graph as well as utility functions to drive that portion of the graph. From that description, animation behaviors are in fact animgraph views themselves with the exception that they have an execution flow. It is better to think of graph views as utility libraries and the animation behaviors as programs. Multiple behaviors can share and make use of a single graph view, allowing us a greater level of code reuse and helping to reduce the cost incurred when the animgraph changes. In our example, we would have a "full-body graph view" that would know about the topology of the "full-body state machine" and offer functions to help trigger the transitions between the states, for example, set full-body state (IDLE).

To execute a given task, animation behaviors require some instruction and direction. This direction comes in the form of an animation order. Animation orders are sent from the gameplay systems and contain all the necessary data to execute a given behavior. For example, if the gameplay systems want to move a character to a specific point, they would issue a "move order" with the target point, the desired movement speed, the character's end orientation, and so on. Each animation order has a type and will result in a single animation behavior (e.g., a "move order" will result in a "move behavior"). Animation orders are fire-and-forget, in that once an order is issued, the gameplay system doesn't have any control over the lifetime of the animation behavior. The only way that behaviors can be cancelled or have their orders updated is by issuing additional orders, as detailed in the next section.

In addition to the animation orders, we have the concept of *animation behavior handles* that are returned for each order issued. These handles are a mechanism through which the animation behaviors and gameplay systems can communicate with one another. Primarily, the handles are a way for the gameplay systems to check on the status of an issued animation order (i.e., has the order completed, has it failed, and, if so, why?). An animation handle contains a pointer to the animation behavior through which it can perform necessary queries on the state of the behavior. In some cases, for example, a player character, it is useful to be able to update a given program on a per frame basis (i.e., with the controller analog stick inputs that will be translated into animation control parameter settings by the behavior each frame).

We show a simple timeline representing the interaction between a gameplay system and an animation behavior for a simple "move" order in Figure 12.7. It is important to note

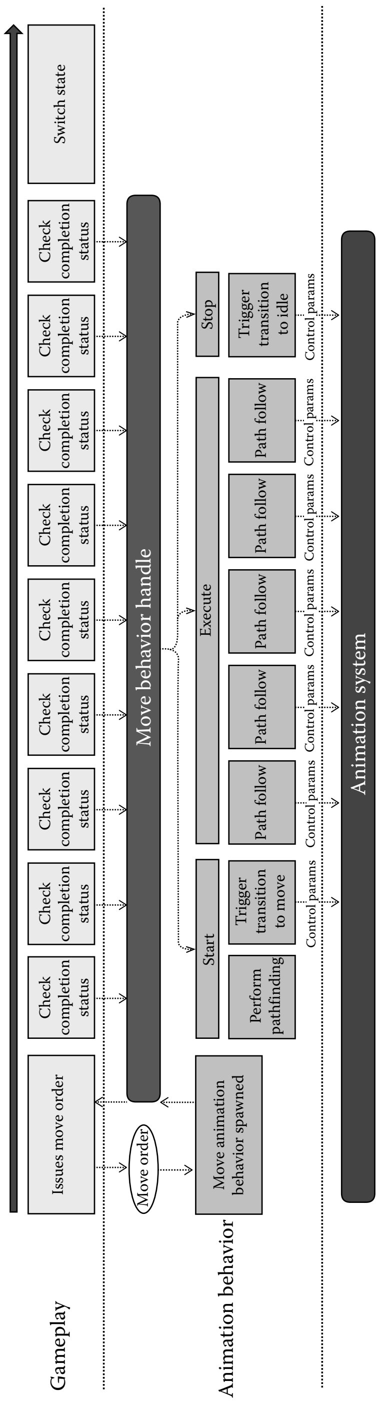

Figure 12.7
Timeline for an animation behavior.

how all communication between the animation behavior and the gameplay system occur through the animation handle.

In addition to the three update stages, animation behaviors also feature a post-animation update "postprocess" stage. This stage is mainly used to perform animation postprocessing such as trajectory warping but can also be used for post-physics/post-animation pose modification (i.e., inverse kinematics [IK]). As a note, IK and physics/animation interactions should ideally be performed as part of the animation update but not all animation systems support this.

Since we will have multiple animation behaviors covering all of a character's actions, we need some mechanism to schedule and execute them. This is where the animation controller comes in.

12.7 Animation Controller

The animation controller's main role is that of a scheduler for the animation behaviors. It is the primary interface used by the higher-level gameplay systems to issue requests to the animation system through the use of animation orders. It is responsible for creating and executing the animation behaviors. The animation controller also features animation behavior tracks (queues) used for layering of animation behaviors. For example, actions such as "look at," "reload," or "wave" can be performed on top of other full-body animations (e.g., "idle" or "walk"), and, as such, we can layer those actions on top of full-body actions at the controller level. In previous games, we found it sufficient (at least for humanoid characters) to have only two layers: one for full-body actions and one for layered actions [Anguelov 13, Vehkala 13]. We also had different scheduling rules for each track. We only allowed a single full-body behavior to be active at any given time, whereas we allowed multiple layered behaviors to be active simultaneously.

For the full-body behaviors, we had a queue with two open slots. Once a full-body animation order was issued, we would enqueue an animation behavior into the primary slot. If another full-body order was received, we would create the new animation behavior and first try to merge the two behaviors. Merging of animation behavior is simply a mechanism through which an animation order can be updated for a behavior. For example, if we issue a move order to point A, we would spawn a move animation behavior with the target A. If we then decided that point B is actually a better final position, we would issue another move order with point B as the target. This will result in another move animation behavior being spawned and merged with the original move behavior thereby updating its order; the second behavior is then discarded. Once the merge process completes, the behavior will then detect the updated order and respond accordingly. If an animation order results in a full-body animation behavior of a different type than the already queued behavior, we would have queued the new behavior to the second slot and updated it, but we would have also notified the original behavior to terminate. Terminating a behavior forces it to enter the stopping stage and complete. Once an animation behavior completes, it is dequeued and is not updated any more. This means that we can in essence cross-fade between two full-body actions allowing us to achieve greater visual fidelity during the transition.

The merging mechanism does have the requirement that all behaviors be built in a manner that supports order updating. While this manner of updating animation behaviors might seem strange at first, it has significant benefits for the gameplay code. The main one is that gameplay no longer needs to worry about stopping and waiting for animations to complete or how to transition between the different animation states, as this is all handled at the controller/behavior level. This transition micromanagement at the gameplay level is also extremely problematic when trying to delegate animation control between different systems, for example, when triggering an in-game cut scene, the cinematics system requires animation control of the character. When control is requested, the character could be in any animation state. The cinematic system needs to resolve that state in a sensible way, and this has been extremely difficult to achieve in the past without coupling unrelated systems (i.e., giving knowledge of the AI system to the cinematics system). With our approach, we can now delegate control to various systems without having to create any coupling between unrelated systems. For example, animation behaviors could be written for cinematics, and when the cinematics code takes control, it could issue those orders that would terminate existing orders and result in sensible transitions between orders. In fact, the cinematics system can even reuse the same locomotion orders that the AI is using, since they are entirely system agnostic. Imagine we needed to have an nonplayable character (NPC) climb a ladder in a cut scene. Instead of fully animating the cut scene, or trying to script the AI to climb the ladder, we could simply issue the animation order directly in the cinematics system without any knowledge of the AI or the current state of an NPC.

There is also an additional benefit of this approach on the animation side. If for whatever reason we have a barrage of orders from the gameplay systems, that is, behavior oscillation, our full-body queuing mechanism will simply overwrite/merge the queued behavior with whatever new behaviors it is ordered to perform. This greatly reduces the visual glitches that traditionally arise from these kind of gameplay bugs. On the downside, it does make those bugs harder to detect from a quality assurance (QA) perspective, as there is no visual feedback now, so we greatly recommend that you implement some sort of animation order spam detection.

When it comes to the layered behaviors, we can have any number of behaviors queued, and it is up to gameplay code to ensure that the combination makes sense. We also merge layered behaviors in the same manner as for the full-body behavior, giving us the same set of update capabilities.

There is one last thing to discuss when it comes to the scheduling of animation behaviors: behavior lifetimes. The lifetime of a behavior is not synchronous with that of the gameplay state that issued the original order. Once a behavior completes, it is dequeued, but we may need to keep the behavior since the handle may still be checked by gameplay code, which updates at a different frequency. The opposite may also be true, wherein a gameplay state issues an animation order that then completes without waiting for the order to complete. As such we decided to control animation behavior lifetime through the concept of shared ownership. A behavior is kept alive (in memory), while either a handle to it still exists or it is still on one of the animation controller update queues. This can be easily achieved through the use of an STL shared_ptr smart pointer. The final architecture across all system layers is presented in Figure 12.8. For more details on the controller/behavior architecture, readers are referred to [Anguelov 13].

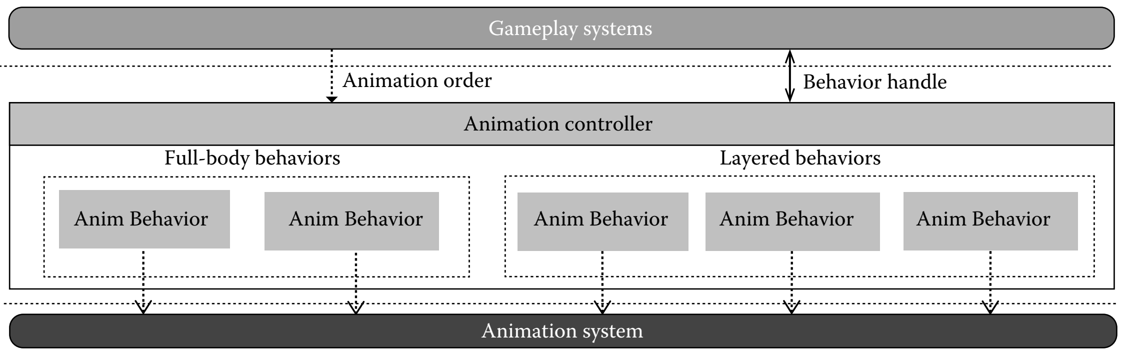

Figure 12.8

The final SoC architecture.

12.8 Benefits of an SoC Animation Architecture

Up until this point, we have discussed the SoC as a means to solving some existing problems. However, it is important to mention that there are some additional benefits in moving to an SoC architecture, which might not be immediately clear, and so we would like to highlight a few of them.

12.8.1 Functional Testing

The first clear benefit from this architecture is that systems can now be functionally tested in isolation. For example, if we wish to create some tests for the AI system, we can create a dummy animation controller that will receive orders and complete/fail them as desired without actually running any animation code. This will greatly simplify the task of debugging AI issues, as the animation code can be entirely removed from the equation. In the past with all the intertwined systems, we would never really be sure what the root cause of a bug was. This is also very true for animation testing. On a past project, we had built a stand-alone scripting system for testing the animation layer. This scripting system would issue the exact same orders as the AI, but allows us to build animation function tests in complete isolation. This was a massive win for us on previous projects when maintaining and verifying animation functionality, even across multiple refactoring phases of the gameplay code.

12.8.2 System Refactoring

Another huge benefit of this approach is that when we wish to make significant animation changes, it is much safer and easier to do so. This architecture allows us to replace character actions, one by one, without any risk of breaking the gameplay code. Furthermore, this approach allows us to perform nondestructive prototyping. When building a new version of an action, we can simply build a new behavior alongside the existing one and have a runtime switch to select between the two within the controller.

The beauty of this approach is that we can swap out behaviors at runtime without the gameplay code even being aware. If we combine this technique with the function testing mentioned earlier, it allows us to build a new version of an action and compare the two actions side by side (which alone is priceless), without having modified neither the original action nor the gameplay code. This allows us to rapidly prototype features with bare minimum functionality and expose them to the gameplay systems early on, all while building the final versions alongside them, allowing us to experiment and break actions without affecting the build.

12.8.3 Level of Detail

Being able to switch between behaviors at runtime without affecting gameplay allows us to leverage this to build a dynamic level of detail (LOD) system for the animation. In scenario's where the lifetime of characters is significant and AI updates are required even for offscreen characters (i.e., artificial life simulations), we require some mechanism to reduce or remove the cost of animation updates. If NPC locomotion was based on the animation (animation driven displacement), then this becomes relatively complex to achieve without a clear separation of the animation and gameplay systems.

With our approach, we can build several sets of cheap animation behaviors that can be dynamically swapped out at runtime based on a character's LOD level [Anguelov 13]. When we have an NPC at the highest LOD, we would want to run our default animation behaviors. As the character moves away and drops in LOD, we could exchange out some of the expensive layered behaviors with lightweight ones so as to reduce the cost. Once the character moves offscreen, then we can replace all animation behaviors with dummy behaviors that simply track the animation state needed to resume a high LOD behavior when needed.

For example, with locomotion, at the highest LOD, we would run the animation locomotion fully as well as having a layered footstep IK behavior enabled. At a medium LOD, we would replace the footstep IK behavior with a dummy behavior while keeping locomotion untouched. At the lowest LOD (offscreen), we would replace the locomotion with a simple time-based update on the given path, as well as estimating the velocity and state of the character (crouched, standing, etc.). Once this character comes back into view, we would simply swap back to the standard locomotion behavior and continue seamlessly. We suggest that you build separate behaviors for the different LODs, as this allows you to create "LOD sets" for different characters using various combinations of the behaviors. For example, you might not want to disable the footstep IK for huge characters even at medium LOD, since it may be more visible than for smaller characters.

12.9 Conclusion

In this chapter, we have presented an approach for decoupling your gameplay systems from your animation systems. We discussed the potential improvements to productivity and maintenance offered by this approach as well as provided advice on how to move toward a similar architecture.

References

[Anguelov 13] Anguelov, B. and Sunshine-Hill, B. 2013. Managing the movement: Getting your animation behaviors to behave better. *Game Developers Conference*. http://www.gdcvault.com (accessed May 11, 2014).

[Greer 08] Greer, D. 2008. The art of separation of concerns. Online article: The aspiring craftsman. http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/ (accessed May 11, 2014).

[Russel 03] Russel, S.J. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*, 2nd edn. Pearson Education, Englewood Cliffs, NJ.

[Vehkala 13] Vehkala, M. and De Pascale, M. 2013. Creating the AI for the living, breathing world of hitman: Absolution. *Game Developers Conference*. http://www.gdcvault.com (accessed May 11, 2014).