25 Animation-Driven Locomotion with Locomotion Planning

Jarosław Ciupiński

25.1 Introduction

In the race to increase immersion in video games, every aspect of a game has to be improved. Animation-driven locomotion is one way to increase the realism of character movement. This doesn't just mean having lots of animations, as playing them in a random order will look unrealistic. The solution to this problem is to plan actions so that every animation is perfectly coordinated with *future* movement. To sum it up in few words: in animation-driven locomotion, a character's movement comes directly from the animations.

However, due to the dynamic nature of games, just playing the animations is not enough. Some adjustments are required to move a character in the desired direction and to the desired spot. For that reason, the execution of a plan is important for making animation-driven locomotion work and using it to fulfill its aesthetic requirements.

This article approaches the task as follows: A high-level path is used to guide the incremental generation of an animation-driven path. An animation-driven path is comprised of *actions*, which are broken into three categories. *Transfer actions* are used to cover longer distances in roughly a straight line. *Pretransfer* and *posttransfer* actions are optionally used to move into and out of transfer actions. The entire system described here is a revised version of what shipped in the game *Bulletstorm*.

25.2 Animation and Movement Architectures

In many cases, locomotion does not exist as a separate subsystem. Responsibility for a character's movement is often divided between the AI, gameplay, physics, and animation systems. To have more control over "what is happening and why" in locomotion, it is better to move as many responsibilities related to movement (creation of the navigation path, taking care of actual movement, queuing, checking if the target is reached, etc.) together, providing a clean and easy-to-use interface. The best place to put locomotion code, if not in a separate layer between AI/gameplay/scripting and animation, is in an animation subsystem itself.

Animation-driven locomotion uses data from animation to move a character. What this means is that movement conforms to animation data and velocity data contained in the animations. Locomotion subsystems may make further changes to adjust velocities to fulfill movement requests. To make animations easier to work with, they should follow some basic rules. For example, the root bone (if root motion in your animation code translates directly to the character's velocity) should move as closely as possible to a straight line or curve.

While this is not a hard requirement and in some cases may not even be desired (e.g., for drunk characters), it will make locomotion planning more predictable (characters will be less likely to leave the planned path), and move execution will be easier.

If only simple looped animations are used, then there is no need for any planning and the locomotion can be fully reactive. This means that played animations are chosen to match movement which is already planned. The character's AI then selects its own velocity and animations to try and make everything look appropriate. Such approaches are simple to implement, but in many cases do not look natural, especially when movement is starting or stopping.

To make movement look more natural, transition animations are used. These are animations for starting, stopping, changing directions, and other nonlooped actions. Note that these animations take time and require space to be played properly. Therefore, a system that just responds to current requests will work, but only in some simple and straightforward situations. For example, such a system will have no problem with a character running forward 10 meters and starting to stop 2 meters before the destination point. But, if the path is more complicated, a character may easily miss the point where it was requested to stop. There are often several variants for stopping animations, but checking against all possibilities for every frame is too expensive. As the AI knows what path to take to get to its final destination, preplanning animations is the natural solution.

25.3 Preparation

When starting work on animation-driven locomotion, it is strongly advised that you talk to animators and AI programmers as much as possible to decide what you want to achieve. For example, animators may desire really long, nice-looking animations to cover various situations in order to achieve nice aesthetics. However, these may be problematic to handle, as long animations will result in a less responsive system and will require more physical space to perform. This may mean that the system will need to plan ahead more than just a few steps.

You should also decide the kinds of movement that you want to have. Running and walking, for example, might only be done in the forward direction, while other directions will be covered with short steps or in sequence (making turns and moving forward). Also, it is important to decide whether obstacles can be traversed before starting work on animations and the planning system. If so, consider what kind of obstacles there will be, how they should be approached by the character, how they should be stored in the navmesh, and how they are processed. Animators, AI programmers, and/or designers may want to have other features and rules present. An example might be a rule to have specific animations for shooting when taking steps in any direction, instead of allowing shooting to be overlaid onto walking or running animations.

25.4 Locomotion Planning

It is best to divide planning for locomotion into separate modules, each having a distinct and clearly defined purpose:

1. A navigation path-processor
2. A planner that creates an action-stack
3. An animation system that executes the action-stack

It is strongly advised to add an additional module that works as an interface to other game subsystems. This module would accept new requests, prepare orders for animation modules (what they should do now), and work as a central hub for exchanging data. Such a module really helps with finding problems with locomotion; it makes it easier to find invalid requests and decide whether it was the animation/locomotion subsystem problem or another module that failed.

25.4.1 Navigation Path-Processor

The handling of movement requests starts with the creation of a navigation path. It is worthwhile to store the navigation path as both polygons and points. Due to unexpected events that may take place during movement (e.g., other objects moving in the way), the navigation path might require adjustment or complete recreation of the whole path. Storing polygon data will reduce the need to access the navmesh. It also gives more freedom when adjusting point-based paths, which are the base representation of the planner (Figure 25.1).

The creation of point-based paths should start with a simple approach such as string-pulling or the funnel algorithm that will give the shortest path connecting the start and end points. Please refer to the following resources for more information on pathfinding, string-pulling, and the funnel algorithm [Demyen et al. 06, Cui et al. 11].

The first job for the path processor is to take care of points that are close to each other, as they don't provide any extra information and may just complicate the subsequent steps in locomotion planning. It is easier to assume that every segment of the path is of minimum length, chosen arbitrarily, or based on the shortest stopping animation. This means that the three subsequent segments (starting at the current location of the character) will cover, for example, at least 2 or 3 meters.

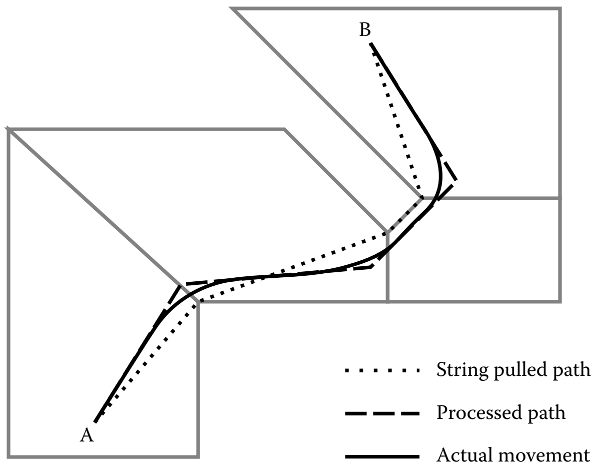

Figure 25.1

A path from point A to point B is provided to the locomotion system as polygons and is further processed. Actual movement doesn't exactly correlate to the processed path, as for locomotion planning it is more important where the animation should start and where it will end.

In some cases, when dealing with longer movement animations (that require more space), it is better to do further adjustments to the location of points. For example, after string-pulling a path, there might be two subsequent points that change the direction of movement by 90° in opposite directions and are very close to each other. If they are too close, it will be impossible to play two "sharp-turn-while-running animations" one after another and to (roughly) stay on path. If there is enough space around these points, they might be moved further away from each other to give enough space to play both animations. If it isn't possible to move the points further away, there should still be a fail-safe solution provided: stopping at the first turn point, taking a step towards the next point, and starting to run from there. In order to avoid affecting the fluidity of movement, such fail-safe solutions should be used rarely.

In some cases it might be useful to add extra points along the path that would make approaching some locations easier. For example, instead of running to an obstacle that should be jumped over, stopping next to it, and then jumping, an extra point could be used to run at the obstacle from a better angle. Due to memory limitations, there might be just two animations for "jumping from standing" and for "jumping while running straightforward" and both might look terrible when running toward an obstacle from a wide angle; therefore, approaching from a better angle is critical to making it look good.

25.4.2 Action-Stack

The *action-stack* is a list of actions required to follow the current segment of the path. It is more feasible to choose actions starting from the end (from the desired goal state). This means that the first actions to be chosen are last to be executed (Figure 25.2).

When the path has been processed, it should be in the form of points with some extra data describing how to behave at each given time (crouch, do not run, jump over). Only

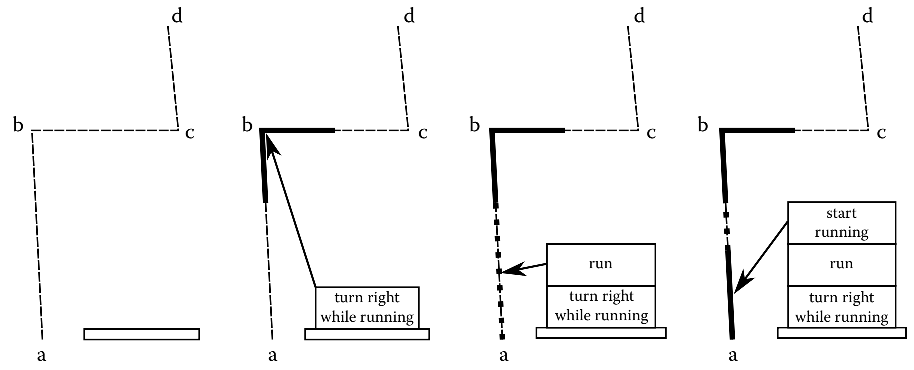

Figure 25.2

In this example we are creating an action-stack for segment "ab" of the path, starting at point "a". It is essential to know how much space animations need (for animation-based actions this is marked as a thick line) so only animations that can fit in given segment are chosen. The space required to execute an animation-based action is also used to determine when to first trigger the action (a dotted line represents a "run" action that at some point should be interrupted with the "turn right while running" action).

two or three segments of the path should be considered when creating an action-stack for the following reasons:

1. The creation of the action-stack is computationally intensive, and if done for whole path, would result in significant performance spikes.
2. We have no idea what will happen in the future, so there really is no need to plan everything ahead. A character may run into another character or may be ordered to do something else.

Ideally, just one segment of the path needs to be covered by the action-stack. When all actions from the stack are performed, the character should be at the best point to go on to the next segment, and a new action-stack can be created at that time. However, the two or three subsequent segments should still be considered when creating the first action-stack, in order to best enter subsequent segments. There is also another reason for considering subsequent segments: In some of the cases it might be possible to use one animation to cover several segments. One example might be entering cover behind a corner.

Even if planning actions are only performed for one segment, knowledge of what the character will be doing afterwards is valuable. Will the character run or slide? Does it need to stop there or should it proceed at full speed? This tells it what state (location, direction, gait, or pose) the character should be in at the end of the current segment. There is also, of course, the current state of the character. Other information might describe the current segment, such as distance, and what animations are allowed or disallowed (e.g., character can't run here). Given this data we have two options to cover the segment:

1. Do one action that will directly transition the character from current state straight to the desired state.

2. Do several actions:
   A pretransfer action that will ready the character for a transfer action (e.g., start running in specific direction).
   A transfer action that will move the character over longer distances (e.g., walk, run).
   A posttransfer action that will bring the character into the desired state (e.g., stop at a point, perform sharp-turn while running).

25.4.2.1 Direct Transition

A direct transition gives the best-looking results but requires lots of animations that cover many possibilities. In an actual game these should only be used to cover special cases. For example, walking extremely short distances can be accomplished by an animation taking a single step or two steps. For such short distances there is no need, and it might look strange to start running and then immediately stop.

25.4.2.2 Transfer Action

Using a transfer action is more feasible for longer distances. It requires animations to be divided into three basic groups (as described previously: pretransfer, transfer, and posttransfer), which can be heavily reused in different combinations resulting in smooth character animation.

Each action (pretransfer, transfer, and posttransfer) is optional. Since a character may already be in a "transfer" state, there may be no need to do anything at the end of the segment. Additionally, the transfer itself may be omitted as in some cases it might be practical to just play pretransfer and posttransfer animations back-to-back.

In games, it is not possible to have pretransfer or posttransfer animations for all possible situations. This may happen because of memory limitations, budget limitations, or development time limitations. Therefore, we might remove or not create animations for rare situations. When dealing with cases for which there is no appropriate animation, the planner will have to solve the problem in a different way.

An example of such a case is starting to run when the character is crouching. Let's assume that there is no animation for this case and follow what the planner does. The planner first tries to find one animation that will take the character from a crouched stance (current state) to running 20 meters away (desired state). There, of course, is no such animation. The planner then tries to find posttransfer, transfer, and pretransfer actions, but it can't find an action to take the character from crouching to running. Suppose there are just animations for "character starting to run from standing" in eight directions, and there are animations that bring a character from a crouch to a standing stance. The planner should then try to use the crouch to standing animation, execute one action-stack, and try to approach the problem again. After playing the animation of the character crouching to standing, it is possible to use one of eight "stand to run" animations.

In some cases it might be possible to use actions or animations that take a character from the current state to the desired state, but only with an extra action. For example, there could be an animation that is walking to the right while shooting, but in the beginning the character faces away from the target, so before the walking-shooting animation plays, the character should turn toward the enemy.

Similar extra actions may be required when a transfer action brings a character to a desired location, but it is not yet in the desired state (e.g., the character isn't turned in the right direction or is not in the desired stance).

25.4.2.3 Implementation Suggestions

For implementation, we strongly suggest dividing data into three groups:

1. **Transfer information (walk, run, walk crouched, jump over, slide under) that has:**
   A list of transfer animations.
   A list of posttransfer animations with info about the final character state (e.g., rotation, stance, how much space does it take, or any other information needed). Note that some of the posttransfer animation entries may store that this is not the final animation to reach a desired state and the planner needs to do something more.
2. **Stance information (standing, crouching, stealthy) that has**
   A list of stance idle animations.
   A list of pretransfer animations with information about the state they take a character into (transfer state, in which direction the character will move, how much space is required, etc.).
   A list of animations that change directly from one stance to another.
3. **Direct transitions:**
   A list of animations that take a character from any state to any other state, although in most of the cases it might be enough to have such a list just for stances.

Pretransfer, posttransfer, and direct transition animations may also be described in a separate place with details irrelevant to the decision taken by the planner, but having general information about the transfer which is useful during execution.

Some of this information can be collected algorithmically. For example, how does every animation connect to the transfer animation (to start a transfer animation, to match a pose, or to adjust movement in such a manner that the posttransfer animation will be triggered at the right moment).

Other information (mostly those required by the planner) may be collected automatically, although it is often useful to enter the data or at least tweak it by hand. An example is the space required for an animation. Some "starting to run" animations may need 2 meters of space. Some animations that are taking characters from one spot to another may only cover from 1 to 3 meters. While this can be computed automatically, entering data by hand gives much more cohesion and control over what each system will do in a given situation.

Note that it is important to have fail-safe solutions that would prevent the character from being stuck in one pose, even if it means turning towards the final destination and walking there step-by-step.

25.4.3 Executing Actions

During planning, actions are divided into a few different types, including pretransfer, transfer, posttransfer, whole segment animations, and extra actions. Conversely, *execution* of actions can be divided into only two groups: transfer and nontransfer actions. The following sections describe the execution of each group.

25.4.3.1 Nontransfer Actions

Nontransfer actions just play an assigned animation with small adjustments to velocity and rotation, if required.

When changing stance (standing up or crouching), it is enough to play an animation without any adjustments. For pretransfer actions, a directional adjustment is required in order to get the character moving in the correct direction when the pretransfer animation reaches its end. Most of the posttransfer actions require characters to be at a precise spot facing the right direction, so there is a requirement to adjust these as well. Adjustments to both location and direction are needed for actions that take a character from one place to another.

Parameter adjustments should be determined by the planner, meaning that the execution of all such actions does not differ.

25.4.3.2 Transfer Actions

Transfer actions are looped animations (although there can be random or in-order animations following one another) that take a character from one point in space to another. Besides matching the correct direction and ending at the required location, a character might be required to end in a specific pose in order to make a seamless blend to a posttransfer action. For example, a stopping animation that starts with a character on the right foot requires a character to be on that same foot when the stopping animation begins.

This may require altering character velocity, such as slightly speeding up or slowing down movement. This works nicely for longer distances, but adjusting a pose over short distances can result in a velocity adjustment that is too big or results in the character switching to a posttransfer animation too far away from the target location. In such cases the pose either has to be ignored, or there should be posttransfer animations that differ in the starting pose (on the left foot or right foot). Although variants can be decided during planning, they can also be picked up during execution. This may come in handy if a character had to adjust its path slightly for other reasons.

An alternative to speeding up or slowing down the character's velocity is modifying the playback rate of the animation. It is important to remember to maintain the playback rate when switching animations and to adjust the playback rate gradually. If the playback rate is not kept, animations may seem to speed up and slow down immediately, resulting in strange-looking character behavior.

Code for handling transfer actions should try to deal with any obstacles, keep characters in formation, or perform any required path adjustments. With navigation corridors and information about the end-point of transfer actions, it is possible to do some adjustments to the movement of a character and still end at the point requested by subsequent actions. For example, if a character runs and notices a new obstacle in front of it, it may alter the direction of movement early enough to avoid hitting the obstacle, without risking that the character won't be able to end at a point where the next action should start.

There are also situations in which a character will end up outside of a known navigation corridor or at some point of execution it might become obvious that it will miss the next action's starting point or won't get there at all. In such cases, either the action-stack should be rebuilt or the whole navigation path should be reconsidered.

Hitting an obstacle should be handled in a similar way. It may be impossible to avoid hitting an obstacle or it may be decided that a character may not even try to avoid running into one. The latter solution works well enough in practice. When a character hits an obstacle, it can play an evasive animation and, after it is finished, request a new navigation path (although in some cases rebuilding of action-stack may be enough).

25.4.4 Inverse Kinematic (IK) Controllers

As the system makes lots of adjustments to movement, IK controllers for feet should be used to cover corrections by removing, hiding, or at least reducing the foot-sliding effect. A simple two-bone IK solver is enough for human limbs [Juckett 08]. A proper animation will have no foot-slide, which means that when a foot is put down on the ground, it doesn't move until it is picked up.

During execution of the action-stack, the velocity of a movement animation is increased or decreased without speeding up or slowing down the animation playback rate (as the reason to speed up or slow down is to match a pose at a given point), which unfortunately results in foot sliding. For example, if the velocity is increased, the foot will slide forward; if there are additional rotations or other adjustments, the foot may also slide sideways. In these cases, an IK controller tries to keep the foot where it was originally placed.

Other IK controllers may be used to adjust the torso location in reference to the feet to help with situations in which the feet are kept behind or in front of the character. This may happen if the character stopped and antisliding controllers kept the feet in places other than originally expected.

25.5 Other Information about Locomotion Planning

The following is additional information about locomotion planning.

25.5.1 Performance

Locomotion does not require significant CPU resources during the execution stage. Everything is already planned, and it is just about keeping the character's movement faithful to the plan. In contrast, reactive locomotion requires checking and possibly updating all possible actions during every frame.

However, while the execution stage causes no problems with performance, it is important to note that locomotion planning may cause significant CPU spikes. If there is a need, most of the spikes can be neutralized by delaying any of the following processes, trying to delay actions with lowest priority first:

1. Lowest priority: handling a new navigation path
2. Middle priority: action-stack creation for standing characters
3. Highest priority: action-stack creation for moving characters

In the worst-case scenario, some characters might get stopped. Please keep in mind that when a character is stopped (playing a stopping animation), the CPU situation may get much better. In-game, the resulting behavior may look like a bug, as the character has stopped and started running again, but remember that there is always the potential to play a new animation. In particular, an animation for looking around, scratching your head, or stumbling will all offset the user perception of poor AI behavior.

25.5.2 AI Requests for Movement

An AI that relies on animation-driven locomotion should be patient; that is, it should not send too many requests in too short of a time period. If the AI changes its mind too often, a character may get stuck in repeating “start” and “stop” animations over and over. This can be partially prevented through locomotion systems that provide extensive feedback, so the AI does not have to “worry” that a character is not moving yet or is doing something else. Not every AI request can be handled immediately, as pre- and posttransfer actions are usually not interruptible and should be left until they’re finished.

The locomotion system should be careful not to treat every AI request as something completely new and unconnected, as this results in creating completely new paths. New paths often mean that the character will stop and start to move again in the same direction. In many situations, the AI just needs to change the very end of a requested path, so the currently executed action-stack (with part of the navigation path already processed) is still sufficient for local movement.

25.6 Commercial Implementation

Planning, as described in this article, is a revised version of the planning implemented for *Bulletstorm* (developed by People Can Fly, part of Epic, published by Electronic Arts in 2011). The actual implementation relied on finite-state machines—generalized versions of transfer and stance descriptions mentioned in previous sections. This means that the whole system was data driven, although some cases required a separate approach, which was handled by special code. Mantling over and sliding under objects, for example, were added late in production.

The code for this implementation was part of the animation tree (distributed over a few animation nodes) with a separate structure (called AnimationProxy) used for communication with other game systems. Source code is available for UE3 licensees.

25.7 Conclusion

Animation-driven locomotion with planning brings a believable look and feel to a game. Characters move in a more natural and fluid manner. The basic implementation of planning is quite easy and, as it is data driven, adding more animations is simple. The same code can be used for characters that have different behaviors, although the fine-tuning of the system may require experience and time.

References

[Cui et al. 11] X. Cui and H. Shi. “Direction oriented pathfinding in video games.” *International Journal of Artificial Intelligence & Applications (IJAIA)*, Vol. 2, No. 4, October 2011. Available online (http://airccse.org/journal/ijaia/papers/1011ijaia01.pdf).

[Demyen et al. 06] D. Demyen and M. Buro. “Efficient Triangulation-Based Pathfinding.” Department of Computing Science, University of Alberta Edmonton, 2006. Available online (http://www.aaai.org/Papers/AAAI/2006/AAAI06-148.pdf).

[Juckett 08] R. Juckett. “Analytic Two-Bone IK in 2D.” http://www.ryanjuckett.com/programming/animation/16-analytic-two-bone-ik-in-2d, 2008.