



## D5.5 Tools for Splicing together animation clips



sauce

<b>Grant Agreement nr</b>	780470
<b>Project acronym</b>	SAUCE
<b>Project start date (duration)</b>	January 1st 2018 (36 months)
<b>Document due:</b>	December 31st 2019
<b>Actual delivery date</b>	December 20th 2019
<b>Leader</b>	DNEG
<b>Reply to</b>	Mungo Pay - mungo@dneg.com
<b>Document status</b>	Submission Version

**Project funded by H2020 from the European Commission**

<b>Project ref. no.</b>	780470
<b>Project acronym</b>	SAUCE
<b>Project full title</b>	<b>Smart Asset re-Use in Creative Environments</b>
<b>Document name</b>	D5.5 Tools for splicing together animation clips
<b>Security (distribution level)</b>	Public
<b>Contractual date of delivery</b>	31st December 2019
<b>Actual date of delivery</b>	December 20th 2019
<b>Deliverable name</b>	Tools for splicing together animation clips
<b>Type</b>	Demonstration
<b>Status &amp; version</b>	Submission Version
<b>Number of pages</b>	26
<b>WP / Task responsible</b>	DNEG
<b>Other contributors</b>	-
<b>Author(s)</b>	Mungo Pay - DNEG
<b>EC Project Officer</b>	Ms. Adelina Cornelia DINU, Adelina-Cornelia.DINU@ec.europa.eu
<b>Abstract</b>	When creating crowd shots for VFX shots in films, the reusability of animations becomes a consideration when bidding on shows as the cost of capturing bespoke animation data can be significant. To ameliorate this cost, we present a suite of tools that allow crowd artists to create new animation from existing clips in an automated manner by utilising a motion graph implementation, augmented with a constraint based trajectory editing toolkit.
<b>Keywords</b>	Animation, Motion Graph, Solver, Constraints
<b>Sent to peer reviewer</b>	Yes
<b>Peer review completed</b>	Yes
<b>Circulated to partners</b>	No
<b>Read by partners</b>	No
<b>Mgt. Board approval</b>	No

## Document History

Version and date	Reason for Change
1.0 09-10-19	Document created by Dr Mungo Pay
1.1 02-12-19	Version for internal review
1.2 20-12-19	Final version for submission

# Table of Contents

<b>EXECUTIVE SUMMARY</b>	<b>6</b>
<b>BACKGROUND</b>	<b>6</b>
<b>INTRODUCTION</b>	<b>6</b>
Main objectives and goals	7
Methodology	7
Terminology	7
Convention	7
Relation to the Self-Assessment Plan (D1.2)	7
<b>MOTION GRAPHS</b>	<b>8</b>
Use Cases	8
Path Following	9
Labelled Animation	9
Transition Map	10
Pose Similarity	10
Local joint positions	10
Local joint velocities	10
Root joint world velocity	10
Root joint world angular velocity	10
Graph construction	11
Searching for Motion	12
Path Similarity	12
Labels	13
Search Optimizations	13
Integration with deliverable D5.4: Tools for Editing Mo-Cap Data	<b>14</b>
Combining Tools	15
Evaluating Animation using Laplacian Shape Deformer	15
Current Limitations	16
X-Z Solves	16
Interactive Editing Performance	17
Spatial Constraints with User-Defined Frames	17
<b>Houdini integration</b>	<b>17</b>
Wrapping the core functionality	18
Creating custom packed primitives	18
Animation editing tools	18
Converting Houdini data to constraint data	19
Viewport interaction	20
CTAM full Houdini integration	21

<b>Conclusion</b>	<b>26</b>
<b>References</b>	<b>26</b>
<b>Web references</b>	<b>26</b>
<b>Acronyms and abbreviations</b>	<b>26</b>

## 1 EXECUTIVE SUMMARY

When creating crowd scenes for VFX productions, artists rely either on an animation library comprising clips created on previous productions, or a mo-cap department creating bespoke motion for their particular show. At DNEG, whilst there is tooling to setup and trigger transition points between animations, this tooling currently relies on a simulation framework. Whilst extremely prevalent throughout the industry, and incredibly powerful, simulation has associated costs, including a lengthy iteration cycle for complex setups. Because of this, DNEG has a goal to move away from simulation frameworks so as to increase the speed of turnaround of shots. As such, a new approach is proposed to allow artists to define and interact these transition points.

In this deliverable, we describe work done to automatically create a motion graph from a set of input animations by analysing optimal transition points, and constructing appropriate transitions. We also detail the integration of the motion graph with the *Constraint Based Path Editing Tool* described in Section 5.1 of Deliverable 5.4. This *Combined Trajectory and Motion Solver*, or *CTAM* solver as it will be referred to in this document, is then extended to allow for additional constraints to augment its capabilities.

When combined with user facing tooling, this new method provides a powerful alternative to simulation techniques, whilst maintaining user control and speeding up the iteration cycle for artists. The tools have yet to be tested in a production environment, so the extent to which this tooling will improve the speed and efficiency of representative shots is as yet unknown. However, initial testing is extremely positive.

## 2 BACKGROUND

Work package 5 focuses on asset transformation and this document describes the work done in relation to WP5T3. It also acts as a companion piece, building on some of the work outlined in WP5T2 which is described in deliverable D5.4. In that deliverable, a trajectory editing toolkit is described which is extended here. For more information about the techniques employed to produce that work, the reader is directed to that document.

The tools that were specified to be included in this delivery document were — “*Various tools implementing methods from motion graph-related publications.*”

As such, this document describes the work done to augment and extend the trajectory editing tools of WP5T2. It describes some use cases that previously would be satisfied by a simulation framework, whilst assessing whether the same requirements can be addressed using a more *artist-driven* framework.

This work is designed to both speed up the turnaround of shots, making the work both cheaper and of a higher quality, whilst also reducing the requirement for additional assets to be created from either the animation or mo-cap departments.

## 3 INTRODUCTION

To address the requirements described in WP5T3, this deliverable focuses on tooling for asset synthesis, focussing on the construction of new animation data. The tooling presented allows artists to automatically generate a motion graph from input animations, label sections of animations with specific actions, and specify constraints to alter the animation. This is all done using interactive viewport controls.

Chapter 4 introduces motion graphs as a concept, and some of the mechanisms by which they can be constructed. It then details our implementation, and some of the design choices that were made.

Chapter 5 discusses the integration of the motion graph work with the trajectory editing toolkit detailed in deliverable D5.4 along with some optimisations that were adopted to speed up evaluation.

Chapter 6 sets out the integration of the tooling described in chapter 5 into SideFX Houdini<sup>1</sup>. It describes the use of viewport interaction tools and how attribute data is converted into our core data-structures.

Chapters 7, 8, 9 and 10 cover conclusions, references, web references and abbreviations.

### 3.1 Main objectives and goals

---

The objectives of this work package are simple:

- To improve the quality of crowd shots at DNEG by speeding up the iteration cycle of artists.
- To allow artists to re-use previously made assets in a new context.

These objectives serve to make the production of crowd shots more efficient and therefore less expensive.

### 3.2 Methodology

---

The work was structured in such a way so as to be both as performant as possible, whilst also having user facing interactive tools that artists can use. As such, the approach taken was that of a C++ backend and a DCC specific front-end.

Implementing the toolsets in this way ensured that we not only had a large degree of control when it came to optimisations, but also allowed us to be DCC agnostic in terms of the core functionality. This is important when developing tooling for VFX as we are not fully in control of the direction of development for third party software. Whilst we are in constant communication with the producers of most third party software, priorities can change, so we need to be as flexible as possible to those changes.

For the purposes of this project, all of the user interfaces for the tooling was developed in SideFX Houdini™ as this is currently the software used at DNEG for crowd creation.

### 3.3 Terminology

---

**Motion Graph:** An interconnected graph of motions that can be traversed to produce dynamic motion based on some path based input.

**Packed Primitive:** A SideFX Houdini™ concept allowing binary data to be passed through the node graph.

### 3.4 Convention

---

In this deliverable will use *italics* for emphasis and monospace for code and pseudo code.

### 3.5 Relation to the Self-Assessment Plan (D1.2)

---

As described in the self-assessment plan, over the course of the development of the tooling, we have continuously been doing benchmark testing and in-house user testing. Moving forward from this deliverable, user evaluation in experimental production and pipeline evaluation between months M25-36 will be carried out, and described as part of work packet 8 in deliverable D8.3.

---

<sup>1</sup> Houdini is a procedural animation application used widely in the visual effects industry. It was developed by software studio SideFX.

## 4 MOTION GRAPHS

Crowd shots will often require characters to follow user defined paths, perform actions for varying lengths of time, and display variation amongst up to hundreds of thousands of characters. With only a limited number of animations available to fulfill these requirements, artists need to loop, trim and blend these animations which, when done manually, can be a long and laborious task.

If, for example, an artist needed an animation in which the character stands for 5 seconds, walks along a path, jumps over an obstacle and continues walking, they would need to loop animations and blend from one animation to another. This would require them to scrub through a number of animations, visually comparing frames that are suitable to blend, then requesting an animator to produce the blended animation.

The construction of anything more complex than this simple example can take a long time, and would need to be done for every combination of bespoke animations that were required for their show. This could also include multiple versions of the same animation type, but with different transition points to add in some variation so that all characters aren't using the exact same motion, commonly known as *twinning*. If we introduce the trajectory of those characters into the requirements, the number of combinations becomes completely unmanageable.

To address this problem, we have implemented a Motion Graph [1] implementation which can automatically produce seamlessly blended animations given user-defined constraints.

Motion graphs offer a data-driven technique to very quickly build the best available animation using a database of animation defined by the user. The animation that is output is subject to user-defined constraints, generally a path for the character to follow and labels which can be used to specify that certain sections of animation are included.

It does this by comparing all animations in its database, finding poses that are similar, and defines them as blend/transition points. If pose x from anim A is found to be a good match for pose y of anim B, it will create a set of blended frames of animation from A to B at those points.

Sections of animation from the inputs, and the newly created blended animation are then stored as nodes in a graph which can be searched. The best path through the graph is then evaluated based on how closely they follow an input trajectory and adhere to label constraints. The ultimate goal is for it to return an optimal animation from the best branch of the graph search.

### 4.1 Use Cases

---

Some form of automatic motion generation like this is heavily used in the games industry because they have a requirement on realistic animation being generated based on user inputs. This section details a couple of the added benefits of such a system for visual effects (VFX) and in particular crowd generation.

### 4.1.1 Path Following

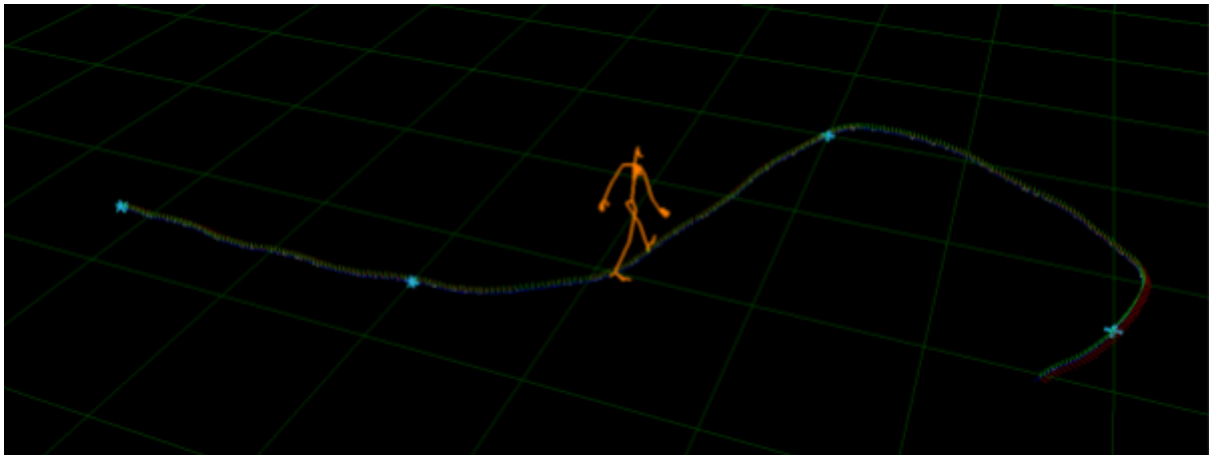


Figure 4.1: Appropriate turning animation is found based on a path defined by 4 positional constraints.

In this example, the motion graph is constructed using a single animation clip of a character walking forward, and performing a number of turns. The input animation comprises the character walking forward in a straight line, turning 90° right, continuing straight, then turning left 90°. The character's path is determined, in this case, by a number of positional constraints.

The animation built by the motion graph which seamlessly loops the straight line walk for as long as it needs, transitions into a left turn then two right turns with additional frames of straight walking in between.

### 4.1.2 Labelled Animation

Trajectory alteration is not the only use case for a motion graph. It is also possible to label specific sections of motion, or whole animations with semantically meaningful information and use that as an input to direct how the graph is traversed.

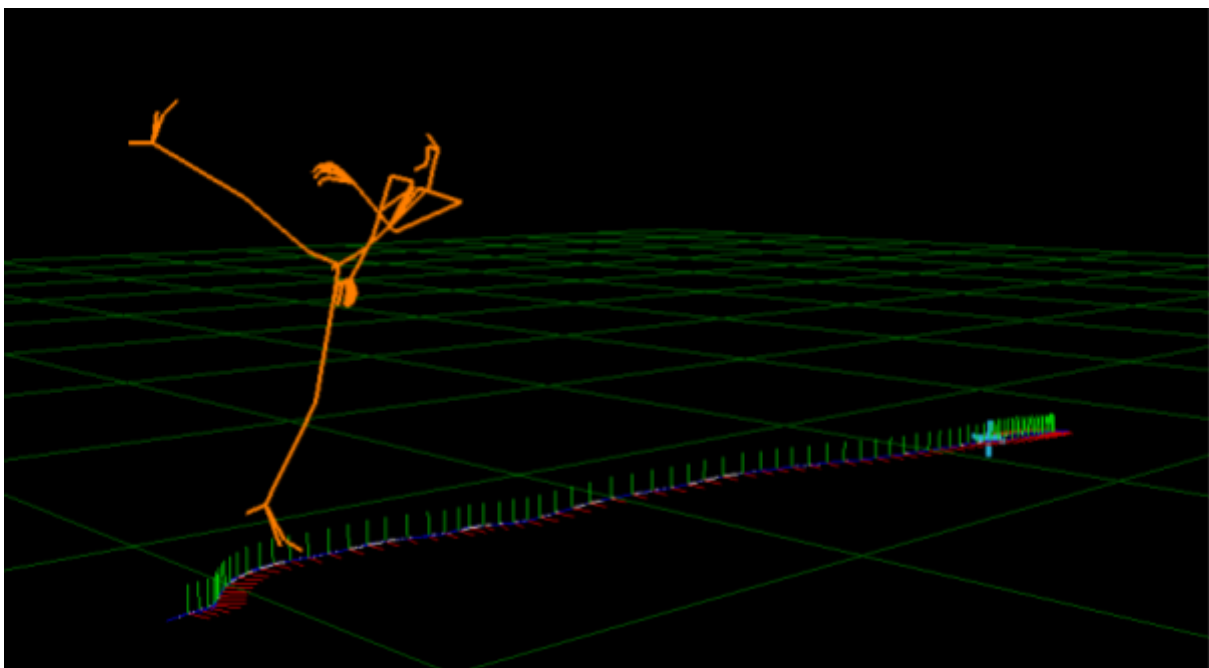


Figure 4.2: A running animation transitions into a kick can be labeled as both for later motion graph traversal.

A common crowd shot in film involves people in stadiums watching and cheering. Although a motion graph is especially useful for locomotion, labels and a stationary path can still be used to easily put together a standing and cheering animation, which can be randomized and timed for a realistic stadium crowd.

## 4.2 Transition Map

---

The detection of good transition frames, which are used to blend from one animation to another, is a preprocessing step in which every frame of animation is not only compared with every frame of all other animations, but also itself. From this comparison, each pair of frames is assigned a cost.

Depending on the input density of the graph, a number of pairs with the lowest cost are used as the transition points in the graph [1].

### 4.2.1 Pose Similarity

We currently use four metrics for evaluating if two poses match well [2] —

#### 4.2.1.1 Local joint positions

For every matching joint in the skeletons of the two poses compare the difference of their positions local to their parent joint.

#### 4.2.1.2 Local joint velocities

Calculated as the change in local rotation from the next frame's joint position to the previous. Local joint positions can find matching poses but velocity can differentiate poses that are travelling in different directions for example, a forward walk pose and backward walk.

#### 4.2.1.3 Root joint world velocity

Calculated as difference in root joint world position from the previous frame to the next. Comparing poses from a walk and run will return a high cost.

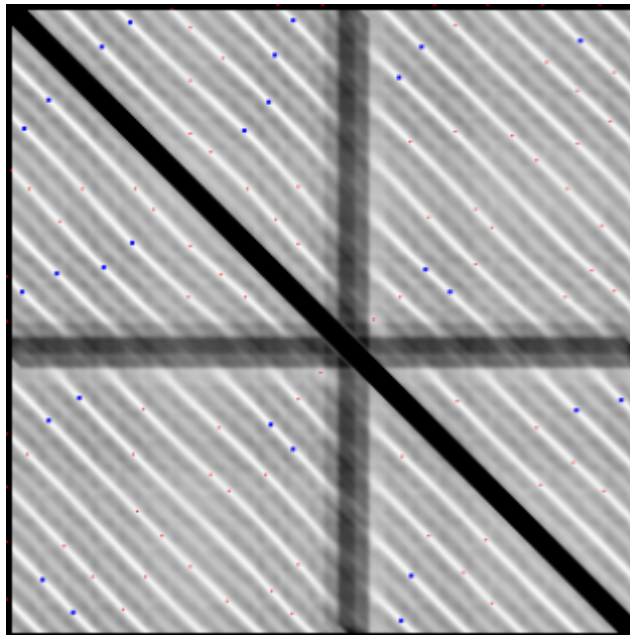
#### 4.2.1.4 Root joint world angular velocity

Change in root joint world rotation from the previous frame to the next. Differentiates poses that may be turning in different directions, or constant rotation animations to turning ones.

The aim in selecting transition points is firstly to find strongly matching poses, but also to spread the transitions out amongst the animations so the graph is well connected and capable of producing any number of frames of animation. To do this we select the local minima with the lowest costs.

If frame  $x$  and  $y$  of two animations evaluate to a low cost, it is likely their neighbouring frames will too, which would likely cluster the transitions in one section of the animation [1].

When comparing one animation to itself, comparisons of frame  $x$  and frame  $y$  are not accepted as options for transitions if  $y-x$  is within a certain radius, as these will be found to match well but will not provide us with a useful transition that reaches another part of the animation. This explains the black diagonal line in figure 4.3.



A walk animation is compared to itself, where the  $x$  and  $y$  pixels represent frames of the two animations. White areas have a low transition cost and dark are high. Blue dots signify pairs of frames chosen as transition points, red are additional pairs that would make good transitions.

Figure 4.3: Visual representation of optimal transition points in an animation measured against itself.

### 4.3 Graph construction

We use the list of pairs of transition frames to split up the input animations and place the segments into nodes which will make up the directed graph. Nodes are always connected in one direction to nodes that represent an animation that can be appended. At transition frames, we create a new segment of blended animation which is stored on a node and then inserted into the graph.

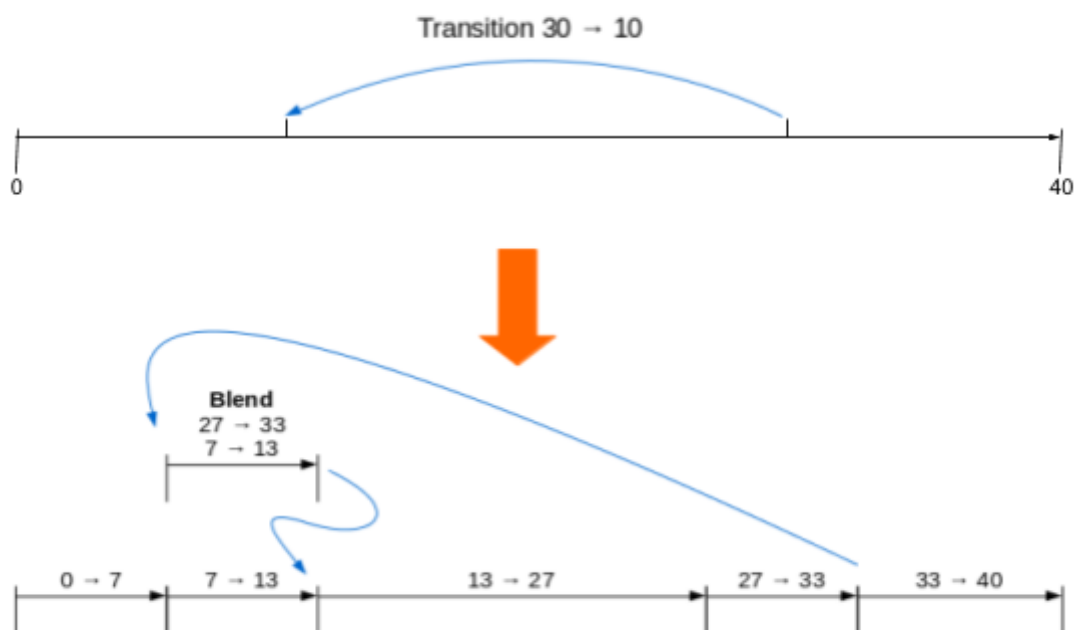


Figure 4.4: Visual representation of node graph construction where each segment represents a node in the graph and the blend duration is 7 frames.

When searching the graph we do not want to visit any nodes that lead to a dead end [1]. Therefore nodes which do not have any outgoing edges leading into the best connected nodes of the graph are removed, as these will limit the breadth of animation available or produce animation that will not be able to loop.

To do this we run a graph Strong-Connected-Components algorithm, available in the Boost-Graph-Library [3], which can be used to filter out nodes that are not part of the best connected section of the graph.

## 4.4 Searching for Motion

---

Building good animation is an optimisation problem in which we traverse the graph, evaluating each node's animation to assign a cost. The resultant animation is the one with the lowest accumulated cost at the end of this traversal. A low cost animation is one that follows the path input by the user as closely as possible, and adheres to any input label constraints.

Starting at a given node, we sequentially visit any nodes connected to each other through their directed edges, branching when nodes have more than one edge. Each visit involves appending the node's animation to an animation built up from previously visited nodes in this branch. The cost of the most recently appended frames is then evaluated and added to the summed cost of that branch.

For the standard motion graph implementation, we have implemented a path similarity metric, described here. This works well for when we have an input trajectory driving the output, but can have some limitations when we need the output motion to exactly follow a path. These limitations will be described in section 5.1.1.

### 4.4.1 Path Similarity

We find the squared distance from the root position of each evaluated frame to an interpolated position on the path [1]. Before evaluating, the animation is transformed so that the path starts are the same. Moving forward, it would be good to evaluate whether this can be improved by using techniques that fit the entire animation to the path, such as SVD [4].

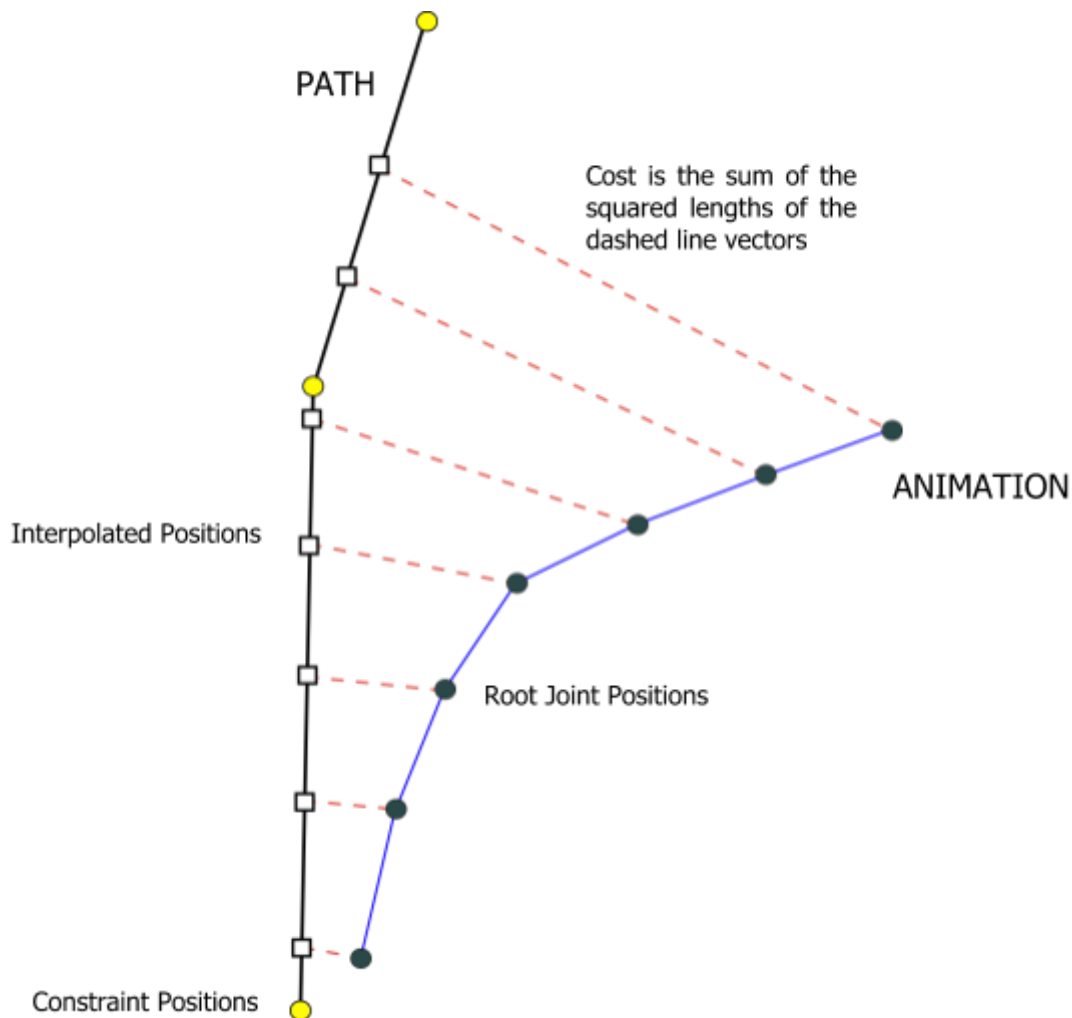


Figure 4.5: Determining a cost function for path similarity.

#### 4.4.2 Labels

For the purposes of the motion graph, a label is a string tag associated with a frame range semantically describing sections of an input animation. When a user wants the motion graph to insert a specific style of animation, they must first label their input source animations as desired. For example, any animations with a walking section may be labelled "Walk" on those frames.

It is then possible for the artist to create label constraints which guide the motion graph to output animation where the specified frames contain that label. This is enforced during the motion graph search by applying a high cost onto frames that do not satisfy the label constraints, which pushes the search into transitioning into a correctly labelled animation as soon as possible.

#### 4.4.3 Search Optimizations

On any search of the motion graph we find an animation that has the lowest cost where any search of a branch is terminated, and when the animation of that branch has enough frames to satisfy the constraints. A well connected graph with a few hundred frames of animation may have millions of potential branches meaning a full depth-first-search of all branches is impractical.

Instead we employ a branch and bound search method [1]. During the search, the optimal animation and its cost is cached, and any branch that accumulates a score greater than this will immediately terminate. This greatly reduces the number of branches searched, and for very large graphs in our testing, could reduce the evaluation time from hours to seconds.

However, the number of branches needing to be searched increases exponentially as the desired animation length increases. Therefore we can optimize the search by evaluating a fixed number of frames at a time, then restart the search. To ensure that the quality of animation is not too adversely affected by this technique, the search restarts at an earlier point than where it finished last time to ensure a few frames of padding are incorporated. Whilst the effectiveness of this particular optimisation varies with animation length, the estimated speed up for common use cases was between 5 and 10 times faster.

## 5 Integration with deliverable D5.4: Tools for Editing Mo-Cap Data

In work package D5.4 we implemented a constraint-based path-editing tool which uses a Laplacian shape deformer to edit the motion path of an animation while preserving the local features of that path, resulting in animation with minimal foot sliding or loss of detail from the source mo-cap data [5].

Although both the motion graph and deformer can be decoupled and are useful tools on their own, combining them elevates the results either one can produce and give us an extendable solution for interactively creating realistic animation. For the purposes of this document, we will refer to this solution as a *Combined Trajectory and Motion* solver, or *CTAM* solver.

The user interacts with the solver by providing a database of animations for the motion graph to use, and a set of constraints which are used by both the motion graph and the Laplacian deformer.

The data flow of the CTAM solver is shown in figure 5.1.

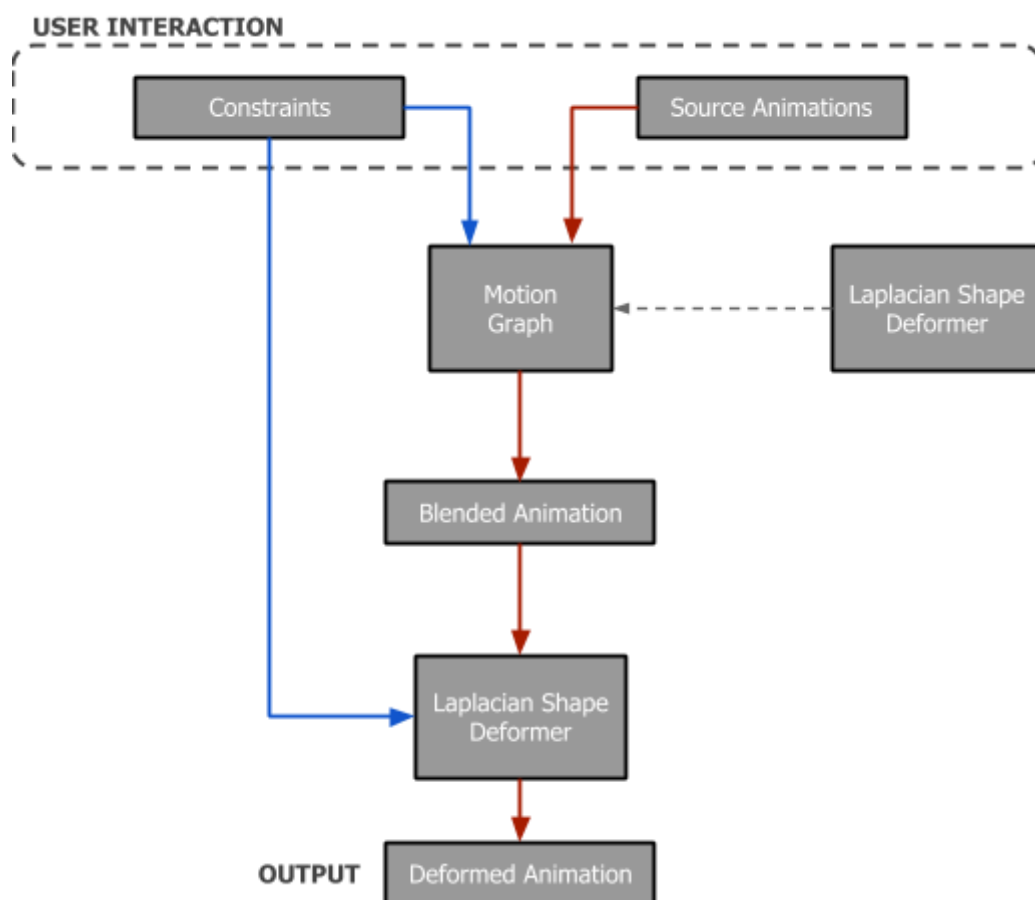


Figure 5.1 CTAM solver data flow.

## 5.1 Combining Tools

A motion graph returns the best animation available based on the source animations that have been used to construct it. It is therefore highly dependant on the source animations for how well the output animation can follow the given path. For example, if we construct the graph with walking animations including  $90^\circ$  turns, but given a path with a  $45^\circ$  turn, the animation shown below in figure 5.2 may be the lowest cost choice that the motion graph can return.

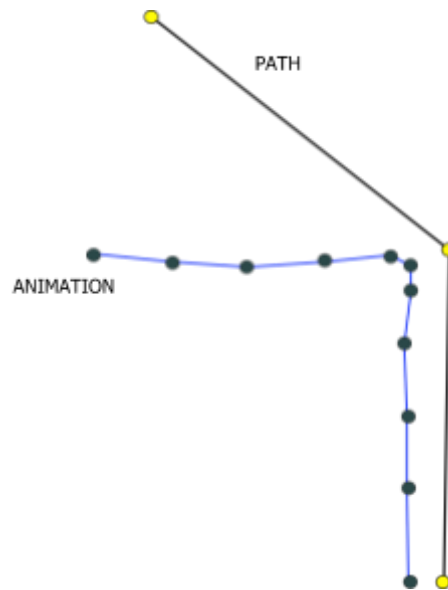


Figure 5.2: Non-optimal motion graph output based on input trajectory.

We therefore need to edit the root positions of animation so it can more closely follow the path. For this we use the Laplacian shape deformer described in deliverable D5.4.

As described in that document, this path editing tool minimizes deformation as much as possible, but without any motion synthesis, any large edits will still noticeably scale the path. This can result in foot-sliding, and unnatural speed of movement in animations.

However, by inputting the deformer with the animations that have been synthesised from the motion graph using the same constraints, a small amount of deformation is required, so foot sliding is minimised.

### 5.1.1 Evaluating Animation using Laplacian Shape Deformer

Feeding the output of the motion graph into the deformer ultimately produces seamless animation that fulfills the continuous requirements of the motion solver. However, it is also possible to integrate the deformer into the evaluation step of the motion graph search, improving the selection process for the best animation.

When evaluating an animation, we use the Laplacian deformer to solve the animation for the user-defined constraints, then measure how much deformation is required. We call this measurement the *Deformation Energy* [5]. This is defined as the change in the local coordinates of each position of the path. The greater the change in positions, the higher the cost assigned to that animation.

The difference in the Deformation Energy method and Path Similarity method (sub-chapter 4.1.1) can be illustrated in the evaluation of the following two animations —

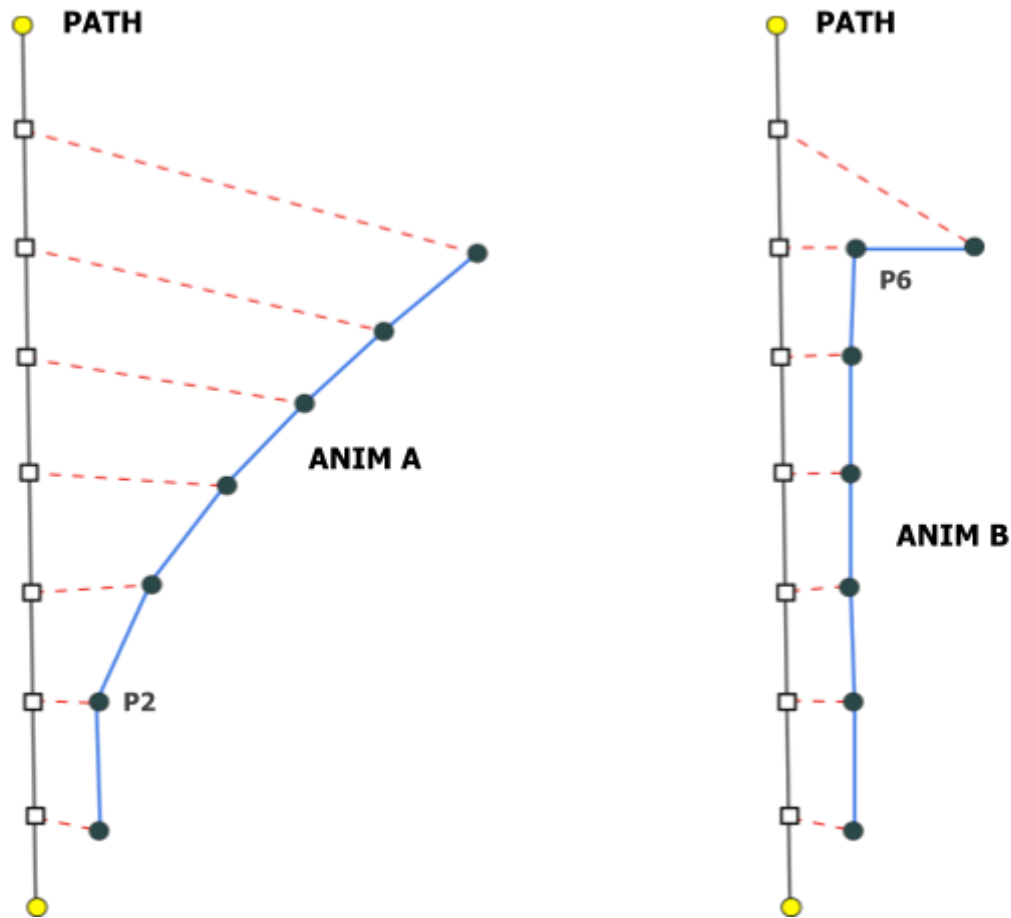


Figure 5.2: Comparison of two animation evaluation methods, Path Similarity and Deformation Energy.

Using the Path Similarity method, Anim B would be returned as it has a smaller accumulated error in its positions. However, when using the Deformation Energy method Anim A would be chosen, as only a small amount of deformation at point P2 is needed to straighten the animation and fit it to the path. Anim B, on the other hand, would need a larger deformation at P6.

When the animations output by the motion graph are ultimately deformed to fit the constraints, Anim A gives us a higher quality animation with less obvious deformation. The deformed animation of B would display foot sliding and shifting of weight around P6 as the right angled cut turn is straightened to fit the path.

## 5.2 Current Limitations

The solver is a powerful animation editing tool and can be extended with additional constraint types or animation post-processing steps such as layering or IK. There are however still limitations in its usability. This section details some of the elements that could use improvements.

### 5.2.1 X-Z Solves

The Laplacian shape deformer currently only solves for values of root joint positions on the x-z plane, whilst the y position is simply projected onto the floor. Further investigation and work is needed to produce a full 3D solve, and to select appropriate animations in the motion graph which depend on the changes in height of a trajectory.

This would allow for fast creation of animations moving on stairs and steep slopes, which are typically among the more challenging animation requirements.

### 5.2.2 Interactive Editing Performance

The performance goal of the solver is for users to be able to click and drag constraints in an interactive viewport, and create new animations in real time. Although the solver can reach speeds of up to 60fps with a low number of animations in the motion graph and short output animations, interaction becomes less reactive as these factors increase.

The limiting factor in the solver's performance is currently the motion graph search, for which the time complexity can increase exponentially as the density (number of nodes) of the graph increases. Ideally a flexible database of animations could be re-used in the motion graph with various types of locomotion and actions, but in order to keep the density of the graph low a user would need to tailor the animations used to that only fill the requirements of their current scene, potentially using different motion graph instances for different agents.

### 5.2.3 Spatial Constraints with User-Defined Frames

Currently to describe a path for an agent to follow, the user may place World Position constraints which specify the root position at which a given frame of the output animation should be positioned. Although this gives the user a low level of control of the animation produced, this can easily lead to noticeable stretch of the animation when too few frames are specified for the distance between sequential constraints.

For example below, two constraints have been set on frame 0 and frame 6, at a distance further than any animation in the motion graph travels over 6 frames, therefore stretching the path.

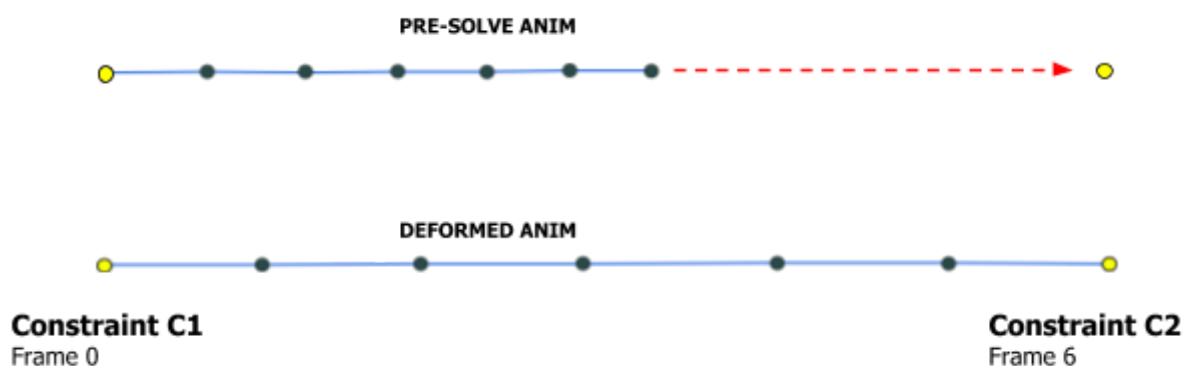


Figure 5.3: Stretched keyframes resulting from spatio-temporal constraints.

To obtain high quality animation the user must either move the constraints closer or increase the frame number of C2, which can take a large number of adjustments and iteration.

To address this, it would be possible to introduce a constraint that creates an ordered path with no requirements for frames to be set. This could potentially offer a simple method for creating animation with minimal foot sliding, but with less control over the timing of the animation.

## 6 Houdini integration

When integrating new functionality into any DCC, it is important to design the tooling to match standard workflows so as not to alienate the users who are used to working in particular ways. As such, a lot of care and attention was spent ensuring that the workflows married up with standard Houdini best practices at DNEG. An added benefit of this approach, was that we were able to leverage a lot of standard tooling and UIs that are provided by SideFX, which reduced the development burden.

## 6.1 Wrapping the core functionality

This section details the building blocks developed so that we could convert the data from native houdini data into the C++ data objects required for the CTAM solver to work.

### 6.1.1 Creating custom packed primitives

Houdini allows for custom data to flow through its node graph using a concept known as *packed primitives*. A packed primitive intended as a way to reduce the memory footprint of many instances of heavy geometry, whilst providing some viewport representation to the user. Houdini ships with many packed primitives including an *Agent* packed primitive which allows crowd data to be encapsulated and passed through the graph. The concept also provides a mechanism to wrap custom blind data, which will be interpreted by the nodes that know how to evaluate it.

To allow the processing of animations, we needed to be able to pass our custom animation and motion graph data structures through the houdini graph for later use as input animation for crowd agents.

The reasoning behind wanting to create these custom packed primitives is that, whilst Houdini provides many contexts in which that data can be manipulated (channel data, compositing etc.), the crowd workflows are in a geometry context which only operates on geometric data.

The level to which attributes are exposed through the Houdini attribute interface is up to whomever is exposing the data. For our purposes, it was felt that the ability to manually edit any of the animation data through scripting languages was not required, so a rather shallow integration was adopted. This meant we only needed to expose serialisation/deserialisation mechanisms, and drawing code. This decision might need to be revisited in the future if scripting does become a requirement.

### 6.1.2 Animation editing tools

To allow artists to quickly edit sections of animation we developed some trimming and looping tools along with tooling that allows them to label sections of animation with semantically meaningful tags.

The trimming tooling allows artists to minimise the amount of animation data going into the motion graph, which in turn makes the solver more performant. This is because the size of the motion graph, and therefore its speed of evaluation is determined by its density.

The tools shown in figures 6.1 and 6.2 allow artists to quickly and easily trim, or loop animations.

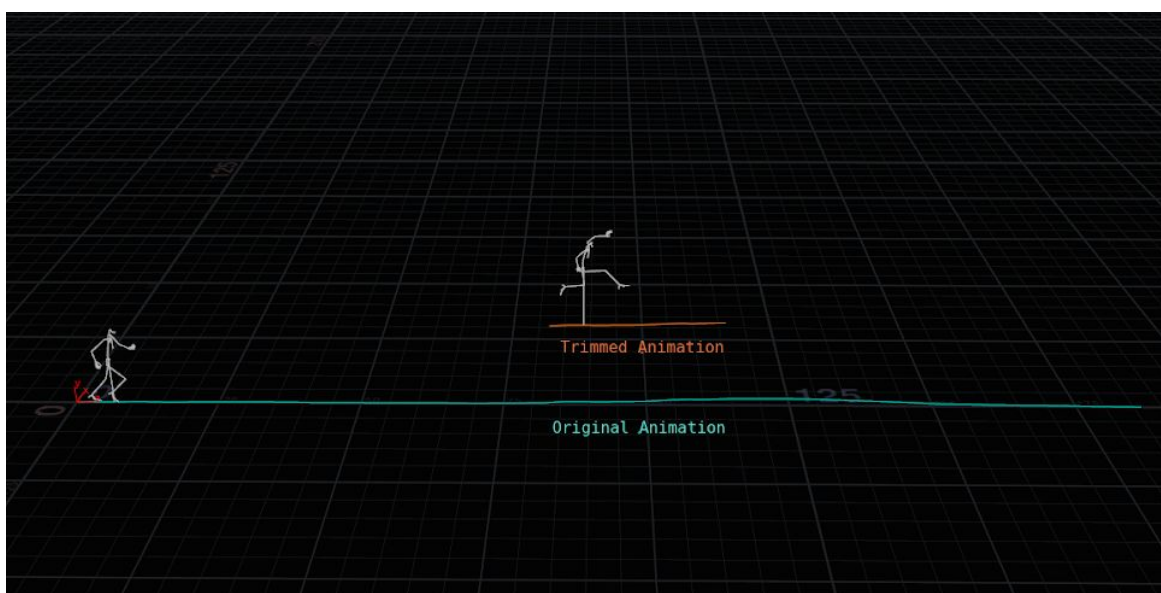


Figure 6.1: Tools for trimming animations

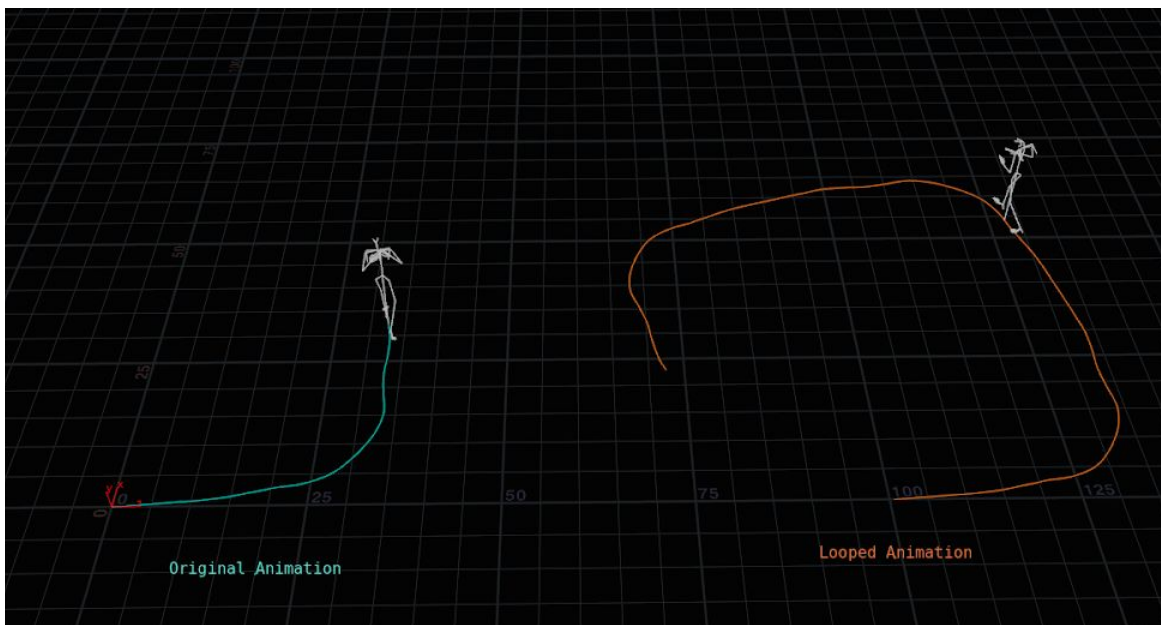


Figure 6.2: Tools for looping animations

Figure 6.3 shows how sections of the motion can be labelled. This can be done on an entire animation, or, in this example, multiple subsections of an animation.

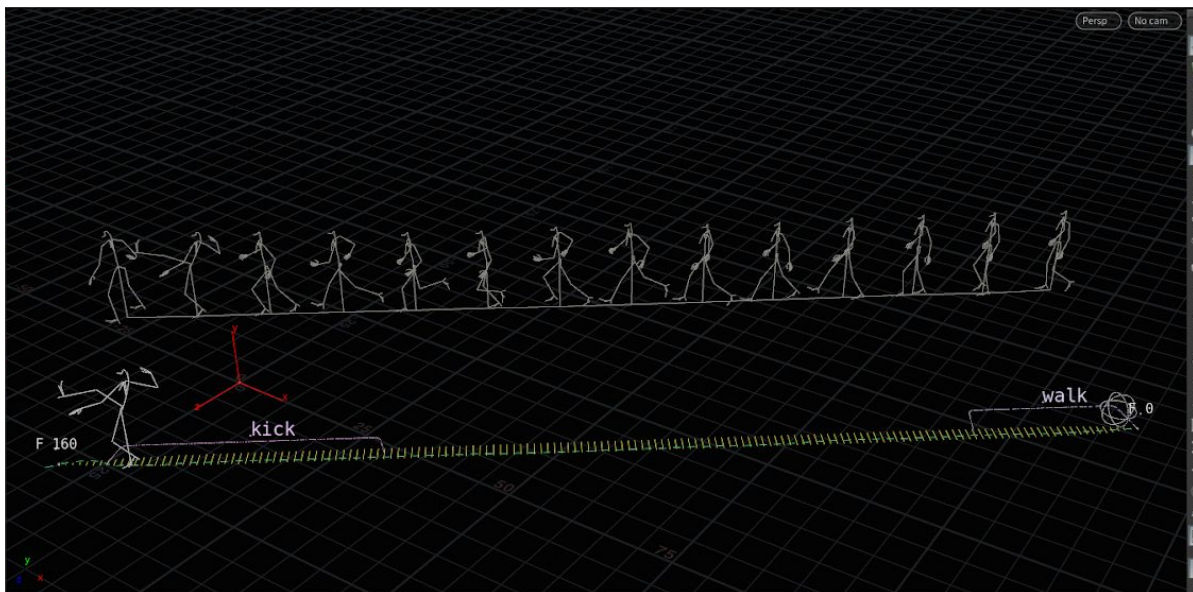


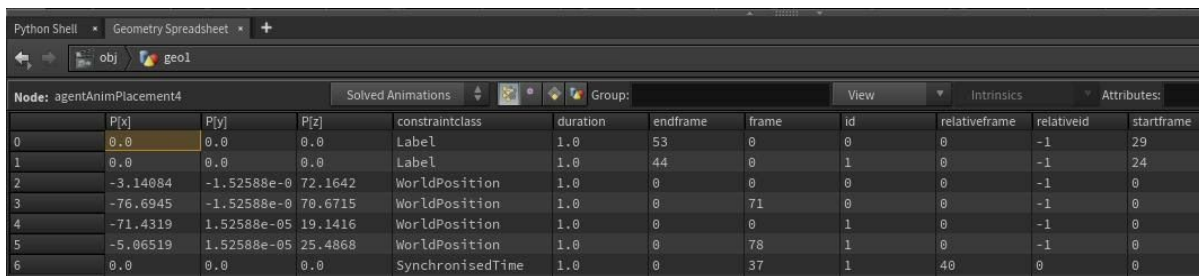
Figure 6.3: Tools for labelling animations

### 6.1.3 Converting Houdini data to constraint data

The Houdini API allows for arbitrary data to be added to point instances, and for that data to be connected in some way to controlled by UI elements. This is the standard way that a Houdini TD will manipulate geometry as data passes downstream, and is extremely quick for prototyping.

By creating C++ nodes that interpret this point data and convert the attributes to the appropriate constraint types as C++ objects, it is possible to easily connect up the UI elements as inputs to the CTAM solver.

An example of how this point data is formed is shown below in Houdini's geometry spreadsheet (a spreadsheet view of arbitrary data associated with points or primitives.)



	P[x]	P[y]	P[z]	constraintclass	duration	endframe	frame	id	relativeframe	relativeid	startframe
0	0.0	0.0	0.0	Label	1.0	53	0	0	0	-1	29
1	0.0	0.0	0.0	Label	1.0	44	0	1	0	-1	24
2	-3.14084	-1.52588e-0	72.1642	WorldPosition	1.0	0	0	0	0	-1	0
3	-76.6945	-1.52588e-0	70.6715	WorldPosition	1.0	0	71	0	0	-1	0
4	-71.4319	1.52588e-05	19.1416	WorldPosition	1.0	0	0	1	0	-1	0
5	-5.06519	1.52588e-05	25.4868	WorldPosition	1.0	0	78	1	0	-1	0
6	0.0	0.0	0.0	SynchronisedTime	1.0	0	37	1	40	0	0

Figure 6.4: Constraint class data represented as houdini attributes.

These parameters are fed into a factory mechanism that, based on the `constraintclass` attribute, will construct the appropriate constraint, and feed in its parameters correctly for evaluation.

## 6.2 Viewport interaction

To make the tooling user friendly, the addition of viewport controls is essential. It has always been possible to provide custom viewport interaction in Houdini by implementing input hooks as exposed by the Houdini Development Kit (HDK). The downside of doing this however is that the development cycle is fairly involved, the docs are sparse, and turnaround times can be slow.

From Houdini 17, SideFX introduced a python module called *Viewer States*, which wraps the functionality of the input hooks into a higher level library. This meant that prototyping viewport interaction tools became a lot faster since any changes didn't require recompilation of the tooling and a restart of the application. For this reason, the tooling developed for this project was done using these Python wrapped *Viewer States*.

Figures 6.5 and 6.6 show how these viewer states are exposed to the user.

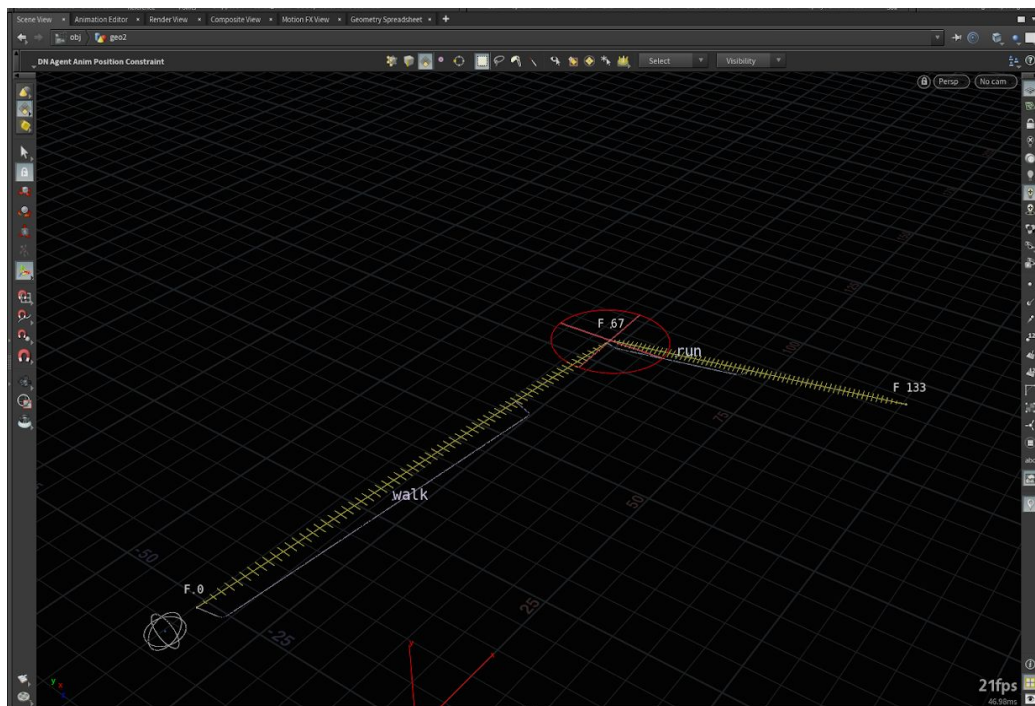


Figure 6.5: Viewport interaction to allow users to be able to quickly edit constraints

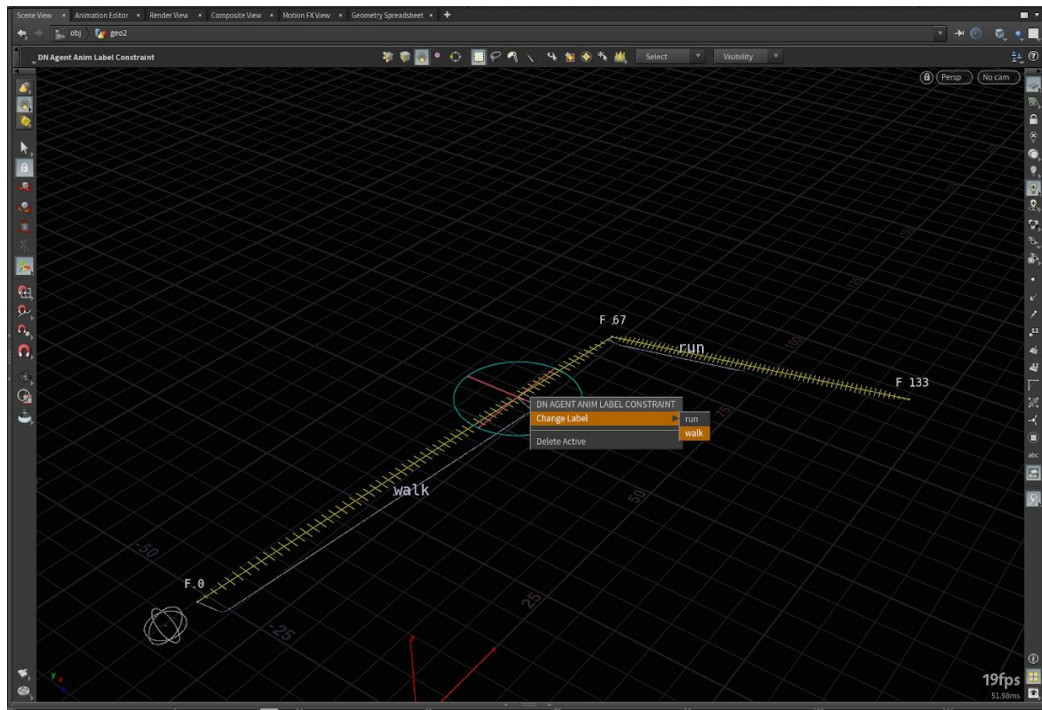


Figure 6.6: Viewport interaction with right click menu to allow users to be able to quickly edit constraints

The red selector gizmo snaps to the path widget and allows the user to quickly and easily change the position of the constraint. They can also edit the frame by using the mouse scrollwheel. Right click functionality has also been implemented, so artists can specify which sections of animation should contain certain labelled sections of animation.

The viewport interaction has been designed to allow artists to quickly and easily switch contexts so that they can seamlessly add different constraint types. This also allows for us to be extensible with the interaction, as development progresses and more constraint types are introduced.

### 6.3 CTAM full Houdini integration

The full integration allows artists to place constraints in Houdini, and solve for animation. Figure 6.7 below shows the result of a solve where four spatio-temporal constraints are placed in a scene. In this example, there is a motion graph constructed from two animations, a walk and a run. The fact that a short distance is covered between the second and third constraints means that the walk cycle is a better fit, so should be selected by the solve.

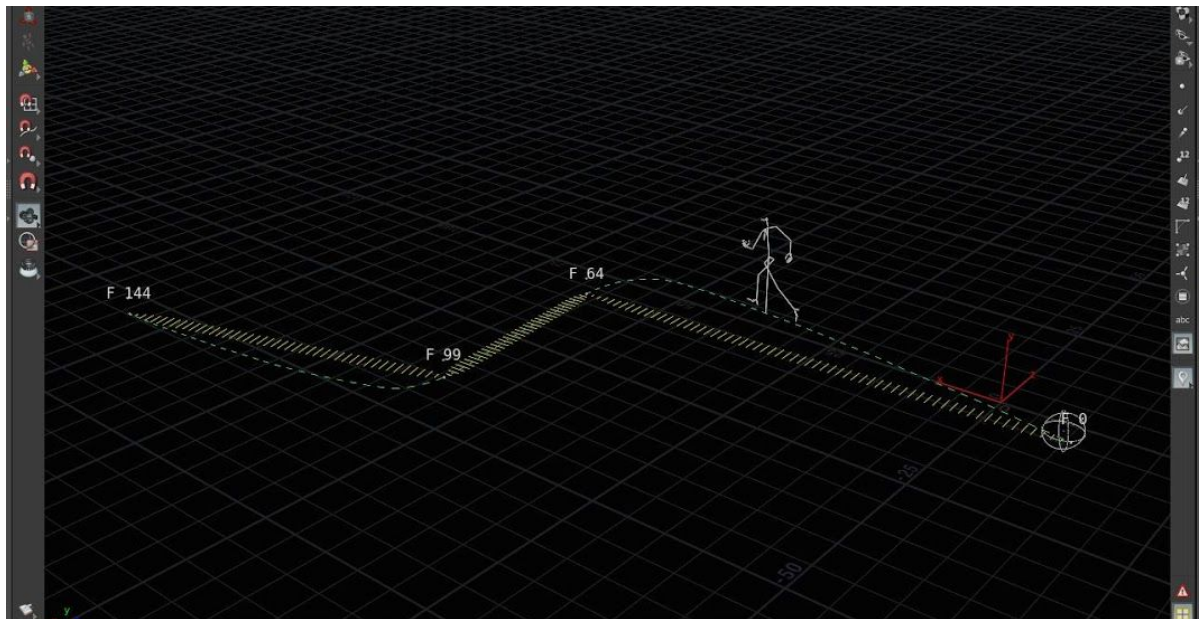


Figure 6.7: Spatio-temporal constraints deforming the characters path

Figure 6.8 shows the resultant animation where the green animation is sections of running, and orange is sections of walking. This result is achieved purely from the world position constraints determining that transitioning between these animations fit the curve better. There are no label constraints in this setup.

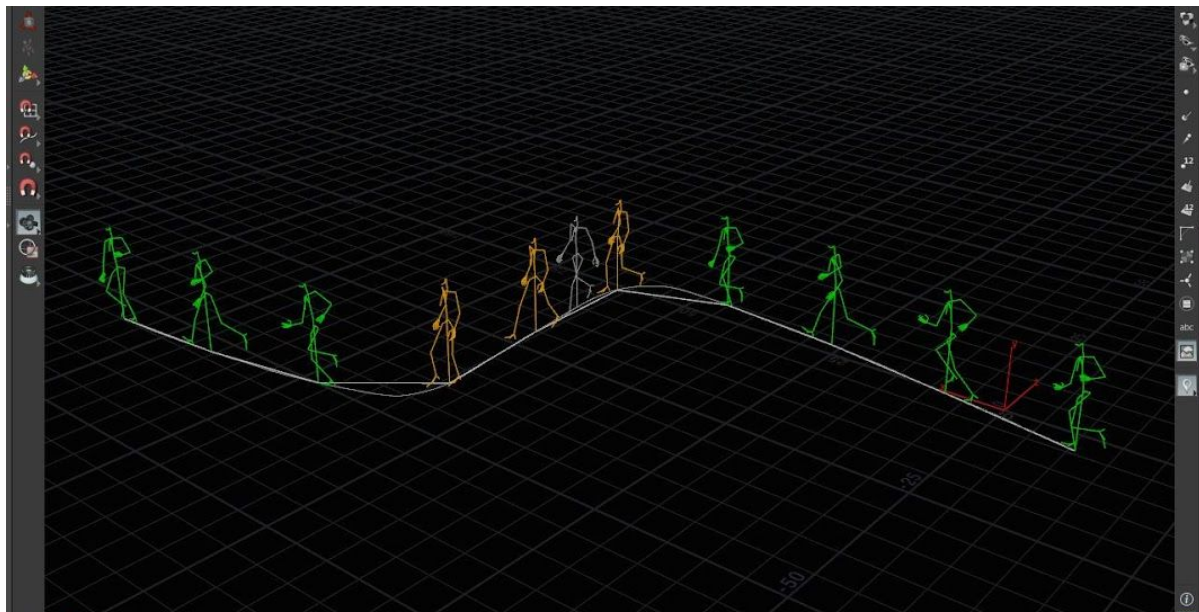


Figure 6.8: Resulting animation from constraints

For examples where we need to enforce a relational constraint between two animations, we have viewport controls that allow for that. Figure 6.9 below shows how this information is presented to the user. Here, the purple dotted line shows that one animation is constrained spatially to the other at a given frame. The blue line shows the distance and vector direction that the constraint must adhere to.

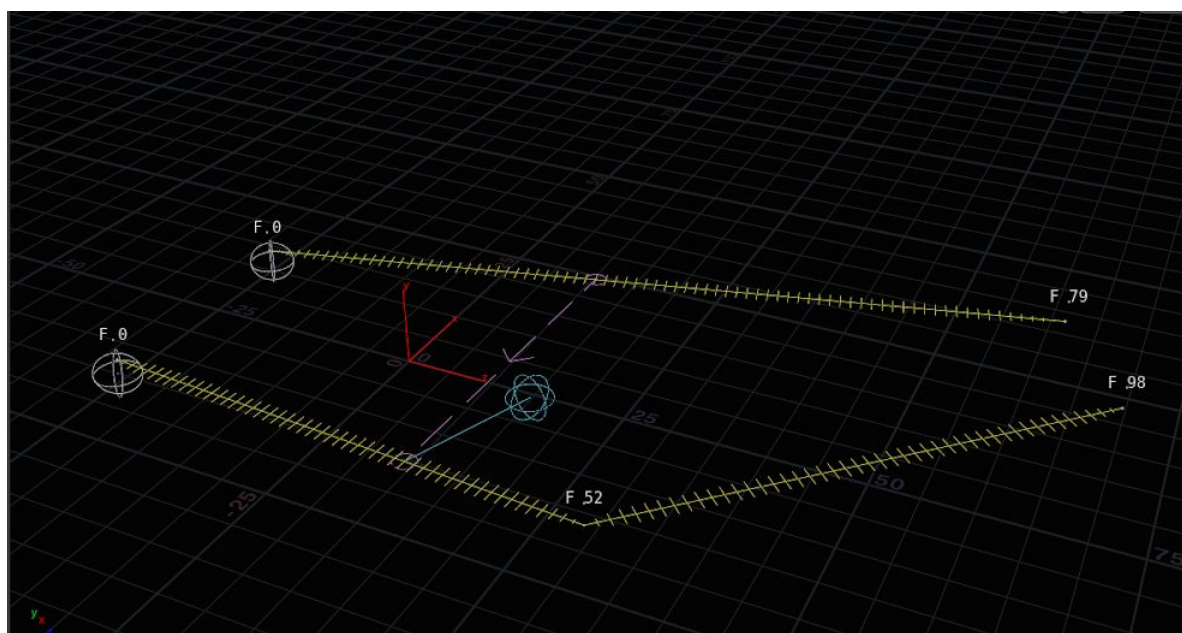


Figure 6.9: Relational spatial constraint between two animations at a given frame

When this constraint is solved for, the resulting path of the top animation is deformed to satisfy the constraint. Figures 6.10 and 6.11 show the resulting trajectory for the constrained animation, and the motion that is solved for, in this case, two running characters.

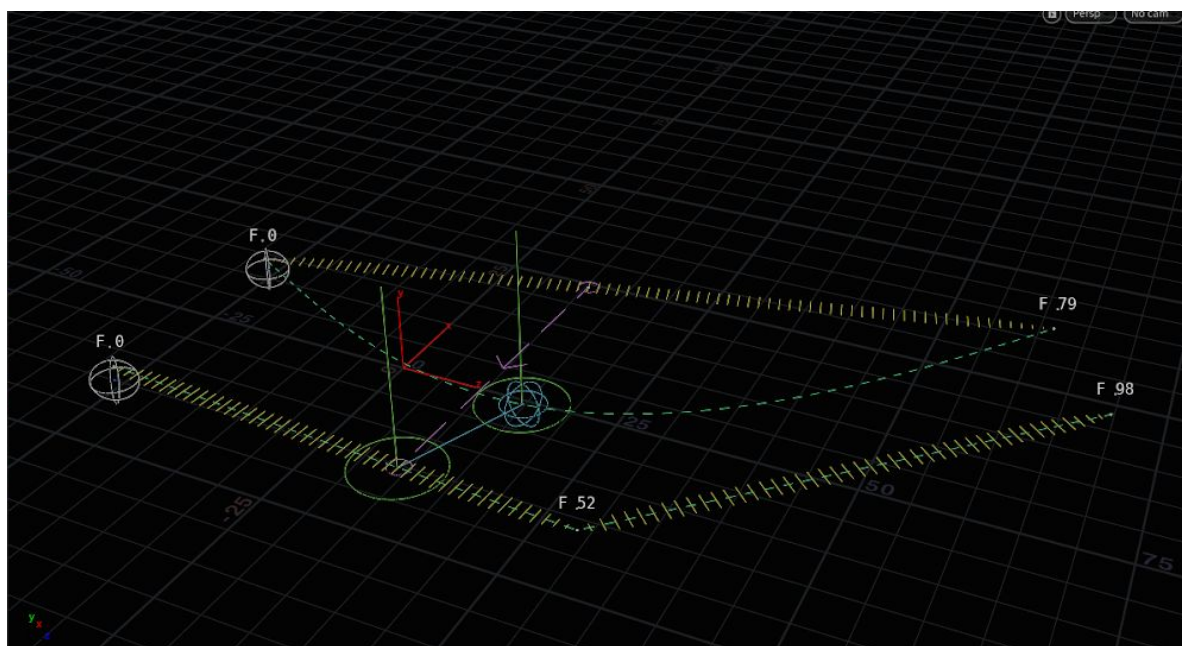


Figure 6.10: Deformed trajectory from relational spatial constraint between two animations.

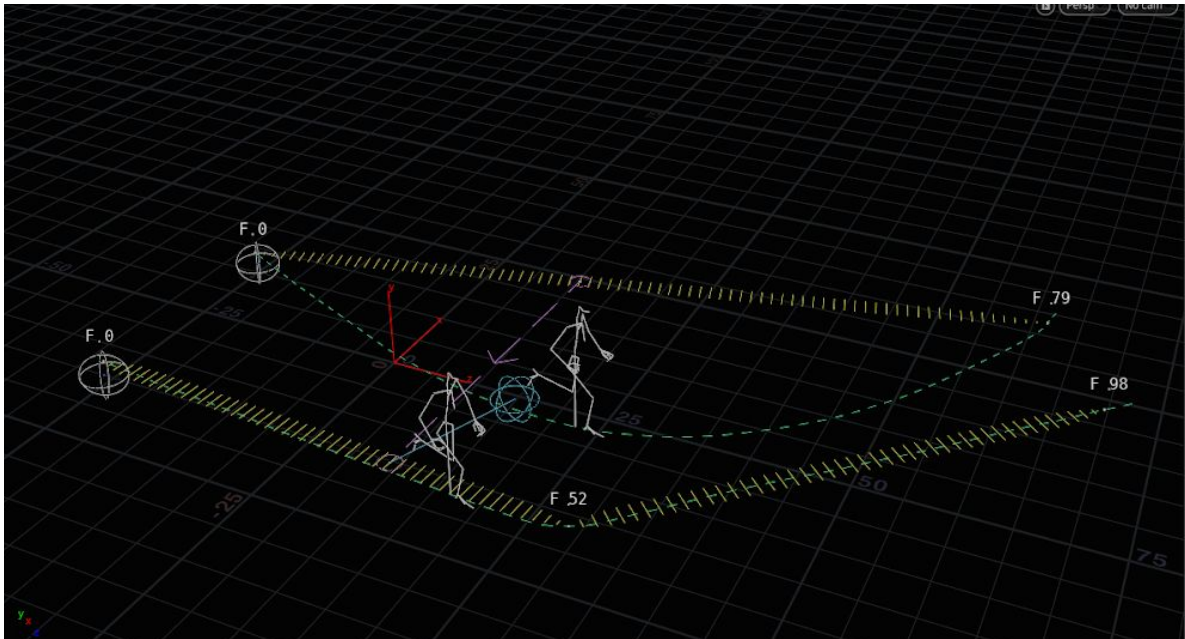


Figure 6.11: Solved animation from relational spatial constraint between two animations.

When combinations of these constraints are layered together, it starts to become possible to produce some complex interaction between agents. Figure 6.12 shows an interaction between two agents where one character kicks the other, who then falls down. This is done by enforcing relational constraints, along with label constraints, and offsetting the timing to trigger the action at the same time.

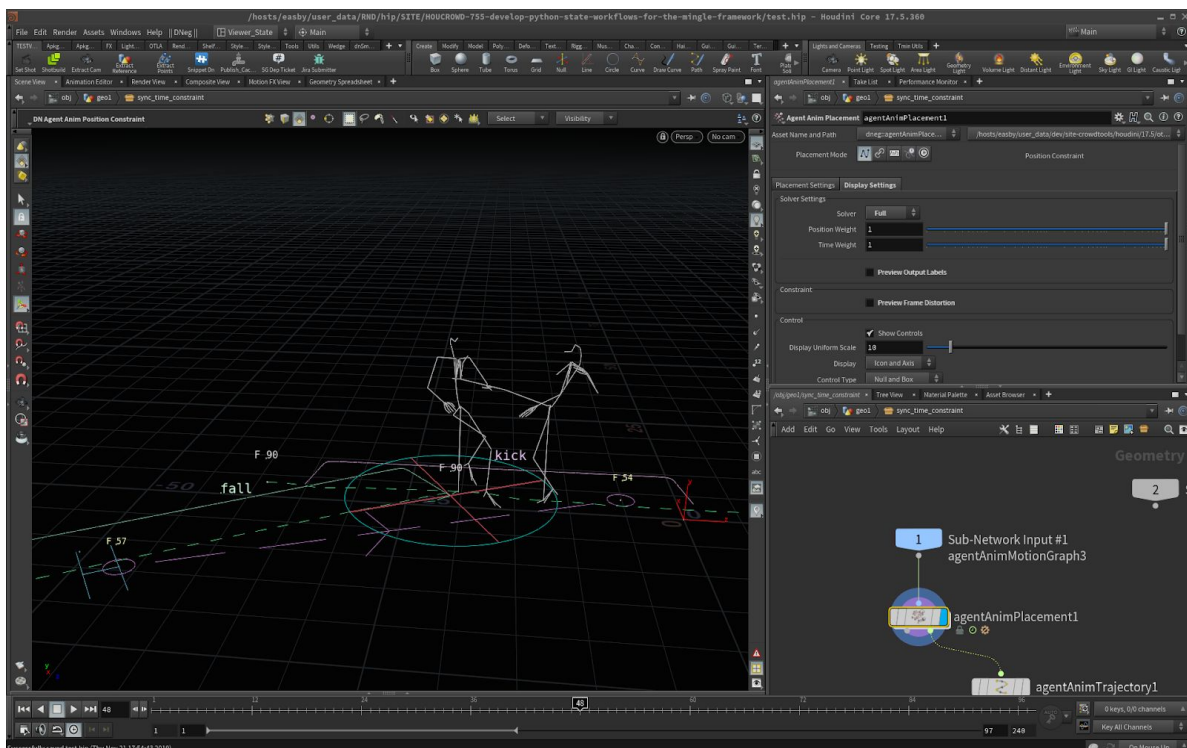


Figure 6.12: Interaction of animation produced from a combination of constraint types.

In Figure 6.13, a number of agents are constrained using both spatial and label constraints to run and then jump over boxes at a specified point in space.

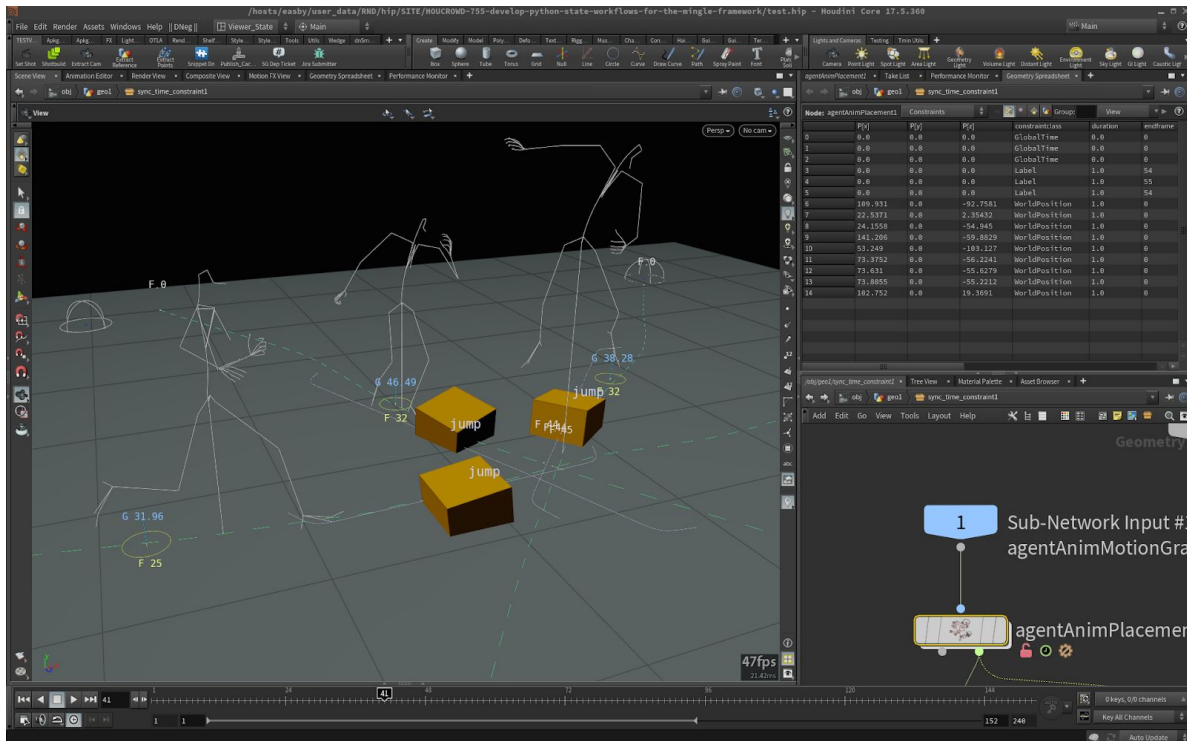


Figure 6.13: Characters jumping over boxes based on an enforced label constraint.

By adding more constraints, the complexity of the interaction can be extended whilst the interactive controls stay manageable. In Figure 6.14 we have one agent waiting on top of a structure, whilst another walks past. At the specified time, the first agent jumps down off the structure and performs a karate chop. The second agent dodges the attack, and runs off scared.

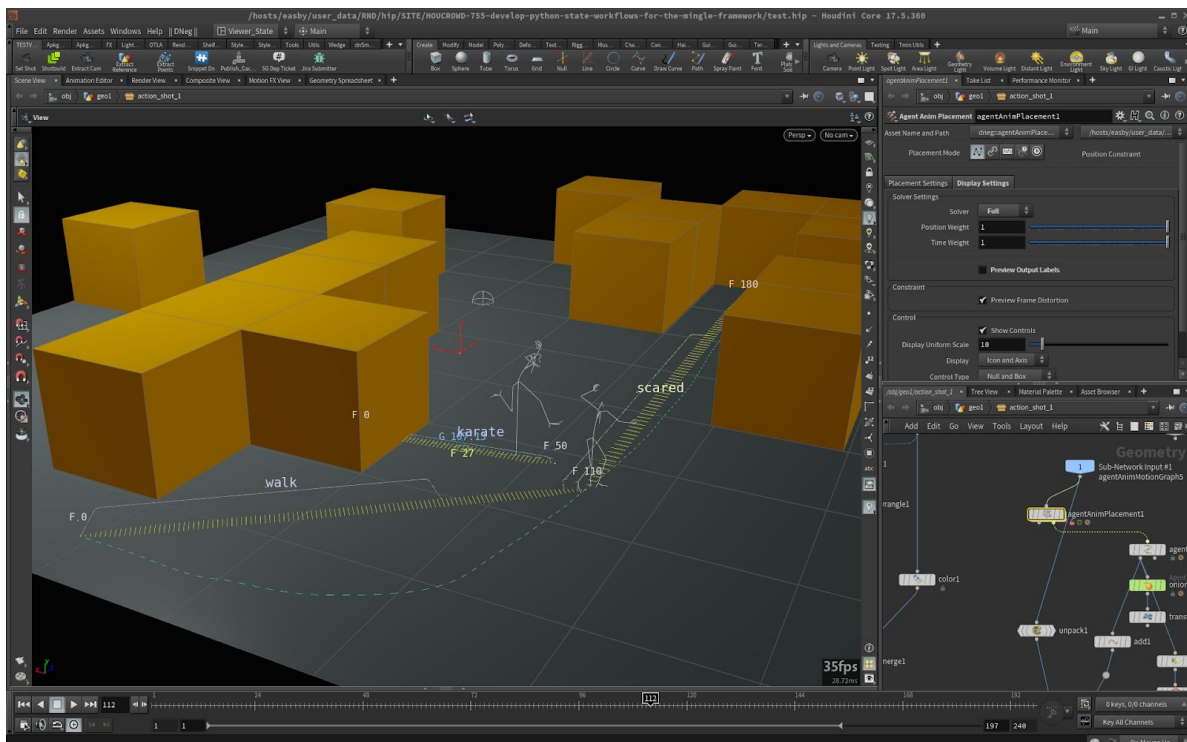


Figure 6.14: A character jumping off a box and karate chopping another who runs off scared.

## 7 Conclusion

Most of the animation synthesis techniques described in this deliverable are areas of research that have been described in academia. However, to our knowledge, this is the first time such tooling has been developed in a production environment for the use in VFX. Whilst still in its early stages, we believe that this technology has the power to transform the way that crowds are produced across the industry.

The goal is to prove that most, if not all, of the benefits of simulation for crowd production can be rivaled with a system that also provides user control and a faster iteration cycle. Whilst there are certainly still some limitations in the tooling at present, we hope that with artist feedback, the toolkit can be enhanced, and make it production ready. In the coming months, we will start testing this tooling on production shots to see if this optimism is founded.

Following on from this initial testing phase, work to evaluate the effectiveness of this new tooling as a replacement for current techniques will be detailed as part of work package 8, specifically WP8T3 *Prototype Evaluation* and results will be detailed in deliverable D8.3.

## 8 References

- [1] Kovar, Lucas, Michael Gleicher, and Frédéric Pighin. 2002. "Motion Graphs." In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '02*, 473–82. doi:10.1145/566570.566605.
- [2] Lee, Kang Hoon, Myung Geol Choi, and Jehee Lee. 2006. "Motion Patches: Building Blocks for Virtual Environments Annotated with Motion Data." In *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06*, 898–906. doi:10.1145/1179352.1141972.
- [4] Sorkine, Olga, and Michael Rabinovich. 2009. "Least-Squares Rigid Motion Using Svd." *Technical Notes*, no. February: 1–6. [http://www.iqf.ethz.ch/projects/ARAP/svd\\_rot.pdf](http://www.iqf.ethz.ch/projects/ARAP/svd_rot.pdf)
- [5] Kim, Manmyung, Kyunglyul Hyun, Jongmin Kim, and Jehee Lee. 2009. "Synchronized Multi-Character Motion Editing." In *ACM Transactions on Graphics*. Vol. 28. doi:10.1145/1531326.1531385.

## 9 Web references

- [3] <https://www.boost.org>

## 10 Acronyms and abbreviations

- **API** - Application Programming Interface
- **CTAM Solver** - Combined Trajectory And Motion solver
- **DCC** - Digital Content Creation
- **HDK** - Houdini Development Kit
- **VFX** - Visual Effects