D5.4 Tools for editing mo-cap data

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

Project funded by H2020 from the European Commission

Project ref. no.	780470
Project acronym	SAUCE
Project full title	**S**mart **A**sset re-**U**se in **C**reative **E**nvironments
Document name	D5.4 Tools for editing mo-cap data
Security (distribution level)	Public
Contractual date of delivery	December 31st 2019
Actual date of delivery	December 20th 2019
Deliverable name	Tools for editing mo-cap data
Type	Demonstration
Status & version	Submission Version
Number of pages	27
WP / Task responsible	DNEG
Other contributors	-
Author(s)	Mungo Pay - DNEG
EC Project Officer	Ms. Adelina Cornelia DINU, Adelina-Cornelia.DINU@ec.europa.eu
Abstract	The use of simulation as a tool to create crowd sequences has been an industry standard for the past two decades, however, there are limitations to this approach when applied to the fast paced nature of a VFX production schedule. This report contains information about the approaches taken to increase reusability of animation assets so as to increase the quality of crowd production shots whilsts reducing the amount of artist time to do so.
Keywords	Animation, mo-cap, trajectory editing, constraint, Laplacian
Sent to peer reviewer	Yes
Peer review completed	Yes
Circulated to partners	No
Read by partners	No
Mgt. Board approval	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 05-12-19	Final version for submission

Table of Contents

1 EXECUTIVE SUMMARY

When authoring large scale crowd animations for feature films, the use of simulation techniques to produce *realistic* results has been an industry standard in visual effects (VFX) for the past two decades. However, whilst being an incredibly powerful methodology, there are drawbacks. Primarily, artists aren't able to make minor edits without having to re-simulate the entire scene, as each frame of the simulation is deterministic and dependant on the previous one. As such, the turnaround of shots can be extremely time consuming as full simulations can take minutes, if not hours, to complete depending on the complexity.

The goal of this work is to speed up the turnaround of crowd shots at DNEG by allowing crowd artists to be able to reuse, edit and repurpose animations used as inputs to the crowd system. The deliverable aims to describe the tooling developed and the desired results of:

- Providing tooling so that allows artists to alter the trajectories of characters in a scene in an *artist directable* manner.
- Allows artists to reuse and repurpose animation so as to reduce the requirement of bespoke animations having to be created.
- Speed up turnaround of shots by creating automated tooling that addresses common use-cases, in particular; collision avoidance.

By altering the approach from an evaluation of agent states on each frame, to evaluating the whole animation of characters within the scene, this deliverable demonstrates the potential of an art driven and simulation free workflow for crowd creation.

We demonstrate a range of path editing tools and evaluate the usability of each within the context of high fidelity digital crowd creation. We also demonstrate an approach to collision avoidance based on this tooling, as well as an animation layering framework.

Whilst a full evaluation of how the tools perform in a production environment has not yet been carried out, we conclude that the tooling shows a lot of promise and that they will be beneficial to our artists. We also provide the groundwork for extension into motion synthesis techniques which will be described in deliverable D5.5.

2 BACKGROUND

Work package 5 focuses on asset transformation, and this document describes the work done in relation to WP5T2. It also acts as a companion piece for some of the work outlined in WP5T3 which is described in deliverable D5.5. In that deliverable, parts of the trajectory editing toolkit is used as an input to the animation synthesis techniques.

The three tools that were specified to be included in this delivery document were as follows:

- "A tool that allows to alter character's path, based on user input or environment description."
- "A tool that allows to change style of an animation, or to match style of two animations to allow them to be seamlessly combined."
- "A tool that allows to solve for collisions by applying minimal adjustments to animations already placed in the scene."

The work done to alter a character's path can be seen more as a framework that can be extended or augmented to solve both the first and third bullet points in that list. The tool described in the second bullet point was adapted slightly, with the approval of the consortium, to focus more on animation layering. This change was made for a number of reasons, though primarily because the tooling was both more beneficial to the company, and more fully addressed the goal of asset re-use which is a key aim of the SAUCE project.

3 INTRODUCTION

To address the requirements described in WP5T2, this deliverable focuses on tooling to edit mo-cap data in an artist friendly manner. The tooling presented allows artists to make minor edits to a character's path, layer together multiple animations onto a single character to produce new performance data, and generate edits to the path of characters to remove collisions.

Chapter 4 focuses on the background of path manipulation, describing desired characteristics of such tooling.

Chapter 5 discusses different implementations, how they address the desired characteristics and what limitations they possess.

Chapter 6 outlines extensions to trajectory editing to handle collision avoidance.

Chapter 7 describes the work done for the animation layering toolkit, which allows artists to apply sections of animation from multiple clips to produce new performances.

Chapters 8, 9 & 10 cover conclusions, references and web references.

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.
- To provide tooling that automates tasks that occur frequently.

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

3.2 Methodology

As there are a few tools associated with this deliverable, and in particular the trajectory editing toolkit covers a few different implementations, the methodology for each task is quite varied. That said, a similar strategy was adopted where practical, namely the development of a C++ backend and a DCC specific front-end. Implementing the toolsets in this way ensures that we not only have a large degree of control when it came to optimisations, but also allows us to be DCC agnostic in terms of the core functionality. This is important when developing tooling for VFX as studios are not fully in control of the direction of development for third party software. Whilst we maintain communication channels with the producers of the third party software used at the studio, priorities can change, so we need to be as flexible as possible to those changes.

For the purposes of this project, testing of initial concepts was done using a simple standalone in house OpenGL viewer. For artist interaction, the user interfaces for tooling was developed in SideFx Houdini™ as this is currently the software used at DNEG for crowd creation. More detail about the integration will be described in deliverable D5.5.

3.3 Terminology

Agent: A single animated character in a crowd system.

Character: Used interchangeably with **Agent**.

Constraint: A condition of a solve that determines that output animation.

OpenGL: An industry standard open source graphics drawing library.

Laplacian Solver: A system that ingests input animations and constraints and produces new or altered animations that satisfy those constraints.

LU Solver: Solver for sparse matrix of linear equations taking the form Ax = b by way of LU decomposition [5].

QR Solver: As with the sparse LU solver, solves a system of linear equations but using QR decomposition [5].

3.4 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 package 8 in deliverable D8.3.

4 Requirements of path editing tools

It would perhaps seem reasonable to assume that path editing tools might have the same set of requirements as curve editing tools since the trajectory of a character can be simply represented as a curve. However, there are a number of restrictions that must be satisfied when dealing with animations as we need to ensure that the characteristics of the original animation are maintained.

The goal of the path editing tools are to allow artists to quickly edit the trajectory of locomoting characters in an intuitive manner and art-directable. This chapter covers some of the requirements when designing path editing tools.

4.1 Speed of interaction

As one of the primary goals of this work is to ensure that the turnaround of shots is sped up, speed of interaction is a lynchpin for any path editing tooling. For simple path editing of a single character, anything short of a fully interactive response to path edits will dramatically impact the artists user experience with the tool.

4.2 Animation driven positioning

Simply editing the trajectory of character animation by moving keyframes in space would have unintended side effects on the motion. If, for example, the trajectory was extended with no additional processing, then the distance travelled by the agent would increase whilst the animation speed would remain constant. This would introduce foot sliding which would make the animation appear unnatural.

Figure 4.1 illustrates how this animation stretching would result in accelerated movement over the same timeframe where the dots depict keyframes of the animation.

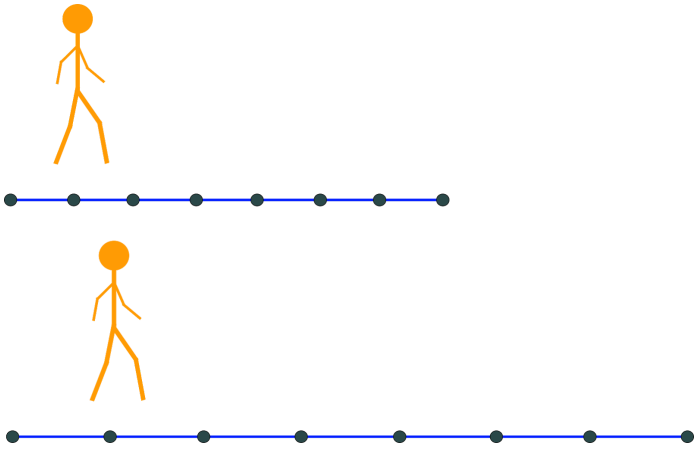


Figure 4.1: Animation stretching leading to foot-sliding

It is therefore important to apply some form of resampling to ensure that this footsliding is minimised.

It should be noted however, that without some mechanism for procedurally adapting the character's motion based on the trajectory using, for example, a motion graph implementation, there will always be some degree of foot sliding introduced by altering an agent's path. This is especially true when introducing turns, as the bipedal motion for walking round a corner is quite different to that of the motion when walking forward.

That said, any trajectory editing tool that simply remaps animation should at least preserve the distance travelled.

4.3 Positive and negative constraints

Whilst there are many use cases where an artist may wish to specify that a character is at a certain location at a specified time, there are also cases where they may which to specify that a character avoids a specific location. For the purposes of this document, we will refer to these use cases as positive and negative constraints respectively.

4.3.1 Positive constraints

Figure 4.2 below illustrates how the placement of positive spatio-temporal constraint might look. The blue crosses denote positive constraints placed in space, associated with a given frame. From this, a trajectory can be formed.

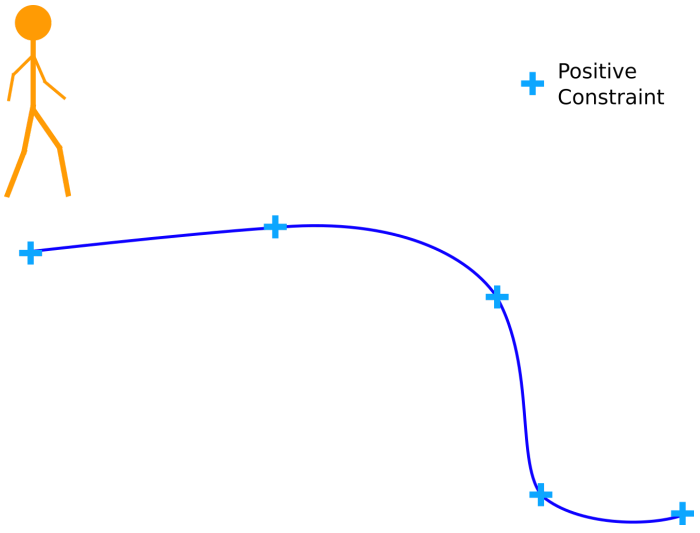


Figure 4.2: Path following using positive constrates

The properties of the trajectory, and how the constraints should connected, is not covered here, but will be detailed further in section 4.5.

4.3.2 Negative constraints

Negative constraints might be used when there are certain areas that agents need to avoid. One example would be if there were obstacles in a scene and an artist wanted to specify that no agent's trajectory should pass through a certain area. Another example is that of collision avoidance with

other characters in the scene, where we might want to iteratively push agents away from each other when a collision has occurred.

Figure 4.3 below illustrates how negative constraints could be used to act as an area of avoidance.

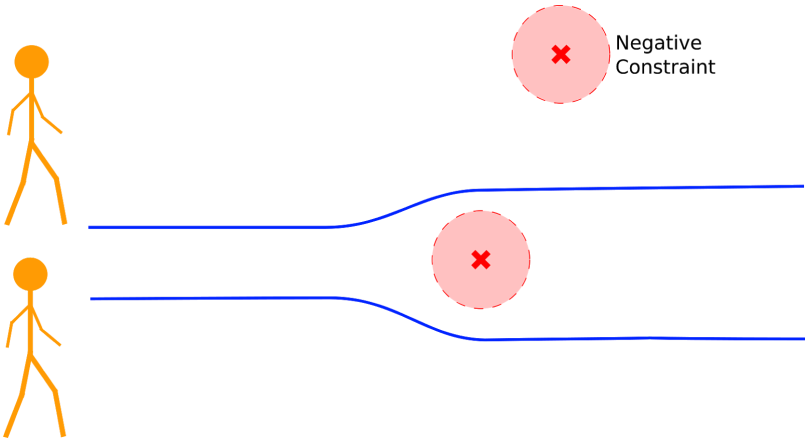


Figure 4.3: Path following using negative constraints

4.4 Multi-character constraints

When dealing with the interaction between multiple characters, it is often useful to be able to specify constraints that affect more than one agent. This is useful in situations where characters need to interact with each other. If, for example, an artist had animations of two characters walking towards each other and then fighting, they would need to to be able to specify a distance and frame time as any interaction like this is both spatially and temporally linked.

For a tool that allows this kind of linking of animation to occur, the resulting animation would need to be robust to other edits that might occur later. Without this robustness, each edit could affect previously defined interactions, and editing would become incredibly laborious.

4.5 Shape preservation with second order derivative smoothing

If a constraint is applied to an animation in such a way that it moves the position of the agent at a given time, it is necessary to ensure that the effects of that constraint are distributed as evenly as possible over the course of the animation. The figure below demonstrates how a constraint applied to a section of animation at time Pi would deform the animation trajectory if no smoothing were to be applied.

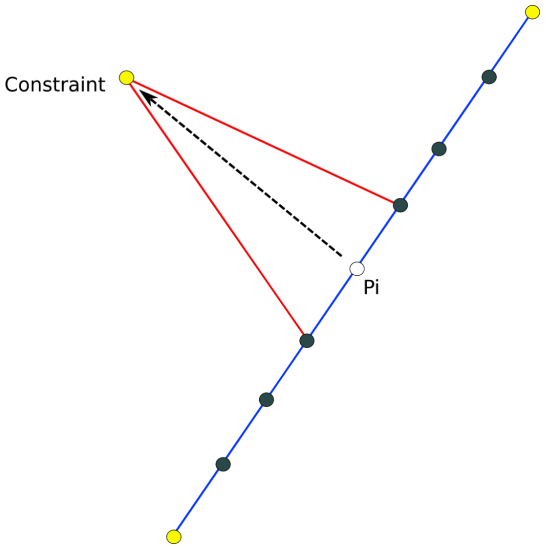


Figure 4.4: Animation constraint with no shape preservation

To ameliorate this kind of discontinuity, it would be desirable to have some kind of second order smoothing algorithm to spread the effect of such a perturbation over the duration of the animation clip, or at least between any other constraints. The results of such a smoothing function are shown below.

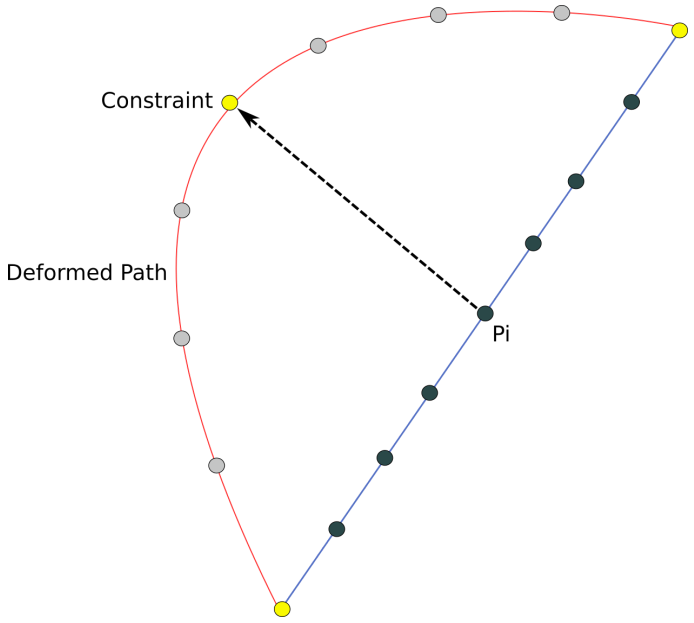


Figure 4.5: Animation constraint with shape preservation applied

In this example, the first and last points of the animation are constrained in place, and the point Pi is constrained and moved. The remaining points are also transformed in order to preserve the shape of the path, attempting to minimize changes in trajectory and the distances between each point. By ensuring that this deformation is as smooth as possible, it is possible to minimize obvious foot sliding and abrupt rotation in the resulting animation.

This simple transformation above can be achieved with a curve-editing tool, but for any more detailed shapes, this would require many manual edits and rely on the user to preserve the shape by eye.

5 Implementations of path editing

In this chapter, we describe some of the path editing tooling that has been developed to solve some of the requirements laid out in chapter 4.

5.1 Spline manipulation

In our first implementation, we attempted to make a simple spline manipulation tool that remaps the trajectory of the character by translating its root joint based on a control spline, whilst maintaining the distance travelled by the character so as to minimise any foot sliding artifacts introduced by stretching the animation.

The viewport controlled spline editing workflow offers a standard spline editing toolkit with controls in the viewport. This allows an artist to edit the trajectory in real-time using the agent's full animation trajectory curve. One downside of this technique, as described in section 4.2, is that it can introduce some foot sliding when a character turns abruptly. This becomes particularly obvious for animations where the character is walking in a straight line but the input curve contains a sharp turn. This is a factor that artists must consider, but for quick edits, especially when the character is far from the camera, this simple toolkit can dramatically improve the turnaround time of certain shots.

Figure 5.1 shows this tool implemented in SideFX Houdini™ with the trajectory of a number of agents being altered using the white spline controls. Each of these animations have been altered from a single straight walk animation.

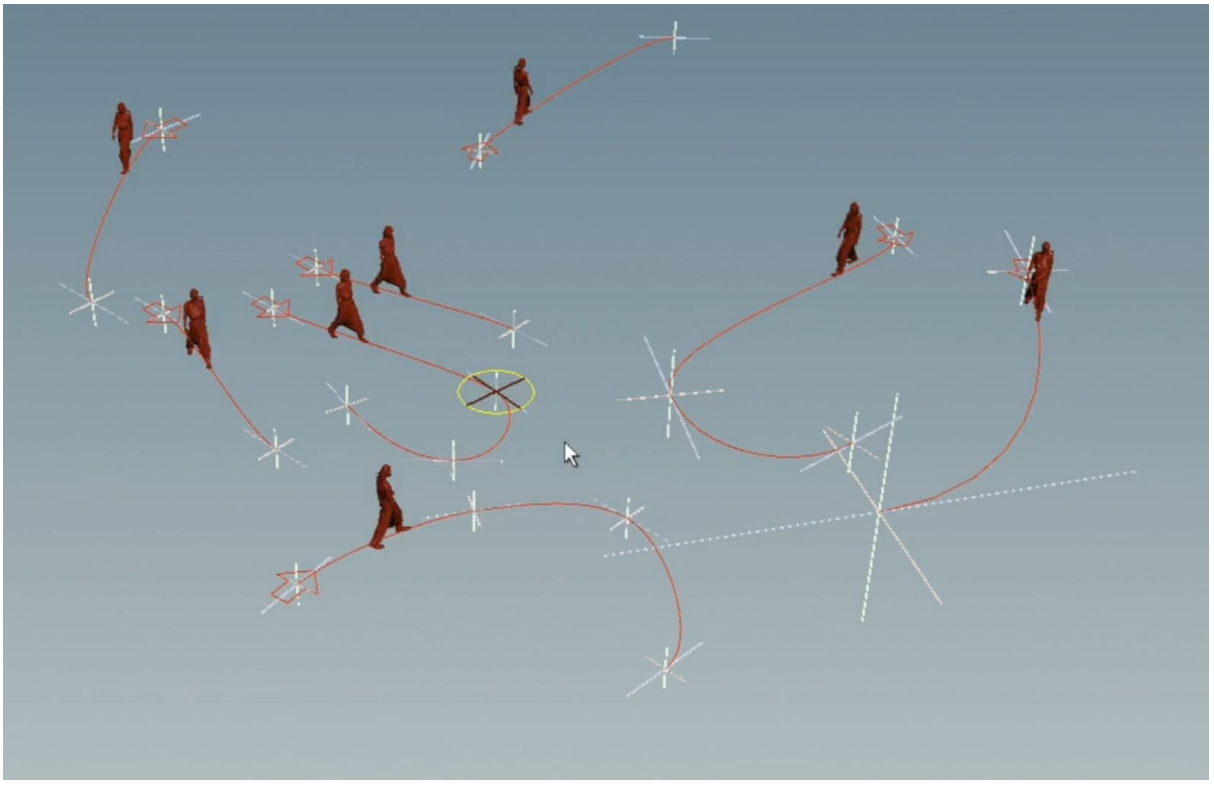


Figure 5.1: Path editing using spline manipulation

5.2 Implementing shape preservation algorithms

Whilst the spline manipulation tool was powerful, it is very easy to introduce sharp turns and therefore, obvious foot sliding. To address this, we decided to investigate algorithms that better handle the foot sliding by preserving the shape of the trajectory. We prototyped a couple of solutions, one iterative solver, and one laplacian solver and evaluated them in the context of trajectory editing.

5.2.1 **Local Features**

In order to preserve the shape and scale of a deformed path we form error functions that minimize changes in the local features of the path. The 2D path formed by the motion path can be represented as a triangular mesh where each position forms a triangle with its immediate neighbour, as depicted in figure 5.2.

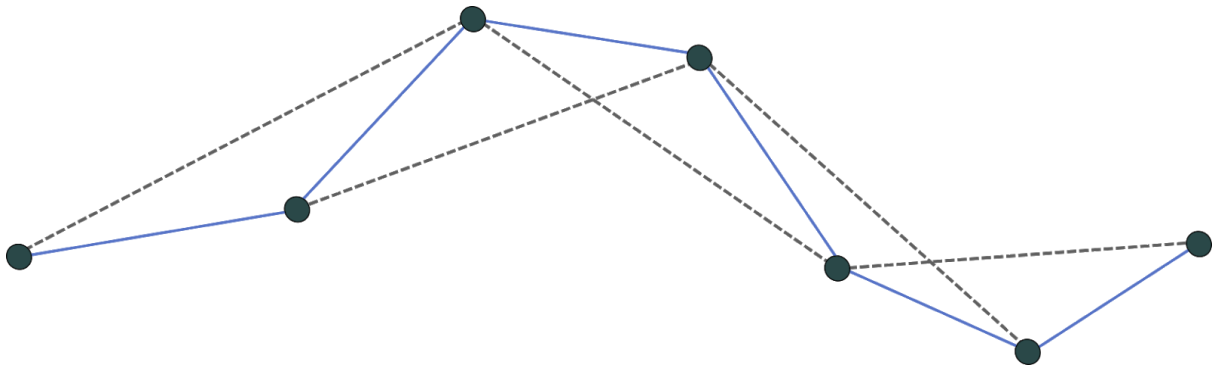


Figure 5.2: Shape preservation using triangulation error metric

Any point P_i on the path may be represented with as its local coordinates, x and y, within its triangle.

$$P_i = P_{i-1} + x(P_{i+1} - P_{i-1}) + yR_{90}(P_{i+1} - P_{i-1}) \quad [1]$$

where R_{90} is a rotation matrix $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$

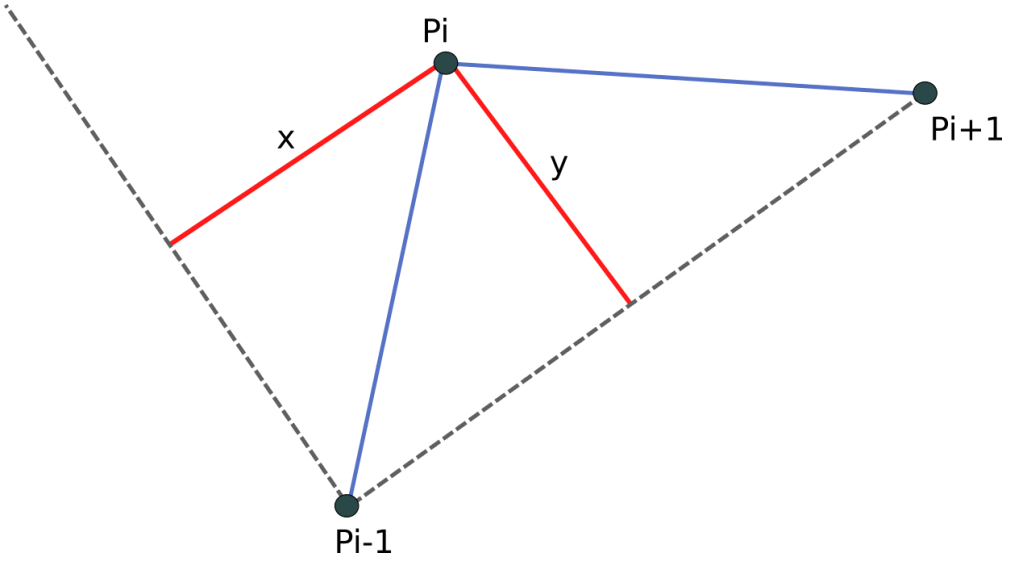


Figure 5.3: Local co-ordinates for the constructed mesh

The desired solution for the deformation would move any points to their constrained positions while introducing no change in the local x and y coordinates. However, for any deformation that involves

scaling the mesh it is very unlikely there will be a solution that achieves this, so we find an approximate least squares solution, which attempts to minimize changes in the local features while satisfying constraints —

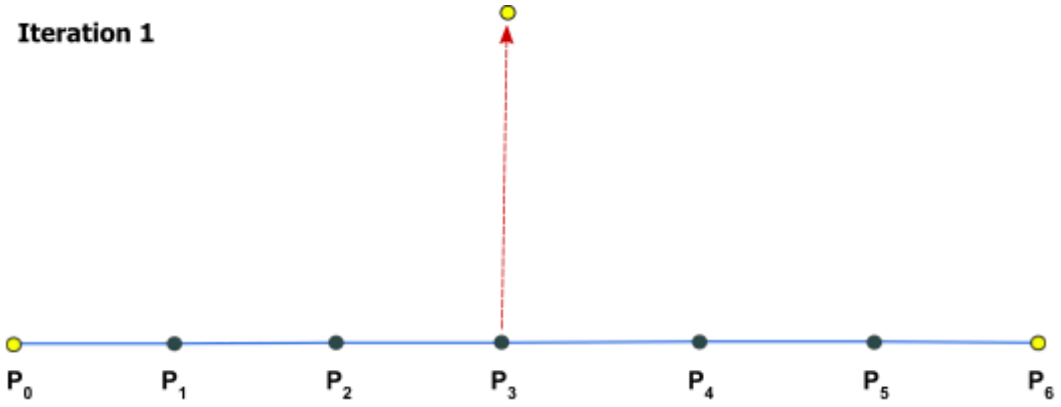
$$E(P) = \sum_{0->n} \| p_i^{rest} - p_i^{deformed} \|^2 + w \sum_{i \varepsilon C} \| p_i^{deformed} - c_i \|^2 \quad [1]$$

where *P* is the path, *n* is the number of points, *c* are constraints and *w* is the constraint weight.

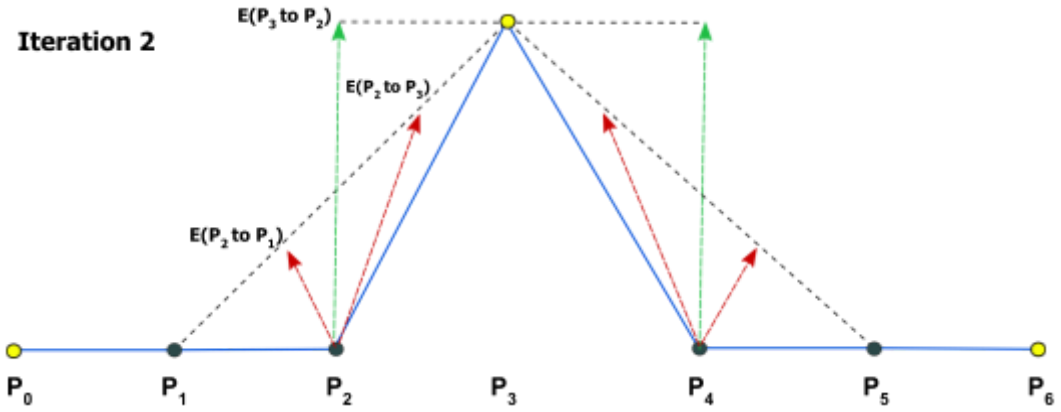
5.2.2 **Iterative Error Minimization**

Our first attempt at a mechanism for solving this was to implement an iterative deformer. For each iteration, every point on the path accumulates one error for its local position compared to its rest local position, and another error for any constraints applied to that point. The point is then moved to minimize that error. After enough iterations, the path is smoothly transformed to satisfy the constraints.

Below we show how errors are calculated and applied through the initial iterations of a solve. An error for a position is represented as a translation vector which is the amount the position should be moved to minimize the error. A position has separate errors to its left and right neighbours, as well as errors from its neighbours to itself.

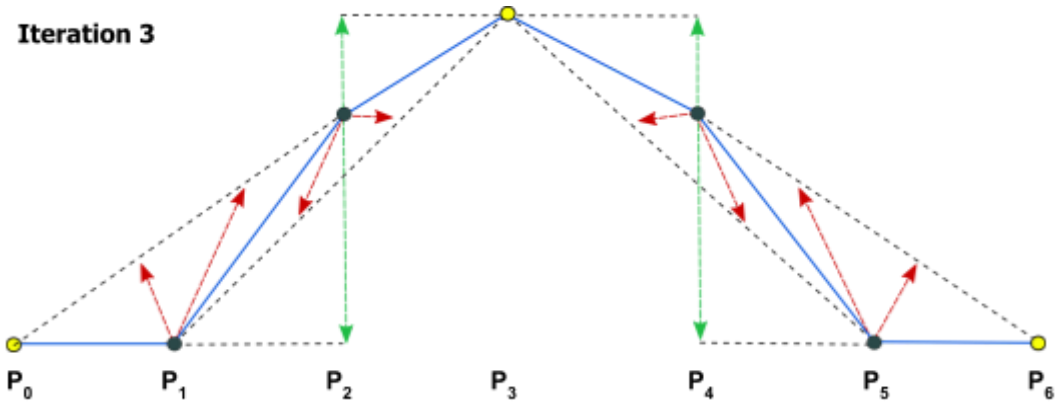


P_3 has an error to a position constraint so is moved by that error vector.



After P_3 is moved, its neighbouring points now have error for their positions local to their neighbours. Taking P_2 as an example, the red arrows shows the error of its local position in the coordinate system P_1->P_3. The green arrow shows the error applied to P_2 from P_3.

The errors for any point are summed and then averaged for the number of neighbours which have an influence on the point. In the case of P_2 we divide the summed error by 4, as we are minimizing error to and from its adjacent neighbours.



A method for reducing the number of iterations required is to consider error beyond just the adjacent neighbours. This requires applying a weight average of error. The weight of the error can be calculated from the proximity of a point to its neighbour.

This solver had the advantage of being easier to understand than other methods and quick to implement as a prototype, but was ultimately too slow for interactive editing.

5.2.3 **Laplacian Shape Editing**

In an attempt to speed up the solve, another solution was explored. As-Rigid-As-Possible 2D mesh editing [1] makes use of the Laplacian coordinates of a mesh to form two least-squares solutions of linear equations that can be solved very efficiently.

Instead of directly editing the vertices of the mesh, the differential coordinates of the vertices can be used, with the constraints providing the *known* vertices on the mesh [3].

The goal of each edit is to translate and rotate local coordinates to preserve the shape/trajectory of the path, as shown in the shape preserving least-squares solution, and also to scale to reduce the amount of squash and stretch.

Unfortunately, it is not possible to form a single error function that will both preserve the translation and rotation, as well as the scale [1]. With As-Rigid-As-Possible editing, the problem is split into two least squares solutions for shape and scale preservation. The result of the shape preserving step can then be passed to the scale preserving step.

Step 1: Shape Preservation

$$E(P) = \sum_{0->n} \| p_i^{rest} - p_i^{deformed} \|^2 \qquad [1]$$

Step 2: Scale Preservation

Using the deformed positions from step 1, for any point $p_i^{deformed}$ the normalised vector to the next point $p_{i+1}^{deformed}$ is found and scaled by the length of the respective rest vector to form $v_i^{deformed}$. The purpose of this was to minimize any change in these vectors so the following least squares error function could be formed —

$$E(P) = \sum_{0->n} \| (v_i^{rest} - v_{i-1}^{rest}) - (v_i^{deformed} - v_{i-1}^{deformed}) \|^2 \qquad [4]$$

Breaking these error functions down into their partial derivatives allows for the formation of a system of linear equations that was efficiently solved using a sparse matrix LU solver [5].

By using the Laplacian shape editing solver, the speed of interaction was dramatically improved over the iterative solver, meaning that individual edits could be made in real time. This addresses the initial requirement in section 4.1 stating that the editing would need to be interactive for any trajectory editing tools to be usable.

5.3 Group motion editing

Building on top of the laplacian shape editing mechanism described in section 5.2, we wanted to give users more control when editing crowd scenes involving a large group of characters moving together. This situation arises often when producing shots, with examples including scenes where soldiers are in formation, or pedestrians are constrained to walking on a footpath in close proximity to one another. Manually editing the trajectory of individual agents can be tedious, especially when there are any more than just a few agents to deal with.

Group motion editing [2] allows users to edit all the paths of a group at the same time, maintaining the spacing between characters while preserving the local features of their paths which reduces foot sliding and large rotations.

5.3.1 **User Interaction**

After laying out a group of characters each with animation, a user can click and drag from any point on the path of a character and move it, creating a position constraint for that point. This will simultaneously edit each path of all characters, making it very quick to edit to follow more complex trajectories. An example of a number of characters being edited as a group is shown in figure 5.4

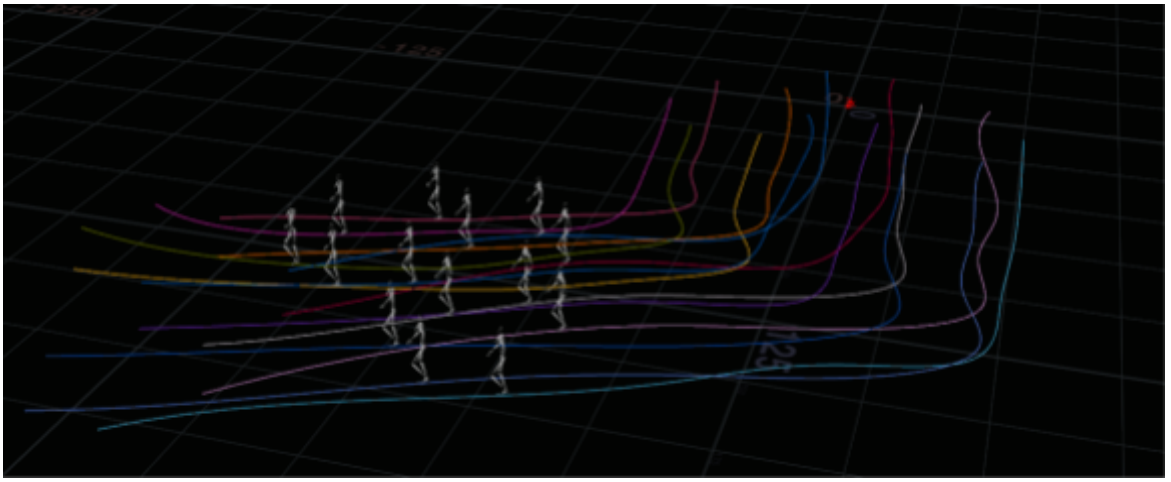


Figure 5.4: Path editing with formation preservation using group motion editing

Here the group of 14 agents were all walking in a straight line, but a single rotational edit changes all of their trajectories whilst maintaining their proximity to one another.

5.3.2 **Implementation**

To achieve these results, we use the Laplacian solver described in section 5.2 with a small extension. This solver forms an error function that attempts to minimize changes in the local coordinates of each point on a path while adhering to positional constraints set by the user. While the deformation described in the previous section was applied on a mesh created from the motion path of an

animation, group motion editing connects the individual path meshes of each character, adding edges between neighbouring points and applying deformation to the entire mesh at once.

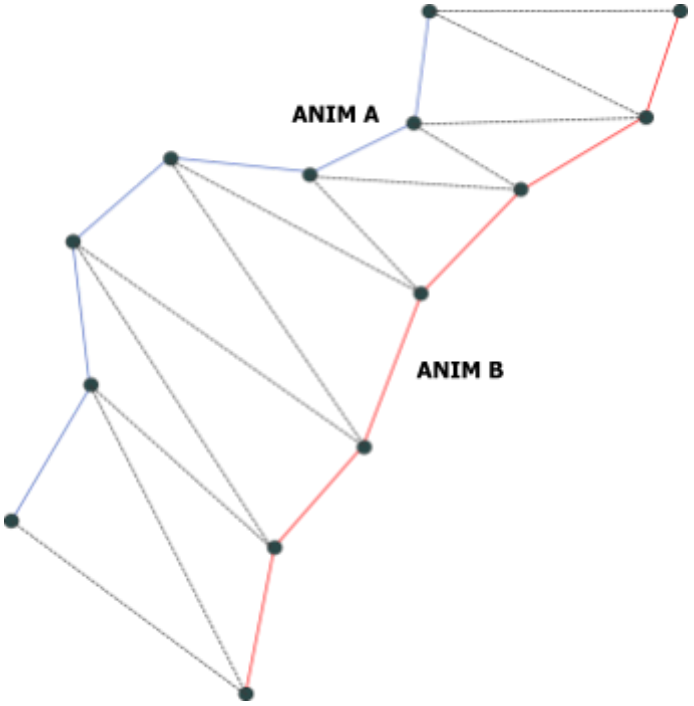


Figure 5.5: Mesh construction for As-Rigid-As-Possible deformation for group motion editing

5.3.3 **Further Work**

Although the shape deformation solver attempts to preserve the spacing between, the minimization of error over the entire mesh can still lead to overlapping trajectories, in which case, manual edits or some form of collision avoidance algorithm would need to be applied. As part of this work packet, we have implemented an iterative collision avoidance algorithm which will be detailed in chapter 6.

Any deformation of a motion path may stretch or squash the distance between points, resulting in foot sliding and unrealistic speed of movement. This is partially addressed in Group Motion Editing [2] by time warping deformed animations to match their original speeds, although the foot sliding will still be present. Instead, the group motion editing could be integrated with the motion graph implementation developed for deliverable D5.5 which, based on the desired position constraints, creates new animation data that may have more or fewer frames of animation depending on any edits made. However, this is future work and not in the scope of this deliverable.

5.4 Constraint based path editing tool

The group motion editing implementation represents a powerful step forward towards a simulation free crowd editing toolkit. That said, it doesn't provide us with all the tools required to move away from simulation completely. This is because it is designed mostly as a deformation framework that happens to use crowd characters as its "mesh".

For this reason, we have developed another tool that allows artists to define constraints, and solve for appropriate animation. Part of the reason for this is that we want to be able to define character interaction. This will be covered in more detail in section 5.4.2.2 with our introduction to relative position constraints.

For simple trajectory editing however, we can introduce spatio-temporal constraints that specify that an agent must be at a particular point in space at a given time. The deformation must satisfy the constraint of following the path but also minimize the amount of deformation of the original captured data in order to maintain a realistic performance.

Our editing tool provides the user with the means to easily make large edits, while preserving the details of the source data at interactive, real time performance.

5.4.1 **Animation Processing**

The spatial animation data being edited is the trajectory/path the character will follow. To have a starting point for deformation, this trajectory must first be extracted from the animation. This is done by simply using the position of the root joint at each frame. For animations without a root joint, this joint is added by projecting onto the ground directly below the hip joint. The rotation of the joint is calculated from the tangent of the path, i.e, the normalized vector between the previous frame's root position, and the next frame [4].

For stationary animations where the tangent of the path may have any arbitrary direction, the rotation is set as a constant. Stationary sections of animation are described by the user using a tagging system in which a given frame range is tagged as "Stationary". This tag is also used later during shape deformation as a hard constraint to maintain the shape of stationary animation, as even slight deformation is very obvious and damaging to fidelity.

Methods exist to automatically detect stationary animation but to handle a large number of animations accurately is beyond the scope of this project. Although requiring manual user input, a tagging system is a robust and re-usable solution.

5.4.2 **Constraints**

Constraints act as a user-interface for editing animation data. We provide constraints for editing the motion path, time-mapping and interaction between characters [4]. Each constraint is placed upon frames of animation, and where appropriate may have a duration of frames to which they are applied.

5.4.2.1 World Position

At frame f, places the root position of the animation at position p. Used as a handle for editing the motion path of an animation.

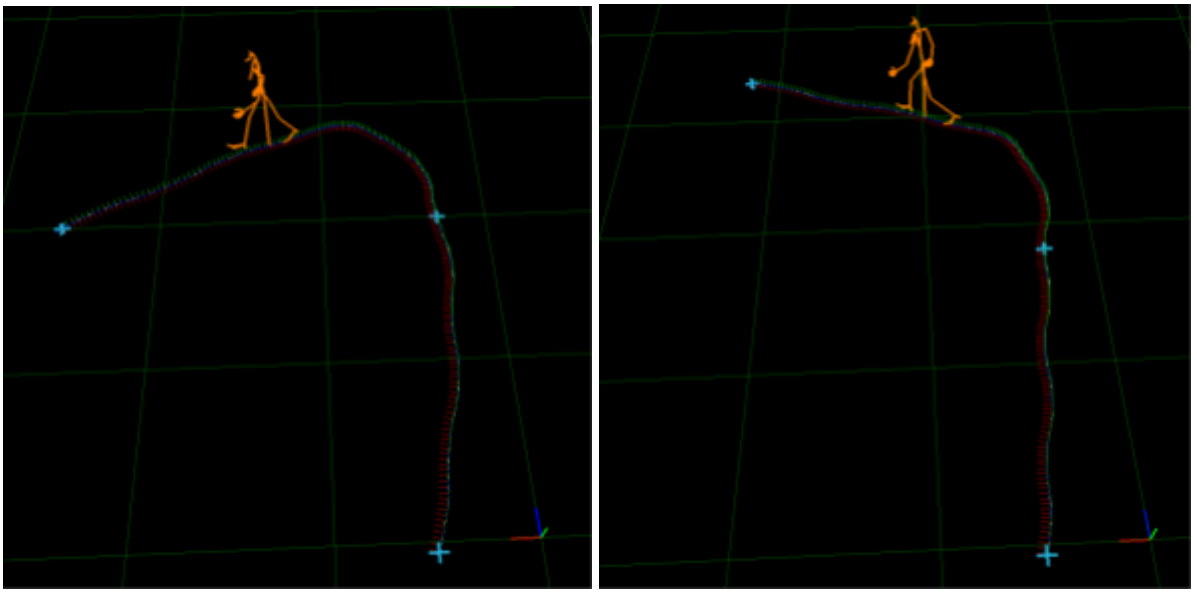


Figure 5.6: Example of editing World Position Constraints.

5.4.2.2 Relative Position

A relative position constraint constrains the root position of animation A at frame f1, to the root position of animation B at frame f2. The direction and distance between the frames is set with a user-defined constant vector. This may be used for character interaction e.g. two characters meeting, walking alongside each other or fighting. The constraint is easily set and maintained regardless of changes in other constraints.

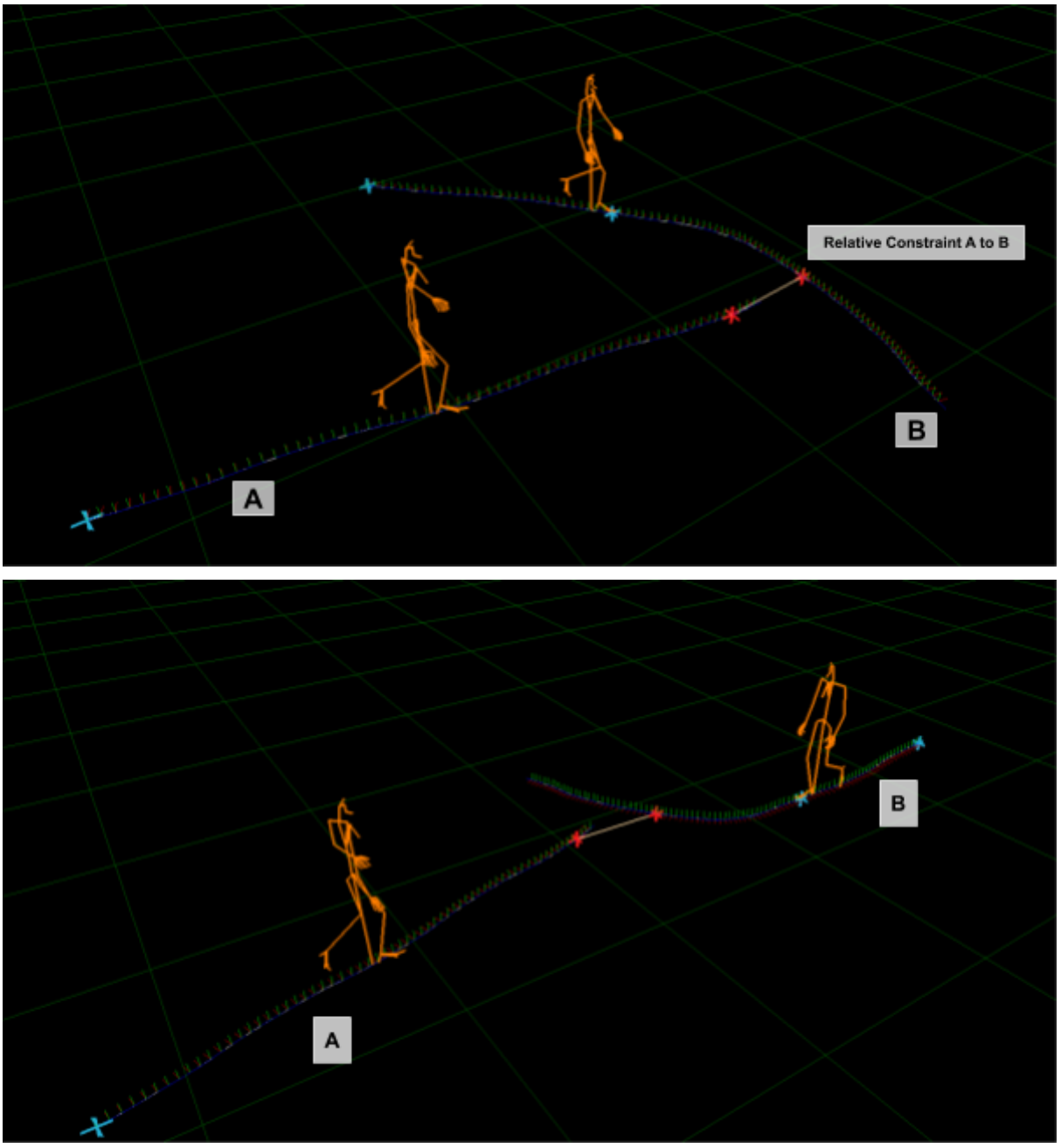


Figure 5.7: The relative constraint from A to B is maintained while Anim B is edited.

5.4.2.3 World Time

This constrains frame f to occur at time t, smoothly time warping the animation.

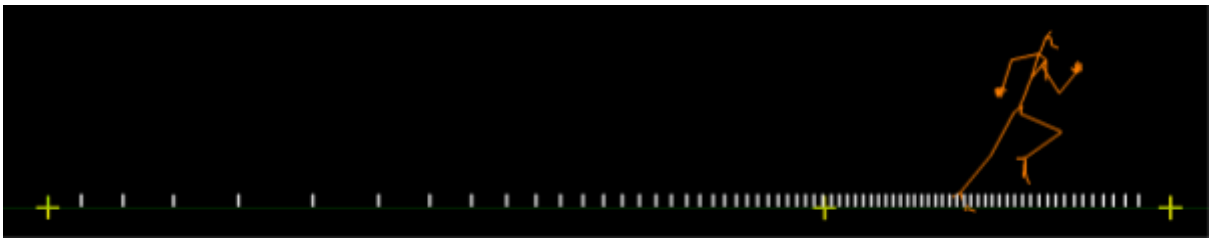


Figure 5.8: Root positions drawn at constant time intervals demonstrate the smooth time warping.

5.4.2.4 Synchronised Time

Similar to the Relative Position constraint, constrains the time of frame f1 of anim A to frame f2 of anim B, with a constant time difference of t. Can be used in various situations of character interactions, shaking hands, clashing swords, or reacting to an in-scene event can all be precisely timed and resolved regardless of edits to the timing of the animations prior to the constraint.

5.4.2.5 Stationary Animations

Any deformation of a stationary section of an animation will result in very obvious sliding or rotation. Therefore, for any frames that have the user-defined "Stationary" tag, we add the shape preserving function as hard constraints across both steps, which will prevent any deformation.

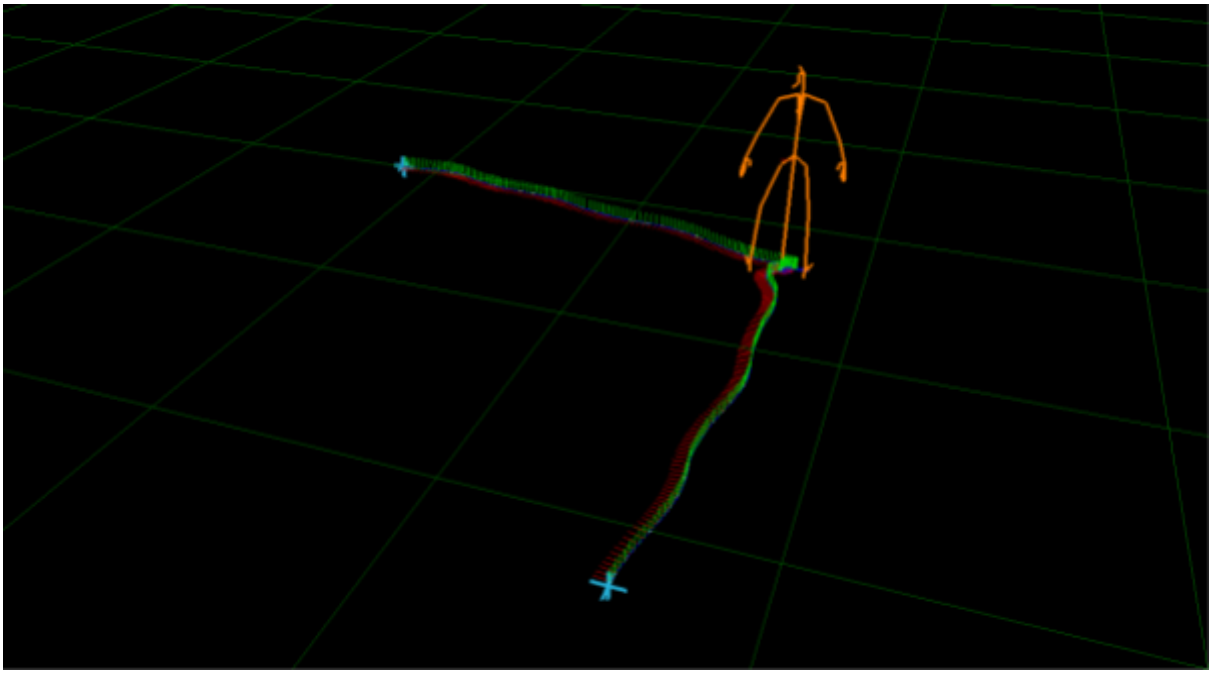


Figure 5.9 Locomoting animation with a stationary section in the middle.

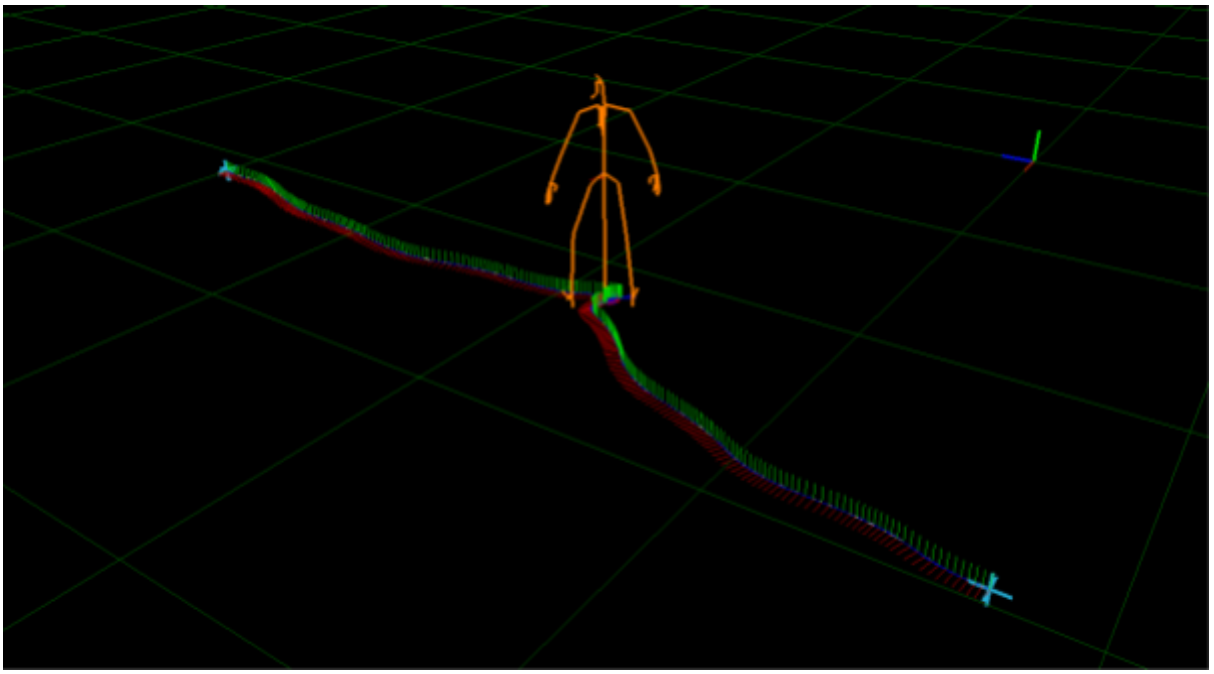


Figure 5.10: World position constraints are edited but no deformation occurs at the stationary section of the animation

5.4.2.6 Laplacian Time Warping

Time mapping data describes what time each frame of an animation occurs at which can be warped to meet the timing requirements the user has set using constraints.

The undeformed animations have a constant time interval between frames. In order to smoothly time warp an animation we form an error function that will try to minimize changes in the interval between frames. Unlike the spatial deformation, we only need to form a single error function [4], which in our system is the same as the scale preserving function of the spatial deformation —

$$E(T) = \sum_{0->n} \| (interval_i^{rest} - interval_{i-1}^{rest}) - (interval_i^{deformed} - interval_{i-1}^{deformed}) \|^2$$

where *T* is the time map data of the animation, and interval is the difference between the time at frame *i* and *i+1*.

5.4.3 **Linear System Solver**

Systems of linear equations can be represented as matrices, for example —

$$a_1 x + b_1 y + c_1 = 0$$
$$a_2 x + b_2 y + c_2 = 0$$

a, b and c are known constants, and we are solving to find x and y. This can be translated into matrix form as follows —

$$\begin{bmatrix} a1 & b1 \\ a2 & b2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} c1 \\ c2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

In order to solve the least squares problems we need to put the data into a form that can be solved by a LU solver —

$$Ax = b$$

where A is a sparse matrix of r rows and c columns. It stores the partial derivatives of our least squares solutions.

x is a vector with r elements of unknown values which is what we are solving for. The result of x will store the positions the deformed animation paths, or the time values for the animation time maps.

b is a vector of known values, i.e constraints and desired differences between partial derivatives of the points/times.

We use the Eigen library [5] which provides a number of solvers for sparse linear systems, of which we found the umfpack [6] LU solver to be the fastest of those tested. One issue that could arise with a LU solver was in cases where matrix A was a singular (non-invertible) matrix. A sparse LU solver requires a matrix that can be inverted, so in cases where it is not we fall back to using one of the more stable but slower options such as a sparse QR solver [5].

6 Collision avoidance via path editing

The constraint based path editing tool described in section 5.4 provides a framework on top of which a collision avoidance scheme can be built. This chapter details some of the approaches explored.

All of these approaches involve changing the trajectory of an agent when a collision is detected, pushing the agents away from the point of collision. Any change in trajectory however may push an agent into the path of another agent, causing a new collision. We therefore iteratively apply these adjustments — checking for collisions, pushing trajectories and repeating until no more collisions have been found or a maximum number of iterations has been reached [2].

Collisions are found by sampling the positions of agents at constant time intervals, and for each pair of agents find the maximal point of overlap between the collider capsules placed on the agent. For each maximal collision we then use the Laplacian path editing tool detailed in section 5.2 to move the colliding points of the two trajectories apart in the direction of the vector between the two points. We do this by creating a positional constraint between the points and solving for the paths of the agents, subject to the user-defined constraints. The formation of these collision constraints underwent a few iterations before finding a robust solution.

6.1 Relative Position Constraint

We initially used Relative Position constraints, seen in section 5.4.2.2, which enforce a world direction and distance between two points. However, this only yielded reasonable results in very simple scenarios. Constraining the world direction between two points could lead to unnatural trajectories with a large amount of squash or stretch, particularly as the constraints accumulate with each iteration and therefore may conflict with one another.

6.2 Mesh-based Constraints

The next iteration represented the collision constraints as an edge in the colliding path meshes, maintaining the distance between collision points in the same way local features of the path are handled. Representing the constraint using local coordinates generally gave smoother results, and could handle straight-forward scenarios.

Figure 6.1 demonstrates the result of such a collision avoidance scheme.

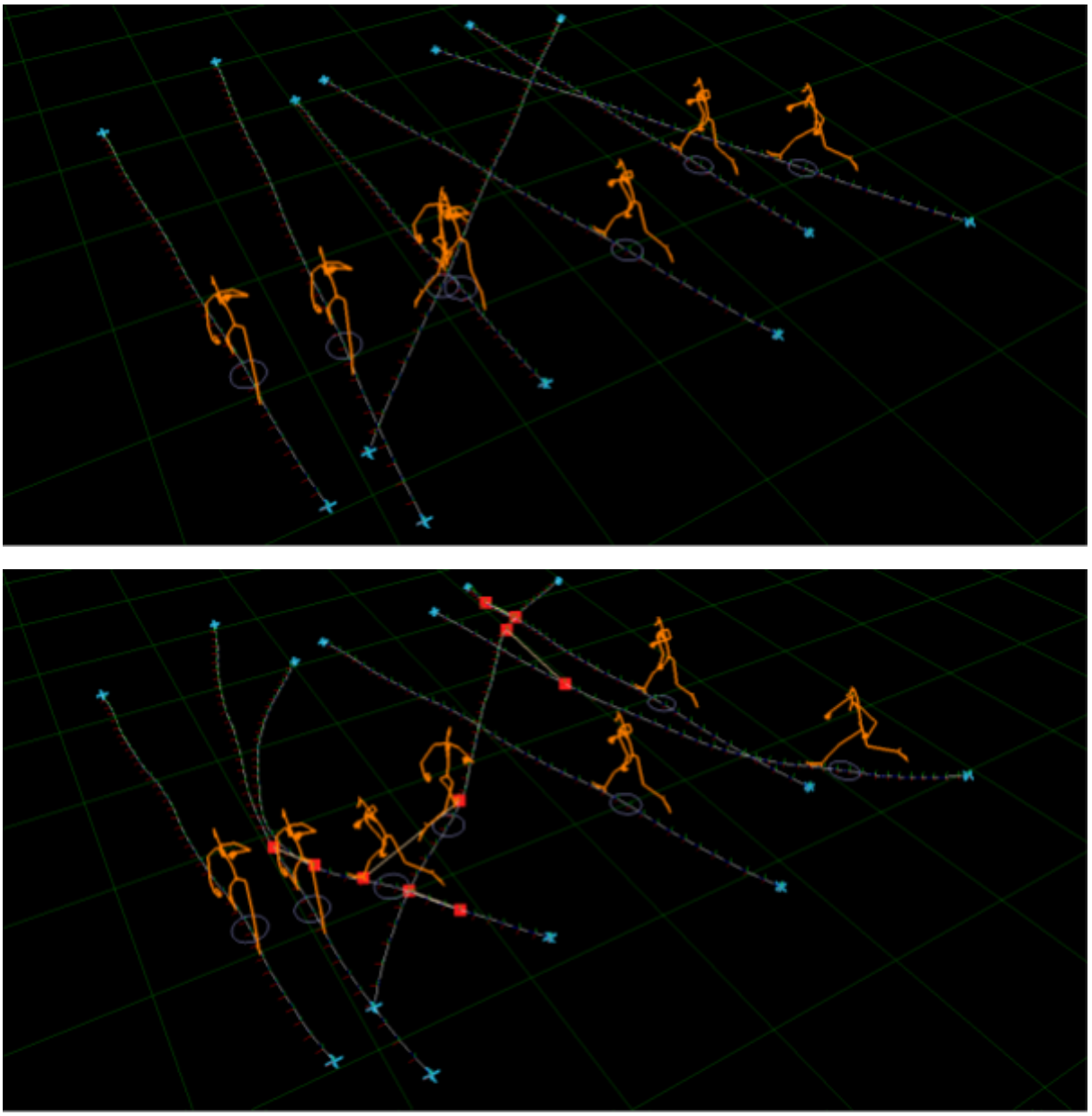


Figure 6.1 Shape and scale preserving constraints are added between colliding points.

However, as with the relative position constraints, situations that required more iterations to solve, i.e, densely populated areas of agents with numerous collisions, led to an accumulation of conflicting constraints which resulted in a nest of colliding paths that didn't converge on a solution.

6.3 Solution

The previous attempts both involved adding new constraints to the original source animation each iteration and resolving, which led to over-constrained paths bunching together. The solution we found to this problem was to use the resulting animations of the previous iteration as new source animation of the solve. We then only apply the mesh-based constraints of the previous sub-chapter for the maximal collisions at each iteration.

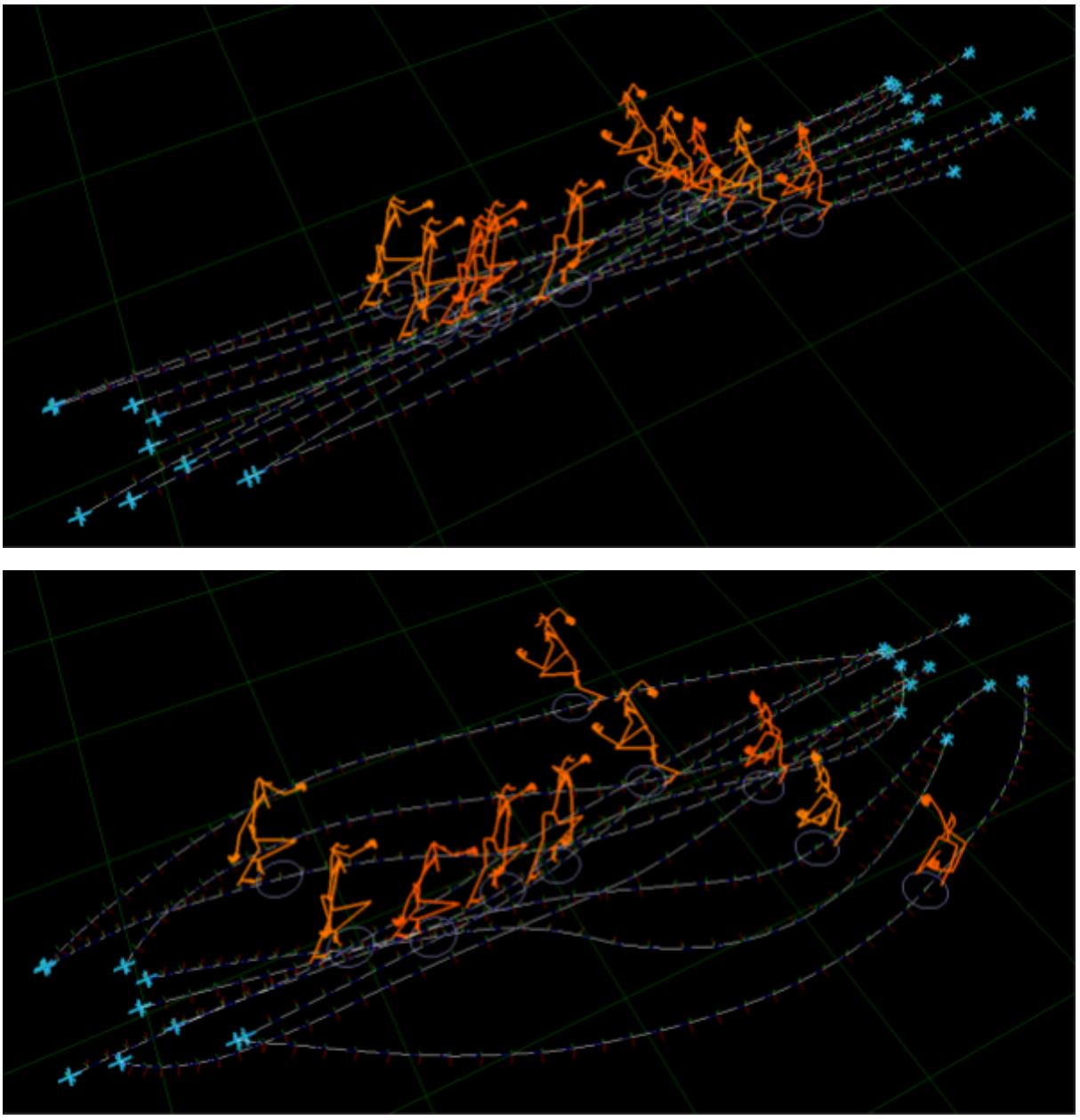


Figure 6.2 A densely positioned set of agents resolve their collisions after 10 iterations.

One downside of this approach is that we may accumulate additional error in the paths with each iteration, leading to some foot sliding. As we are using an already deformed animation as the source of a new solve, we no longer maintain the shape preserving local features of the original animation, and with each iteration may introduce more and more deformation. However, as most collisions can be resolved in a low number of iterations the error is generally acceptable.

This solution also is also computationally slower than previous attempts, as we need to fill sparse matrices with new data rather than using the existing data of the original animation. As such, we conclude that collision avoidance should be applied as a post-processing function rather than during user-editing.

7 Animation layering toolkit

Animation layering involves overlaying the poses of one animation, which we will call the layer animation, over the poses of a source animation. Layering can be applied at run-time, reducing the need for an animator to produce new assets, and giving artists flexibility to re-use existing

animations. This allows them to create a wide variety of new animations that fulfill the requirements of their shot.

A common use of layering is to apply a different upper-body animation on top of a source locomotion animation (walking, running, etc.). A scene involving a street of walking pedestrian could re-use stationary animations of characters using their phones, fidgeting, gesticulating etc., and apply these on top of their walk animations to introduce more variety and life. The layered animations are very quick and simple to edit and add no per-character memory cost.

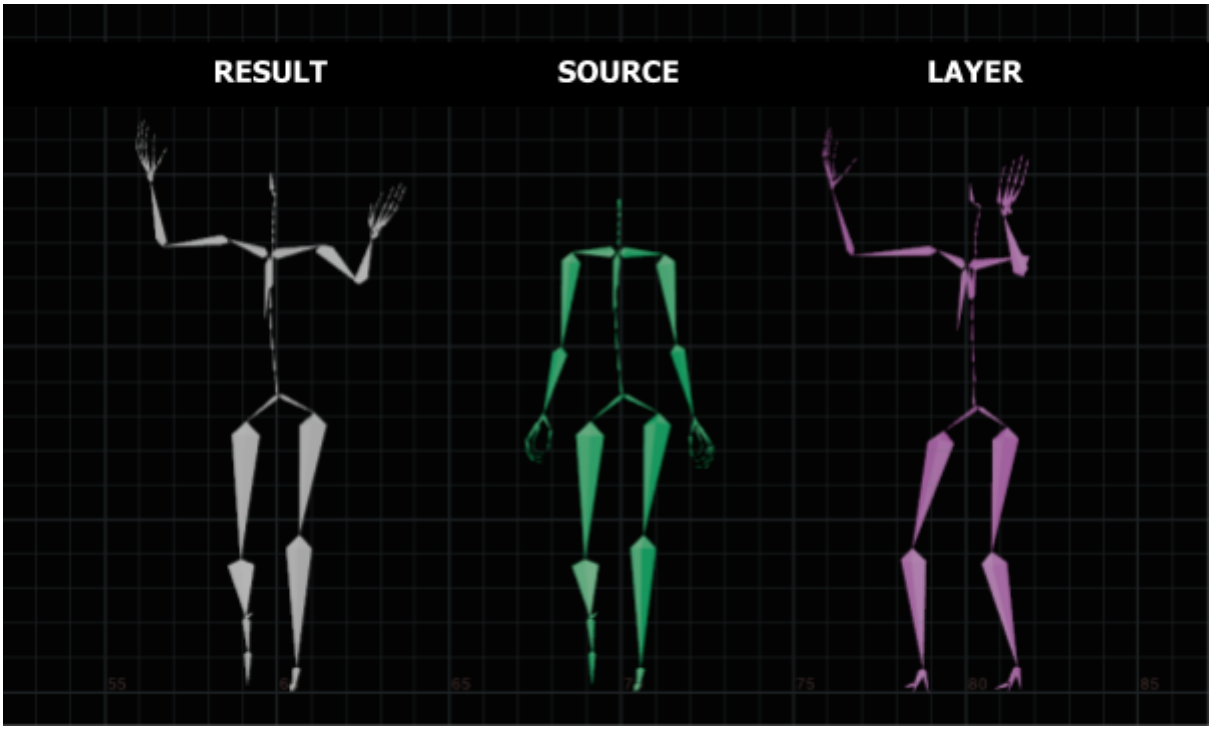


Figure 7.1: Gesticulating layered animation applied to walking source animation.

7.1 Layering Techniques

To layer an animation we first specify a joint of the source animation skeleton. The layering is applied to that joint and every child joint throughout the skeleton. To apply upper body animation we may select a hip, or lower spine joint, which will layer the upper body animation from the spine to the fingers of each arm and the head.

A weight is also specified, which can change how much of the layer animation to apply on top of the source animation. This can be used to alter the look of the animation and potentially add randomisation or variety if multiple characters are using the same animations.

We have implemented two methods for layering, blending and additive.

7.1.1 **Blending**

For each pose in the source animation, we linearly interpolate the local joint transforms from the source transforms to the layer transforms. With a weight of 1.0, the resulting animation will use 100% of the local joint transforms of the layer animation, and 0.0% of the source transforms.

7.1.2 **Additive**

For the majority of situations additive layering produces higher quality animations than blending. Instead of linearly interpolating, the layer animation is added on top of the source animation, preserving both animations [7].

For example, we may apply a stationary layer animation of a character talking on their phone to a walking source animation. When blending, this can produce a stiff output animation where the whole upper body swing with the movement of the walking hips, while the spine and shoulders lose the sway that comes with a walk. As additive layering applies the layer local transforms on top of the source, we have the animation talking on the phone while preserving the walking movement which produces a better quality output.

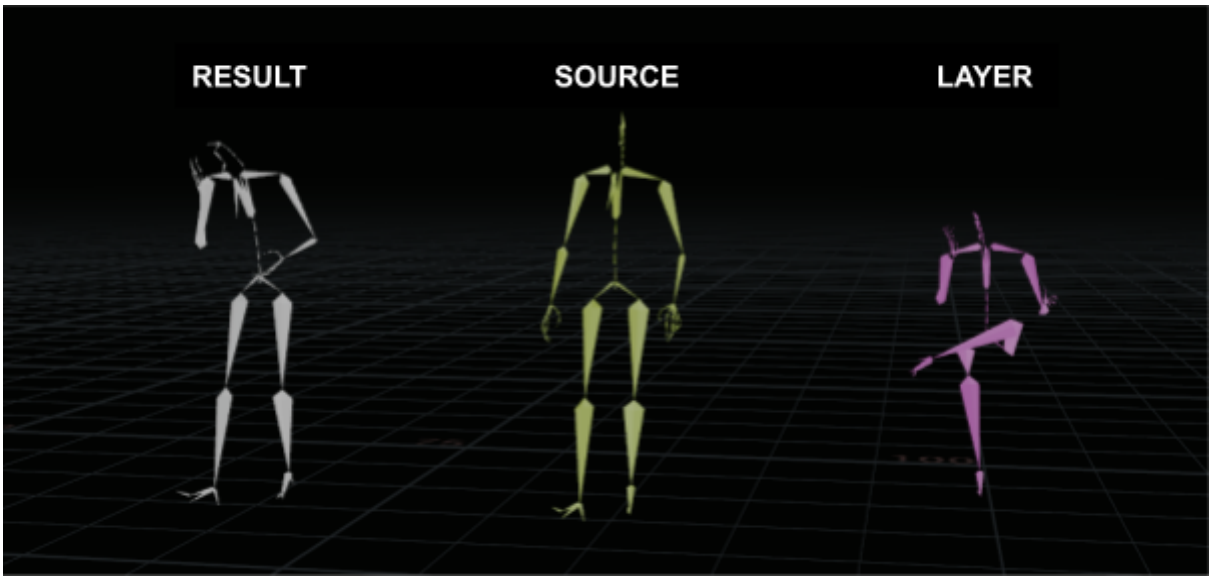


Figure 7.2: Seated and on the phone layered animation applied to walking source animation.

8 Conclusion

Over the course of the time spent on the work package, significant progress has been made in developing new workflows for our artists that don't require simulation.

The tools set out in this document are designed to improve the quality and speed of turnaround of crowd shots at DNEG. Some of the tooling described, specifically the animation layering toolkit, has been deployed to artists and is already allowing artists to re-use sections of animation instead of requiring new animation to be created. The full extent to which this tooling will speed up the turnaround of shots is still yet to be determined, and will require further investigation.

When evaluating the path editing tooling, whilst the developer testing of workflows appears to address a lot of the user requirements, it is unclear to what extent the new features will improve the workflows of artists until they are tested in a production environment.

Integration work with asset synthesis techniques has been employed and is discussed in further detail in deliverable D5.5.

9 References

[1] Igarashi, Takeo, Tomer Moscovich, and John F. Hughes. 2005. "As-Rigid-as-Possible Shape Manipulation." In *ACM Transactions on Graphics*, 24:1134–41. doi:10.1145/1073204.1073323.

[2] Kwon, Taesoo, Kang Hoon Lee, Jehee Lee, and Shigeo Takahashi. 2008. "Group Motion Editing." In *SIGGRAPH'08: International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2008 Papers 2008*. doi:10.1145/1360612.1360679.

[3] Sorkine, Olga, and Marc Alexa. 2007. "As-Rigid-As-Possible Surface Modeling." In *Eurographics Symposium on Geometry Processing (SGP)*, edited by Alexander Belyaev and Michael Garland, 109–16. Aire-la-Ville, Switzerland: The Eurographics Association. doi:10.2312/SGP/SGP07/109-116.

[4] 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.

10 Web references

[5] http://eigen.tuxfamily.org/

[6] http://faculty.cse.tamu.edu/davis/suitesparse.html

[7] https://www.gdcvault.com/play/1012300/Animation-and-Player-Control-in