An automatic tool to facilitate authoring animation blending in game engines

Luis Delicado
Universitat Politècnica de Catalunya
Barcelona, Spain
ldelicado@cs.upc.edu

Nuria Pelechano
Universitat Politècnica de Catalunya
Barcelona, Spain
npelechano@cs.upc.edu

ABSTRACT

Achieving realistic virtual humans is crucial in virtual reality applications and video games. Nowadays there are software and game development tools, that are of great help to generate and simulate characters. They offer easy to use GUIs to create characters by dragging and drooping features, and making small modifications. Similarly, there are tools to create animation graphs and setting blending parameters among others. Unfortunately, even though these tools are relatively user friendly, achieving natural animation transitions is not straight forward and thus non-expert users tend to spend a large amount of time to generate animations that are not completely free of artefacts. In this paper we present a method to automatically generate animation blend spaces in Unreal engine, which offers two advantages: the first one is that it provides a tool to evaluate the quality of an animation set, and the second one is that the resulting graph does not depend on user skills and it is thus not prone to user errors.

CCS CONCEPTS

• **Computing methodologies** → *Animation*; • **Software and its engineering** → *Interactive games*;

KEYWORDS

Realistic Animations, Authoring Animations, Unreal Engine

ACM Reference Format:
Luis Delicado and Nuria Pelechano. 2019. An automatic tool to facilitate authoring animation blending in game engines. In *Motion, Interaction and Games (MIG '19), October 28–30, 2019, Newcastle upon Tyne, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3359566.3360063

1 INTRODUCTION

In recent years we have seen how the game industry is creating games where not only the main character has realistic movements, but also all the avatars of the NPCs (Non-Player Characters) are improving in both appearance and behaviours. In the past, we could observe that games would only use a handful of characters which very often had no behaviours at all (e.g. they would simply stand in a place with at most a basic idle animation, or had a cyclic animation such as walking following a pre-defined trajectory with poor connection between trajectory and animation). Nowadays, thanks to hardware improvements in graphics cards and the availability of popular game engines, we can find games that show many NPCs that make the environment highly realistic and dynamic.

Human motions are in high dimensional space, and there are many solutions in the literature that can handle character animation synthesis successfully but at a high performance cost, which is why they are not included in game engines. When real time is required for large numbers of characters, it is typically necessary to reduce the problem to a lower dimensional space. This is exactly what state of the art game engines do, by providing GUIS to represent animation graphs where transitions are set by the user (both the variable triggering the transition, and the interpolation values along the transition) and using 2D blend spaces (where the axis typically represent velocity and turning angle, but the user can pick others if needed). The main goal of algorithms that focus on synthesizing animations with a low dimensional space is to find a good trade off between performance and plausibility, so that they can be used in real time for large groups of agents.

Academic methods that work in high dimensional space are not yet included in popular game engines. Therefore, when a user needs realistic animations for a game or an academic work where the research effort is elsewhere (e.g: crowd simulation), it is necessary to fall back to standard solutions that are included in current game engines. One could think that current game engines, already solve most of these problems, but this is not really the case, unless you are an expert game developer. Game engines such as Unreal [Unreal 2018] or Unity [Unity 2019], offer tools with nice GUIs where the user can drag&drop animations, and create transitions between them. But it is completely up to the user to determine what variables and values lead the blending of animations. Very often, the results of this blending are quite chaotic and full of artefacts. This is the main motivation for this paper: to research the possibility of creating automatically such animation blending spaces to ease the work of non-expert game programmers.

In this paper we have developed an automatic method to locate animations correctly in the typical 2D blend space offered by game engines. We also present a method to further extend the limited parametric space of game engines, and thus achieve higher dimensionality. The advantages of our method are twofold: (1) our automatic allocation of animations allows the user to determine the quality of an animation set, by highlighting problematic areas of the blendspace or recommendations on how the resulting animations could be improved by replacing or adding certain animations; and (2) since the animations are located automatically on the blend space, it reduces the amount of errors introduced by misalignment of animations or lack of user skills. We believe that this method presents a very useful tool for game developers.

2 RELATED WORK

There has been a large amount of work in the literature on motion blending [Feng et al. 2012] and motion graphs [Kovar et al. 2008]. Motion Graphs are build by creating a node for each pose of an animation clip, and transitions between nodes represent smooth changes between positions. There have been extensions to this concept focusing on increasing the connectivity of the graph [Zhao and Safonova 2009]. However, game engines do not represent a node per position but a full clip. Therefore, transitions are only allowed by blending between two animation clips, and links represent possible blends depending on the value of a user defined variable. In order to carry out motion control for games, there are recent locomotion control methods based on neural networks [Holden et al. 2017; Zhang et al. 2018] that provide very impressive results. Unfortunately, these recent techniques are not typically included in current game engines, and thus not available for the general public.

Since many of the motion control methods are time consuming and can only be applied to a handful of characters, there have been other approaches that attempt to provide good animations with less time consuming techniques, so that they can be used in real time crowd simulation [Pelechano et al. 2011].

Other research use animation graphs to control the foot position [Beacco et al. 2015; Curtis et al. 2011]. These approaches typically solve the foot sliding issues that appear in root simulation approaches. However, they are harder in computer complexity because they require the use of two graphs, one for each foot.

The work by Narang et al., achieved smooth and plausible locomotive behaviours for small crowds (over 60 agents) in real time. Collision avoidance, was computed taking into account high-dimensional human motion and bio-mechanical constraints [Narang et al. 2018].

Animation can be based on the inverted pendulum model, where the feet position is calculated to follow a trajectory while avoiding obstacles [Singh et al. 2011; Tsai et al. 2010]. The work by Hwang et al [Hwang et al. 2018] combined an inverted pendulum model with a physical model to represent high-dimensional character motions. Animation based on inverted pendulum model can run in real-time only for one character.

The way animation synthesis is handled by game engines such as Unreal [Unreal 2018], or Unity [Unity 2019] is by providing a 2D blend space where the use manually locates each animation clip based on two parameters (velocity and turning angle), more parameters can be added by using some graph representation where the user defines the conditions to change between animations or blendspaces [Gregory 2018]. The typical 2D blend space is driven by velocity and turning angle, because those are the main input values from the user to move the character representing the player. If other actions are needed, such as jump, duck or crouch down, they can be included by having a graph with specific transitions to trigger those animations. There are some animation tools compatible with game engines that offer nice blending for an existing animation set, such as SmartBody [Shapiro 2011], however if the user needs to add its own animations it is necessary to create a new blend space. SmartBody is not intended to be a framework for the development of character animation via a well-defined interface or plugging in animation blocks, therefore naive user will still struggle with the generation of new blend.

Generating good blend spaces and animation graphs, requires users to be experts in understanding how transitions, values of parameters and location of animation clips within the blend space, will impact the quality of the synthesized animations. The goal of this paper is to present a simple yet powerful tool, to assist users to obtain good animation results regardless of their experience or lack of. To the best of our knowledge, this is the first paper describing a tool that has been customized to work with the constraints imposed by state of the art game engines. More specifically our current implementation works with Unreal Engine 4 (UE4)[Unreal 2018], but note that all game engines have a similar animation synthesis representations.

3 PARAMETRIC BLEND SPACE IN GAME ENGINES

Current game engines provide tools to generate a parametric space using user friendly GUIs. However, it is completely up to the user to determine the values of the parameters that best describe each animation and manually position them in such a parametric space. Very often, if the user is not an expert on animation, any error or misalignment introduced in the blend space, will lead to artefacts in the results. Moreover, animation synthesis is a high dimensional problem. However, game engines limit the user to represent the animation space with a low number of parameters. Typically the low dimensional space represents a handful of parameters to follow trajectories, which introduces artifacts. Animations that are similar in style in the higher dimensional space, may not be projected as close points in the low dimensional 2D space, and the opposite, simulations that are very different in style may be projected very close in the 2D space. Therefore, blending two near-by points in the 2D space may end up blending two very different animations.

The main goal of our work was to develop an automatic tool that could extract the relevant information from the animations (such as velocities and angles) and compute the correct position of those animations in the blend space provided by Unreal Engine. By doing this, we eliminate user errors introduced during the manual set-up of the blend space. Our tool also provides useful feedback to the user by highlighting situations that may lead to other artefacts.

Once the animations have been located as vertices in the blend space, Unreal will automatically create a Delaunay triangulation with those vertices, in order to cover the maximum area represented by those vertices. (combining all the Delaunay triangles provides a convex hull). Any animation corresponding to a position inside such convex hull will be simulated by blending between the three animations in the triangle containing such point (using the barycentric coordinates as blending weights [Gregory 2018]). Unfortunately, this triangulation is simply based on geometrical properties and not on the animation properties. Therefore, the edges connecting vertices of a triangle have no information about how different those animations may be. This issue, introduces further blending artefacts. For example, it is well known by animators that blending opposite animations (e.g. walking forwards with walking backwards, left and right side steps) will result in abnormal feet movement, so if a triangle happens to include two opposite animations, the user will need to find a way to modify the triangulation. This can be done by either adding more vertices in the blend space, or else modifying the range of values considered for the X and Y axis, so that the Delaunay Triangulation will be updated. There is thus very little control from the user to modify the triangulation, and so advanced skills are required, or else it turns into an iterative trial and error process.

3.1 Allocate animations

The blend space provided by Unreal Engine is presented as a 2D grid, with each axis controlled by one parameter. Typically, those two parameters are the velocity and the angle of the trajectory. To allocate the animation set we use the following interpretation: the velocity is represented as the root displacement between the first frame and the last frame over a period of time of one second, and the rotation is represented as the angle between the torso forward orientation at the first frame and the last frame of the animation clip (assuming that torso orientation is aligned with the velocity vector). Since the value of those parameters can be extracted from each animation clip, we can automatically locate it in the exact position of the 2D blend space grid described above.

To calculate the velocity of an animation, the animation is played and the distance between initial and final transformations are computed and divided by the time elapsed (all game engines provide functionalities to extract such information). Similarly, the rotation can be computed using the difference between the initial and final orientation.

Once the values of those parameters are known, we can automatically allocate the animation in the most adequate position of the grid. Initially the implementation of the UE4 for the blend space uses a regular grid, which means that we have a discretized space and thus we can not allocate the animation clip in the exact position that we wish. To achieve higher resolution, we can increase the subdivisions of the grid, which by default is $4x4$, to $100x100$, thus reducing the precision errors. After allocating all the animations, those that are not exactly in a valid point are displaced automatically to the nearest grid point. The grid dimensions should be adjusted to our animation set.

In figure 1 we show an example for an animation set containing 16 animations. Each animation is located as a node based on its velocity and angle. The total set of 16 nodes forms a graph which is automatically triangulated by Unreal Engine. This representation provides a mapping between any desired velocity and orientation angle, and the corresponding triangle in the graph that will be used to create the new animation by blending between the 3 animations located at the vertices of the triangle.

3.2 Information extracted from Blend Space

Our automatic allocations of animation clips can help the user to evaluate the quality of the animation set, and identify key animations that are missing in the current set, to improve the final interpolated results.

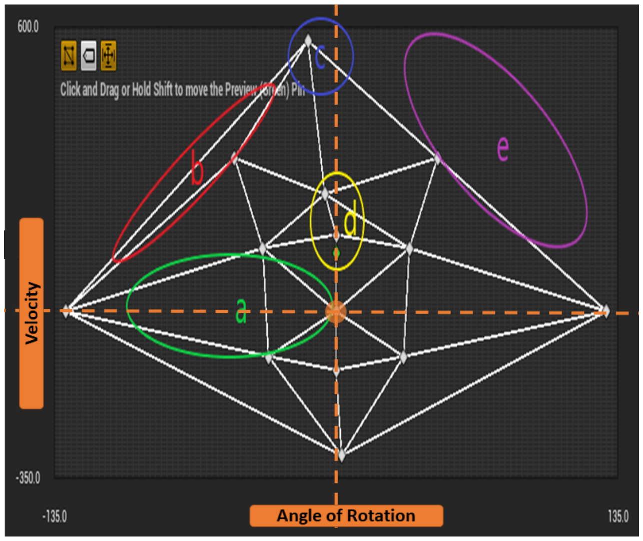

600.0
Click and Drag or Hold Shift to move the Preview (Green) Pin
c
b
e
d
a
Velocity
-350.0
-135.0
Angle of Rotation
135.0

Figure 1: Automatic blend space highlighting problematic areas: a) Crossing axis, b) Acute angles, c) Misalignments , d) Animations too close, and e) Uncovered zones.

Since finding or creating huge sets of animations can be cumbersome, we can use the automatic graph to evaluate which animation clips should be added, modified or removed. Following such recommendations, will result in improved animation synthesis to encompass all the locomotion behaviors needed for our characters.

There are five types of problematic triangulation effects that our automatic method can solve or highlight (Figure 1), which are: (a) Crossing axis, (b) Acute angles, (c) Misalignments, (d) Too close animations and (e) Uncovered areas.

Our system provides an automatic solution to problem c, it highlights and gives recommendations to the user to solve problems b and e, and it gives the user the choice between an automatic solution or a guided manual one to add/delete animation clips to solve problems a and d.

3.2.1 Crossing Axis (a). We know that two animations are opposite, when they appear at different sides of either the vertical or the horizontal main axis in the 2D blend space. So if we identify those axis being the horizontal and vertical segments passing though our idle animation ($\vec{v} = 0$ and angle of rotation $\alpha = 0$), we can easily detect those triangles with opposite animations because they intersect with either one or both axis.

The only way to guarantee that there will be no triangles crossing those edges, is by making sure that we will have 4 key animations corresponding to the edges of the vertical and horizontal segments. Later on, using the idle animation and each of those 4 key animations, we can create interpolated animations to avoid triangles crossing the edges (Figure 1 shows the main axis in orange).

To create the horizontal edge, it is enough with one turning 90° animation (since the opposite one can be automatically created by mirroring this animation). Therefore we will add the following two animations: $(\vec{v}, \alpha) = (0, -90°)$ and $(\vec{v}, \alpha) = (0, 90°)$.

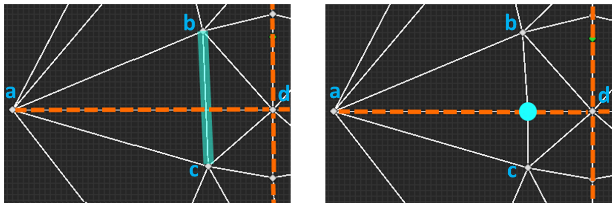


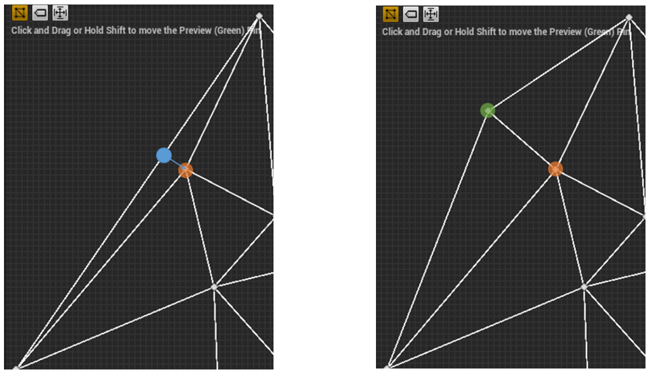
Figure 2: Problem with a triangle edge (in cyan) that crosses the main axis $\vec{v} = 0$ on the left. On the right, the result after adding a new animation (cyan dot) over the axis to avoid triangles intersecting with such axis.

Ideally the user should also include a walk forward and backward animation without any angle, which correspond to $(\vec{v}, \alpha) = (MaxForwardSpeed, 0°)$, and $(\vec{v}, \alpha) = (MaxBackwardsSpeed, 0°)$.

What must be avoided, is triangle edges crossing the main axis with animations that are further from the axis than a given threshold τ_χ (we have empirically found $\tau_\chi = 5$ to provide perceptually good results). Figure 2 (left) shows an example of a situation where this threshold is not satisfied with the horizontal axis (distance from both b and d to the horizontal axis is above τ_χ). In this case, there are two animations with $\vec{v} = 0$ that appear in the extremes of the blend space representation. This provokes that other animations may end up connected crossing the horizontal axis of $\vec{v} = 0$. The solution to this problem involves including new animations over such axis to create additional triangles as indicated in figure 2 (right). To obtain such new animation clips the user can either add his own, or simply allow the system to provide new ones automatically. The automatic animation is computed by calculating the intersection between the axis and the triangle edge (b,c). Then two blending weights are calculated based on the relative distance from the intersection point to the two closest points over the axis (a and d), with the sum of those two weights being 1. Finally the new automatic animation point is obtained by interpolating between those two animations (a and d) with the computed blending weights. The new animation created by interpolation, automatically modifies the triangulation, removing the intersection with the main axis.

3.2.2 Acute angles (b). The second problem is caused by triangles containing very acute angles, which also implies long edges with small area. This can introduce blending artefacts because there will be a vertex very close to an edge, which means that small movements within the blend space, turn into big switches between animation blending parameters. For example, Figure 3, on the left side, shows such a situation. To interpolate the blue point, the barycentric coordinates will provide a blending between the two animations along the large edge of the triangle and 0% influence by the animation marked with the red dot. As we then move slightly towards the red dot, the barycentric coordinates will rapidly assign a weight close to 1 to the red dot and almost no influence from the other two vertices of the triangle. This will result in abrupt interpolation changes.

This problem can be mitigated by adding a new animation that leads to better triangle shapes, as illustrated in Figure 3. Note that our system simply highlights this problem, so that the novice user can understand what area of the blend space is likely to introduce artefacts. With this information the user can try to add alternative animations that our system will automatically locate and provide a new triangulation.


Click and Drag or Hold Shift to move the Preview (Green)
Click and Drag or Hold Shift to move the Preview (Green)
Figure 3: Problem with triangle with long edge but small area and result of add an extra animation to reduce the long edges and increase the triangle areas.

3.2.3 Misalignment (c). The third problem that can be detected analysing the blend space is related to the expected parameterization of our animations. Unlike the previous problems, which are due to triangulations that ignore animation properties, the misalignment problem is caused by the unexpected locomotion of an animation. When the user creates a blend space manually, some animations are really easy to allocate correctly. For instance, the idle animation will have parameter values corresponding to $(\vec{v}, \alpha) = (0, 0)$, a forward walking or running will have $\alpha = 0$, so they correspond to very specific positions in the blend space (along the Y axis). The problem appears when an animation seems to correspond to one of those easy to allocate key positions, but it has a small offset because it was not correctly created.

In the example used to illustrate our method, we can observe this problem in the running forward animation (the animation on the top of the axis $\alpha = 0$). This animation was meant to be used to have the character running forward, however our automatic allocation shows that it has a small turning angle. The fact that the animation clip is not located on the vertical axis, already shows the user that this animation is not a good match for walk forward. This means, that if the user manually allocates such animation over the Y axis, he would introduce an error in the character trajectory after playing this animation for a period of time.

Our method, not only highlights this problem to the user, but it also solves it. Now that the animation is correctly allocated in the blend space, when the user moves the character forward in a straight line, the game engine will not use the wrong animation. Instead it will compute a new interpolated animation between the misaligned run forward clip and the other two in the triangle that contains the desired locomotion parameters. The new result will thus guarantee that the character follows the correct trajectory.

3.2.4 Too close animations (d). If the set of animations is composed of animations from many different data-sets, it is likely to have animations that appear very different (in gait, style, etc) but with their parameter values in $(\vec{v}, \alpha)$ being very close in the 2D blend space. In this cases, it is wiser not too use several animations that fall too close in the blend space, since small changes in the input values may result in large changes of animation behavior. Moreover, having too many similar animations, end up having a negative impact on performance, without increasing the locomotion space coverage.

Analysing our automatic blend space, it is easy to notice those sets of animation clips, and the recommendations would be to keep a good representative one and eliminate the others, or else the user could choose to create a new one that is an interpolation of the existing ones.

In order to do this, the user can input a distance threshold, τ_d, and the system will highlight those animation that are closer than τ_d. By default we use $\tau_d = 8$ since we have observed empirically that it is a good value to eliminate animations that are either too similar, and thus do not add value to the blend space, or else are too different despite their close proximity in the blend space, and thus the constant switching between them results in unnatural looking animations.

3.2.5 Uncovered areas (e). Finally, the automatic allocation of animations allows us to identify areas of the parameter space that will not be covered by any animation. Those areas correspond to the space outside the convex hull of the triangulation. Once those areas have been identified, the user can generate new nodes that satisfy the velocities and rotation angles that are not yet covered by the animation set.

3.3 Providing placeholders

Once all the problems have been highlighted, our system adds placeholders into the automatic blend space. Those placeholders provide information about nodes that should be included to fix all the possible sources of animation artefacts. Each placeholder indicates the values of velocity and rotation that would be needed for the new animations. With the information provided by the placeholders, the user can create a few specific animations that will greatly enhance the final animation synthesis or even accept the recommend ones that the system can automatically generate for cases a, c and d. Figure 4 shows the result after introducing the placeholder. Orange placeholders are necessary animation clips, whereas green ones can be considered suggestions, as they may not be necessary for the user. For instance, placeholders in the corners for case e (coverage) may not be needed if the user considers that there is no need for animation outside the current convex hull. Placeholder for case d indicates that having the two animations within the green ellipsoid may introduce artefacts if the animations are very different at a higher dimensionality space that is not covered by the 2D blend space.

4 INCREASING THE DIMENSIONALITY OF THE 2D BLEND SPACES

To incorporate styles and other locomotions where velocity is not alligned with the torso (e.g. side steps), it is necessary to have a higher parametric space. In this section we explain how we can increase the parametric space within a game engine that only provides 2D or 1D representations.

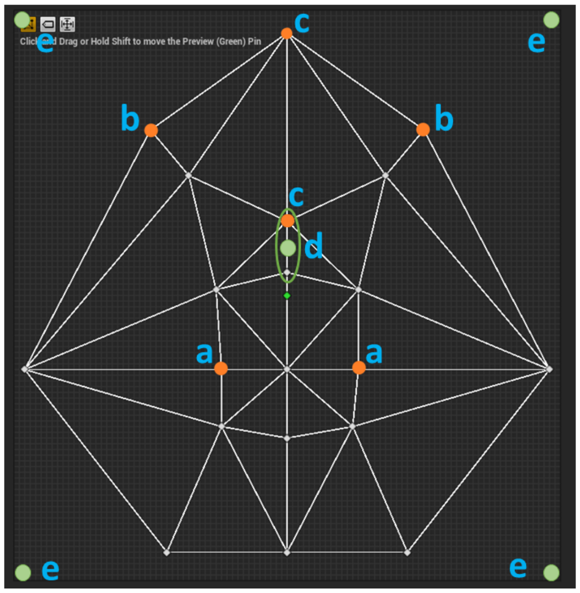

Click and Drag or Hold Shift to move the Preview (Green) Pin
c
e
e
b
b
c
d
a
a
e
e

Figure 4: Result after including placeholders.

Taking into account the limitations of game engines, we could use several 2D spaces with a flag that enables only one of them at a time [Holden et al. 2017]. We can therefore use two blend spaces (both with the parametric space based on $\vec{v}$ and α), one to simulate walking and turning with the torso oriented as the velocity vector BS_1, and another one BS_2 to handle lateral steps. A switch could be used to enable BS_1 or BS_2. However, by including a blend function instead of a switch, we can have $\beta \in [0, 1]$, and thus obtain a smooth blending between three parameters $(\vec{v}, \alpha, \beta)$.

The difference between this approach and a real 3D blend space is that we blend three animations from BS_1 and three from BS_2 , to then blend those two results. With a 3D space, we could select a tethraedron and blend only four animations in one step.

Styles could be handled similarly, by having a 2D blend space for each style. However, given the difficulty of generating large data bases of natural looking animations for each style, we investigated how to add styles to our current database and blend spaces, without requiring a large number of additional animations. Adding styles to the upper body, can be easily incorporated in game engines, by keeping a 2D blend space for the locomotion, and blending or overwriting different styles or actions exclusively affecting the upper body (see Unity animation controllers and Unreal animation layers). The difficulty appears when we want to add style to the legs movement, to simulate for example characters walking with a limp, drunk, being agile or tired.

To add style without generating a new 2D blend space, we propose to combine style from a simpler 1D blend space with a larger 2D space containing a full range of locomotor behaviors. We simply need to generate a 1D space with animations that walk forward and backward in a straight line with a specific style. Since this set of animations is significantly smaller, it is thus easier to find examples or generate new ones. Then the velocity $\vec{v}$ is used to get a result from the 1D BS_{style}, and the pair $(\vec{v}, \alpha)$ to obtain a results from the locomotion 2D blend space. Finally both results can be blended with a user defined parameter *style* describing how intense the style needs to be noticed in the final results.

5 RESULTS

Looking at the example used to illustrate the steps of our algorithm, the analysis described in section 3.2, suggests that we need to add at least five new animations in order to improve our final animation synthesis (see Figure 4). Since we can use a mirroring animation to obtain symmetric behaviours, we actually only need three animations. To split the long axis, we could add two animations for running with a slight turn left or right, to describe a curve. Next we need an additional animation on the $\vec{v} = 0$ axis (and its symmetric) to avoid triangles cutting the horizontal axis (note that we could also simply accepts the ones that the system can automatically generate as explained in section 3.2.1). And finally, it is recommendable to add an animation to represent running forward with $\alpha = 0$, and remove the one that appears at the top with a misalignment.

Adding style by combining a 1D and a 2D blend space, can improve variability to any animation set. However, there are limitations in the way that this method can be incorporated in current game engines. For example, in the case of the 1D drunk style, if we assign 0.3 to the parameter *style* it will look less drunk than if we assign 0.5. This presents a trade-off in the control between adding styles and accuracy of the trajectory, since higher weight assigned to style implies a lower weight for the 2D locomotor blend space.

In Figure 5 we show results of drunk trajectories combining our 2D blend space with a new 1D space containing a walk forward/backward drunk animation.

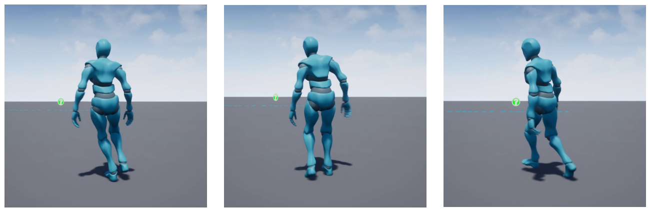


Figure 5: 1D drunk animations blended with 2D normal locomotion.

Finally, as can be observed in the videos [1] our method can successfully combine two 2D blend spaces, to achieve a larger number of trajectories with different torso orientation.

The main advantage of our work, is that performance depends on the game engine, since our method is computed off-line and does not increase the computational cost. We show in the videos examples with small crowds, where we have used the same automatic blend space for all characters, but it would be easy to add variety by combining the main locomotion 2D blend space with other 1D blend spaces to add style. The frame rate achieved will depend on the game engine, since our method simply aids the authoring animation steps previous to simulation. Our solution could work with other game engines, as the concept of animation graphs and 2D triangulated blend spaces is the de-facto method in the industry.

[1]https://www.cs.upc.edu/~npelechano/videos/MiG2019.mp4

6 CONCLUSION AND FUTURE WORK

This paper presents a new method to facilitate the generation and analysis of character animation synthesis using Unreal Engine 4. Our first contribution is an automatic method to generate blend spaces in UE4, achieving good result regardless of users' skills. Furthermore, the result of our method provide a powerful tool to highlight relevant information about the animations quality. An analysis of the resulting blend space, can be carried out to identify flaws in the animation data set, such as: motions that will not be well covered, or that will suffer from misalignment. The perceptual quality of the resulting interpolated animations will still strongly depend on the 3 animations chosen from this representation, but our automatic allocation guarantees that the characters will follow the input trajectory with precision. This is a problem inherent from mapping the high dimensional space of human movement into a lower one that game engines can work with to synthesis animations in real time. The second contribution consists on increasing the realism of character animation by combining blend spaces, either by including style or a larger range of locomotion types.

As future work, it would be interesting to carry our perception studies to quantify the specific properties of the animation vertices that makes animations suitable or not to blend between them.

ACKNOWLEDGMENTS

This work has been funded by the Spanish Ministry of Economy and Competitiveness and FEDER under grant TIN201788515-C2-1-R.

REFERENCES

Alejandro Beacco, Nuria Pelechano, Mubbasir Kapadia, and Norman I. Badler. 2015. Footstep parameterized motion blending using barycentric coordinates. *Computers & Graphics* 47 (2015), 105–112.
Sean Curtis, Ming C. Lin, and Dinesh Manocha. 2011. Walk This Way: A Lightweight, Data-Driven Walking Synthesis Algorithm. In *ACM Conf. on Motion in Games*.
Andrew Feng, Yazhou Huang, Marcelo Kallmann, and Ari Shapiro. 2012. An analysis of motion blending techniques. In *ACM Conf. on Motion in Games*. Springer, 232–243.
Jason Gregory. 2018. *Game engine architecture*. AK Peters/CRC Press.
Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned neural networks for character control. *ACM Transactions on Graphics* 36 (2017), 42:1–42:13.
Jaepyung Hwang, Jongmin Kim, Il Hong Suh, and Taesoo Kwon. 2018. Real-time Locomotion Controller using an Inverted-Pendulum-based Abstract Model. *Computer Graphics Forum* 37, 2 (2018), 287–296.
Lucas Kovar, Michael Gleicher, and Frédéric Pighin. 2008. Motion graphs. In *ACM SIGGRAPH 2008 classes*. ACM, 51.
Sahil Narang, Andrew Best, and Dinesh Manocha. 2018. Simulating Movement Interactions Between Avatars & Agents in Virtual Worlds Using Human Motion Constraints. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE, 9–16.
Nuria Pelechano, Bernhard Spanlang, and Alejandro Beacco. 2011. Avatar locomotion in crowd simulation. *International Journal of Virtual Reality* 10, 1 (2011), 13.
Ari Shapiro. 2011. Building a character animation system. In *ACM Conf. on Motion in Games*. Springer, 98–109.
Shawn Singh, Mubbasir Kapadia, Glenn Reinman, and Petros Faloutsos. 2011. Footstep navigation for dynamic crowds. *Computer Animation and Virtual Worlds* 22, 2-3 (2011), 151–158.
Yao-Yang Tsai, Wen-Chieh Lin, Kuangyou B Cheng, Jehee Lee, and Tong-Yee Lee. 2010. Real-time physics-based 3d biped character animation using an inverted pendulum model. *IEEE transactions on visualization and computer graphics* 16, 2 (2010), 325–337.
Unity. 2019. Unity3D. https://unity3d.com/
Unreal. 2018. Unreal Engine 4. https://www.unrealengine.com/
He Zhang, Sebastian Starke, Taku Komura, and Jun Saito. 2018. Mode-adaptive neural networks for quadruped motion control. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 145.
Liming Zhao and Alla Safonova. 2009. Achieving good connectivity in motion graphs. *Graphical Models* 71, 4 (2009), 139–152.