Increasing diversity and usability of crowd animation systems

Ciprian Paduraru
University of Bucharest and Electronic Arts
Bucharest, Romania
ciprian.paduraru2009@gmail.com

ABSTRACT

Crowd systems are a vital part in virtual environment applications that are used in entertainment, education, training or different simulation systems such as evacuation planning. Because performance and scalability are key factors, the implementation of crowds poses many challenges in many of its aspects: behaviour simulation, animation, and rendering. This paper is focusing on a different model of animating crowd characters that support dynamically streaming of animation data between CPU and GPU. There are three main aspects that motivate this work. First, we want to provide a greater diversity of animations for crowd agents than was possible before, by not storing any predefined animation data. Another aspect that stems from the first improvement is that this new model allows the crowd simulation side to communicate more efficiently with the animation side by sending events back and forth at runtime, fulfilling this way use-cases that different crowd systems have. Finally, a novel technique implementation that blends between crowd agents' animations is presented. The results section shows that these improvements are added with negligible cost.

Keywords
animation, crowd, skeleton, GPU, blending, memory, compute shader, vertex shader

1 INTRODUCTION

Crowd simulations are becoming more and more common in many computer graphics applications. It is a critical component nowadays in video games industry (games like Fifa 2017®, Ubisoft's Assassin's Creed®, RockStar's Grand Theft Auto® series), evacuation planning software (e.g. Thunderhead's Pathfinder software®) or phobia treatments applications, where it is being used to create realistic environments. However, crowd simulation and rendering is a costly operation and several techniques were developed to optimize CPU, GPU or bandwidth usage to have as many agents as possible. In our paper, by agent, we mean an individual visible entity in the crowd. Entities can be humans, cars or every other instance the client desires to configure and use. We split a crowd system implementation into two logical parts: simulation and render side. The simulation usually deals with driving behaviours, actions, and events for crowd agents, while the render side deals with animation and rendering of the crowd's agents. The focus of this paper is on the render side, and especially on how to deal with the agents' animations. The main motivation to study animation systems for crowds is that realistic motion and diversity of animations are important aspects of user perception. There are three contributions that this paper adds to the field of animations for crowd systems:

- We do not require to store all animations pose data on the GPU side as similar solutions does. Instead, we use a streaming model of poses which has insignificant performance overhead; This allows crowd systems to have a greater diversity of animation data than before ([Rud05], [Ngu07], [Sho08], [Bea14]).

- Fulfill the requirements of modern crowd systems, where the simulation side often needs to request complex blending between the motion of groups or individual agents, by feeding dynamic input states or events. The inverse communication is also possible: animations can trigger events to the simulation side (more details on Section 3).

- At the moment of this paper, the first documented method to perform blending between animations of agents that share the same animation data streams with different time offsets.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Overall, we consider that by using our approach, animation systems for crowds can get more usability and variation with minimal performance loss.

The rest of the paper is organized as follows. Related work in the field is presented in Section 2. A deeper introduction into current techniques and some other technical requests that are driving our work are presented in Section 3. Section 4 describes the usability of our framework and strategies that we use to implement it. Comparative results are shown in Section 5. Finally, conclusions are given in the last section.

2 RELATED WORK

The ability to adjust the skin mesh of a character without having to redefine the animation is highly desirable, and this is possible using a technique called vertex skinning [Ryd05], [Bea16]. By using vertex shaders, a lot of the computation that was previously done on the CPU side is now moved on GPU side, which proves to be faster for this kind of computations [Sho08]. The main technique applied for rendering animated characters using vertex skinning in crowd systems is called skinned instancing, which significantly reduce the number of draw calls needed to render agents. The way skinned instancing work in [Rud05], [Ngu07], [Sho08], [Bea14] is by writing all the animations data on a GPU texture at once, then have the vertex shader perform skinning operations by extracting the correct pose from that texture for each individual instance. In our approach, the difference is that the GPU texture doesn't have to be filled at initialization time with all animations data. Instead, with some bandwidth cost, we send the poses of some template animations from CPU to the GPU, facilitating this way client driven blending between motions.

While the skinned instancing and skeletal animation are highly desirable for rendering high-quality agents for crowds, mainly because of the current GPUs capabilities nowadays, there are other techniques that can be used to limit the number of polygons rendered without having users notice artifacts. These methods are described below and are used in our framework to decide how to render characters at a lower level of details (LOD). Even if the focus of our paper is to use skeletal animations using instancing for crowds, these techniques can reduce substantially the resources needed to render and animate agents at lower LODs. Image-based approaches have been used to simplify rendering of large crowds in an urban simulation [Tec00]. A discrete set of possible views for a character is pre-rendered and stored in memory. A single quad is then used when rendering the character by using the closest view from the set. Also, as mentioned in [Mil07], impostors present great advantages in current graphics hardware when rendering crowd characters using instancing. In the shader program, the current viewing frustum and character heading can be obtained to compute the texture coordinates that are most similar to the current view and animation pose. In the same category of optimizations, but targeted for simplifying the work of content creators (animators), is the cage-based method mentioned in [Kim14]. This method can be used for lower level of details in parallel with our technique, to increase the number of animations available for lower LODs.

In our approach, we consider that blending between motions of individual or larger groups of agents can be requested by the simulation side (e.g an agent exiting from a boids behavior because an emergent event started). This means that a decision mechanism at simulation layer (e.g. a finite state machine, a behaviour tree or even a neural network) feeds our animation blending mechanism with concrete parameters. Several papers presented below provided an inspirational point for our work and can be used in conjunction with our techniques as plugins.

Motion warping [Wit95] is one of the core techniques used to create whole families of realistic motions which can be derived from a small subset of captured motion sequences and by modifying a few keyframes. [Kov02] presents a method that automatically blends motions in the form of a motion graph. This method can be used by the locomotion system of a crowd to choose the animations needed at runtime from an initial database of assets. The main strategy is to identify a portion between two motion frames that are sufficiently similar such that blending is almost certain to produce a nice looking transition. The identification process compares the distances between the pairs of frames of the two motions to identify the best points for transition. The concept of registration curve was introduced by [Kov03]. Considering the problem of blending two motions, the registration curve creates an alignment curve which aligns the frames for each point on the timewarp curve. A statistical model that can do unsupervised learning over a motion captured database and generate new motions or resynthesize data from it, is introduced by [Bra00] under the name of Style Machines. Finally, an interesting survey of the animation techniques used by several games engines is presented by [Greg14]. This proved to be a motivation for this paper, due to the complex requirements of different game engines.

Compression techniques have also been studied in several papers. One that can be adapted to the same use case as our paper, is [Ari06] where the author is mainly using PCA and clustering (along other tricks, such as virtual markers per bone instead of angle) to compress clips of a motion database that is to be streamed at runtime. The same problem is also tackled in [Hij00] and [Sat05]. Another technique to reduce the memory

needed to store animations is to use dual quaternions representation [Kav07]. These memory footprint optimization techniques can be used independently of the methods described in this paper to allow even more variety. A study for improving crowd simulations in VR is described in [Pel16].

3 METHODS

This section provides an introduction on how skeletal animation works, how the existing solutions are implementing this technique for crowd animations and the new requirements for different crowd systems that appeared over the few past years.

3.1 Basics

One of the most used techniques for animating 3D characters is skeletal animation [Mag88]. Using this technique, a skeleton (represented by a hierarchy of bones) and a skin mesh must be defined for an animated character (a skin mesh is a set of polygons where vertices are associated with their influencing bones and their weights; the process is called rigging). A pose is a specification that assigns for each bone of the skeleton a geometric transformation. The process of skeleton and mesh authoring is defined in a reference pose (usually called bind pose). An animation can be defined by a series of keyframes, each one defining a pose and other metadata attributes or events (as described below). When playing an animation for a given character, the vertices of its skin mesh will follow the associated bones (also called skin deformation). The formula for computing each vertex transform is given in Eq. 1. Consider that each vertex with the initial transform v in the reference pose is influenced by n bones. $M_{ref_i}^{-1}$ is the inverse transform matrix of the ith bone's reference pose transform. This moves the vertex multiplied on the right from model space to the bone's local space. Multiplying this with M_i, the world space bone's transform in the current pose, returns the vertex transform with respect to bone i at the current pose. Finally, multiplying this with the associated weight w_i, and summing up all results give the final vertex transform in world space.

$$v' = \sum_{i}^{n} w_i M_i M_{ref_i}^{-1} v, \quad where \sum_{i}^{n} w_i = 1 \qquad (1)$$

3.2 Current Techniques

The common pattern in animating crowds is to store the entire animation data set on the GPU memory [Ngu07], [Sho08], [Bea16]. The memory representation is a texture where animations are contiguously stored on texture's rows, each row representing a single pose for a single animation. Since the bone's transformation can be stored as a $4x3$ float matrix, and knowing that a texel (a cell in a texture) can store 4 floats, then each bone transformation for a pose can be stored in 3 columns of a texture. The skin mesh geometry is stored on GPU memory, in vertex buffers. A vertex shader program created for skinning transforms each vertex according to equation 1. The weights and $M_{ref_i}^{-1}$ matrix are constant and provided at initialization time. Knowing the current time and animation index assigned to each vertex, transform M_i, is sampled from the texture mentioned above.

The animations must be shared between agents since the memory requirements and processing power won't allow playing individual animations per each agent on large crowds at a reasonable framerate (e.g. have a set of normal walk animations being shared by all agents having a walking speed). To break repetition, an offset system is usually used: if two agents A and B are sharing the same animation T, then agents can have different time offset in this animation, randomly assigned. If offsets are different, then the user could hardly notice the sharing since the postures of agents A and B are different at each moment of time [Ngu07].

3.3 Current limitations and requirements

In the past few years, the requirements for animation systems have evolved significantly [Greg14], and the current documented implementations of crowd systems described above are not able to satisfy these requirements. A collection of these are defined below:

1. Animation state machines instead of simple animation clips are more and more common, with transitions between clips decided by a decision-making layer on the simulation side.
2. Generalized two-dimensional blending depending on input parameters feed at runtime.
3. Animation clips can be authored with event tags such that when the playback gets at certain points on their timeline it triggers an event to the simulation side (e.g play a sound when an agent is hit at correct timing).
4. Partial skeleton blends depending on state and animation layering: agents should be able to play multiple animations at the same time (e.g. walking and waving hands only when observing the human user).

Since the animation data is statically stored in a GPU texture for performance reasons, the above requirements can't be satisfied because input feed and decisions dynamically taken from the simulation side can have only a limited effect on the agent's animations (i.e. the system could allow only simple blending between existing poses). Also, storing the entire animation data set on GPU would significantly limit the number of possible animations that a crowd system can use. These

two main limitations are addressed by our solution and described in Sections 4 and 4.3.

3.4 Animation controllers in our framework

As stated in Section 3.2, a visual animating character definition can be represented as a pair $AnimDef = (Skeleton, Skinmesh)$. Additionally, the implementation of its animation needs a pose buffer and a root animation controller: $Anim_A = (AnimDef_A, P_A, Ctrl_A)$

The pose buffer represents the transformation data for each bone at the current time of the animation: $P_A(i)$ = transform of the ith bone of the animation A's skeleton. The concept of animation controller used in our framework is similar to the one presented in [Greg14] and [Uni16]. Usually, every animation framework system has a visual editor that let users customize a controller and its internal evaluation operation. A base use case is to define an animation controller as representing a single motion clip (no child). Its evaluation returns the pose data at the specified time parameter, and it could involve decoding the animation clip data and performing interpolations between keyframes. Controllers can be represented as trees of operations. Evaluating a controller at time t, means evaluating recursively its children nodes then combining the pose buffer from each child into its own pose buffer (the one attached to the animation it is controlling).

Listing 1: Pseudocode for evaluating a controller

```
ControllerA::Evaluate(t)
    ChildrenList = {C_i |
      C_i is the i^th child controller}
    Evaluate(t, ChildrenList)
    P_A = Combine( P_{C_i} ).
```

Another base use case is to use a controller to blend between two animations (A and B). Such a controller can have two children controllers: $Ctrl_A$ and $Ctrl_B$. As shown in Eq. 2, the resulting pose of evaluating $Ctrl_C$ is an interpolation between the resulting poses of its two children by variable s (normalized blend time; 0 means start, 1 end).

$$P_c(t) = P_a(t) + (P_b(t) - P_a(t)) * s, \quad s \in [0,1]. \quad (2)$$

Complex trees of controllers can be customized for an animation. For instance, one could use a blend mask to consider only parts of the bones from each children controller. Eq. 3 presents a controller evaluation with three children: from $Ctrl_A$ it takes only the pose for head, $Ctrl_B$ gives the pose for arms, and finally $Ctrl_C$ provides the pose for legs of a biped character. A blend mask is defined as an array of $0-1$ values and has the same length as the pose array. The dot product between the two cancels the pose for bones that are not interesting for the mask (e.g. only the set of bones $S = \{i | B_{mask}(i) = 1\}$ are considered).

$$P_r(t) = B_{head} * P_a(t) + B_{arms} * P_b(t) + B_{legs} * P_c(t). \quad (3)$$

Another example of common animation controller are state machines ([Greg14]) where the transitions are generated / evaluated by triggers / values set from the simulation side (e.g. an AI system). If the controllers above can be implemented on GPU side using shaders for optimizing performance, the ideal running place for the controller representing a state machine would be CPU because of the tight communication between simulations and online data that are usually provided on the CPU side. Another motivation for running controllers on the CPU side instead of GPU, is the use case of authoring motion clips with certain events on their timeline that need to be communicated back to the CPU side. The frequent communication from GPU to CPU would cause serious performance problems that could eliminate the benefits of doing the math operations in shader programs. Usually, modern animation frameworks allow users to customize their own controllers and inject them into the animation system using a provided editor.

4 IMPLEMENTATION

Our solution to solve the requirements 1-4 described in 3.3 and the memory limitation of having all animation data set in memory is addressed by using a mixed model between streaming animations data from CPU to GPU, and storing only a part of the animation data set on the GPU memory.

4.1 User authoring and control

In our framework, an animation stream represents an extension to the tuple definition of an animation (A), as given in Section 3.4: $AStream_A = (AnimDef_A, P_A, Ctrl_A, PH_A)$. The last parameter added is a circular buffer storing the history of the last M values evaluated for P_A. Parameter M can be configured by user and represents the number of frames to be saved in the PH_A buffer - many applications typically consider sampling at $30Hz$ or $60Hz$).

The user input for our framework system can be defined as a tuple: $UserConfig = (StreamPool : AStream[], UniquePool : Anim[], NT, M)$. The StreamPool array contains streams of animations that can be shared by multiple agents. As a demo setup example for a crowd animation system, three types of animation streams could be defined: one playing a walk controller

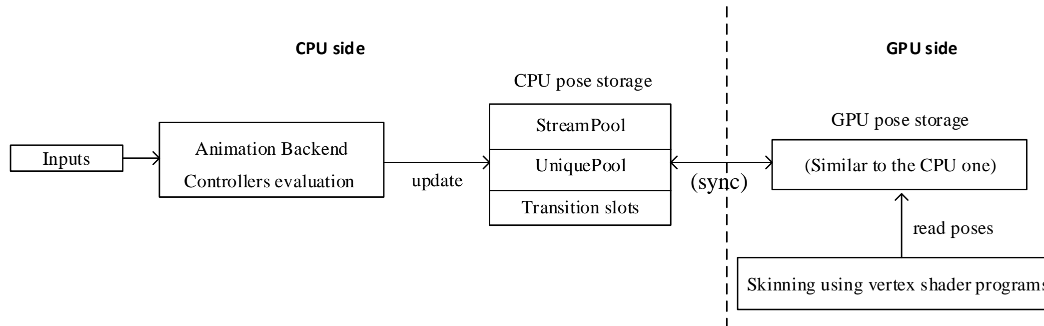

Figure 1: Overview of the CPU-GPU data flow on each frame.

animation, another playing an idle, and finally one waving hands to the user when approaching the camera. Initially, the agents are assigned to one of the streams in the *StreamPool* array, and one offset value in the space of the history poses of that animation stream (random normal distribution or customized by user).

More specifically, if each crowd agent has a unique id assigned, then its animation reference can be defined as a tuple: $AgentAnim_{ID} = (ID_{SI}, ID_O)$, where the first argument is its stream index in StreamPool, and the second is the offset in the history of that stream's saved poses (PH_{SI}). The offset value is relative to the PH's head (i.e. an offset value of 0, means the head of the ring buffer, while an offset N, represents N frames behind head - $PH(N)$). The purpose of the pose history buffer is to allow multiple agents that share the same animation stream look differently by having different offsets. From a quality perspective, having randomizations both at stream index and offset levels decreases significantly the probability of user observing agents that play the same animation at any point in time. This probability can be controlled by adding more or fewer streams of animations and by modifying parameter M.

UniquePool is an array of simple animation definitions that are not meant to be shared between agents - e.g. users can inject at runtime live motions recorded which they want to replicate for the agents in the crowd. Using a streamed animation for this use-case would generate a useless memory footprint for storing the PH parameter. Finally, parameter NT defines the number of preallocated transition slots. Transition slots are internal customizable components that can be used to blend the animation of a single agent from his current animation (and offset) to another one (more details about them in Section 4.3). At runtime, user can request a transition from the current animation of an agent to one of the animations from the stream or unique pools:

- $TransitionToStreamed(ID, streamIndex)$
- $TransitionToUnique(ID, uniqueIndex)$

4.2 CPU-GPU data flow

The data flow between CPU and GPU on each frame is presented in Figure 1. The *Inputs* block is responsible for gathering the inputs for the animation system (e.g. decision making systems deriving states for animation controllers executing state machines or live recorded motions providing online data). Component *AnimationBackend* executes all registered animation controllers and updates their pose information. The CPU pose storage object contains the history poses for *StreamPool*, and only the current pose for unique animations and transition slots. If the pose data for *UniquePool* and transition slots look like an array (one component for each used animation), the *StreamPool* has a more complex representation, presented in Figure 2.

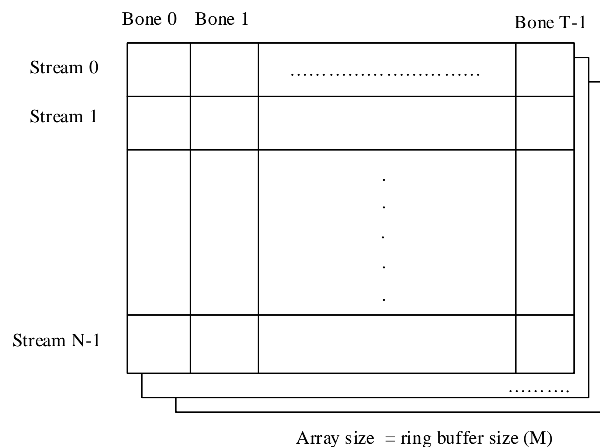

Figure 2: StreamPool storage representation for both CPU and GPU, having N pooled animations, T maximum number of bones and M slices. Each row of a slice represents a pose buffer and has enough capacity to store all skeletons used in the animation system.

At each frame F, for each streamed animation (S), the root controller attached to these animations is evaluated

and writes data on slice (array index) $F \bmod M$, on row $IndexOf(S)$. All these updates (plus the unique and transition slots' poses) are packed and sent to GPU to keep in sync both pose storages. The GPU needs the pose data for skinning purposes, since the shader knows for each vertex the global animation index (unified index system between streamed, unique pools and transition slots), the offset (valid in the case of streamed animations) and fetches the transformation matrix associated with the bones that affect that vertex. The CPU side needs the same pose data because of the blending system explained in the next section.

4.3 Blending between animations

The blending operations in the crowd animation systems is different from the traditional blending mainly because of the opportunities that appear in reusing computations, sharing strategies, and the offset system added to support variety. The transition slots from Figure 1 are responsible for blending between animations, and support transition requests between any animations A and B, with both of them $\in \{UniquePool, DynamicPool\}$. A transition slot T can be defined formally as a tuple:

$Tr_T = (P_T, Ctrl_T, A, B, O_A, O_B, T_0, L, StartP_T, TargetP_T)$. As with the previous definitions, $Ctrl_T$ is the controller attached to this animation and needs to be evaluated to write the output - P_T, representing the current pose of the transition. Skinning is performed using this current pose, similar to the system described in 4.2 (but considering the offset as 0 since only the current pose is written). A, B are the source respectively the target animation for blending, while O_A and O_B, are the offsets that the agent using this transition slot has in each animation (because animations can have different lengths, or for variety purposes, the user can request to blend the agent to a different offset than his current one). T_0 is the start time when the transition started, while L represents the transition duration. Finally, the last two components are explained in the next sub-section.

4.4 Evaluation dependency graph

A typical blend operation between animations A and B using controllers would follow the steps defined in Section 3.4: evaluate controllers for both animations then combine the resulted poses. In the crowd animation system presented in this paper, and knowing that most of the agents are transitioning between streamed animations, blending can be done faster by reusing evaluation results on each frame. The strategy used by our implementation is to create a dependency graph, where tasks (nodes) are the set of all active controllers, and links between them represent dependencies. There is a link between $Ctrl_1$ and $Ctrl_2$, if controller $Ctrl_1$ needs the evaluation results (pose) from $Ctrl_2$. In the case of animations $\in \{StreamPool, UniquePool\}$, there is no dependency. If the transition T is used to play a blending animation between A and B, then $Ctrl_T$ depends only on $Ctrl_B$. The reason why it doesn't depend on $Ctrl_A$ too is that a snapshot of A can be saved, as shown below. Parameter $TargetP_T$ represents the target pose and is evaluated per frame, as follows:

- If $B \in UniquePool$, then $TargetP_T = P_B$ (the pose evaluated in the current frame)
- If $B \in StreamPool$, it must take the value from the pose history of B corresponding to the agent offset in the target animation: $TargetP_T = PH_B(O_B)$.

Parameter $StartP$ represents the pose that the agent had at the moment of transition request in animation A (fixed during transition and internally initialized at the request moment of time):

- If $A \in UniquePool$, then $StartP_T = P_A$
- If $A \in StreamPool$, $StartP_T = PH_A(O_A)$.

Practically, the blending operation is done on each frame between $StartP_T$ and $TargetP_T$, by using an interpolation that considers the current time of transition $(t - T_0)$ and total transition time (L). Since the starting pose is static, an additive blending method is suitable. Listing 2 shows the pseudocode for evaluating the transition controllers.

Listing 2: Pseudocode for transition controller with S representing the normalized blend time, and P the result of evaluation. Note that, according to the definition given above, when transitioning from animation A to animation B, the $StartP$ variable holds the current pose of animation A at the moment of the transition request.

```
TransitionController::Evaluate(t)
    wait Ctrl_B job to finish
    S = (t−T0)/L
    if B ∈ UniquePool then
        TargetP = P_B
    else
        TargetP = PH_B(O_B)

    DiffPose = (TargetP − StartP)*S
    P = StartP + DiffPose
```

4.5 Parallelization on CPU and GPU

The dependency graph described in the previous subsection can be parallelized efficiently since only the evaluations of transition controllers have dependencies. More, there is one subtle observation that can remove the dependency if the target animation is in $StreamPool$: the wait on $Ctrl_B$ is needed only if

$PH_B(O_B)$ is not already computed (there is a good change to avoid the wait since agent's offsets are statistically behind the head of the pose buffer). For the same use case, the biggest parallelization improvement in our implementation was to move the transition controllers evaluation on the GPU side (i.e using compute shaders). These computations are suitable for compute shaders since the only job of the evaluation is to perform a straight interpolation between poses, and these are already cached on the GPU memory in sync with the CPU storage (Section 4.2).

5 EVALUATION

The animation techniques described in this paper were used in the FIFA17® game (https://www.easports.com/fifa). The purpose was to animate a crowd of over 65000 agents displaced in a football stadium, representing different categories of football fans (home, away or ultras). Their animations involved: scoring celebrations, disappointing reactions, anticipation of goals, disagreeing the referee or players' decisions, walking around chairs, etc. Each of these were considered templates and in our tests, we had 256 animations defined in $StreamPool$, and a maximum offset (M) of 60 frames. The number of transition slots allocated was 800, and this number was mainly targeted to support the mexican wave celebration in the stadium (i.e. characters in a specified stadium's area were supposed to stand up and raise their hands at specified moments of time).

For the first purpose of this paper, we are analyzing below the variety of animations in the stadium's crowd that was possible using the techniques described in [Ngu07], [Sho08], [Bea16] (and used in the previous editions of our game) compared against our new implementation. The mixed technique between sending pose data from CPU to GPU and storing only a part of the animations data on the GPU memory increases the variety of supported animations for crowd agents.

On top of this optimization, our new animation system supported dynamic input events as the ones described in Section 3. We were able to make the crowd more realistic with event driven behaviors. One example was agents waving hands or performing different animations on specific bones when a goal scorer was close to them, which was not possible before using preprocessed animations data stored on the GPU memory.

For each skeletal motion clip, on each sampled frame, data must store the SQT (scale, quaternion and transform) of each bone. Denoting by $NBones$ the number of bones of the skeleton and by $SizePerBone$ the average compressed data for storing the SQT per bone, then the pose data size for a single frame is $PoseSize = NBones \times SizePerBone$. Denoting by $MotionLength$ the average number of motion keys per animation, the average size of a clip is: $ClipSize_{OLD} = MotionLength \times PoseSize$. The memory allocated for animation data must fall under a GPU budget specified by the application. If this variable is denoted by $MemBudget$, then the number of clips that can be used with the previous methods is: $NumClips_{OLD} = \lfloor \frac{MemBudget}{ClipSize_{OLD}} \rfloor$.

By using our new approach, the GPU data that needs to be stored for each clip size is $ClipSize_{NEW} = M \times PoseSize$ (recall that M represents the configurable parameter for the maximum offset value that an agent can have in its shared animation stream). This means that the number of clips that can be stored now relative to the old method is: $NumClips_{NEW} = \lfloor \frac{MotionLength}{M} \times NumClips_{OLD} \rfloor$. The left term is always in $(0,1]$ since the maximum offset cannot exceed the number of frames in the animation.

Figure 3: Image showing a side of a stadium in a football match

Analyzing this in the context of our application, where a skeleton with 82 bones was used and with the help of the techniques presented in [Ari06], [Hij00], [Sat05], the resulted $PoseSize$ was reduced from $3.2KB$ to $0.8KB$. $MotionLength$ was 300 since the average clip length was 10 seconds, and the sampling rate 30 frames per seconds (fps). In these conditions, the $ClipSize_{OLD} = 246KB$. With a GPU memory budget of $520MB$, if the previous version of application supported only 216 clips, by using a particular maximum offset value M of 60 frames, resulted in supporting up to 1080 different animation clips at the same time. The value used for M was high enough to let the 65000 agents in the crowd look like they perform different animations with only 256 animation streams, and a maximum offset (M) of 60 frames (the number of bones used was 82). The quality of the animations (i.e. if agents animations look different from each other) was evaluated by a quality assurance team who also did some tuning over the variables such that we get good enough results without stressing performance.

In terms of performance, the only theoretical problem could be the bandwidth between CPU and GPU, to keep the pose buffers in sync for the animations driven from the CPU side. However, with modern GPUs and computer architectures, a theoretical bandwidth of 32 GB/s (as specified by PCIe 3.0 standard) between CPU and GPU, would allow our application to send data for more than 100.000 animations on each frame considering our current setup and the other resources competing for the same bandwidth. The techniques described are scalable and can be used on any GPU as long as it satisfies the memory requirements and computational power (shader units) given by the configuration parameters and application's budgets.

Regarding to reactions, consider the case when the ball hits the crowd (i.e one or more agents). In this case, the agents which are effected would blend between their current animation and a hit by ball animation by using a transition slot. The target blend animation could be either a pooled one (hit by ball animation being shared by multiple agents) or a unique one (specific to one agent, client application having custom for the agent using the animation).

6 CONCLUSION

This paper presented some techniques for increasing the diversity and usability of animation systems in applications that use crowds of agents. To increase the diversity, this work changed the strategies used in previous approaches by not storing all the animations data on GPU memory, and by creating a data flow between CPU and GPU. Having animation controllers that can be evaluated using this strategy provides several advantages for the client system in terms of usability: the client is now able to send events back and forth between a simulation layer (e.g. a decision-making system) and an animation system. The techniques presented are used for skeletal animations that target a high level of details and can be combined with existing rendering and animation techniques targeted for a lower level of details. Then, the paper describes a blending technique that is able to perform an efficient transition between agents' animations, considering the sharing strategies specific to crowd systems. Also, a way to parallelize the controllers' evaluation both on CPU and GPU side was sketched. The crowd source code is currently inside a commercial package that we plan to decouple and make it open source for future research. We also want to invest more time in improving the parallelism of the computations and on the AI side of the agents since now we support better animations and instant reaction events.

7 REFERENCES

[Ari06] Arikan, O. Compression of motion capture databases, In Proceedings of SIGGRAPH, ACM, pp. 890-897, 2006.

[Bea14] Beacco A, Pelechano N, CAVAST: The Crowd Animation, Visalisation, and Simulation Testbed, Proceedings of Spanish Computer Graphics Conference, CEIG, pp: 1-10, 2014.

[Bea16] Beacco A, Pelechano N, Andujar C, A Survey of Real-Time Crowd Rendering, Computer Graphics Forum 35(8), pp: 32-50, 2016.

[Bra00] Brand M, Hertzmann A. Style machines, Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co, pp: 183-192, 2000.

[Greg14] Gregory, J. Game Engine Architecture, Second edition, Chapter 11: Animation Systems, CRC Press, pp. 543-647, 2014.

[Hij00] Hijiri T., Nishitani K., Cornish T., Naka T., Asahara S. A spatial hierarchical compression method for 3d streaming animation. Proceedings of Web3D-VRML ACM, pp. 95-101, 2000.

[Kav07] Kavan L., Collins S., Zara J., O'Sullivan C. Skinnig with dual quaternion. In Proceedings of the Symposium on Interactive 3D Graphics (SI3D), pp 39-46, 2007.

[Kov02] Kovar Lucas, Gleicher Michael, Pighin F. Motion graphs, ACM Transactions on Graphics (TOG)., 21(3), pp: 473-482, 2002.

[Kov03] Kovar Lucas and Gleicher Michael. Flexible Automatic Motion Blending with Registration Curves. In Proceedings of ACM SIGGRAPH, Eurographics Symposium on Computer Animation, pp. 214-224, 2003.

[Kim14] Kim J, Seol Y, Kwon T, Lee J, Interactive manipulation of large-scale crowd animation, ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014, 33(4), 2014.

[Mag88] Magnenat-Thalmann N, Laperrire R, Thalmann D, Montreal U. D. Joint-dependent local deformations for hand animation and object grasping, Proceedings on Graphics interface'88, pp. 26-33, 1988.

[Mil07] Millan E, Isaac Rudomin. Impostors, pseudo-instancing and image maps for GPU crowd rendering. Proceedings of the The International Journal of Virtual Reality, Volume 6, Issue 1, pp. 35-44, 2007.

[Ngu07] Nguyen H. GPU Gems 3, Chapter 2: Animated Crowd Rendering. Addison-Wesley, pp. 39-52, 2007.

[Poi09] Poirier M., Paquette E. Rig retargeting for 3d

animation. In Proceedings of the Graphics Interface Conference ACM Press, pp. 103-110, 2009.

[Rud05] Rudomin I, Millan E, Hernandez Z. Fragment shaders for agent animation using finite state machines.Simulation Modelling Practice and Theory, Volume 13, Issue 8 (November), Elsevier, pp. 741-751, 2005.

[Ryd05] Ryder G, and Day A. M, Survey of Real-Time Rendering Techniques for Crowds, Computer Graphics forum 24(2), Wiley, pp: 203-215, 2005.

[Sat05] Sattler M., Sarlette R., Klein R. Simple and efficient compression of animation sequences. In Proceedings of Eurographics Symposium on Computer Animation, pp. 209-217, 2005.

[Sho08] Shopf Jeremy, Joshua Barczak, Christopher Oat, Natalya Tatarchuk , March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU, Proceeding of ACM SIGGRAPH, pp 52-101, 2008.

[Tec00] Tecchia F, Chrysanthou Y. Real-time rendering of densely populated urban environments. In Proceedings of the Eurographics Workshop on Rendering Techniques, Springer, pp. 83-88, 2000.

[Uni16] Unity engine manual - animation section https://docs.unity3d.com/Manual/AnimationSection.html

[Wit95] Witkin A, Popovic Z. Motion warping, Proceedings of the 22nd annual conference on Computer graphics and interactive techniques. ACM, pp. 105-108, 1995.

[Pel16] Pelechano N. and Allbecky J. M., In Proceedings of IEEE Virtual Humans and Crowds for Immersive Environments (VHCIE), pp. 17-21, 2016.