

Animation system: Performance and Concurrency

Yordan Gyurchev
yordan@gyurchev.com

Introduction.....	3
1. Memory and performance.....	3
1.1 Memory usage in multiprocessor environment.....	3
1.2 Types of memory acces	4
1.3 Relocatable data structures	4
2. Animation system	4
2.1 System Structure	4
2.1.1 Levels of abstraction.....	4
2.1.2 High level.....	5
2.2 Low-level System Implementation.....	5
2.2.1 Animation state	5
2.2.2 Animation player	6
2.2.3 Updating animation players	8
2.2.4 Animation blenders & mixers.....	8
2.2.5 Updating animation blenders	10
2.2.6 Other animation entities (tasks)	10
2.2.7 Management: Tasks and Resources	11
3 Practical results and application.....	11
3.1 Next-Gen: Power the Instancing.....	11
3.2 Load balancing.....	11
3.2 Level of detail updates	12

Introduction

We will describe the approach we take (and the structures involved) to make our animation system achieve better performance on the PS2 platform. Furthermore the approach described contains exploitable concurrency that can be utilized on next-gen platforms.

First we are going to cover the memory performance constraints behind designing the animation system and then we are going to move onto the actual animation system, discussing the implementation details.

This document is NOT a manual how to compress/decompress animations or how to build your animation tool-chain. It will NOT be discussing quaternions, splines or any maths to do with animation. Nor it will discuss skinning or any rendering techniques.

Abbreviations used:

PU – Processing Unit. One of these things a multiprocessor machine has: processors, cores, etc.

1. Memory and performance

For some time now memory has been the bottleneck in performance critical applications. Exploiting concurrency is the next step toward better performance. The memory resource, however, is shared¹. This emphasizes even more the importance of using memory friendly access patterns.

1.1 Memory usage in multiprocessor environment

Usually PUs have caches that are aimed at reducing bus contention and latency times. Caches are automatic although PUs usually expose instructions for prefetching and synching data. For obvious reasons random memory access is not the best practice when it comes to caches. The only way to get most of PU cache is to improve the memory access patterns used in an algorithm.

In certain implementations PUs have dedicated local storage (memory) that can be explicitly used by the programmer. It has low access times. Careful usage of this local memory not only speeds up the task running on the respective PU but also other tasks as it generates less bus activity.

In PS2 we have CPU local memory (called in some references "scratch pad"). In some next-gen platforms there will be multiple processing units with local memory.

So to maximize performance in our animation system we need to follow two main rules:

- Aim for structures that support better locality without compromising abstraction and functionality

¹ We assume shared memory architecture as most game consoles use it.

- Utilize the local PU memory (where available) as much as possible and minimize main memory access (bus activity).

1.2 Types of memory access

Usually when it comes to memory access patterns they come in two types: **read-only** and **read-write**. Read-only memory can be left in the shared memory but ultimately if there are many reads it would be best placed in local memory so it doesn't burden the bus. Read-write memory is even heavier because it also requires management if more than one processing units are going to use it.

1.3 Relocatable data structures²

We make heavy use of what we call a header-data structure in order to keep our data structures flexible while achieving locality and task parallelism with low memory penalties. In short a header-data data structure is a solid block of memory that contains a small “offsets” map (the header) of the actual data in the block. You can move the whole block in memory (to and from local memory) and still support complicated data structures and dependencies between them.

2. Animation system

The animation system uses read-only resources called here “animation streams” to playback animations and update skeletons. The skeletons are then used to render animated meshes: characters, cameras, animated backgrounds etc.

A good animation system provides not only the ability to playback animations but also ability to blend animations together. Blending is used to make smooth transitions between animations and on more interactive level to combine multiple animations into one final animation according to some input parameters.

2.1 System Structure

In our animation system we base everything around a structure called **animation state**. The animation state represents the skeleton pose in an arbitrary point in time. Everything the animation system does results into generating or updating an animation state. We can then create more complex structures based around the state but ultimately the result of any operation is going to be stored in a state.

2.1.1 Levels of abstraction

The low level is where animations are played, blended, mixed etc. The low level part of the animation system needs to be built for performance.

² This section is now a separate document. You can find it at:
<http://www.gyurchev.com/docs/relocate.pdf>

The high level is the façade that the system exposes to the user of the animation system. This level supports high level of abstraction and is meant to be easy and flexible to use leaving the performance critical parts to the low level systems.

2.1.2 High level

A quick list of features that high-level interface provides:

- **Animation Agent** - If the user wants to play animations on a skeleton she needs to create an animation agent. The animation agent handles requests and commands to the animation system for that skeleton. An agent can contain multiple animation channels and mix/blend between them.
- **Animation Channel** – a channel produces ultimately an animation state. It can be a simple animation channel – plays animations or a complex blended channel blending multiple animations at a time. Animation channel is an abstract interface and can be specialized through derivation. Of course the less animation channels there are on a particular agent the faster it will be. Channel can also store parameters for complex behaviours like blend parameters, facing direction, etc.
- **Animation Post Processing** – too often the final animation pose needs to be tweaked after it is calculated. This is achieved by adding one (or none) or more post processing behaviours. Animation noise will be a good example of a post processing behaviour.

2.2 Low-level System Implementation

2.2.1 Animation state

The animation state itself consists of a header and an array of bone transforms (position, rotation)³. The header contains:

- System information: type of animation state, transforms offset (from the start of the structure), total size of the structure
- Animated entity information: skeleton bone count, delta time
- Velocity info – to move the animated entity (should we require to)
- Absolute root node information – valid for cut scenes and pre-canned animated sequences
- Priority values

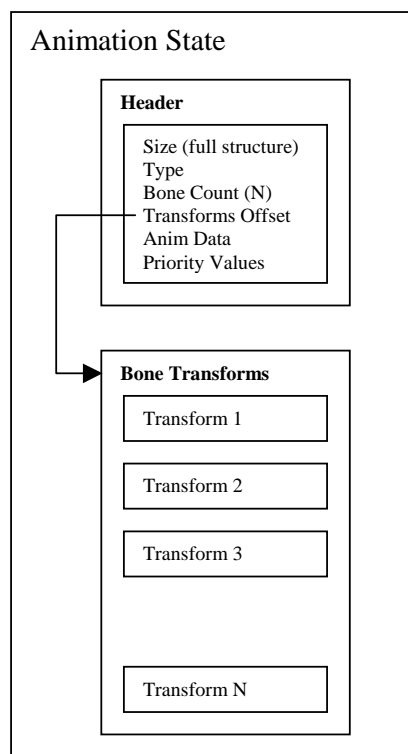
It is important to point out that the animation state is used as a base for other more complex structures. When the header is extended to contain other information (in the case of “derived” structures) the transforms offset points to the start of the transforms array. Using **total size** and **transforms offset** the animation state can be moved around in memory and manipulated without knowing the exact type of the structure and its use.

³ We don't use scale but it can easily be added to the transform structure

Here are the steps needed to use an animation state (or an animation state derived entity) with PU local memory:

1. Move **total size** bytes from the start of the animation state to local memory
2. Read/Update the animation state transforms (can be obtained through **transforms offset**) – the point of the whole exercise – fast access
3. Copy (if updated) the animation state back to its original location

Priority values are used to calculate task prioritization and constraints in a concurrent environment. More on that later.



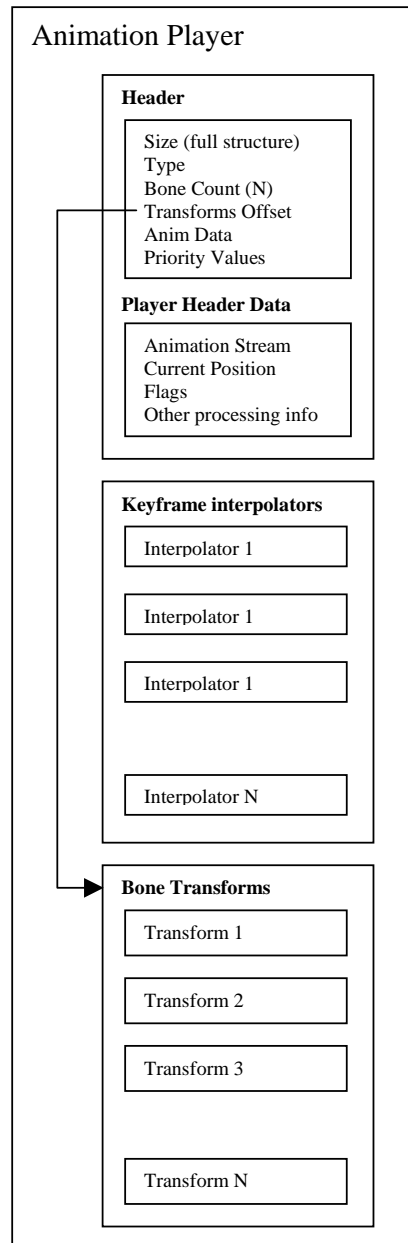
2.2.2 Animation player

The animation player is used to play an animation as the time advances by parsing an animation stream and updating an animation state.

Most often an animation stream contains only **key-frames**. Meaning that the animation player needs to fill the gaps in time, interpolating between the key-frames. In fact an animation player spends the majority of its processing time interpolating. In

order to do so the animation player will need some kind of help structures to keep track of control⁴ key-frames so it can interpolate between them.

So, an animation player will contain an animation state, a pointer and a state of the animation stream and an array of interpolation structures (one for each bone).



⁴ Depending on the compression/interpolation scheme there can be two (linear) or more key-frames used as control ones in the interpolation process. In our particular system we even split them into rotation and position key-frames in order to maximize compression.

Information that goes in the header is:

- Animation state header structure
- Pointer to the animation stream
- Current position (time) in the animation stream
- Flags for looping, reached end, etc
- Data key processing information (animators put specific info key-frames in the animation for game play purposes)

As you can see from the picture we have added a lot more information into the header but we can still treat this animation player structure as a simple animation state as the transform offset still points to the array of bone transforms.

2.2.3 Updating animation players

The update step of an animation player entity falls into the category of the “embarrassingly parallel” tasks. This means that animation players can be updated in any order and in parallel, as they have no internal dependency constraints. They use one read-only memory resource (the animation stream) and the delta time elapsed since the structure was last updated (subject to update before animation system is updated).

This property of the animation player update tasks allows us to schedule them each to a separate processing unit. So updating the animation player list falls into so called “task parallelism” pattern:

For each animation player

- **Schedule for execution on a free PU**

A word of caution: although each task (player) is running on a separate PU and all read-write data structures are in local memory there are resources that are still in the shared main memory – the animation streams. In the majority of the cases the animation stream is used to fetch new key frames. However if the delta times are large or/and key-frames densely populate the timeline it is possible that using this read-only resource will become a performance problem⁵.

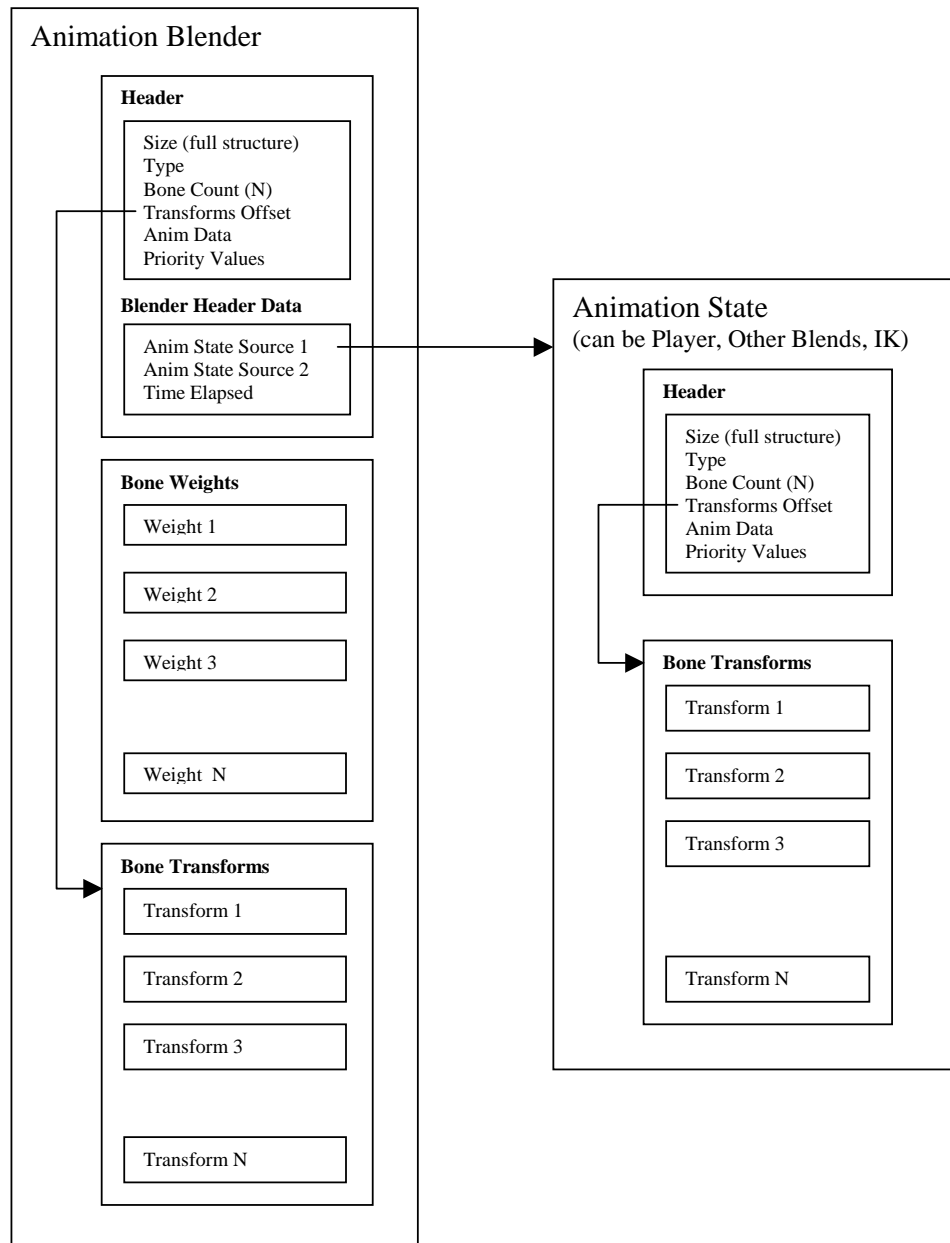
2.2.4 Animation blenders & mixers

Going back to the animation state we want to introduce a different type of entity - the animation blender. An animation blender can blend one or more animation states into one result animation state. An important quality of the animation blender is that it uses input animation states regardless what these animation states are result from. The sources can be animation players or other blends.

⁵ Shared memory is usually cached through a data cache and the animation stream is very sequential in essence so some pre-fetch hints to the PU might improve this without further trouble.

Animation blenders come in various types. Most often we use the ease in/out blender to smoothly blend in a new animation on the same animation channel. This is the one we are going to take a closer look at.

As with the animation player we start with our animation state structure. We need to add some parameters to the header such as: pointers to the two source states, total time of the blend, elapsed time in the blend, pointer to array of bone weights⁶.



⁶ In order to achieve full control over the blending process the user can supply per bone weight that is used to multiply the global fraction of blending between the two states. This has no application in ease in/out blending but in more complicated cases where overlay animations are played.

So our blender header is going to look like this:

- Animation state structure
- Source animation state pointers (these can be players, other blenders, etc)
- Time information: total, elapsed
- Pointer to array of weights

2.2.5 Updating animation blenders

There is a large difference between the player list and the blender list updates. Blenders cannot be updated in a random order as some of them might be sources for other blenders and need to be updated first. Another constraint is that players need to be updated before we begin updating the blenders as players are usually the sources of blends.

This is where we make use of the priority values in the animation state.

Lets illustrate the case with ease in/out blenders. If an ease in/out blender is to blend two states its priority value would be the lowest of the source priority values minus one thus making sure that any source states have been calculated before starting the blend. An animation player would have the highest priority value by default.

So when it comes to updating the blender list we need to sort it by priority and then for each priority (going down) that is in the list update all blenders that match that priority. Important feature is that once we schedule all blenders for a given priority we need to wait for all of them to complete in order to be sure that our constraints are satisfied.

For every priority (dec)

- **For every blender in that priority**
 - Schedule for execution on a free PU
- **Wait for all PUs to finish before moving to next priority**

All other issues are similar to the player update call handling. The only difference is that when scheduling an animation blend we need to upload all source animation states into local PU memory as well. They are going to be subject to heavy access after all. However as they are strictly read-only we don't need to return them back to normal memory.

2.2.6 Other animation entities (tasks)

Following the formula we can derive various animation tasks that can result in animation states. Examples could include IK behaviours, rag-dolls, etc.

2.2.7 Management: Tasks and Resources

All low-level animation entities (players, blenders, etc) can be completely managed internally in the animation system. This allows for:

- resource/memory management – all structures are solid blocks of memory - roughly the same size (depending on bone counts)
- streamlined processing – they can be organized in lists by type allowing for improved instruction cache performance.
- load balancing – manager can decide how is best to allocate tasks to processors

3 Practical results and application

We have a working implementation of the described system on the PS2. The available local PU memory there (scratchpad) is 16Kb. This imposes limitation of the number of bones (transforms) a skeleton can have. For example when updating animation players we double buffer the local memory⁷ effectively decreasing it to 8Kb. This allows for only 62 bones to be stored in local memory plus the relevant headers (numbers are dependent on the size of our specific data structures).

We have tested it with 40 animated skeletons (40 bones each) playing different animations at the same time. This takes 6% of one frame (at 60Hz) CPU time. Performance analyzer shows that main data cache misses happen when the animation streams are heavily accessed. There are hardly any instruction cache misses as all players and blenders are updated in lists.

3.1 Next-Gen: Power the Instancing

Instancing is a method for rendering vast amount of objects (in this case characters) on screen. Although the mesh is the same - the position, orientation and other properties like skeleton pose can be different. So if modern hardware allows us to render a large amount of meshes we have to be able to produce a large amount of animated skeleton poses.

3.2 Load balancing

Treating low-level animation tasks in the way described allows us to schedule their execution on multiple processing units dynamically based on availability of these units. It also allows the animation system to fit in as a piece of larger load-balancing system that controls multiple parts of the game.

Dynamic scheduling is usually associated with problems arising from large difference in tasks sizes and execution time. Never treat tasks as equal units. Usually (with small exceptions like IK) task execution time is proportional to the bone (transform) count.

⁷ While the CPU processes one animation player the DMA controller moves the next/previous one in/out of memory.

3.2 Level of detail updates

The low-level management can decide to skip certain updates based on update priority accumulating the time for the next update. This way the CPU load is reduced at expense of smoothness of the animation. Nobody likes animation artifacts so if implemented this system needs careful management of level of detail metrics to get maximum performance with minimum animation artifacts.

Per-bone level of detail is also possible. This would however require additional information in our player structures to indicate bone priorities. An alternative method is to implicitly arrange the bones in “level of detail” order in the transforms array.

Summary

The described approach achieves high level of utilization of the available computational power by reducing the amount of time PUs spend in waiting for data to be read or written.

The solution is based on task parallelism but there is a limit of the effectiveness of this approach especially if tasks are too small. When tasks are too small the time spend (the serial term) to setup and finalize them cancels the performance gain of the concurrency.

References:

[1] Timothy Mattson, Beverly Sanders, Berna Massingill; “Patterns for Parallel Programming”

[2] Dominic Mallinson, Mark DeLoura; “CELL: A New Platform for Digital Entertainment”

<http://research.scea.com/research/html/CellGDC05/index.html>

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; “Elements of Reusable Object-Oriented Software”

[4] Hemal Bodasing; “C++ Optimization Strategies on PlayStation 2”
<http://www.ps2-prop.com>