LGE – Animation System powered by the GPU

Game Engine III

Jibran Syed

Introduction

Animation is an integral part of many game engines in order to present fluid, moving characters. The reality is that animation is complicated and involves a lot of data, even more data than just the model's mesh itself.

In addition, high end engines must be able to support many instances of animated characters running at high performance. The obstacle here is that animations involves a lot of data and are also computationally expensive.

The solution to these problems is to implement animations in such a way that to data can be pipelined into the GPU for maximum performance, but still be controlled by the CPU without having tons of data transferring between the two.

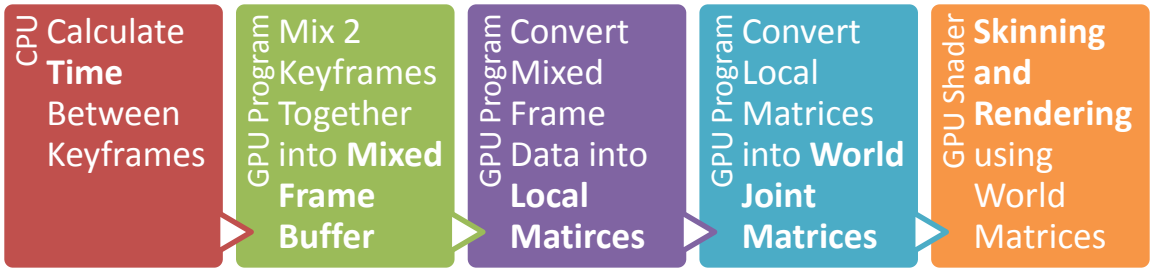
The LGE game engine now supports such a system that utilizes GPU procedures that handle data existing in the GPU that stays in the GPU. Conceptually these procedures would be called Compute Shaders, but LGE doesn't actually use OpenGL's compute shader feature. It instead uses special vertex shader procedures that have the OpenGL Rasterizer disabled, and using special buffers called Transform Feedback Buffers.

The idea is that animation data is stored into multiple buffers on the GPU (utilizing OpenGL's VBOs) that can be bound as array buffers, then sent into these "compute" shaders for processing. Both the procedure and the data would stay in the GPU, but is invoked by the CPU in order to control the pipeline. The result of the GPU procedure is written onto another buffer that can be used as input in another GPU program.

The whole animation pipeline begins in the CPU but mostly performs in the GPU. First, animation data is loaded at the beginning of the game. Then when animation needs to be calculated, the animation frame time is determined before being sent into the GPU in order to mix two keyframes.

The mixed frame buffer is converted into an array of local transform matrices relative to the root joint. This buffer of local matrices is converted into world transform matrices using hierarchy data to determine the order of computation (which is important). This world matrices buffer is sent into the skinning shader to morph the mesh then draw it.

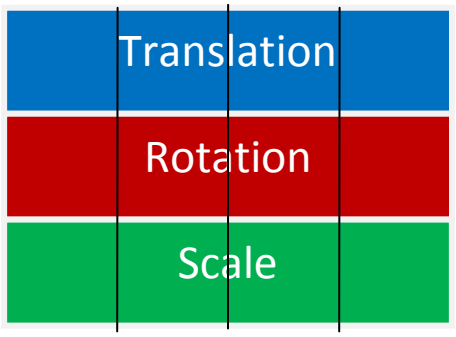
The following diagram describes the animation data pipeline visually:



Animation Set and Animation Controller

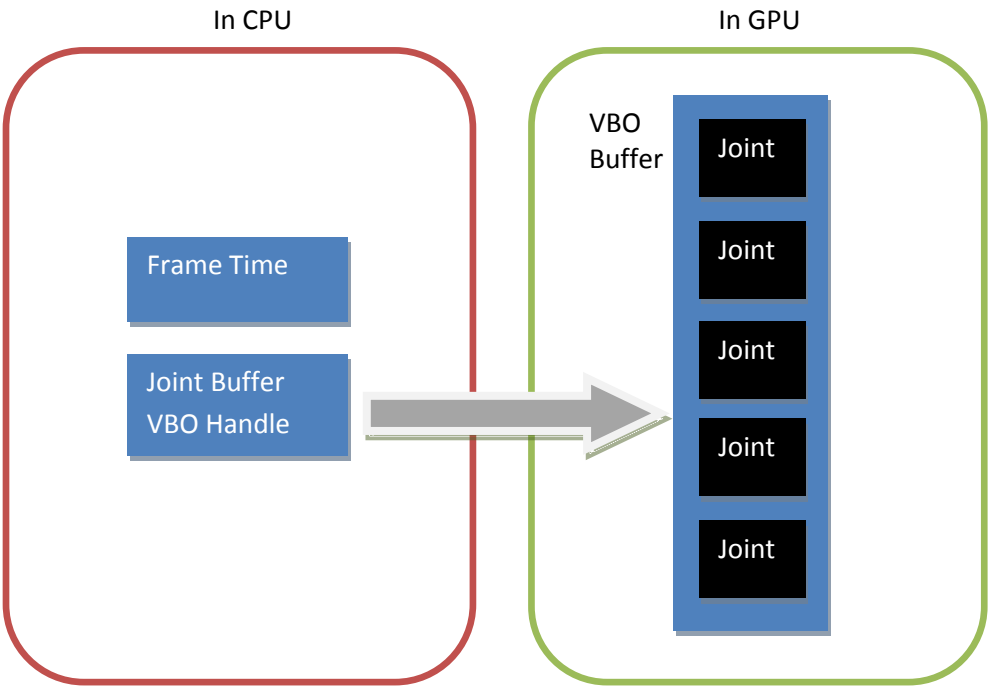
In LGE, there are two different classes to handle animation data: one for data that doesn't change and the other for data that does. These would be AnimationSet and AnimationController, respectively.

AnimationSet holds animation data that doesn't change. This data includes joint hierarchy, inverse bind pose matrices, and animation clips. A clip represents a single playing animation, such as waking or punching, for example. Clips compose of multiple keyframes, which represent the current transform of all the joints at the time of that keyframe. A joint transform is composed of a translation, rotation, and scale (all 4-element vectors). The following diagram depicts the layout of a single joint transform block:



The 4 divisions in each sector represent that each sector is a 4-element vector. Specifically, Translation and Scale are Vector4's, and Rotation is a Quaternion.

A keyframe in the GPU is essentially a buffer of joint transform blocks.
The following diagram shows a keyframe and its data between the CPU and GPU:

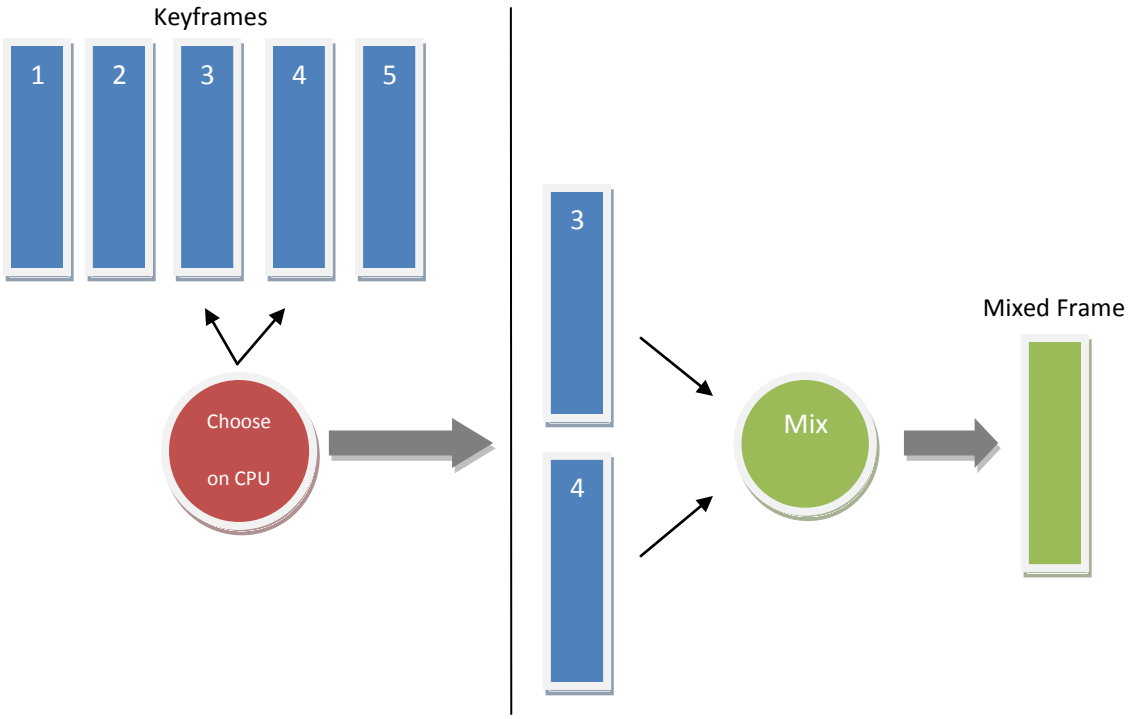


An AnimationController contains data that *does* change and is used to instance animation in LGE. What that means is that two animated objects can have their own AnimationControllers but refer to the same AnimationSet.

Data tracked in the AnimationController include the final frame mix (blend of two keyframes), matrix buffer (both local and world space), play states (like Play and Pause), and current frame time.

Keyframe Mixing

When playing a clip, the AnimationController on the CPU calculates the current frame time, and uses this normalized time to blend between two keyframes. Typically, the frame time will fall between two keyframe timings. These two keyframes then get blended on the GPU making a mixed frame in the GPU. The following diagram demonstrates this:



Converting Joint Data to Local Matrices

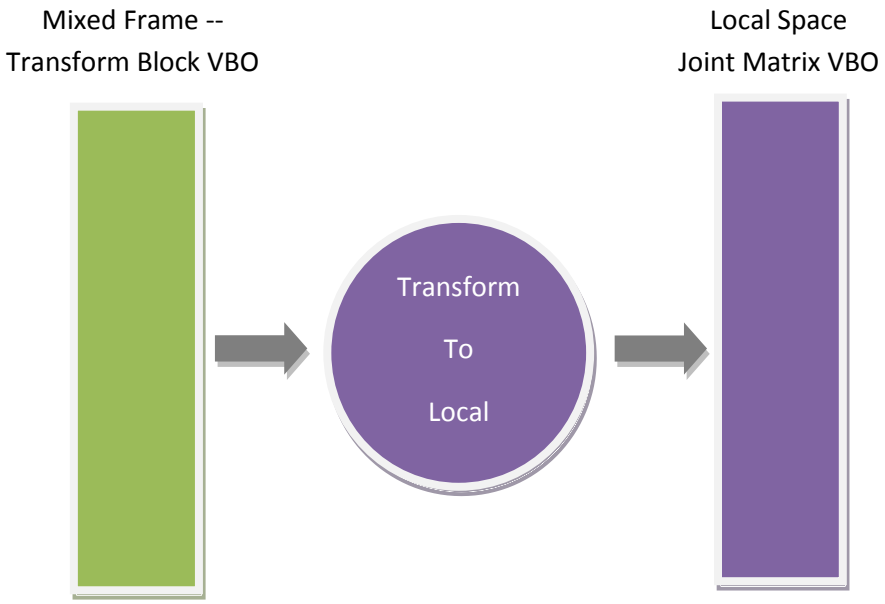
After calculating the mixed frame, the joint transforms in that frame are converted into matrices in local space (local relative to the root joint) using a GPU procedure. Since the transform blocks and matrices are both in local space, the conversion from transform block to matrix is as simple as calculating the model matrix:

$$L = SRT$$

Where L is the local matrix, S is the scale matrix derived from the transform's scale data, R is the rotation matrix or quaternion derived from the transform's rotation data, and T is the translation matrix derived from the transform's translation data. All matrices are assumed to be row-major.

In the GPU, the procedure processes the buffer of transforms and converts to the buffer of matrices quickly using the GPU's parallelism.

The following diagram demonstrates the procedure of converting transform blocks into local matrices:



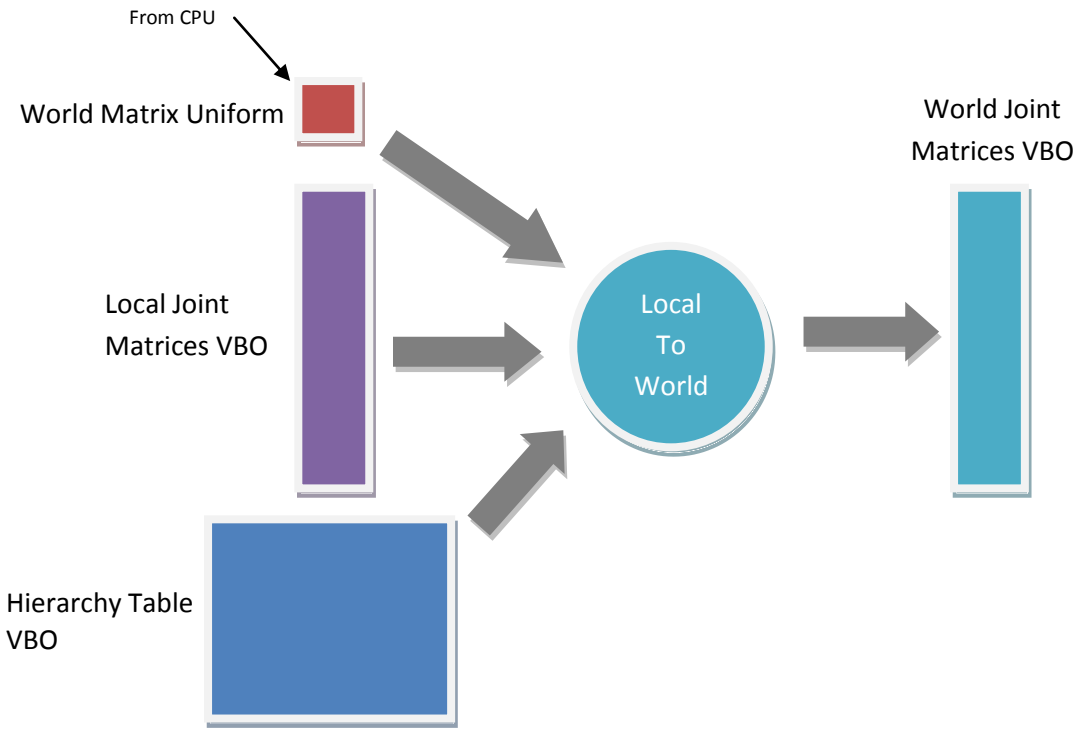
Converting Local Joint Matrices to World Matrices

Next, the local joint matrices need to be converted into world space joint matrices. This is more complicated than it looks, because a lot of hierarchy data is needed in order to determine what matrix to process first.

Matrices higher in the hierarchy need to be processed first so that matrices lower in the hierarchy can get the correct parent information. Hierarchy data is composed of a buffer of indices indicating what is the parent of the current index.

This hierarchy data is then used alongside the local matrix buffer and the object's world matrix in order to calculate the joint matrices in world space, which is required for skinned animation.

The following diagram demonstrates this procedure:



Mesh Skinning and Rendering

And finally, this world space joint buffer is sent into the rendering pipeline where skinning and drawing of an animated mesh are done.

Skinning is the process of morphing vertices on a mesh depending on how much a joint moves from the original positions of the vertices of the mesh. The original position of the mesh is known as the "bind pose" or the "T pose".

Skinning is a complicated calculation that requires a lot of data. The idea is that every vertex is influenced by a number of joints (we'll call *n*). These joints influence the vertex by a percentage or normalized amount called a *weight*. The more the weight, the more influence a joint has on a vertex. These joints and their weights morph the position of the vertex given the joint's transform in world space (which we calculated before) and the inverse matrix of the bind pose. These two matrices are needed to get the bone transform in "bone space" in order to do vertex transforms. This is explained in the following equation:

$$v_s(t) = \sum_{i=0}^{n-1} w_i \cdot \mathrm{J}_i(t) \cdot \mathrm{P}_i^{-1} \cdot v$$

Where v is the unskinned vertex, v_s is the skinned vertex, w is the weight, $\mathrm{J}(t)$ is the world joint matrix at the given animation time t, P^{-1} is the inverse bind pose matrix, i is the current joint

index of influence, and *n* is the number of joints influencing this vertex. The skinned vertex is them transformed with regular model-view-projection matrix to render the final animated mesh. The following diagram demonstrates the data flow: