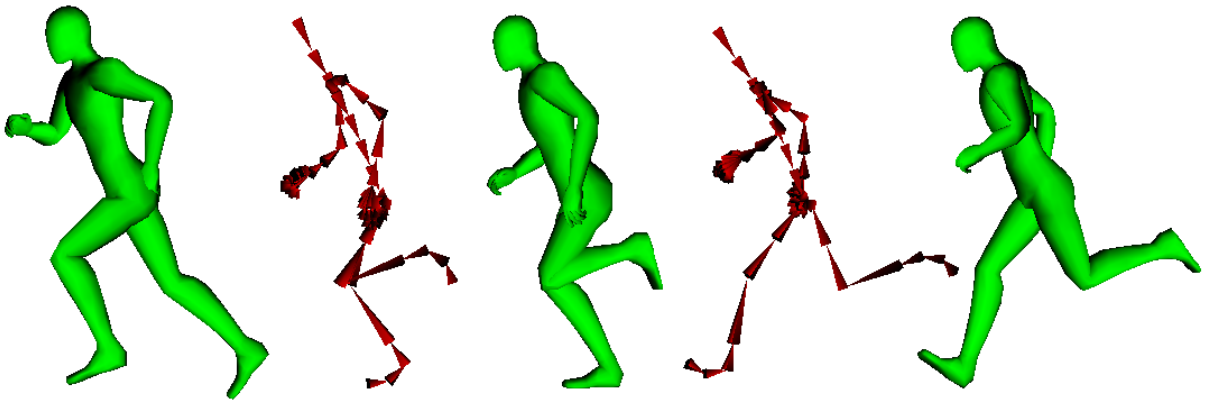

Animation System

Design Document

Syed Quadri

June 10, 2018

Changelist: 310614

Video Link: https://www.youtube.com/watch?v=_PmS38YOc1w

1. Introduction

Creating an animation system that integrates with a custom game engine is no easy feat. There are several features that are required to make custom models animate correctly. In this design document, I discuss how I approached in implementing a first attempt at an animation system. It should be noted, that optimization was not the goal of this system. The goal was to simply get a character model to animate accurately. At the end of this document, I go into details of what I could improve upon if I were to attempt to make another animation system or refactor the current system.

2. Setting up the Skeletal Pose

The first building block of an animation system is to create the skeleton pose of a model at a particular frame. We first need to extract relevant animation data from a model file(FBX file format). We modified the FBX converter provided by Autodesk to extract the hierarchy data and transform data for each bone at a certain frame of a clip. The extracted data would then be packaged into a custom file format (.SPU extension) that I could use inside my custom game engine. To do this, we created a standalone generic archiver (VODKA – Figure 1) that first takes raw binary files and converts them into Chunks. Another archiver (LIU – Figure 2) then packs all the Chunks together into one Package. Then in our game engine, we have an Extractor (eat) to filter one chunk from a package. This archiver allows for many dissimilar data types to be included into one file. We use this archiver in several other converters that I'll discuss in later sections.

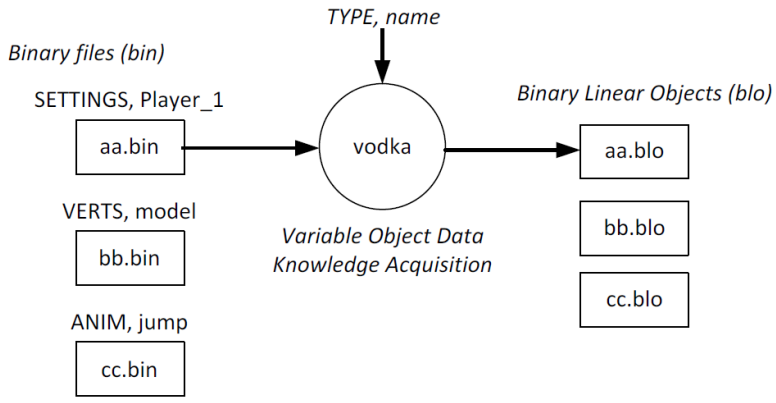

Figure 1 – VODKA

Within our game engine, inside the animation class, we have private method that reads in this SPU file and retrieves the Bone_Data and Skeleton_Data using the eat runtime function. We then proceed to create the skeleton hierarchy based on each bone's index and parent index using a PCS tree structure. The parent bone needs to be created before the child. Luckily, we sorted the bones in the converter based on a bone's parent index. The skeleton is attached to a root GameObjectRigid object, then the remaining bones are of type GameObjectAnimation. Each bone is created using a bone model, which is a simple pyramid model, but elongated. After the creation of the skeleton hierarchy we proceed to create the animation clips by storing each frame data into a Frame_Bucket object. The animation class then holds a pointer to the first bucket with pHeadBucket, which contains an array of all the bone in the hierarchy for the result pose. Then the next bucket will hold the actual frame and bone data for first

frame of the clip. Now that we have all the animation data imported and setup into the engine, we start bringing these animations to life.

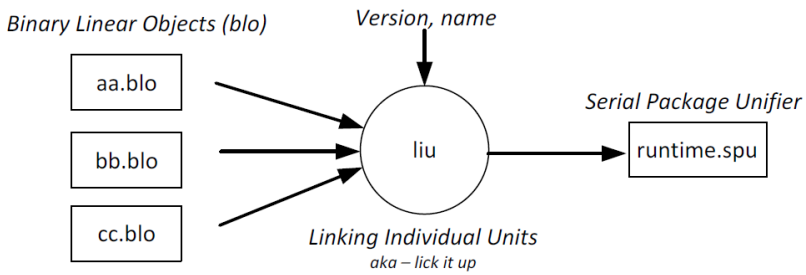

Figure 2 - LIU

3. Skinning – Vertex Blending

The skeleton of a model is often never seen in the final game. Usually a skin or mesh encapsulates the skeleton and mimics the same animation. We get the skin from the FBX model file, but without proper deformation techniques, our model will not be able to animate its skin. Vertex blending, or skinning is the process deforming the vertices of a skin, so that it can essentially move with the bone(s). To do this, we use the following equation:

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p} \quad \text{where} \quad \sum_{i=0}^{n-1} w_i = 1, \quad w_i \geq 0.$$

Where w_i is the weight of a particular bone i, $B_i(t)$ is the bone transform in world space at time t, M_i^{-1} is the inverse of the initial bone transform in world space (bind pose), p is a vertex of the model, and n is the number influential bones. The summation of the weights must be equal to 1 (weights must be normalized). Each vertex of a skin can be influenced by multiple bones. In our case, we limit the number of influential bones to the 4 heaviest weights of a particular vertex. As before, we modified the provided FBX converter by Autodesk to extract the weights and bone indices for each vertex of the model. We normalized the weights before committing the skin data to the archiver. The world matrices for each bone and the bind pose matrices were already extracted before when we created the skeletal pose using the Animation Converter. The vertices and triangles of a model were extracted and created in a similar fashion using the Model Converter tool, which was created for the Model Viewer in a previous project.

Inside the Skin class of our game engine, we used the eat function to extract the weights and bone indices from the packaged SPU file. We then create separate Vertex Buffer Objects (VBOs) for the vertices, triangle list, weights, and bone indices. These VBOs are then linked to the same Vertex Array Object (VAO) for the model. Then in our vertex shader we call on these attributes and use the equation above to define the position of a vertex. During an animation setup, we link the Skin to the Animation object. Please refer to figure 5 to view relationship in a UML diagram.

4. LERP Blending – Frozen Transition Cross-Fade

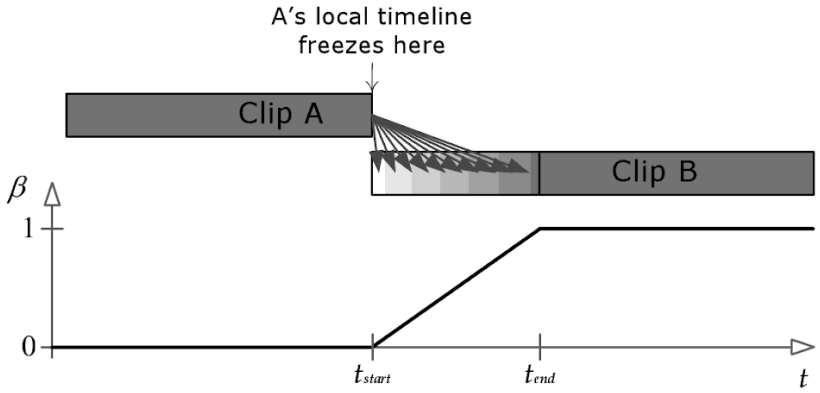

Figure 2 – Frozen Transition

For our character model to smoothly transition from one clip to the next, we need to apply some type of cross-fade. In our game engine, we utilize a frozen transition cross-fade, which stops the first clip from animating, while starting the second clip. Inside our Animation class, we can define t_{start} and t_{end}, which is initially set to 0.0f and 1.0f, respectively. This means that the transition from one clip to the next will last one second. During the Update() method, we first check if our character model is blending, which is set when the user presses the "C" key to switch to the next clip. If we are blending, then we stop processing the first clip's animation, and instead process the second clip's animation. Each clip has its own Time Controller, which we'll look at in more detail in section 7. A separate Time Controller is also utilized when animating the blend between two clips. It takes that Time Controller's current time value *t* to determine how far along within the transition we are. Using linear interpolation, we determine the LERP factor with the following equation:

$$t_s = \frac{t - t_{start}}{t_{end} - t_{start}}$$

Then we use that LERP factor when we send it as uniform to the BlendAnimations compute shader, which uses OpenGL's mix(vec4, vec4, float) function to linearly interpolate the translation and scale. We use a custom SLERP method to spherically interpolate the rotation. The translation, scale, and rotation are then used to compute the blended local transform. This is all done on the GPU, as well as processing the animation and computing the pose. We look at how we take advantage of the GPU next.

5. Utilizing the GPU

Modern GPUs are significantly stronger than CPU and are thus able to process heavy tasks such as rendering with ease. We take advantage of the GPU's power by computing all the animation details on the GPU. We do this by using compute shaders and storage shader buffer objects (SSBOs). We created 4 separate compute shaders: one for computing the local transforms for each bone as it transition from one frame to next of the same clip (animation.comp.glsl); another for converting the locals to world space, thus getting the world transforms of each bone (localToWorld.comp.glsl); another for orienting the bones of a skeleton, which entails determining the height of each bone in reference to its parent

(OrientBones.comp.glsl); and the last shader which computes the local transforms for each bone, as the skeleton blends between two separate clips (blendAnimations.comp.glsl).

The animation class holds a pointer to all the SSBOs, which are setup and bound initially and then later utilized throughout the render pipeline. There are a total of 12 SSBOs: transSSBO for the translations of the current clip data, which includes all the bone data for all the frames; scaleSSBO for the scales of the current clip data; rotSSBO for the rotations of the current clip data; scaleA_SSBO for the first clip's scale frame data, which includes all the data for each bone at a particular frame; rotA_SSBO for the first clip's rotation frame data; transA_SSBO for the first clip's translation frame data; scaleB_SSBO, rotB_SSBO, transB_SSBO are similar counter parts, but for the second clip's frame data; localSSBO for the local transforms of all the bones of the currently processed pose; worldSSBO for the world transforms of all the bones of the current pose; parentTableSSBO for holding each bone's parent's indices in order; and orientSSBO which holds the bone orientation transform which is used to update the individual bone GameObjectAnimation objects.

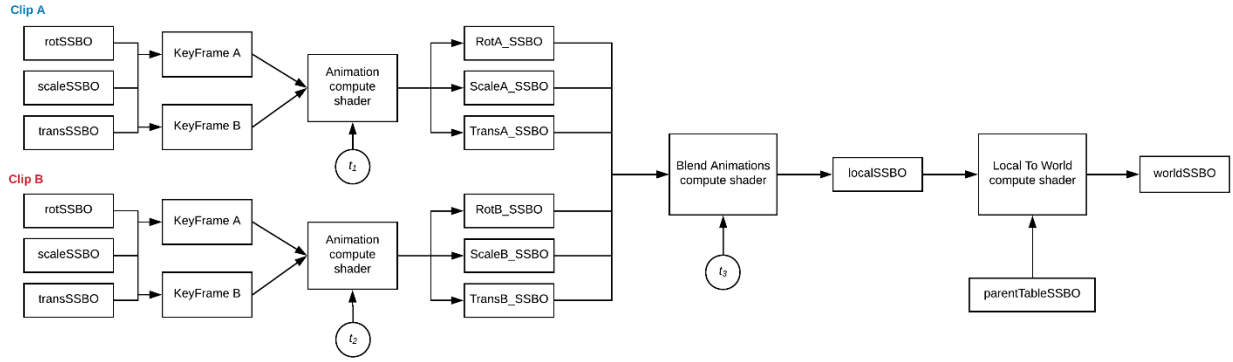

Figure 3 - Blending 2 animation clips

As an example, imagine we are trying to blend a humanoid model from it's current run animation to it's punch animation. First, we use the animation compute shader to determine the local transforms of all the bones for each clip, which is stored in respective SSBOs for clip A and clip B. As a side note, if we were not blending, we would also compute and store the local transform into the localSSBO. We then the blendAnimations compute shader and recall each clip's respective SSBOs to interpolate between the two clips and store the local transform in the localSSBO. Then we use the localToWorld compute shader, along with the localSSBO and parentTableSSBO as inputs to determine the world transforms of all the bones. This can be shown in figure 3. As an optional last step, we use the OrientBones compute shader with the worldSSBO and parentTableSSBO to orient the bones and store the transforms inside the orientSSBO, which is then called upon by the CPU, which updates the orientation of each bone game object. Moreover, mitigating heavy tasks to the GPU can lead to increased overall performance.

6. Dropped Keyframe Compression

Some animation clips can be flustered with frames that rarely differ from nearby frames. This can lead to wasted memory space and more time for processing these animations. An efficient way to alleviate these drawbacks is to compress the data, specifically using the dropped keyframe compression method (Xiao, Jun, et al). Using this method, we first define reference and candidate frames. We calculate the

interpolated bone angles (see equation below) for each frame between the reference and candidate frames.

$$\theta_i^{(k)} = \cos^{-1}\left(\frac{\vec{B}_i^{(k)} \cdot \vec{B}_i^{(center)}}{\left\|\vec{B}_i^{(k)}\right\| \left\|\vec{B}_i^{(center)}\right\|}\right)$$

We compare these interpolated values with the real values for that particular frame by computing the difference. If the difference is less than the threshold, than we continue to the next bone, and ultimately the next frame. However, if the difference is greater than the threshold, than we retrace back to the previous frame and add it to a list of compressed frames. We then reset our reference frame to point to this retraced previous frame and set our candidate frame to the next frame from the reference. We repeat the process until we reach the last frame, which we automatically add. This entire process can be displayed in figure 4. Again, we modified the FBX converter provided by Autodesk to reduce the number of frames. The data is archived and imported to the engine as before with the Skeleton_Data and Bone_Data when creating the skeletal hierarchy. The threshold ultimately must be fine-tuned for each animation to get the desired result. The converter that was created has a command option "-t" which takes in a floating number as the threshold value. Leaving this blank will simply return the result with a threshold value of 0.0f. There are other compression techniques out there that I was not able to implement. I discuss these in section 9 as improvements to be made to this system.

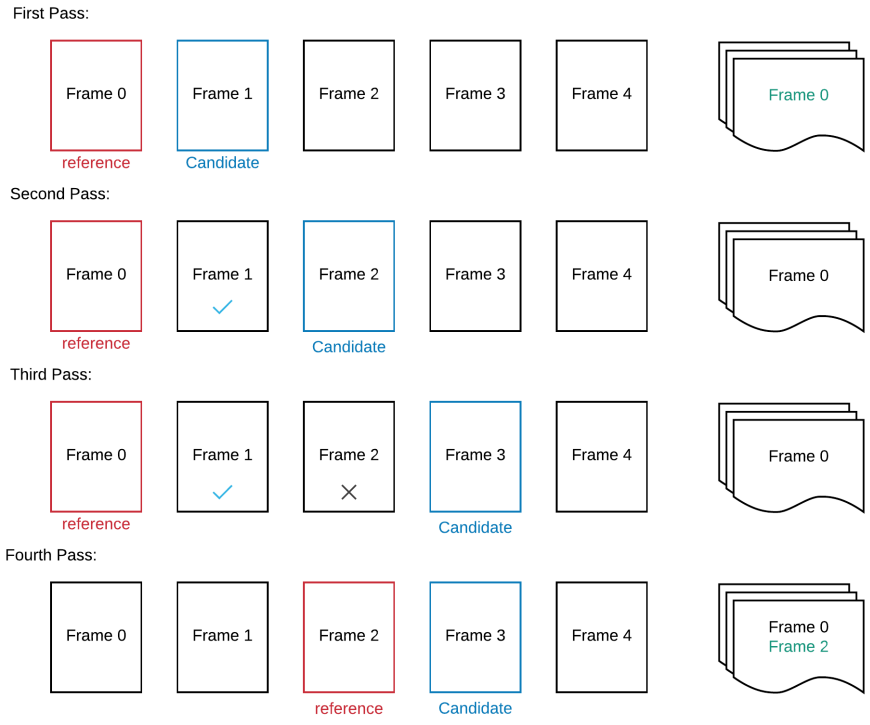

Figure 4 - Dropped Keyframe Compression

7. Time Controller

The final piece to the animation system is the animation controller. In my game engine, I denote similar functions to a Time Controller class. The Time Controller simple holds on to the current time. It manipulates this time to play, pause, and switch between animations. It can also play the animations in reverse and switch between fast/slow speeds. It loops each animation, so that once the current time is greater than the length of a clip, it resets it back. A Time Controller is required for each animation that is currently playing and is setup when creating the clips. The Time Controller has a linked list of the clips for each animation. As mentioned earlier, when blending between two animations, there are essentially two Time Controllers: one that is shared between clip A and clip B; and another for the blended animation. Refer to figure 6 for the UML class diagram of the Time Controller. The Time Controller acts as a debugging tool to view the animated model in detail. Likewise, it sets everything else in motion.

8. Conclusion

Overall, this animation system utilizes several features to make it function. We had to first create an archiver for packaging data into our game engine, which is utilized in multiple converters that we modified. We created a skeleton hierarchy and the clips for the animation. Using another converter, we extracted the relevant weights and bone indices for each vertex of a model. We then had to rely on vertex blending to deform our model's skin appropriately. We took advantage of the GPU's power and compute shaders to process the animation, including blending between different clips. Our compression converter utilized a method for extracting pertinent keyframes to reduce the overall number of keyframes of a clip. Our Time Controller glued everything together and manipulates the animation with easy user control. At the end, the animation system proposed does an effective job at rendering animations but could be improved upon in the future.

9. Future Improvement

One of the main takeaways from this paper, is that this was a first attempt at a functional animation system. However, there could have been several improvements if I went back and refactored the system architecture, making use of more design patterns for easier readability and performance. Compression was touched upon last, and I feel that there could have been more done. On a second attempt, I would separate the scale and translation for each bone for each frame, as this would remain constant. The only change would be the rotation. Thus, this would compress data by 2.5-fold. Another compression technique would be to bit mask the real value of the quaternion, as it can be deduced with the following equation:

$$d = \pm\sqrt{1 - a^2 - b^2 - c^2} \; where \; q = [a \; b \; c \; d]$$

However, the sign cannot be deduced. Thus, the bit mask trick is to save the sign to least significant bit of *c* value. This in turn compressed data by another 25%. All in all, I learned plenty from creating my first animation system, and a second attempt will only improve upon it.

10. UML Diagrams

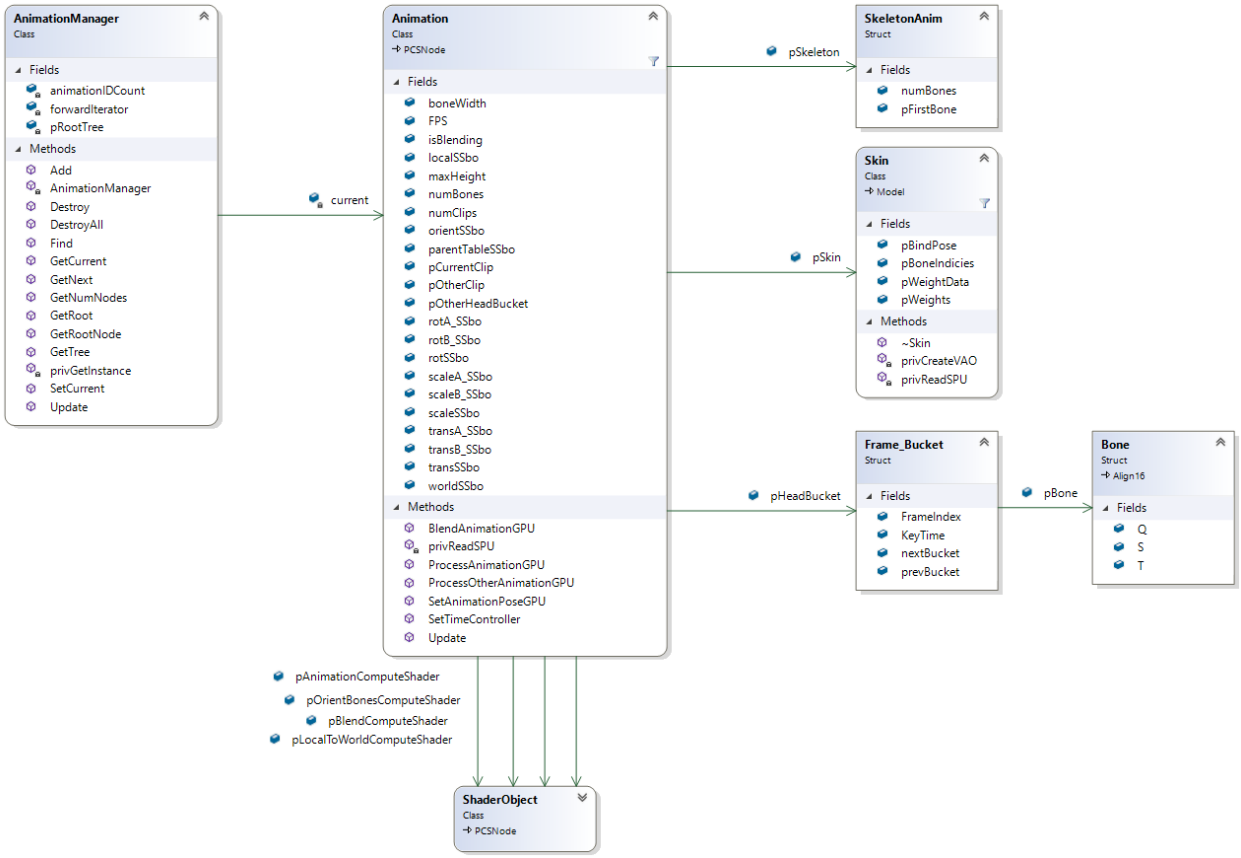

Figure 5 - Animation Class

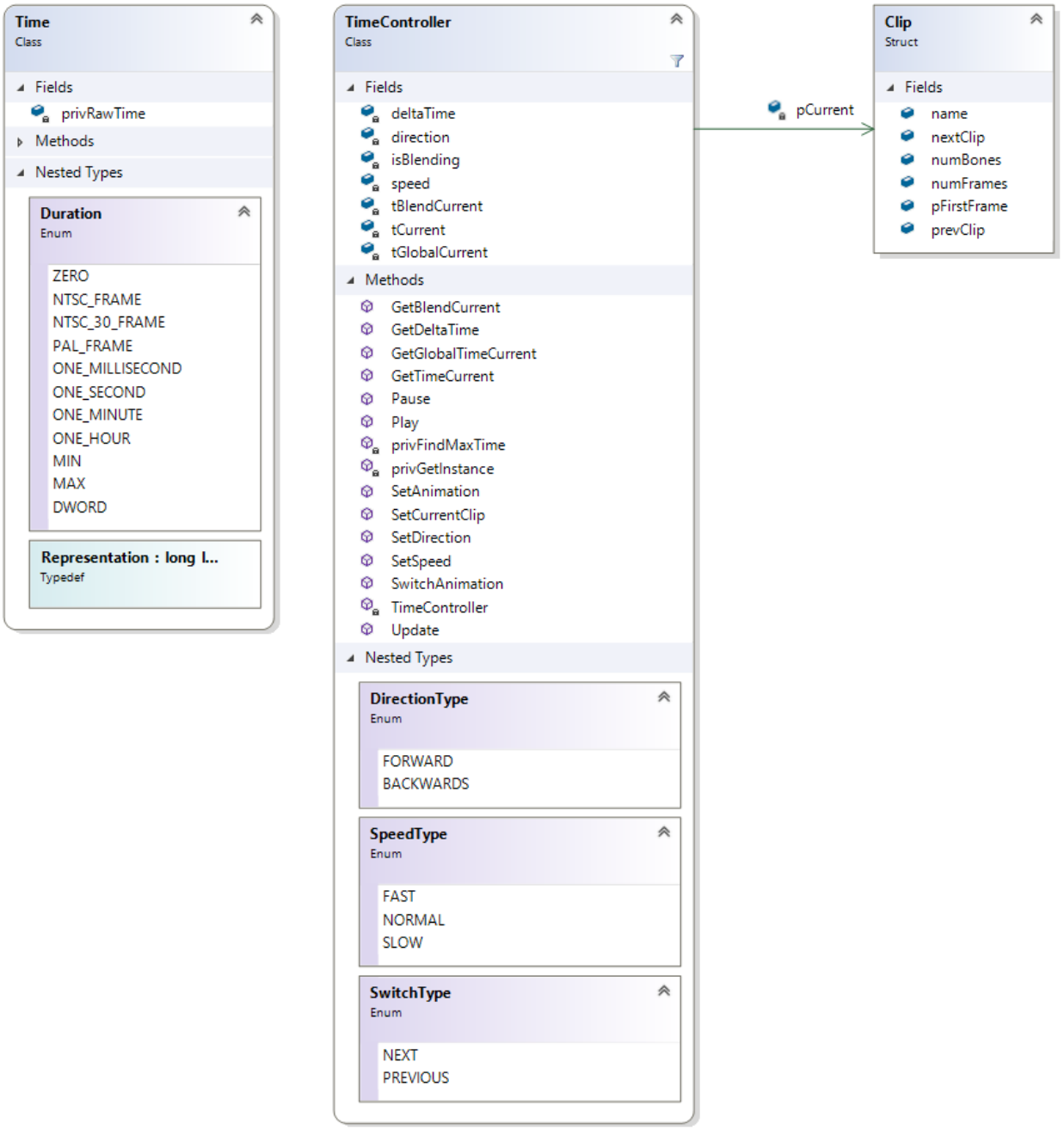

Figure 6 - Time Controller Class

Works Cited

Akenine-Moller, Tomas, et al. “Vertex Blending.” *Real-Time Rendering*, Third ed., Taylor & Francis Group, LLC, 2008, pp. 80–85.

Gregory, Jason. “Animation Systems.” *Game Engine Architecture*, Second ed., Taylor & Francis Group, LLC, 2014, pp. 543–646.

Xiao, Jun, et al. “An Efficient Keyframe Extraction from Motion Capture Data.” *Advances in Computer Graphics*, 2006, pp. 494–501., doi:10.1007/11784203_44.