Skeletal Animation in a Shading Atlas Streaming environment

Florian Komposch BSc*
Supervised by: Dipl.-Ing. Philip Voglreiter BSc†

Institute of Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

This paper describes a solution to integrate skeletal animation and animated hierarchical transformations into a high-quality Virtual Reality environment. As a base framework, we use the Shading Atlas Streaming environment. In contrast to widely available Virtual Reality systems, Shading Atlas Streaming decouples server side rendering and display tasks on the client. Rather than transmitting pre-computed images, Shading Atlas Streaming uses a combination of potentially visible geometry and shading information which it transmits in an asynchronous fashion.

Since the client actually renders the currently visible geometry and directly applies shading information from the atlas, potential network delay and bandwidth limitations may generate occlusion artifacts for animated scene content when using a straight-forward approach. Further, client display frame rates are much higher than the server update rates by design. For animations to work properly, we need to account for appropriate interpolation.

We introduce an animation system that is capable of dealing with the asynchronous behavior of Shading Atlas Streaming and provides smooth animations on the client, irregardless of the frame rate of the server update rate.

Due to the comparably low capabilities of modern Virtual Reality headsets, our approach must incur only a small performance footprint on the client. We approach this by linearization of animations and a completely GPU-based implementation of the skeletal animation system.

The system is designed to perform the main animation workload on the typically high performance server PC and leaves the client with only one additional interpolation task for frame rate upsampling.

Keywords: Rendering, Animation, Skeletal Animation, Virtual Reality, Shading Atlas Streaming, Vector Streaming, Vulkan

*florian.komposch@student.tugraz.at
†voglreiter@icg.tugraz.at

1 Introduction

Animated content is a staple feature in computer graphics applications. With the recent trend towards Virtual Reality (VR) in both research and industry, new concepts for satisfying the ever rising display capabilities were presented. With higher frame rates and resolutions and the trend leaning towards wireless display devices, simple image transmission [9] and client side interpolation become increasingly difficult due to both the computational demand on the Head Mounted Displays (HMD) and the bandwidth limitations.

New concepts, such as Shading Atlas Streaming (SAS) developed by Mueller et al. [6], attempt to alleviate these issues by decoupling content generation and client display. Traditional VR applications send a constant stream of pre-rendered images to the client, which then performs image-based upsampling tasks. In contrast, SAS computes a potentially visible set of geometry for a short time frame into the future, and also generates shading information for that visible set. The shading information is stored in an Atlas that is, alongside the visible set, transfered to the HMD. The HMD then rasterizes the potentially visible set and directly texture maps the Atlas information instead of performing computationally expensive shading operations. The authors show that this can drastically reduce the bandwidth since the server does not need to exhibit an update rate as high as with image based approaches.

However, this complicates the implementation of animation systems. Traditional image-based approaches to VR do not need any additional effort for animated objects, since all animations are implicitly interpolated along with the rest of the image information. For distributed environments, such as SAS, we present a system that integrates well with asynchronous frame rates, latency hiding and frame rate upsampling on the client.

1.1 Skeletal animation theory

Skeletal animation is a technique to intuitively deform and animate 3D models. It relies on a hierarchical transformation concept that links joints (or bones) with additional blend parameters for complex, yet smooth animations. Since skeletal animation is commonly used in game

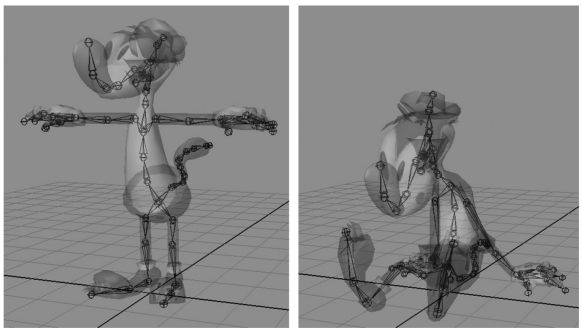

Figure 1: On the left, "T" pose and on the right a deformed skeleton, with skin applied depicted by Jason Gregory [3]

engines we summarize used concepts from Gregory et al. [3].

Skeletal animation is typically prepared in two stages. In the rigging stage, a hierarchical structure is generated. This structure closely resembles a skeleton consisting of bones, especially in natural objects such as human characters, and is named likewise. Each joint in this skeleton describes a relative transformation to its direct ancestor in the hierarchy and the related subspace.

It is necessary that the rig is designed to fit with the corresponding mesh in order to generate comprehensible animation outputs.

The second creation stage is the skinning process, which connects vertices of the mesh with bones of the rig and defines weights with which each bone influences a vertex during animation. Various techniques exist, including manual processing, least square distance approaches or comparably complex algorithms like James and Twigg [4] described for skeleton and skinning estimation in large scale scenes.

In theory, vertices of a mesh could have different bone counts, but are often limited to a fixed count for efficient GPU implementations.

A skin is applied to a skeleton in its bind pose, also referred to as "T" pose. The bind pose defines the state in which the model is not deformed by the initial skeleton. To ensure this behavior and further grant an efficient skeleton evaluation a *bind_offset* is generated. This offset is the inverse of the initial bone to world transformation. It is applied in the bone evaluation step to counter transformations which would be applied through the initial bone position.

Most animation engines implement an additional offset layer in bone space such that the initial bone position can differ from the bind pose. The bind pose is designed to ensure a simple binding procedure, but is usually not directly used in the animation process.

Figure 1 shows an example of a bind pose and an evaluated animated position.

Using the offset layer, a model can be bound to the same rig with various initial positions, for example in time-shifted animations or deformations that were not originally intended. This also allows animators to reuse skeletal animations without repeating the rigging process.

Another advantage is that different models could have the same animation with just separate skinning information, which is also a key element of the work James et al. [4] proposes.

Moreover, animation systems can provide many more additional features. They could support animation blending or may also include warping to different rigs as described by Gregory et al. [3]. While we do not directly support such features and the resulting combinatorial complexity, implementing them into our system is mostly straight forward.

We instead focus on a typical animation engine which comprises two stages. The update stage is usually decoupled from the rendering or presentation stage to allow asynchronicity of animation updates and rendering frames. In the update stage, the skeleton is evaluated with the corresponding timestamp via a recursive walk through the hierarchy.

In the render stage, the updated matrices are used in combination with the skinning and bone offset data to transform all animated vertices according to the updated skeleton.

1.2 The Shading Atlas Streaming (SAS) environment [6]

In modern virtual reality (VR) systems, it is common that a high-end pc is combined with tethered VR headsets. The PC acts as a server and streams pre-generated images to an HMD client. In light of recent development and the demand for wireless solutions, the main bottleneck in this approach is limited data transfer between the participants.

Therefore, a tradeoff between quality and latency must be solved depending on the application's purpose.

Some approaches use all-in-one VR headsets that do not suffer the transfer restrictions, but lack computational performance and visual quality.

As both of those technologies have drawbacks, SAS aims to deliver a solution that combines both concepts so that the resulting VR experience is enhanced. A precondition of this approach is to maintain high quality while limiting the transmission bit rate.

The SAS framework developed by Mueller et al. [6] serves as the baseline for our implementation. As previously mentioned, the server performs visibility and shading passes to prepare the scene information. Then the client rasterizes the scene again but utilizes visibility and shading information from the server. To achieve a low client GPU load, the server generates object-space shading information for each triangle potentially visible in the near future. To estimate the near future visible set, several future view points are predicted and the resulting ex-

actly visible sets from each view point are combined into a potentially visible set. This also incorporates animation information for each of these view points. The client receives the entire potentially visible set, which allows for small movement or rotation within the resulting view cell without requiring updated information.

Further, the server generates and transmits the shading information of all geometry in the potentially visible set. This is done as an object-space shading method utilizing the *Shading Atlas*. The Shading Atlas is divided into patches, which are groups of one to three triangles. This considers temporal coherence and can be efficiently encoded using standard H.264 encoding [11] for network transfers.

The client efficiently texture maps the atlas information on the geometry of the potentially visible set without invoking expensive fragment shader operations.

With this, the SAS approach delivers higher quality renderings in comparison with all-in-one VR solutions and at lower bandwidth requirements. Further, since the client rasterizes the potentially visible set locally, no occlusion artifacts are generated like in comparable image-based rendering (IBR) VR systems. Additionally, the client is not bound to the server frame rate and can perform frame-rate upsampling with the given data as long as the HMD's viewpoint changes are within the margin of the potentially visible set.

2 Related Work

This project combines interactive streamed VR content and skeleton animation. It is directly related to VR remote rendering and local animation systems.

Remote rendering is described by Shi et al. [12] as rendering 3D graphics on a server system and displaying the results on a client device. The decoupling adds an additional transmission layer which introduces limitations like bandwidth and latency. Those limitations are especially relevant in an interactive real-time environment like VR.

There are different approaches to challenge those limitations. For example, Image Based Rendering is widely used to approach the latency of high-quality image transfers by generating intermediate images by warping old ones with the latest viewport. To allow the warping with different views the image has to be in a super-resolution. Those approaches are designed for animated or interactive views but do not account for animated scenes.

A highly distributed approach is asynchronous time warping (ATW) described by Oculus [9]. It is not only used in streaming environments to hide latency like the Oculus Rift, but can also be used for frame rate upsampling like in the Oculus Quest if the render pipeline does not meet the expected frame rate.

There are also more complex strategies for real-time warping, which utilize proxy geometry like Mark et al. [5] describe in their work. They use a depth image of the last available frame to generate a simplified geometric proxy which can be rendered from a new viewpoint. The missing color information can be generated through the old image.

Oculus presented the technology Asynchronous Spacewarp 2.0 [10] building on the predecessor [8] which utilizes also an additional depth image for synthesizing. This technology also uses a Positional Timewarp which can provide six degrees of freedom for the view estimation instead of the predecessors three. With this combination, Oculus claims to reach good results even if the real frame rate drops below the halve of the desired one.

There are also systems, which utilizes Pixel Flow in combination with IBR to synthetic intermediate images, from Myszkowski et al. [7] to reduce render latency in high-quality walk-through animated sequences. Parts of this approach could also be used in a streamed animated scene as the concepts of movement, time and spatial coherence are the same.

Animation systems in general handle transformations in a scene. Typical animation systems support all kinds of animation techniques like Rigid Hierarchical Transformation, Per-Vertex Animation or Morph Targets. Skeletal animation is an additional common concept for animation, especially in character animation. As a reference, we used the animation engine design presented by Gregory et al. [3]. It uses keyframe animation applied on a hierarchical transformation to describe an animated skeleton. A corresponding 3D geometry is bound to that skeleton by a binding stage. In the update pass, the geometry will be transformed accordingly to the actual skeleton position. The animation system is further responsible to steer the animation when to start, stop, time warp or even blend it with other animations. The system itself is usually controlled by the actions of a user or can also act as a replay engine of recorded animations.

3 Animation system integration

This section handles the integration of an animation system into the SAS environment.

We describe the loading of animated objects and the general initialization of the animation system. We then proceed to describe the integration into the framework in three steps. The first is a simple forward renderer. This step does not involve communication to a separate client and we use it for benchmarking the more complicated procedures.

In a second step, we describe the integration into the Shading Atlas engine while using a local client. In the final step, we describe the necessary enhancements upon the previous step when also considering network communication and asynchronous behavior.

We use the overview presented in Figure 2 to sketch the server and client implementation to line out which parts of SAS are successively modified to achieve each of the

aforementioned goals.

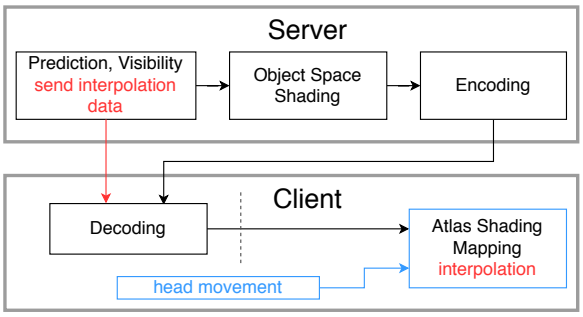

Figure 2: SAS pipeline overview, with modified stages highlighted red, the blue sections represent a decoupled part from the server to perform frame-rate upsampling

3.1 Loading and initializing animated Objects

All integration steps commonly use a novel *AnimationManager* that has direct access to the scene information on the server. Since all rendering systems utilize the same scene geometry, we can directly benchmark the output in terms of computational performance and visual quality. All required animation data structures are prepared for GPU usage and grouped in one *AnimationMaterial* per model.

We decided to use Autodesk's Film Box (FBX) file format [1] to access animated scenes. The FBX format is capable of storing substantially more scene information than simple animations, therefore we preprocess scenes accordingly during loading. One example would be limiting the number of vertex to bone connections to exactly four bones, which is a rather common upper bound we observed. Moreover, we extend the existing loading procedure which generates intermediate data not saved in the proxy files, such as levels of detail or tangents. Most importantly, we sample the skeletal animation information directly form the FBX files. For the purpose of streaming animation information to the client, however, we require a linearly sampled representation of all present keyframe animations.

In general, it is not possible to correctly convert an evaluated animated path into a new set of keyframes due to decomposition artifacts in line with sampling theory. The number of required samples is comparably large and introduces considerable data redundancy. We further need to support additional rigid animations of entire models on top of local skeletal animation. These can be defined in the original FBX file, again as hierarchical transformation and we support an additional layer of freedom for scene generation through a configuration file. This configuration describes the scene with all its models to load and also a model matrix path for each model. Through this layer, the framework allows a presampled rigid animation for each mesh. However, the rigid animation curve might not be sampled in sufficient intervals, which potentially leads to animation artifacts.

The loading procedure culminates in a combination of mesh and *AnimationMaterial*. The following properties of *AnimationMaterials* are relevant for the rendering pipeline.

- global animation info[1]
- animation info[2]
- saved vertices[2]
- skeleton:
  - skinned vertices
  - bone hierarchy meta
  - bone hierarchy
  - inverse bone offset
  - animation meta
  - keyframes

The animation system is designed to allow instantiation of meshes. The animation manager keeps track of the initialized meshes and detects duplicate skeletons for reuse throughout the entire scene.

The first, *global_animation_info* binding contains the actual time, an array of all prediction time stamps and the prediction count. The next two properties, *animation_info* and *saved_vertices* have to be generated for each individual model. The Animation Manager handles this through a model to binding mapping which holds the corresponding vulkan descriptor set. In contrast, the same skeleton information buffer is used for all models with the same animated mesh.

The *animation_info* structure describes additional data used in various shading stages in the server pipelines. That information includes whether the given animated model has a skeleton or is only transformed by a hierarchical transformation define in the FBX file. This structure also provides additional animation steering data like time shift or repetition parameters.

The visibility stage draws the whole scene for each predicted view and saves the visible triangle IDs. This produces the exact visible set for each predicted viewport and combines them into a potentially visible set. It is the foundation for future steps in the SAS pipeline like the level of detail selection in the Shading Atlas. As the evaluated vertex positions will be needed multiple times in the further pipeline steps we cache them in the *saved_vertices* array.

3.2 Forward renderer

The forward renderer is our test bench and does not use a server-client model. It draws all models in a standard render pipeline and directly presents the results on the screen. This builds an appropriate environment to describe the skeletal animation in the vertex shader.

[1] Buffer is global for all models.
[2] Buffer is model dependent.

Typically, per-vertex information, such as bone bindings and blend parameters, is provided through vertex attributes in the vertex shader. However, SAS utilizes a more complex structure of successive stages and we cannot deliver the necessary data in the traditional way.

The crucial part of the skeletal animation is a list of bones connected to each vertex and correlated weights. We use the local vertex index within a mesh as direct mapping between the vertex index and its corresponding skinning data in GPU memory.

In the forward rendering pipeline, this is simply obtained by the Vulkan vertex shader built-in variable *gl_VertexIndex* and the additional *vertex_offset* variable. If the mesh has a deformer, i.e. it is animated by a skeleton, the vertex position will be constructed through the corresponding *SkinnedVertex* entry as described in the *AnimationMaterial*.

To evaluate the per-vertex bone matrix, all connected (up to four) bones are evaluated and then combined accordingly to the per-vertex weights.

The calculation of one bone matrix is done by multiplying its local transform with all parent transform until the root node is reached. Additionally, we need to apply the *bone_offset* transformation to this whole chain. This hierarchy is defined in the *bone_hierarchy_meta* array which holds a mapping for each bone to a list of all its parent bones needed for the evaluation. This list is hierarchically sorted and enables a GPU implementation to travers the skeleton in the correct order.

Additionally, bones have a mapping to their corresponding keyframe data and animation start and stop times. Sampling the keyframes with constant intervals allows for fast and efficient lookup of this data. Each keyframe stores local translation, rotation, and scale per bone. To fulfill rotations SO(3) properties described by Rausch et al. [2], we interpolate the data as quaternions and then transform them to rotation matrices for vertex transformation. We interpolate translation and scale separately and, in a final step, combine them with the interpolated rotation. Since the forward renderer does not use view predictions, the actual server time is used for the transform interpolation.

Finally, we combine the skeleton-based deformation with per-model transformation matrices.

The model matrix consists of two parts. The first is the model matrix added in the configuration file and the second is the model matrix defined in the FBX scene. The FBX model matrix is defined through a hierarchical transformation, which we processed in the animation managers' initialization, in the same manner as a bone. For each animated model, we defined one bone hierarchy index in the *animation_info*, which we use as a model matrix without the use of a *bone_offset* multiplication.

To use the frameworks configuration file intuitively, the there defined matrix will be applied at last to transform the FBX scene model into the configurations desired space.

3.3 Shading Atlas Renderer

The Shading Atlas Renderer features the aforementioned server-client model, but is implemented as standalone executables with two distinct parts. In the non-networked mode, the client accesses the server data via memory copy and consequentially saves transmission and preprocessing steps which are described in more detailed in the next Section 3.4.

The animation system adds a new concept of interpolated vertices to SAS. Interpolated vertices allow client-side interpolation of each vertex position between server updates within the range of predicted view points. For each animated vertex, the server generates separate *InterpolatedVertex* entries per predicted view point.

The client is aware of the associated timestamps and generates intermediate vertex positions by interpolating those entries, resulting in a smooth animation, even if the server update rate is considerably lower than the client frame rate. However, we cannot extrapolate beyond the provided predicted view points and require sufficient server update rates or, alternatively, appropriate predictions, in order to avoid visual artifacts.

Let us first consider the client-side interpolation in order to understand the resources the server needs to provide. On the client, the Atlas Mapping Stage rasterizes the geometry provided by the server and texture maps the transmitted Shading Atlas without any additional shading computations. The new animation system adds the *InterpolatedVertices* array, a vertex mapping, and client animation information to this pipeline.

The vertices drawn in this stage are a compact copy of the visible vertices as computed in the server's visibility stage. This implies that a mapping to the interpolated vertex array needs to be generated since the direct affiliation is lost in the compression step.

The mapping flags non-animated vertices in order to distinguish them accordingly. Further, the mapping must be cleared before each server update and hence all values which are not actively set are considered non-animated.

Client animation information consists of the actual time and time stamps for which the *InterpolatedVertices* were evaluated. With that information, client interpolates the correct vertex positions during animation.

Let us next discuss the adaptations of the server rendering pipeline. In contrast to simple forward rendering, SAS features multiple stages in order to determine visibility of geometry and the appropriate Shading Atlas. The most relevant stages are as follows:

- *IDBufferStage:*
  In this stage, the animation material generates the animated vertex positions for each of the multiple predicted view points and saves them for each mesh in the corresponding *saved_vertices* array.
- *LevelSelectionStage:*
  The level selection is only called for visible geometry and therefore the processed vertex index is a vertex attribute. With the vertex index, the first animated position can be fetched and forwarded to the shading stage.
- *ShadingStage:*
  In the non-networked application, this ShadingStage fills the compact vertex buffer for the atlas mapping stage. Therefore, this stage is also responsible to create the entries in the *InterpolatedVertices* buffer and the *vertex_mapping* for the final stage. Interpolated vertices are generated in parallel and sorted through an atomic counter. The vertex positions are acquired through the saved vertex positions array and the atomic counter value is the corresponding mapping value.

3.4 Shading Atlas Networked Renderer

The Shading Atlas Networked Renderer extends the previously described pipeline by adding an additional networking component. Since no direct access to the server data is available, the server sends both geometry information and the Shading Atlas, and the client consequentially needs to process this data before displaying.

As the networking is only simulated in this renderer, the data to send will only be copied into new arrays and then preprocessed as if it would actually be sent over a network. However, it also allows us to simulate network properties such as latency or transmission rates.

The additional animation information mostly affects transmission of geometry. SAS uses a *VertexSend* message that contains a vertex position and its index for indirect rendering. This message is sent for all visible vertices after the LevelSelectionStage finishes computation. This induces that, for each predicted view point, we need to send the correlated animation information for each animated vertex.

Rather than single vertices, the server sends compact arrays of multiple vertices that are reconstructed into renderable meshes on the server.

It is not practical to generate the same structure for the animated vertices since we cannot use the animated vertex data after the next server update. For each visible animated vertex, the predicted vertices are sent as a compact *VertexMessage* array. At the end of this message, the timestamps for the predictions are appended.

This results in the same array generated in the non-networked shading stages, we called the *InterpolatedVertices*. To generate the mapping between the vertex index and interpolation data, we need to process all interpolation messages and build a mapping between their index and the vertex position in the model.

Next, the client preprocessor generates its local draw buffer and utilizes the mapping to create a compressed animation mapping that correlates with the received visible vertices. In the final atlas mapping stage, we can use the *gl_VertexIndex* to access the mapping to the presampled vertex positions, for interpolation with the actual client time.

4 Results

We performed all tests on a Windows desktop PC (NVidia GeForce RTX 2080 Super GPU, 8 GB VRAM, Intel i7-8700k CPU 4.2 GHz, 32 GB RAM). We observed that 0.7GB GPU memory was used by the system and therefore only 7.3 GB remained for the application.

All tests use a square resolution of 1300 pixels with 90° FOV. In comparison to Mueller et al. [6] we do not test with standard game scenes, but rather want to push the limits of the animation system artificially.

The foundation for the test scene is an animated dragon mesh, which will be instantiated multiple times to stress the system.

The dragon mesh contains 22k vertices which form 38k triangles. The mesh maps to 136 bones which are animated over a four seconds time interval, which we loop infinitely. We use a 100ms sampling interval, leading to 4660 keyframes per loop. We instance the dragon model along a three dimensional grid in order to stress the system further.

First, we measure the performance of the system without any animation integration as baseline. This allows us to compute the pure performance loss owed to the the animation systems.We provide the FPS measurements of the server and client to display the performance losses on each part of the system. These measures are depicted in Table 1, comparing scene one on the non-animation base system and animated version.

Scene two runs on the same grid but only with the animated version of the dragon. The results were tested from a static view depicted in Figure 3, with a viewing distance chosen that all dragons per test are visible.

If the animation system is used without animations, the system drops between 30 and 10 percent of its possible frames on the server and up to 20 % on the client. This performance drop is due to additional memory allocation and checks if animations have to be performed per vertex on the client.

Since the CPU system runs asynchronously with the GPU implementations some staged executions are stalled due to linear dependencies on previous invocations. We recognize the client frame rate drop at 81 dragons in the

# Dragons	FPS					
	Server			Client		
	Static S1	Animated S1	Animated S2	Static S1	Animated S1	Animated S2
9	138	98	38	370	355	370
25	129	73	20	322	341	330
49	108	86	11	266	302	255
81	86	70	7	239	220	230
121	69	49	5	67	60	190
169	52	49	3	47	38	150

Table 1: Depicting FPS results on a static and an animated scene; Static S1 represent the server base line

non-animated state, hinting at a general hardware over utilization for specific pipeline stages.

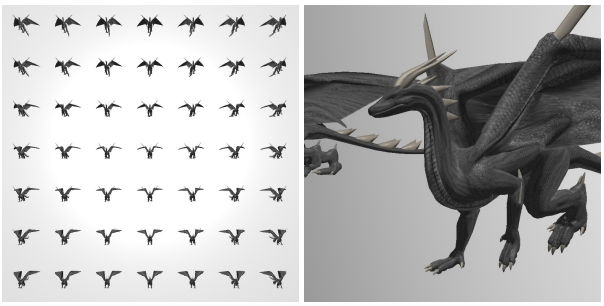

Figure 3: Testing view, and close up

For our test, we compute three view predictions that are 33 ms in the future, generating 100 ms time frame in which the animated scene components can be interpolated on the client without any additional server information.

We observe that the highest possible server update rates drop significantly in proportion to the number of animated objects since each animated vertex evaluates its corresponding bones without reusing previously computed results. Figure 4 depicts the table values of the test scenes as diagram. We can use the animated test scenes curves to identify how many animated vertices can be utilized before the server frame rate drops too low for practical usage.

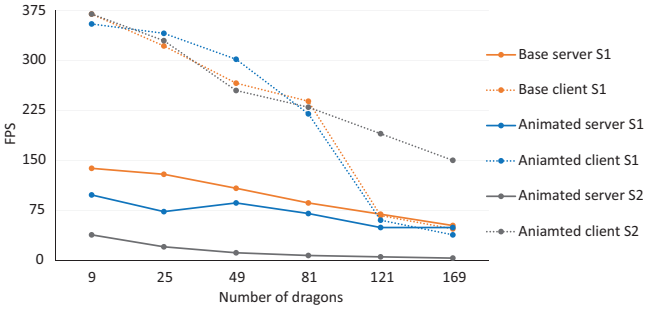

Figure 4: FPS Evaluation

Our results directly align with the hardware suggestions of the original SAS paper. While the server suffers beyond a certain number of animated vertices, the client maintains a high frame rate. This comes at the fact that also model matrix changes are transmitted via interpolated vertices which can be evaluated on the client more efficiently than the current model matrix implementation of the framework.

If the server and client run on separated hardware, performance drops through resource struggles are bypassed and by computing enough predictions the update time on the client can also be bridged.

The biggest limitation for the system is the data transmission rate between the participants. Adding more view point predictions increases the number of samples for animated vertices. In its current state, this results in an addition of 48bytes per visible animated vertex per prediction frame.

Interpolation error We need to subsample the animation curves twice in our system, first in the FBX preprocessing and again on the client between server predictions. Since we do not want to excessively oversample the animation curves, this induces minor deviations. We evaluate this error where at its maximum, which coincides with the original keyframes of the input file.

For our dragon skeleton, the maximum error values are: $translation = 2.57\%$, $rotation = 2.54\%$ and $scale = 0.2\%$. To evaluate the client-side sampling error we use the same server prediction rate as in the performance tests and evaluate the same timestamps again. With this procedure, we reach an accumulated $translation = 2.31\%$, $rotation = 6.32\%$ and $scale = 0.66\%$ loss compared to the original animation curves. These errors are comparably low and visually barely detectable.

Animation tearing artifacts The interpolation concept on the client introduces a new dependency between server frame rate and prediction time delta. If these do not fit together, the sampled animation from the server update will not match the previously sampled animation present on the client. This will introduce animation tearing artifacts between two subsequent server updates since they obviously do not fit together. This effect is strongly visible if the prediction time is way higher than the server frame rate. The linear client interpolation between the predicted position will not be close enough to the actual updated vertex position evaluated on the server. This could be tackled by

a smooth updating procedure to the arriving interpolated vertex data but would additionally distort the animations.

At the moment this can not easily be implemented as the server-client time synchronization turns out to be problematic. In this distributed environment it is hard to exactly synchronize between server prediction and actual client time, even if the delays generated through transmission, preprocessing and displaying are known. Therefore, the system has to run fast enough that SAS can mask the time distortion and the server-client updates do not diverge.

5 Conclusion, Future Work

In this paper, we described a high-quality skeletal animation approach for distributed VR environments. We outlined a method to evaluate skeletal animation directly on the server to overcome certain hindrances at VR clients. Through the extension of the framework with a light-weight client-sided interpolation procedure, we achieve frame rate upsampling of animated scenes. This upsampling reduces requirements for server update rates and guarantee a time-continuous scene representation on the client. While the current implementation of *InterpolatedVertices* allows optimal client frame rates, the server update rate suffers.

In future work, the system could be enhanced through a clever animation sampling approach. Also as mentioned in the previous section the update functionality of the animation system could be reworked to improve server performance. Other bone matrix update approaches could be introduced which are skeleton depending in comparison to the per-vertex variant we choose for the actual implementation.

To reduce animation tearing artifacts, a client-sided round robin system for the animated vertex positions could be introduced which also would save data transfer if predictions are overlapping. This could be further used as a method to reduce the demand for sending animated vertices for all predictions at once and would additionally hide latency.

References

[1] Inc. Autodesk. Autodesk, FBX Adaptable file format for 3D animation software. https://www.autodesk.com/products/fbx/overview, 2019. Visited January 30, 2020.

[2] Rutwig Campoamor-Stursberg and Michel Rausch de Traubenberg. *Group Theory in Physics*. WORLD SCIENTIFIC, 2018.

[3] Jason Gregory. *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2014.

[4] Doug James and Christopher D. Twigg. Skinning mesh animations. *ACM Trans. Graph.*, 24:399–407, 07 2005.

[5] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, page 7–ff., New York, NY, USA, 1997. Association for Computing Machinery.

[6] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. Shading atlas streaming. *ACM Transactions on Graphics*, 37(6), November 2018.

[7] Karol Myszkowski, Przemyslaw Rokita, and Takehiro Tawara. Perceptually-informed accelerated rendering of high quality walkthrough sequences. In Dani Lischinski and Greg Ward Larson, editors, *Eurographics Workshop on Rendering*. The Eurographics Association, 1999.

[8] OculusVR. Asynchronous Spacewarp. https://developer.oculus.com/blog/asynchronous-spacewarp/, 2016. Visited January 30, 2020.

[9] OculusVR. Asynchronous Timewarp. https://developer.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/, 2016. Visited January 30, 2020.

[10] OculusVR. Asynchronous Spacewarp 2.0. https://www.oculus.com/blog/introducing-asw-2-point-0-better-accuracy-lower-latency/, 2019. Visited January 30, 2020.

[11] Iain Richardson. H.264 and mpeg-4 video compression : video coding for next-generation multimedia / iain e. g. richardson. *SERBIULA (sistema Librum 2.0)*, 01 2004.

[12] Shu Shi and Cheng-Hsin Hsu. A survey of interactive remote rendering systems. *ACM Comput. Surv.*, 47(4), May 2015.