Marcus Benjamin Johansson

Skinned Animation Textures

Master's thesis in Computer Science
Supervisor: Professor Theoharis Theoharis
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Marcus Benjamin Johansson

Skinned Animation Textures

Master's thesis in Computer Science
Supervisor: Professor Theoharis Theoharis
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Department of Computer Science (IDI)

Master's Thesis

Skinned Animation Textures

Author:
Marcus Benjamin Johansson

Supervisor:
Professor Theoharis Theoharis

June, 2021

Abstract

Skeletal animations are a widely used technique to pose and animate 3D models with the use of a virtual skeleton. This process of deforming vertices by the means of a skeleton structure is called skinning, and is usually performed every simulation step. Skinning can be computationally expensive when a large number of animated characters are drawn concurrently. This thesis explores techniques which samples animations, and encodes them into pixels on to an image. This allows the animation data to be stored on the graphics processor, potentially reducing communication overhead. The thesis looks at two ways of encoding the animations: vertex animation textures, and bone animation textures. The vertex implementation encodes the deformed vertices directly, skipping the real-time skinning step altogether. Bone textures still perform the skinning step, but are independent on the model complexity, which generally means smaller textures that can be shared between models. The first technique have been previously used by game developers to achieve higher performance when rendering large amount of skinned animated characters, while bone textures, as implemented in this thesis, is a novel approach.

The two techniques were developed and tested in an application, producing animation textures from existing skeletal animations. It supported options to change encoding methods and sampling frequencies through a simple graphical user interface. The encoded animations were visualized side by side in a window to be compared to the ground truth animation produced by traditional skinning methods. A performance rating metric was established to evaluate the relative difference between the skinning implementations tested in this thesis. The findings concluded that animation texture skinning is able to surpass the performance of traditional skinning techniques by orders of magnitude, when the number of draw calls is the performance bottleneck.

Preface

This thesis is the result of the work performed over the course of the spring semester 2021 at the Department of Computer and Information Science (IDI), at the Norwegian University of Science and Technology (NTNU). My passion for 3D graphics, and the knowledge gained through the preceding fall project were the basis for the research conducted in this paper. I want to thank Professor Theoharis Theoharis for being a valuable source of knowledge, experience, and expertise throughout the study program.

Table of Contents

List of Figures

Glossary

associative $(A \cdot B) \cdot C = A \cdot (B \cdot C)$. 3

commutative $A \cdot B \neq B \cdot A$. 7

iterative When a function or algorithm is applied an explicit number of times. This feature is useful when iterating over a collection, with a known number of items. 44

offline rendering Rendering process that is too slow for real time applications, usually due to the complexity of the rendering algorithm (path tracing, global illumination). 1

online rendering Rendering process that is fast enough for real time applications. They typically produce less accurate images than offline rendering. 1

OpenGL A cross platform graphics API. 10, 15, 20, 30, 44, 47, 63

recursive When a function or algorithm is applied in it's own definition. This feature is useful when traversing non-cyclic graphs/trees, where each recursive call can delve into child nodes. 4, 43, 44

shader A program that runs on the GPU. Vertex shaders run for every vertex in the VAO, while fragment shaders run for every fragment/pixel in the frame buffer. 1, 10, 25, 27, 56, 59

texel A pixel in a texture. 20, 56, 64

texture A 2D matrix containing RGB(A) intensity values. Can be visualized as an image. vii, 1, 12, 25, 27–29, 40

VAO Vertex Array Object; an OpenGL object storing state needed to supply vertex data. 10, 15, 19, 25, 43, 44, 50, 52

vertex (plural; Vertices) Points in 3D space that together with triangles defines the surface of a polygonal model.. vii, 1, 25, 27, 40, 56, 59

Acronyms

AABB Axis Aligned Bounding Box. vii, viii, 10, 11, 20, 21, 23, 27, 45, 48, 49, 56, 66

API Application Programming Interface. 1, 3, 10, 13, 15, 47

CPU Central Processing Unit. ix, 1, 12, 13, 15, 19, 25, 30, 39, 43, 51, 52, 56, 59, 63, 64, 66

FPS Frames Per Second. 28, 33, 41, 43, 46, 59, 65

GLM OpenGL Mathematics library. 15

GPU Graphics Processing Unit. vii–ix, 1, 2, 9, 10, 12, 13, 15, 25, 26, 30, 33, 39–46, 51, 52, 55, 57, 63, 64, 66

GUI Graphical User Interface. 14, 64

RAM Random Access Memory. 15

SDK Software Development Kit. 15, 17, 63

VFX Visual Effects. 1

1 Introduction

3D animation is a technique that involves transformation and deformation of 3D models over time to simulate motion. It is often used in both offline rendering, and online rendering applications like in VFX, and video games respectively. Many popular graphics API's support a feature called hardware instancing (See 2.5), which allows the GPU to draw multiple clones of the same 3D model without having to communicate with the CPU for each instance. While this can result in significant performance gains, a problem arises if the model need to be deformed through skeletal animations[14]. The important aspect of hardware instancing, is that the 3D models are clones, and can not be individually updated after the initial draw command. If one were to render an instanced 3D model with skeletal animations, every instance would synchronously be playing the exact same animation. While hardware instanced models may not be updated from the CPU, the GPU can be programmed through the vertex shader to modify them instead. This thesis explores the idea of sampling skeletal animation data into textures stored on the GPU, which decode and play the animations through the vertex shader. The thesis will look at two main approaches for storing skeletal animation data. The first method involves sampling the deformed vertices directly, which means the skinning procedure is skipped altogether (Note: the thesis will still refer to this technique as vertex texture skinning). The second method samples the skeleton transformations themselves instead, meaning the skinning step is still required.

The thesis will evaluate an implementation with the stated animation techniques and traditional methods through performance benchmarks, to determine the advantages and disadvantages of using them. The first technique which stores animated vertex data into textures is not new, as it has been implemented in different but similar terms before[25] [6]. The more novel technique is encoding the skeleton structure itself, greatly reducing the texture sizes needed to animate complex 3D models. Dual quaternions are also incorporated into the implementations, as a potential substitution of transformation matrices. The thesis outlines three hypotheses based on intuition and previous findings:

1.1 H1

Dual quaternion skinning will always outperform matrix skinning.
Insinuates dual quaternions are always cheaper to compute compared to matrix transformations, in the context of skinning.

1.2 H2

Bone texture skinning will always outperform traditional GPU skinning.
Implies the new technique of storing the skeleton bone transformations into textures to allow hardware instancing, is less expensive than traditional skinning methods.

1.3 H3

Vertex texture skinning will always outperform bone texture skinning.

1.4 Structure

The report is structured as follows:

Introduction	Introduces the project
Background	Covers needed background theory and technology
Methods	Shows how the application was designed and built
Results & Discussion	Contains the results and data of the techniques used and evaluates the findings
Conclusions	Provides the conclusion as a summary of the thesis findings, and explores potential improvements and interesting directions the implementation could be taken in

1.5 Demonstration Video

A demonstration video of the application, showing some of the findings presented in this thesis can be found in this link:
`https://youtu.be/7wh44bQvmtk`

2 Background

This chapter covers the technology used and background knowledge to fully appreciate the techniques covered in this thesis. It includes related works and other relevant techniques widespread in graphics today.

2.1 Theory

2.1.1 Polygonal Models

Polygonal 3D models are the main data structures used in this paper due to their wide adoption within graphics API's and hardware today. The models contains a list of 3D points in space called vertices, and an index-buffer that define polygons where each corner holds a reference/index to a vertex. A polygon data structure may also contain additional attributes for each vertex in the model. Typical attributes include:

- *Normal vector*, specifies the orientation of the vertex.

- *Texture coordinates*, a 2D representation of the polygons, used for texture mapping. Normally utilized for projecting images/textures onto the polygonal model.

2.1.2 Transformation Matrices

A transformation matrix is a 4x4 matrix data structure that can be used to change coordinate systems of homogeneous 3D vectors[23]. A point or vector is homogeneous if it contains 4 components where the last is equal to one: $v = [x, y, z] \longrightarrow v_h = [x, y, z, 1]$. This is done to be able to represent translation as a linear transformation.

The matrices may contain location, rotation, and scale information, and can be composited of multiple transformations as they are associative. This is advantageous

as we can apply a large number of transformations to a list of vertices using only one transformation matrix. A transformation matrix that is used to define the location, rotation, and scale of an object in a scene is usually called a **model** matrix denoted M.

2.2 Scene Graphs

A scene graph is a data structure often used in graphical applications[23]. It is a tree structure which consists of nodes typically representing transformations, geometric primitives, sounds, process volumes (triggers or simulations), or other scene graphs. Each node can have any number of children, but only one parent. Each child inherits the transformation of its parent node, such that the location, rotation, and scale is relative to its parent node. All scene graphs have one common ancestor, usually called the root node - often referred to as the abstract 'scene node'. The scene node acts as the universal entry point to traverse the data structure, where we can compute each node's transform recursively in a depth first fashion using the parent node's transform. These transform components are usually modeled as $4x4$ matrices, that propagate the node transformations through matrix multiplications.

2.3 Skeletal Animation

When approaching animation of characters containing joints and bendable parts, static models are less desirable. One of the proposed solutions were based on Labanotation[4], which abstracts the body into joints. Each joint is described by its location and trajectory through space. Motion could then be expressed through five modes of description:

- Direction sign - Gives the translation of a joint.

- Revolution sign - Describes the rotation of a joint.

- Facing sign - Provides the orientation of a particular point on the surface of the model.

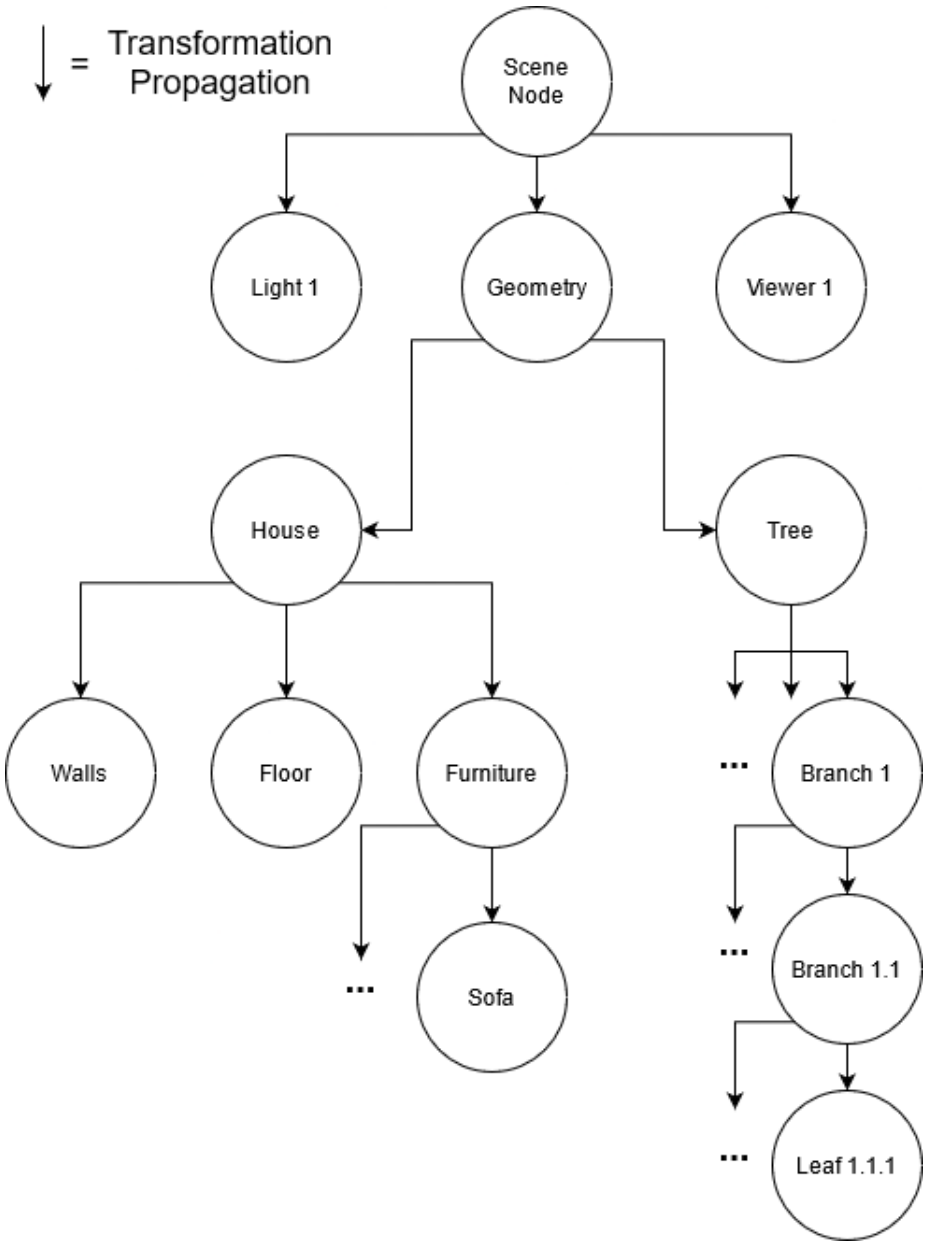


Figure 1: Example of a scene graph where each node exists relative to it's parent node.

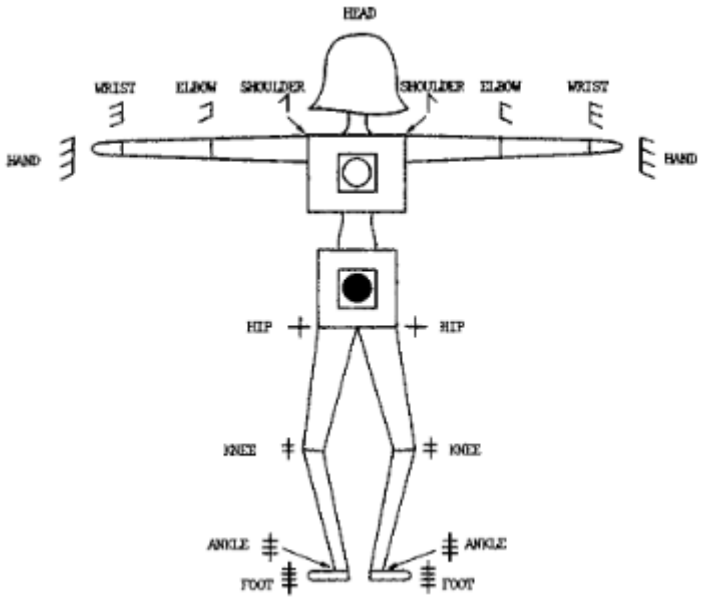


Figure 2: Abstraction of the human body into Joints[4].

- Contact sign - Establishes which parts are in contact with each other or other objects.

- Shape - Involves the shape of a body part by tracing paths or formations.

This was then further advanced into a skeleton data structure that defines bones and the joints that connect them[18]. Each joint also defines an angle to determine the bending direction. Motion could then be modeled through saving skeleton poses into discrete time intervals called key-frames, which the computer interpolates between to simulate smooth motion.

2.3.1 Bone Hierarchy - A Self Contained Scene Graph

The skeleton data structure is constructed like a scene graph - a sub graph within the main scene graph. Each bone is represented as a node which contain one parent bone, and any number of child bones. Like scene nodes, to get the global transformation of a bone, the chain of parent transformations must be calculated, starting with the

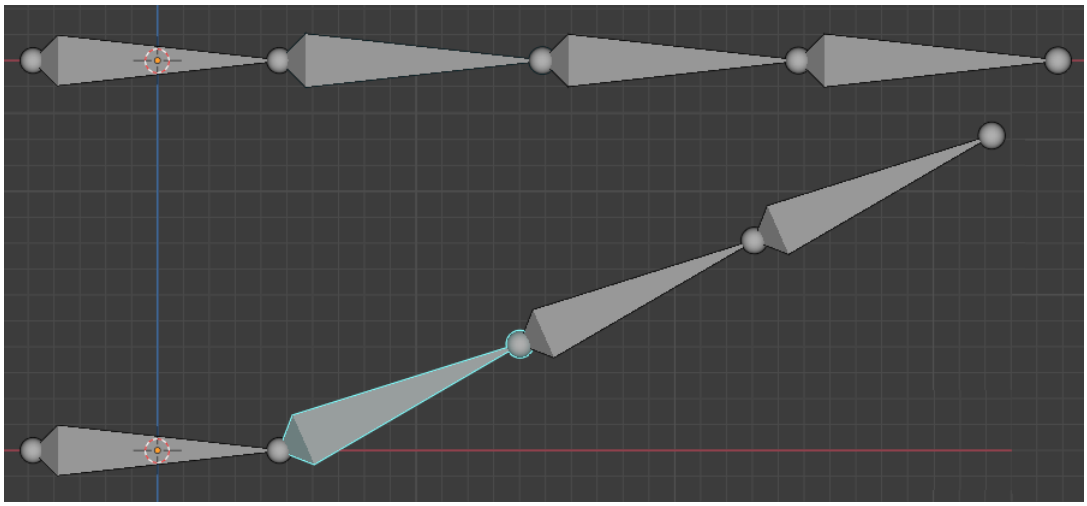


Figure 3: Example of a bone chain. When the second bone rotates, the ancestor bones are deformed with it, even if their individual transformation has not changed.

root bone:

$$W_k = \prod_{i=0}^{k} M_i \tag{1}$$

Where W is the world/global matrix, k denotes the current bone, $k-1$ is the parent bone, and M_0 is the transformation of the skeleton root bone. It is important that the order of multiplication is correct, as the transformations are generally not commutative.

Virtual skeletons are mainly useful in the visualization stage of a 3D model. Skinning is the act of transforming the vertices and thus deforming the model according to the current configuration of the skeleton structure[17][11]. Bindings are defined for each vertex and bone, containing a weight value $[0, 1]$ specifying the influence a bone has on a given vertex. The final skinned vertex position is the sum of the bone transformations, multiplied by their influence:

$$v'_i = \sum_{j=1}^{n} w_{i,j} \cdot D_j \times v_i \tag{2}$$

Where v'_i is the skinned vertex, v_i is the initial vertex position, D is a bone deformation transformation, $w_{i,j}$ is the bone influence on the vertex, and n is the number of bones in the skeleton.

2.3.2 Quaternions

Quaternions are a 4D vector that are an extension of complex number theory, and is defined as

$$q = w + (xi + yj + zk) = (w, v) \tag{3}$$

where w is the real part, and $(i, j, k) = v$ are imaginary[10]. Unit-quaternions where $|q| = 1$ can be used to hold rotational data[23]. In the same way transformation matrices can be applied to vectors through multiplication, the same principle holds using quaternions. The advantage of using quaternions over transformation matrices is apparent mainly during animation, where linear interpolation between two rotations can be performed without encountering gimbal-locking. Gimbal-locking constitutes a situation where two of the rotational axes are driven in to a parallel configuration, losing one degree of freedom. This issue is not apparent with quaternions, as using quaternions to store rotations allows the interpolation to be continuous over the rotational axis. This is done through the use of spherical linear interpolation, which ensures the quaternions are normalized throughout the procedure.

2.3.3 Dual Quaternions

While quaternions can be used for rotations, using two of them makes it possible to hold translation information as well[15]. Dual Quaternions combine quaternions with dual number theory and are defined as:

$$dq = q^r + q^d\epsilon \tag{4}$$

where q^r is the real part, and $q^d\epsilon$ is the dual part[5]. This is ideal for holding transformation information, as they consist of only 8 components vs the 12 found in transformation matrices (4x3, where the last row is omitted)[16]. Another benefit is that dual quaternion skinning suffers less from the "candy wrapper" artifact which may appear when bones twist 180°relative to their parent, as seen in figure 4.

Figure 4: "Candy-wrapper" artifact[14]

2.4 Graphics Pipeline

To fully appreciate the methods and implementations presented in this thesis, some high level knowledge about the graphics pipeline is necessary. The pipeline consists of multiple stages, in which data and commands travel through to compute and produce graphics[7]. Within the GPU, there are a variety of fundamental units operating in parallel, each with their own specific purposes. These include vertex and index fetching, the vertex shader (transform and lighting), fragment shader, and raster operations. The vertex shader and fragment shader are programmable, and can drastically change how the graphics are computed. The vertex shader stage is executed once per vertex, while the fragment shader is executed once per fragment/pixel in the frame buffer, where the final output is stored.

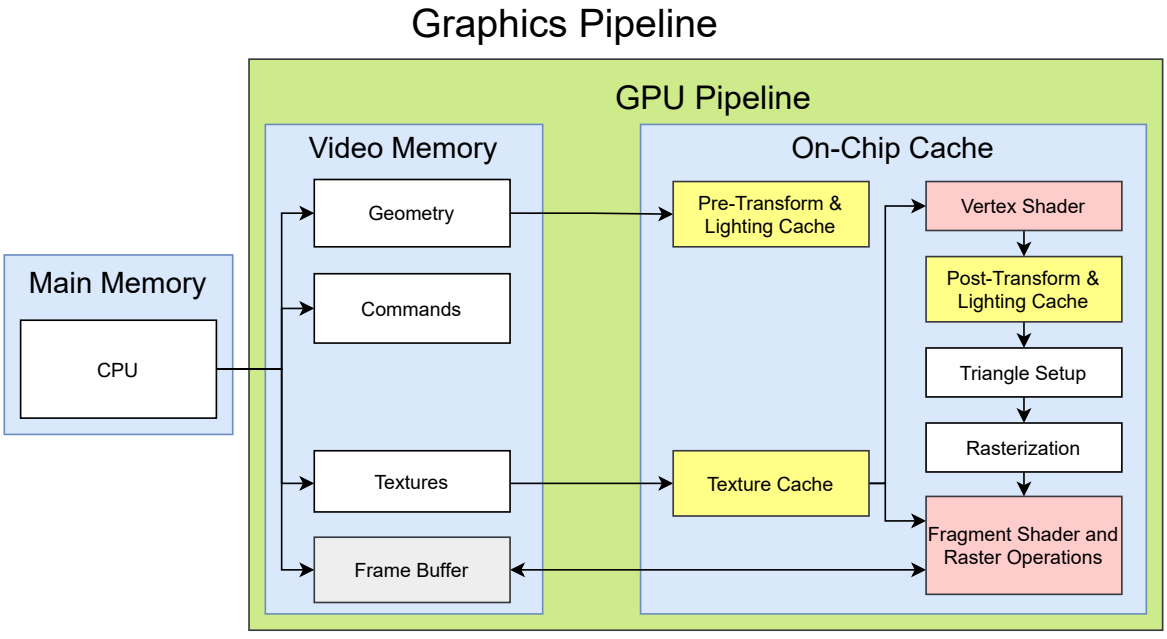


Figure 5: The graphics pipeline[7]. Programmable stages are marked in red.

2.5 Instancing and Draw Calls

Instancing is a technique reusing data to draw multiple, usually identical, instances of the same object[23]. Combined with a scene graph, multiple instances can be placed in the scene, each bound to different nodes, thus allowing for different positions, rotations, and scales. The geometric information is simply referencing one "original" instance, eliminating the need for duplicated data. This may potentially save a significant amount of memory, as well as speeding up the calculation of drawing the instanced nodes - that is, if the nodes are to behave exactly the same over a particular simulation step.

Implementation wise, OpenGL supports instancing in the form of loading the geometric data into array buffers connected to a single VAO. For each instance to be drawn in the scene, the VAO must be referenced, and values essential to the rendering process is passed to the shader and GPU through uniform variables[24]. One example of such a variable is the computed node transformations. These variables are called uniform due to the fact that they remain constant from one shader invocation to the next within a particular draw call. The process of loading uniforms to the shader invokes a state change in the shader program[9] which involves GPU communication.

OpenGL also includes API functions to draw multiple instances within the same draw call[8]. This method is essentially the same as invoking multiple draw commands without changing the shader program state, except for an additional internal integer variable accessible in the shader, equal to the index of the currently rendered instance. This form of instancing is often referred to as hardware instancing[20].

2.6 Axis Aligned Bounding Boxes

An Axis Aligned Bounding Box (AABB) is a subcategory of bounding volumes[23]. Bounding volumes are generally used to improve efficiency in many different algorithms. This is usually done by encapsulating a complex volume with a relatively simpler volume representation. Computationally expensive algorithms may query

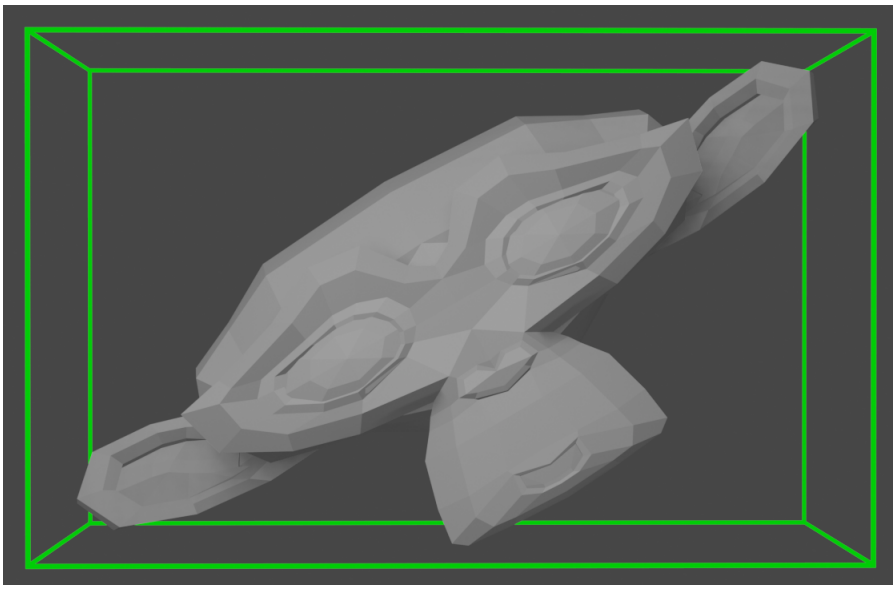


Figure 6: An example of an Axis Aligned Bounding Box.

the simpler volume first, before the complex model is evaluated, such that all models that fail the simple query can skip the expensive algorithm altogether. AABB's are one of the simplest forms of closed bounding volumes. They are defined by two parameters: its position, and its scale. This means AABB's are always aligned with the axes to the coordinate system it resides in, and cannot be rotated to better fit a volume in envelops, as seen in figure 6.

2.7 Student's T-test

The t-test is a statistical test where the test statistic follows a Student's t-distribution under a null-hypothesis[22]. The test can be used to measure if the difference in mean of two data sets are statistically significant. In such cases, the null-hypothesis would state that the two means are equal. To prove or disprove the null-hypothesis, a t-score is calculated from the equation:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} \tag{5}$$

where $\bar{x}$ is the sample mean μ_0 is the population mean, s is the sample standard deviation, and n is the sample size. From the t-score, a p-value can be found from a look-up table to determine the probability of obtaining test samples at least as extreme as the observed samples, were the null-hypothesis is correct. If the p-value

is below the chosen threshold for statistical significance (α, usually 0.05), the null-hypothesis can be rejected, and one can accept the alternative hypothesis (the two means are not equal).

2.8 Related Work

2.8.1 Vertex textures

Vertex textures are not a novel technique. At the Unite 2016 conference [25], Jonas Norberg introduced a new way of rendering a large number of animated characters in the game engine Unity3D. His method involved putting animation-data in textures, such that the expensive process of animating a large amount of skinned models are off-loaded to the vertex shader through the use of hardware instancing. This method was the main inspiration for the fall project, mentioned in 2.8.3. Norberg's implementation was aimed at lower powered systems like phones, to achieve better performance. Multiple animations were assembled into one texture, by the use of compression and texture-atlases, which were decoded and played through the vertex shader. This allowed them to use instancing to render multiple animated models in one draw call, increasing performance considerably.

In 2017, Yi Fei Boon presented *4,000 Adams at 90 Frames Per Second*[6], which used a near identical approach to Jonas Norberg. The title of the talk refers to a 3D model named "Adam", consisting of over 69 700 vertices and 139 100 triangles, being drawn 4 000 times per simulation step, 90 times per second through the use of their implementation. They did not implement any form of compression, but advocated for stacking multiple animations into one texture.

Ashraf and Junyu [3] chose to use a hybridized approach, whereby four distinct animation methods were presented and tested. The first method were CPU skinning, in which all skinning operations were done on the CPU. The second was GPU skinning, where the skeleton was passed to the vertex shader for each instance, doing the skinning on the GPU. Third and fourth used a novel texture approach where a "dummy" fragment shader was used to skin the model, and store post-skinning vertices to a texture. This made it possible to take advantage of instancing, where

for the third method, each model could look up the skinned vertices in the vertex shader. In the last technique, the computed texel vertices was read back to the CPU in a CPU-GPU alternation strategy, instead of using the vertex shader. The model vertices was then updated with the deformed vertices, and finally rendered. This implementation does not skin every animated character individually, where each instance in a predefined group are playing the exact same animation. The position, rotation, and scale was varied, and placed within other groups playing different animations to reduce cloning artifacts. The read-back (fourth) method rated highest in the performance statistics, then came GPU skinning, vertex fetching, and finally CPU skinning at the bottom. They attribute the poor performance of the vertex fetching technique to the expensive cost of texture look-up's through the vertex shader in the graphics API that was used.

2.8.2 Bone textures

In 2013, Rudomin, Hernández, Gyves, Toledo, Rivalcoba, and Ruiz [20] publicized *GPU Generation of Large Varied Animated Crowds*. In the paper, they propose a novel approach to character modeling and animation. Their implementation uses texture-based methods to model, rig, skin, and animate varied crowds. They achieve variety through the use of template sub-models (head, arms, torso, legs etc.) which are combined together to form a final character. The characters all share the same internal attributes, including the texture coordinates, which they use to form a universal skeleton in texture space that can be used to skin the sub-models. The paper did not disclose any performance statistics of their approach.

2.8.3 Fall project

The fall project[13] is a precursor to the research presented in this thesis. It explored a method of storing and animating skinned 3D polygonal models by sampling vertices at fixed intervals during the animation timeline (vertex textures). The implementation succeeded by encoding vertex deviations to textures to be loaded to the GPU to minimize communication between the two processors. This allowed for hardware instancing of skinned animated 3D models, reducing draw calls considerably.

No benchmark were present however. The goal of the fall project was to evaluate the implementation related to its simplicity, flexibility, efficiency, and usability. The project concluded that together with a rudimentary GUI the method and its output could be altered to maximize animation fidelity relative to the generated texture size. In terms of implementation and usability in a hypothetical content pipeline or existing program, special tools need to be developed to take advantage of the technique. Further, the project included no form of compression on the texture data, and argued that the produced texture sizes could be significantly reduced through different means of animation compression.

3 Methods

3.1 Overview

In this section the implemented methods and techniques are defined in detail. This includes the libraries and hardware used, such that one could construct a similar solution.

3.2 Hardware Specification

This is a list containing the hardware architecture of the system when the application was built, tested, and evaluated:

Platform	Windows 10 64-bit
CPU	Intel Core i7-9770K @4.80 GHz
GPU	Nvidia GTX 1080TI
RAM	32GB DDR4

3.3 Application Control Flow

The application is implemented using OpenGL in the C++ programming language. It can import fbx files through the fbx SDK provided by Autodesk. During importation, the file is searched for relevant data structures containing geometry, skeletons, and animations. The data is given through fbx classes and data structures, which are converted to data structures used by the rest of the application (OpenGL Mathematics library (GLM)). This process is done through a custom FbxAnimationConverter class, which is also responsible for animating and skinning the mesh during the rendering loop. After the mesh is converted, a VAO and other accompanying buffers are generated to be able to use the OpenGL rendering API. The application supports importing multiple meshes per fbx file, in which this process is repeated for each one. This process is visualized in figure 7

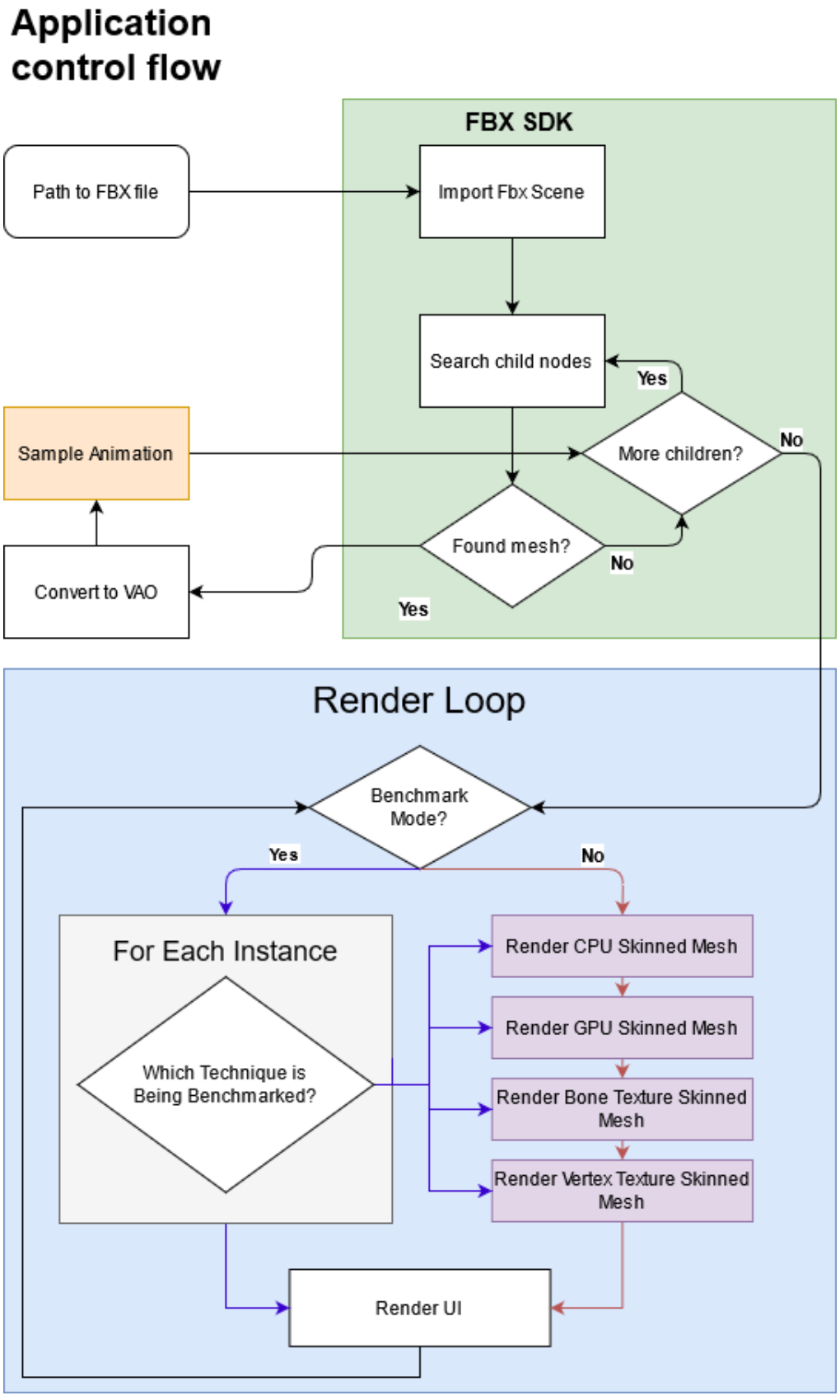


Figure 7: Application control flow

3.3.1 Skeleton data structure

The skeleton representation in the fbx files are simply nodes within the scene, containing a transformation - constituting of a location, rotation, and scale. Both the fbx SDK, and the skeleton data structure used in the application takes the form of a scene graph (as seen in figure 8). The initial transformation of each bone represents the resting pose/rest-pose of the skeleton, where no deformation is present. For humanoid characters this pose is usually called a T-pose[9] or an A-pose, where the character may resemble the aforementioned letters, as seen in figure 9.

One or more meshes may be bound to a skeleton through another data structure named a cluster. These are bindings that map vertices in a mesh to the skeleton. This mapping contain a weight value which determines the influence each bone has on each vertex. The final vertex position is the sum of the applied bone transformations, multiplied by their respective weights:

$$v'_i = \sum_{j=1}^{n} w_{j,i} \times v_i \times B_j \times L_j^{-1} \tag{6}$$

where v'_i is the final deformed vertex, v_i is the initial vertex position, n is the number of bones in the skeleton, B_j is the bone transformation in world space, L_j is the bone link matrix, and $w_{j,i}$ is the binding weight between vertex i and bone j.

When the skeleton data-structure is in the rest-pose configuration, no vertices should be moved from their initial position during the skinning step. For this reason, the link transformation L^{-1} must be used, which is simply the inverse world-transforms of the bones in the resting pose. When skinning the vertex, the bone transformation is multiplied with the respective link transformation, which would result in the identity transform in the case of the rest-pose.

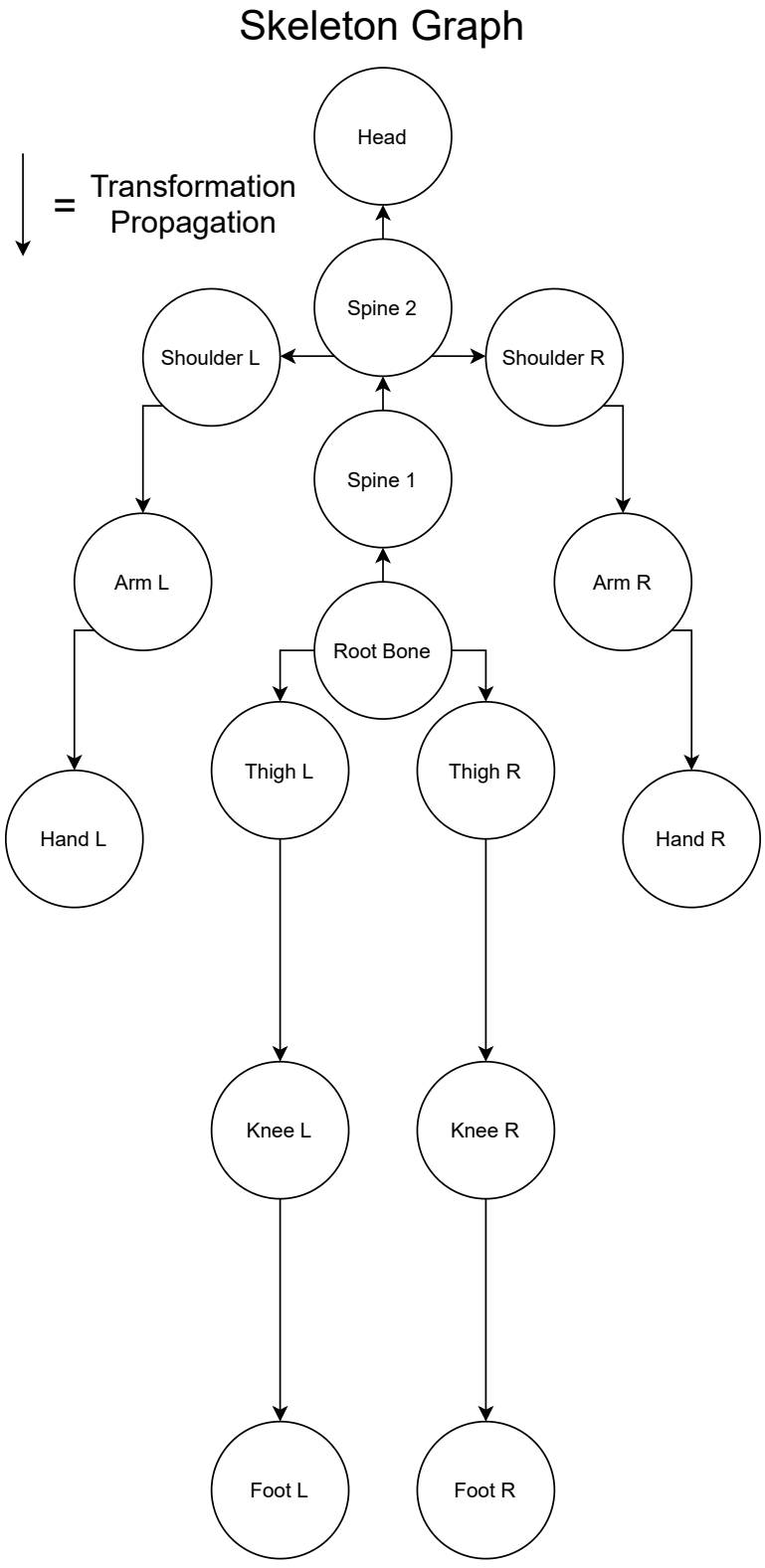


Figure 8: The skeleton data structure is a scene graph, where every node represents a bone. This figure represents a (simplified) humanoid skeleton.

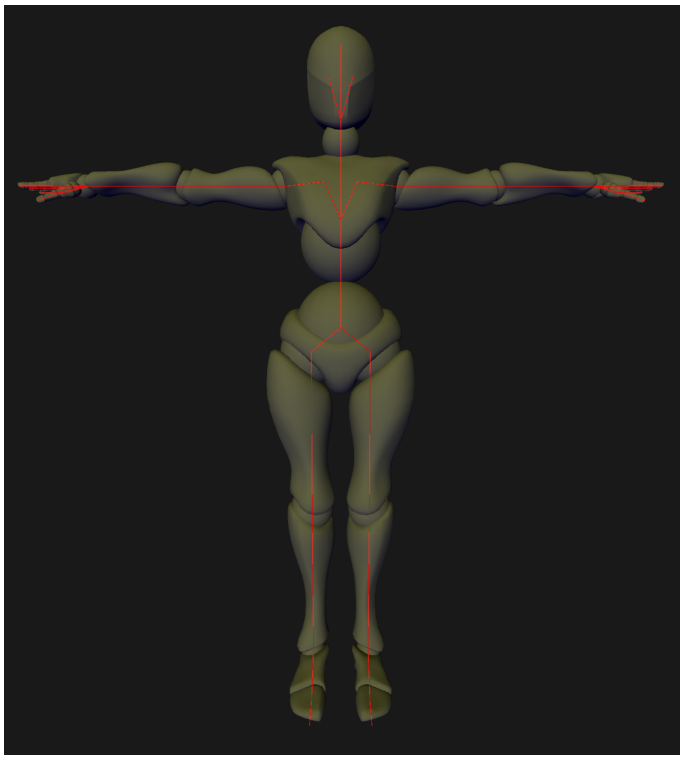


Figure 9: T-pose - the rest-pose of this particular character when no animation is played. The red lines represent the bones in the virtual skeleton.

It is important to note that the vertices are duplicated for each polygon during the VAO conversion. This is to support both flat and smooth shading, as with flat shading the vertex normals may differ per polygon. To avoid duplicating the vertices in the animation sampling process, each vertex gets an additional vertex attribute, pointing to their "true" index in the original model with no vertex duplicates. It is this attribute which is later used to look up the correct row index in the animation texture.

3.4 Sampling Animation Key-frames

3.4.1 Vertex Samples

To be able to encode vertex positions into a textures, each and every vertex position throughout the animation needs to be known. For this reason, a CPU skinning method must be used. The skeleton is animated at a fixed interval chosen by the user, which is defaulted to the native frame rate of the animation. At every interval, the mesh is skinned by the CPU to sample the deformed vertices.

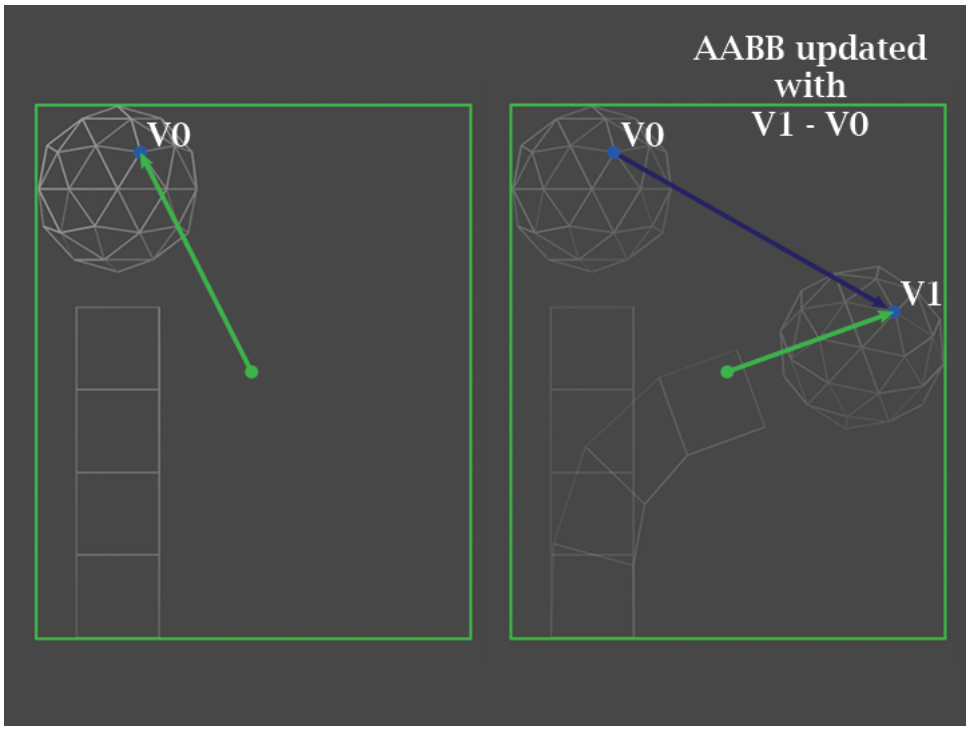


Figure 10: Vertex sampling example. The deviation from the rest pose is being sampled, not the vertex position itself.

When the vertex positions are being encoded into texels, an AABB is updated during the sampling process. For each column in the texture, we encode a positional delta rather than explicit vertex locations to maximize precision. This is important as this implementation uses a 24-bit encoding scheme, which means only one byte is used per channel. For each frame, and for every vertex, a minimum and maximum vector is updated to construct the AABB. The center of the bounding box is then calculated and encoded into the texture, along with its dimensions. This allows for a more efficient encoding scheme, where the encoded vertex position is relative to the AABB center, where its 256 steps of precision is confined within the bounding box dimensions, utilizing the limited precision the most efficiently.

Another consideration is the size of the texture itself. For the vertex animation textures, the first column of texels are reserved for metadata about the texture, such that a shader can be reused for multiple different animation textures. This includes the length of the animation, sampling/frame rate, and the AABB definition. For the rest of the texture, the rows represents the individual vertices in the model, while the columns consists of the different key-frames in the animation. By testing, it would seem OpenGL prefer texture sizes which are a power of two. Arbitrary sizes

appeared to produce strange artifacts, which disappeared when enforcing a power of two rule.

3.5 Bone Samples

Bone sampling shares many similarities with vertex sampling. In this instance, AABB's are also used to improve encoding precision of the bone samples, but rather than using one bounding volume for the entire mesh, an AABB is used for every bone in the skeleton, as seen in figure 12. Metadata, as with vertex animation textures, are stored in the first column. Since bone animation textures use one AABB for each bone, the amount of metadata texels required increases considerably.

3.5.1 Matrix Samples

For matrix transformations, the translation information is stored in the last column. To be as size efficient as possible, each row of the 4x3 matrix are encoded. This means that three texels with four components (RGBA) are used to encode one bone transform. Because the rotational values (3x3 matrix) always range from $[-1, 1]$, the values are mapped directly to 8 bits of precision: $[0, 255]$. The translation information is encoded relative to the AABB center, and divided by the bounds: $t = \frac{t_0 - C}{B} \cdot 255$ where t is the encoded translation, t_0 is the sampled translation, B is the bounding volume dimensions, and C is the bounds center. Each bone matrix is encoded as such:

$$\begin{bmatrix} r & r & r & t_x \\ r & r & r & t_y \\ r & r & r & t_z \end{bmatrix} \Rightarrow \begin{bmatrix} R & G & B & A \\ R & G & B & A \\ R & G & B & A \end{bmatrix}$$

AABB metadata can be encoded using 3 texels, as to not exceed the number of rows in the texture (except for the two extra texels used for animation length, and sample rate) which means we have $3 \cdot 4 = 12$ 8-bit values to work with. For each bounding volume, both the center and bounds is encoded in an integer part, and a decimal part. The center integer is cast directly to the first three texels where we assume each component does not exceed 255(absolute). The center decimal part is mapped to the three next texels, from $[-1, 1] \Rightarrow [0, 255]$. The sign of the decimal part gives

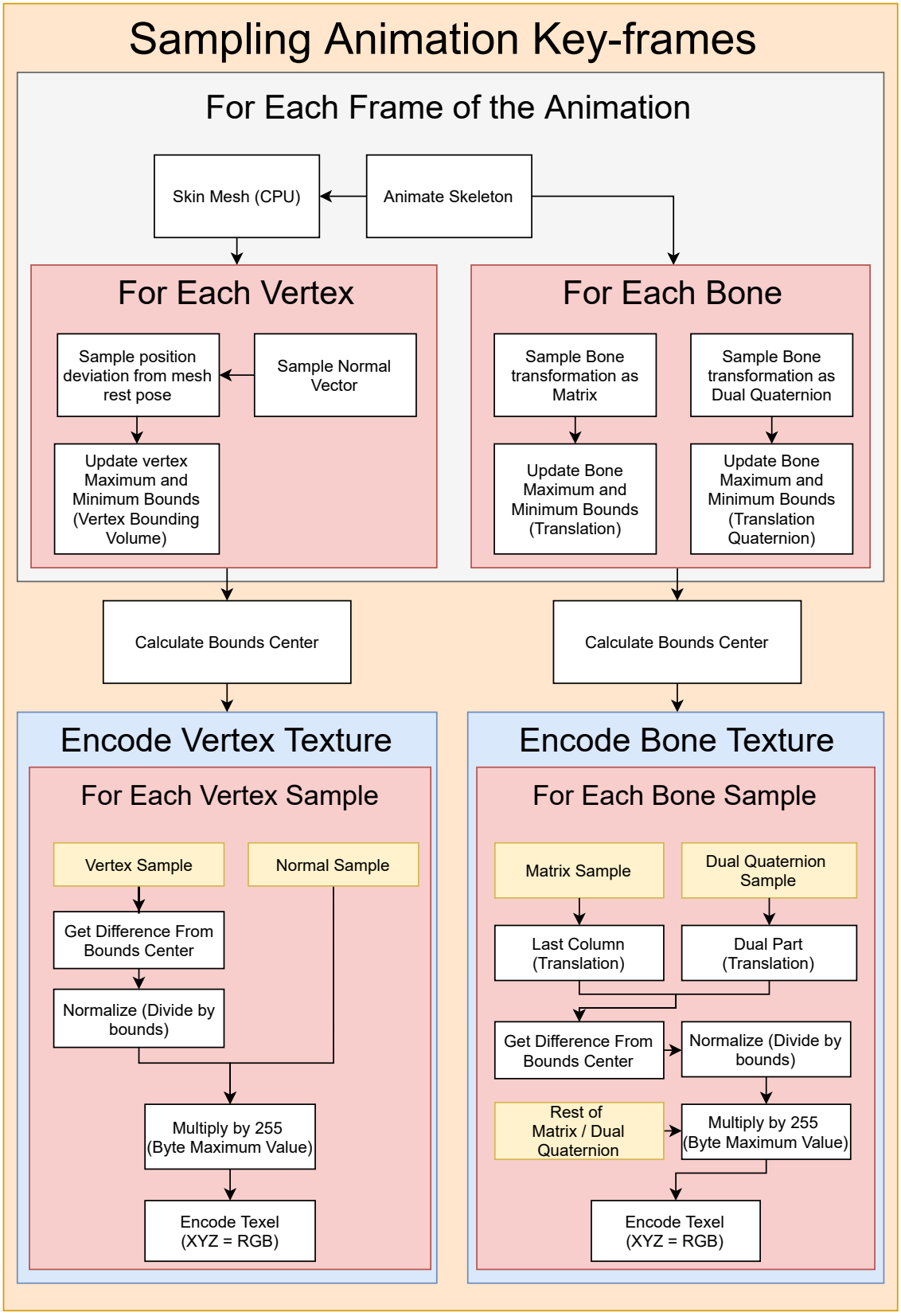


Figure 11: Animation Sampling control flow.

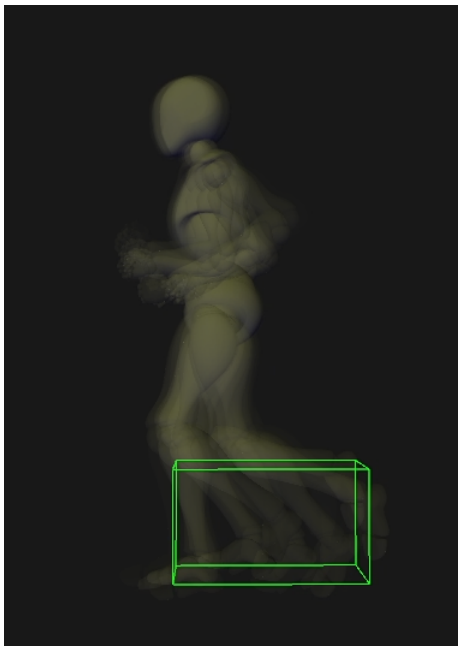


Figure 12: An AABB is constructed for every bone in the skeleton. Here the left ankle bounding volume is visualized.

the sign of the whole decoded component in the shader. The same procedure is done for the AABB bounds, however we gain 1 bit of precision because the bounds are always positive such that they can be mapped from the range $[0, 1]$ instead.

3.5.2 Dual Quaternion Samples

Dual quaternions have the advantage of being 50% smaller in terms of memory footprint, compared to transformation matrices. As with matrix encoding, AABB's are also used for each bone to increase translation precision, and are encoded in the first metadata column in the texture. This unfortunately also induces one less texel for AABB metadata encoding. This implementation only encodes the integer parts of the bounding volume, omitting the decimal value completely. The encoded dual part is then found by $q^d = \frac{q_0^d - C}{B} \cdot 255$ where q^d is the encoded dual part, q_0^d is the sampled dual part, B is the bounds, and C is the bounds center. The real part responsible for the rotation is always a unit-quaternion, such that we may map them directly from $[-1, 1] \Rightarrow [0, 255]$ similar to the rotation matrix. In the texture, each part of the dual quaternion gets one texel each:

$$\begin{bmatrix} q_x^r & q_y^r & q_z^r & q_w^r \\ q_x^d & q_y^d & q_z^d & q_w^d \end{bmatrix} \Rightarrow \begin{bmatrix} R & G & B & A \\ R & G & B & A \end{bmatrix}$$

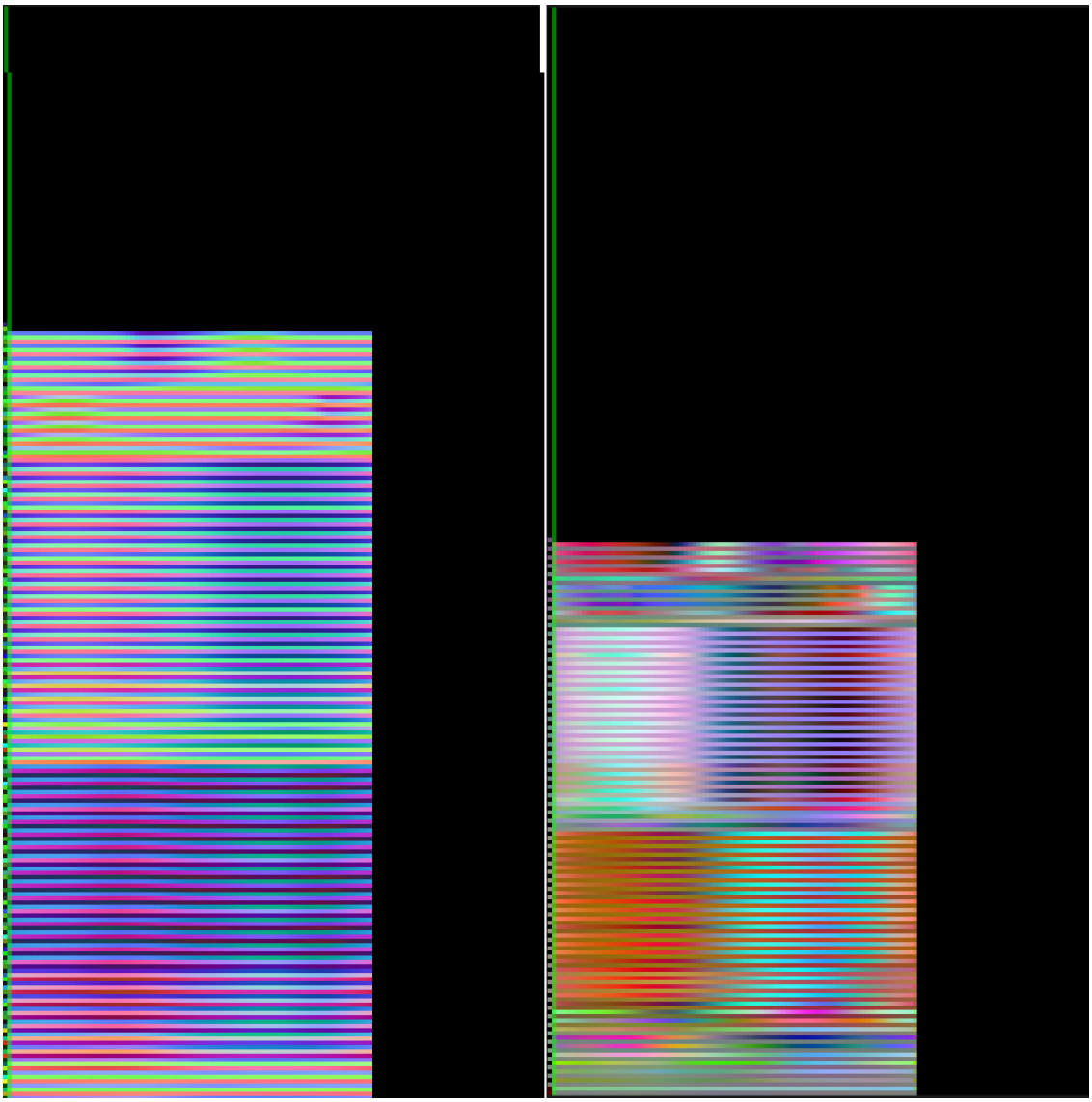


Figure 13: **Left:** Matrix texture. **Right:** Dual quaternion texture. The resulting animation textures from the vampire model, playing the strut animation. The green line is a play head visualizer, showing which texels are currently being sampled in the animation by the shader. Note: the alpha values are not present in this visualization.

3.6 Rendering

This section presents the rendering loop of the application, which is important as context for the benchmark. Certain steps are universal for all the implemented rendering pipelines, including computing model matrices in the scene graph, and extracting the view-projection matrix from the camera view point.

3.6.1 CPU Skinning

The first rendering method uses only the processor to skin the vertices. This technique is not used in the benchmark, but is included to be able to sample the vertices to an animation vertex texture. In the rendering step, the skeleton is animated for each instance being rendered. The model matrix is also updated per instance. Then for each mesh per instance, the mesh is skinned and updated in the VAO. Finally the mesh is rendered by a simple vertex shader, which is only responsible for applying the model matrix to the vertices and normals.

3.6.2 GPU Skinning

The GPU skinning pipeline is the first technique being used in the benchmark. Similar to the CPU skinning, the skeleton is animated for each instance being rendered. The computed bone transformations (matrices or dual quaternions) are then loaded to the shader through uniform variables, along with the instance model matrix. Finally, each mesh of the current instance are drawn.

In the vertex shader, each vertex uses vertex attributes (byte vectors) for indexing three bone transformations in the uniform bone array computed by the CPU in the previous step. The bone transformations are then aggregated and weighted by another vertex attribute (default float-32 vector) to form the final deformation transformation. For dual quaternions, an additional normalization step is required to ensure the real part remains a unit-quaternion. Finally, the vertex and normal are deformed by the computed deformation transformation. The GPU skinning pipeline is visualized in figure 14.

3.6.3 Vertex Texture Skinning

Vertex texture skinning was previously developed in the fall project, and remains similar in this implementation. For each mesh in the original instance, the animation textures are bound before the mesh is drawn instanced. Continuing in the vertex texture shader, each vertex reads the metadata located in the first column of the texture to correctly play the encoded animation. These include animation length

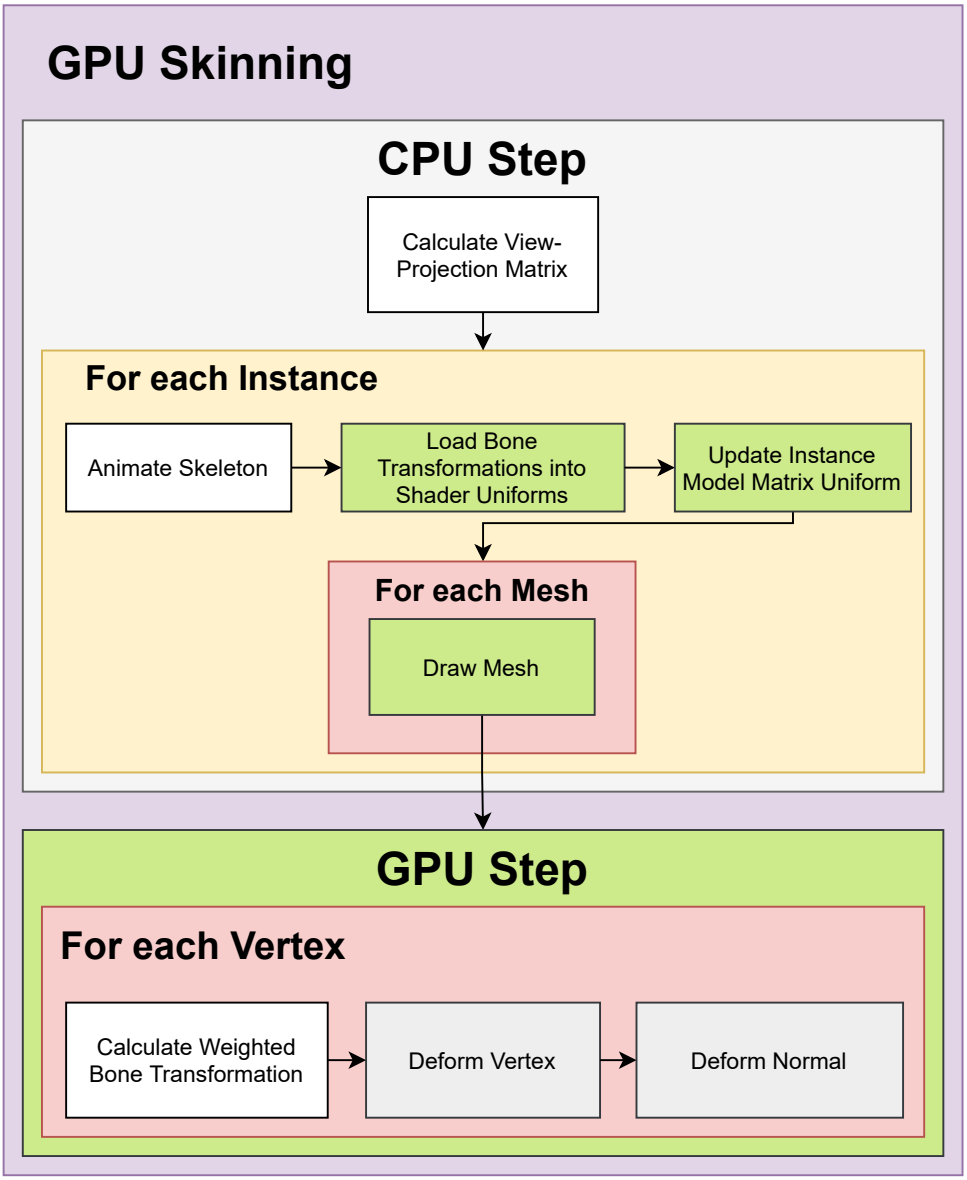


Figure 14: GPU skinning pipeline - ambiguous for both matrix and dual quaternion implementations.

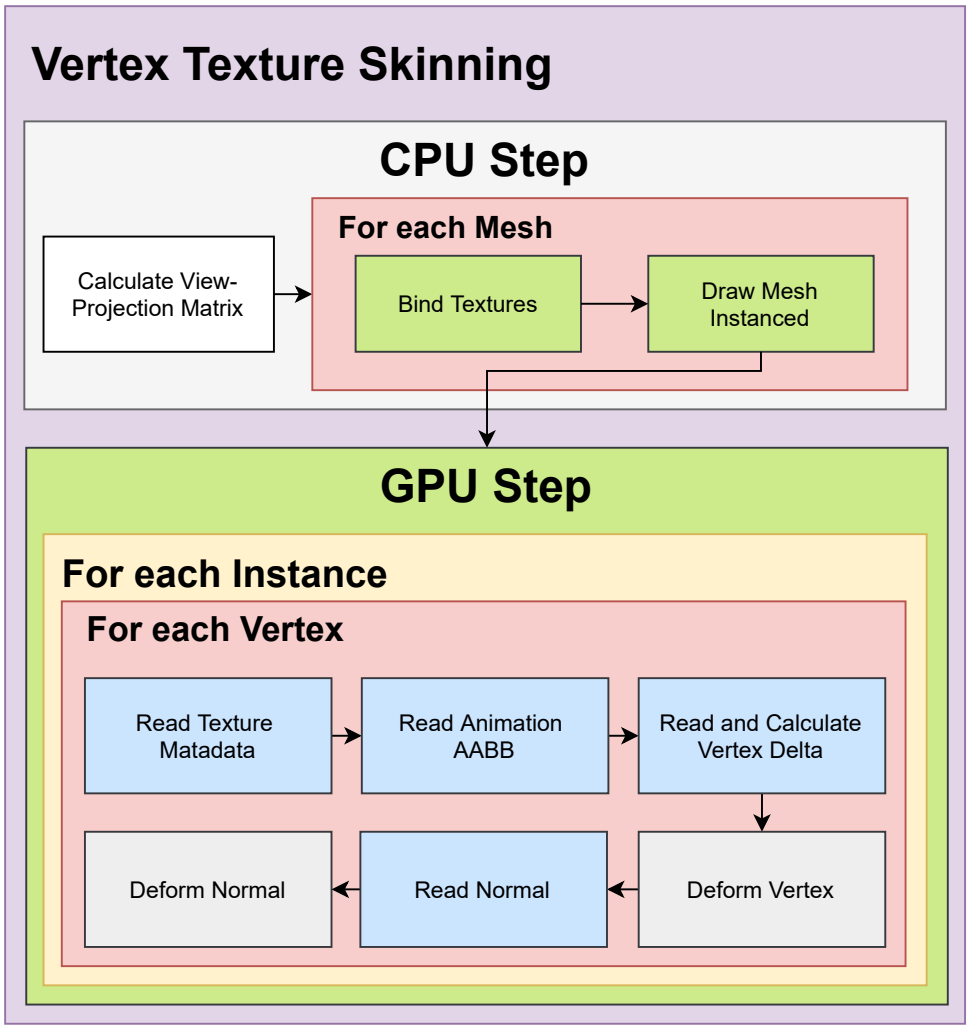


Figure 15: vertex skinning pipeline.

(number of texels used in the horizontal axis), sampling rate, and the AABB. Based on the current animation time and sampling rate, two texels are read from the texture. The values are decoded and linearly interpolated between to find the final vector. This decoded vector is then added to the initial vertex position, to form the output vertex. The same steps are repeated for the normals, but must be done with a separate animation normal texture. This process is visualized in figure 15.

3.6.4 Bone Texture Skinning

In the bone texture pipeline, loading the link transformation are done as an additional step before rendering the meshes, to be able to encode the bone transforms in world space. Next, for each mesh in the original instance, bind the animation bone textures, and subsequently draw the mesh instanced. In the bone texture shader,

metadata is also read per vertex to decode the animation data correctly. Next, vertex attribute indices (byte vector) are used to read three bone transformations (three texels for matrices, two for dual quaternions), to be aggregated and weighted by a second vertex attribute (default float-32 vector). In the case of dual quaternions, they are also normalized to ensure the real part remains a unit-quaternion. This step is repeated to find a second deformation transformation to interpolate between, according to the current animation time. For both matrices and dual quaternions, linear interpolation is used. Because the dual quaternions are always normalized, employing spherical linear interpolation is unnecessary to achieve accurate rotational data. The vertex and normal is then finally deformed by the computed deformation transformation. The bone texture skinning control flow is visualized in figure 16.

3.6.5 Benchmarking

This section presents the benchmark algorithm used to get performance metrics from the rendering techniques specified in the sections above. The benchmark uses exactly one view port, testing one rendering technique at a time. The metric being sampled is time since last simulation step (frame), known as the time delta. The data is transformed to Frames Per Second (FPS) ($\frac{1}{t_d}$ where t_d is time delta) in the final output, where a greater value represents better performance. FPS is also used as a more familiar performance metric commonly found in other graphics applications. This means the output data will look skewed, resembling a $\frac{1}{x}$ function versus a linear proportional graph.

Benchmark Algorithm

The benchmark algorithm is executed in a loop:

- Increment pre-sample clock until 500 ms has elapsed.
- Sample frame times for 1000 ms, and average the result.
- If the frame time difference from the previous sample is within the threshold $[1, 2)$ ms, or the current number of instances is one more than the previous sample, save the data point.
- If the frame time is above or equal 2 ms, set the number of instances $I = I_0 + \frac{I_1 - I_0}{2}$ where I is the new instance count, I_1 is the current instance count,

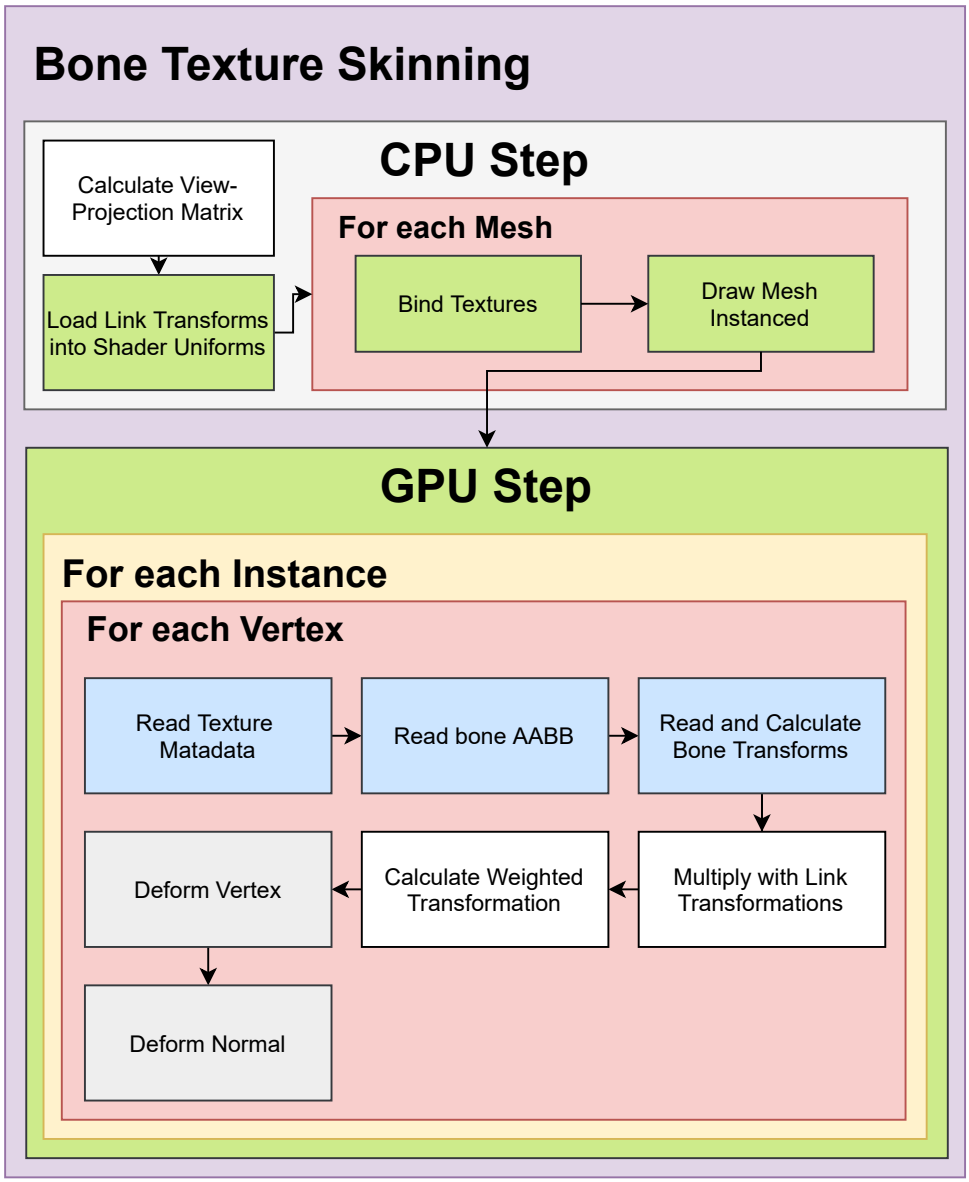


Figure 16: Bone texture skinning pipeline - ambiguous for both matrix and dual quaternion implementations.

and I_0 is the instance count of the previous sample point.

- If the frame time is below 1 ms, set the number of instances $I = I_0 + 2(I_1 - I_0)$.

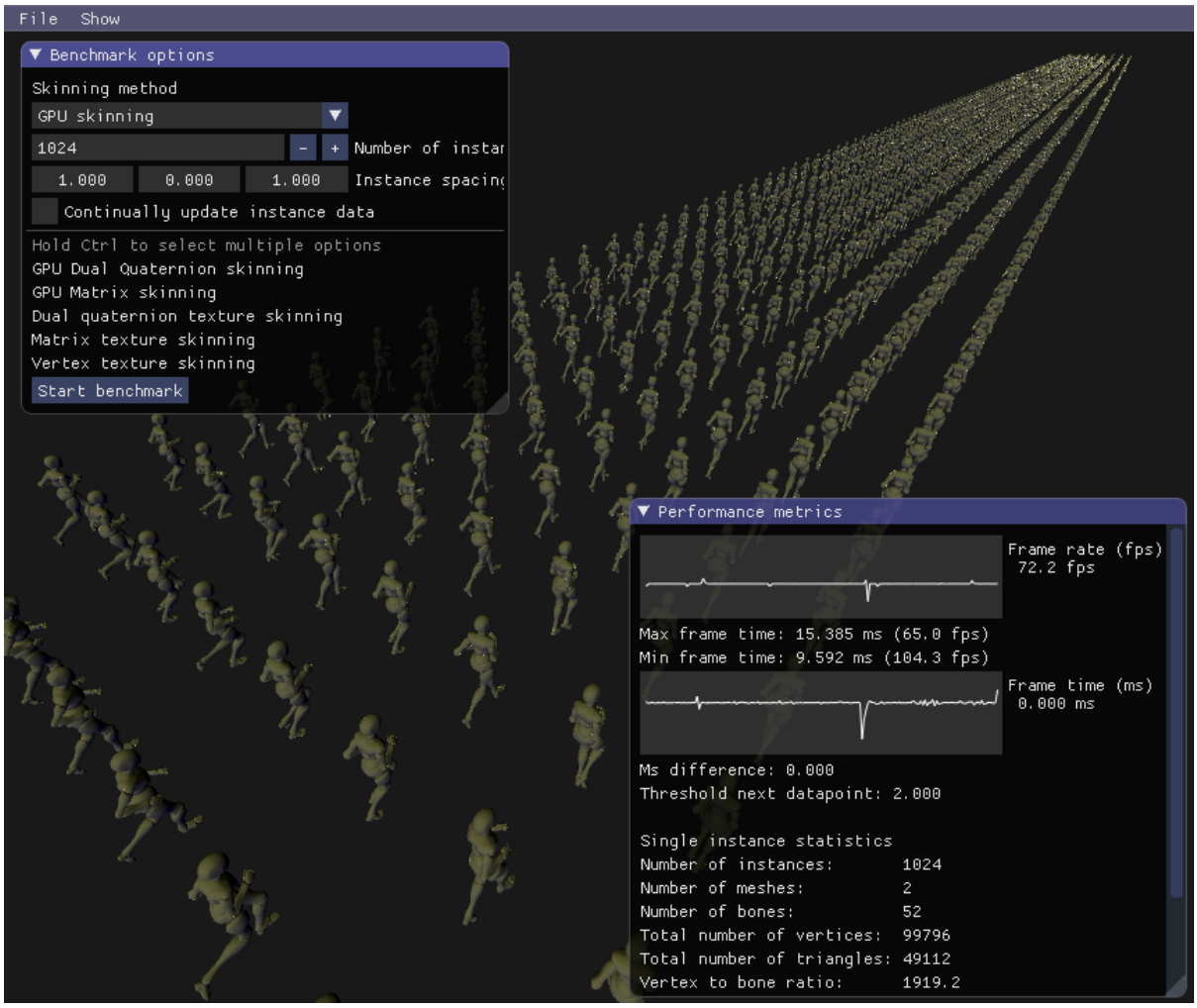


Figure 17: View of the benchmark mode of the application. Here 1024 instances of the X bot model is being rendered with the GPU skinning pipeline.

3.6.6 Instance Manager

The instance manager is a class used to control and manage the instances being rendered during the benchmark. The class contain three collections: positions, rotations, and animation offsets. They are used to update the model transformations of the instances being rendered, and controls the instance animation time. For the non-instanced rendered techniques (CPU and GPU skinning), the model matrix is being updated between each draw call. For the instanced meshes, the model matrices are updated per vertex through vertex attributes. An attribute divisor is used to tell OpenGL to update the attribute per instance versus per vertex. The benchmark uses the pre-sample clock before sampling any data to avoid instance generation having an impact on the performance results.

3.7 Model Previews

Models, rigs, and animations (with the exception of the Box model) were graciously provided by Mixamo[19].

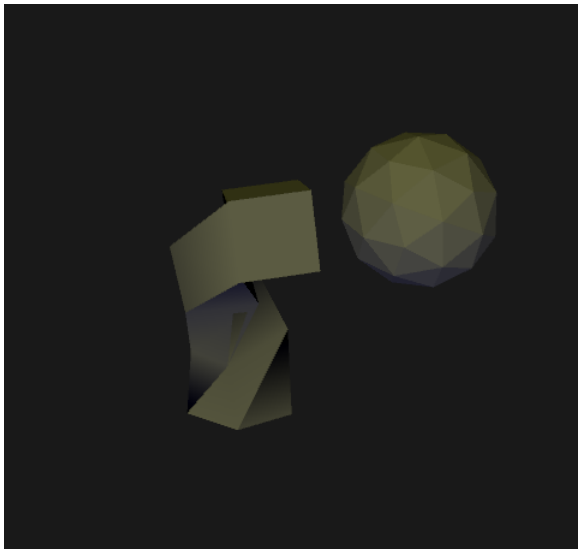


Figure 18: Box

Figure 19: Vampire

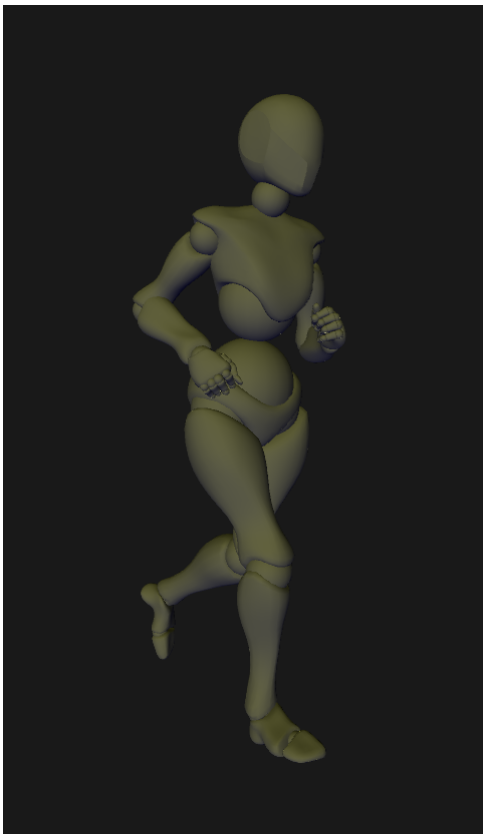


Figure 20: X Bot

Figure 21: Y Bot

4 Results and Discussion

In this section the results of the benchmarks are presented. Discussion and evaluation on the hypotheses presented in the introduction are also covered here.

4.1 Results overview

The results presented were produced from data gathered from the benchmark algorithm shown in 3.6.5. The testing was done on a build of the application with full optimization flags enabled in the compiler. No form of culling was performed on the models where every instance was skinned and rendered regardless of being visible in the view port of the application. The results are presented by model, each with tables showing the complexity of the mesh, skeleton, and the size of the animation texture (excluding texels needed to fill power-of-two requirement). The graphs represent the performance results of each skinning method in Frames Per Second (y-axis), per number of instances rendered (x-axis). The GPU skinning method is the only one not employing hardware instancing (See 2.5), where each mesh rendered induces a new draw call.

4.2 Box

Animation Name	Test Animation
Meshes	2
Total Vertices	312
Total Triangles	116
Bones	4
Vertex to Bone Ratio	78:1

Table 1: Model complexity of the Box model.

	Dual Quaternion Texture	**Matrix Texture**	**Vertex Texture**
Width	249	249	249×2
Height	10	14	42 + 20
Total Texels	2 490	3 486	15 438
Format	8-bit RGBA	8-bit RGBA	8-bit RGB

Table 2: Animation texture sizes of the Box model. Note: vertex textures need one additional texture for normals.

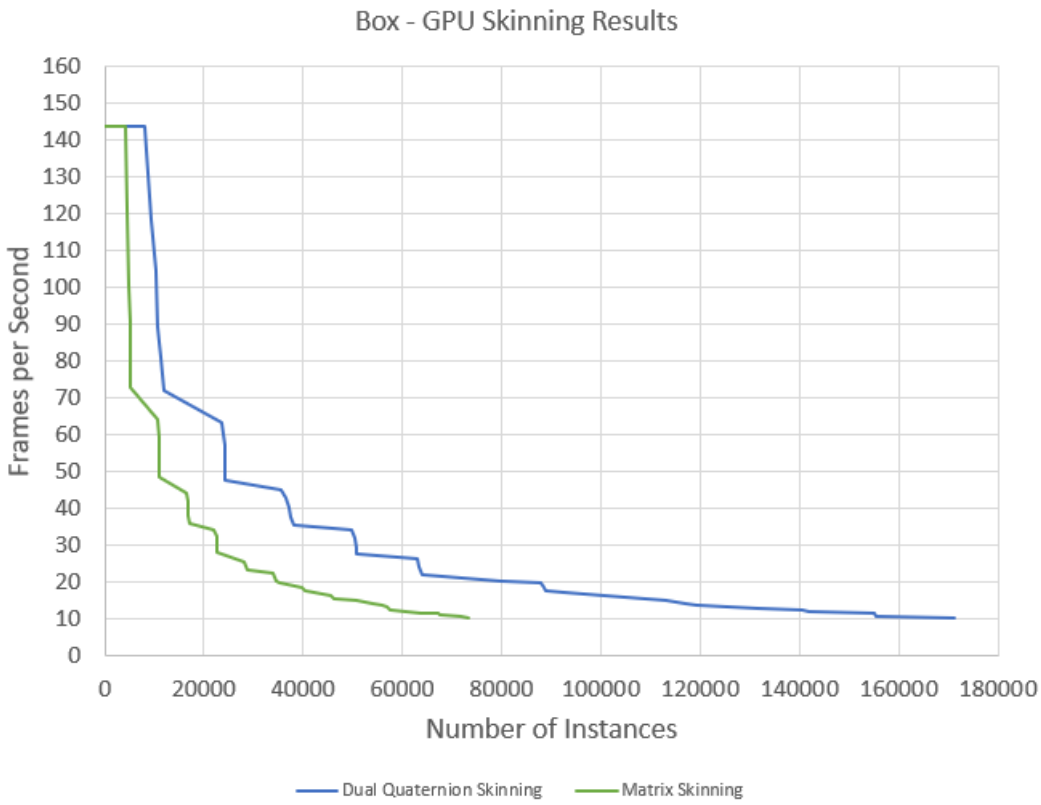


Figure 22: Box - GPU Skinning Performance

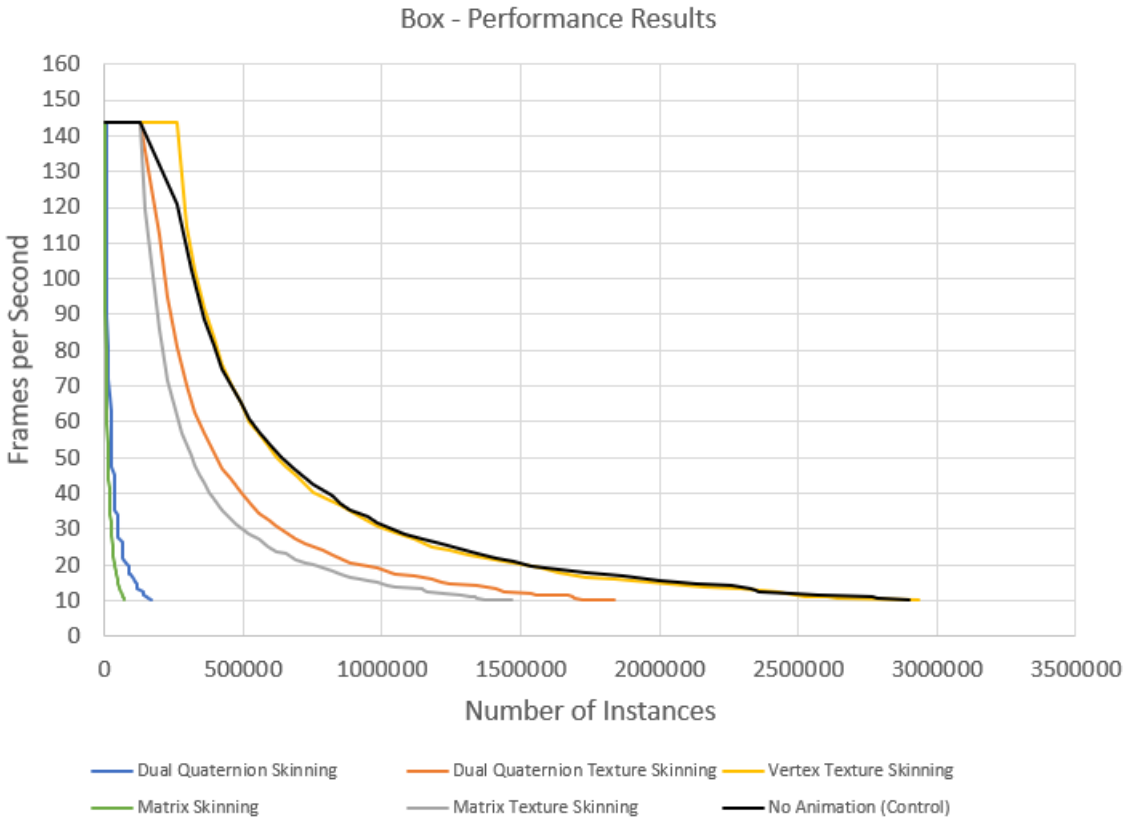


Figure 23: Box Performance

4.3 Vampire

Animation Name	Strut
Meshes	1
Total Vertices	45 066
Total Triangles	15 022
Bones	65
Vertex to Bone Ratio	693:1

Table 3: Model complexity of the Vampire model.

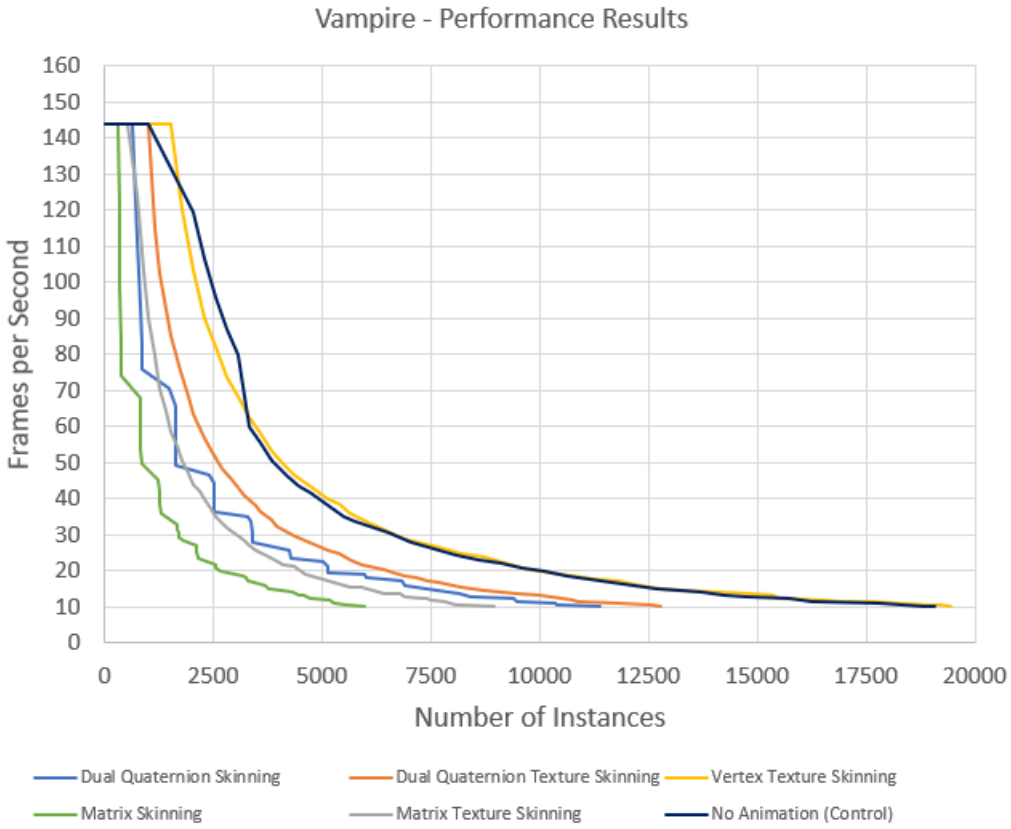


Figure 24: Vampire Performance Results

	Dual Quaternion Texture	**Matrix Texture**	**Vertex Texture**
Width	86	86	86
Height	132	197	7 870
Total Texels	11 352	16 942	676 820
Format	8-bit RGBA	8-bit RGBA	8-bit RGB

Table 4: Animation texture sizes of the Vampire model.

4.4 X Bot

Animation Name	Jogging
Meshes	2
Total Vertices	99 796
Total Triangles	49 112
Bones	52
Vertex to Bone Ratio	1919:1

Table 5: Model complexity of the X Bot model.

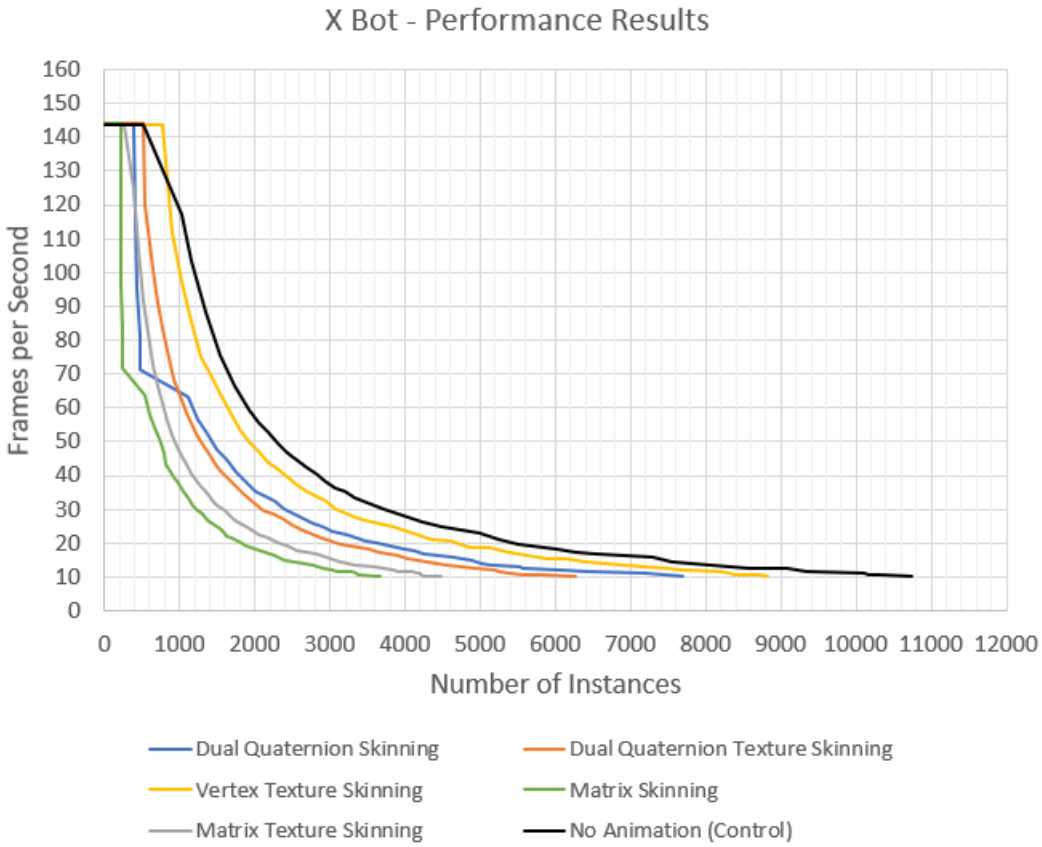


Figure 25: X Bot Performance Results

	Dual Quaternion Texture	**Matrix Texture**	**Vertex Texture**
Width	154	154	154 × 2
Height	106	158	14 232 + 10 514
Total Texels	16 324	24 332	3 810 884
Format	8-bit RGBA	8-bit RGBA	8-bit RGB

Table 6: Animation texture sizes of the X Bot model.

4.5 Y Bot

Animation Name	Hip-hop Dancing
Meshes	2
Total Vertices	319 944
Total Triangles	159 272
Bones	67
Vertex to Bone Ratio	4775:1

Table 7: Model complexity of the Y Bot model.

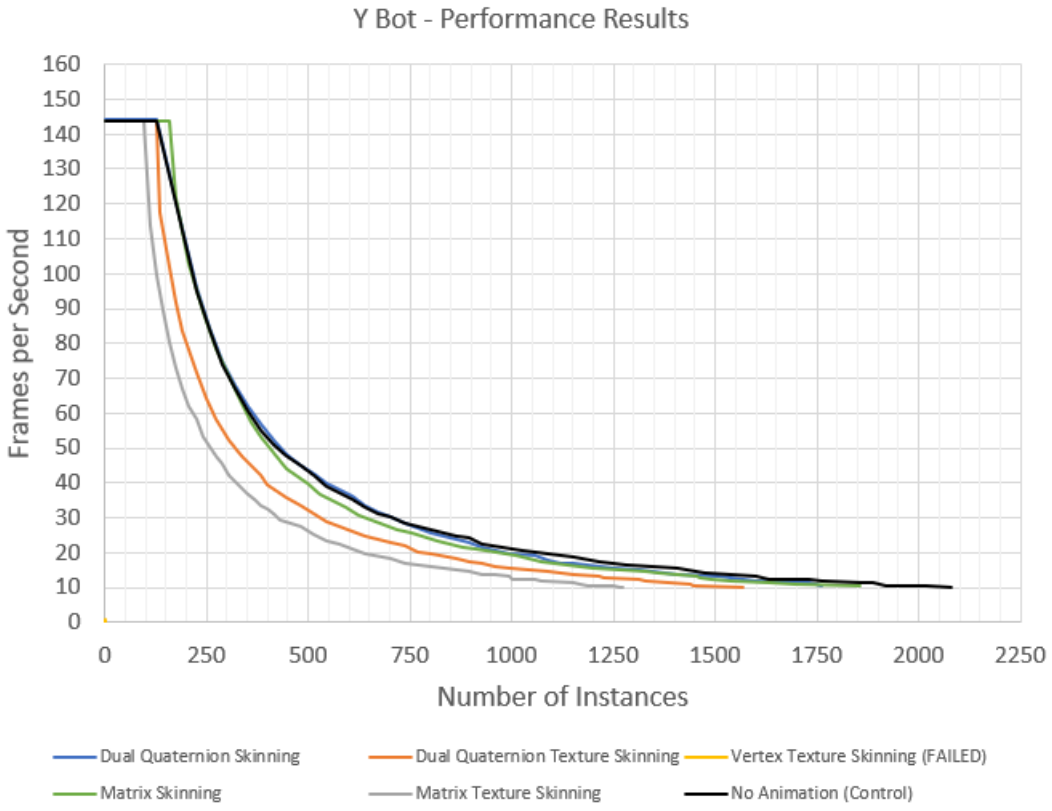


Figure 26: Y Bot Performance Results

	Dual Quaternion Texture	**Matrix Texture**	**Vertex Texture**
Width	132	132	N/A
Height	136	203	N/A
Total Texels	17 952	26 796	N/A
Format	8-bit RGBA	8-bit RGBA	N/A

Table 8: Animation texture sizes of the Y Bot model.

4.6 Performance Results

The performance aspect of animation textures is perhaps the most important property for considering them in a real world scenario. The main advantage comes from the possibility of using hardware instancing to render a large number of skinned meshes, potentially improving performance by limiting CPU - GPU communication. Before any results were known, a number of hypotheses were constructed based on preconceived expectations:

1. **Dual quaternion skinning will always outperform matrix skinning.**
   - Insinuates dual quaternions are always cheaper to compute compared to matrix transformations, in the context of skinning.
2. **Bone texture skinning will always outperform traditional GPU skinning.**
   - Implies the new technique of storing the skeleton bone transformations into textures to allow hardware instancing, is less expensive than traditional skinning methods.
3. **Vertex texture skinning will always outperform bone texture skinning.**

To test them, a benchmark was used measure the relative performance between animation textures and traditional skinning methods (See 3.6.5).

4.6.1 Relative Performance Breakpoints

Table 9 holds data of the represented rendering techniques at different frames per second break points.

	Box			Vampire		
FPS	**120**	**60**	**30**	**120**	**60**	**30**
Control	100%	100%	100%	100%	100%	100%
GPU DQ	3.5%	4.5%	5.1%	38%	49%	52%
GPU Matrix	1.8%	2.1%	2.3%	16%	25%	26%
DQ Texture	75%	63%	63%	56%	62%	67%
Matrix Texture	56%	50%	48%	38%	46%	47%
Vertex Texture	113%	100%	100%	88%	99%	102%

Table 9: Box and Vampire models performance breakpoints. Relative number of instances rendered compared to control/no animation. The vertex texture skinning in some instances outperforms the control as can also be perceived in figure 23. This is an artifact coming from the linear interpolation between sample points.

	X Bot			Y Bot		
FPS	**120**	**60**	**30**	**120**	**60**	**30**
Control	100%	100%	100%	100%	100%	100%
GPU DQ	41%	58%	64%	100%	100%	100%
GPU Matrix	21%	32%	33%	92%	95%	89%
DQ Texture	53%	57%	57%	71%	77%	77%
Matrix Texture	38%	40%	41%	58%	59%	61%
Vertex Texture	88%	87%	83%	0%	0%	0%

Table 10: X Bot and Y Bot models performance breakpoints. Relative number of instances rendered compared to control/no animation.

4.7 Hypothesis 1 (1.1)

4.7.1 Dual Quaternion VS. Matrix Skinning

In 3/4 of the performance benchmarks, the dual quaternion implementations appeared to outperform their matrix counterpart, with the exception being the Y bot model where the results did not appear to sufficiently deviate. The students t-test can be used to determine if the means of two data sets are significantly different

from each other. To use the t-test, we need a metric to compare between the two benchmarks. FPS, and number of instances are only representative of performance when paired together, which is why a combined unit; **Peformance Rating**(defined as the FPS $\times$ number of instances), is applied in these tests. The t-test also assumes the samples (fps / number of instances) follows a normal distribution, which is the assumption in this thesis.

The box model is the most simplistic, both in terms of vertex- and bone count. To test one part of the first hypothesis, we create a null-hypothesis: *The mean difference in performance between GPU dual quaternion skinning and GPU matrix skinning is not statistically significant.* Using the t-test for the two traditional GPU skinning methods with a statistical significance threshold $\alpha = 0.05$ we get:

	Dual Quaternions	**Matrices**
Mean $\bar{X}$	1.44E6	6.49E5
Variance σ^2	1.18E11	2.20E10
Samples n	39	38
Degrees of Freedom $d.f.$	52	
t-stat t	13.19	
P($T \leq t$) two-tail p	$3.25E-18$	

Table 11: t-test of the Box model comparing GPU dual quaternion skinning to GPU matrix skinning, assuming the variances are unequal.

Because $3.25E-18 < \alpha$ we can reject the null-hypothesis, and say that the performance difference between the two methods are statistically significant *for the Box model*. This method is used to test every dual quaternion technique versus the corresponding matrix implementation, to confirm or disprove hypothesis 1. In table 12 the initial row represents the relative mean difference, displaying the relative performance disparity between the two data structures. The percentage gives the method with the highest mean performance rating a positive score for dual quaternions, and a negative for matrices, meaning 0% represents the results were equal.

From the t-tests, we can see the null-hypothesis can be rejected for every instance, except for GPU skinning in the Y Bot model. However, in this instance both techniques are also not deviating from the control benchmark. This suggests

		Box	Vampire	X Bot	Y Bot
GPU DQ VS Matrix	**R.M.D.**	122%	98.9%	97.9%	4.62%
	P score	3.25E-18	3.87E-17	2.12E-17	0.24
DQ- VS Matrix textures	**R.M.D.**	25.1%	45.6%	36.4%	23.5%
	P score	7.49E-08	9.75E-16	2.93E-13	5.16E-08

Table 12: T-tests checking if the mean Performance Rating is statistically significant ($\alpha < 0.05$) for dual quaternion skinning versus matrix skinning.
Relative Mean Difference (R.M.D.) = $\bar{X}_{dq}/\bar{X}_m$ where $\bar{X}_{dq}$ is the mean performance rating for dual quaternion skinning, and $\bar{X}_m$ is the mean performance rating for matrix skinning. A positive percentage means dual quaternions had a higher mean performance rating.

that the skinning computation was insignificant to the results, and was not the main performance bottleneck in that specific instance. Taking this into account, we can confirm hypothesis 1, for the hardware architecture specified in chapter 3.

Recalling that to represent dual quaternions, one requires 8 floating point numbers vs. the 12 in transformation matrices (4x3). During the deformation step, a vertex needs 24 multiplications and 21 additions for dual quaternions, but only 12 multiplications and 8 additions for the 4x3 matrices. It is important to note that the shader samples 3 bones per vertex, which all need to be weighted by their influence, thus increasing the number of operations of the dual quaternions to $24+8\cdot3 = 48$ multiplications and $21+8\cdot3 = 45$ additions, compared to $12+12\cdot3 = 48$ multiplications and $8 + 12 \cdot 3 = 44$ additions for the matrices. Further is each bone multiplied by the link transform, to get the correct vertex transformation. Combining two dual quaternion requires 48 multiplications and 40 additions. To combine a 4x3 matrix, 48 multiplications and 36 additions are needed. This results in dual quaternions doing $48 + 48 \cdot 3 = 192$ multiplications and $45 + 40 \cdot 3 = 165$ additions, compared to $48+48\cdot3 = 192$ multiplications and $44+36\cdot3 = 152$ additions. Theoretically, this should mean the matrix transformations are slightly cheaper to compute, which conflicts with the performance results. This may be implementation dependent, as a simple loop comparing 2^{20} dual quaternion multiplications to 2^{20} matrix multiplications gave an identical computation latency.

To understand the GPU skinning bottlenecks better, simple A/B testing were conducted where each feature configuration was sampled. There were three points of interest in the rendering pipeline that could affect the performance:

- CPU skeleton computation.
- Uploading the skeleton to the GPU.
- Switching VAO for each mesh per instance.

The Box model was used, as it had the lowest GPU skinning performance score, relative to the other techniques. All features were enabled at the start of the test, until 60 000 instances were reached - constituting in 30 FPS. Continuing, the features were individually disabled, and the performance rating is sampled. This process is repeated until all combinations have been exhausted.

Skeleton Computation	**Skeleton GPU Upload**	**VAO Switch**	**P.R. Dual Quaternions**	Relative %	**P.R. Matrices**	Relative %
ON	ON	ON	1 800 000	100%	960 000	100%
ON	OFF	ON	1 920 000	107%	1 020 000	106%
OFF	ON	ON	2 460 000	137%	2 460 000	256%
OFF	OFF	ON	2 760 000	153%	2 760 000	288%
ON	ON	OFF	960 000	53%	540 000	56%
ON	OFF	OFF	1 020 000	57%	540 000	56%
OFF	ON	OFF	3 720 000	207%	3 660 000	381%
OFF	OFF	OFF	4 320 000	240%	4 320 000	450%
Control (No Animation)			31 200 000	1 733%	31 200 000	3 250%

Table 13: A/B Testing of the GPU dual quaternion skinning at 60 000 instances. The performance rating is equal FPS × number of instances. The Control is taken max instances at 30 FPS. Data is sampled and averaged over 120 simulation steps/frames.

The samples are averaged over 120 simulation steps. The results can give some insight into the cost of each step during the GPU skinning pipeline. The first interesting detail is that the skeleton computation is the most expensive operation, less so for dual quaternions versus matrices. When disabling this feature, both techniques achieved equal performance. The problem may lay in the recursive nature of the skeleton computation algorithm, where the additional memory footprint of the matrices is responsible for the performance loss during the skeleton graph traversal.

The skeleton upload step appears to be relatively inexpensive, while its performance impact does seem to scale with the amount of data dispatched to the GPU. When the switch was disabled, the skeleton data was required to be uploaded twice, once for each mesh. This was presumably the main factor of the performance boost seen in the last configuration. The upload step also appears to favor dual quaternions over matrices, likely coinciding with the 50% size discrepancy of the two data

structures.

Another noteworthy finding is that VAO switching has a significant cost associated with it. The explanation is the skeletons must be computed n times, where n is equal to the number of meshes the 3D model is composed of. For the Box model, $n = 2$ such that the skeleton is computed twice per instance, when VAO switching is disabled. When skeleton computation is disabled, a relative performance increase of 50% is observed. To reduce the impact of skeleton computations, caching the skeleton, using multi-threaded/parallel solutions, or converting the algorithm from a recursive- to an iterative nature may help significantly. This implementation initially did not use either, but a caching paradigm was implemented to confirm the performance benefit. The result of caching the skeleton transformations resulted in the performance rating rising to 2 040 000 (113% relative) for dual quaternions and 1 980 000 (206% relative) for matrices. The instance model matrices which are needed to position the instances in the scene was also disabled together with the skeleton computation step. When these were disabled and combined with skeleton transform caching, the result represented the missing performance gains from the previous result of disabling skeletal computations.

The final insight to be discussed, is the case when all features are disabled. This configuration produced results orders of magnitude worse compared to the performance results of the control test. This configuration corresponds to drawing the same VAO without changing the internal rendering state. This in theory should be equivalent to hardware instancing, according to the OpenGL specification[8]. This appears to highlight the fixed cost of draw calls, as the GPU pipeline executes $2 \cdot 60\ 000 = 120\ 000$ draw commands to the GPU each simulation step, compared to just 2 in the hardware instanced methods. This fact was discovered through the use of the Nvidia Nsight Graphics debugging tool, and may be vendor/architecture dependent.

4.8 Hypothesis 2 (1.2)

The next claim that Bone texture skinning will always outperform traditional GPU skinning, needs only one conflicting t-test to reject the hypothesis. Looking at the Y

Bot model performance graph, a significant gap can be perceived between the bone texture methods versus the rest. By testing if the performance difference between dual quaternion texture skinning and GPU dual quaternion skinning is significant, we can reject hypothesis 2:

	GPU Dual Quaternion Skinning	**Dual Quaternion Texture Skinning**
Mean $\bar{X}$	20 012.14	15 579.11
Variance σ^2	1.08E7	6.47E6
Samples n	43	40
Degrees of Freedom $d.f.$	78	
t-stat t	6.89	
$\mathbf{P}(T \leq t)$ **two-tail** p	$1.25E-9$	

Table 14: t-test of the T Bot model comparing GPU dual quaternion skinning to dual quaternion texture skinning, assuming the variances are unequal.

$1.25E-9$ is below α, and we can conclude that the the performance advantage GPU skinning exhibits over bone texture skinning are statistically significant, which means we can reject hypothesis 2.

Looking at the performance graph of the Y Bot Model, the bone texture pipeline performs significantly worse than all other techniques (ignoring vertex texture skinning). This model is the most complex in terms of number of vertices. To better understand the intricacies of the bone texture pipeline, we can map out the differences it has versus vertex texture skinning:

- Bone textures need to access more texels per vertex.
- Bone textures need to perform deformation calculations/skinning.

From these differences, three stages present in the bone texture pipeline is explored: *Metadata texel reads* - each bone needs to decode the AABB to correctly decode the bone transformation, *Bone interpolation* - Needed for smooth motion in-between sample frames, and *Skinning* - The act of actually deforming the vertices with the decoded bone transformation. Another A/B test can then be used to explore which operations are the most computationally expensive:

Metadata Texel Reads	**Bone Interpolation**	**Skinning**	**P.R. Dual Quaternions**	Relative %	**P.R. Matrices**	Relative %
ON	ON	ON	15 900	100%	13 780	100%
ON	OFF	ON	18 020	113%	16 960	123%
OFF	ON	ON	16 430	103%	15 370	112%
OFF	OFF	ON	16 960	107%	17 490	127%
ON	ON	OFF	16 430	103%	14 310	104%
ON	OFF	OFF	17 490	110%	17 490	127%
OFF	ON	OFF	16 960	107%	15 370	112%
OFF	OFF	OFF	18 550	117%	18 550	135%
Skip 1/3 Bones			19 610	123%	19 610	142%
Skip 2/3 Bones			20 140	127%	20 140	146%
Control (No Animation)			21 730	137%	21 730	158%

Table 15: A/B Testing of the GPU dual quaternion skinning at 530 Y Bot instances. The performance rating is equal FPS × number of instances. The Control is taken max instances at 30 FPS. Data is sampled and averaged over 120 simulation steps/frames.

From the data, bone interpolation seems to be the most expensive step. Interpolation doubles the number of texture reads, as two samples are an inherit requirement for interpolation procedures. Disabling this feature gave the biggest performance gain, especially for matrix textures. The implementation computes the two weighted deformation transformations, and then interpolates between them for the final transform. By instead reading and interpolating the decoded texels first, and then weighting the interpolated transforms, a 6% performance increase was observed for dual quaternions, and 12 % for matrices. The amount of texel reads remains the same, except in this procedure the texels are accessed in closer proximity to each other. This is beneficial as the GPU contains a texture cache as seen in figure 5, which makes subsequent reads located close together less expensive[7].

The texture metadata reads represent $2 + 6 \cdot 3 = 20$ texel reads for dual quaternions and $2 + 9 \cdot 3 = 29$ for matrices. As they are all located in the first column, they exhibit some data locality, and can be reasonably cached by GPU. Because all metadata is constant, a potential to "hard-code" the values into the shader is possible, at the disadvantage of having to recompile the shader for different animation textures. This approach may also work for link transformations, as these also remain constant. Link transformations are currently provided through uniform variables in the current implementation.

Skinning the vertices with the decoded deformation transformations are comparatively cheap to the operations involving texture reads. Both dual quaternion-

and matrix skinning appear to be relatively similar in cost.

Because every vertex can be influenced by three bones at a time, an additional test was done to see potential performance gains of skipping one and/or two bones during the bone transformation decoding. This would effectively reduce texture reads by 33% and 66% respectively (ignoring metadata). From the result, one could deduce this process is likely responsible for the last performance difference versus the control performance score. Individually, 3D models may require a different amount of deformation bones to correctly deform vertices. This is a constant metric that could also be encoded as metadata, or defined during shader compilation, to further optimize this solution.

4.9 Hypothesis 3 (1.3)

From the performance results, one can already conclude that indeed, vertex texture skinning are less expensive than bone texture skinning from the initial results. As vertex textures only need one texel read per vertex (excluding metadata), and performs no skinning operations, the relative performance ratings are to be expected. The biggest problem with vertex textures are the size needed for complex models. In this implementation every vertex is represented in the rows of the texture. When the meshes are complex, the required texture size becomes considerably large. In the case of the Y Bot model, it exceeded the maximum texture size set by the OpenGL API, failing completely. Constructing textures whereby a vertex is not inherently locked to a specific row, but rather a texel offset, one could remedy this problem considerably, and reduce wasted pixels introduced by the power-of-two rule. Animation compression could further be used to limit the amount of texels used to encode low frequency information, a technique previously showcased by Jonas Norberg[25].

When considering more cache coherent texel fetches for the bone texture implementation, along with no vertex duplication (mentioned in 4.13), the performance disparity between vertex textures and bone textures narrows considerably. In fact, only the box model had a vertex texture performance rating that was statistically significant, relative to the bone textures. In no circumstances did the bone texture pipeline perform significantly better than vertex textures. Based on the newfound

information discovered in 4.13, hypothesis three can be confirmed at sufficiently simplistic meshes, with low vertex counts.

4.10 Texture Precision

The implementation constructed every animation texture with 8-bits of precision per channel, which is the default precision format for most image files. This means that the values represented in the textures can represent $2^8 = 256$ steps. To use the space most effectively, AABB's are used to define the bounding space for which the 256 steps are defined. In terms of animation quality, vertex texture skinning does exhibit precision artifacts for large and complex meshes, as every vertex shares one AABB. Further, when limiting the sampling rate to 6Hz, a noticeable "bobbing" artifact appears, originating from linearly interpolating rotational motions. Matrix bone textures need to be specified in world space, then multiplied by the link-transformation to get the final deformation transform in the shader. This is done because encoding the deformation transforms directly, greatly diminishes the effectiveness of the AABB's, as they are no longer united in a common coordinate system. When world transformations are used in matrix textures, they become indistinguishable from the original animation, at sufficient sampling rates. At 6Hz, matrix bone textures also exhibit the same visual artifact as vertex textures. This could be solved by deforming the vertex by each bone transform, and then using spherical linear interpolation to get the final correct vertex position.

The same strategy is used for dual quaternions, but they do not exhibit the same "bobbing" artifact at low sampling rates, *without utilizing spherical linear interpolation*. The reason for this advantage is the normalization step at the end of the final transformation computation. Looking at figure 34, the magnitude of the rotational real part of the dual quaternion is always equal 1. At higher sampling rates, the final animation does look noticeably imprecise, where motions appear jittery in comparison to the ground truth, even after enforcing world space encoding. Because the dual part is a factor of the real part of the complete dual quaternion, the translation data does not exist in Cartesian coordinates, diminishing the value of AABB's in this implementation. Encoding the raw position in the texture, and computing the final dual quaternion in the shader instead, could remedy the precision artifact

observed, and improve visual quality. This is possible because the translation dual part is constructed through the rotational real part ($q^d = \frac{1}{2}(0, t_x, t_y, t_z) \times q^r$, where q^d is the dual part, t is the translation vector, and q^r is the real part)[16]. Additionally, the current encoded dual part is constructed with a real part of higher precision than the one encoded in the shader. The resulting decoded dual quaternion may not have corresponding real and dual parts, possibly exacerbating the artifacts that can be seen in the final animation. Another consideration is using textures with a higher precision format, however this may increase texel sampling latency.

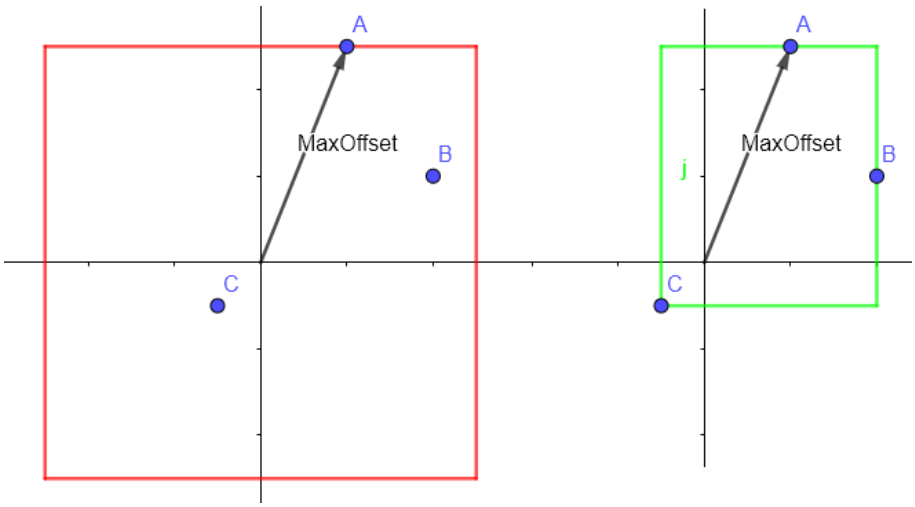


Figure 27: Benefits of using an Axis Aligned Bounding Box, where the limited precision is utilized more efficiently.

The fall project included an additional approach for vertex texture encoding which uses four components (x, y, z, w) instead of three. This version have been used previously in [25] and [6]. The vector direction is encoded in the first three components, and the normalized magnitude in the fourth. This gives us an angular precision of $p(v) = (\frac{90}{128}) = 0.703^{\circ}$, and a scalar precision of $p(d) = \frac{x}{256}$ where x is the maximum deviation distance. Other than using one additional byte for precision, this method encodes the vertices using discrete parametric rays instead of Cartesian coordinates. The four component method encodes the vectors in spherical hulls, which is not useful for bounding boxes. Tri-axial ellipsoids could be substituted for AABB's, which is essentially a sphere defined by three different orthogonal radii. An additional two values must be encoded for the normalized magnitudes for each of the ellipsoid axes, increasing the needed texels from four to six. If precision becomes a problem this approach may be helpful.

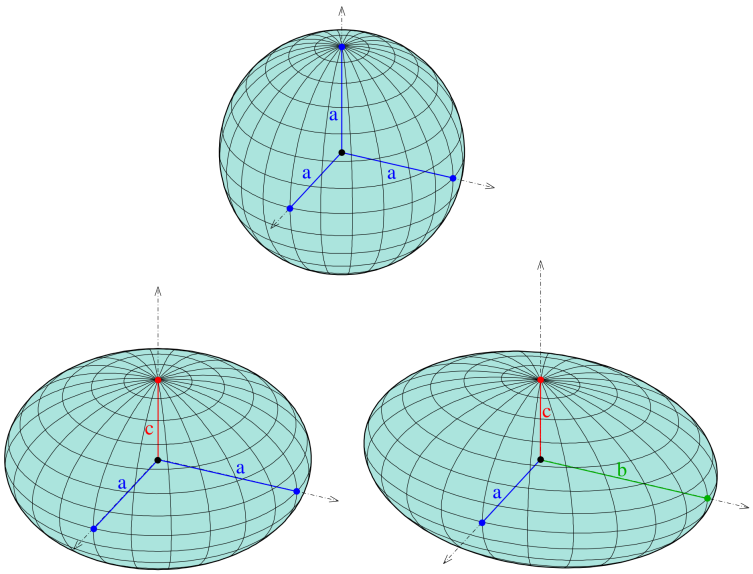


Figure 28: Examples of ellipsoids[1]. Tri-axial ellipsoid is at the bottom left, where each axis have a unique magnitude.

4.11 Vertex Textures Versus Bone Textures

Vertex texture skinning can be superior to bone texture skinning in terms of performance, as concluded in 4.9. The main advantage of bone textures is their independence of the model itself. Bone textures encode the skeleton, which means it's size is fixed, regardless of the mesh complexity. This also makes it easier to animate models consisting of multiple meshes, as each mesh can share the same animation texture. Comparatively, vertex textures need unique vertex- and normal textures for each mesh, while the texture size directly correlates to the mesh complexity.

4.11.1 Texture Space efficiency

The vertices are duplicated in the VAO, however using an additional vertex attribute makes it possible to point to the correct index in the original model, and eliminates duplicated texels in the animation texture.

Every vertex in the model is represented in each time step/key frame of the animation texture, even if the next sample point is identical to the previous one. The presentation at Unite 2016[25] improved this issue by using animation compression. It worked by splitting the texture into predetermined time intervals, as seen in figure 29. This allowed them to individually compress each vertex at each time

region, where vertices with low frequency deviations would be compressed the most. This was achieved by encoding time information using one additional texture that holds the correct offset in the time dimension. This essentially means that each time region has its own sampling frequency. Finally, to maximize texture space usage, they allowed vertex segments to reside on the same row, reducing wasted space considerably. This meant the shader needed additional information to know the start index for each vertex, as the rows no longer corresponded to the vertex indices. The same approach should be possible with bone textures, where low frequency changes in the bone transformations can be represented with fewer texels.

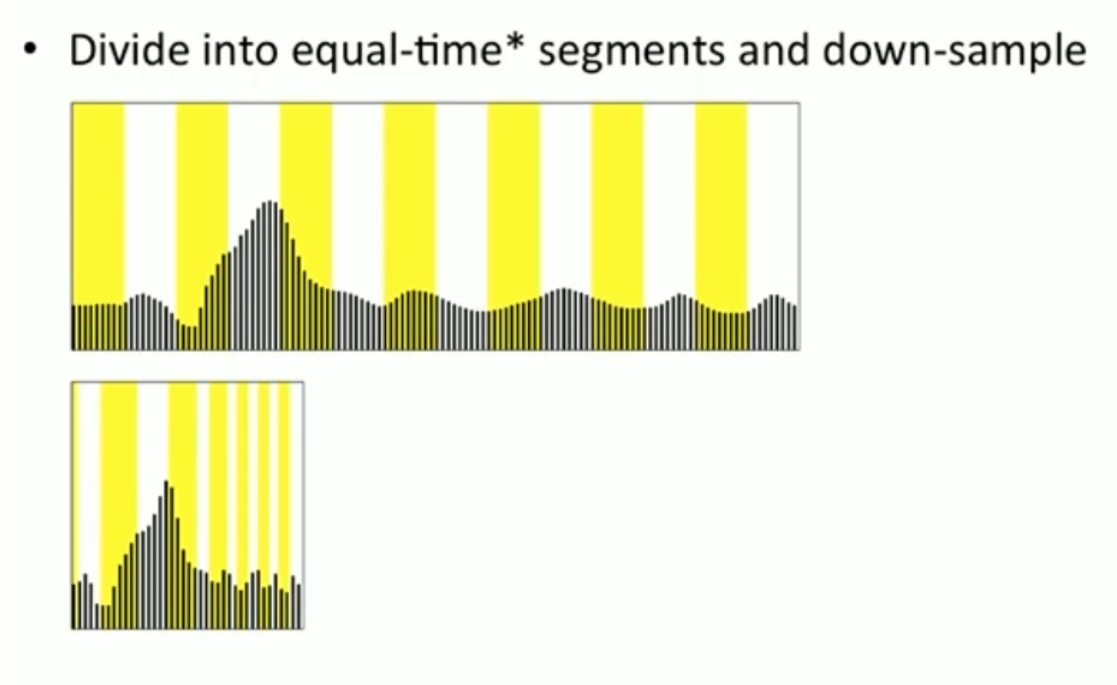


Figure 29: Animation compression excerpt taken from the Unite 2016[25] presentation

4.12 Animation Texture Advantages and Disadvantages

The biggest advantage of using animation textures is the ability to play skinned animations with hardware instancing. Within certain criteria where the model complexity is low, the performance advantage can be orders of magnitude higher, compared to traditional skinning methods, as concluded in 4.8. This was mainly the result from reducing draw calls and much of the overhead cost from CPU-GPU communication. Because of it's instancing nature, every model are effectively clones of each other. In crowd simulation where animation textures can be greatly advanta-

geous, model variability if often desired. This means additional efforts need to be employed to reduce cloning artifacts. Building models consisting of multiple independent but interchangeable meshes like in [20] can be used to great effect. If every mesh utilizes a mutual skeleton structure, the bone textures can be shared between all models.

Because the animation data is baked into textures, they cannot be easily modified. Techniques that require procedural animation like inverse kinematics, and rag dolls, may prove difficult to implement, versus a traditional approach where the skeleton resides on the main memory, and can easily accessed and modified by the CPU.

4.13 Duplicated Vertices

One consideration that likely have severely impacted the benchmarks in a negative manner, is the duplicated vertices from the VAO generation step. Vertices were originally duplicated as to support flat shading, where each vertex could contain a unique normal for each polygon it was a part of. As an example, the X Bot model consisted of 24 746 vertices as of importation, which got transformed to 99 796 vertices after VAO conversion. That results in a duplication rate of $\frac{99796}{24746} = 4.03$. Because the number of texel fetches is proportional to the amount of vertices, the performance results were most likely affected. A second benchmark were performed which removed all vertex duplicates, to properly test the impact a reduced vertex count has on the performance. For this benchmark, the traditional GPU implementations were using a cached skeleton structure, skipping the real-time skinning step for each instance. The bone texture pipeline were also utilizing the more cache coherent texel fetches in the shader, discussed in 1.2.

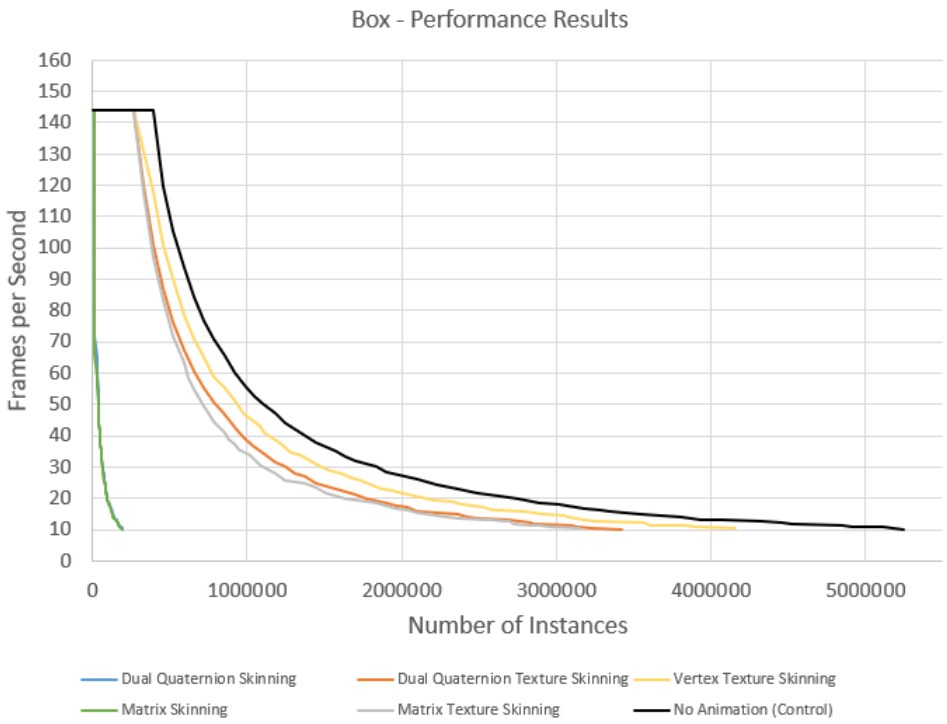


Figure 30: Box Performance - No vertex duplicates. 5.03 × less vertices.

Figure 31: Vampire Performance - No vertex duplicates. 5.73 × less vertices.

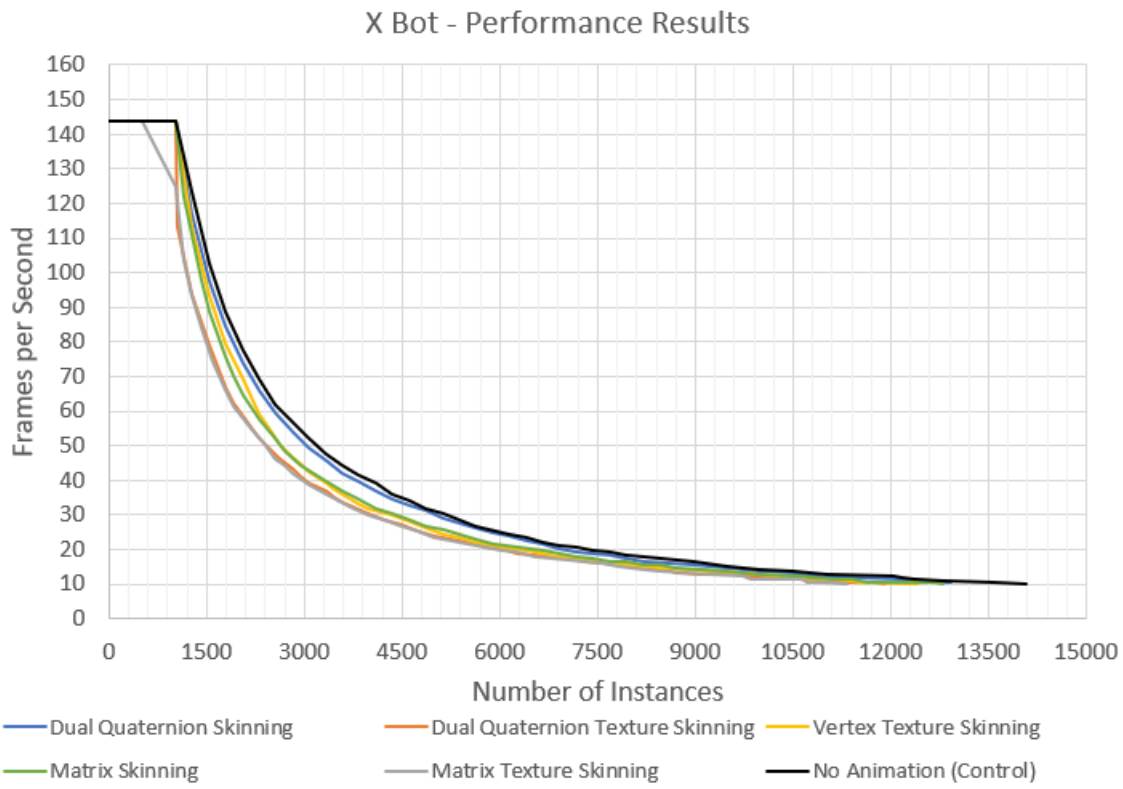


Figure 32: X Bot Performance - No vertex duplicates. 4.03 × less vertices.

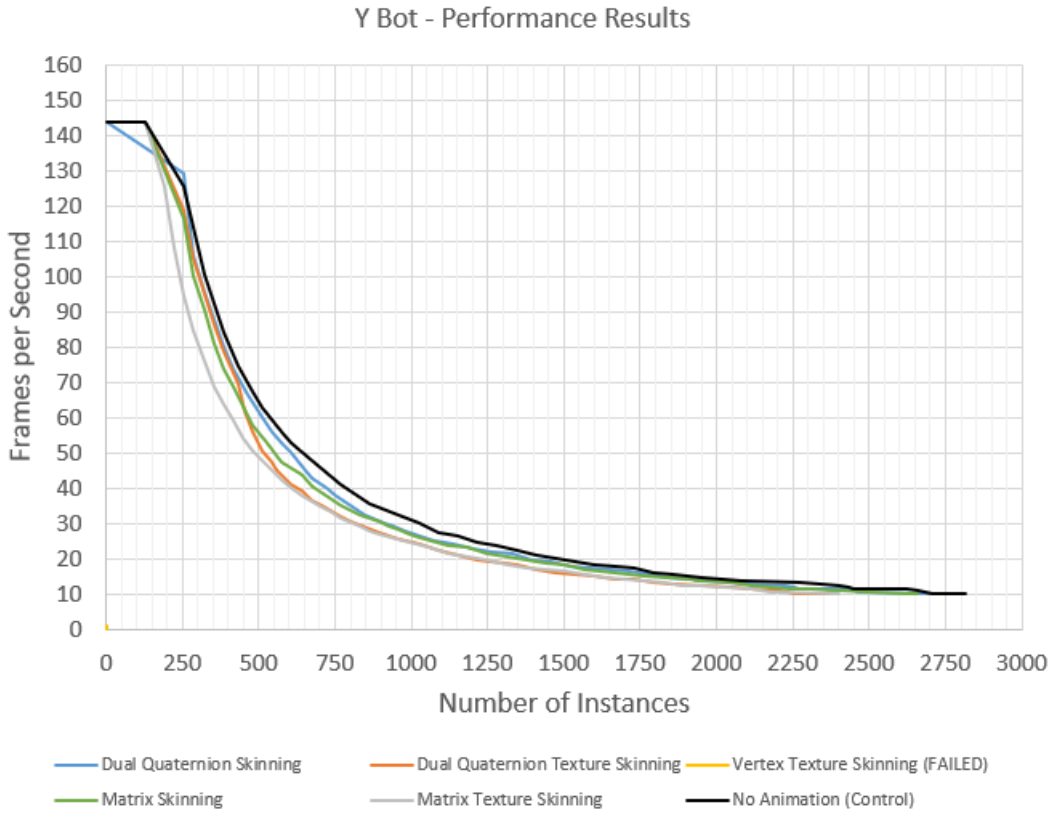


Figure 33: Y Bot Performance - No vertex duplicates. 4.01 × less vertices.

From looking at the new performance graphs, every model achieved a higher performance score compared to their previous result. Looking at the control benchmark, an estimated 100-35% (from lowest- to highest model complexity) performance gain can be attributed from the reduced vertex count. More notably, the relative difference between the different skinning techniques has narrowed considerably. An argument can be made to say this benchmark is a more candid comparison than the first. This is due to the GPU pipeline being limited to using cached versions of the skeleton hierarchy, thus operating on the same grounds as the bone texture pipeline. Based on these findings we can conclude further, that the main advantage of texture based approaches is the reduction of draw calls. Rendering simple models need considerably more draw calls to reach the same number of triangles, compared to a complex model.

		Box	**Vampire**	**X Bot**	**Y Bot**
GPU DQ VS Matrix	**R.M.D.**	4.26%	12.6%	9.31%	4.90%
	P score	0.40	0.0044	0.016	0.18
DQ- VS Matrix textures	**R.M.D.**	7.02%	33.8%	3.20%	4.79%
	P score	0.08	9.00E-11	0.39	0.23

Table 16: T-tests checking if the mean Performance Rating is statistically significant ($\alpha < 0.05$) for dual quaternion skinning versus matrix skinning.
Relative Mean Difference (R.M.D.) = $\bar{X}_{dq}/\bar{X}_m$ where $\bar{X}_{dq}$ is the mean performance rating for dual quaternion skinning, and $\bar{X}_m$ is the mean performance rating for matrix skinning. A positive percentage means dual quaternions had a higher mean performance rating.

From the t-tests of the new benchmarks, we can still confirm that dual quaternion skinning do perform better than matrix skinning, albeit less indisputable. This also speaks to the importance of cache coherent texture fetches, as the difference between dual quaternion texture skinning and the equivalent matrix technique have narrowed considerably, even though matrices need 50% more texels to represent a full transformation. One interesting tidbit is the dual quaternion textures seemingly favoring the Vampire model, which consists of a single mesh. Both the dual quaternion texture and the matrix texture are the same size, due to the power-of-two rule as can be seen in figure 13.

4.14 Ease of Implementation

The application supports importing the proprietary fbx format. A majority of popular 3D applications supports exporting 3D animations with the fbx file extension, as these files are relatively common. To be able to generate vertex textures, only the final transformed vertices need to be known. Bone textures in contrast, require knowledge of the underlying skeleton structure to work. If only considering using vertex textures, using the fbx format is unnecessary. Another open source file format; Alembic, only contains baked data, which would constitute "post-skinned" vertices in regards to animations[2]. This could facilitate a simpler algorithm that can ignore the CPU skinning step altogether.

A consideration is the storage of the generated animation textures. This implementation always generates the animation textures on demand, which takes time to compute. Writing the textures to a permanent storage would alleviate this issue, and allow for exportation to other applications. The lossless PNG image format is one possibility. For a potential program to be able to use the produced animation texture, it would need the capability to import PNG files. As the PNG format is a common standard for encoding image data, implementing such functionality should be fairly straight forward.

To be able to deform a model with an animation texture, the host program needs access to the vertex shader in some form. Some programs used in the 3D industry may not have direct access to the programmable shader stages, but rather present them through abstracted forms as materials. These approaches are also usually accompanied with a node system to simplify shader/material creation. The open source 3D application, Blender, is an example which presents shader creation in this form[21]. As long as the application offers some form of a vertex offset, animation texture skinning should be possible.

The final effort needed to use the animation texture is actually writing/constructing the animation texture shader. The shader needs to correctly decode the animation texture which depends on the sampling technique used. Constants like sampling frequency, animation length, and Axis Aligned Bounding Box definitions also need to be present in order to correctly decode the texels.

5 Conclusions

5.1 Conclusion

This thesis has explored the concept of encoding animation data into textures, which have been mainly evaluated in terms of performance, while accuracy, and memory footprint of the tested implementations were also discussed. For evaluation, three hypotheses were presented, based on intuition and previous knowledge. In terms of hypothesis 1 (1.1), dual quaternions always achieved a higher performance score relative to the corresponding matrix implementation, but the difference was not statistically significant for every tested scenario. The thesis concluded the size difference of the two data structures was the main reason for the performance discrepancy, as both traditional GPU skinning and bone texture skinning need to access 50% less data in the case of dual quaternions compared to matrices. For hypothesis 2 (1.2), the traditional GPU skinning technique achieved equal or even surpassed the performance of bone texture skinning in some instances, thus disproving the hypothesis. Bone texture skinning produced superior performance results by orders of magnitude when rendering simple models. The reduced vertex count allowed the number of draw calls to become significant in regards to traditional GPU skinning, which the bone texture pipeline avoided through the advantage of hardware instancing. The last hypothesis were initially confirmed for all scenarios where the texture size was not a problem. After some further testing, texel fetch optimizations was implemented through improving texture cache coherence. Continuing by removing duplicate vertices, the performance advantage of the vertex texture skinning technique disappeared in every benchmark, except for the simplest box model. Hypothesis 3 was therefore confirmed for simple meshes consisting of few vertices. This also highlighted the importance of data locality, as to better utilize the texture cache.

The thesis further discussed the advantages of bone textures being independent of the meshes utilizing them. The size of bone textures is dependent on the number of bones the skeleton structure is composed of, where the generated texture can be shared between different meshes. In contrast, vertex textures need two unique textures per mesh (vertex, and normal) which size is directly proportional to the

mesh complexity.

5.2 Further Work

This section explores possible improvements and techniques that can improve the results discussed in the results.

5.2.1 New Encoding Schemes

One potential avenue is new methods for encoding vertex positions. This thesis used Cartesian coordinates to encode positions in space, and raw rotational data with 8-bits of precision. The fall project included an additional encoding scheme for vertex textures, using parametric rays. Here the x, y, and z components represent the normalized direction vector, while w is the normalized magnitude. This could potentially be used in conjunction with tri-axial ellipsoids (see figure 28) for a potential bounding volume solution, where the maximum axis magnitude is included in the shader.

Because the real part of a dual quaternion represents a unit-quaternion, its length must equal 1. This characteristic can allow for reconstructing one of the components in the shader, reducing the amount of encoded values required down to three. This combined with utilizing the world space position of the bone, instead of the actual dual part (as discussed in 4.10), only six components would need to be encoded when considering dual quaternions. The two extra values could be exploited to encode data necessary for compression, like time information as seen in Jonas Norberg's implementation[25], or simply be ignored to reduce the memory footprint by 25%.

5.2.2 Animation Blending

One technique often used in game development is blending multiple animations together to form a new pose, which does not explicitly exist in the sampled animation[12]. Animation blending can allow for smooth procedural transitions between different animations, such as from walking to running. The same benefits of ani-

mation blending could potentially be applied to skinned animation textures as well, where the blending operations take place in vertex shader instead of being computed by the CPU.

5.2.3 Testing Other Architectures

The results presented in this thesis are dependent on the hardware architecture used. A proposed next step is to conduct equivalent tests for systems of different hardware configurations. Lower powered architectures, like phones, and embedded systems may exhibit different performance traits than those found in this thesis.

5.2.4 Animation Compression and Artifact Heuristics

The application provides a 1:1 animation preview of the ground truth input together with the generated texture animations. The visualization is important, as it helps highlight errors and artifacts in the encoded animations. Measures and statistics like FPS and texture size are also given in the interface to give some insight into how the given parameters affect output. The possibility for a factual measurement of the error/deviation of the texture animation versus the ground truth, would be useful to more effectively give a better insight where additional samples may be needed. The aliasing effect visible in the matrix texture skinning, and vertex skinning techniques, originated from linearly interpolating between samples where rotational context were not taken into account. This artifact is not apparent in dual quaternions, as they are normalized prior to vertex deformation step.

Consider figure 34 where a bone modeled as a vector rotates 90° between two sample points; A and B. An accurate interpolation between the two samples would generate points along a circular arc, with center equal the bone origin. Linear interpolation rather produces points in a straight line resulting in an inconsistent magnitude along the path. This inconsistency could be used to construct a heuristic to estimate aliasing between samples, for example by subtracting the area of the interpolated path (triangle) from the area of the circular arc (circle). The heuristic could be shown to the user and/or be used as a cost in the sampling algorithm to determine an optimal sampling frequency to achieve high texture space efficiency,

while minimizing the aliasing effect. This heuristic may be more useful for matrix bone texture generation, as vertex textures are not directly skinning vertices in the shader. Directly using the distance between the initial and deformed vertex position would likely be a superior heuristic in this case.

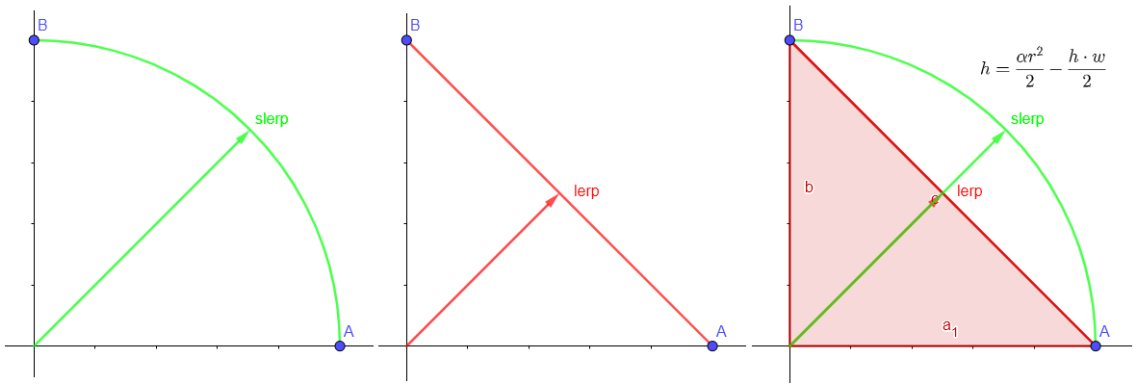


Figure 34: Difference between spherical linear interpolation and linear interpolation.

Spherical linear interpolation (slerp) is a technique that accurately interpolates directional vectors along a circular path, defined by this equation:

$$p' = p_1 \frac{\sin[(1-t)\Omega]}{\sin(\Omega)} + p_2 \frac{\sin(t\Omega)}{\sin(\Omega)} \tag{7}$$

Where $0 \leq t \leq 1$ is the interpolation parameter, and Ω is the angle subtended by the arc. This formula can be utilized with matrix texture skinning, where a vertex needs to be deformed by two deformation matrices, where the final vertex position is the product of using slerp on the two deformed vertices. This could significantly reduce the aliasing effect produced by low sample counts when considering matrix textures.

References

[1] Ag2gaeh. *Ellipsoid - Wikipedia*. [Online; accessed 12. Jun. 2021]. June 2021. URL: https://en.wikipedia.org/w/index.php?title=Ellipsoid&oldid=1027558780.

[2] *Alembic*. [Online; accessed 6. Dec. 2020]. July 2016. URL: http://www.alembic.io.

[3] Golam Ashraf and Junyu Zhou. "Hardware Accelerated Skin Deformation for Animated Crowds". In: *Advances in Multimedia Modeling*. Springer, Berlin, Heidelberg, Jan. 2007, pp. 226–237. ISBN: 978-3-540-69428-1. DOI: 10.1007/978-3-540-69429-8_23. URL: https://link.springer.com/chapter/10.1007/978-3-540-69429-8_23.

[4] Norman Badler and Stephen Smoliar. "Digital Representations of Human Movement". In: *ACM Comput. Surv.* 11 (Mar. 1979), pp. 19–38. DOI: 10.1145/356757.356760.

[5] William Kingdon Clifford. *Mathematical papers, 1845-1879*. [Online; accessed 19. May 2021]. May 2021. URL: https://archive.org/details/mathematicalpap00smitgoog.

[6] GameDaily Connect. *4,000 Adams at 90 Frames Per Second | Yi Fei Boon*. [Online; accessed 2. Dec. 2020]. May 2017. URL: https://www.youtube.com/watch?v=rXqKu9uC0f4.

[7] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Boston, MA, USA: Addison-Wesley Professional, Mar. 2004. ISBN: 978-0-32122832-1. URL: https://www.amazon.com/GPU-Gems-Programming-Techniques-Real-Time/dp/0321228324.

[8] *glDrawElementsInstanced - OpenGL 4 Reference Pages*. [Online; accessed 18. May 2021]. Apr. 2021. URL: https://khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawElementsInstanced.xhtml.

[9] *GLSL Object - OpenGL Wiki*. [Online; accessed 18. May 2021]. Jan. 2021. URL: https://www.khronos.org/opengl/wiki/GLSL_Object.

[10] Sir William Rowan Hamilton. *Elements of Quaternions, 1805-1865*. [Online; accessed 19. May 2021]. May 2021. URL: https://archive.org/details/elementsofquater00hamirich.

[11] Alec Jacobson et al. "Skinning: Real-time Shape Deformation". In: *ACM SIGGRAPH 2014 Courses*. 2014.

[12] Daniel Jeppsson. "Realtime character animation blending using weighted skeleton hierarchies". PhD thesis. Institutionen för datavetenskap, Lunds tekniska högskola, 2000.

[13] Marcus Benjamin Johansson. *Exploring GPU Animation Textures*. Dec. 2020.

[14] Ladislav Kavan and Jiri Zara. "Real-Time Skin Deformation with Bones Blending". In: (2003). [Online; accessed 13. Sep. 2020]. URL: https://www.cs.utah.edu/~ladislav/kavan03real/kavan03real.html.

[15] Ladislav Kavan et al. “Skinning with Dual Quaternions”. In: *2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM Press, May 2007, pp. 39–46. URL: https://www.cs.utah.edu/~ladislav/kavan07skinning/kavan07skinning.html.

[16] Ben Kenwright. “A Beginners Guide to Dual-Quaternions: What They Are, How They Work, and How to Use Them for 3D Character Hierarchies”. In: *Semantic Scholar* (2012). URL: https://www.semanticscholar.org/paper/A-Beginners-Guide-to-Dual-Quaternions%3A-What-They-to-Kenwright/5f560a48d89cadfd9ff04f41c942ebbbf0fea35a.

[17] Jeff Lander. “Skin Them Bones: Game Programming for the Web Generation”. In: *Game Developer Magazine* (May 1998). URL: http://www.darwin3d.com/gamedev/articles/col0598.pdf.

[18] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. “Joint-dependent local deformations for hand animation and object grasping”. In: 1989.

[19] *Mixamo*. [Online; accessed 22. May 2021]. Apr. 2021. URL: https://www.mixamo.com/#.

[20] Isaac Rudomin et al. “GPU Generation of Large Varied Animated Crowds”. In: *Revista Computación y Sistemas; Vol. 17 No.3* (Sept. 2013). ISSN: 1405-5546. URL: https://repositoriodigital.ipn.mx/handle/123456789/17229.

[21] *Shader Editor — Blender Manual*. [Online; accessed 12. Jun. 2021]. June 2021. URL: https://docs.blender.org/manual/en/latest/editors/shader_editor.html.

[22] Student. “The Probable Error of a Mean”. In: *Biometrika* 6.1 (Mar. 1908), pp. 1–25. ISSN: 0006-3444. URL: http://www.jstor.org/stable/2331554.

[23] T. Theoharis et al. *Graphics and Visualization: Principles & Algorithms*. A K Peters/CRC Press, Oct. 2007. ISBN: 978-1-56881274-8. URL: https://www.amazon.com/Graphics-Visualization-Principles-Algorithms-Theoharis/dp/1568812744.

[24] *Uniform (GLSL) - OpenGL Wiki*. [Online; accessed 18. May 2021]. Jan. 2021. URL: https://www.khronos.org/opengl/wiki/Uniform_(GLSL).

[25] Unity. *Unite 2016 - Rendering a Large Number of Animated Characters Using the GPU*. [Online; accessed 1. Dec. 2020]. Dec. 2016. URL: https://www.youtube.com/watch?v=1ZPcXcCBFIs.

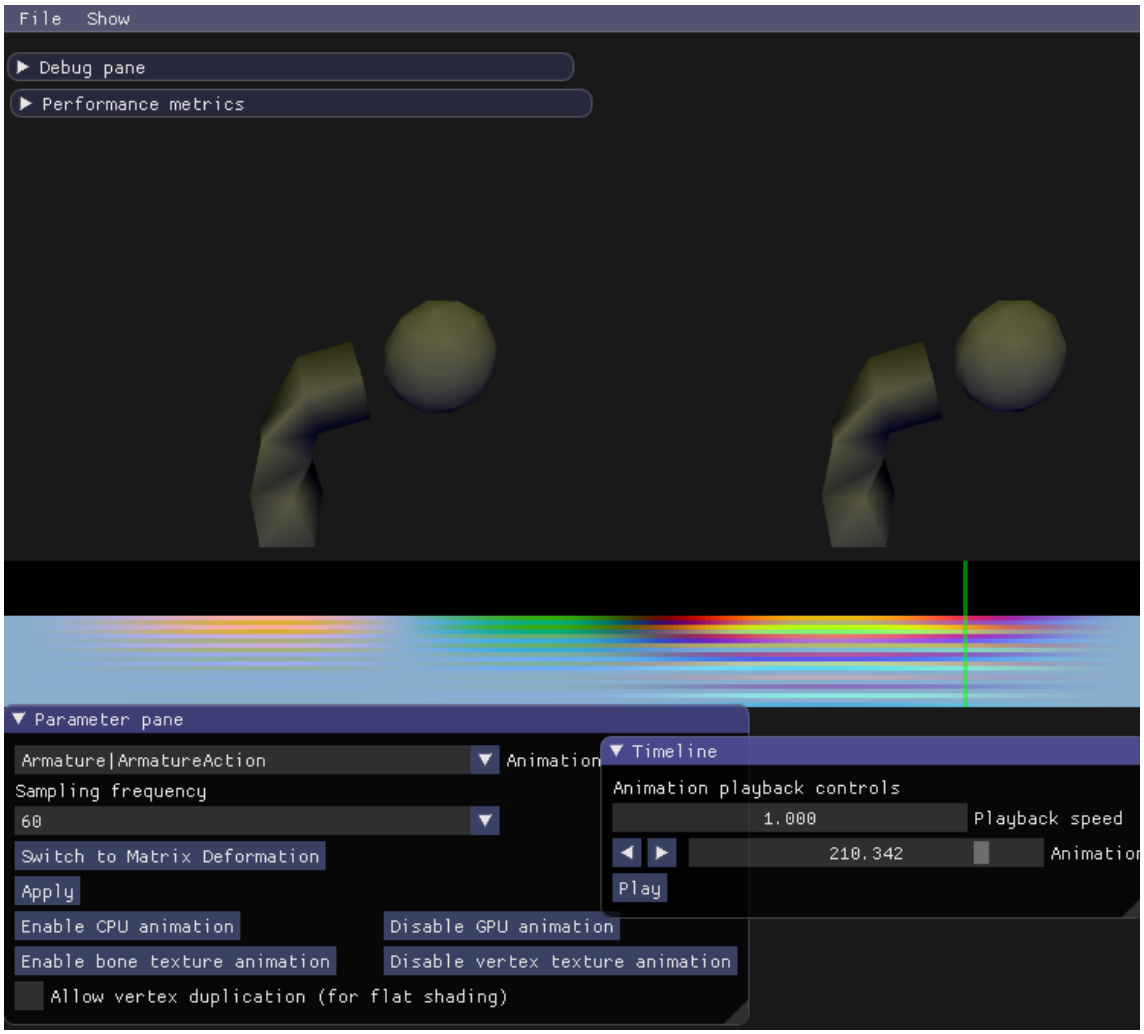


Figure 35: Application window

Appendix

A User Interface

The animations in the application are rendered in a re-sizable window. The application is using the GLFW library for OpenGL development, to more easily achieve cross-platform support (although, the FBX SDK currently used only supports windows). The application window can be divided up into to four view-ports, each displaying a different skinning technique as seen in figure 36. These include the CPU skinning, GPU skinning, bone-texture skinning, and vertex texture skinning pipelines. The scale of the imported models are normalized, such that they are always visible when playing the animation.

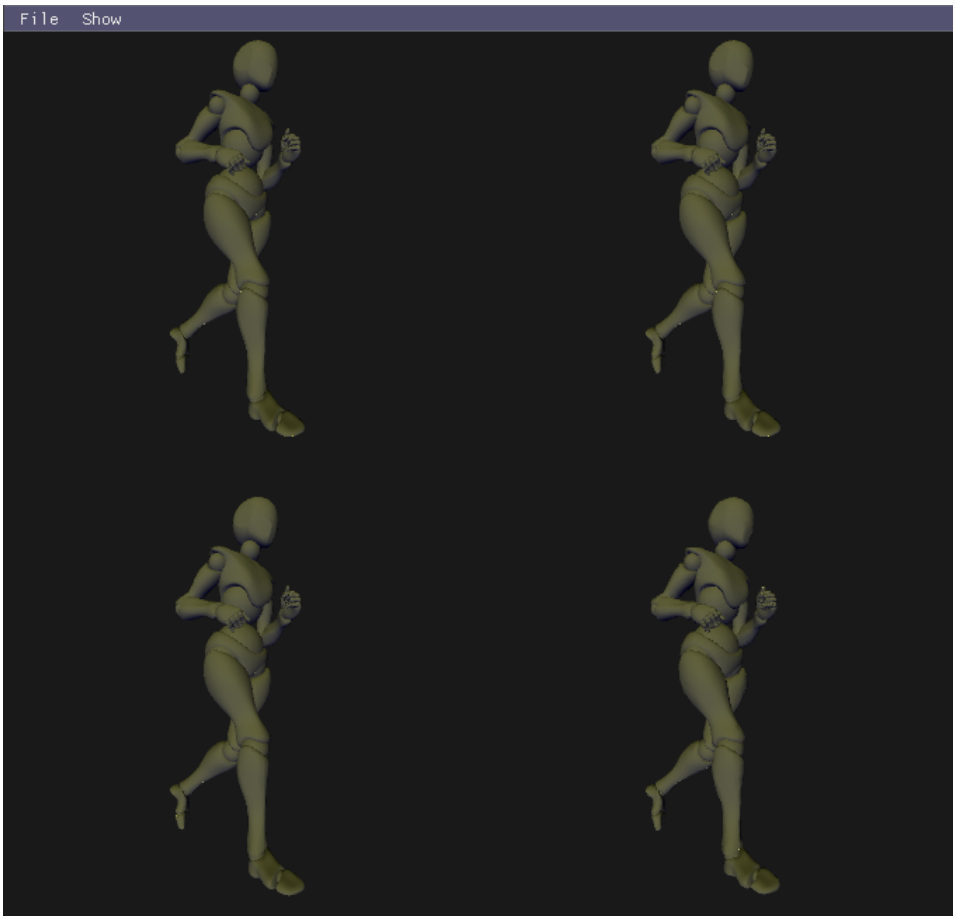


Figure 36: Comparison-mode view port of the application. **Top Left:** CPU skinning **Top Right:** GPU skinning **Bottom Left:** Bone texture skinning **Bottom Right:** Vertex texture Skinning.

An additional quad is also rendered in the window, behind the animating models. Its job is to visualize the generated textures, helping in analyzing and debugging the encoding process with different parameters. The quad has the same dimensions as the visualized animation texture, such that the texture is displayed without stretching- and or compression artifacts. A green vertical line is also rendered on top of the texture, acting like a play head pointing to the texels currently being read by the shader.

A.1 DearImGUI

To draw the user interface, the C++ library, DearImGUI, is used to render a simple Graphical User Interface which is used to both adjust animation sampling parameters, as well as other controls helping the user to inspect the final output animation.

The library allows for creation of collapsible, movable, and re-sizable panels which can be fitted with a range of multiple different widgets. The application is separated into two distinct modes: comparison-mode, and benchmark-mode. In the comparison-mode, the GUI is separated into five distinct panels that can be shown or hidden from the 'Show' file menu. They include: the parameter pane, the timeline, texture statistics, performance metrics, and a debug pane.

The parameter pane provides a combo-box used to change the animation being played, followed by another combo-box to modify the animation sampling frequency, ranging from 6 Hz to 120 Hz. Below them is a button to apply the chosen parameters, for which the animation textures will be re-sampled. Next, there exists four buttons for toggling the rendering of the different skinning techniques. Lastly, is a checkbox for toggling enforcing no vertex duplication.

On the timeline pane, controls for tuning the animation playback is given. The first controls the animation playback speed, the second contains a drag-able slider representing the current animation frame, and the button at the bottom gives the user the ability to pause and play the animation playback. The arrow buttons, step one key-frame forwards or backwards in the encoded animations.

Texture statistics are shown on its respective pane. It contains a combo-box to chose which texture to be visualized, along with a check-box to enable or disable the visualizing texture quad. Shown below them, are the respective statistics of the chosen texture.

The performance metrics is found on the largest pane, first displaying two graphs of FPS and frame time respectively. Further, is two values only important during benchmarks; Ms difference from last sample, and the frame time difference threshold needed for next sample point. Below them, are statistics about the number of instances being rendered, useful for estimating performance impact.

The last pane is used for debugging, first giving an integer slider where a bone in the skeleton can be indexed. From the selected bone, the transformation can be viewed (either matrices or dual quaternions). The transformation can be shown with or without link-transformations applied, specified by the button below the integer slider. The debug vector is used to enable or disable features in the application

and shaders, to help with debugging. Next is a check-box to visualize the skeleton with red lines when rendering with the traditional GPU skinning method. Next is visualizing the AABB used when encoding the bone transform into textures, and the last is changing the visualized AABB to the one used by vertex textures instead.

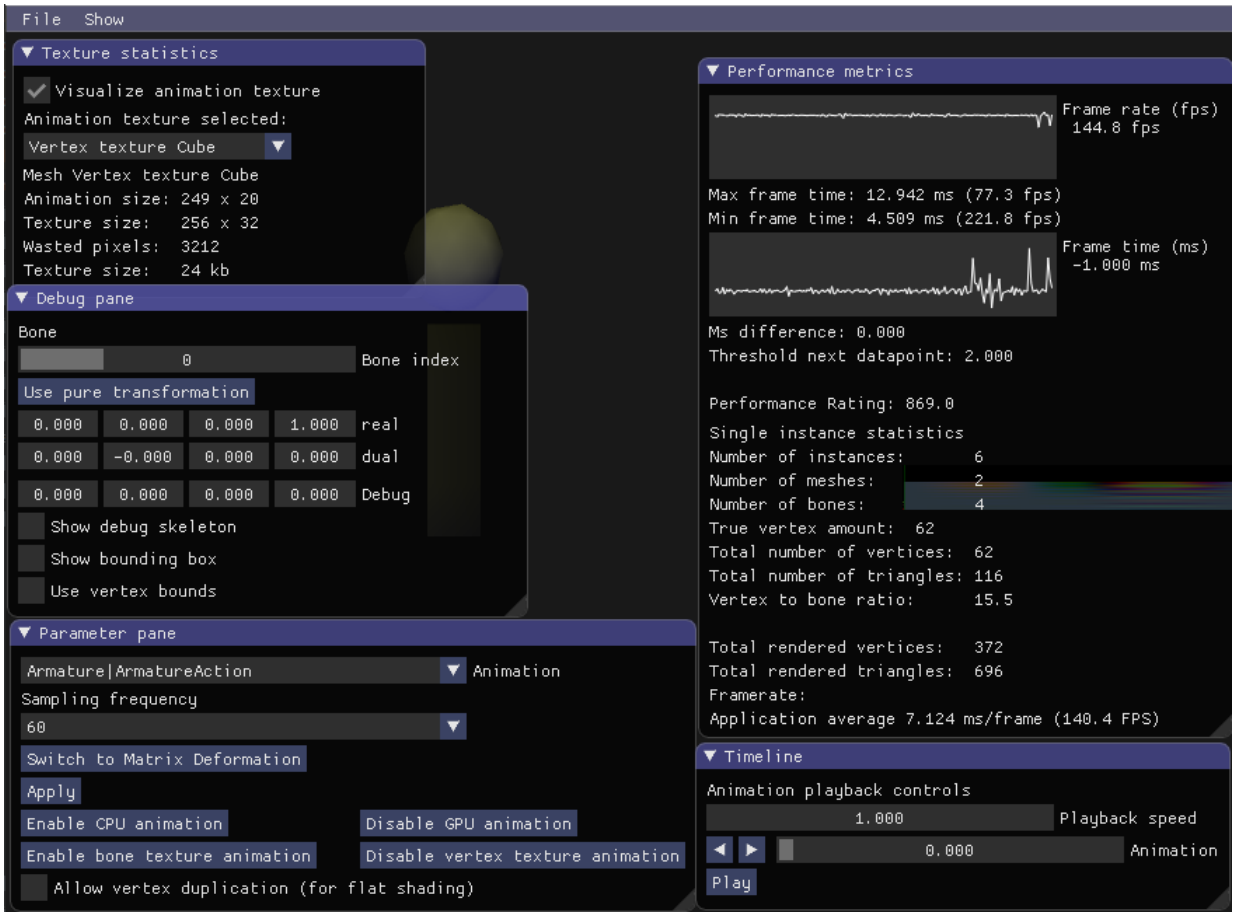


Figure 37: Demonstration of the DearImGui panes used in the comparison-mode in the application.

In the benchmark-mode accessed in the file menu, the benchmark panel becomes visible. The first combo-box selects the current skinning method, ranging from CPU skinning, GPU skinning, bone texture skinning, vertex texture skinning, an Control (no skinning). Next is the number of instances rendered. Then the spacing between the rendered instances, before a check-box to continually update the instance data each simulation step. Next is a multi-select list, specifying which methods are to be tested in the benchmark. Finally, the benchmark is initiated by the "Start benchmark" button.

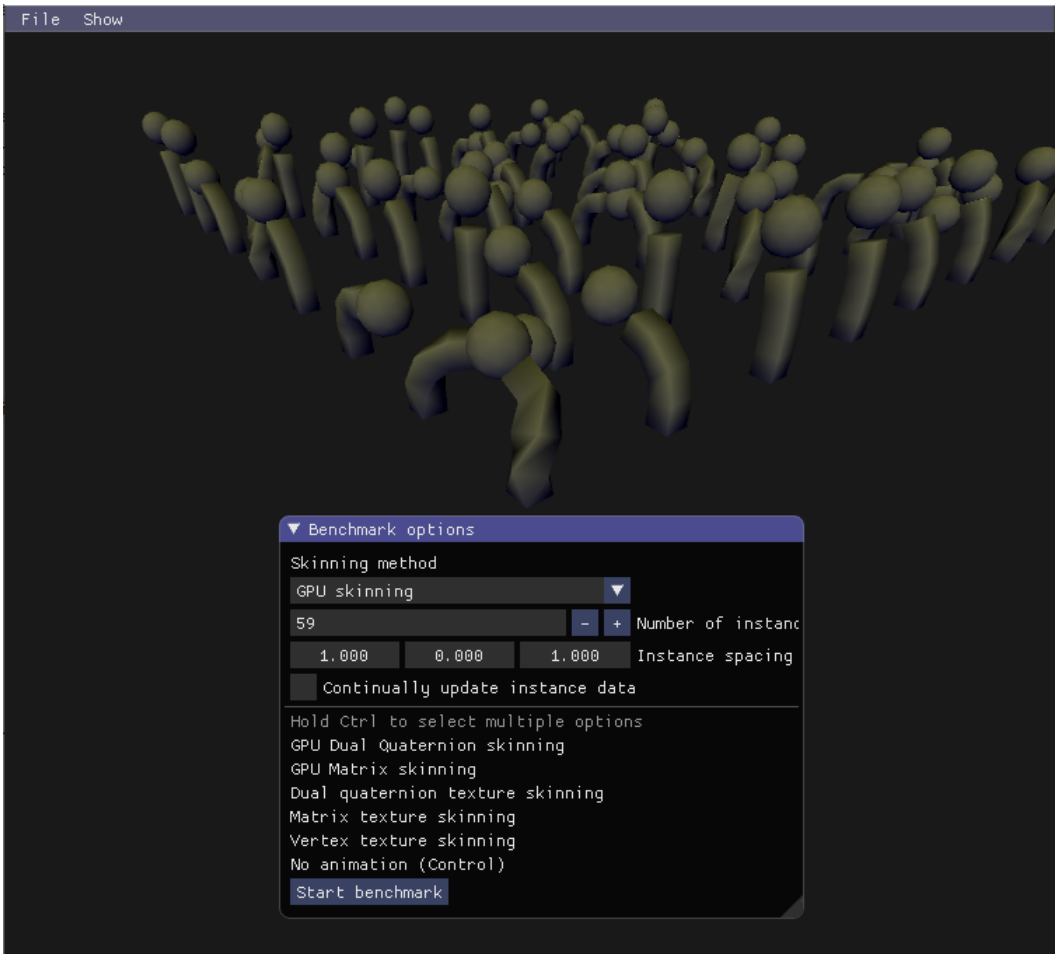


Figure 38: Benchmark-mode panel.

A.2 Other Controls

Keyboard Controls

Rotate view port	Right mouse button + drag
Translate camera vertical	Middle mouse button + drag
Zoom view port	Scroll wheel
Translate texture visualizer	Ctrl + middle mouse button + drag
Zoom texture visualizer	Ctrl + Scroll wheel
W, A, S, D	Translate camera forward, left, backward, right
E, Q	Translate camera up, down
Reset camera position	R
Skip one key frame forwards	Right arrow key
Skip one key frame backwards	Left arrow key

Table 17: Keyboard control to navigate the view port.

A.3 Importing Animations

When the application is started, a default animation file is imported. To import a different fbx file in the application, the new file needs to be drag-and-dropped into the application window.

B Matrix Bone Texture Encoding and Decoding

B.1 Encoding - C/C++

```
for (int boneIndex = 0; boneIndex < boneCount; ++boneIndex) {
    auto centerBonePos = bonePositionCenters->at(boneIndex);
    auto bounds = boneBounds->at(boneIndex);
    auto multiplier = 255.f / bounds;

    for (int frameIndex = 0; frameIndex < numFrames; ++frameIndex) {
        auto matrix = matrices[boneIndex * frameCount + frameIndex];
        auto positionDifference = matrix[3] - centerBonePos;
        auto positionVector = positionDifference / bounds;

        // Encode only three first columns, as last is encoded in the w
        ↪  component.
        for (int columnIndex = 0; columnIndex < 3; ++columnIndex) {
            auto column = matrix[columnIndex];
            unsigned char x;
            unsigned char y;
            unsigned char z;
            unsigned char w;

            auto positionComponent = positionVector[columnIndex];
            x = (unsigned char) (column.x * 127 + 128);
            y = (unsigned char) (column.y * 127 + 128);
            z = (unsigned char) (column.z * 127 + 128);
            // We encode the positional data in the alpha channel
            w = (unsigned char) (positionComponent * 127 + 128);

            auto columnInTextureIndex = boneIndex * widthInBytes * 3 +
            ↪  columnIndex * widthInBytes + frameIndex * 4;

            textureBuffer[columnInTextureIndex] = x;
            textureBuffer[columnInTextureIndex + 1] = y;
            textureBuffer[columnInTextureIndex + 2] = z;
            textureBuffer[columnInTextureIndex + 3] = w;
        }
    }
```

```
}
```

B.2 Decoding - GLSL

```
mat4x3 getBoneMatrix(int frameIndex1, int frameIndex2, float interpolation, int
↪  boneIndex, vec3 boneCenter, vec3 bounds){

    uint boneInTextureIndex = boneIndex * 3;

    vec4 col11 = texelFetch(animationTextureSampler, ivec2(frameIndex1,
    ↪  boneInTextureIndex), 0) * 2 - 1;
    vec4 col12 = texelFetch(animationTextureSampler, ivec2(frameIndex1,
    ↪  boneInTextureIndex + 1), 0) * 2 - 1;
    vec4 col13 = texelFetch(animationTextureSampler, ivec2(frameIndex1,
    ↪  boneInTextureIndex + 2), 0) * 2 - 1;
    vec3 col14 = vec3(col11.w, col12.w, col13.w) * bounds + boneCenter;

    vec4 col21 = texelFetch(animationTextureSampler, ivec2(frameIndex2,
    ↪  boneInTextureIndex), 0) * 2 - 1;
    vec4 col22 = texelFetch(animationTextureSampler, ivec2(frameIndex2,
    ↪  boneInTextureIndex + 1), 0) * 2 - 1;
    vec4 col23 = texelFetch(animationTextureSampler, ivec2(frameIndex2,
    ↪  boneInTextureIndex + 2), 0) * 2 - 1;
    vec3 col24 = vec3(col21.w, col22.w, col23.w) * bounds + boneCenter;

    vec3 col1 = mix(col11.xyz, col21.xyz, interpolation);
    vec3 col2 = mix(col12.xyz, col22.xyz, interpolation);
    vec3 col3 = mix(col13.xyz, col23.xyz, interpolation);
    vec3 col4 = mix(col14, col24, interpolation);

    mat4x3 worldMatrix = mat4x3(col1, col2, col3, col4);
    mat4 linkMatrix = mat4(transformationLinkMatrices[boneIndex]);

    return worldMatrix * linkMatrix;
}
```

```
mat4x3 getDeformationMatrix(int frameIndex1, int frameIndex2, float
↪  interpolation){
    mat4x3 deformationMatrix =  getBoneMatrix(frameIndex1, frameIndex2,
    ↪  interpolation, boneIndices.x, bone1Center, bone1Bounds) * boneWeights.x;
    deformationMatrix += getBoneMatrix(frameIndex1, frameIndex2, interpolation,
    ↪  boneIndices.y, bone2Center, bone2Bounds) * boneWeights.y;
    deformationMatrix += getBoneMatrix(frameIndex1, frameIndex2, interpolation,
    ↪  boneIndices.z, bone3Center, bone3Bounds) * boneWeights.z;

    return deformationMatrix;
}

void main()
{
    float animationTime = mod((time + instanceAnimationTime) * frameRate,
    ↪  float(animationLength));

    int startIndex = int(animationTime);
    int endIndex = (startIndex + 1) % (animationLength);
    float interpolation = min(animationTime - float(startIndex), 1);

    mat4x3 defMat = getDeformationMatrix(startIndex, endIndex, interpolation);
    vec3 finalPos = defMat * vec4(position, 1);
    normal_out = normalize(mat3(M) * mat3(defMat) * normal_in);

    gl_Position = VP * M * vec4(finalPos, 1);
}
```

C Dual Quaternion Bone Texture Encoding and Decoding

C.1 Encoding - C/C++

```
for (auto boneIndex = 0; boneIndex < boneCount; ++boneIndex) {
    auto quatCenter = quaternionCenters->at(boneIndex);
    auto quatBounds = quaternionBounds->at(boneIndex);
    for (int frameIndex = 0; frameIndex < numFrames; ++frameIndex) {
        auto dq = dualquats[boneIndex * frameCount + frameIndex];

        auto real = dq[0];
        auto dual = dq[1];

        // The dual part holds the position
        auto dualDifference = dual - quatCenter;
        auto dualPart = dualDifference / quatBounds;

        auto encodedQuat = glm::dualquat(real, dualPart);

        for (int i = 0; i < 2; ++i) {
            auto offset = (2 * boneIndex + i) * widthInBytes + frameIndex * 4;

            auto quat = encodedQuat[i];

            auto x = (unsigned char) (quat.x * 127 + 128);
            auto y = (unsigned char) (quat.y * 127 + 128);
            auto z = (unsigned char) (quat.z * 127 + 128);
            auto w = (unsigned char) (quat.w * 127 + 128);

            textureBuffer[offset] = x;
            textureBuffer[offset + 1] = y;
            textureBuffer[offset + 2] = z;
            textureBuffer[offset + 3] = w;
        }
    }
}
```

C.2 Decoding - GLSL

```
vec4 quaternionMultiply(vec4 p, vec4 q){
    float x = p.w * q.x + p.x * q.w + p.y * q.z - p.z * q.y;
    float y = p.w * q.y + p.y * q.w + p.z * q.x - p.x * q.z;
    float z = p.w * q.z + p.z * q.w + p.x * q.y - p.y * q.x;
    float w = p.w * q.w - p.x * q.x - p.y * q.y - p.z * q.z;
    return vec4(x, y, z, w);
}

mat2x4 dualQuaternionMultiply(mat2x4 p, mat2x4 o){
    vec4 real = quaternionMultiply(p[0], o[0]);
    vec4 dual = quaternionMultiply(p[0], o[1]) + quaternionMultiply(p[1], o[0]);
    return mat2x4(real, dual);
}

mat2x4 getDualQuaternion(int frameIndex1, int frameIndex2, float interpolation,
↪  int boneIndex, vec4 quatCenter, vec4 bounds){

    int boneInTextureIndex = boneIndex * 2;

    vec4 real1 = texelFetch(animationTextureSampler, ivec2(frameIndex1,
    ↪  boneInTextureIndex), 0) * 2 - 1;
    vec4 real2 = texelFetch(animationTextureSampler, ivec2(frameIndex2,
    ↪  boneInTextureIndex), 0) * 2 - 1;
    vec4 dual1 = texelFetch(animationTextureSampler, ivec2(frameIndex1,
    ↪  boneInTextureIndex + 1), 0) * 2 - 1;
    vec4 dual2 = texelFetch(animationTextureSampler, ivec2(frameIndex2,
    ↪  boneInTextureIndex + 1), 0) * 2 - 1;

    vec4 real = mix(real1, real2, interpolation);
    vec4 dual = mix(dual1, dual2, interpolation);

    mat2x4 worldDualQuat = mat2x4(real, dual * bounds + quatCenter);
    mat2x4 linkQuat = linkQuaternions[boneIndex];
    return dualQuaternionMultiply(worldDualQuat, linkQuat);
}
```

```
mat2x4 normalizeDQ(mat2x4 dq){
    float magnitude = sqrt(dot(dq[0], dq[0]));
    mat2x4 ret = dq / magnitude;
    return ret;
}

mat2x4 getDeformationDualQuaternion(int frameIndex1, int frameIndex2, float
↪  interpolation){

    mat2x4 deformationDQ = getDualQuaternion(frameIndex1, frameIndex2,
    ↪  interpolation, boneIndices.x, bone1Center, bone1Bounds) * boneWeights.x;
    deformationDQ += getDualQuaternion(frameIndex1, frameIndex2, interpolation,
    ↪  boneIndices.y, bone2Center, bone2Bounds) * boneWeights.y;
    deformationDQ += getDualQuaternion(frameIndex1, frameIndex2, interpolation,
    ↪  boneIndices.z, bone3Center, bone3Bounds) * boneWeights.z;
    mat2x4 result = normalizeDQ(deformationDQ);
    return result;
}

vec3 deformDualQuat(mat2x4 dualQuat, vec3 vec){
    vec4 real = dualQuat[0];
    vec4 dual = dualQuat[1];
    vec3 result = (cross(real.xyz, cross(real.xyz, vec) + vec * real.w +
    ↪  dual.xyz) + dual.xyz * real.w - real.xyz * dual.w) * 2 + vec;
    return result;
}

vec3 rotateQuat(vec4 quat, vec3 vec){
    vec3 uv = cross(quat.xyz, vec);
    vec3 uuv = cross(quat.xyz, uv);
    return vec + ((uv * quat.w) + uuv) * 2;
}
```

```
void main()
{
    float animationTime = mod((time + instanceAnimationTime) * frameRate,
    ↪  float(animationLength));

    int startIndex = int(animationTime);
    int endIndex = (startIndex + 1) % (animationLength);
    float interpolation = min(animationTime - float(startIndex), 1);

    mat2x4 dq = getDeformationDualQuaternion(startIndex, endIndex,
    ↪  interpolation);
    vec3 pos = deformDualQuat(dq, position);
    normal_out = normalize(mat3(M) * rotateQuat(dq[0], normal_in));

    gl_Position = VP * M * vec4(pos, 1);
}
```

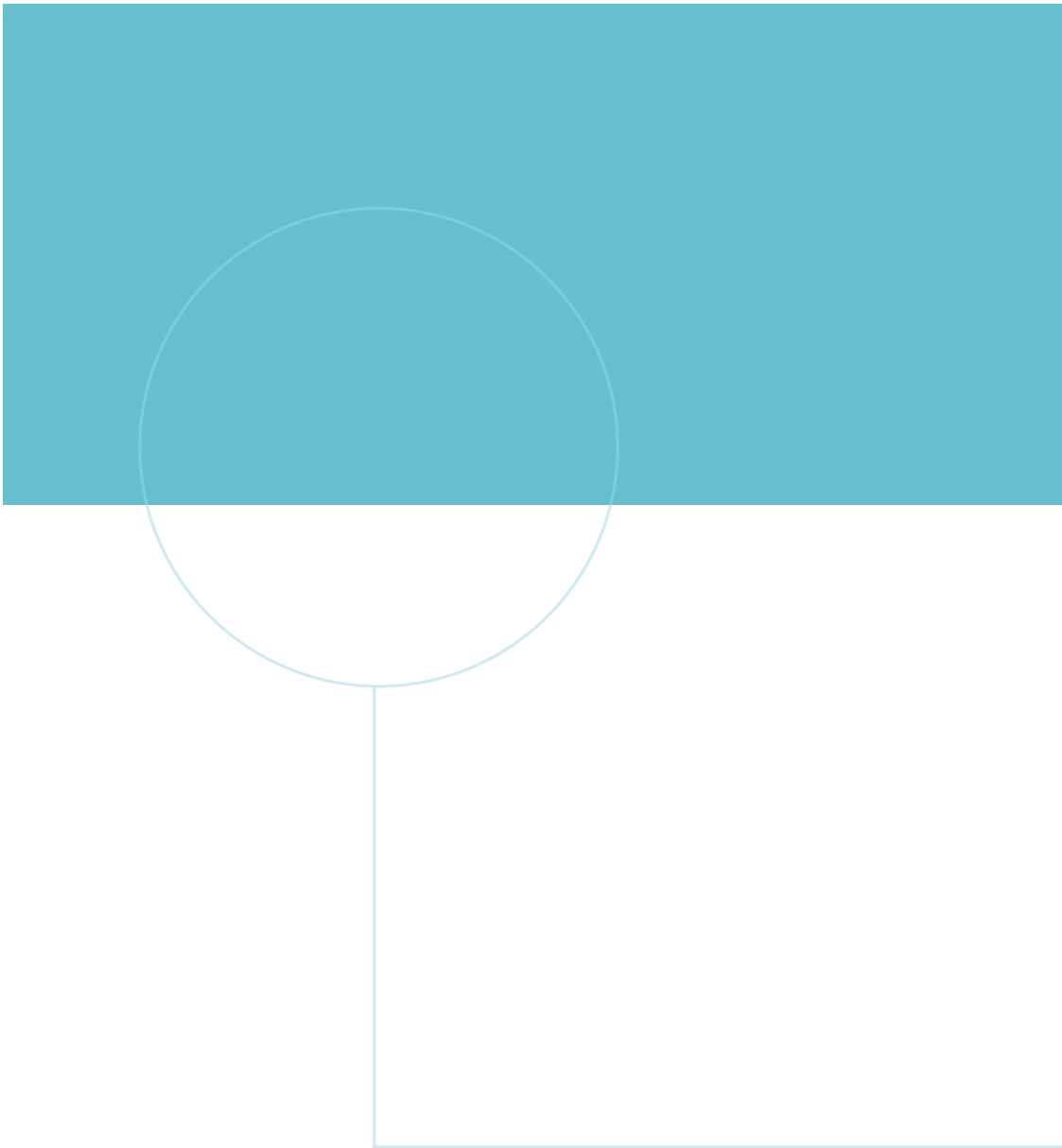
D Vertex Texture Encoding and Decoding

D.1 Encoding - C/C++

```
for (int vertexIndex = 0; vertexIndex < numVertices; vertexIndex++) {
    int rowOffset = vertexIndex * textureWidthBytes;

    for (int frameIndex = 0; frameIndex < frameCount; frameIndex++) {

        auto position = samples[frameCount * vertexIndex + frameIndex];
        auto relativeToCenter = position - vertexCenter;
        auto withinBounds = relativeToCenter / vertexBounds;

        auto x = (unsigned char) (withinBounds.x * 127.f + 128);
        auto y = (unsigned char) (withinBounds.y * 127.f + 128);
        auto z = (unsigned char) (withinBounds.z * 127.f + 128);

        textureBuffer[rowOffset + frameIndex * 3] = x;
        textureBuffer[rowOffset + frameIndex * 3 + 1] = y;
        textureBuffer[rowOffset + frameIndex * 3 + 2] = z;
    }
}
```

D.2 Decoding - GLSL

```
vec3 getVertexPosition3Channel(int startIndex, int endIndex, float
↪  interpolation){

    vec3 startPos = (texelFetch(animationVertexTextureSampler, ivec2(startIndex,
    ↪  trueIndex), 0).xyz * 2 - 1) * vertexBounds + vertexCenter;
    vec3 endPos = (texelFetch(animationVertexTextureSampler, ivec2(endIndex,
    ↪  trueIndex), 0).xyz * 2 - 1) * vertexBounds + vertexCenter;

    vec3 finalPos = mix(startPos, endPos, interpolation);
    return finalPos;
}

void main()
{
    float animationTime = mod((time + instanceAnimationTime) * frameRate,
    ↪  float(animationLength));

    int startIndex = int(animationTime);
    int endIndex = (startIndex + 1) % (animationLength);

    float interpolation = min(animationTime - float(startIndex), 1);

    vec3 finalPos = getVertexPosition(s, e, interpolation);

    vec3 startNor = texelFetch(animationNormalTextureSampler, ivec2(s,
    ↪  trueIndex), 0).xyz * 2 - 1;
    vec3 endNor = texelFetch(animationNormalTextureSampler, ivec2(e, trueIndex),
    ↪  0).xyz * 2 - 1;
    vec3 finalNor = normalize(mix(startNor, endNor, interpolation));

    normal_out = normalize(mat3(M) * finalNor);
    gl_Position = VP * M * vec4(position + finalPos, 1);
}
```



NTNU

Kunnskap for en bedre verden