Matrix Palette Skinning and Dual Quaternions

Semester Thesis

Lukas Mathias Novosad

July, 2007

ETH Zürich
Department of Computer Science
Applied Geometry Group

Supervisors:
Prof. Dr. Mark Pauly
Dr. Robert Sumner

Abstract

Modeling and animation is a very important topic in visual computing. One of the most widely used paradigms is called "skinning".

The skinning procedure of a model is usually based on an underlying skeleton. This skeleton first gets deformed and then these deformations, commonly referred to as "bones", are applied to the vertices of the actual model. Each vertex of the model is influenced by a weighted sum of the nearest few bones.

The work in this thesis aims to improve skinning performance by moving operations from the CPU to the GPU. Compared to previous approaches, we want to raise the number of bones the GPU can handle. Furthermore, the model size, e.g. the number of primitives, may be huge.

Finally, we also consider an alternative approach to classical skinning by using dual quaternions. These may reduce artifacts which occur with matrix-based skinning and can also increase the performance.

Contents

Chapter 1

Introduction

1.1 Motivation

Skinning is one of the most important techniques for character animation and deformation. Usually, it is based upon a skeleton underlying the actual model and each vertex of the model gets influenced by some of the nearest bones in the final animation. The trend in recent years was to increase the degrees of freedom per model, e.g. the number of deformations or bones possible. On one hand, this is due to the increased amount of memory and computing power available nowadays. Furthermore, digital content creators want to achieve a higher level of realism. However, there is a discussion going on, whether to use the CPU for skinning, or the GPU. In the past, skinning on the GPU was limited to a few bones only. This has of course changed in the last couple of years. On the other hand, skinning code for the CPU can be made more sophisticated and dynamic. But this will change with the new GPU generations which have a unified shader architecture. This will lead to more sophisticated shader APIs which will be able to create more dynamic GPU code. The goal of this thesis is to enhance to current work on deformation from my supporters [1] with a hardware accelerated skinner on the GPU. We take two different approaches. First, we consider the classic, matrix-based technique, and then, as an alternative, we take the approach based on dual quaternions.

1.2 Problem space

We are in the area of model skinning, which is used in all sorts of applications where model deformation and animation is needed. One of the most interesting and demanding application is certainly digital and animated movie production. Another target are computer games and data visualization and reconstruction. Our method falls into the category of geometric skinning methods. These provide a direct interaction between the underlying bones

and the model skin. Most of them are based on matrices or quaternions. Recently, methods started appearing which are based on dual quaternions, most notable is [2]. There are also physically-based approaches to skinning. These aim to simulate the whole body structure with all the different tissues and bones, and most importantly, the interaction between all these layers. Naturally, physically-based approaches usually require much more computational costs due to the realism level they achieve. Another approach would be to use a database, which, for example, uses motion captured data to blend as good as possible into the desired, new position of the character. However, these approaches are limited in generality by the number of provided references in the DB. Furthermore, they also require a lot of parameter tuning, which can be very tedious and cumbersome. Geometric approaches, we believe, are currently the best way to achieve interactive performance for model skinning and deformation.

1.3 Contribution

This thesis provides two approaches to hardware accelerated skinning. Both are implemented as GPU shader programs written using NVidia's Cg - C for graphics. One of the shaders takes the classical approach of matrix palette skinning, but with a rather large amounts of bones and model vertices compared to previous implementations. The other shader implements a rather new technique of skinning based on dual quaternions, as suggested in [2]. This approach has the potential to reduce artifacts which can occur when matrices are used for skinning. Additionaly, the performance may be better since dual quaternions require less data to be stored and passed to the GPU.

1.4 Document Overview

The structure of this document is as follows:

Background After the introduction, we provide more detailed descriptions on the background of the thesis. This includes mathematical concepts, such as dual quaternions and skinning equations. Furthermore, we list the technologies used in this thesis.

Implementation Following the background section, we describe details about the implementation such as the newly created classes and shader programs we have written. This should give the reader a good overview and provide a better insight. Anybody interested in particular details should of course read the actual implementation code.

Results The results chapter states what we have achieved with the shader programs. We provide timings which we measured and compare them to the previous rendering and deformation routine.

Conclusion At the end, we draw some conclusions about the experiences made and the difficulties encountered while writing this semester thesis. We also provide an outlook on possible future work which could be done in this area of research.

Chapter 2

Background

At the beginning, we researched previous work done in the field of skinning and dual quaternions. We found several examples of hardware accelerated skinning for different GPU generations. Along with them came all the explanations about the limitations on the number of bone matrices and the individual GPU peculiarities. Our aim was to use the new GeForce8 series to lift these limitations and not only to improve the skinning performance, but also to increase the amount of bones which can be used to skin a model.

In this chapter we provide information about the mathematical concepts applied and technologies used in this thesis. We assume, that the reader is familiar with ordinary quaternion algebra.

2.1 Skinning Equations

In this section we provide background information on skinning and the involved equations. The most basic skinning equation is

$$v' = (\sum_{i=0}^{n} w_i * T_i) * v \tag{2.1}$$

where for each vertex i the weights w_i must fulfill the following condition

$$\sum_{i=0}^{n} w_i = 1 \tag{2.2}$$

with

$$0 \leq w_i \leq 1$$

This corresponds to a convexity constraint on the skinning weights. The T_is

represent the transforms. Usually these are matrices, but other concepts such as dual quaternions are also possible. Naturally, the equation and operators must change based on the mathematical concept used for skinning.

2.2 Dual Numbers

Before we give a short introduction to the algebra of dual quaternions, we first explain the concept of dual numbers. Dual quaternions are build on dual numbers.

Notation

Dual numbers are very similar to complex numbers, that is they have a real part and a so-called dual part. A dual number is written as

$$\hat{a} = a_0 + \epsilon * a_\epsilon$$

where a_0 denotes the real part, a_ϵ the dual part and ϵ is the dual unit satisfying

$$\epsilon^2 = 0$$

which is the difference to complex numbers, where the imaginary unit squares to -1.

Addition

Two dual numbers $\hat{a}$ and $\hat{b}$ are added as follows:

$$\hat{a} + \hat{b} = (a_0 + \epsilon * a_\epsilon) + (b_0 + \epsilon * b_\epsilon) = (a_0 + b_0) + \epsilon * (a_\epsilon + b_\epsilon)$$

Multiplication

Two dual numbers $\hat{a}$ and $\hat{b}$ are multiplied in the following manner:

$$\hat{a} * \hat{b} = (a_0 + \epsilon * a_\epsilon) * (b_0 + \epsilon * b_\epsilon) = (a_0 * b_0) + \epsilon * (a_0 * b_\epsilon + a_\epsilon * b_0)$$

Conjugate

The conjugate of a dual number is again similar to the conjugate of a complex number:

$$\bar{\hat{a}} = a_0 - \epsilon * a_\epsilon$$

Inverse

The inverse of a dual number is defined as:

$$\frac{1}{a_0 + \epsilon * a_\epsilon} = \frac{1}{a_0} - \epsilon * \frac{a_\epsilon}{a_0^2}$$

and only exists if $a_0 \neq 0$.

Square root

The square root of a dual number is computed as:

$$\sqrt{a_0 + \epsilon * a_\epsilon} = \sqrt{a_0} + \epsilon * \frac{a_\epsilon}{2 * \sqrt{a_0}}$$

and only exist if a_0 is positive.

2.3 Dual Quaternions

Dual quaternions can be seen as four component vectors of dual numbers. It has a four component vector containing the non-dual vector q_0, and a four component vector q_ϵ which represents the dual vector:

$$\hat{\mathbf{q}} = \mathbf{q_0} + \epsilon * \mathbf{q}_\epsilon$$

Both the non-dual and the dual part can be described as an ordinary quaternion. From this the conjugation follows in a straightforward manner.

Conjugate

The conjugate of a dual quaternion is defined as:

$$\hat{\mathbf{q}}^* = \mathbf{q_0^*} + \epsilon * \mathbf{q}_\epsilon^*$$

where * denotes the ordinary quaternion conjugation. Note that there are several different ways to define the conjugation of dual quaternions depending upon what properties the dual quaternions must fulfill. Another way to define it, for example, would be $\hat{\mathbf{q}}^* = \mathbf{q_0^*} - \epsilon * \mathbf{q}_\epsilon^*$.

Norm

The norm of a dual quaternion is written as:

$$\|\hat{\mathbf{q}}\| = \sqrt{\hat{\mathbf{q}} * \hat{\mathbf{q}}} = \|\mathbf{q_0}\| + \epsilon * \frac{\langle \mathbf{q_0}, \mathbf{q}_\epsilon \rangle}{\|\mathbf{q_0}\|}$$

Inverse

The inverse of a dual quaternion is computed as:

$$\hat{\mathbf{q}}^{-1} = \frac{\hat{\mathbf{q}}^*}{\|\hat{\mathbf{q}}\|^2}$$

and only defined if $\mathbf{q_0} \neq \mathbf{0}$.

2.3.1 Unit dual quaternions

There exist special dual quaternions called unit dual quaternions. Similar to ordinary unit quaternions they fulfill the property

$$\|\hat{\mathbf{q}}\| = \mathbf{1}$$

and this holds if and only if $\|\mathbf{q_0}\| = 1$ and $\langle \mathbf{q_0}, \mathbf{q}_\epsilon \rangle = 0$ as can be immediately verified by looking at the definition of the norm for a dual quaternion.

From this it also follows that unit dual quaternions are always invertible since this simply corresponds to a conjugation of the dual quaternion. Same as with ordinary quaternions, transformations are easily concatenated by multiplying unit dual quaternions. The product of two unit dual quaternion is again a unit dual quaternion.

2.3.2 Properties

Just like ordinary quaternions, dual quaternions are associative and distributive, but not commutative. They are applied to vertices the same way as quaternions:

$$v' = \hat{q} * v * \hat{q}^*$$

Rotation

A 3D rotation is represented as a dual quaternion where the dual part is equal to zero $\mathbf{q}_\epsilon = \mathbf{0}$. As a consequence we may use the standard quaternion rotation. For a vector $\mathbf{v} = (v_x, v_y, v_z)$ in 3D, the corresponding unit dual quaternion is $\hat{\mathbf{v}} = 1 + \epsilon(v_x i + v_y j + v_z k)$. Now the rotation of a vector by a dual quaternion $\hat{\mathbf{q}}$ is computed as $\hat{\mathbf{q}} * \hat{\mathbf{v}} * \hat{\mathbf{q}}^*$. By using the fact that the dual part is zero, we can simplify:

$$\hat{\mathbf{q}}\hat{\mathbf{v}}\hat{\mathbf{q}}^* = \mathbf{q_0}(1 + \epsilon(v_x i + v_y j + v_z k))\mathbf{q_0^*} = 1 + \epsilon\mathbf{q_0}(v_x * i + v_y * j + v_z * k)\mathbf{q_0^*}$$

and see that this indeed corresponds to a common quaternion rotation.

Translation

In contrast to ordinary quaternions, dual quaternions can also describe a 3D translation. A translation by a vector $\mathbf{t} = (t_x, t_y, t_z)$ is achieved by the dual quaternion $\hat{\mathbf{t}} = 1 + \frac{\epsilon}{2}(t_x i + t_y j + t_z k)$. This means that dual quaternions work with half the translation distance, similar to ordinary quaternions which work with half the rotation angle. As before with the rotation, we simplify

$$\hat{\mathbf{t}}\hat{\mathbf{v}}\hat{\mathbf{t}}^* = \hat{\mathbf{t}}(1 + \epsilon(v_x i + v_y j + v_z k))\hat{\mathbf{t}}^* = 1 + \epsilon((v_x + t_x)i + (v_y + t_y)j + (v_z + t_z)k)$$

and see that this indeed corresponds to a translation by $\mathbf{t} = (t_x, t_y, t_z)$.

2.4 Matrix Palette Skinning

In case of the matrix-based approach the T_is in (2.1) take the form of a matrix. This approach has the advantage that the transformation matrix can potentially contain a rotation and a translation, and furthermore, the transformation may also scale and shear. This is not possible with dual quaternions which can only handle rigid-body transformations. The drawback of this approach is the fact, that artifacts can occur when the rotation matrices are linearly interpolated. The rotations start to deviate and are not pure anymore. This can, for example, result in a folding of the mesh. The deformation framework "Embedded Deformation for Shape Manipulation" [1] from the supporters of this thesis is based on matrices.

A very notable publication is "Skinning Mesh Animations" from Doug L. James and Christopher D. Twigg [3] which often serves as reference for work done in this area. Other good resources we used are "EigenSkin: Real Time Large Deformation Character Skinning in Hardware" from Paul G. Kry, Doug L.James, and Dinesh K. Pai [4] as well as "DyRT: Dynamic Response Textures for Real Time Deformation Simulation with Graphics Hardware" from Doug L. James and Dinesh K. Pai [5].

2.5 Dual Quaternion Skinning

For the dual quaternion approach the T_is in (2.1) stand for dual quaternions. Naturally, the equation also changes due to the mathematical structure of dual quaternions and becomes

$$v' = \sum_{i=0}^{n} w_i * (dq_i * v * \hat{dq_i}) \tag{2.3}$$

The advantage of this approach is the property of dual quaternions, that linearly blended quaternions don't deviate and the rotations stay pure. This can reduce artifacts which occur for the matrix-based approach. Furthermore, a dual quaternion requires only eight values to be stored compared to twelve for a matrix. The drawback is the lack of scale and shear transforms since dual quaternions can only represent rigid-body transforms.

The most valuable resources we used here are "Skinning with Dual Quaternions" from Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan [2] and their previous work "Dual Quaternions for Rigid Transformation Blending" [6].

2.6 NVidia's Cg

Cg stands for "C for graphics" and it is developed and maintained by NVidia[1]. They provide a compiler called "cgc" which is able to generate assembler output for several different vertex and fragment shader profiles. This also includes hardware from other GPU manufacturers such as ATI and is very usefull for the development process. Furthermore, there are several plugins available for development tools such as Maya or 3ds max.

2.7 NVidia's GeForce8 series

We picked this GPU generation since it's computational power is very high at the moment compared to other GPUs available. The GeForce 8800 GTX offers 128 shader pipelines and has 768MB of RAM. Furthermore, it has a unified shader architecture and supports DirectX 10. Another feature is the new generation of shader programs called geometry shaders. These are able to work on larger input and output primitives. Before, the shader input and output consisted of one single vertex. Now, for example, triangles can be processed by a geometry shader program. Finally, there is a new extension called NV_transform_feedback which can capture the output of a shader program. With this extension, a deformed model from the GPU can be read back to the GPU with a few lines of code.

2.8 GPGPU

GPGPU stands for "General-Purpose computation on GPUs". There is a website specifically dedicated to it[2]. Basically, this means that one can use

[1] NVidia Cg homepage, http://developer.nvidia.com/page/cg_main.html
[2] GPGPU homepage, www.gpgpu.org

the GPU functionality for different purposes than the ones it was designed for. We use the texture units to store data which we don't use to texturize object or models. Instead, we store matrices, weights, or indices in the texture which we need for the skinning procedure. This means, that we read the matrices out of the texture in the shader program and use them for skinning. Due to the large texture sizes (8192x8192 on the GeForce8 series) one can easily store large amounts of data. Each time the data gets updated on the CPU, a new texture is uploaded to the GPU.

Chapter 3

Implementation

In this chapter we provide details about the implementation and the source code. First, there is a section about a toy example we used to test the basic shader programs with. Then we describe the classes and functions added to the framework of [1]. Finally, we explain the two shader programs we wrote for the matrix-based and the dual quaternion approach to skinning.

3.1 Toy example

We used a small engine to get a better understanding for the skinning procedure. The skinning example consisted of two bones, e.g. a lower and upper arm with rotations about the elbow and shoulder. Most of this code was hard-coded but it enabled us to write a shader class in order to test our first shader versions.

We also implemented a sketch for the dual quaternion classes in this engine. They are described in section 3.2.2 containing the new classes provided by this semester thesis.

3.2 Beyond the toy example

After having written and tested the basic class for shader handling we integrated it into the deformation framework. The first adaption we had to do, was to include the center vertex for each bone. Each vertex to be skinned is first translated to the respective center vertex, then the deformation is applied in this embedded subspace, and finally the deformed vertex is translated back. At the very end, the corresponding weight is applied.

3.2.1 Modified classes

Here we list all the classes we have modified in order to include the shader handling.

GLCanvas

We added a function called `RenderDefPoseGPU` which is responsible for setting up and maintaining the data structures needed for the shader program. It also enables the shader program, which then deforms and renders the skinned model. Finally, the transform feedback functionality is triggered in this function.

Furthermore we extended the `Initialize` function to initialize the `MPSkinner` shader class. This includes the texture handles and arrays used for the vertex buffer objects.

RiEdit

Here we added a function called `ReadPoseGPU` which reads back the deformed model from the GPU to the CPU. It uses the NV_transform_feedback extension from NVidia which was newly designed for the GeForce 8 GPU series. The extension is capable of catching the vertices generated by a vertex or geometry program. We believe that this feature is extremely valuable to the graphics community and will be heavily used. Unfortunately, the read back model appears to be correct but seems to be wrongly transformed in space. This will be solved in the future.

3.2.2 New classes

MPSkinner

This class is responsible for the handling of the shader programs. It initializes needed graphics extensions and reads in the source code of the shader program. Furthermore, it also offers handles to the variables and textures in the Cg vertex shader program. Additionally, there are a few helper functions, for example, there is a method to calculate vertex dimensions to be used for the textures on the GPU.

Dual Number

This class implements the dual number with its functions as described in section 2.2.

Dual Quaternion

This class implements the dual quaternions and the required functions and operators as described in section 2.3.

Quaternion

The quaternion class is provided for reasons of completeness. It is used by the dual quaternion class as described in section 2.3 about dual quaternions. It provides functionality such as conversion to and from a rotation matrix. Also the conjugation and blending of quaternions is available.

3.3 Shader programs

Here we provide details about the shader programs used for the skinning approaches. For a more detailed insight, the reader should read the actual implementation code.

3.3.1 Data structures

First we describe the data structure used in the vertex shader programs and the interface used for both shader programs.

Vertex Buffer Objects

In order to improve memory management and vertex shader rendering, we use four vertex buffer objects (VBOs[1]) to pass the vertex position, normal, color, and secondary color to the shader programs. The secondary color is used to pass the number of bones and the current VBO index, to be used with the current vertex, to the shader programs. This is illustrated in figure 3.1. Here the secondary color VBO is named VBO_3.

Transformation indices

The indices to be used to access data needed for the current vertex are stored in an index texture. Each texel stores up to four indices as depicted in figure 3.2. Our deformation framework currently supports up to four bones influencing a single vertex. The index into the index texture itself is passed in a VBO as described in section 3.3.1.

[1] http://developer.nvidia.com/object/using_VBOs.html

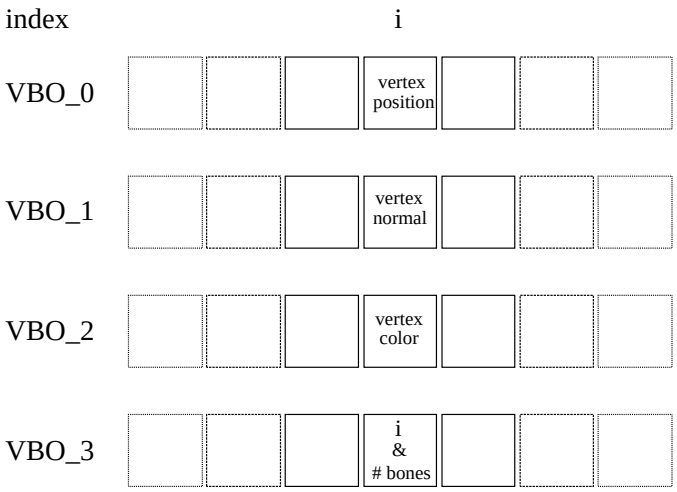

Figure 3.1: VBO structure

Center vertices

The center vertices describe the reference center point of the subspace in which the deformation takes place. Each index obtained from the index texture references a vertex in the center texture. This is illustrated in figure 3.3.

Bone texture

The bone texture stores the skinning deformations. Each deformation requires 12 real numbers to be stored. This corresponds to three texels from the bone texture, since each RGBA texel stores four values of type float. See figure 3.4 for reference. Depending upon which shader program is active, these values are used to build either a bone matrix or a dual quaternion bone. As before with the center vertices, each index from the index texture references one bone in the bone texture.

Weight texture

Finally, the weight texture contains the skinning weights. The weights are accessed using the VBO index stored in the secondary color, similar to the transformation indices. These weights fullfill the condition stated in equation 2.2. Figure 3.5 serves as an illustration.

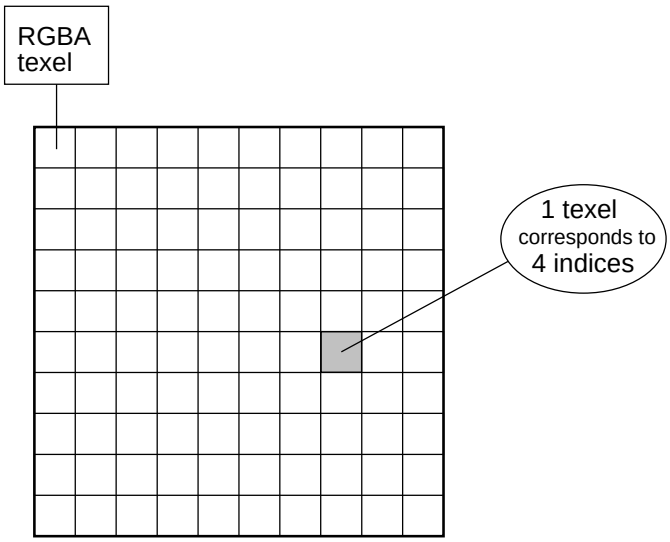

Figure 3.2: Index texture

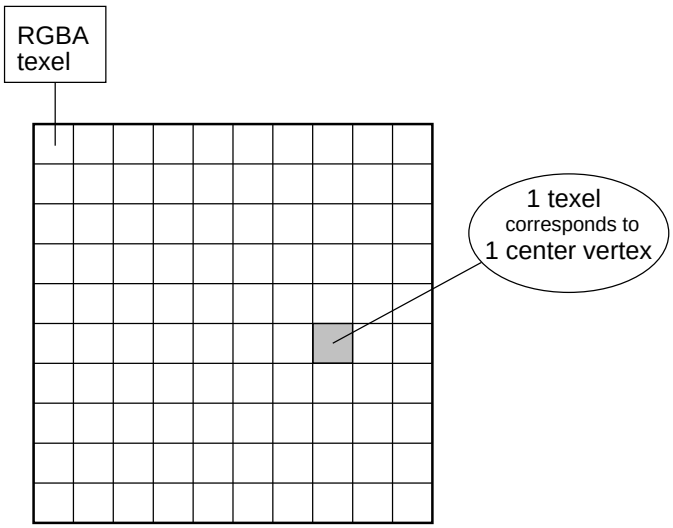

Figure 3.3: Center texture

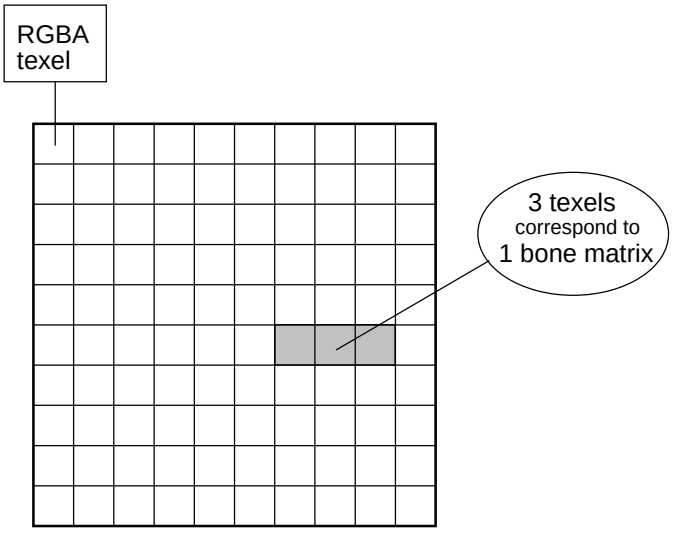

Figure 3.4: Bone texture

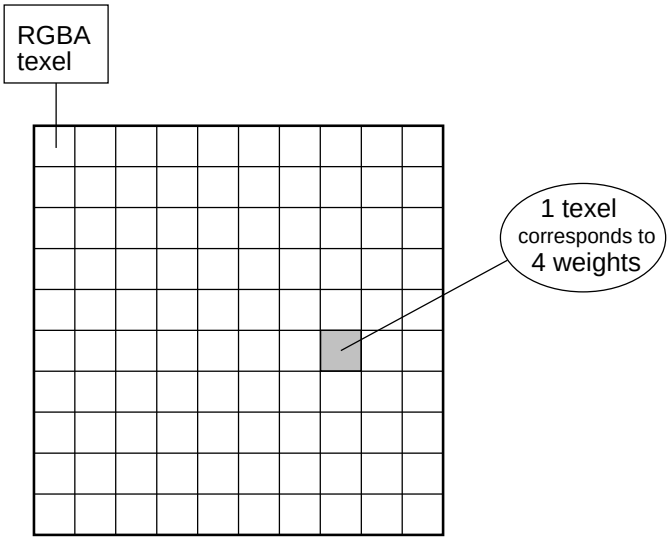

Figure 3.5: Weight texture

Shader Interface

After having described the data structures, we now move on to the shader interface. The vertex shader takes as input the vertex position, normal, color, and the number of reference bones used for the current vertex. As explained before, these are all stored in vertex buffer objects to increase the performance. Additionally, the textures holding the skinning indices, weights, reference vertices, and the transformation matrices, as desribed in the previous sections, are passed to the shader program. The first three are set once at the beginning on program initialization while the transformation matrices must be all updated for each execution of the rendering loop in order to take into account the deformation process.

The output of the vertex shader consist of the skinned and model-view-projected vertex position and the vertex color. Since we do not use any fragment shader, the fragments are computed by the standard OpenGL rendering pipeline. Furthermore, the vertex positions recorded with the NV_transform_feedback extension have to be muliplied by the inverse model-view-projection matrix in order to transform the vertices back to model space.

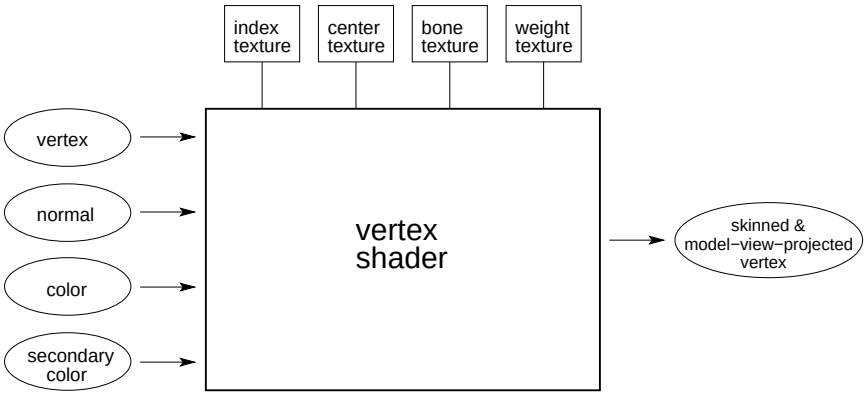

Figure 3.6: Shader interface

3.3.2 MatrixPaletteSkinningVS

This vertex shader provides the skinning functionality based on matrices. It uses GPGPU techniques to implement skinning for hundreds of bones, reference vertices, weights and indices. The complete source code is provided in appendix A.

Processing

Inside the shader program, the VBO index stored in the secondary color is used to extract the skinning indices out of the index texture and the skinning weights out of the weight texture. The skinning indices, in turn, are then used to extract the transformation matrices and center vertices from the corresponding textures to be used with the current vertex. This works because the index texture stores the skinning indices and weights in the same order as the vertex data in the VBOs and the VBO index is passed to the shader program using the secondary color VBO as described in section 3.3.1.

By this, the vertex shader always knows where to look for the data needed for the current vertex. When all the needed data is ready and set up, the matrix, for example, is build with a small subroutine, the vertex is skinned according to the an extended version of equation 2.1 which now also includes the center vertices:

$$v' = \sum_{i=0}^{n} w_i * (T_i * (v - c_i) + c_i) \tag{3.1}$$

where c_i stands for the reference vertex representing the transformation center for transformation T_i.

In the vertex shader, the Cg code to apply one bone matrix to the current vertex looks as follows:

```
// finally apply the bone matrix to the vertex position and normal
finalPos = finalPos + (mul(bone,
                           float4(IN.Position - center, 1.0f)
                           )
                       + center
                       ) * weights.x;
```

Where `bone` is a float3x4 matrix and `center` is a float4 vector corresponding to the reference center vertex.

3.3.3 DualQuaternionVS

The dual quaternion shader implements skinning with an alternative approach. It has the potential advantage, that instead of 12 real numbers, just 8 would have to be passed for each bone. But the drawback is, that only rigid body transformations are available. Scale and shear transformations would have to be implemented by a few additional scale and shear matrices. This would again raise the number of values passed to the shader program. We do not support scale and shear transforms for the dual quaternion approach at the moment. These would have to be blended with the rigid

body transforms which are provided by the dual quaternions. Furthermore, we use the same interface as for the matrix-based approach, thus we still pass 12 real number values for each transform. The source code is provided in appendix B.

Processing

The processing follows the same principle as for the matrix-based approach. The only difference is that the transformations from the texture holding the transforms are converted to dual quaternions instead of matrices. As a consequence, the skinning is done according to the following equation, which is an extended version of 2.3 now taking the reference vertices into account:

$$v' = \sum_{i=0}^{n} w_i * (dq_i * (v - c_i) * \hat{dq_i} + c_i) \tag{3.2}$$

The source code section for this is simplified to:

```
// finally apply the dual quaternion to the vertex position and normal
finalPos = finalPos +
           (applyDQtoVertex(dualQuat,float4(IN.Position - center, 1.0f))
              + center
            ) * weights.x;
```

where `applyDQtoVertex` is the following subroutine motivated by [2]:

```
/*
 * This function multiplies a dual quaternion
 * with the vertex position.
 *
 */
float3 applyDQtoVertex(float2x4 dualQuat, float4 position)
{

  float len = length(dualQuat[0]);
  dualQuat /= len;

  float3 outPosition = position.xyz +
                       2.0*cross(  dualQuat[0].xyz,
                                   cross(dualQuat[0].xyz,position.xyz)
                                 + dualQuat[0].w*position.xyz
                                 );

  float3 transPosition = 2.0 * (  dualQuat[0].w*dualQuat[1].xyz
                                - dualQuat[1].w*dualQuat[0].xyz
                                + cross(dualQuat[0].xyz, dualQuat[1].xyz)
                                );

  outPosition += transPosition;

  return outPosition;
}
```

The output is basically the same as for the matrix-based approach. The only problem is the fact, that at the moment, we simply ignore the scale and shear factors from the bone transform. But these factors can be included as future work and the rendering output still looks quite good as can be seen in the results section.

Chapter 4

Results

In this chapter we show some results we achieved with this thesis. We compared our GPU shader programs to the previous CPU version and found an improvement by a rough factor of 5. Additionally, the GPU programs also skin the vertex normal, the CPU version we refer to does not do so at the moment. The timings for the GPU version do not include the time for the readback of the vertex shader output from the GPU to the CPU. The reason for this, is the fact, that we encountered some problems with the read back vertices: The model appears to be correct, but wrongly positioned in space. This will be fixed in the future. The time measurements where performed on an Intel Core 2 Duo processor with 2.4GHz, 2GB RAM, and a GeForce 8800 GTS.

4.1 Matrix Palette Skinning

The deformed and rendered model of the matrix-based approach is perfectly correct. Furthermore, we experienced a considerable speedup, compared to the previous CPU version, while interacting with the deformation framework. As the timings table 4.1 below shows, the performance is indeed very good.

Considering the fact, that we have not yet used a geometry shader or the CUDA (Compute Unified Device Architecture) library from NVidia[1], there is still room for further improvement of the shader program. Be it a purely better performance or even broader shader functionality, such as a geometry program, which would be able to update the deformation graph from [1] or the model which is being skinned, directly on the GPU.

[1] CUDA homepage: http://developer.nvidia.com/object/cuda.html

Model	Vertices	Nodes	Deform&Render
Dino	10,002	222	0.20 ms
Dino	10,002	425	0.25 ms
Dino	10,002	1,048	0.35 ms
Giraffe	79,226	221	4.40 ms
Raptor	24,982	287	0.40 ms
Raptor	85,792	287	5.10 ms

Table 4.1: Timings for matrix based approach

4.2 Dual Quaternion Skinning

The capabilities of the dual quaternion skinning shader are basically the same, e.g. the large amount of bones which can be handled, for example. Also the timings for the dual quaternion approach are very similar to the ones from the matrix palette skinner. As already mentioned in the previous chapter, we currently ignore the scale and shear factors from the transformation data. The blending of these will certainly require some additional time. As a consequence, the deformed and rendered model does have some wrong sections. But surprisingly, there are not many of them. Although scaling and shearing is ignored, the rendering output still looks very good. Most noticeable is the drifting of the model for large scale deformations. Then the model visually leaves the handle, to which it is attached to. This can be seen in figure 4.1, which shows the output of the matrix-based shader, and in figure 4.2, which shows the dual quaternion shader output. Pictures 4.5 and 4.6 show the advantage of the output generated with the dual quaternion skinner. It reduces artifacts such as mesh folding, shown in pictures 4.3 and 4.4, which occurs for the matrix-based approach. The timings measured for the dual quaternion approach are listed in table 4.2.

Model	Vertices	Nodes	Deform&Render
Dino	10,002	222	0.20 ms
Dino	10,002	425	0.25 ms
Dino	10,002	1,048	0.35 ms
Giraffe	79,226	221	4.40 ms
Raptor	24,982	287	0.40 ms
Raptor	85,792	287	5.00 ms

Table 4.2: Timings for dual quaternion approach

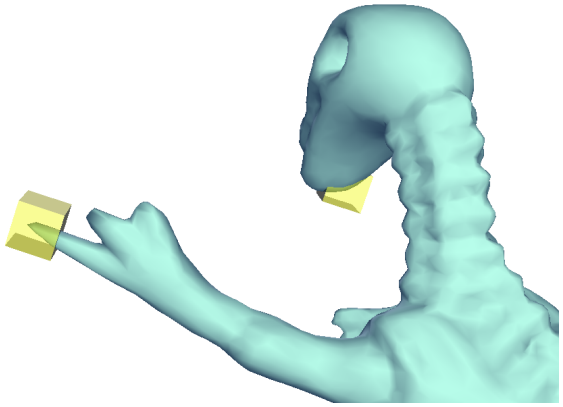

Figure 4.1: Dino finger stretched with MPS

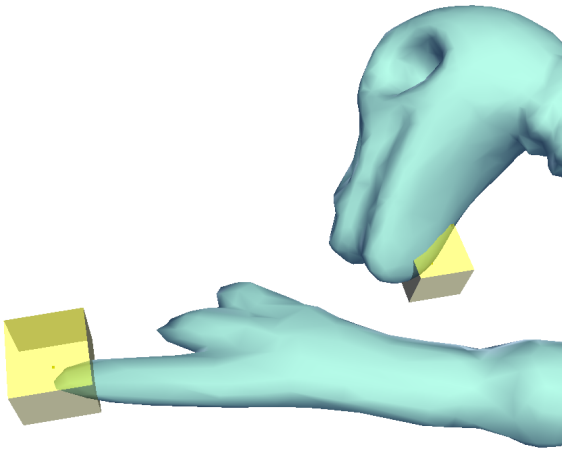

Figure 4.2: Dino finger stretched with DQS

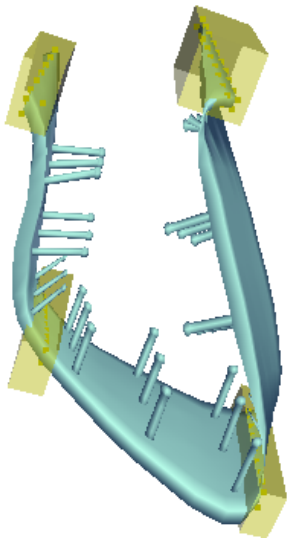

Figure 4.3: Overview of MPS bend bar

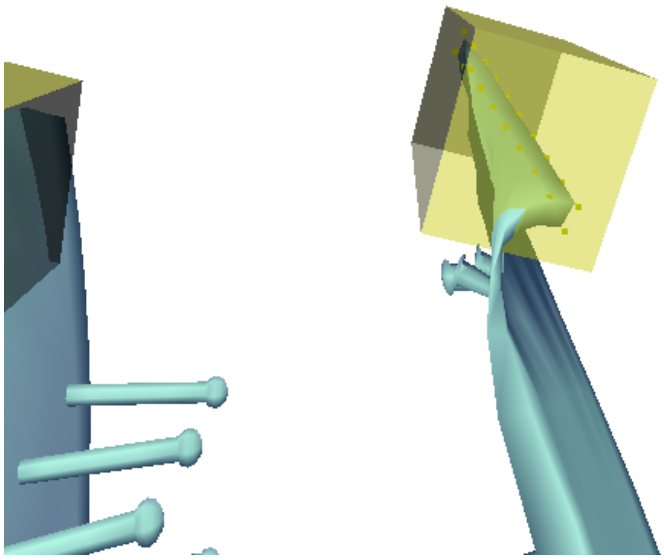

Figure 4.4: Detail of MPS bend bar

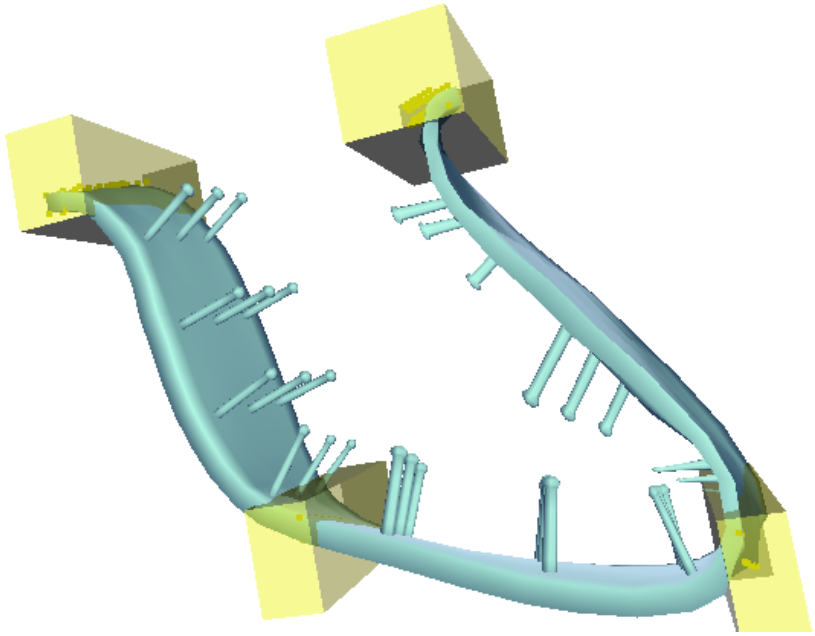

Figure 4.5: Overview of DQ bend bar

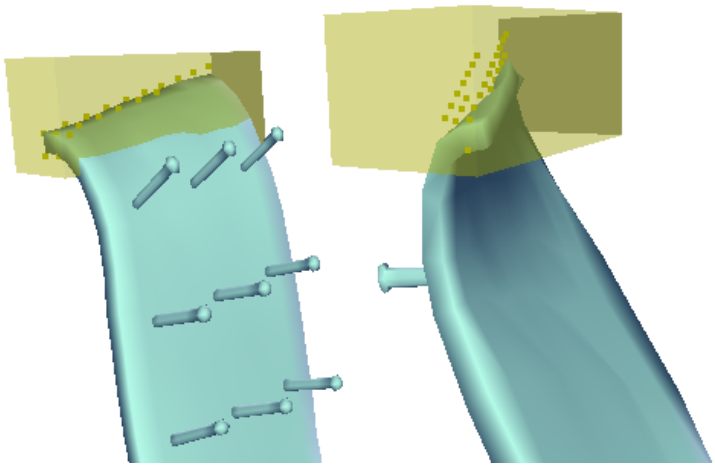

Figure 4.6: Detail of DQ bend bar

Chapter 5

Conclusion

5.1 Contribution

We have successfully implemented a matrix-based vertex shader for model deformation and skinning. Not only were we able to increase the performance of the deformation framework, we also managed to lift the restrictions on the number of bones and vertices of previous implementations. Furthermore, we tested an alternative approach by using dual quaternions for the rigid-body parts of the transformations. This approach reduces artifacts which occur with the matrix-based shader program.

5.2 Limitations

The only limitation comes from the fact that dual quaternions only describe rigid-body transformations. Due to this, it remains to find a way to how to incorporate scale and shear transforms in the skinning procedure. One approach would be include additional matrices which are then blended to the dual quaternion skinned vertices. This would require some additional computational costs. The number of bones our transformation framework can theoretically support by using a single texture unit on the GeForce8 is:

$$floor((8192^2)/3) = 22'369'621$$

Naturally, this is limited by other components such as hardware or the deformation solver on the CPU. And it would only make sense to if the model would as well have at least that many vertices.

5.3 Future work

Considering future work, it remains to integrate the dual quaternions into the whole deformation framework instead of the shader program only. This can be done based on the sketch of the dual quaternion classes provided by this thesis. Furthermore, the above mentioned scale and shear transforms should be implemented.

Another, rather practical test would be to set up a system consisting of two GeForce8 GPUs, a high spec CPU, and 4GB of RAM. This system would be fed with models having a high vertex count and a large deformation graph to see how the performance of the deformation framework scales.

Furthermore, the CUDA framework may offer possibilities for another approach. It would be a good idea, for example, to see if more computational tasks, such as the calculation of the transformations, could be solved on the GPU by using this new GPGPU API.
Finally, there is potential to broaden the functionality of the shader programs with the help of geometry shader programs. As mentioned in the chapter on results, one way to take advantage of the new API would be to dynamically change the deformation graph or the model to be skinned on the GPU.

5.4 Personal experience

At the beginning, I read the material listed in the bibliography section. This gave me a good overview of the research area. To get a better grasp to the subject, I implemented a very simple skinning example in software. For this, I used a little engine which I wrote in a semester project. Once this toy example was finished, I adapted one of the example shader programs for matrix palette skinning to our needs and integrated it into the little engine. Additionally to the common vertex, normal, and color state attributes, I also used multitexturing and secondary colors to pass the needed data to the shader program.

As a next step, I made myself familiar with the existing deformation framework where the GPU skinner would be integrated. Then it was time to integrate the shader into the existing deformation framework. One of the hardest things was to include the center vertices which we did not need for the toy example. In a first version, I used the texture units to pass them to the GPU by glMultiTexture calls. The performance of this first shader, howerver, was not any better than the CPU version, or at least not much.

One of the most challenging and interesting tasks was to take advantage of the new graphics API of the latest GeForce 8. First, I let the Cg compiler produce an assembler output for the latest profile supported, which was the vp30 for the GeForce 7. I started to modify the assembler code in order to use the latest API, but I encountered several problems. I realized, that I had to read a lot of specifications, so I could understand how to incorporate it. NVidia has a OpenGL extensions specification with about 2000 pages; I read about 500 of them. Obviously, it is much easier to work with a high level API, as it is provided in the Cg Toolkit, than with an assembler interface, such as the ARB_vertex_program or the new NV_gpu4_program. Then NVidia released a new Cg version for the GDC 2007 and I could use cgc to produce correct !!NVvp4.0 output. But the performance was still the same.

Then we decided to take another approach, which was, for example, presented at the GDC (Game Developers Conference) 2007. I stored the center vertices and vertex weights, as well as the bone matrices and reference indices as textures and passed them to the GPU. I used the NV_texture_rectangle and NV_float_buffer extensions to do so. With this, we arrived at a shader program which was indeed faster than the CPU pendant of it.

Concerning the dual quaternion approach, we designed new classes to implement the functionality. It remains to test and integrate them into the deformation framework. Furthermore, the scale and shear transforms need to be added to the dual quaternion approach. Right now, these are simply ignored.

We also used the NV_transform_feedback extension to record the output of the vertex shader which consists of the deformed model. Unfortunately, there is a problem with the readback vertices. The model appears to be correct, but it is spatially misplaced.

This thesis offered me a great opportunity to improve my C++ coding skills and knowledge about GPUs, most importantly about the jungle of extensions which are available. Furthermore, I experienced how it is to work with APIs and hardware which is being developed at the moment when one actually would like to use it. Sometimes, this means that one has to wait until a new compiler or SDK is released.

Finally, I would like to thank my mentors for their constant support and encouragement I received while writing this semester thesis.

Bibliography

[1] Mark Pauly, Robert W. Sumner, and Johannes Schmid. Embedded deformation for shape manipulation. *ACM Transactions on Graphics (SIGGRAPH 2007)*, August 2007.

[2] Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Skinning with dual quaternions. In *2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 39–46. ACM Press, April/May 2007.

[3] Doug L. James and Christopher D. Twigg. Skinning mesh animations. *ACM Transactions on Graphics (SIGGRAPH 2005)*, 24(3), August 2005.

[4] Paul G. Kry, Doug James, and Dinesh Pai. Eigenskin: Real time large deformation character skinning in hardware. In *In Proceedings of the ACM SIGGRAPH Symposium on Computer Animation*, pages 153 – 160, July 2002. AVI Video available at: http://www.cs.cmu.edu/ djames/-movies/eigenskin.avi.

[5] Doug James and Dinesh Pai. Dyrt: Dynamic response textures for real time deformation simulation with graphics hardware. *ACM Transactions on Graphics (SIGGRAPH 2002)*, 21(3):582 – 585, July 2002. MPEG Video available at: http://www.cs.cmu.edu/ djames/movies/dyrt.mpg.

[6] Ladislav Kavan, Steven Collins, Carol O'Sullivan, and Jiri Zara. Dual quaternions for rigid transformation blending. Technical report TCD-CS-2006-46, Trinity College Dublin, 2006.

[7] Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158, New York, NY, USA, 2001. ACM Press.

List of Figures

List of Tables

Appendix A

MPS vertex shader


```
/*
 * This is the input structure.
 *
 */
struct appdata {
  float3 Position:  POSITION;
  float3 Normal:    NORMAL;
  float3 Color:     COLOR0;
  float3 NumBones:  COLOR1;
};


/*
 * This is the output structure.
 *
 */
struct vpconn {
```

```
    float4 Hposition: POSITION;
    float4 Color0:    COLOR0;
};

/*
 * This function builds the bone matrix out of the
 * values read from the texture containing the global
 * transforms.
 *
 * It returns a matrix of type float3x4
 * which depicts the bone matrix.
 *
 */
float3x4 buildBone(float4 b1, float4 b2, float4 b3)
{
    return float3x4( float4(b1.xyz, b3.y),
                     float4(b1.w, b2.xy, b3.z),
                     float4(b2.zw, b3.xw)      );
}

/*
 * This function calculates the RECT index
 * into the index and weight texture.
 *
 */
float2 calcRECTIndex(float totIndex, float maxWidth)
{
    float x = totIndex;
    float y = 0.0f;
    while( (x-maxWidth) >= 0.0f)
    {
        x = x - maxWidth;
        y = y + 1.0f;
    }
```

```
    return float2(x,y);
}

/*
 * This is the main function.
 *
 */
vpconn main(appdata IN,
            uniform float3 eyePosition,
            uniform float3 dimTEX,
            uniform float3 lightPosition: state.light[0].position,
            uniform float4x4 mv: state.matrix.modelview,
            uniform float4x4 mvp: state.matrix.mvp,
            uniform samplerRECT _centers: TEXUNIT0,
            uniform samplerRECT _bones:   TEXUNIT1,
            uniform samplerRECT _weights: TEXUNIT2,
            uniform samplerRECT _indices: TEXUNIT3,
            uniform samplerRECT _defPose: TEXUNIT4)
{
    vpconn OUT;

    // initialize temp variables
    float3 finalPos = {0.0f, 0.0f, 0.0f};
    float3 finalNormal = {0.0f, 0.0f, 0.0f};

    // calculate index into the textures
    float2 index = calcRECTIndex(IN.NumBones.z,dimTEX.x);

    // obtain weights and indices from the textures
    float4 weights = texRECT(_weights, index);
    float4 indices = texRECT(_indices, index);


    // switch based on the number of bones to be
    // used for the current vertex
    if(IN.NumBones.x == 0.0f)
```

```
{
    finalPos = IN.Position;
    finalNormal = IN.Normal;
}

else if(IN.NumBones.x == 1.0f)
{
    // obtain center vertex
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    // build the transformation matrix
    float3x4 bone = buildBone( texRECT(_bones,float2(index,0)),
                               texRECT(_bones,float2(index+1.0f,0)),
                               texRECT(_bones,float2(index+2.0f,0)) );
    // finally apply the bone matrix to the vertex position and normal
    finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.x;
    finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.x;

}
else if(IN.NumBones.x == 2.0f)
{
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    float3x4 bone = buildBone( texRECT(_bones,float2(index,0)),
                               texRECT(_bones,float2(index+1.0f,0)),
                               texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.x;
    finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.x;

    center = texRECT(_centers, float2(indices.y,0)).rgb;
    index = 3.0f * indices.y;
    bone = buildBone( texRECT(_bones,float2(index,0)),
                      texRECT(_bones,float2(index+1.0f,0)),
                      texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.y;
    finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.y;
```

```
}
else if(IN.NumBones.x == 3.0f)
{
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    float3x4 bone = buildBone( texRECT(_bones,float2(index,0)),
                               texRECT(_bones,float2(index+1.0f,0)),
                               texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.x;
    finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.x;

    center = texRECT(_centers, float2(indices.y,0)).rgb;
    index = 3.0f * indices.y;
    bone = buildBone( texRECT(_bones,float2(index,0)),
                      texRECT(_bones,float2(index+1.0f,0)),
                      texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.y;
    finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.y;

    center = texRECT(_centers, float2(indices.z,0)).rgb;
    index = 3.0f * indices.z;
    bone = buildBone( texRECT(_bones,float2(index,0)),
                      texRECT(_bones,float2(index+1.0f,0)),
                      texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.z;
    finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.z;

}
else if(IN.NumBones.x == 4.0f)
{
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    float3x4 bone = buildBone( texRECT(_bones,float2(index,0)),
                               texRECT(_bones,float2(index+1.0f,0)),
```

```
                                texRECT(_bones,float2(index+2.0f,0)) );
  finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.x;
  finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.x;

  center = texRECT(_centers, float2(indices.y,0)).rgb;
  index = 3.0f * indices.y;
  bone = buildBone( texRECT(_bones,float2(index,0)),
                    texRECT(_bones,float2(index+1.0f,0)),
                    texRECT(_bones,float2(index+2.0f,0)) );
  finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.y;
  finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.y;

  center = texRECT(_centers, float2(indices.z,0)).rgb;
  index = 3.0f * indices.z;
  bone = buildBone( texRECT(_bones,float2(index,0)),
                    texRECT(_bones,float2(index+1.0f,0)),
                    texRECT(_bones,float2(index+2.0f,0)) );
  finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.z;
  finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.z;

  center = texRECT(_centers, float2(indices.w,0)).rgb;
  index = 3.0f * indices.w;
  bone = buildBone( texRECT(_bones,float2(index,0)),
                    texRECT(_bones,float2(index+1.0f,0)),
                    texRECT(_bones,float2(index+2.0f,0)) );
  finalPos = finalPos + (mul(bone, float4(IN.Position - center, 1.0f)) + center) * weights.w;
  finalNormal = finalNormal + mul((float3x3)bone, IN.Normal) * weights.w;

}

// if normals should not be skinned, simply pass through the given normal
if(IN.NumBones.y == 0.0f)
{
  finalNormal = IN.Normal;
}

// now let's do some lighting
```

```
    finalNormal = normalize(mul(mv, float4(finalNormal,0.0f)));
    float3 lightDir = normalize(lightPosition - mul(mv, float4(finalPos,1.0)).xyz);
    float3 halfVec = normalize(eyePosition + lightDir);

    // set specular power
    float specularPower = 0.5f;
    // compute lighting coefficients
    float4 coeffs = lit( dot(finalNormal, lightDir), dot(finalNormal,halfVec), specularPower);

    // specifiy lighting parameters
    float4 AmbiLight = {0.2f, 0.2f, 0.2f, 1.0f};
    float4 DiffLight = {143.0f/256.0f, 188.0f/256.0f, 143.0f/256.0f, 1.0f};
    float4 SpecLight = {1.0f, 1.0f, 1.0f, 1.0f};

    float4 AmbiMat = {0.3f, 0.3f, 0.3f, 1.0f};
    float4 DiffMat = (IN.Color,1.0f);
    float4 SpecMat = {1.0f, 1.0f, 1.0f, 1.0f};

    float4 EmisMat = {0.1f, 0.2f, 0.3f, 1.0f};

    // add the ambient, diffuse, specular component
    float4 tempColor = AmbiLight*AmbiMat*coeffs.x + // ambient term
                       DiffLight*DiffMat*coeffs.y + // diffuse term
                       SpecLight*SpecMat*coeffs.z;  // specular term

    // finally add the material emission
    OUT.Color0 = tempColor + EmisMat;

    // apply the model-view-projection transform to the vertex position
    OUT.Hposition = mul(mvp, float4(finalPos.xyz,1.0f));

    return OUT;
}
```

Listing A.1: MPS vertex shader

Appendix B

Dual Quaternion vertex shader


```
/*
 * This is the input structure.
 *
 */
struct appdata {
  float3 Position:  POSITION;
  float3 Normal:    NORMAL;
  float3 Color:     COLOR0;
  float3 NumBones:  COLOR1;
};


/*
 * This is the output structure.
 *
 */
struct vpconn {
```

```
    float4 Hposition: POSITION;
    float4 Color0:    COLOR0;
};

/*
 * This function create a quaternion out of the
 * passed 3x3 rotation matrix. The different cases
 * are based on the trace of the matrix.
 *
 */
float4 createQuatFromMatrix(float3x3 matrix)
{
    float trace = matrix._m00 + matrix._m11 + matrix._m22 + 1.0f;
    float4 quat;

    if(trace > 0.00000001)
    {
        float s = 2.0f * sqrt(trace);
        quat = float4( float3(matrix._m21-matrix._m12, matrix._m02-matrix._m20, matrix._m10-matrix._m01)/s, 0.25f*s);

    }else{

        if( (matrix._m00 > matrix._m11) && (matrix._m00 > matrix._m22) )
        {
            float s = 2.0f * sqrt( 1.0f + matrix._m00 - matrix._m11 - matrix._m22);
            quat = float4(0.25f*s, float3(matrix._m01+matrix._m10, matrix._m02+matrix._m20, matrix._m21-matrix._m12)/s );

        }else if(matrix._m11 > matrix._m22)
        {
            float s = 2.0f * sqrt( 1.0f + matrix._m11 - matrix._m00 - matrix._m22);
            quat = float4( (matrix._m01 + matrix._m10)/s,
                           0.25f * s,
                           (matrix._m12 + matrix._m21)/s,
                           (matrix._m02 - matrix._m20)/s   );

        }else{
```

```
        float s = 2.0f * sqrt( 1.0f + matrix._m22 - matrix._m00 - matrix._m11);
        quat = float4( (matrix._m12 + matrix._m21)/s,
                       (matrix._m20 + matrix._m02)/s,
                       (matrix._m01 - matrix._m10)/s,
                       0.25f * s );
    }
  }

  quat = quat/length(quat);
  return quat;
}

/*
 * This function multiplies two quaternions.
 * Note that quaternions are represented as
 * as arrays of float4. Due to this, addition is
 * provided by the shader API.
 * (float4 + float4)
 *
 */
float4 mulQuats(float4 quat_1, float4 quat_2)
{
  return float4( quat_1.w*quat_2.x + quat_1.x*quat_2.w + quat_1.y*quat_2.z - quat_1.z*quat_2.y,
                 quat_1.w*quat_2.y - quat_1.x*quat_2.z + quat_1.y*quat_2.w + quat_1.z*quat_2.x,
                 quat_1.w*quat_2.z + quat_1.x*quat_2.y - quat_1.y*quat_2.x + quat_1.z*quat_2.w,
                 quat_1.w*quat_2.w - quat_1.x*quat_2.x - quat_1.y*quat_2.y - quat_1.z*quat_2.z );
}

/*
 * This function builds the dual quaternion representing
 * the skinning bone out of the values read from the texture
```

```
 * containing the global transforms.
 *
 * It returns a matrix of type float2x4
 * which depicts the dual quaternion.
 *
 */
float2x4 buildBoneQuaternion(float4 b1, float4 b2, float4 b3)
{

    float4 rotQuat = createQuatFromMatrix(float3x3(b1.xyz,float3(b1.w,b2.xy),float3(b2.zw,b3.x)));
    float4 transQuat = {b3.y/2.0f, b3.z/2.0f, b3.w/2.0f, 0.0f};
    return float2x4(rotQuat, mulQuats(transQuat,rotQuat));

}

/*
 * This function multiplies a dual quaternion
 * with the vertex position.
 *
 */
float3 applyDQtoVertex(float2x4 dualQuat, float4 position)
{

    float len = length(dualQuat[0]);
    dualQuat /= len;

    float3 outPosition = position.xyz + 2.0*cross(dualQuat[0].xyz,
                                        cross(dualQuat[0].xyz,position.xyz) + dualQuat[0].w*position.xyz);

    float3 transPosition = 2.0 * (  dualQuat[0].w*dualQuat[1].xyz
                                  - dualQuat[1].w*dualQuat[0].xyz
                                  + cross(dualQuat[0].xyz, dualQuat[1].xyz));

    outPosition += transPosition;
```

```
    return outPosition;
}

/*
 * This function multiplies a dual quaternion
 * with the vertex normal.
 *
 */
float3 applyDQtoNormal(float2x4 dualQuat, float3 normal)
{
    float len = length(dualQuat[0]);
    dualQuat /= len;

    float3 outNormal = normal + 2.0*cross(dualQuat[0].xyz, cross(dualQuat[0].xyz, normal) + dualQuat[0].w*normal);

    return outNormal;
}

/*
 * This function calculates the RECT index
 * into the index and weight texture.
 *
 */
float2 calcRECTIndex(float totIndex, float maxWidth)
{
    float x = totIndex;
    float y = 0.0f;
    while( (x-maxWidth) >= 0.0f)
    {
        x = x - maxWidth;
        y = y + 1.0f;
    }

    return float2(x,y);
}
```

```
/*
 * This is the main function.
 *
 */
vpconn main(appdata IN,
        uniform float3 eyePosition,
            uniform float3 dimTEX,
            uniform float3 lightPosition: state.light[0].position,
            uniform float4x4 mv: state.matrix.modelview,
            uniform float4x4 mvp: state.matrix.mvp,
            uniform samplerRECT _centers: TEXUNIT0,
            uniform samplerRECT _bones:   TEXUNIT1,
            uniform samplerRECT _weights: TEXUNIT2,
            uniform samplerRECT _indices: TEXUNIT3,
            uniform samplerRECT _defPose: TEXUNIT4)
    {
    vpconn OUT;

    // initialize temp variables
    float3 finalPos = {0.0f, 0.0f, 0.0f};
    float3 finalNormal = {0.0f, 0.0f, 0.0f};

    // calculate index into the textures
    float2 index = calcRECTIndex(IN.NumBones.z, dimTEX.x);

    // obtain weights and indices from the textures
    float4 weights = texRECT(_weights, index);
    float4 indices = texRECT(_indices, index);


    // switch based on the number of bones to be
    // used for the current vertex
    if(IN.NumBones.x == 0.0f)
    {
        finalPos = IN.Position;
```

```
    finalNormal = IN.Normal;
  }

  else if(IN.NumBones.x == 1.0f)
  {
    // obtain center vertex
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    // build the dual quaternion representing the rigid-body transformation
    // at the moment, the scale and shear factors are ignored
    float2x4 dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                             texRECT(_bones,float2(index+1.0f,0)),
                                             texRECT(_bones,float2(index+2.0f,0)) );

    // finally apply the dual quaternion bone to the vertex position and normal
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center ) * weights.x;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.x;

  }
  else if(IN.NumBones.x == 2.0f)
  {
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    float2x4 dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                             texRECT(_bones,float2(index+1.0f,0)),
                                             texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center ) * weights.x;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.x;

    center = texRECT(_centers, float2(indices.y,0)).rgb;
    index = 3.0f * indices.y;
    dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                    texRECT(_bones,float2(index+1.0f,0)),
                                    texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center) * weights.y;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.y;
```

```
}
else if(IN.NumBones.x == 3.0f)
{
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    float2x4 dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                             texRECT(_bones,float2(index+1.0f,0)),
                                             texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center ) * weights.x;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.x;

    center = texRECT(_centers, float2(indices.y,0)).rgb;
    index = 3.0f * indices.y;
    dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                    texRECT(_bones,float2(index+1.0f,0)),
                                    texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center ) * weights.y;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.y;

    center = texRECT(_centers, float2(indices.z,0)).rgb;
    index = 3.0f * indices.z;
    dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                    texRECT(_bones,float2(index+1.0f,0)),
                                    texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center ) * weights.z;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.z;

}
else if(IN.NumBones.x == 4.0f)
{
    float3 center = texRECT(_centers, float2(indices.x,0)).rgb;
    float index = 3.0f * indices.x;
    float2x4 dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                             texRECT(_bones,float2(index+1.0f,0)),
                                             texRECT(_bones,float2(index+2.0f,0)) );
```

```
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center ) * weights.x;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.x;

    center = texRECT(_centers, float2(indices.y,0)).rgb;
    index = 3.0f * indices.y;
    dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                    texRECT(_bones,float2(index+1.0f,0)),
                                    texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center) * weights.y;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.y;

    center = texRECT(_centers, float2(indices.z,0)).rgb;
    index = 3.0f * indices.z;
    dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                    texRECT(_bones,float2(index+1.0f,0)),
                                    texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center) * weights.z;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.z;

    center = texRECT(_centers, float2(indices.w,0)).rgb;
    index = 3.0f * indices.w;
    dualQuat = buildBoneQuaternion( texRECT(_bones,float2(index,0)),
                                    texRECT(_bones,float2(index+1.0f,0)),
                                    texRECT(_bones,float2(index+2.0f,0)) );
    finalPos = finalPos + ( applyDQtoVertex(dualQuat, float4(IN.Position - center, 1.0f)) + center) * weights.w;
    finalNormal = finalNormal + applyDQtoNormal(dualQuat, IN.Normal) * weights.w;

}

// if normals should not be skinned, simply pass through the given normal
if(IN.NumBones.y == 0.0f)
{
  finalNormal = IN.Normal;
}

// now let's do some lighting
finalNormal = normalize(mul(mv, float4(finalNormal,0.0f)));
```

```
    float3 lightDir = normalize(lightPosition - mul(mv, float4(finalPos,1.0)).xyz);
    float3 halfVec = normalize(eyePosition + lightDir);

    // set specular power
    float specularPower = 0.5f;
    // compute lighting coefficients
    float4 coeffs = lit( dot(finalNormal, lightDir), dot(finalNormal,halfVec), specularPower);

    // specifiy lighting parameters
    float4 AmbiLight = {0.2f, 0.2f, 0.2f, 1.0f};
    float4 DiffLight = {143.0f/256.0f, 188.0f/256.0f, 143.0f/256.0f, 1.0f};
    float4 SpecLight = {1.0f, 1.0f, 1.0f, 1.0f};

    float4 AmbiMat = {0.3f, 0.3f, 0.3f, 1.0f};
    float4 DiffMat = (IN.Color,1.0f);
    float4 SpecMat = {1.0f, 1.0f, 1.0f, 1.0f};

    float4 EmisMat = {0.1f, 0.2f, 0.3f, 1.0f};

    // add the ambient, diffuse, specular component
    float4 tempColor = AmbiLight*AmbiMat*coeffs.x + // ambient term
                       DiffLight*DiffMat*coeffs.y + // diffuse term
                       SpecLight*SpecMat*coeffs.z;  // specular term

    // finally add the material emission
    OUT.Color0 = tempColor + EmisMat;

    // apply the model-view-projection transform to the vertex position
    OUT.Hposition = mul(mvp, float4(finalPos.xyz,1.0f));

    return OUT;
}
```

Listing B.1: Dual Quaternion vertex shader