FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Dynamische Molekulare Animation: GPU Skinning für Molekulare Daten

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Sebastian Haushofer
Matrikelnummer 01525970

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc. Prof. Ivan Viola
Mitwirkung: Mgr. David Kouřil

Wien, 1. Juli 2018

Sebastian Haushofer	Ivan Viola

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Dynamic Molecular Animation: GPU Skinning for Molecular Data

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Sebastian Haushofer
Registration Number 01525970

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Ivan Viola
Assistance: Mgr. David Kouřil

Vienna, 1st July, 2018

Sebastian Haushofer | Ivan Viola

Erklärung zur Verfassung der Arbeit

Sebastian Haushofer
2123 Kronberg, In Kellerbergen 7

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2018

Sebastian Haushofer

Acknowledgements

Above all, I want to thank my parents for supporting me through my bachelor studies at university and making this possible. Besides, I want to express my gratitude to my supervisors David Kouřil and Ivan Viola for answering my technical questions and providing valuable feedback on my work. Last but not least, I also want to thank everybody who inspires me by coming up with new and original ideas of all kind, all around the world.

Kurzfassung

Vertex Skinning ist ein weitgehend verbreiteter Algorithmus in der Computergrafik und wird exzessiv verwendet um 3D Meshes basierend auf einem Skelett zu animieren. Aus Effizienzgründen wird das Skinning üblicherweise auf der Grafikkarte anstatt des normalen Prozessors durchgeführt. Typische Anwendungsbereiche für den Algorithmus sind Animationsfilme und Computerspiele. Diese Arbeit befasst sich mit dem Skinning von molekularen Daten auf verschiedenen Detaillevels (LODs), anstatt dem Skinning von herkömmlichen Meshes. Dazu werden individuelle Kugeln, welche einzelne Atome von größeren molekularen Strukturen repräsentieren, anstatt von Vertices eines Meshes animiert und gerendert. Die dazu verwendeten atomaren Daten können als mehr als nur Partikel betrachtet werden, da hierarchische Zusammenhänge einzelner Atome, welche größere Gruppen darstellen, bekannt sind. Dadurch ist es möglich die Animation auf verschiedenen hierarchischen Detaillevels (HLODs) abzuspielen. Auf dem atomaren HLOD können sich die einzelnen Atome komplett unabhängig voneinander bewegen. Auf den anderen HLODs werden mehrere Atome nach den hierarchischen Zusammenhängen in Gruppen zusammengefasst und können sich dadurch nur als eine gemeinsame Einheit bewegen. Für jedes HLOD existieren hierbei mehrere Distanz Detaillevels (DLODs), welche sich abhängig von der Distanz der Kamera zum gerenderten Objekt anpassen. Wenn ein einzelnes Atom von einer zu großen Distanz beobachtet wird, würde es beim herkömmlichen Rendern ohne DLODs kleiner als ein Pixel werden, was in Rauschen resultieren würde. Deshalb ist es notwendig, mehrere Atome in Cluster zu vereinigen und sie durch ein Superatom zu ersetzten, wenn sich die Kameradistanz vergrößert.

Abstract

Vertex skinning is a well-known algorithm in computer graphics and is excessively used for animating 3D meshes based on a skeleton. For efficiency reasons vertex skinning is usually performed on the GPU instead of the CPU. Typical use cases of the algorithm include animated movies and computer games. This thesis deals with skinning molecular data on different levels of details (LODs) instead of meshes. Individual spheres which represent atoms of a larger molecular structure, are animated instead of individual vertices of a mesh. The data used for animation is more than just particle data, because the hierarchy of atoms which form larger structures is known. This allows running the animation on different hierarchical LODs (HLODs). On the atom HLOD different atoms can move individually and independently of each other. On the other HLODs several atoms are put into one larger group which is then only allowed to move together as one unit. Apart from the different HLODs, multiple distance LODs (DLODs) are introduced as well. When an individual atom is viewed from a distance large enough, it will become smaller than one pixel which will result in noise. Therefore, it is necessary to cluster multiple atoms together and replace them with one larger super-atom as the camera distance increases.

Contents

CHAPTER 1

Introduction

Cell biology is complex and many new discoveries are made every year, which results in an enormous amount of new data. Some of this data describes the internal structures of a cell on an atom based level in 3D space. The best way to communicate this data to experts and other people is through visualizations. Because 3D worlds are difficult to interpret using only a projection on a single 2D image, it is important that the used visualizations are animated. Creating one specific illustration is not as powerful as developing an interactive system, because it only gives a limited view on a given scene. An interactive visualization allows the users to look at a given scene however they want to, which is important for the learning process. To make interactive visualizations possible real time computer graphics are needed, because the computer cannot know in advance from which point of view the scene will be rendered next, as the users are free to move around as they like.
Nothing in cell biology is static. For example, there are structures responsible for transportation purposes, such as a kinesin motor protein. In a good visualization, the required movements for dynamic objects should look as realistic as possible. One way to achieve this is with the help of skeletal animation. This technique is commonly used on meshes in movie productions and video games for animating characters. We propose a solution for using skinning on molecular data in real time. By leveraging the semantic information that comes with molecular structures, we extend the traditional vertex skinning technique with atom grouping based on semantic features, and rendering on different levels of details (LODs).

1.1 Problem Statement and Aim of the Work

The goal of this thesis is to come up with an efficient solution which performs skinning on molecular data on different hierarchical levels of details (HLODs) and distance levels

of details (DLODs) on the GPU. A HLOD is one specific configuration of all atoms split into multiple larger groups, where all atoms of a group are animated with the same parameters so that they move as one unit. In this context, moving as one unit means that the distances and orientations of the atoms in a group stay the same relative to each other during the entire animation. A DLOD is the clustering of several atoms into one larger super-atom which is rendered instead of all the individual atoms. A DLOD consists of several super-atoms. Each of the HLODs has to operate at different DLODs. Depending on the HLOD different constraints for clustering atoms into larger super-atoms need to be considered. In the atom HLOD, where each atom moves independently of the others, no special considerations need to be taken into account about which atoms are grouped together into super-atoms. However, if the animation operates on a HLOD where several atoms are part of a larger group, the atoms of such a group need to be put into a super-atom before atoms of different structures are merged together into super-atoms.
The main aim for the results of the thesis is to playback realistically and naturally looking animations of atoms grouped in a protein structure. Creating the animation content should be as simple as possible for the user. The idea is to take the necessary protein data from the protein data bank (PDB), which provides all data for free. The PDB file format is a standard which is used to describe where the individual atoms of a protein are positioned, what type they are (for example carbon) and how they are grouped together into larger structures. These files can be imported into 3D modeling software such as Maya or Blender with the help of a plugin. This allows the user to create a skeleton and animations for a protein inside the 3D modeling software. Once the animation is finished it can be exported into a commonly used file format such as collada (DAE) and loaded and displayed by this application. Another goal of the project is that it is only required to create the animations for one skeleton. The other skeletons which are required to playback an animation on different HLODs correctly are then generated automatically within the application.

1.2 Methodological Approach

Several well-known algorithms in computer graphics and software engineering in general are needed to make this project possible. First of all, as already mentioned above, the entire application is fundamentally based on the vertex skinning algorithm. The basic idea is to create a skeleton made out of bones which are organized in a hierarchical tree structure. This means that when a parent bone is moved or rotated, all children bones, children's children bones and so on will move with it to remain in the same relative alignment to their parent. The skin is a metaphor for the mesh which is put on top of the skeleton. Each vertex can be assigned to one or multiple bones with a weight and will then move accordingly with the bones it is assigned to. A more detailed explanation is given in Section 2.1.
In the usual vertex skinning process the bone weights for each vertex are assigned manually. Modern 3D modeling software offers the option to paint the weights onto a texture to speed up the development process and ensure good usability. However, weight

painting is still a very time consuming task which takes a lot of fine tuning. We do not need such an amount of control over the skinning, so we can do the weight assignment based on simple heuristics. Therefore, it should be possible to calculate the weights fully automatically within the application. To achieve this, the highest weight is assigned to the bone with the closest distance to a given vertex according to a distance function. Only bones which are directly connected to that bone in the skeleton's hierarchy are then given a weight as well.
To group several atoms into a larger super-atom for the DLODs a clustering algorithm is required. The k-means algorithm offers a simple yet very efficient solution to this problem and is well explained by Bradley et al. [BF98]. It works very well for this project, because it is designed to group sphere shaped structures of elements together. It takes the number of total clusters the elements of the original set should be grouped into as input and outputs the center points of these new clusters. At first, the algorithm chooses the center points randomly by taking sample values from the original set. It then calculates the distance from each element to each cluster center by some metric which can be chosen based on the given problem. In this case the Euclidean distance works great, because the atoms are positioned in 3D space. Each element is assigned to the cluster it has the closest distance to. Next, the new center is computed for each cluster by taking all elements which are assigned to the cluster into account. Then, the new distances to the cluster centers are computed again and the entire process is repeated until no cluster center changes anymore after another iteration.
A proper approach towards the thesis has been required to solve all these tasks. The skinning has been implemented into the Marion rendering framework, which is further discussed in Section 2.3. At first, a simple test animation has been created with Blender and the basic skinning has been implemented. Afterwards, the skinning on different HLODs and DLODs has been integrated into the framework and a GUI has been created to switch between the different LODs and rendering settings. Then, proper animations have been created and tested within the application to verify that the implementation works as intended. During the entire development process there have been weekly meetings with David Kouřil, the assistant advisor, who has given valuable feedback on the development process and ideas for improvement. Ivan Viola, the advisor, has provided the direction in which the application should be developed and has assigned which features need to be considered. In the end, the results have been shown to Drew Berry, a world class expert in the field of molecular visualizations.

1.3 Structure of the Work

An introduction to GPU vertex skinning and a comparison of different blending techniques and their advantages and drawbacks is given in Chapter 2. It also describes the rendering pipeline of the Marion framework, where the molecular skinning is integrated into. Chapter 3 discusses how the skinning algorithm has been adapted to work with molecular data instead of meshes and introduces a sophisticated approach for calculating bone weights and IDs for the atoms automatically within the application. This chapter also

describes two techniques to run a given animation on different HLODs of atom groups. Besides, a clustering method, which considers the group structure of atoms to show the animation at different DLODs, depending on the camera's distance to the rendered object, is discussed. Chapter 4 explains details of the implementation and its GUI. Suggestions for future improvements, such as a specific file format for animated molecular data, are explained in Chapter 5 and approaches to solve these ideas are discussed. Finally, a summary is given in Chapter 6.

CHAPTER 2

State of the Art

This chapter describes the current state of the art of skinning techniques. Although the different techniques are all based on interpolating position and rotation data between several bones, there are many different ways how this interpolation or blending is done. Here, the different approaches and their advantages and drawbacks are discussed.
Besides, an explanation of the Marion rendering pipeline, in which the molecular skinning is implemented, is given.

2.1 The Vertex Skinning Algorithm

Vertex skinning is a commonly used animation technique in computer graphics which makes heavy use of quaternion math, and is described in detail in the following section. It is also very well explained by Kavan et al. [KŽ03].

2.1.1 Algorithm Overview

Three things are required to make vertex skinning possible: a skeleton, a skin and weights which assign each vertex of the skin to one or more bones of the skeleton.
The bones of the skeleton are organized in a hierarchical tree structure and are moved relative to each other to create a pose. The idea is to put this skeleton in different poses which form key frames for the animation and to interpolate between these poses for rendering individual frames. When a specific bone is moved it also moves all its children bones with it so that they remain at the same relative position to their parent bone. In 3D space such a transformation can be described by a 4x4 matrix. If translations are not allowed in the skeleton and the root bone is centered at the origin, a 3x3 matrix can be used instead. To compute the position of a bone for a specific key frame all the transformation matrices starting from the root node down to the bone itself have to be multiplied recursively. This works, because each matrix represents a coordinate system

transformation relative to its parent. Let M_i be the 4x4 matrix describing the translation and rotation of the i_{th} bone relative to its parent bone. If T_i holds the translation part and R_i the rotation component, M_i is given by Equation 2.1.

$$M_i = T_i * R_i \tag{2.1}$$

Here i represents the i_{th} bone along a specific branch from the root node. The matrix P_i which describes the absolute transformation of the i_{th} bone from the origin can then be calculated with Equation 2.2.

$$P_i = M_0 * ... * M_{i-1} * M_i = \prod_{n=0}^{i} M_i \tag{2.2}$$

This matrix needs to be calculated for each bone and for each key frame separately and can be computed recursively. However, it cannot be directly multiplied with a vertex's position to obtain its animated position, because it represents the transformation from the origin and not from the bone's initial position. Therefore, the matrix O_i which holds the transformation from the origin to the bone's initial position in the default pose needs to be calculated as well. This calculation can also be done with Equation (2.2). The matrix O_i is also known as the inverse bind transform. The final matrix A_i which transforms a bone from its default pose into a specific animation pose is computed with Equation 2.3.

$$A_i = P_i * O_i^{-1} \tag{2.3}$$

Intuitively, Equation (2.3) can be read as, transform the bone from its default pose position to the origin and then transform it to its animated pose position. This also explains why the inverse bind transform O_i^{-1} is used, as it holds the transformation from the default pose position to the origin while O_i stores the opposite.
If a vertex is assigned to be animated by multiple bones, a weight w_i is stored for each bone assignment for the vertex. The sum of all weights for any vertex must always equal 1. The simplest way to calculate the position of a vertex of the skin during animation is achieved by linear blend skinning. Here, the position p of vertex v can be calculated as a weighted average over all n bone transforms which have influence on the vertex with Equation (2.4).

$$p = \sum_{i=1}^{n} w_i * A_i * v \tag{2.4}$$

Assuming a vertex v is only assigned to bone i its position p at a specific key frame during animation can simply be calculated with Equation 2.5.

$$p = A_i * v \tag{2.5}$$

Equation (2.5) is simply a special case of Equation (2.4), where $n = 1$ and $w_1 = 1$. Linear blend skinning can lead to rendering artifacts and has several other flaws. These problems and alternative approaches to solve them are described in the next Section 2.2.

To sum up, the strategy and pipeline to calculate the positions of all vertices for a given moment in time is as follows. First, the current animation time is updated and the two key frames before and after the current timestamp are determined. To retrieve the matrix of every bone i for the current timestamp, an interpolation between the two matrices M_i of the previous and next key frame is done. Then, the matrices A_i are calculated for all bones i with the interpolated matrices M_i, where O_i^{-1} only has to be precomputed once for each bone, as the default bind pose is a constant. The matrices A_i of all bones are passed to the vertex shader as uniform values, because they are the same for each vertex. Afterwards, the position of each vertex is calculated in the vertex shader. To make this possible, it is necessary to pass the weights and IDs of the bones which have influence on a given vertex to the vertex shader as well, for every vertex. This process is also known as GPU vertex skinning. The benefit of performing skinning on the GPU is a massive increase in speed. Only the matrices of the bones need to be calculated on the CPU and the actual positions of all vertices can be processed in parallel on the GPU.

2.1.2 Quaternions

To calculate the positions of all vertices at any point in time between two key frames, it is necessary to interpolate between the transformations of the previous and next key frames. Interpolating the translation part is trivial and can be done by linear interpolation. However, interpolating between rotations is trickier. Euler angles, which are the simplest tool to encode rotations, store the amount of rotation around the x, y and z axes separately. They depend on the order of execution, which means that a rotation given by rotating around the x, y and z axes in this order will not necessarily give the same result as the same rotation executed in a different order like y, z, x. Besides, Euler angles can cause gimbal lock, which is a phenomena that occurs when two of the rotational axes align and one degree of freedom is lost as rotation around any of the two aligned axes will yield the same result. Euler angles cannot be used for interpolation without artifacts in many cases for this reason.
To overcome all of these drawbacks, quaternions are typically used in computer graphics to represent rotations in 3D space. They are explained in great detail by Hanson et al. [Han05] and Dam et al. [DKL98]. Two properties of them are that they avoid gimbal lock and interpolation between them can be computed easily, efficiently and uniquely. On top of that, they are more compact in storage and only require 4 floats instead of 9 floats which are necessary to store a 3D rotation matrix. Quaternions are an extension of complex numbers and have three imaginary units i, j and k instead of one, which gives a total of four variables per quaternion as demonstrated in Equation (2.6).

$$q = z + xi + yj + zk = [z, (x, y, z)] = [z, v] \tag{2.6}$$

These four values are used to encode the axis of rotation, which is a 3D vector n and the angle of orientation around this axis θ, which is a single value. These two variables combined clearly define a rotation in 3D space as shown in Figure 2.1.
Knowing both n and θ the respective quaternion is computed with Equation (2.7).

$$q = [cos(\theta/2), n * sin(\theta/2)] \tag{2.7}$$

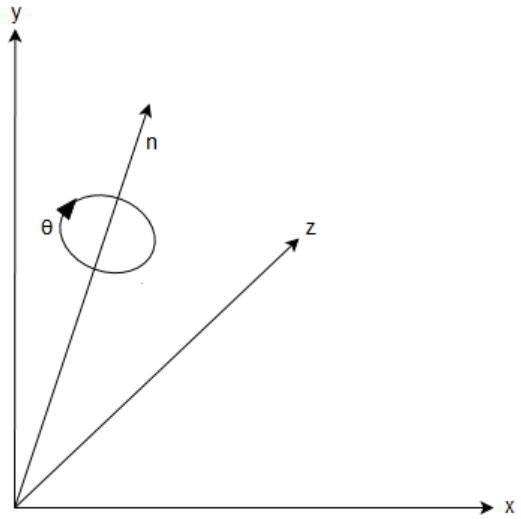


Figure 2.1: Visual representation of a quaternion.

The rules for quaternion multiplication are defined in such a way, that the multiplication of two quaternions gives the combined rotation of both, which means multiple rotations can be applied successively using quaternion multiplication. A detailed proof that this holds is given by Dam et al. [DKL98]. It works the same way as complex number multiplication does. However, there are several additional rules when multiplying the different imaginary units as described in Equation (2.8).

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1 \\ ij = k, jk = i, ki = j \\ ji = -k, kj = -i, ik = -j \end{aligned} \tag{2.8}$$

Besides, interpolation between quaternions works very well. Imagine a quaternion normalized, which means that its length equals one (unit quaternion). Then every possible unit quaternion can be visualized as a point which lies on the surface of a 4D hypersphere. Interpolating between two quaternions then becomes a matter of walking along the shortest arc on the hypersphere between these two points. This interpolation technique is known as quaternion spherical linear interpolation (QSLERP). QSLERP ensures that the rotation changes at constant speed when interpolating between two rotations. The normal quaternion linear interpolation (QLERP) which walks along the straight line between two points on the hypersphere, also known as the chord, would not result in a movement of constant speed. This is because the interpolated quaternions

resulting from QLERP are not of unit length and therefore need to be projected back to the surface of the hypersphere. This projection distorts the point in time when a certain rotation is reached during successive interpolation. However, the interpolated rotations remain correct, only the time when they are reached changes when QLERP is used.
Quaternions can be efficiently converted to rotation matrices and vice versa in linear runtime. This allows them to be used for vertex skinning. At first, the rotation part of the matrix M_i is converted to a quaternion for the previous and next key frame for each . Afterwards, the quaternions are interpolated between the key frames for all bones with QSLERP and converted back to rotation matrices. These matrices are then combined with the interpolated translations and can be used to calculate the matrices A_i for the current timestamp. This process is repeated for every rendered frame.

2.2 Vertex Blending Techniques

Vertex blending is the process of calculating the final animation position of a given vertex based on the bones by which it is moved. Each bone individually would move the vertex to a different position. Thus, these positions need to be blended to obtain a single animation position. This section focuses on the benefits and drawbacks of the different vertex blending techniques. An ideal blending technique should always give a rigid transformation which means that it only consists out of a rotation and a translation after blending, interpolate two transformations along the shortest path and be invariant to the chosen coordinate system.

2.2.1 Linear Blend Skinning

Linear blend skinning (LBS) has already been explained in Section 2.1 and is computed with Equation (2.4). Essentially, this equation transforms a vertex with multiple different matrices and calculates a weighted average of the newly computed positions.
This can lead to visible artifacts in some cases. For example, imagine a vertex right on top of your knee. It would be influenced by your upper leg and lower leg bones approximately by the same amount of around 50 percent each. Transforming the vertex with the upper leg bone would give a specific position and transforming it with the lower leg bone would result in another position. The actual position the vertex is moved to always lies somewhere on the straight line connecting these two positions. The weights control where on the line the vertex is actually drawn in the end. If the lower leg bone was rotated 180 degrees around the bone's axis relative to the upper leg bone, the two calculated points would lie opposite of each other. The calculated average would therefore collapse into the middle and lie straight on the bone as illustrated in Figure 2.2. This artifact is known as the candy wrapper effect, because when rendered, it looks like the mesh is wrapped like a piece of candy at the collapsing point. The artifact is caused because interpolating between matrices does not result in a rigid transformation in many cases. In this scenario the collapsing point is the knee. These artifacts do not become visible in most use cases though, as such large twists are usually not desired anyways,

especially in human models.

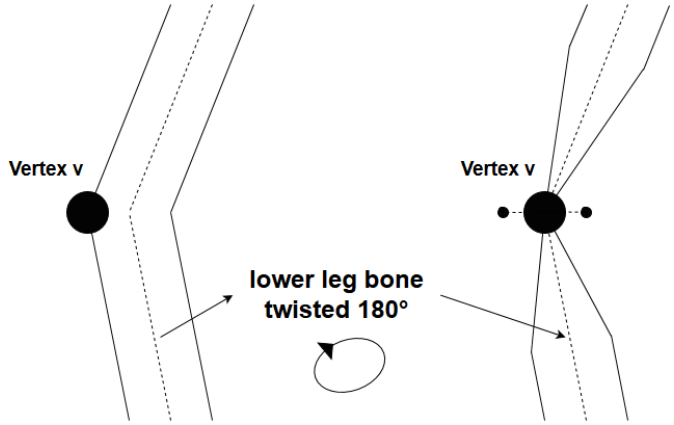


Figure 2.2: The candy wrapper artifact occurs because the two animated positions of a vertex lie opposite of each other with regard to the relevant bones and the vertex's interpolated position therefore collapses to the middle.

The benefit of this technique is that it is very easy and fast to compute and that it will give good results as long as animation scenarios where artifacts would occur are avoided in the first place. It is also independent of the coordinate system and interpolates along the shortest path.

2.2.2 Spherical Blend Skinning

Spherical blend skinning (SBS) avoids the candy wrapper artifact by changing the stage in the pipeline where the blending is applied and has been introduced by Kavan et al. [KŽ05b].
Instead of blending the matrices, the blending is performed on the quaternions representing the rotation parts of the matrices. This way, a vertex will only be transformed to one position, as the rotations are blended before they are applied to a given vertex, which ensures that the blended transformations are rigid. To get a visual analogy to the LBS, a vertex can move on an arc between two points instead of a line when interpolated. This also explains how the candy wrapper effect is avoided, as the average of two points on the other side of each other does not collapse to the center, but moves around it on the arc. The obvious question becomes around which point in space the interpolated rotation should be applied. As it turns out, the ideal choice is the point in space which is as close to itself as possible when it is transformed from its original position by two different matrices. For two bones which are connected this obviously is the contact point of the bones, as it will give exactly the same position when transformed by the two bone's matrices. However, computing the rotation center for n bones which can also be disconnected is a lot more tricky and involves finding the least squares solution of a $\binom{n}{2}$ vector system, because every possible combination of picking two bones out of the n bones has to be

considered. The result of the system is the point in space for which the point's offset caused by two different rotations is minimized summed up over all combinations. This is a typical regression problem and solving this for every possible combination of bones in real time would be a total overkill. Luckily, in real use cases only a few combinations of bone subsets which influence different vertices will occur, as vertices are typically only moved by bones connected to each other. The rotation centers for different bone combinations therefore only have to be calculated once for each frame and can be cached for other vertices which use the same bone combination then. This makes the real time usage of the algorithm possible.
The advantage of SBS is that it overcomes the candy wrapper effect as it always gives a rigid transformation with the cost of lower performance and more complexity than LBS. SBS interpolation is done along the shortest path, but the method is not independent of the coordinate system as the centers of rotation need to be computed.

2.2.3 Dual Quaternion Blend Skinning

Dual quaternion blend skinning (DQBS) always results in a rigid transformation, is coordinate system independent, and interpolates along the shortest path and therefore fulfills all the desired criteria. It is explained with great detail by Kavan et al. [KCŽO07]. As the name suggests, this approach is based on quaternions and dual numbers. The idea of the algorithm is explained without mathematical details in this paragraph. The interested reader may refer to the paper mentioned above for a detailed explanation and the mathematical background of dual quaternions. A dual quaternion stores 8 values instead of 4 which a normal quaternion does. This makes it possible to encode both rotation and translation in a single dual quaternion. It stores the same values as a regular quaternion but also the location of the rotation axis and the offset along the rotation axes in 3D space. Remember, a quaternion can only describe the axis of rotation positioned at the origin without any translation along this axis. Dual quaternions precisely enable these two additional functionalities. The paper proves that computing the weighted average of several dual quaternions with followed normalization will result in a dual quaternion meeting all the criteria mentioned at the top of this section.
To use DQBS for vertex skinning the following steps are required. First, all matrices A_i need to be converted to dual quaternions on the CPU and passed to the GPU. The GPU then calculates the weighted average over those dual quaternions which have an influence on the currently rendered vertex. Afterwards, the interpolated dual quaternion is converted back to a rigid matrix which can then be directly multiplied with the vertex's position to obtain the final position in the current pose. Although this algorithm meets all the ideal criteria, it is slightly slower than LBS, which is why LBS is still commonly used in many systems. However, changing an existing LBS implementation to DQBS only requires small changes in the vertex shader and the conversion of matrices to dual quaternions on the CPU side.

2.3 Marion

Marion is a framework for communicating biology and is described by Mindek et al. [MKS+18]. Marion provides the code base for this implementation. Thus, it is necessary to understand its rendering pipeline in order to be able to integrate skinning into it. One part of Marion deals with rendering large molecular scenes. This part of the framework is based on rendering techniques presented by Le Muzic et al. [LMAPV15]. Rendering molecular data is all about drawing a large amount of atoms, which are represented by spheres, efficiently. Drawing a sphere mesh for every atom would be far too inefficient even if techniques such as instancing were used. Thus, every sphere is rendered using a billboard technique which performs z-depth correction in the fragment shader to give realistic rendering results. To further increase performance only one vertex draw call is invoked for a protein structure built out of up to several thousand atoms. The actual atom positions for a protein are transferred to the GPU with a shader storage buffer object (SSBO) and the according draw calls for the atoms are generated exploiting the tessellation shader. After the tessellation stage, each vertex is transformed to a quad for the billboarding in the geometry shader. Finally, the billboard is rendered in the fragment shader. In the post processing stage, screen space ambient occlusion (SSAO) is performed to increase the depth perception. To avoid rendering thousands of invisible atoms due to occlusion a hierarchical z-buffer is used in combination with spatial constraints, given by the camera's movement in short time intervals.

CHAPTER 3

Methodology

The basic skinning functionality can be applied to molecular data in the same way as to meshes, but it is important that the rendering of the atoms is as fast as possible to enable real time rendering. Therefore, atom draw calls are generated on the GPU and billboards are used for rendering the atoms. The weights for the bones are calculated automatically using the euclidean distance metric. When an animation is rendered on different HLODs, one of two cases can occur depending on the size and structure of the atom groups. In the first case, it is possible to use the same skeleton and animation data as for just a single atom. This holds for small sphere shaped group structures whose size is only a small fraction of the size of the bones of the skeleton. In the second case, it is necessary to generate new bones and new animation data based on the original skeleton and animation. This is required for bigger chain like atom structures whose length exceeds the size of individual bones of the original skeleton. This is due to the fact that atoms at the end of each side of such a chain might be moved by completely different bones of the original skeleton in the original animation. This will make it impossible to move the chain as one unit while keeping different chains connected during animation, if the bones at the ends of the chain have completely different orientations. The solution to this problem is to create a bone for each chain structure and to move both of the bone's ends to the appropriate positions during animation. The different DLODs are computed using the k-means algorithm. For each atom group, one instance of k-means with the appropriate number of resulting clusters is run to ensure that no atoms which belong to different groups are joined together.

3.1 Skinning on Molecular Data

This section describes how the vertex skinning algorithm can be applied to molecular data and how skinning weights for the atoms can be computed automatically.
The calculations of the algorithm do not have to change at all to provide the same

functionality as with meshes. The fact that spheres are drawn for the atoms instead of vertices is irrelevant for the algorithm to operate correctly, as it only considers the position of an element and not its shape. However, the shader rendering pipeline changes, as spheres need to be drawn instead of individual vertices. Each point of a sphere needs to be moved by the same skinning weights so that the shape of the sphere does not get distorted. A straight forward solution would be to render a mesh for each sphere where each vertex of the sphere has the same skinning parameters assigned. This solution has some major disadvantages though. First of all, the rendered sphere would never have the perfect round shape of a real sphere, no matter how many vertices it consists of. Besides, increasing the vertex count would drastically slow down rendering times if thousands of spheres have to be rendered at once. For this reason, a billboarding technique is used, where a camera aligned quad which only consists of four vertices is rendered with the help of a geometry shader for each sphere. Depth correction inside of the fragment shader due to ray-casting then allows reshaping the quad to a perfect round sphere from the camera's point of view. This approach also rapidly speeds up rendering times, as only four vertices need to be drawn per sphere. Apart from that, the rendering calls for the atoms of a protein are invoked with the help of a tessellation shader to decrease vertex shader invocations. Thus, the actual skinning algorithm is executed inside of the tessellation evaluation shader instead of the vertex shader as in ordinary vertex skinning. The weights for each atom are calculated fully automatically inside the application, which saves valuable time. This way weight painting is not required at all and it is also possible to work with a mesh approximation of the actual protein which is being animated in the 3D modeling software. This is possible, because the atom data of a protein has to be loaded from a PDB file either way to obtain the hierarchical structures of the atoms and the animation data is loaded separately. Computing the weight for a specific atom-bone combination is always done based on the distance between them. As proteins are 3D structures which occur in the real world it makes sense to compute the Euclidean distance, although other distance metrics could be used too. Calculating the distance between a bone and a sphere is the equivalent of computing the closest distance from a point to a line which has a start and an end point. There are three different cases which need to be considered as illustrated in Figure 3.1.

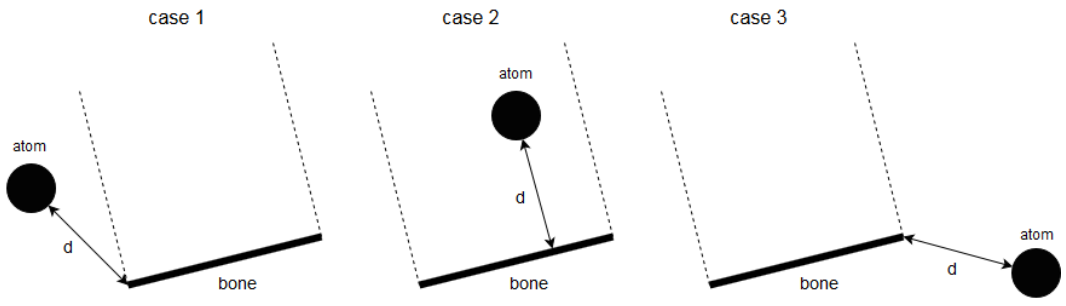


Figure 3.1: Calculating the distance from a sphere to a bone has three different cases.

In the first and third cases, the closest distance is along the line connection the sphere center with the bone start and end point respectively. In the second case, the shortest distance is along the bone's normal vector. The distance d_i from a sphere center to bone i can be calculated with Equation (3.1), where C_i defines which of the three cases needs to be considered and can be calculated with Equation (3.2). It can be computed with the dot product. A_i is the start point and B_i the end point of the i_{th} bone. V is the center position of the atom. This method to compute the distances is also described by Li et al. [LLLW18].

$$d_i = \begin{cases} \left|\overrightarrow{A_iV}\right| & \text{if } C_i < 0 \\ \left|\overrightarrow{B_iV}\right| & \text{if } C_i > 1 \\ \frac{\left|\overrightarrow{A_iV} \times \overrightarrow{B_iV}\right|}{\left|\overrightarrow{A_iB_i}\right|} & \text{if } C_i \geq 0 \text{ and } C_i \leq 1 \end{cases} \tag{3.1}$$

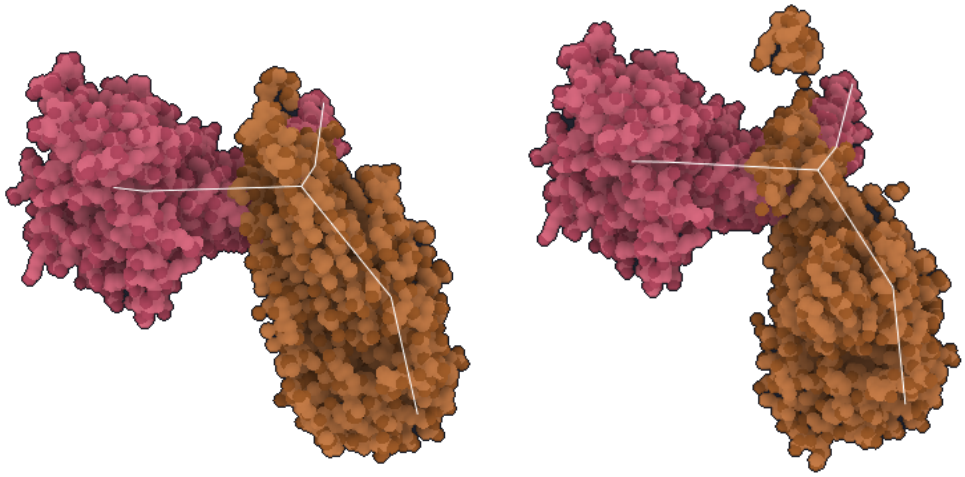
$$C_i = \frac{\overrightarrow{A_iV} \cdot \overrightarrow{A_iB_i}}{\left|\overrightarrow{A_iB_i}\right|^2} \tag{3.2}$$

Once the distances from an atom to all bones are known, there are several options on how to calculate the actual weight values. Generally, bones with closer distances have a larger weight than bones further away. Experimentally, it has been established that Equation (3.3) gives very good looking results. After the calculation, the four largest weights are chosen and normalized so that their sum equals one.

$$w_i = \frac{1}{{d_i}^3} \tag{3.3}$$

Another question that naturally arises is if all bones should be allowed to have an influence on a given vertex. Imagine a human skeleton and a point on the top of your right, upper leg, located on the inner side of the leg. This point is only slightly further away from the left leg's upper bone than the right leg's upper bone. Therefore, it will be moved by both bones by the same weight almost equally, which is not desirable. The movement of one leg should be independent of the other. To overcome this problem, the bone with the largest weight should be chosen as the main bone and only its parent and children bones should be assigned weights as well.

The results of both methods are shown in Figure 3.2. The white lines on top of the atoms symbolize the bones of the skeleton. Figure 3.2b shows the problematic that arises when only allowing connected bones of the main bone to have an influence on an atom. If the closest bone is not the bone which should have the main influence, the approach does not work. In this figure some atoms which belong to one leg are closer to the other leg. Consequently, they move along with the wrong leg which is the case for the orange atoms in the top right corner of the image. One solution to overcome this problem is to make sure that all atoms are closest to the bone they should be assigned to as main



(a) All bones have an influence. (b) Connected bones have an influence.

Figure 3.2: Comparison of assigning weights to all bones or only bones directly connected to the bone with the nearest distance.

bone, when building the skeleton.

3.2 Animation on Different Hierarchical Levels of Details (HLODs)

The PDB file format does not just describe particle data, hierarchical structures of the individual atoms of a protein are also known and can be considered during animation. The goal is to move those atoms which build a larger unit together as one piece. This means, that the space in between them stays the same when they move. The orientation usually has to stay the same too, but as the atoms are only rendered as spheres the orientation can be ignored, because changing it does not affect the visual result in any way. Depending on what a group of multiple atoms looks like, two different cases need to be considered when applying a given skeleton animation on them. In the first case, it is possible to use the same skeleton and animation data as when the atoms are animated individually and independently of each other. In the more complicated scenario a new skeleton and a fitting animation need to be computed. In the end of the section a comparison of the visual differences when animating on different HLODs is given.
The PDB file gives information about the residue structures of the atoms. These structures are shaped like spheres and usually only consist of low atom counts in the scope of tens or hundreds. One such residue is small in comparison to the size of a bone which is used for the skeleton of the protein. For this reason, the weights of two atoms on the opposite end of such a group structure do not vary a lot. Therefore, calculating

a new weight for the center of mass of the structure gives accurate animation results and artifacts due to the united movement of the structure are contained. The center of mass p_c can be calculated as the average of all atom positions p_i inside of a group with Equation (3.4), assuming they have the same size. If the size of the atoms varies, different weights can be assigned to them, depending on their size.

$$p_c = \frac{1}{n} * \sum_{i=1}^{n} p_i \tag{3.4}$$

Each atom of a group has to be assigned to a bone with the same weight to ensure that the group moves as a unit. Grouping atoms is another reason why calculating the weights is done within the application. This way the weights for a group can be calculated with the same algorithm as for an individual atom, using the center of mass as input position. If no such algorithm was built into the application it would be necessary to compute the weight for the group based on the weights of the individual atoms. However, computing the new group weight as the average of the individual atom weights gives incorrect results, because the weights do not change linearly with the distance, but averaging the weights of multiple atoms is a linear operation carried out over the distance. Consider calculating the correct weight for a group consisting of two atoms for one particular bone. The right part of Equation (3.5) calculates the correct weight with the averaged position as input. The left part computes the average of the calculated weights of the two atoms, which does not give the same result at all.

$$\frac{\frac{1}{{x_1}^3} + \frac{1}{{x_2}^3}}{2} \neq \frac{1}{\left(\frac{x_1+x_2}{2}\right)^3} \tag{3.5}$$

To express this in a more general mathematical statement, Equation (3.6) holds when Equation (3.3) is used as $f(x)$.

$$\frac{f(x_1) + f(x_2)}{2} \neq f\Big(\frac{x_1 + x_2}{2}\Big) \tag{3.6}$$

Using the same skeleton for arbitrary group structures of atoms does not always work, especially if a structure has at least the size of a skeleton's bone and stretches over at least two bones of the skeleton. Imagine a chain structure of atoms which begins in the middle of one bone and stretches all the way to the middle of the next bone. The center of mass would then lie approximately on the point where the two bones connect and the chain would thus be moved by both bones equally. If these two bones bent, the two ends of the atom chain would lie equally far away from the bones as illustrated in Figure 3.3. To express this in simpler words, the problem is caused because the atoms at the end of the chain would require completely different weights than the weight calculated for the center of mass, in order to be placed at the correct positions. However, if all atoms of a group have to move as one unit they require the same weights. This gives a conflict with the previous statement. Thus, using the original skeleton for large group structures does

not work, unless all chains are directly placed on top of the skeleton's individual bones. This scenario is usually not given, because the animator often needs more flexibility with placing the bones.

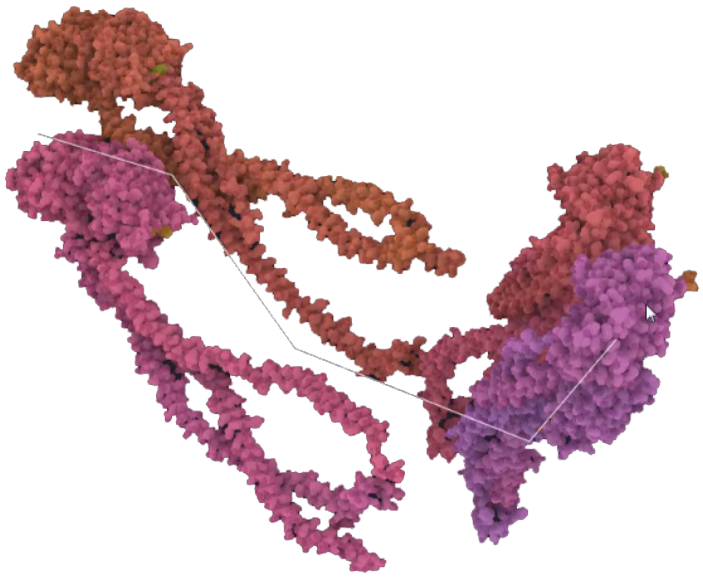
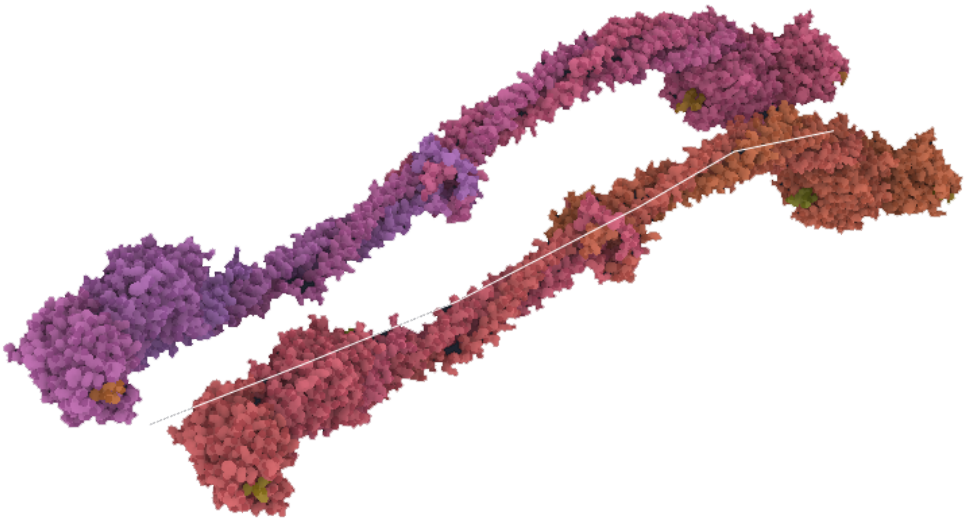


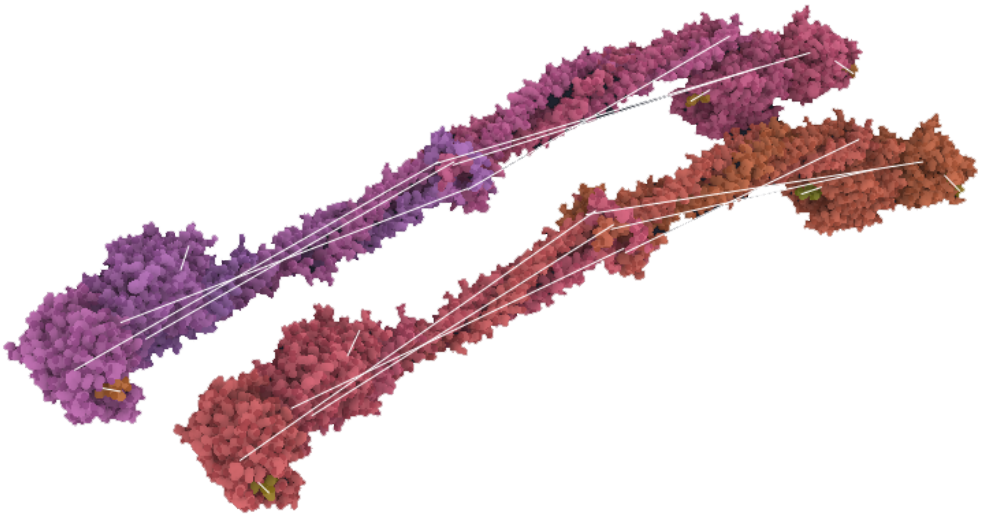
Figure 3.3: The chains break apart because their atoms are all moved by the same weight parameters as their center of mass.

To overcome this problem, a skeleton is built on top of the chains where each chain is assigned to its own bone which stretches across the entire chain. All these bones have the same root parent bone which is positioned at the origin. The animated positions for each new bone are then calculated for every key frame. Consider one particular bone of a chain. Using the original skeleton and animation data, it is possible to calculate where the start and end points of this bone are moved to, for every key frame. This works using the ordinary skinning algorithm, where the weights are simply computed for this bone's start and end points. The new position data is only calculated for each key frame for each bone once and is then stored as the new animation data for the new skeleton. Using this approach, every atom of a given chain is only assigned to its chain's bone with a weight of one. This technique ensures that a chain keeps its orientation correctly during animation. A comparison of an animated pose between the atom HLOD and the chain HLOD is shown in Figure 3.4. This figure also shows the original skeleton (Figure 3.4a) and the automatically calculated skeleton (Figure 3.4b) for the chains in white.

The chain groups are stored in the PDB file and are retrieved from there. Calculating the initial bone positions for a given chain can be done as following. First, an axis



(a) The original skeleton of the protein used for the atom HLOD and residue HLOD.

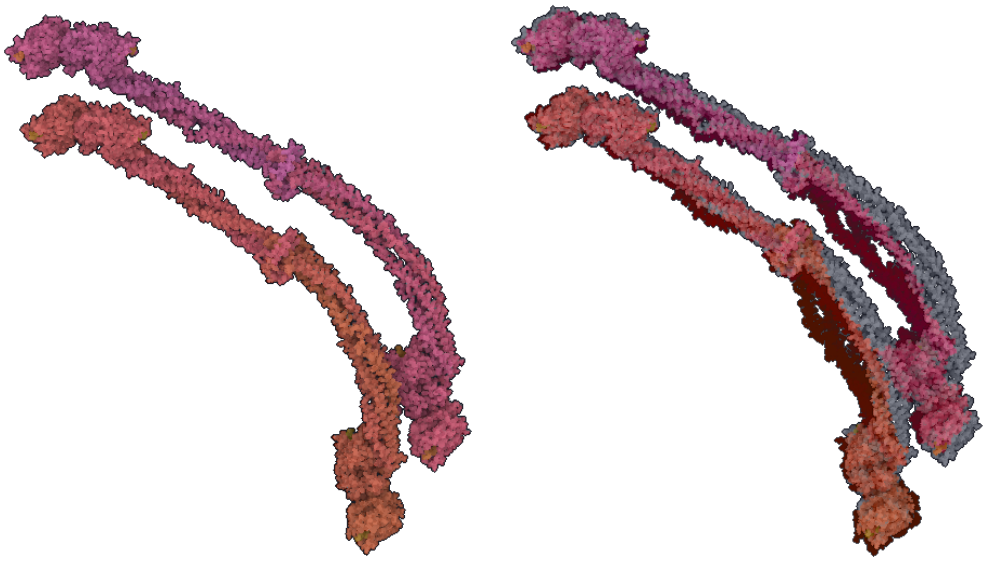


(b) The skeleton to animate group structures in the chain HLOD.

Figure 3.4: Comparison of the skeletons used for different HLODs.

aligned bounding box (AABB) is built around the entire chain. This works by saving the minimum and maximum values of the atoms' positions for each of the x,y and z axes separately. Once all atoms of the group are processed, the minimum and maximum values of all axes are joined into two points in 3D space which represent the minimum and maximum coordinates of the AABB. The axis along which the AABB stretches most is then determined. The center of mass of all atoms of the chain, which lie withing the first ten percent of the AABB's length on the determined axis, is then calculated. The

same is done for the last ten percent. The resulting centers of mass are used as start and end points of the bone for this chain.
One issue of this animation technique, which uses a new skeleton, is that overlaps or gaps between chains can occur. Think of a scenario where one chain stretches over several bones of the original skeleton. If these bones were bent closer together during animation the atoms of the chain would just bend with the bones in the atom HLOD. However, the chain is not allowed to bend in any way in the chain HLOD, because the entire idea was to move the chain as one unit. Using the method described above, the animated positions of the chain's start and end points would be closer together as they originally were in this scenario. One solution to this problem would be to scale the chain and the atoms accordingly, but this is an undesired effect, as the atoms' sizes must not change during animation. The best solution to the problem is to keep the bone and atoms at the original size and share the bone's extra length equally on both ends. The only way to avoid these artifacts completely is to ensure that a chain is not bent by multiple bones in the original animation.



(a) Overlay atom - residue HLODs. (b) Overlay atom - chain HLODs.

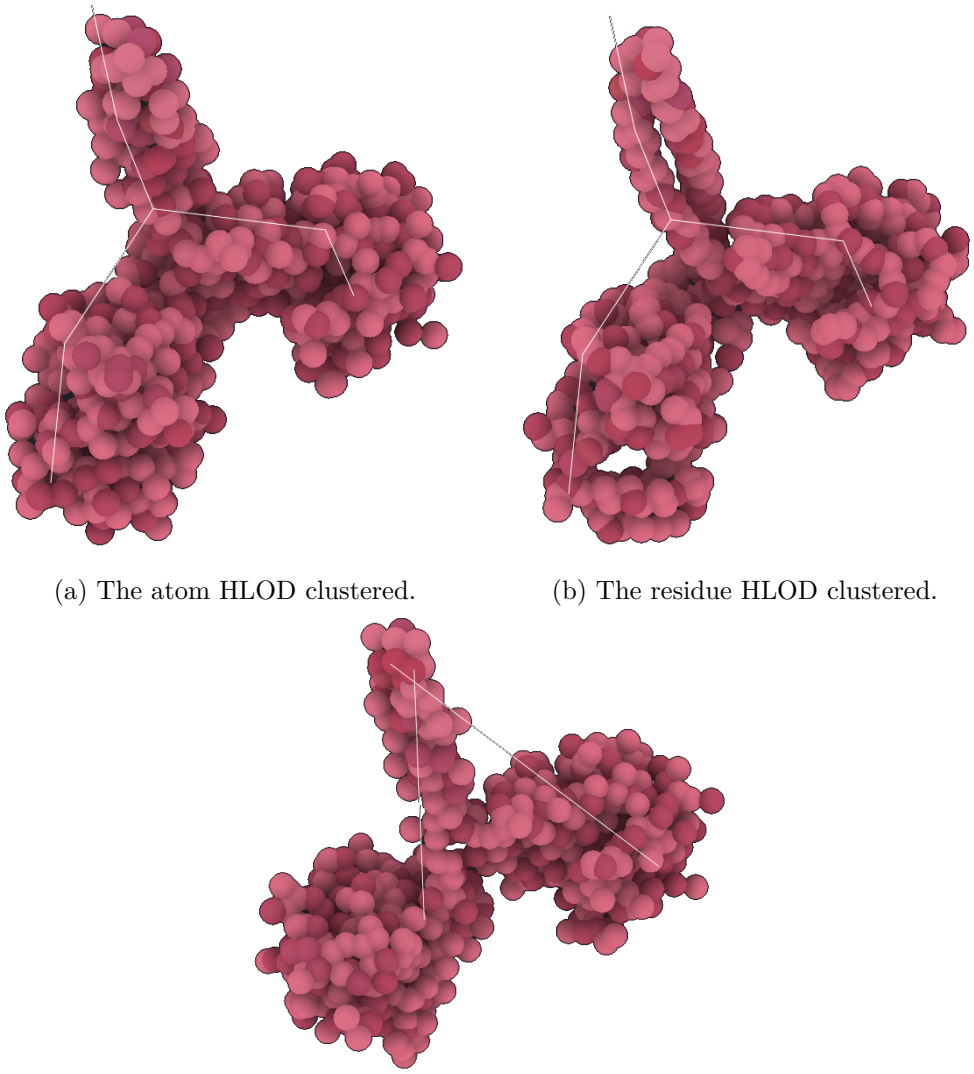
Figure 3.5: Rendering overlay between the atom HLOD (gray) and the other HLODs (color) for a specific key frame.

Figure 3.5 illustrates the rendering differences when animating on the different HLODs. The gray chains represent the atom HLOD and the transparently overlaid chains in color the other HLODs respectively. The visual difference between the atom HLOD and the residue HLOD are barely noticeable at the nearest DLOD as shown in Figure 3.5a. This is because the residues only consist of very few atoms, which are positioned very close to each other. On positions where multiple bones connect a visual difference becomes

more noticeable, because weights become more sensitive to smaller positional differences. In the figure there are a few gray spots which represent a distinction between the two HLODs. A visual difference becomes apparent when comparing the atom HLOD to the chain HLOD like in Figure 3.5b. As each chain is forced to move as one unit and chains are larger structures, the group cohesion becomes noticeable.

3.3 Animation on Different Distance Levels of Details (DLODs)

Atoms are clustered together into super-atoms when the camera moves further away from the visualized protein as described by Le Muzic et al. [LMAPV15]. All atoms of such a cluster are then rendered as one larger super-atom. Depending on the HLOD the protein is currently animated on, the different group structures of the atoms need to be considered for the clustering.
Super-atoms are also rendered as spheres and should therefore consist of atoms positioned in a sphere like cluster. The k-means algorithm which is explained in Section 1.2 and also by Bradley et al. [BF98] results in sphere shaped structures which makes it perfect for this application scenario. Other clustering algorithms such as agglomerative clustering often result in arbitrary shapes. For example, agglomerative clustering, which is a hierarchical algorithm, does not consider the distance to the center of a structure but the distance to any element of the group when joining different elements or groups together. K-means ensures sphere like structures and is also simple to implement and fast.
For the atom HLOD the k-means algorithm can simply be run for all atoms of the protein together, as there are no constraints about which atoms can be fused together. However, for the residue and chain HLODs the algorithm is run once for every group structure. This ensures that atoms of different groups are not joined into super-atoms. The DLOD at which the protein is rendered is determined in the vertex shader based on the camera distance to the protein. The weights and IDs for the super-atoms can be computed exactly the same way as for regular atoms. The super-atoms and their corresponding weights and bone IDs are calculated once at application startup and are then stored in SSBOs. Figure 3.6 shows all three HLODs rendered at the second DLOD in comparison. In the residue HLOD in Figure 3.6b the structure of the chain becomes visible best. In the actual chain HLOD in Figure 3.6c the chain's shape does not become this apparent, because there are no constraints about which regions of the chain are put together into super-atoms. The chain itself consists of the residue structures, which give valuable information for the chain about which atoms make sense to be put into a cluster. The residues give the chain less flexibility where to position the individual super-atoms. This explains the better distribution of super-atoms and the more obvious chain look. In the atom HLOD in Figure 3.6a there are no constraints for the clustering and no grouping structure is recognizable.



(a) The atom HLOD clustered.

(b) The residue HLOD clustered.

(c) The chain HLOD clustered.

Figure 3.6: All three HLODs rendered at the second DLOD. In 3.6b and 3.6c the super-atoms are created with respect to the group structures and the shape of the chains becomes visible. In 3.6a there are no constraints for creating super-atoms and therefore no hierarchical structure is recognizable.

CHAPTER 4

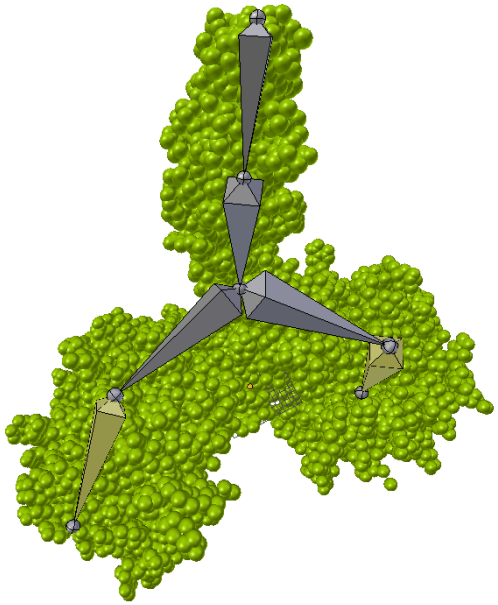
Implementation

This chapter discusses how the animation framework is implemented and why it has been designed the way it is. It also describes the GUI of the application and how to use it.

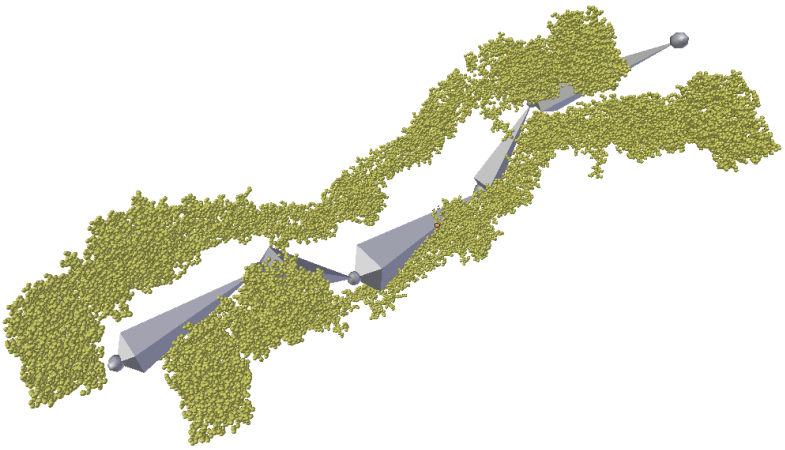
4.1 Implementation Details

As already mentioned in Section 2.3, the entire application is implemented into the Marion framework, which is programmed in C++ and OpenGL. The shader code is therefore written in GLSL. The application uses the Qt library for OpenGL wrapper functionalities, the GUI, and additional data structures like 3D vectors and quaternions and their operations.
The developed animation framework has three major components which are the skeleton, the actual animation data and the bone weights and IDs. The skeleton is stored as a tree of individual bones. Each bone is represented by a class and stores references to its parent and all its children bones. Besides, the inverse bind transform and the initial positions of the start and end points of the bone are stored. The animation data is saved in several key frames. Each key frame stores a map of the bones and their relative transforms to their parents for the current pose. Each bone's transform is stored as a 3D vector for the translation part and a quaternion for the rotation part to allow smooth interpolation between the key frames as described in Section 2.1. The matrix representation of the transform is calculated after the interpolation.
The skeleton and animation data is loaded from a collada (DAE) file at application startup. Collada is a common standard for mesh and animation data, which is based on XML and an export option for all major 3D modeling software tools. The test animations have been created with Blender as illustrated in Figure 4.1. The Assimp library by Schulze et al. [SGK+12] is used to parse the collada files and extract the relevant information. Using a library like Assimp has several benefits. First of all, it supports all common 3D file formats, which means that loading the animation data from other file formats such as

FBX should work without having to change anything in the code at all. Apart from that, it saves valuable implementation time which is wiser spent elsewhere, as writing your own XML parser is a quite time consuming task which would not result in any benefits compared to using a library.



(a) Animation of a kinesin protein.



(b) Animation of the human fibrinogen.

Figure 4.1: Test animations created in Blender.

The molecular data includes the atoms' positions, types, sizes and group structures and is

loaded from a PDB file. The PDB file is necessary, because only it contains information about the hierarchical group structures of the atoms, required for the HLODs. Storing this information in file formats such as collada is not possible in their standard, because defining relationships between individual vertices apart from edges or faces is not a desired feature in usual application scenarios of meshes and therefore not supported. It might be possible to convert the atoms' group relationships into face data, but this would require exploiting the intended purpose of the format and some tricky parsing to fit in the data properly. Instead of doing this, it would make a lot more sense to define a new file format, which is specifically designed for animating molecular data. This is a great idea for future work for several other reasons as well and is further discussed in Section 5.2.
The weights and bone IDs are computed once at application startup for each HLOD on all different DLODs. To speed up this process the k-means algorithm for clustering is executed in a compute shader. This is very efficient, because k-means can be executed in many small parallel tasks. The atom positions and bone weights and IDs are passed to the shaders via SSBOs at application startup. The transformation matrices for all bones are set as a uniform array in the shaders for each frame.

4.2 GUI

The GUI of Marion has been extended so that the user is able to control animations as well. The added animation tab allows the user to switch between the three supported HLODs (atoms, residues, chains) with a drop down menu at any time. Besides, it is possible to enable and disable the rendering of both the original skeleton and the skeleton for the chain HLOD independently with a check box. Finally, the basic animation control operations, which are play/ pause and jumping between the individual key frames of the animation are supported by buttons. The animation GUI is shown in Figure 4.2.

The settings for the HLODs can be controlled with the LOD settings tab which has already been used in Marion. This allows the user to specify at which camera distances which HLOD is chosen for rendering and how big the corresponding super-atoms are rendered.

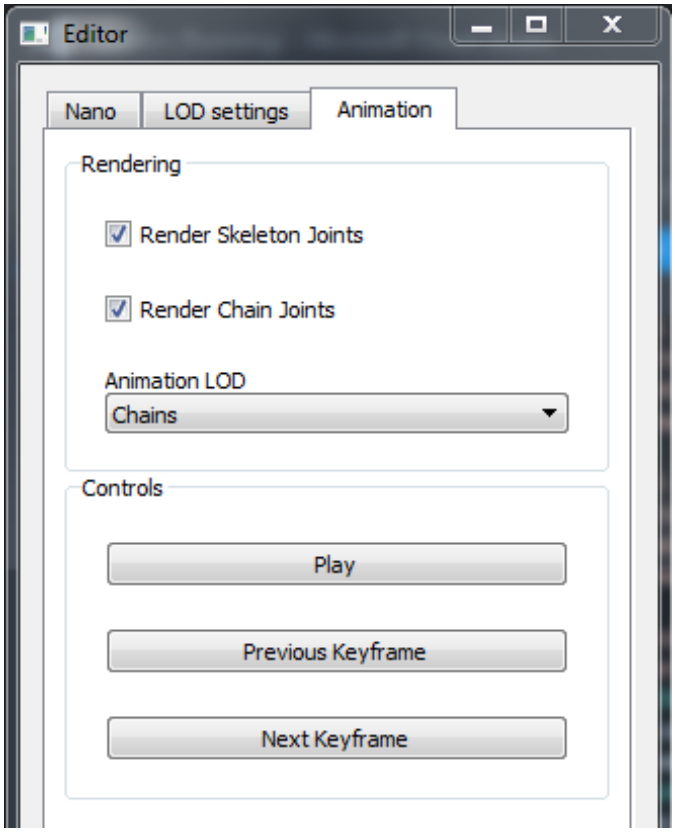


Figure 4.2: A simple GUI to control animations in Marion.

CHAPTER 5

Discussion

There are only few similarities between this project and other research done in the field of skeletal animation. The current research in the field focuses on different aspects, such as capturing motion data and automatically generating animations from it. Many ideas, such as an own file format and proper collision handling, exist for future development of this application.

5.1 Comparison with Related Work

To my best knowledge there is no related work that deals with skeletal animation on molecular data, especially on multiple HLODs and DLODs. These two features are the main contribution of this thesis. The research in the field of skeletal animations has taken a different path in recent years. Researchers have been working on automatically creating the skeleton and/ or animation data from meshes or motion data. Baran et al. [BP07] describe an approach for automatically computing skeleton data from a given mesh. James et al. [JT05] automatically compute the skeleton and animation data from a sequence of meshes, so that only one mesh needs to be stored and the shapes of the other meshes are rendered by running the calculated animation. In comparison to these approaches, the skeleton for the chain HLOD can be computed with a much simpler method in this application, because constraints for creating the skeleton are known. The constraint is that each chain, whose atom members are always known, should be moved by exactly one bone, which stretches along the entire chain.

5.2 Future Work

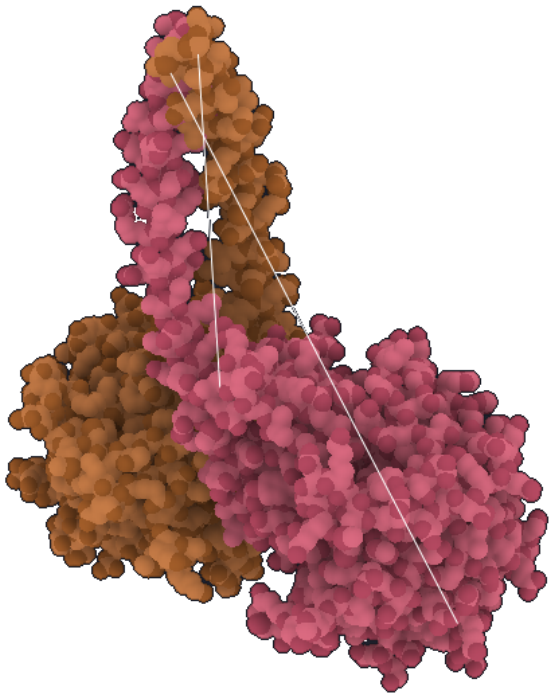
There are some technical aspects of the implementation which have room for improvements and other features which would be nice to have in the future.
As explained in Section 4.1 two different files are currently required to load a protein

animation. The collada file stores the skeleton and animation data and the PDB file holds the atom data. This is not really an issue, because this solution works fine, but it is not good practice to load data which belongs together from two different sources. A better solution would be to create an own file format which is tailored for the purpose of molecular animation. I will refer to this format with MOL for molecular from here on. A good idea would be to define an XML scheme for it and save the data in appropriate XML files. This would not only allow saving all of the atom positions, their group structures, the skeleton and the animations, but would also enable storing the required data for all HLODs and DLODs and not just the nearest atom HLOD. This brings up the next issue, which is that all HLODs and DLODs are currently computed at application startup. This is fine for a proof of concept which this application is, but it does not scale well. Computing this data for a scene with only one animated protein is fast, but imagine a scene with hundreds or thousands of different proteins. In this case, it would be far too inefficient and an unnecessary computational overkill. The HLOD and DLOD information for a given protein does not change as long as the chosen HLOD super-atom counts stay the same, so computing this data once and storing it in something like a MOL file is an important feature for a production level application. The MOL file format would store the (super) atom positions and their according bone weights and IDs for every HLOD at every DLOD. Each group structure of every HLOD would store references to the atoms which belong to it for all the DLODs. However, to use the MOL format effectively, it would be required to write a plugin for a 3D modeling tool of choice which is able to store the created scene in this format. Alternatively, an easier solution could be to write a converter which takes the collada and the PDB file which belong together as input and creates the according MOL file as an output. Using the MOL file format, the actual visualization application would only need to read in all the data without having to calculate any weights or clusters at startup.

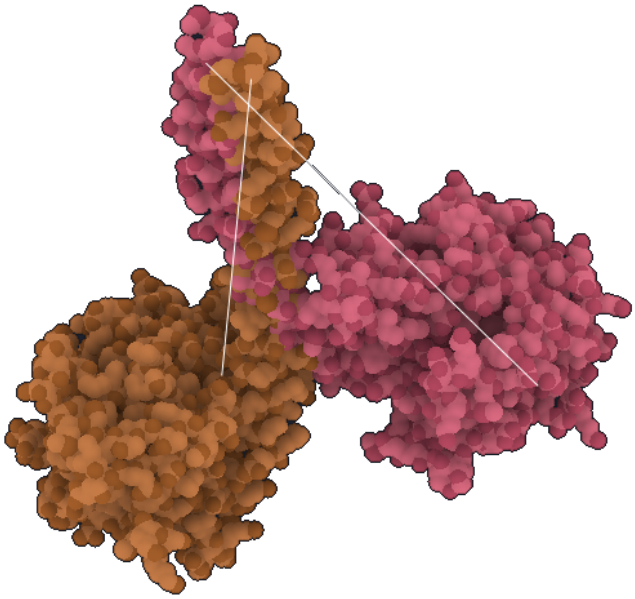
One issue which has not been covered within the scope of this project are collisions of all kind. In general two different types of collisions need to be distinguished, which are collisions with other objects in the scene and self-collisions. Collisions with other objects should not even have to be considered at the moment, as the animated objects are not designed to be moved interactively and the animator should make sure that the different animated objects do not overlap at any time. Methods to detect collisions with other objects have been developed by Macagon et al. [MW03] and Kavan et al. [KŽ05a] and are based on moving mesh approximating bounding spheres or cylinders, according to the animation.

Self-collisions on the other hand only start to become a real issue when skinning is applied to molecular data. This is due to the fact that atoms which are represented by spheres are animated instead of vertices which represent an infinitely small point in space. Using a mesh any given triangle which is animated can be distorted, but the vertices of that triangle will never collide unless they are moved to the exact same point in space. However, atoms do collide a lot easier, as they are not just a point in space but an actual sphere object. This means, that two atoms collide when they get closer to each other than the sum of their radii. Meshes only have the problem of self-collision if

two usually disconnected parts of a mesh are moved too close to each other by different bones, for example if a hand is moved into the stomach for a human mesh. Solutions for this type of problem exist and are based on position based dynamics (PBD), as described by Mueller et al. [MHHR07]. The idea of PBD is to define constraints so that objects do not collide with each other, which are then resolved iteratively and independently of each other until a solution for all constraints is found. The principle of PBD can also be applied to skinning of meshes to avoid self-collisions and is discussed by Bender et al. [BDDZ13]. The idea is to put a layer built of oriented particles on top of the mesh and resolve the collisions between these particles. Using particles enables to drastically reduce the required amount of objects which need to be considered for intersections. For meshes it is not necessary to use a particle for every vertex, as adjacent vertices will never collide. Thus, multiple vertices can be assigned to one particle. This is not the case for molecular data, which means a particle is required for every atom. This results in a massive decrease in rendering speed. An idea to overcome this issue is to precompute the collisions and store a positional correction value for every atom for specific key frames. These key frame intervals might need to have a tighter time interval than the key frames used for interpolation between animations. This is because an atom can collide with several different atoms between two animation key frames, as the movement from one key frame to another is not constrained to be linear at all. Collisions can be calculated in advance because the animation does not depend on any external influences. This technique would work by first applying the ordinary skinning algorithm to every atom and then adding the interpolated positional offset due to the collisions to the atom's calculated skinning position.
The last issue is that chain groups of atoms can collide with each other in the chain HLOD in special cases. This is due to the fact that the entire chain is moved as a unit with the same weight and bone parameters. Atoms close to each other therefore can be moved by completely different parameter settings if they do not belong to the same chain. Figure 5.1 demonstrates the problem. This is an extreme case as the two chains are moved by the same bone at the atom HLOD, but by two different bones at the chain HLOD for the region at the top of the image where the chains are twisted with each other. In this animation, the chains are supposed to represent legs and are animated to do a walking cycle. Because the chains are constrained to move as one unit, they go straight through each other if they cross each other when going from one extreme of the walking cycle to the other, as Figure 5.1b shows. At the two extremes of the cycle there is no collision as visualized in Figure 5.1a. To solve this issue, PBD could be used to define constraints, which make sure that the atoms of a group do not move apart. Of course the collision constraints between different chains would still be required as well. Just like the inner collisions, this could also be calculated in advance and a positional correction value could be stored for each chain.



(a) The chains before collision.



(b) The chains during collision.

Figure 5.1: Collision occurs at the chain HLOD due to over-stretching.

CHAPTER 6

Summary

A system which runs molecular animations based on a skeleton in real time has been presented. It does not only allow running an animation on individual atoms which are independent of each other, but also enables animation playback where entire group structures of atoms move together. This is the main contribution of this thesis. Two different methods to achieve this have been explained. The first approach can be applied to small sphere shaped group structures and computes the same skinning weights for all atoms of a group using its center of mass. It is necessary to have an algorithm which calculates the skinning weights for any position in 3D space built into the application to make this work. An algorithm to achieve this which works based on distances using the Euclidean metric has also been introduced. The second group structure method can be applied to large chain shaped structures and generates a new skeleton for the animation automatically. Each chain is assigned to exactly one bone and moves with it accordingly. The bones' positions during animation are calculated based on the original animation data. Apart from that, a smart clustering method to run a given animation on different LODs depending on the camera's distance to the object has been implemented to solve issues with rendering noise. This is the second major contribution. Depending on the group structures which are currently considered during animation, atoms of a group are clustered into larger super-atoms using the k-means algorithm. The total number of rendered super-atoms depends on the camera's distance and is configured to avoid rendering noise. Ideas for future improvements are collision detection and resolution based on PBD between individual atoms and an own file format to store all of the required animation data in one file. The results have been presented to Drew Berry, a world class expert in the field of molecular visualization, in an informal session. He described the implemented system as extremely useful for his work, because a system for visualizing complex 3D animations for molecular data in real time has been lacking so far. He also mentioned the potential of skeletal animations being used for other purposes, for example untangling RNA whose information is being read by ribosomes.

Bibliography

[BDDZ13] J Bender, J Dequidt, C Duriez, and G Zachmann. Physically-based character skinning. 2013.

[BF98] Paul S Bradley and Usama M Fayyad. Refining initial points for k-means clustering. In *ICML*, volume 98, pages 91–99. Citeseer, 1998.

[BP07] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Transactions on Graphics (TOG)*, 26(3):72, 2007.

[DKL98] Erik B Dam, Martin Koch, and Martin Lillholm. *Quaternions, interpolation and animation*, volume 2. Datalogisk Institut, Københavns Universitet Copenhagen, 1998.

[Han05] Andrew J Hanson. Visualizing quaternions. In *ACM SIGGRAPH 2005 Courses*, page 1. ACM, 2005.

[JT05] Doug L James and Christopher D Twigg. Skinning mesh animations. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 399–407. ACM, 2005.

[KCŽO07] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 39–46. ACM, 2007.

[KŽ03] Ladislav Kavan and Jiří Žára. Real time skin deformation with bones blending. 2003.

[KŽ05a] Ladislav Kavan and J Žára. Fast collision detection for skeletally deformable models. In *Computer Graphics Forum*, volume 24, pages 363–372. Wiley Online Library, 2005.

[KŽ05b] Ladislav Kavan and Jiří Žára. Spherical blend skinning: a real-time deformation of articulated models. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 9–16. ACM, 2005.

[LLLW18] Jun Li, Feng Lin, Xiuling Liu, and Hongrui Wang. High performance automatic character skinning based on projection distance. *3D Research*, 9(1):9, 2018.

[LMAPV15] Mathieu Le Muzic, Ludovic Autin, Julius Parulek, and Ivan Viola. cellview: a tool for illustrative and multi-scale rendering of large biomolecular datasets. In *Eurographics Workshop on Visual Computing for Biomedicine*, volume 2015, page 61. NIH Public Access, 2015.

[MHHR07] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

[MKS+18] Peter Mindek, David Kouřil, Johannes Sorger, Daniel Toloudis, Blair Lyons, Graham Johnson, M Eduard Gröller, and Ivan Viola. Visualization multi-pipeline for communicating biology. *IEEE transactions on visualization and computer graphics*, 24(1):883–892, 2018.

[MW03] Vadim Macagon and Burkhard Wünsche. Efficient collision detection for skeletally animated models in interactive environments. *Proceedings of IVCNZ'03*, pages 378–383, 2003.

[SGK+12] T Schulze, A Gessler, K Kulling, D Nadlinger, J Klein, M Sibly, and M Gubisch. Open asset import library (assimp). *Computer Software, URL: https://github.com/assimp/assimp*, 2012.