EXTRACTION OF SKINNING DATA
BY MESH CONTRACTION WITH
COLLADA 1.5 SUPPORT
(Master’s Thesis)
MARTIN MADARAS

Advisor: Mgr. Ing. Tomáš Ágošton
Bratislava, 2010
I hereby declare that I have written this thesis by myself, only with help of referenced literature, under the careful supervision of my thesis advisor.

..................................
Abstract

The most common approach to animate models and determine their shape attributes in computer graphics is using skeletons. The skeleton and skinning weights can be either assigned manually or computed from an input mesh. This Master’s Thesis proposes the extraction of a skeleton and skinning weights from a mesh, describes how to store computed data in Collada 1.5 and use it for an animation. Firstly, the mesh is contracted using constrained Laplacian smoothing in a few iterations. Then the most important vertices from the contracted mesh are chosen as control points. Multiple edges are removed and vertices that are very close to each other are merged. We select and collapse a vertex pair with the minimum cost in every iteration using a greedy algorithm. The greedy selection is applied repeatedly until we have the requested number of bones. In the next step the skinning weights are computed according to if we want rigid or soft skinning. In the postprocessing stage the user can inspect the skeleton by previewing skinning deformations, make desired changes and export the skeleton to Collada 1.5. Transformation matrices used in a hierarchical skeleton tree are not transformed to joint’s local transformation frame, so they are immediately compatible with majority of animation software and libraries. After the Collada file containing the mesh, the skeleton and the skinning data is exported, data can be imported in animation software such as 3D Studio Max, Blender or Maya and a skinning animation can be rendered.

Keywords: skeleton extraction, mesh contraction, Collada 1.5, skinning
Abstrakt

Najčastejší prístup na animovanie modelov a získovanie vlastností ich tvaru v počítačovej grafike je použitím kostier. Kostra a kontrolné váhy potrebné na animáciu môžu byť nastavené manuálne alebo vypočítané z geometrie vstupného modelu. V tejto diplomovej práci popisujeme extrahovanie kostry a kontrolných váh zo vstupnej geometrie, spôsob ako ich uložiť do formátu Collada 1.5 a následne použiť na animáciu. Najprv sa vstupná geometrická siet stlačí pomocou kontrolovaného vyhľadzovania s použitím Laplaciana v niekoľkých iteráciách. Následne sú zo stlačenej siete vybrané najdôležitejšie vrcholy ako kontrolné body. Pre tento výber je použitý greedy algoritmus, ktorý v každej iterácii vyberie páry minimálnym ohodnotením. Viacnásobné hrany sú takisto odstránené a na konci extrakcie kostry máme žiadaný počet kostí. V nasledujúcom kroku sú vyrátané kontrolové váhy podľa toho či chceme pevnú alebo voľnú animáciu povrchu. Užívateľ môže vyskúšať kostru, spraviť potrebné úpravy a exportovať dáta do Collady 1.5. Žiadaná transformačná matice používaná v kostrovom strome nie je transformovaná do lokálnej bázy uzla, takže matice sú priamo kompatibilné s väčšinou animačných softvérov a knižníctvom. Po exportovaní geometrickej siete, kostry a animačných dát môže byť vyrenderovaná animácia povrchu v animačných softvúroch ako 3D Studio Max, Blender alebo Maya.

Kľúčové slová: extrakcia kostry, kontrakcia siete, Collada 1.5, animácia povrchu
# Contents

1 Introduction .......................... 1

2 Related work .......................... 2
   2.1 Skeleton extraction .................. 2
   2.2 Skinning ................................ 3

3 Proposed solution ...................... 5
   3.1 Graph conversion ...................... 5
      3.1.1 Joining and splitting of objects .... 5
   3.2 Mesh contraction ...................... 6
      3.2.1 Laplacian smoothing ................ 7
      3.2.2 Linear equation .................... 9
   3.3 Skeleton construction ................. 10
      3.3.1 Mesh graph simplification .......... 11
   3.4 Binding skin vertices ................. 12
      3.4.1 Skinning weights .................. 13
      3.4.2 Joint matrices ..................... 15
   3.5 Collada 1.5 support .................. 15

4 Implementation .......................... 16
   4.1 Three layer model ..................... 16
      4.1.1 Interface ......................... 16
CONTENTS

4.1.2 Engine ........................................ 16
4.1.3 Core ........................................ 17
4.2 Language and libraries ............................... 17
4.3 Application work flow ............................... 18
4.4 Graph conversion ................................... 18
  4.4.1 Visualization of vertex joining ................. 20
4.5 Mesh contraction .................................... 21
4.6 Skeleton construction .............................. 21
4.7 Collada DOM ...................................... 24
4.8 Skinning on GPU .................................... 27

5 Results .............................................. 29
  5.1 Dependency on resolution and dimensions ........... 33
  5.2 Robustness ...................................... 34
  5.3 Computation time .................................. 34
  5.4 Further applications .............................. 35
    5.4.1 Detection of symmetry axis from the skeleton .... 35
    5.4.2 Automatic segmentation of model into primitives .... 36

6 Conclusion ......................................... 40
  6.1 Future work ...................................... 41
    6.1.1 Simplification of the mesh in the preprocessing stage .... 41
    6.1.2 Inverse kinematics for the skinning deformation .... 41
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Joining and splitting of the mesh graph</td>
<td>6</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of surface smoothing</td>
<td>8</td>
</tr>
<tr>
<td>3.3</td>
<td>Opposite angles corresponding to the edge</td>
<td>8</td>
</tr>
<tr>
<td>3.4</td>
<td>Half-edge collapse</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>Computation of the geodesic distance</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Work flow supported by our application</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Joining visualization</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Contraction of a pig model</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Contraction of a hand model</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Extraction of the skeleton from low resolution geometry</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of our skeleton to the manually rigged one</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Extraction of the skeleton from high resolution geometry</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of application of QEM method</td>
<td>42</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A frequently used approach for animation and modification of 3D models is based on creating articulated hierarchical structures - skeletons. Skinning data as a skeleton tree and weights can be either assigned manually or computed from an input mesh. The first option is most often chosen by artists, although sometimes it is unnecessary and time consuming. The skeleton has to be created (or imported from templates), rigged into the mesh and influence weights have to be set. Skilled artists are able to create and rig the skeleton in a short time, but sometimes they have to make a lot of rigging adjustments during the skinning process. In this Master’s Thesis we present how to automatically compute the hierarchical skeleton and skinning weights from an input mesh and use them in a skinning animation using Collada 1.5 as export format between our application and a graphic animation software such as 3D Studio Max, Blender or Maya. Our application also provides a way how to examine the computed skeleton before exporting. The skeleton can be inspected by applying skinning deformations using direct kinematics. The interface also allows to dynamically add or remove a bone or a branching if the user thinks some changes are needed.
Chapter 2

Related work

2.1 Skeleton extraction

Numbers of algorithms have been proposed to compute a skeleton from the mesh geometry. There are three main groups of algorithms: volumetric methods, example based methods and geometric methods. In this Master’s Thesis we focus on geometric methods, they are the most suitable for meshes, because there is no conversion needed. Furthermore, the volumetric methods share common drawback of potential loss of details caused by inappropriate discretization resolution. Example based methods are not applicable as well, because we expect as an input model mesh defined by one pose only. The geometric methods work directly on polygon meshes. The most widely used geometric methods are Reeb graph based methods [AHLD07], Voronoi diagram based [DS06] and Laplacian smoothing based methods [ATC+08].

Voronoi diagram based methods compute the skeleton from the medial axis obtained by extracting the internal edges of the Voronoi diagram. These methods are quite slow in comparison to Reeb graph or Laplacian smoothing based ones and do not guarantee that the result will capture all desired features.

Reeb graph based methods need a suitable real-value function, defined on
the model surface, for a successful extraction of a skeleton. Using this function, nodes of 1D graph can be computed. This graph encodes topology of the mesh and after resampling it is used as a base for the skeleton. The method based on a harmonic function proposed by [AHL07] captures after resampling all the features of the model well, but requires the user to specify the boundary condition explicitly.

Laplacian smoothing based methods work directly on the mesh geometry. The main idea of this approach is to apply a well defined filter on mesh vertices. Some of these methods solve the Laplacian system with uniform [SCO04] and some with different weights to constrain the global smoothness and the volume preservation [NISA06, ATC+08].

A few more approaches to the skeleton extraction problem are worth to mention. [TT98] extract skeleton by simplifying the Voronoi skeleton with a small amount of user assistance. [IWM+03] use repulsive force fields to find a skeleton. The problem has received a lot of attention in recent years and yet the design of a simple and robust method for extracting curve-skeletons remains a research challenge [CSM07].

2.2 Skinning

Many types of mesh deformations can be performed by a skeleton-driven deformation. However, there are types of mesh deformations such as wrinkles, skin folds and another non-bone-driven deformations, where skeleton-driven deformation is not a sufficient option. Examples of non-bone-driven deformation methods are surface based methods [LSLCO05, YZX+04] and volume based methods [ZHS+05]. Unfortunately, these methods are not suitable for a real time animation of high resolution meshes in present. Because of its efficiency and simple GPU implementation the most popular skeleton-driven method still remains linear blend skinning (LBS), also known as skeleton subspace deforma-
tion. Some real time skinning works have focused on improving the LBS by inferring the character articulation from multiple meshes.

A few solutions to the problem of finding skinning weights were proposed [BP07], but the methods are either resolution dependent [KT03] or the weights do not vary smoothly along the mesh [WP00], causing artifacts with high resolution models.
Chapter 3

Proposed solution

3.1 Graph conversion

For running our graph algorithms we need to have the input mesh as one connected object. The object needs to be converted into a 3D graph defined by an edge matrix $E$. It is quite common that models are composed of more objects. These objects appear to be connected visually, but edges in the model structure between these objects are not defined - Figure 3.1a. Also the opposite problem has to be considered. There can be edges defined in an input mesh which connect parts that should not be connected - Figure 3.1b. These edges are remains of the work of graphic designers or artifacts after format conversion and therefore have to be excluded.

3.1.1 Joining and splitting of objects

Using a simple depth-first search suitable joining distances can be found to connect or disconnect all graph components. The algorithm works in two phases. First, we construct a mesh graph from an input mesh. This mesh graph is constructed in a straight-forward way from the original model structure and
CHAPTER 3. PROPOSED SOLUTION

Figure 3.1: (a) A situation, where the joining of the mesh graph is needed. (b) An opposite approach, the splitting of the mesh graph is needed.

may consist of components. In the next step, we compute distance between each pair of components. For each component, the joining distance is computed as a minimum of distances to each other component. In the second phase, we construct the mesh graph again, using joining distance tolerance computed in the first phase. This means that each vertex is joined with vertices which lie in the joining distance radius. This condition joins the closest vertices in neighbouring components which creates a one-component graph. An opposite approach can be used when we want to avoid cycles in the final skeleton. We can either compute or manually set splitting distance tolerance. All the edges with a distance smaller than this tolerance will be removed.

3.2 Mesh contraction

For contraction of the generated mesh graph we use the contraction algorithm using Laplacian smoothing proposed by [ATC+08]. The algorithm does not alter geometry connectivity (final skeleton curve is homotopic to the original mesh), is noise sensitive and works directly on the mesh geometry (the model does not have to be resampled). Geometry contraction removes details from the surface by applying Laplacian smoothing.
3.2.1 Laplacian smoothing

Vertex positions are smoothly contracted along their normals by solving the equation (3.2). The Laplacian smoothing operator (3.1) was introduced by [DMSB99] for surface smoothing. An example of the surface smoothing can be seen in Figure 3.2. Laplacian smoothing operator $L$ is the $n \times n$ square matrix. This operator is applied on $n$ vertices in vector $V$ as a filter. Term $LV$ approximates curvature flow normals, so solving $LV' = 0$ removes normal components of vertices and contracts the geometry, resulting into a new set of vertices $V'$.

$$L_{ij} = \begin{cases} w_{ij} = \cot \alpha_{ij} + \cot \beta_{ij} & \text{if } (i, j) \in E \\ \sum_{(i,k) \in E} -w_{ik} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

(3.1)

where:

$E$ - is the set of edges defined in the previous section during a graph conversion process

$\alpha_{ij}, \beta_{ij}$ - are the opposite angles corresponding to the edge $(i, j)$ - Figure 3.3

The Laplacian coordinates $LV = [\delta_1^T, \delta_2^T, \delta_3^T, \ldots, \delta_n^T]^T$ can be referred to as contraction constrains, because they provide the forces to contract the mesh. The term $\delta_i$ equals $-4A_i k_i n_i$, where $A_i$, $k_i$, $n_i$ is the area of adjacent faces surrounding the vertex $i$, the approximate local mean curvature and the approximate outward normal of vertex $i$, respectively. Thus, for vertices with the same shape of the surrounding area, the scale of the Laplacian operator coordinate is proportional to the length of the edges adjacent to the vertex.
CHAPTER 3. PROPOSED SOLUTION

Figure 3.2: Example of using the Laplacian smoothing operator by [DMSB99] for surface smoothing. Stanford bunny: the original mesh (left) and mesh after the smoothing operator was applied (right).

Figure 3.3: Opposite angles corresponding to the edge used in curvature flow calculation [DMSB99]. A vertex $x_i$ and its adjacent faces (left) and one edge corresponding to one term used in the calculation of the operator $L$ (right).
CHAPTER 3. PROPOSED SOLUTION

3.2.2 Linear equation

Unconstrained solving of this equation contracts the mesh graph into a single point, so the equation is solved in more iterations with carefully chosen weights which control the contractions. $W_L$ and $W_H$ are diagonal weighting matrices which control the contraction process. Weighting matrices have to be updated after each iteration to drive the iteration process into a desired state. By increasing $W_{L,i}$ we can increase the collapsing speed for vertex $i$ and by increasing $W_{H,i}$ we increase the attraction weight to attract vertex $i$ to its current position. All the $W_{L,i}$ are in the next step multiplied by a predefined constant ($s_L$) and $W_{H,i}$ are updated in such a way that the attraction weight is multiplied by a ratio of the change of the area of faces adjacent to the vertex $i$. If the area of adjacent faces surrounding vertex $i$ is smaller, the multiplicative term is higher. Thus, the vertex $i$ is more attracted into its current position and in the next iteration the geometry is less contracted in the vertex $i$. 

\[
\begin{bmatrix}
W_L \\
W_H
\end{bmatrix} V' = 
\begin{bmatrix}
0 \\
W_H V
\end{bmatrix} \tag{3.2}
\]

Each step of iterative contraction process works as follows ($t$ denotes the iteration number):

1. Solve \[
\begin{bmatrix}
W_L^t L \\
W_H^t
\end{bmatrix} V^{t+1} = 
\begin{bmatrix}
0 \\
W_H^t V^t
\end{bmatrix}
\]

2. Update $W_{L,i}^{t+1} = s_L W_{L,i}^t$ and $W_{H,i}^{t+1} = W_{H,i}^0 \sqrt{A_{i,j}^0 / A_i^t}$, where $A_{i,j}^0$ and $A_i^t$ are the original and current areas of adjacent faces for vertex $i$, respectively.

3. Compute the new Laplacian operator $L^{t+1}$ with the vertex positions computed from previous iteration $V^{t+1}$ using equation (3.1).

The iteration process converges when the volume is close to zero. After each iteration, the volume approximation has to be computed. In our implementation
we used an approximation algorithm which subdivides the bounding box of the model into an octree structure.

During the contraction process, while constructing a new Laplacian operator, we have to check for degenerate faces to avoid any possible numerical errors such as infinity values or division by zero. If a face is contracted into a zero area, one of the opposite angles or both angles can be 0° or 180°. The term $L_{ij}$ contains cotangent functions of these angles and can reach infinity values. Such values are unacceptable, because the linear solver works with float arithmetics and the values prevent the solver to converge successfully. We deal with the situation by collapsing such faces into a single point. This point is afterwards moved into the center of the collapsed face. The collapse does not alter manifold connectivity and avoids the creation of infinity values during the construction of the Laplacian operator.

3.3 Skeleton construction

The contracted mesh graph from last iteration is simplified, very close vertices are merged and a greedy algorithm is used to select the most important control points. In this section we are going to work with already contracted vertices from previous section, they will be denoted as $\tilde{v}_i$. During the collapsing process, for each control point, the collapsed vertices into this control point are stored in a hash map. Vertices are merged into control points and every control point is shifted into the center of its local mesh area which can be computed from the stored hash map. For each control point, the volume of the mesh region it represents in the original mesh is computed. The control point which represents the largest volume is chosen as the skeleton root.
3.3.1 Mesh graph simplification

The first step in the mesh graph simplification is to collapse all edges, whose vertex distance is smaller than a predefined threshold. Vertices in such pairs can be interpreted as the same control point, because the distance between them is very small and the influence over the mesh vertices is almost the same. The second step is to collapse edges which are the least important. For every edge the cost value is computed and edges with minimum cost are collapsed. For simplicity we apply half-edge collapse. The half-edge collapse \((i \to j)\) merges vertex \(i\) to vertex \(j\) and removes all the faces that are incident to the collapsed edge. The half-edge collapse \((\tilde{v}_2 \to \tilde{v}_1)\) is shown in Figure 3.4. This step is repeated so many times that in the end we will have the desired number of control points. The cost value is computed as a weighted sum of a sampling cost term and a shape cost term.

The sampling cost term (3.3) penalizes collapsing which generates long edges, because in that way we will lose the good mapping between the skeleton and the surface. The term is computed as a weighted sum of distance of adjacent vertices to the collapsed vertex.

\[
SCT_a(i, j) = \left\| \tilde{v}_i - \tilde{v}_j \right\| \sum_{(i,k) \in \tilde{E}} \left\| \tilde{v}_i - \tilde{v}_k \right\| \tag{3.3}
\]

where:

\(\tilde{E}\) – is the current simplified edge set
CHAPTER 3. PROPOSED SOLUTION

The shape cost term (3.4) works in almost the same way as in QEM simplification method [GH97] with one change. The error matrices are computed over the edges, because the contracted mesh has zero area faces, so the original volume based approach cannot be used. A symmetric $4 \times 4$ matrix $Q$ is associated with every vertex. $Q$ is defined in such a way that term $F_i(p) = p^TQ_ip$ is a squared distance between point $p$ and the edge $(i, j)$. The initial error matrix for vertex $i$ is the sum of all squared distances to its adjacent edges. For more detailed description of these matrices, their initialization and their use in calculation of the shape cost term we refer to [ATC+08]. For a given contraction $(\tilde{v}_i, \tilde{v}_j)$ a new matrix $Q$ needs to be derived to approximate the error at $\tilde{v}_j$. Error matrices from previous iterations are stored, so each cost update involves only matrix addition. The shape cost term guarantees to keep the shape of the contracted mesh graph as undisturbed as possible during the simplification. The idea to assign these cost terms to each edge after the iterative contraction converges successfully origins from [ATC+08].

$$\text{SCT}_b(i, j) = F_i(\tilde{v}_j) + F_j(\tilde{v}_j)$$

(3.4)

3.4 Binding skin vertices

Once we get the skeleton, we bind the mesh vertices to its joints. If we attach a rigid model, the skin is supposed to be inflexible. Therefore we only anchor a mesh vertex to one nearest control point. In other way, when we want to animate a character, we want the vertices to transform smoothly. In this case, mesh vertices have to be anchored to more control points with corresponding weights.
3.4.1 Skinning weights

Skinning indices are computed by finding a set of closest control points to each vertex. The geodesic distance is used as a distance measure. A distance between each pair of vertices on the mesh graph from 0th iteration (after conversion from an input mesh) is calculated and stored in matrix $D$. It is calculated using Floyd-Warshall algorithm [CLRS90], before the mesh graph is contracted. For each control point $C_k$, the closest mesh graph vertex is found, for instance $v_j$. Then, the resulting geodesic distance (3.5) between the control point $C_k$ and the mesh vertex $v_i$ is computed as a sum of distance between $v_j$ and $v_i$ on the mesh graph calculated by Floyd-Warshall algorithm and the euclidean distance between $C_k$ and $v_j$. The illustration is shown in Figure 3.5.

$$gd(i, k) = D[i, j] + d(C_k, v_j) \quad (3.5)$$

where:

$d(C_i, v_k)$ – is euclidean distance between the mesh vertex $v_j$ and $k^{th}$ control point $C_k$

Weights (3.6) are assigned in a way that weight sum for each vertex is equal to 1.0. Fractions are constructed in a way that weights are indirectly dependent on the geodesic distance. This construction guarantees that the closer control points will have greater influence over mesh vertices than the further ones. The geodesic distance is a real-value function defined on the mesh surface. Because the function varies smoothly along the mesh, the resulting weights are fluently distributed over the mesh regions.

$$weight(i, k) = \frac{1}{\underset{k' \in S}{\sum} \left( \frac{1}{gd(i, k')} \right) \frac{1}{gd(i, k)}} \quad (3.6)$$
Figure 3.5: This figure shows computation of the geodesic distance between the control point marked with red cross and the mesh vertex (1) on the bottom right. The red path is the shortest path on the mesh calculated by Floyd-Warshall algorithm and the blue line is the euclidean distance between the selected control point and the closest vertex (2) to this control point on the mesh.

where:
\[ \text{gd}(i, k) = \text{geodesic distance between the mesh vertex } v_i \text{ and } k^{th} \text{ control point } C_k \]

\[ S = \text{the set of control point indices controlling the vertex } v_i \]

Floyd-Warshall algorithm has time complexity \( O(n^3) \), so it takes quite a long time on models with higher number of vertices. To optimize that time, we can use a downsampled mesh for this computation. For the downsampling, we
can use previously mentioned QEM simplification method [GH97]. The down-
sampled mesh preserves the mesh branching, tunnels and important vertices
well. A cow mesh downsampled by QEM method can be seen in Figure 6.1.

3.4.2 Joint matrices

Bind pose matrices and current transformation matrices for all the nodes are
stored in the global (root local) space. They are not transformed into node’s
local space. The disadvantage is that during the skinning preview the matrices
have to be transformed into node’s local space. On the other hand, the main
advantage is that the skeleton structure is compatible with the majority of
animation software.

3.5 Collada 1.5 support

Model data and all important skinning data is saved in a compatible way into
the Collada .dae XML file. Collada is a Collaborative Design Activity for es-
ablishing an interchange file format for interactive 3D applications. Collada
supports storing of the mesh geometry, the hierarchical skeleton structure, in-
dices, weights and inverse bind pose matrices. With the version 1.5 comes the
possibility to store kinematics as well. To use the full support of a kinematics
model in Collada 1.5, computation of approximation of the minimum, the max-
imum and the current angle for each joint is needed. Angles can be computed
using the inverse kinematics approach or assigned manually by the user. In our
implementation we use the latest version of Collada DOM 2.2 with Collada 1.5
[Son08] support.
Chapter 4

Implementation

4.1 Three layer model

The whole application is composed of three independent layers:

1. Interface
2. Engine
3. Core

4.1.1 Interface

The interface is a form based GUI with components from Visual Studio 2008 .NET framework. The OpenGL context is attached on the panel handler. The interface visualizes the OpenGL scene to the user and processes all inputs from the user by interpreting them to the engine layer.

4.1.2 Engine

The engine is a middle layer, arranging visualization and all data transfer between interface components and core data structures and algorithms. The en-
CHAPTER 4. IMPLEMENTATION

engine controls a viewport camera, all the OpenGL function calls and shaders through GLEW extension. This layer also provides the I/O functionality as importing and exporting of models and computed skinning data.

4.1.3 Core

The core stores all data structures and provides main geometric methods for the mesh contraction and the skeleton extraction. The core can be used as an independent library, when no visualization and I/O functionality is needed. Each call of a core method entails creation of a new thread, which wraps and handles the process. After finalizing the calculation, the thread passes the results to the main engine thread and terminates.

4.2 Language and libraries

As a programming language for the implementation we have chosen C++, because of its performance and the wide variety of open source libraries. These open source libraries are crucial parts of all three layers and the majority of them were not available for any other language or development environment. That is one of the main reasons, why we decided to use Visual Studio 2008 as the development environment. Libraries used in all three layers:

1. Boost (serialization of objects, hash maps)
   http://www.boost.org/

2. Collada DOM (to store and create a Collada XML file)
   http://sourceforge.net/projects/collada-dom/

3. Corona (creating of screen capture images)
   http://corona.sourceforge.net/
CHAPTER 4. IMPLEMENTATION

4. FreeType font (writing text into the scene)
   http://www.freetype.org/

5. GLEW (extension for shader support)
   http://glew.sourceforge.net/

6. Jama (QR equation solver)
   http://math.nist.gov/tnt/jama_doxygen/

7. OpenGL (scene visualization)
   http://www.opengl.org/

8. TNT (storing of matrix structures)
   http://math.nist.gov/tnt/

4.3 Application work flow

The application work flow starts with importing a model. The user can either import model only, or model containing the skeleton or model containing all the skinning data. In case that user imports only the model, contraction and skeleton extraction have to follow in the next step. In other cases, when the user imports also the skeleton or the skeleton with the skinning indices and weights, these steps can be skipped. User can continue with computation of the skinning data, skeleton testing and modification. In the end the application provides exporting of the model with the computed skinning data into Collada 1.5 file. Work flow supported by our application can be seen in Figure 4.1.

4.4 Graph conversion

An input model structure, either structure from .3ds binary file or Collada .dae XML file, is transformed into the mesh graph structure in a straight-forward
Figure 4.1: Work flow supported by our application.
way. The edge set matrix is derived from the model edge connectivity. The mesh graph contains a hash map of indices, mapping indices from the converted mesh graph to the visualized input mesh.

4.4.1 Visualization of vertex joining

When either the joining distance is manually set or the minimal joining distance is calculated, vertex joinings between the components can be visualized. This feature allows the user to see how the components are going to be connected to each other - Figure 4.2. If the user is not satisfied with visualized joinings, he can change the joining distance and the visualized joinings are immediately updated according to the new value.
4.5 Mesh contraction

In each iteration the mesh graph vertices are contracted by Laplacian smoothing. To compute new vertex position, the linear equation (3.2) is solved in least-squares sense by QR decomposition. The mesh graph is stored in four instances. The original version of mesh graph after the conversion was made is stored in the core. Another instance is stored in the engine to visualize the current mesh graph contraction. Additionally, in each iteration step we work with two copies of mesh graph in the core. The first is a mesh graph, which comes to contraction process as input mesh and the second is computed and forms the output of the contraction. After the linear equation is solved, new vertex positions and updated contraction weights are set to the output mesh graph. When current iteration step is completed, the input mesh graph is released and the output mesh graph is copied into the engine to be visualized there during the next iteration step. Some examples of visualized contraction steps can be seen in Figure 4.3 and Figure 4.4.

4.6 Skeleton construction

In our implementation the user can choose how many control points he wants. We used 24 control points by default which worked well for almost all GPUs and also it is a sufficient number to control the skinning process of complex high resolution models. A greedy algorithm is applied to select edges with minimum cost, where the half-edge collapse should be applied. By applying the half-edge collapse on a pair of vertices, the edge set matrix $E$ is modified and a mapping of indices is updated. To keep the mapping of indices, two hash maps are used. The first hash map stores collapsed indices. The key is an index of a vertex in the mesh graph before skeleton construction phase and the value for each key is an index of a vertex, into which the original vertex was collapsed. The second
Figure 4.3: Examples of the mesh contraction of a pig model. (Top-left) The converted mesh graph. (Top-right) The contraction after 1 iteration. (Bottom-left) The contraction after 2 iterations. (Bottom-right) The contraction after 3 iterations, the volume is close to zero.
Figure 4.4: Examples of the mesh contraction of a hand model. (Top-left) The converted mesh graph. (Top-right) The contraction after 1 iteration. (Bottom-left) The contraction after 2 iterations. (Bottom-right) The contraction after 3 iterations, the volume is close to zero.
hash map stores the inverse mapping. The key is index of vertex, which still have not been collapsed. For each such an index the value is a list of indices, which correspond to vertices collapsed into vertex with the key index. Such a mapping is useful for computation of an approximative volume corresponding to each skeleton branch or visualization of these corresponding regions. By selecting the skeleton node, all the corresponding vertices, which were collapsed into this skeleton node can be selected.

4.7 Collada DOM

In the <library_geometries> node, there is stored source array with geometry vertices, source array with geometry normals and the <triangles> node, which stores indices to these two source arrays. In the <library_controllers> all the skinning data is stored. The first child node is the skeleton root inverse bind pose, which is in our implementation always filled with identity matrix. Next child nodes are three source arrays. The first is array with joint names, the second is array with the inverse bind pose matrices for each bone and the third contains computed skinning weights. The last child node is a node called <vertex_weights>, which stores computed indices and indices to weight source array. The computed indices define which bone influences which mesh vertices. The whole skeleton structure is hierarchically stored in the <library_visual_scenes> node. Each skeleton bone node contains joint type, local transformation matrix, child nodes and string identifier, which maps this node to previously stored skinning data. A simple example of exported Collada 1.5 XML can be seen here (only the main structure was preserved and irrelevant data as amount of objects and counters was replaced by ...).

An example of exported Collada 1.5 XML

```xml
<?xml version='1.0' encoding='UTF-8'?>
```
<COLLADA xmlns="http://www.collada.org/2008/03/COLLADASchema" version="1.5.0">
  <library_geometries>
    <geometry id="geometry_0">
      <mesh>
        <float_array id="geometry_0-positions-array" count="..." digits="...">
          ...
        </float_array>
        <technique_common>
          <accessor count="..." source="#geometry_0-positions-array" stride="3">
            <param name="Z" type="float"/>
            <param name="Y" type="float"/>
            <param name="X" type="float"/>
          </accessor>
        </technique_common>
        <source id="#geometry_0-positions" name="position"/>
      </mesh>

      <float_array id="geometry_0-normals-array" count="..." digits="...">
        ...
      </float_array>
      <technique_common>
        <accessor count="..." source="#geometry_0-normals-array" stride="3">
          <param name="Z" type="float"/>
          <param name="Y" type="float"/>
          <param name="X" type="float"/>
        </accessor>
      </technique_common>
      <source id="#geometry_0-normals" name="normal"/>
    </geometry>

    <float_array id="geometry_0-uv-array" count="..." digits="...">
      ...
    </float_array>
    <technique_common>
      <accessor count="..." source="#geometry_0-uv-array" stride="2">
        <param name="s" type="float"/>
        <param name="t" type="float"/>
      </accessor>
    </technique_common>
    <source id="#geometry_0-uv" name="map"/>
  </library_geometries>

  <vertices id="geometry_0-vertices">
    <input semantic="POSITION" source="#geometry_0-positions"/>
  </vertices>

  <triangles count="...">
    <input offset="0" semantic="VERTICES" source="#geometry_0-vertices" set="0"/>
    <input offset="1" semantic="NORMALS" source="#geometry_0-normals" set="0"/>
    <input offset="2" semantic="TANGENT" source="#geometry_0-uv" set="0"/>
    <p>...</p>
  </triangles>
</COLLADA>
<technique_common>
  <accessor count='...' source='#geometry_0-skin-joints-array'>
    <param name='JOINT' type='Name'/>  
  </accessor>
  <source id='#geometry_0-skin-bounds'>
    <float_array id='geometry_0-skin-binds-poses-array' count='...' digits='...'/>
    ...</float_array>
  </source>
  <accessor count='...' source='#geometry_0-skin-binds-poses-array' stride='64'>
    <param name='TRANSFORM' type='float4x4'/>
    ...</accessor>
  <source id='#geometry_0-skin-weights'>
    <float_array id='geometry_0-skin-weights-array' count='...' digits='...'/>
    ...</float_array>
  </source>
  <accessor count='...' source='#geometry_0-skin-weights-array'>
    <param name='WEIGHT' type='float'/>
    ...</accessor>
  <source>
    <joints>
      <input semantic='JOINT' source='#geometry_0-skin-joints'/>
      <input semantic='INV_BIND_MATRIX' source='#geometry_0-skin-binds-poses'/>
      <joints>
      <vertex_weights count='...'/>
      <input offset='0' semantic='JOINT' source='#geometry_0-skin-joints'/>
      <input offset='1' semantic='WEIGHT' source='#geometry_0-skin-weights'/>
      <vcount>.../</vcount>
      ...</vcount>
      ...</joints>
      <vertex_weights>
      <skin>
      </controller>
    </library_controllers>
    <library_visual_scenes>
    <visual_scene id='scene'>
      <node id='node_0'>
        <instance_controller url='#geometry_0-skin'><skeleton id='skeleton_root'/></instance_controller>
        ...</node>
      <node id='skeleton_root'>
        <joint0 sid='joint0' type='JOINT'>
        <translate>.../</translate>
        <joint1 sid='joint1' type='JOINT'>
        <translate>.../</translate>
        <joint1 sid='joint1' type='JOINT'>
        <translate>.../</translate>
        ...</joint1>
      </node>
    </visual_scene>
    </library_visual_scenes>
  <scene>
    <instance_visual_scene url='#scene'/>
  </scene>
</technique_common>
4.8 Skinning on GPU

Our framework provides linear blend skinning implemented on GPU for real-time examination of computed skeletons. It is the most suitable method how to inspect the skeleton structure and data, because of its efficiency and simple GPU implementation. After the skeleton is computed, the "bind pose" world-space snapshot of all transformation matrices of the skeleton nodes is taken, denoted as $B_i$, for each skeleton node. During the real-time deformation process, transformation matrices are computed. Each transformation matrix, storing the current affine transformation, denoted as $P_i$, is computed each time the user manually changes skeleton nodes. In each frame, the resulting transformation $M_i$ is computed as $M_i = P_iB_i^{-1}$ on CPU and uploaded into GPU. New deformed vertices are computed using GLSL shader as:

$$v' = \sum_{i=0}^{n} w_i M_i v_i$$  \hspace{1cm} (4.1)

where:

- $M_i$ - is the resulting transformation matrix computed on the CPU as $M_i = P_iB_i^{-1}$
- $w_i$ - is the associated weight
- $v_i$ - is the original vertex position in the $M_i$ coordinates system

```glsl
uniform mat4 boneMatrices[24];
attribute float numBones;
attribute vec4 index;
attribute vec4 weight;

void main(){
    /* skinning vertex shader */
}"
```
CHAPTER 4. IMPLEMENTATION

```c
...  
// initialize new vectors*/
vec3 transformedNormal = vec3(0.0);
vec4 transformedPosition = vec4(0.0);

// set attribute iterators*/
vec4 indexIterator = index;
vec4 weightIterator = weight;

for (int i = 0; i < int(numBones); i++){
    mat4 m44 = boneMatrices[int(indexIterator.x)];
    float w = weightIterator.x;

    /* transform the offset by bone i*/
    transformedPosition += m44 * gl_Vertex * w;

    mat3 m33 = mat3(m44[0],xyz, m44[1],xyz, m44[2],xyz);

    /* transform normal by bone i*/
    transformedNormal += m33 * gl_Normal * w;

    /* iterate weights and indices*/
    indexIterator = indexIterator.yzw;
    weightIterator = weightIterator.yzw;
}

gl_Position = gl_ModelViewProjectionMatrix * transformedPosition;
normal = transformedNormal;
...
Chapter 5

Results

We have tested the framework on a wide range of different models. We have achieved good results with both high resolution and also with low resolution models. The more vertices the model has, the better skeleton can be computed, but the algorithm takes more time. Also the low resolution models (less than 5000 polygons) can be contracted in a good way, but it is harder to set good contraction weights. Setting the right contraction weights is the most problematic part of this approach. Low resolution models are more sensitive for high curvature differences and can be easily over contracted. Weights often have to be set manually and the user needs some experience. Contraction of high resolution models is more deterministic and good weights can be set automatically. Table 5.1 and Table 5.2 show how the contraction weights influence the contracted geometry and the number of iterations needed. In the Table 5.1 we set the contraction weight as an input constant and in the Table 5.2 the contraction weight is multiplied by a dependency term discussed in the next section.

The contraction of geometry with lower number of vertices can be seen in Figure 5.1 and contraction of geometry with more vertices can be seen in Figure 5.2 and 5.3. The model of a worm in Figure 5.2 was published with a manually rigged skeleton (Bottom-right image). After the comparison we can conclude
Figure 5.1: The contraction and the extraction of the skeleton from low resolution geometry. Left to right are the converted mesh graph, the mesh graph after 2 iterations, the mesh graph after the last iteration - the volume approximation is close to zero, the extracted skeleton.

that the extracted skeleton is very close to the manually rigged one. It can be used as a sufficient supplicant for a manually created and rigged skeleton. The major difference can be observed on the both ends of the worm and in the largest bend. Nodes of the skeleton tree were pushed into their centers of local mesh areas and that is why they were pushed inside, away from the mesh boundary. The functionality to merge close vertices before applying the greedy algorithm can be a powerful tool to improve some skeleton properties, such as control point distribution, skeleton regularity, symmetry and centeredness. On the other hand, it can cause undesirable features such as losing control points in very small parts of the mesh, for example ears of the panda in Figure 5.3. Because of the merging of vertices in a predefined distance radius, the vertices which were the ears composed of were merged into the head. That is why this feature has to be used wisely. The user has to gain some experience, when to use it and when not to.
Figure 5.2: The contraction and the extraction of the skeleton in few iterations from higher resolution geometry and its comparison to manually rigged skeleton by an artist. (Top-left) An input model. (Top-middle) The converted mesh graph. (Top-right) The mesh graph after 2 iterations. (Bottom-left) The mesh graph after the last iteration. (Bottom-middle) The extracted skeleton. (Bottom-right) The skeleton rigged by an artist.
Figure 5.3: Another example of higher resolution geometry. The extracted skeleton have sparser nodes at the core parts. This feature can be observed, because many faces at core parts are contracted into the same region. The points are also moved towards the mesh boundary of the limbs, because of the shifting of the control points to the center of its local mesh.
5.1 Dependency on resolution and dimensions

As we have previously mentioned, each term of Laplacian coordinates \( \delta_i = -4A_i k_i n_i \) is proportional to the area of adjacent faces surrounding the vertex \( i \), denoted as \( A_i \). If we want to contract all the types of models with the same weight system, the solution has to satisfy two conditions. The first condition is that the contraction of two models can not be dependent on their dimensions. The Laplacian operator is suitable for this case, because if the coordinates are proportional to \( A_i \) then the model vertices are contracted more if the dimension is higher and contracted less if the dimension is lower. This results in a property that the contraction speed is not dependent on dimensions of the model. The second condition is that the contraction can not be dependent on the resolution of the model. Models representing the same object have to be contracted equally, regardless of the resolution (if they are downsampled or not). This condition is not satisfied by the operator, because the contraction forces from the Laplacian operator are smaller for denser models. This can be observed, because the area of adjacent faces surrounding each vertex is smaller for denser models, but the dimensions stay the same. The solution how to update the contraction weight proposed by [ATC+08] satisfies the second condition, but violates the first condition. To satisfy both conditions we are looking for such a term \( t \) that the resulting Laplacian coordinates \( W_t tLV \) will be approximately the same for denser and sparser models while the first condition will not be broken. Such a term has to be a function indirectly proportional to the average face area of the model, denoted as \( A \). With this term we can fulfill the second condition, but we will break the first one, because for models with small dimensions this term will be high and they will be over-contracted. We can keep the first condition by making the term directly proportional to the sum of face areas over the model, denoted as \( S \). Thus, the term should be approximately \( t \approx \frac{S}{A} \), this results to \( t \approx \#vertices \). We initially set \( W_{L,t} = W_{L,t} \), so the
resulting Laplacian coordinates $W_{L_i} L V$ will not depend on the resolution of
the model and length of the edges. We have found the term $t = \#\text{vertices}$ as a
good value to modify the input contraction weight. Results showing the usage
of our derived coordinates can be seen in Table 5.2. An example of spheres with
the same radius, but different resolution can be seen in Table 5.3. Before the
correction term $t$ was applied, different weights have to be used for contraction
of each model. After we have included this term, the same input weight could
be used to contract spheres in the same number of iterations.

5.2 Robustness

The mesh contraction and the skeleton extraction phases are pose independent.
It enables extraction of compatible skeletons from different poses of the same
model. By compatible we mean compatible in the sense of the same branching,
tunnels and the homotopy with an input mesh. Also, the length of preserved
edges will be the same, because edges are collapsed in the same order. In the
end of the skeleton construction process, all corresponding skeleton bones are
going to have the same lengths. The geometry meshes in different poses have
corresponding edges of the same length as well, so the skinning weights and
indices will result into identical values.

5.3 Computation time

Table 5.4 shows how much time each stage of skeleton construction and skin-
ning data computation requires. The time was measured on an Intel Core 2 Duo
2.20GHz with 2.5GB RAM, using a single thread implementation for the calcula-
tions. As we can see, the most time consuming parts of the implementation are
solving the linear system and the Floyd-Warshall algorithm. We have already
proposed how the computational time needed for Floyd-Warshall algorithm can
be improved - by downsampling the mesh graph. The time needed for the linear solver can be improved either by using downsampled mesh or by solving the linear system by a multigrid solver. The usage of downsampled mesh graph can corrupt the resulting skeleton if it is downsampled too much. Better results can be achieved using the multigrid solver or both approaches. The multigrid solver implementation [ATC+08] can optimize the time complexity from $O(n^3)$ to $O(n)$. In our case, such an optimization changes the computation time to several seconds instead of thousands of seconds for high resolution models. We tried to show wide variety of models with different number of vertices and different topological properties. Here we can discuss the values in Table 5.4 more. The computation times for the worm model took less time than should be expected according to the number of vertices. The reason is that we did not want to animate eyes, so we excluded these vertices from the mesh graph. Eyes are composed of the majority of vertices (>90%), so the converted mesh graph had less than 10% of the vertices. That is why times $t_2$ and $t_4$ are smaller than expected. On the other hand, $t_1$ took much more time than should be expected, because of the excluding of eye vertices and the mesh graph recreation. We do not present higher resolution models than 14952 vertices (panda), because the $O(n^3)$ complexity causes the time rise rapidly.

5.4 Further applications

5.4.1 Detection of symmetry axis from the skeleton

In some cases it can be assumed that the morphology of an input model is symmetric. It can have two or four legs, two ears, a head and sometimes a tail as well. These parts are centered with respect to the spine. Thus, the computed skeleton should be symmetric with respect to a bone or a segment composed of more bones. Finding symmetries on a graph is NP-complete problem. Consid-
CHAPTER 5. RESULTS

erating some assumptions about the graph an efficient algorithm can be found. The computed skeleton is actually a tree, it does not contain any cycle. The skeleton root is located on the symmetry axis, it should represent the head or bones of the spine. Two subtrees of the skeleton tree are isomorphic if they have the same depth and their root nodes have the same degree. With these assumptions an efficient algorithm based on iterative marking of isomorphic subtrees can be applied [AHLD07].

5.4.2 Automatic segmentation of model into primitives

The skeleton-mesh mapping can be used for a mesh segmentation. Each branch of the skeleton corresponds to a logical component of the model and therefore our branching structure can serve as a useful guide for segmenting the mesh. Firstly, branches have to be ordered according to their approximative volume (average distance between the skeleton node and mapped vertices in the local mesh area). The mesh is iteratively cut starting from the branch with thickest region, until we have the desired number of segments. If we need a higher number of segments than the number of branches, regular nodes (within the branch) can be used. The 1D Laplacian of their local thickness is computed from their adjacent nodes and a node with the smallest 1D Laplacian is selected as the next cutting node [ATC+08].
### Table 5.1

<table>
<thead>
<tr>
<th>Model</th>
<th>Avg. face area</th>
<th>$W_L$</th>
<th>$W_H$</th>
<th>$S_L$</th>
<th>#iterations</th>
<th>Skeleton quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder</td>
<td>24.97720</td>
<td>1.05</td>
<td>0.45</td>
<td>0.19</td>
<td>2</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>good</td>
</tr>
<tr>
<td>Low-res char</td>
<td>17.4988</td>
<td>1.6</td>
<td>0.58</td>
<td>0.1</td>
<td>2</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>bad</td>
</tr>
<tr>
<td>BE fighter</td>
<td>109.8192</td>
<td>6.0</td>
<td>2.8</td>
<td>0.9</td>
<td>2</td>
<td>bad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>bad</td>
</tr>
<tr>
<td>Hand low</td>
<td>0.616914</td>
<td>12.0</td>
<td>5.5</td>
<td>2.2</td>
<td>2</td>
<td>average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>average</td>
</tr>
<tr>
<td>Guyver</td>
<td>0.001348</td>
<td>13.0</td>
<td>6.0</td>
<td>1.9</td>
<td>2</td>
<td>average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>average</td>
</tr>
<tr>
<td>Hand high</td>
<td>0.616914</td>
<td>16.5</td>
<td>7.5</td>
<td>2.9</td>
<td>2</td>
<td>average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>good</td>
</tr>
<tr>
<td>Pig</td>
<td>0.008773</td>
<td>18.0</td>
<td>8.1</td>
<td>2.3</td>
<td>2</td>
<td>average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>good</td>
</tr>
<tr>
<td>Panda</td>
<td>69.65585</td>
<td>26.0</td>
<td>12.5</td>
<td>4.9</td>
<td>2</td>
<td>bad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>good</td>
</tr>
<tr>
<td>Worm</td>
<td>9.258953</td>
<td>3.2</td>
<td>1.2</td>
<td>0.25</td>
<td>2</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
<td>good</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>4</td>
<td>good</td>
</tr>
</tbody>
</table>

Table 5.1: Table shows how many iterations were needed to collapse the mesh geometry into almost a zero volume depending on the average face area and contraction weights. The term $S_L$, which the collapsing weight $W_L$ is multiplied by after each iteration, was always set to 2.0, which was determined as a suitable constant for increasing of the collapsing speed.
CHAPTER 5. RESULTS

<table>
<thead>
<tr>
<th>Model</th>
<th>Avg. face area</th>
<th>#vertices</th>
<th>$W_L$</th>
<th>$W_H$</th>
<th>$S_L$</th>
<th>#iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder</td>
<td>24.97720</td>
<td>648</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Low-res char</td>
<td>17.49488</td>
<td>720</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>BE fighter</td>
<td>109.8192</td>
<td>3096</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Hand low</td>
<td>0.001348</td>
<td>3660</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Guyver</td>
<td>0.001348</td>
<td>5096</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Hand high</td>
<td>0.616914</td>
<td>8256</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Pig</td>
<td>0.008773</td>
<td>9624</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Panda</td>
<td>69.65585</td>
<td>14952</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Worm</td>
<td>9.258953</td>
<td>19848</td>
<td>0.002 $\sim$ 0.003</td>
<td>1.0</td>
<td>2.0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.2: Table shows how many iterations were needed to collapse the mesh geometry into almost a zero volume depending on the average face area and contraction weights. Initially we set $W_{L,i} = W_{L,i} \#vertices$. We apply this initial setting to avoid dependency on the model resolution and length of the edges.
CHAPTER 5. RESULTS

<table>
<thead>
<tr>
<th>Model</th>
<th>Avg. face area</th>
<th>#vertices</th>
<th>(W_L)</th>
<th>(W_H)</th>
<th>(S_L)</th>
<th>#iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere low-res</td>
<td>125.9</td>
<td>540</td>
<td>1.3</td>
<td>0.7</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Sphere mid-res</td>
<td>24.3</td>
<td>2880</td>
<td>8.0</td>
<td>3.8</td>
<td>1.0</td>
<td>4</td>
</tr>
<tr>
<td>Sphere high-res</td>
<td>3.7</td>
<td>18720</td>
<td>50.0</td>
<td>22.0</td>
<td>1.0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.3: Table shows how many iterations were needed to collapse the mesh of 3 spheres with same radius, but different resolution, before the correction term \(t = \#\text{vertices}\) was applied.

<table>
<thead>
<tr>
<th>Model</th>
<th>#vertices</th>
<th>(t_1)</th>
<th>(t_2)</th>
<th>(t_3)</th>
<th>(t_4)</th>
<th>(t_5)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder</td>
<td>648</td>
<td>&lt;0.1</td>
<td>0.5</td>
<td>0.4</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>1.2</td>
</tr>
<tr>
<td>Low-res char.</td>
<td>720</td>
<td>&lt;0.1</td>
<td>0.3</td>
<td>0.9</td>
<td>&lt;0.1</td>
<td>0.3</td>
<td>1.7</td>
</tr>
<tr>
<td>BE fighter</td>
<td>3096</td>
<td>0.2</td>
<td>39.8</td>
<td>0.9</td>
<td>8.7</td>
<td>3.1</td>
<td>52.7</td>
</tr>
<tr>
<td>Hand low</td>
<td>3660</td>
<td>0.3</td>
<td>58.8</td>
<td>0.5</td>
<td>11.1</td>
<td>3.1</td>
<td>73.8</td>
</tr>
<tr>
<td>Guyver</td>
<td>5094</td>
<td>0.9</td>
<td>230.1</td>
<td>1.8</td>
<td>38.9</td>
<td>6.8</td>
<td>278.5</td>
</tr>
<tr>
<td>Hand high</td>
<td>8256</td>
<td>1.9</td>
<td>968.7</td>
<td>10.2</td>
<td>228.5</td>
<td>26.9</td>
<td>1236.2</td>
</tr>
<tr>
<td>Pig</td>
<td>9624</td>
<td>2.1</td>
<td>1372.1</td>
<td>6.7</td>
<td>331.4</td>
<td>7.3</td>
<td>1719.6</td>
</tr>
<tr>
<td>Panda</td>
<td>14952</td>
<td>6.1</td>
<td>6590.3</td>
<td>7.4</td>
<td>1189.7</td>
<td>75.1</td>
<td>7868.6</td>
</tr>
<tr>
<td>Worm</td>
<td>19848</td>
<td>12.2</td>
<td>11.6</td>
<td>0.5</td>
<td>2.9</td>
<td>11.0</td>
<td>38.2</td>
</tr>
</tbody>
</table>

Table 5.4: Columns \(t_1\), \(t_2\), \(t_3\), \(t_4\) and \(t_5\) show the running time (in seconds) of the mesh graph conversion, mesh contraction, skeleton construction, Floyd-Warshall algorithm and weight computation, respectively.
Chapter 6

Conclusion

In this Master’s Thesis we propose a framework for extracting a skeleton and skinning data from the geometry mesh using an iterative mesh contraction. The approach begins with converting the geometry into a mesh graph. This graph is iteratively contracted and a greedy algorithm is applied to choose the most important subset of vertices. In the next step, these vertices are converted into a hierarchical skeleton. Important skinning data such as indices and weights which are controlling the influence of the skinning process over vertices are computed as well. When the data is computed, our framework also provides a way to inspect the skeleton, simulate the skinning deformation process with full GPU support and allow the user to change the skeleton branching if it is needed. After all, geometry and skinning data can be exported into Collada 1.5 .dae file and transferred into an external application to create animations.

The extracted skeletons have sparse nodes at the core parts of the model. This feature can be observed, because many faces at core parts are contracted into the same region. Computed skeletons are independent of the size and resolution of the models. The approach is insensitive to noise, but works only for closed mesh models with 2D manifold connectivity.
6.1 Future work

6.1.1 Simplification of the mesh in the preprocessing stage

It is worth pointing out the idea that downsampled mesh could be used for the whole mesh contraction and skinning data extraction process. After an input model is converted into the mesh graph, the previously mentioned QEM simplification method [GH97] can be applied. The downsampled mesh preserves the mesh topology well if it is not downsampled too much. A coarser mesh downsampled by QEM method can be seen in Figure 6.1. Such a simplification could cut down the computation time greatly if we are dealing with high resolution models. A compromise should be found between the time cut down and the quality of resulting skeleton, because if the resolution of the mesh drops too much, we lose good behaviour of contraction weights during the Laplacian smoothing process.

6.1.2 Inverse kinematics for the skinning deformation

Inverse Kinematics (IK) techniques enable real-time and off-line control of articulated figures. They provide a calculation of transformation matrices in a skeleton subtree according to available constraints. These constraints can be derived from the location of a subtree node, transformation in the parent node or some other transformation thresholds. New transformation matrices are computed according to the constraints without changing the skinning weights and the skeleton tree branching. By moving a bone in the skeleton structure, new transformation matrices are calculated and immediately applied in skinning deformations. Performing IK during the skinning preview, a brand new way how to inspect the skeleton branching and the skinning data will be offered to the user.
Figure 6.1: An example of application of QEM method from [GH97]. The original model on the left has 5,804 faces. The approximations to the right have 994, 532, 248, and 64 faces, respectively. Note that features such as horns and hooves continue to exist through many simplifications. Only at extremely low levels of detail do they begin to disappear.
Bibliography


BIBLIOGRAPHY


