University of West Bohemia

Faculty of Applied Sciences

Department of Computer Science and Engineering

Diploma Thesis

Evaluation of Dynamic Mesh Compression Algorithms Using Alternative Metrics

Pilsen, 2010

Oldřich Petřík

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Pilsen, 20/5/2010 Oldřich Petřík

Abstract

Evaluation of Dynamic Mesh Compression Algorithms Using Alternative Metrics

In the last decade, with the growing complexity of 3D data, it became necessary to use compression techniques to reduce the storage space needed for such amounts of data. Many algorithms were developped, most of them based on a lossy compression scheme. It became apparent, that the distortion introduced by discarding a part of the data needs to be measured in order to compare different settings and different algorithms. Several distortion metrics were proposed based on various measurement methods.

In this thesis, four state-of-the-art dynamic mesh compression algorithms will be discussed: Dynapack, D3DMC, FAMC and Coddyac. D3DMC and FAMC will also be implemented, while the implementation of the Dynapack and Coddyac algorithms we have already available. The principle and the input parameters of each of the algorithms will be described in detail and the influence of different input parameters on the compression result will be discussed. Two distortion measures will be used: KG and STED. Their performance will be evaluated and compared.

At last, the matter of rate-distortion optimisation will be thoroughly discussed. In result, a method for finding the optimal parameter values of a general compression algorithm for a given dataset will be proposed. Its properties will be evaluated and compared to the current state of the art.

Abstrakt

Evaluace algoritmů pro kompresi dynamických sítí alternativními metrikami

S rostoucí složitostí trojrozměrných dat v posledním desetiletí vznikla potřeba použití kompresních technik, aby bylo možné zmenšit úložný prostor potřebný k uložení těchto datových sad. Bylo vyvinuto mnoho algoritmů, z nichž většina pracuje se ztrátovou kompresí. Použitím takových algoritmů se část dat ztrácí, čímž vzniká určitá chyba. Pro porovnání různých nastavení algoritmu či porovnání různých algoritmů mezi sebou je třeba tuto chybu změřit. Za tím účelem bylo navrženo několik metrik založených na různých principech.

V této práci budou představeny čtyři algoritmy pro kompresi dynamických sítí: Dynapack, D3DMC, FAMC a Coddyac. D3DMC a FAMC budou také implementovány, zatímco implementaci algoritmů Dynapack a Coddyac již máme k dispozici. Bude podrobně vysvětlen princip fungování každého z algoritmů a také jeho vstupní parametry, včetně diskuse jejich vlivu na výsledky komprese. Při tom bude využito dvou metrik pro měření vzniklé chyby: KG a STED. Jejich chování bude následně vyhodnoceno a porovnáno.

Na závěr bude zevrubně diskutováno téma optimalizace datového toku vzhledem k chybě způsobené kompresí. Jako výsledek bude navržena metoda pro nalezení optimálních hodnot parametrů. Její vlastnosti budou analyzovány a porovnány s běžně používanými metodami.

Contents

1

Introduction

Computer graphics has gained much popularity in the recent decades and became a crucial part of many disciplines such as medicine, entertainment or advertising. The processing power of graphics hardware has seen a fast advancement over the recent years making it vastly more powerful than conventional processors. This advancement has made computer graphics capable of solving problems that have been unreachable before.

With the growing hardware performance, a demand for visual quality arises causing still better methods for data acquisition to be developed. While these two branches of computer graphics may be able to keep up with each other, the storage space and data transfer speeds represent significant limitations for the size of the datasets, which, especially for animated cases, may be simply too large. Employing compression is a logical step to reduce or eliminate such limitations.

1.1 Problem Definition

In this work, we concentrate on compression of dynamic triangle meshes with constant connectivity ("dynamic meshes", "animated meshes"). Such data may be seen as a succession of triangle meshes, where each mesh represents a single frame of a uniformly sampled animation. All the meshes then share the same number of vertices and the same connectivity (i.e. each corresponding vertex has the same neighbours in each of these meshes), while the positions of corresponding vertices are time-variable. There are various other ways to represent animated 3D scenes as meshes, such as dynamic triangle meshes with variable connectivity and dynamic quad meshes, or using different approaches, for example skinning [9, 10]. However, this representation is much simpler, while still being suitable for most problems at hand, and can be

derived from the other representations.

The continuing research in the area of dynamic triangle mesh compression has resulted in many compression methods with a wide range of different approaches. To reach the highest compression ratios, most of these methods use a lossy compression scheme, where part of the geometry data containing the least important information (according to the respective algorithm design) is discarded. This introduces a distortion into the animation. In other words, the vertex positions of the compressed animation differ from the original ones. To measure this difference, several distortion metrics were developed based on different measurement schemes. We will use the *KG error* metric proposed by Karni and Gotsman [8], which is the most common distortion measure used in dynamic mesh compression, and the *STED* metric by Váša and Skala [27], which correlates well with human perception of distortion. Together with distortion, we will measure the bitrate ("rate") of the compressed animation, which is expressed in bits per frame and vertex (*bpfv*). This value is equivalent to the inversed compression ratio, as it is constant for uncompressed dynamic meshes.

The compression methods for dynamic meshes use various segmentation and prediction models to reduce redundance present in the data. In most cases, a quantisation process is employed to discretise the data and prepare it for entropy coding. An entropy coder is also often integrated to further decrease the entropy. Additional techniques may be involved adding or improving some properties of the algorithm or adapting it for a specific application.

Most of the above-mentioned compression techniques need to be properly configured in order to deliver the best performance. This is usually done by introducing configuration parameters to the compression algorithm like the coarseness of the quantisation process, number of clusters, etc. Usually, as an algorithm evolves to offer higher compression ratios or improved properties, more techniques get involved in the compression process, thus increasing the number of configuration parameters. The number of parameters can range from one in very simple algorithms like Dynapack (section 2.1) to e.g. five in FAMC (section 2.2) or even more.

Our goal is to find a method that will, for a given compression algorithm, dynamic triangle mesh, and distortion metric, find such parameter values, which will result in an optimal compression result, i.e. the minimal distortion for that bitrate and the highest compression ratio for that distortion.

The problem of finding the optimal configuration for a general compressor, the function of which is unknown to the optimiser, is non-trivial. The commonly used approach is actually an exhaustive search, where a vector of values is specified for each parameter and the compressor is run for all possible configurations created using these values. The configurations producing

the best results are then picked by hand or automatically. The number of times the compressor has to be executed is exponential to the number of parameters with the argument equal to the size of the value vectors. While this approach may be sufficient for compression algorithms with one or two parameters, it may get unreasonably slow for higher parameter counts. We attempt to design an algorithm that will be able to find optimal configurations in a reasonable number of compressor runs.

1.2 Organisation

The rest of this work is divided into five parts. In the first part, four state-of-the-art algorithms for dynamic triangle mesh compression – Dynapack, FAMC, D3DMC, and Coddyac – are presented. The principles of these algorithms are thoroughly explained and their input parameters are described. Each algorithm is discussed in terms of compression performance and properties. In the next part, the problem of rate-distortion optimisation in dynamic mesh compression is discussed. A general optimisation method is proposed and its design is explained in great detail. This work also includes an implementation of the compression algorithms and the proposed optimisation method. The implementation details are explained in the third part. Next, we present and discuss experimental results of our method and compare them with the current state of the art. Finally, a conclusion is drawn in the last part.

1.3 Notation

If not defined otherwise, the following notation will be used throughout the rest of the text:

F	number of frames in the mesh animation
f	a single frame of the animation
V	number of vertices in each mesh of the animation
v	a single vertex in a mesh
$\mathrm{pred}(\xi)$ ...	predicted value of ξ
$\hat{\xi}$	quantised value of ξ
$\tilde{\xi}$	decoded value of ξ

2

Compression Algorithms

2.1 Dynapack

Dynapack [4] is a simple and fast compression method for dynamic meshes based on a local spatiotemporal prediction model. The position of vertex v in frame f is predicted from the positions of three of its neighbours in frame f (spatial prediction) and from the positions of vertex v and the same three neighbours in the previous frame $f-1$ (temporal prediction). The algorithm introduces two extrapolating space-time predictors — *ELP* (Extended Lorenzo Predictor) and *Replica*. The ELP is very fast, but it only is a perfect predictor for rigid translations, while Replica can perfectly describe any combination of rotation, translation, and uniform scaling at the cost of higher computation complexity.

Mesh Traversal

The Dynapack algorithm utilises a mesh traversal approach similar to Rossignac's EdgeBreaker [16]. A Corner Table data structure [17] C is used to describe the topology of the mesh inside both the encoder and the decoder. This table holds $3T$ entries, where T is the number of triangles. The vertex indices of the three corners of a triangle t are stored at indices $3t$, $3t+1$, and $3t+2$ in the corner table. For each corner (index) c, we can determine the triangle $t(c)$ it belongs to by an integer division $t(c) = c \text{ DIV } 3$ and the associated vertex $v(c) = C_c$. Assuming the triangles are consistently oriented, the other two corners of $t(c)$ can be found: the next corner in the triangle ordering $n(c) = c-2$ if $c \text{ MOD } 3 = 2$, otherwise $n(c) = c+1$, and the previous corner $p(c) = n(n(c))$. An opposite corner table O contains for each corner c the opposite corner $o(c)$, i.e. the corner, for which applies $n(c) = p(o(c))$ and $p(c) = n(o(c))$. For corners, whose opposite edge is a border edge, the table contains -1. The corner $l(c)$ to the left of c can also be determined: $l(c) = o(n(c))$, and the one to the right: $r(c) = o(p(c))$. A boolean value $m(t)$ indicates, whether triangle t has already been visited during the traversal.

Similarly, $m(v)$ marks vertices, which have been visited. At the beginning, these indicators are initialised to *false*.

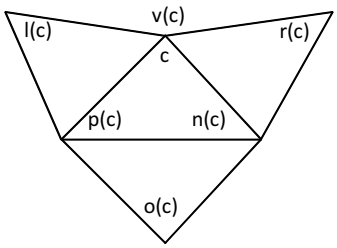

Figure 2.1: Corner operators in the Corner Table mesh representation.

In each frame, each connected component of the animated mesh is traversed by the algorithm in an exactly defined order, so that the process can be repeated in the decoder. First, we select a seed corner c, encode the vertices of its triangle $t(c)$ and mark the triangle and its vertices as visited. Then, a recursive depth-first traversal is performed by calling `dynapack(o(c))`, `dynapack(l(c))` and `dynapack(r(c))`. The recursive procedure is defined as follows:

```
procedure dynapack(c)
begin
    if c = -1 then exit;
    if not m(t(c)) then
    begin
        if not m(v(c)) then
        begin
            encode(position(c, f) - predict(c, f));
            m(v(c)) := true;
        end
        m(t(c)) := true;
        dynapack(r(c));
        dynapack(l(c));
    end
end
```

The `position(c, f)` procedure returns the original position of the vertex of corner c in the f-th frame, and `predict(c, f)` returns the predicted position of that vertex in frame f. Note that for the first frame, a space-only prediction has to be used. For the subsequent frames, the spatio-temporal predictors mentioned above are implemented. The vertices of the seed triangle also have to be encoded differently. In the first frame, these vertices are stored without prediction, while in the other frames, a time-only predictor can be employed using the seed triangles from the previous frames.

Decoding is carried out similarly to encoding. First, the seed triangle of the first frame is decoded. Then, the rest of the mesh of the first frame is traversed using a spatial prediction. The other frames are then decoded in the same way, except that a temporal predictor is used for the seed triangles and a spatio-temporal predictor throughout the mesh traversal. The decoding traversal is performed by issuing three calls: `dynaunpack(o(c))`, `dynaunpack(l(c))`, and `dynaunpack(r(c))`. The `dynaunpack()` procedure is defined as:

```
procedure dynaunpack(c)
begin
    if c = -1 then exit;
    if not m(t(c)) then
    begin
        if not m(v(c)) then
        begin
            position(c, f) := predict(c, f) + decode();
            m(v(c)) := true;
        end
        m(t(c)) := true;
        dynaunpack(r(c));
        dynaunpack(l(c));
    end
end
```

Predictors

- **Space-only Predictor**

  This predictor is used to encode the traversed vertices in the first frame of the animation. It utilises the parallelogram rule to predict the position of the vertex in the opposite corner from the vertices of the known triangle in the same frame. It is exactly the parallelogram predictor used by Touma and Gotsman [22] and by EdgeBreaker. The position of vertex $v(c)$ in frame f is predicted as:

$$v(n(c))_f + v(p(c))_f - v(o(c))_f \; . \tag{2.1}$$

- **Time-only Predictor**

  The prediction of the vertices of the seed triangles in all frames except the first is performed using this purely temporal predictor. It simply assumes the position of the vertex in frame f to be the same as in frame $f - 1$.

- **Extended Lorenzo Predictor**

  ELP is a generalisation of the Lorenzo Predictor [3] employed in compression of regularly

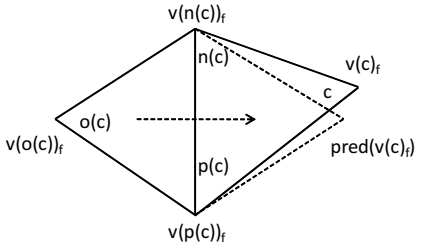

Figure 2.2: Scheme of the space-only predictor in Dynapack.

sampled four-dimensional scalar fields. It is a space-time predictor, which assumes that a spatial prediction error in the current frame is the same as it was in the preceding frame. In other words, it evaluates the prediction of vertex $v(c)$ in frame f as:

$$\underbrace{v(n(c))_f + v(p(c))_f - v(o(c))_f}_{\text{spatial prediction in frame } f} - \Big[\underbrace{v(n(c))_{f-1} + v(p(c))_{f-1} - v(o(c))_{f-1}}_{\text{spatial prediction in frame } f-1}\Big] + v(c)_{f-1}. \tag{2.2}$$

This predictor is the perfect predictor for vertices in regions of the mesh that have been transformed by a pure translation from the previous frame. I.e. if the four vertices of a parallelogram have moved by a constant vector, the spatial prediction error stays the same, thus no further information is needed to obtain the exact position of the predicted vertex.

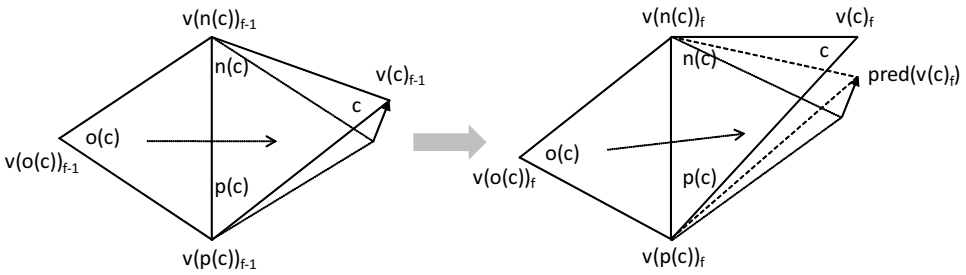

Figure 2.3: Scheme of the Extended Lorenzo Predictor.

- **Replica**

  It is a vector-based predictor. Instead of using a linear combination of known vertex positions, Replica expresses the position of the predicted vertex v(c) in a local coordinate system created from triangle t(o(c)). The local coordinates $[a, b, c]$ are extracted from the known position of the vertex in the preceding frame using the following equation:

$$v(c)_{f-1} = v(o(c))_{f-1} + aA_{f-1} + bB_{f-1} + cC_{f-1} \text{ , where} \tag{2.3}$$

$$\begin{aligned} A_i &= v(p(c))_i - v(o(c))_i \text{ ,} \\ B_i &= v(n(c))_i - v(o(c))_i \text{ ,} \\ C_i &= \frac{A_i \times B_i}{\|A_i \times B_i\|^{\frac{3}{2}}} \text{ .} \end{aligned} \tag{2.4}$$

The predictor assumes the coefficients a, b, and c to stay the same in the current frame, and thus predicts the position $v(c)_f$ as:

$$v(o(c))_f + aA_f + bB_f + cC_f \text{ .} \tag{2.5}$$

The coefficients are obtained from equations 2.6, where $D_i = v(c)_i - v(o(c))_i$.

$$\begin{aligned} a &= \frac{(A_{f-1} \cdot D_{f-1})(B_{f-1} \cdot B_{f-1}) - (B_{f-1} \cdot D_{f-1})(A_{f-1} \cdot B_{f-1})}{(A_{f-1} \cdot A_{f-1})(B_{f-1} \cdot B_{f-1}) - (A_{f-1} \cdot B_{f-1})(A_{f-1} \cdot B_{f-1})} \\ b &= \frac{(A_{f-1} \cdot D_{f-1})(A_{f-1} \cdot B_{f-1}) - (B_{f-1} \cdot D_{f-1})(A_{f-1} \cdot A_{f-1})}{(A_{f-1} \cdot B_{f-1})(A_{f-1} \cdot B_{f-1}) - (B_{f-1} \cdot B_{f-1})(A_{f-1} \cdot A_{f-1})} \\ c &= D_{f-1} \cdot \frac{A_{f-1} \times B_{f-1}}{\|A_{f-1} \times B_{f-1}\|^{\frac{3}{2}}} \end{aligned} \tag{2.6}$$

The use of the local coordinate system gives Replica the ability to perfectly predict the position of a vertex when the associated parallelogram is transformed by any combination of rigid body motion and uniform scaling.

Encoding

The vertex positions of the seed triangles and the prediction residual values are converted to integer numbers using uniform quantisation. First, a minimal axis-aligned bounding box wrapping the whole animation (i.e. completely containing the mesh in each frame) is constructed and the position of its minimal corner $[x_{min}, y_{min}, z_{min}]$ and maximal corner $[x_{max}, y_{max}, z_{max}]$ are sent to the output data stream. Then, each x-coordinate of the position or the residue is quantised using the following equation:

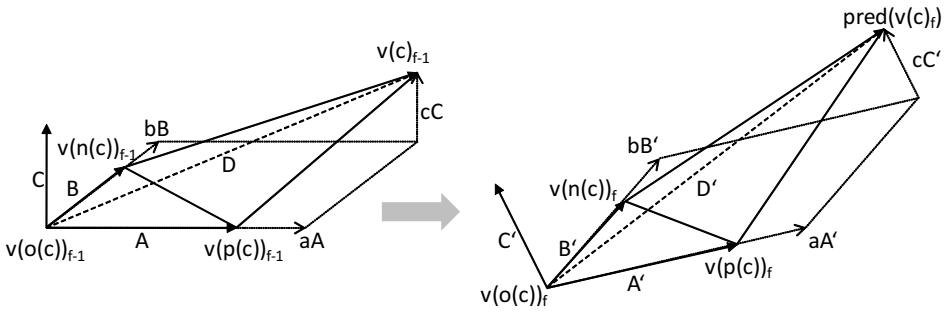

Figure 2.4: Scheme of the Replica predictor.

$$\hat{x} = \text{round}\left(\frac{x - x_{min}}{e \cdot (x_{max} - x_{min})}\right) , \tag{2.7}$$

where e is a user-defined accuracy parameter. The quantised value falls within the interval of $[0; 2^q]$, $q = \log_2\left(\frac{x_{max}-x_{min}}{e}\right)$ being the maximum number of bits needed to represent the value. The y- and z-coordinates are quantised in the same manner. Finally, each quantised triple of $[\hat{x}, \hat{y}, \hat{z}]$ is sent to the output data stream.

Input Parameters

There is only one parameter steering the compression in the Dynapack algorithm. It is the quantisation accuracy parameter, which controls how coarsely the prediction residual vectors will be quantised and thus, how much distortion will be introduced into the data.

Algorithm Discussion

Dynapack is a very simple and fast compression scheme. Its authors make it clear that the aim of it is not to outperform the state of the art algorithms but rather to present a computationally cheap method with a reasonable performance, which could be implemented directly in hardware. For this task, the algorithm works great. However, the algorithm could be greatly improved by employing an entropy coding scheme to the output data stream. For example, the context-adaptive arithmetic coders are very efficient and their hardware implementations are already available.

Note, that the quantisation procedure uses a different step for each coordinate, since the difference of the minimum and maximum value is used as the range for each coordinate. Besides, while this approach might work well for the absolute positions of the vertices, it won't for the prediction residues, because they are scattered around zero, and thus need not to be within the

position range at all.

2.2 FAMC: Frame-based Animated Mesh Coding

This algorithm is based on the *skinning* approach by Mammou et al. [12]. The main idea is to cluster the vertices of the animated mesh into groups, whose motion in each frame can be predicted well with a single affine transformation. Each vertex is then assigned a vector of weights for its cluster and the surrounding clusters describing their influence on the movement of the vertex. The weighted transformations are then applied to the vertex for each frame and a correction vector (or prediction residue) between the transformed and the actual position is calculated. Finally, the entire animation is stored as the group of the first frame mesh (including connectivity), the clustering information, a transformation matrix for each cluster and frame, the weight vector for each vertex and a correction vector for each vertex and frame. We will now describe the process in detail.

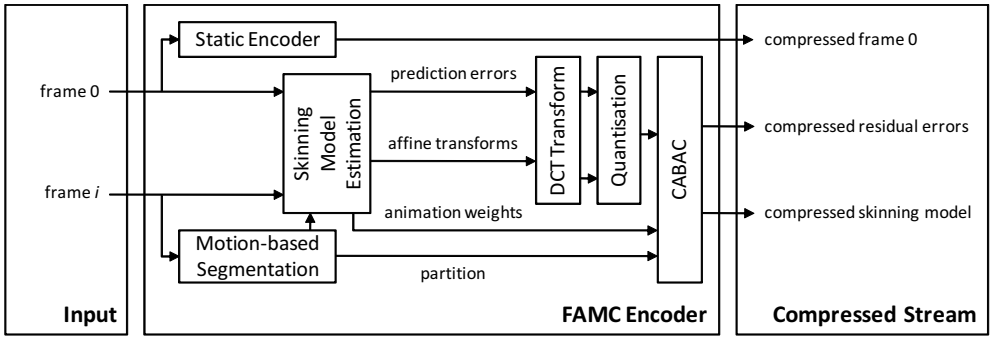

Figure 2.5: Diagram of the FAMC compression algorithm.

Motion-based Segmentation

In this stage, the animated model consisting of frame meshes $M_1 \dots M_F$ is segmented into clusters of points with similar motion. The partition $\Pi = \{\pi_i\}_{i=1\dots N}$ defines N clusters, where each cluster can be in optimal case accurately described using a single affine transform. The FAMC algorithm uses a mean square motion compensation error to define the optimality criterion $E(\Pi)$:

$$E(\Pi) = \sum_{f=1}^{F} \sum_{v=1}^{V} \left\| A_f^{k(v)} p_1^v - p_f^v \right\|^2, \tag{2.8}$$

where $A_f^{k(v)}$ is an affine motion matrix describing cluster motion from the first frame to the f-th frame, and $k(v)$ is an index function, which for a vertex v returns the index k of the cluster v belongs to. The motion matrix has the form described by equation 2.9.

$$A_f^{k(v)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Coefficients t_j describe the translational part of the motion and coefficients a_{ij} describe the rotation and scaling. The matrix is defined by the following equation:

$$A_f^{k(v)} = \arg\min_A \left(\sum_{w \in \pi_{k(v)}} \left\| A p_1^w - p_f^w \right\|^2 \right). \tag{2.10}$$

Given a threshold error E_0, the algorithm is designed to determine a partition Π^* of the input mesh with the minimal number of clusters, for which the resulting motion compensation error $E(\Pi^*) \leq E_0$. This is achieved by hierarchical simplification strategy based on a successive application of the half-edge collapse operator [2]. A half-edge collapse $hc(v, w)$ of an edge (v, w) merges vertex w into vertex v, which remains at its original position. All neighbours of w are connected to v. Vertex v is assigned a set of ancestor vertices v^*:

1. initially: $v^* := \{\}$
2. after $hc(v, w)$: $v^* := v^* \cup w^* \cup \{w\}$

The edge to be collapsed is determined by an objective error function $C(v, w)$, which measures the cost of collapsing edge (v, w). This function is defined by equation 2.11.

$$C(v, w) = \sum_{f=1}^{F} \left(\sum_{u \in \vartheta(v,w)} \left\| A_f^{v,w} p_1^u - p_f^u \right\|^2 \right), \tag{2.11}$$

where $\vartheta(v, w) = v^* \cup w^* \cup \{v, w\}$, and

$$A_f^{v,w} = \arg\min_A \left(\sum_{u \in \vartheta(v,w)} \left\| A p_1^u - p_f^u \right\|^2 \right). \tag{2.12}$$

The coefficients of both matrix $A_f^{k(v)}$ from equation 2.10 and matrix $A_f^{v,w}$ from equation 2.12 can be found using least squares minimisation (LSM) from an overdetermined system of linear equations. First, a matrix M of size $12 \times 3q$ is constructed as described by equation 2.13, where q is the number of vertices in the set in question, i.e. in the cluster π or in the ancestor set ϑ respectively. Matrix M contains vertex coordinates in the first frame. A vector b of coordinates in the f-th frame determines the right hand side. Then, vector a is obtained by solving the system $Ma = b$ by LSM. This vector contains the coefficients of the desired affine transform (equation 2.14). In case the set contains four points or less, the system is exactly determined, or underdetermined, respectively, and there is at least one vector a such that the error function $C(v, w)$ for that set equals zero. This situation occurs at the beginning of the segmentation process and causes the clusters to grow randomly until they have more than four points.

$$M = \begin{pmatrix} x_1^1 & y_1^1 & z_1^1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_1^1 & y_1^1 & z_1^1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_1^1 & y_1^1 & z_1^1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1^q & y_1^q & z_1^q & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_1^q & y_1^q & z_1^q & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_1^q & y_1^q & z_1^q & 1 \end{pmatrix}; \quad b = \begin{pmatrix} x_f^1 \\ y_f^1 \\ z_f^1 \\ \vdots \\ x_f^q \\ y_f^q \\ z_f^q \end{pmatrix} \tag{2.13}$$

$$a = (a_{11},\ a_{12},\ a_{13},\ t_1,\ a_{21},\ a_{22},\ a_{23},\ t_2,\ a_{31},\ a_{32},\ a_{33},\ t_3)^T \tag{2.14}$$

The clustering process starts with the original animated mesh, thus the initial partition Π^0 divides the mesh into $K^0 = V$ clusters, each containing a single vertex. From this state, the half-edge collapse steps are successively performed, in each step collapsing the edge with the lowest cost. A partition obtained after i steps is defined as $\Pi^i = \left\{\pi_k^i = u_k^i \cup (u_k^i)^*; k = 1 \dots K^i\right\}$, where u_k^i is k-th vertex of the mesh resulting from i simplification steps. This process is performed while the global motion compensation error $E(\Pi^i)$ stays below the threshold error E_0.

Skinning-based Motion Compensation

In the previous stage, we have found a segmentation of the animated mesh, where the motion of points in each of the segments can be described using one affine transformation per frame with the motion compensation error not exceeding the specified threshold E_0. However, in

most cases, this description is not accurate enough. Points near the borders of the clusters can, to some extent, also exhibit motion of the neighbouring clusters. Thus, the algorithm introduces the weighted motion compensation model. The motion of a point is here expressed as a weighted combination of the motion of the cluster the point belongs to and the motions of its neighbouring clusters. The predicted position $\hat{p}_f^v$ of vertex v in frame f is then obtained using the following equation:

$$\hat{p}_t^v = \sum_{k=1}^{K} w_k^v A_f^k p_1^v, \tag{2.15}$$

where w_k^v is called the *animation weight*, which is a real value controlling the influence of the motion of cluster k on the motion of vertex v. The animation weights of all the clusters for vertex v together create an *animation weight vector* w^v, which is defined by equation 2.16, where $\theta(v)$ is the vertex set of the cluster including vertex v and its neighbouring clusters. Similarly to the transformation matrices, this vector is found using LSM method. After that, for each frame f and vertex v, the algorithm calculates a correction vector (or prediction residual error) as $\varepsilon_f^v = p_f^v - \hat{p}_f^v$.

$$w^v = \arg\min_{\alpha \in R^K} \sum_{f=1}^{F} \left\| \sum_{k=1}^{K} \alpha_k A_f^k p_1^v - p_f^v \right\|^2 ; \quad k \notin \theta(v) \Rightarrow \alpha_k = 0 \tag{2.16}$$

Encoding

The motion prediction model attempts to minimise the sizes of the residues, so that they can be efficiently stored using an entropy encoder. To further reduce the redundancy present in the data, a global translation vector t_f is determined for each frame f by averaging the motion of all vertices of the mesh as expressed in equation 2.17. These vectors are subtracted from the vertex positions before the segmentation stage, which results in decrease in the translation coefficients of the cluster transform matrices.

$$t_f = \frac{1}{V} \sum_{v=1}^{V} p_f^v - p_1^v \tag{2.17}$$

In the prediction stage, each cluster is assigned a sequence of transformation matrices, one for each frame of the animation. To store the matrices, the algorithm expresses them as a motion of four non-coplanar points and stores the trajectories of these points.

The original design of the algorithm (as described by the MPEG-4 standard [6]) allows one of three methods to be used to transform the point trajectories: Discrete Cosine Transform, wavelet transform based on the Lifting Scheme [1] and direct coding (i.e. no transformation). Since this work aims on comparing the algorithms based on their rate-distortion properties, only the DCT version will be described. According to the authors of the algorithm [13], it is the one that delivers the best performance with respect to this criterion.

In this version, DCT transform is applied to the sequence of global translation vectors, the trajectories of the points determining the cluster motions and also the sequence of residual errors (vectors) of each point of the mesh. As these transformations are one-dimensional, they are applied separately in each coordinate. A Layered Decomposition and Prediction scheme can be employed to enable spatial and temporal scalability of the resulting data stream. As authors show, the use of this scheme does not improve the RD performance of the algorithm, and therefore its description will also be omitted. Detailed information on this topic can be found in [20].

Once the transformation has been performed, all the values are quantised using a uniform quantisation to a certain number of bits q (equation 2.18). A separate value of q is used for the global translation values, the transformation point trajectories, the animation weights, and the residual values, in order to control the compression result precisely. The quantised values, together with the clustering information, are then entropy encoded using the Context-based Adaptive Binary Coding (CABAC) mechanism [11].

$$\hat{x} = \text{round}\left(2^q \cdot \frac{x - x_{min}}{x_{max} - x_{min}}\right) \tag{2.18}$$

Input Parameters

The FAMC algorithm has five input parameters influencing the output data size and distortion. The first parameter specifies the threshold error for the segmentation process. Each of the remaining parameters sets the number of bits q for the quantisation of values in one of the compression stages: the global translation vectors, the cluster transformation matrices, the animation weights and the correction vectors. Since each stage takes in the result of the previous one, they tend to compensate for the errors (both prediction and quantisation) caused by the previous stages. However, as the errors propagate to a subsequent stage, they also increase the size of the data produced by that subsequent stage. For example, increase in quantisation noise in the global translation stage will get compensated in the cluster transformation matrices. The translation elements in all the matrices will change by the same values increasing the redun-

dancy in the data, though the way the matrices are encoded makes it difficult for the arithmetic encoder to efficiently reduce this redundancy in the final output. Similarly, the distortion in the data is highly dependent on the quantisation of the prediction residues as they by design compensate for any errors present in the data after all the previous stages. On the other hand, the larger compensation is needed, the more space will it take to store the residues.

Algorithm Discussion

The skinning approach in general is a very powerful tool in dynamic mesh compression. It attempts to divide the mesh into clusters of vertices with the most similar motion while describing this motion as accurately as possible. The FAMC algorithm improves this approach by extending its properties, like scalable streaming or random frame access. For these properties and a competitive performance, FAMC has been standardised by the Motion Picture Experts Group and included in the MPEG-4 standard. Since then, it has been considered the state of the art. However, we can find some potential flaws that might be degrading the performance of this algorithm.

First, about three quarters of all the edge collapses performed in the mesh segmentation process occur on random edges, as the clusters have too few vertices forcing the collapse cost function to zero. This can cause points with completely unrelated motion to end up in the same cluster. Although such situations are partially compensated by the animation weights, there certainly is a space for improvement, e.g. including a neighbourhood of the edge into the cost function calculation, if the merging clusters have less than five points together.

The algorithm also might be improved by exploiting more the spatial coherence in the data. The Layered Decomposition approach has been employed in the algorithm for this reason, but experiments show that the performance of the algorithm using LD and even LD together with DCT is still worse than with DCT alone. However, using Layered Decomposition also introduces the possibility of progressive data transfer and decoding, which makes the algorithm more versatile.

2.3 D3DMC: Differential 3D Mesh Coding

In 1998, Taubin and Rossignac [21] proposed a compression method for static triangle meshes using their *Topological Surgery* mesh traversal approach. An improved version of this method called 3D Mesh Coding (3DMC) was later accepted as the MPEG-4 standard for static mesh compression [5]. To extend the 3DMC method to dynamic meshes, Müller et al. [15] proposed

a differential octree-based compression scheme – D3DMC. The algorithm describes an animation similarly to conventional video coding. The animation is represented as a succession of *Groups of Meshes* (GOMs), each beginning with an *intra mesh* (I mesh), which is followed by differential meshes (P meshes). The I meshes are coded as static meshes using the 3DMC algorithm, while an octree motion segmentation and prediction is performed on the P meshes.

Spatial Segmentation

The clustering algorithm used by the D3DMC algorithm is based on the spatial clustering by Zhang and Owen [28]. For each mesh (frame) in the sequence, the decoded vertex positions from the previous frame are subtracted from the positions in the current frame. The result is a set of motion vectors Δv describing the motion of the vertices between these two frames. The mesh is then divided into cells of an octree structure and the motion of the vertices in each cell is described by trilinear interpolation of eight motion vectors, one in each corner of the cell.

Each mesh is assigned a minimal bounding box containing all the vertices of the mesh. This box is considered the topmost cell of the octree. Next, a recursive subdivision of the tree is performed, until the motion of the vertices is described with the desired accuracy. For each vertex of the mesh, a trilinear ratio $\rho = [\rho_x, \rho_y, \rho_z]$ is calculated from the position of the vertex $v = [v_x, v_y, v_z]$ in the octree cell as shown in equation 2.19, where $[b_x, b_y, b_z]$ is the position of the cell corner with the minimum x, y, and z.

$$\rho_x = \frac{v_x - b_x}{s_x} \qquad \rho_y = \frac{v_y - b_y}{s_y} \qquad \rho_z = \frac{v_z - b_z}{s_z} \tag{2.19}$$

From this ratio, a set of weights of the cell corners $w_1 \dots w_8$ is calculated using equations 2.20 and the motion Δv of the vertex v is approximated by weighted average $\overline{\Delta v}$ of the corner motion vectors $m^1 \dots m^8$ (equation 2.21).

$$\begin{aligned}
w_1 &= (1-\rho_x)\rho_y\rho_z & w_2 &= (1-\rho_x)(1-\rho_y)\rho_z \\
w_3 &= \rho_x(1-\rho_y)\rho_z & w_4 &= \rho_x\rho_y\rho_z \\
w_5 &= (1-\rho_x)(1-\rho_y)(1-\rho_z) & w_6 &= \rho_x(1-\rho_y)(1-\rho_z) \\
w_7 &= \rho_x\rho_y(1-\rho_z) & w_8 &= (1-\rho_x)\rho_y(1-\rho_z)
\end{aligned} \tag{2.20}$$

$$\overline{\Delta v} = \sum_{i=1}^{8} w_i m^i \tag{2.21}$$

The corner motions are computed using a Least Squares Minimisation from a system of linear equations $Ax = b$, where A is a weight matrix and b is a vector of vertex motions:

$$A = \begin{pmatrix} w_1^1 & 0 & 0 & \cdots & w_8^1 & 0 & 0 \\ 0 & w_1^1 & 0 & \cdots & 0 & w_8^1 & 0 \\ 0 & 0 & w_1^1 & \cdots & 0 & 0 & w_8^1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ w_1^V & 0 & 0 & \cdots & w_8^V & 0 & 0 \\ 0 & w_1^V & 0 & \cdots & 0 & w_8^V & 0 \\ 0 & 0 & w_1^V & \cdots & 0 & 0 & w_8^V \end{pmatrix} \qquad b = \begin{pmatrix} \Delta v_x^1 \\ \Delta v_y^1 \\ \Delta v_z^1 \\ \vdots \\ \Delta v_x^V \\ \Delta v_y^V \\ \Delta v_z^V \end{pmatrix} \tag{2.22}$$

The minimisation process results in vector $x = (m_x^1, m_y^1, m_z^1 \ldots m_x^8, m_y^8, m_z^8)^T$, which contains the components of the corner motion vectors. These motion vectors represent the best estimation of the motion of the vertices enclosed in the currently evaluated cell. Next, the accuracy of the estimation is obtained by substituting the calculated motion vectors into equation 2.21 and subtracting the resulting values from the corresponding vector motions. Two methods can be used to determine the error caused by the estimation: *Max Distance* and *Average Distance*. If the error exceeds a specified threshold, the description of the vertex motion is too inaccurate and the octree cell is evenly subdivided into eight octants. The whole process is then applied to each one of these cells. Once the tree is subdivided enough for all the estimation errors in its leaves to fall under the threshold, the splitting ends.

To determine the corner motions, at least eight vertices have to be present in the cell. After a cell subdivision, some of the child cells may not contain enough vertices. For this reason, D3DMC defines three types of octree cells. *Empty* cells do not contain any vertices and thus may be stored very efficiently. *Sparse* cells contain one to seven vertices, whose motions are stored directly without the motion estimation process. Cells of both these types are automatically treated as leaves and are ignored by the subdivision process. Finally, *Full* cells enclose more than seven vertices, whose motions are estimated by the corner motion vectors. These cells are subject to the subdivision process unless the estimation error is less than the threshold.

Encoding

After the creation of the octree structure has finished, the resulting corner/direct motions in each cell are quantised to a given number of bits. D3DMC employs a uniform quantisation

process similar to the one used in FAMC (equation 2.18). To encode the subsequent frame, the vertex positions in the current frame have to be decoded first. For full cells, the vertex motion estimations are recalculated using the quantised corner motions. In sparse cells, the quantised vertex motions are used directly. The obtained motions are then added to the corresponding vertex positions from the decoded preceding frame. The resulting positions are then subtracted from the positions in the subsequent frame and so on.

For each GOM, the 3DMC-encoded leading I mesh is first written to the output stream. Next, the octree structures of all the P meshes are written. The quantised motions follow, first all their x components, then all y components and finally all z components, each batch encoded by a delta coder (i.e. using constant prediction). Finally, a CABAC method [14] is used to further compress the P mesh data.

Input Parameters

The D3DMC compressor is driven by two main input parameters. The first is the octree split threshold, which sets the maximum acceptable motion estimation error in each cell of the octree. This value controls the depth of the octree and, subsequently, the size of the encoded octree structure and the number of encoded motion vectors. Note that there is a limit to which the octree can be subdivided – once each leaf of the tree only contains eight vertices or less, the error falls to zero and no further subdivision is performed. On the other hand, should the threshold be too high, the initial octree cell will not split and the whole mesh would be encoded by this single cell.

The second parameter is the number of bits used for quantisation of the motion vectors. Since the motions in each frame are calculated using the decoded version of the previous frame, quantisation noise appears in the motion vectors. Setting a too coarse quantisation may decrease the performance of the algorithm by adding too much noise to the motions, and thus forcing more subdivisions of the octree.

Algorithm Discussion

The D3DMC algorithm is based on an idea different from most of the other compression methods – instead of a rather complicated segmentation it uses a simple regular space subdivision. The simplicity of this approach is compensated by a smaller amount of data needed to store the structure. However, this amount could be potentially further reduced, as in most cases, the octrees of the subsequent frames are similar one to another. Instead of storing the whole tree structure, only changes to the structure between each two frames could be stored. Similar approach can be used for the motion vectors. While currently, a delta prediction is applied in

the spatial direction (i.e. inside each frame separately), it would be more sensible to employ a prediction scheme in the temporal direction.

Since the motion of each part of the mesh is described only by the corner motions of its enclosing cell, unwanted artifacts may occur on a border of two cells. These can be especially disturbing, if the corner motions of neighbouring cells cause the cells to overlap each other. In such case, the vertices near the border may form a visible fold on the mesh surface. These artifacts are similar to "blocking", which occurs in JPEG compressed images. Unfortunately, the way the corner motions are calculated and the fact that the octree subdivision is not symmetrical (i.e. there can be subdivided and not subdivided cells on the same level of the tree) does not leave much space for resolving this issue. A simple blending, where the motion of vertices incident to an border-crossing edge would be calculated as an average of the motions of both neighbouring cells, might be a quick solution. Other than that, the only way to get around this problem would be a complete redesign of the segmentation and motion estimation process.

2.4 CoDDyaC: Connectivity-Driven Dynamic Mesh Compression

The Coddyac algorithm [24] is designed to combine very powerful methods exploiting the spatial and temporal coherence in the data: Principal Component Analysis (PCA) in the space of trajectories and an Edgebreaker-like mesh traversal. Most other compression approaches neglect exploiting one coherence type to some extent, and thus are expected to perform worse compared to Coddyac. The algorithm consists of these stages:

1. The trajectory of each vertex is represented as a column in a trajectory matrix of size $V \times 3F$.
2. A basis of the trajectory space is obtained by analysing the trajectory matrix using PCA.
3. N most important vectors of the basis are selected and encoded.
4. Each vertex is assigned an N-component vector of PCA coefficients (feature vector) representing its trajectory as a linear combination of decoded basis vectors.
5. The mesh topology is processed by the Edgebreaker traversal, a parallelogram prediction is employed to predict the feature vectors.
6. Prediction residues are encoded.

Principal Component Analysis

PCA [18] is a statistical method that transforms a set of correlated variables into a set of completely uncorrelated variables – *principal components*. The first principal component describes as much variability in the original set as possible (using a single vector), and each subsequent component accounts for as much of the remaining variability as possible. As a result, the principal components are sorted from the most important one to the least important one. In terms of vertex trajectories, the first principal component is the trajectory that describes the most of the motion of the mesh, the second one describes the most of the remaining motion, etc. Since each principal component is a linear combination of the original vertex trajectories, PCA can be viewed as a simple change of basis, where the new basis consists of the principal components.

To compute the PCA basis, the algorithm uses eigenvalue decomposition of an autocorrelation matrix. First, matrix $B = b_{i,j}$ of vertex trajectories is created, where each column consists of the components of the position of a single vertex in all the frames – a total of $3F$ values. All the trajectories must contain the values in the same order, but the order itself is irrelevant, since it only means a row permutation. In Coddyac, the trajectories contain first the x, y, and z values from the first frame, then x, y, and z from the second one, etc. A matrix of samples $S = s_{i,j}$ is constructed by subtracting an average trajectory $A = a_i$ from the vertex trajectories:

$$s_{i,j} = b_{i,j} - a_i \qquad a_i = \frac{1}{V}\sum_{j=1}^{V} b_{i,j}. \tag{2.23}$$

Next, the autocorrelation matrix $Q = S \cdot S^T$ of size $3F \times 3F$ is computed. The eigenvectors $E_i, i = 1 \ldots 3F$ obtained from eigenvalue decomposition of Q form the PCA basis of the animation. The importance of the eigenvectors is determined by their corresponding eigenvalues (the higher value, the more important vector).

The key idea of the Coddyac algorithm is that most of the importance lies within only a small number of the basis vectors. Thus, the rest of the basis vectors can be discarded without introducing too much distortion into the mesh. A specified number N of the most important vectors is selected. These vectors form a basis of a subspace of the original trajectory space. The trajectory of each vertex is then expressed in this basis as:

$$T_i = A + \sum_{j=1}^{N} c_{i,j} E_j, \tag{2.24}$$

where $C = c_{i,j}$ is the matrix of combination coefficients obtained by multiplication $C = S^T \cdot E$,

E being a basis matrix, in which the k-th column is the k-th most important eigenvector E_k. Each row C_i of C is a so-called *feature vector*, which contains coordinates of the trajectory of i-th vertex in the truncated PCA basis.

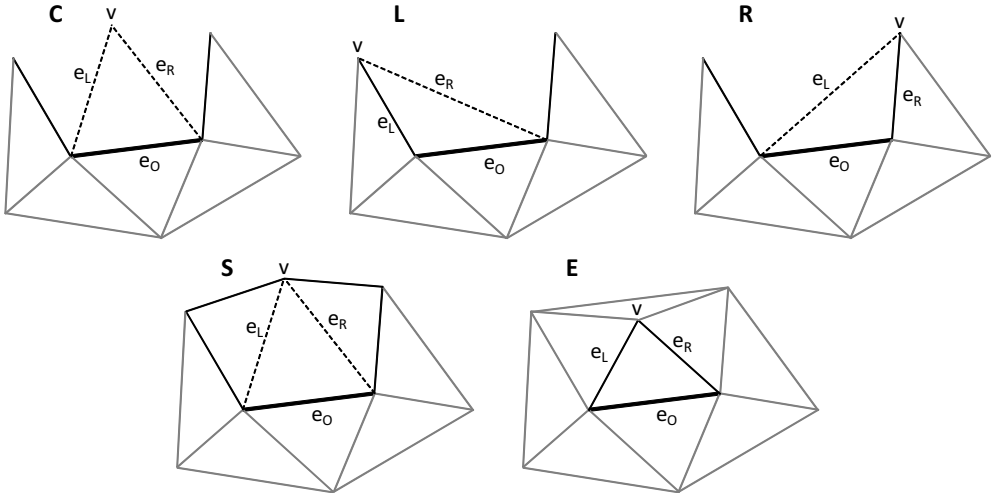

Figure 2.6: Edgebreaker traversal cases. The open edge is marked by a thick line, new edges by a dashed line, and border edges are coloured black.

EdgeBreaker and Prediction

The Edgebreaker is a simple and efficient triangle mesh connectivity encoder proposed by Rossignac in [16]. It is based on a specific traversal through the mesh triangles. First, a starting triangle is selected and directly encoded. The edges of this triangle define the initial progressing border. One of the edges is marked as *open* and the traversal starts growing the border through this open edge. In each step of the traversal, an open edge e_O connects the current triangle with a single incident triangle containing this edge and two other edges, e_L on the left and e_R on the right, which meet in vertex v. Depending on the relation of the border and these edges, one of these cases will occur:

C: Vertex v has not been crossed by the progressing border yet. Both e_L and e_R are added to the border, e_O is removed from the border, and e_R is marked as open.

L: Only e_L is part of the border. e_R is added to the border and marked as open, while e_L and e_O are removed from the border.

R: Only e_R is part of the border. e_L is added to the border and marked as open, while e_R and e_O are removed from the border.

S: Vertex v lies on the border, but neither e_L nor e_R are part of it. Both e_L and e_R are added to the border, e_O is removed from the border, and e_R is marked as open.

E: Both e_L and e_R are part of the border. All three edges are removed from the border and the procedure backtracks the processed triangles to find a border edge. The first one found is marked as open. If no border edge is found, the traversal ends.

Since the PCA step is, in fact, only a change of basis, the feature vectors can be predicted in the same way as the original vertex coordinates. In Coddyac, a simple parallelogram prediction of the feature vectors is performed during the Edgebreaker traversal. Each C-step of the traversal means crossing an open edge e_O from the current triangle $t_{current}$ to the incident one t_{new} and discovering a previously undiscovered vertex v_{new}. Let v_L and v_R be the left and the right vertex of e_O, v_O the remaining third vertex of $t_{current}$, and C_L, C_R, C_O their respective feature vectors. The feature vector C_{new} of vertex v_{new} is then predicted by equation 2.25, and a prediction residue $R_{new} = \mathrm{pred}\,(C_{new}) - C_{new}$ is then calculated from this prediction.

$$\mathrm{pred}\,(C_{new}) = C_L + C_R - C_O. \tag{2.25}$$

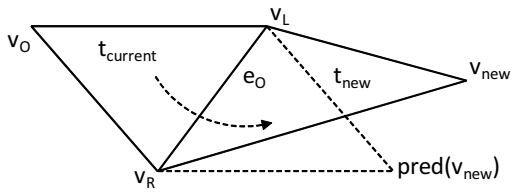

Figure 2.7: Prediction of vertex v_{new} from the known vertices of triangle $t_{current}$ using the parallelogram rule.

Encoding

To successfully decode the animation, the decoder needs to have the basis vectors together with the average vertex trajectory, and the feature vectors for all the vertices. To store the basis, the Coddyac algorithm utilises the Cobra encoder described in [25].

The feature vectors are compressed using the parallelogram prediction described above. The full feature vectors of the first triangle of the mesh traversal are stored, followed by the prediction residual vectors. Each component a of each of these vectors is uniformly quantised as described in equation 2.26, where l is the average edge length of the mesh and q is a user specified fineness of the quantisation. Finally, the quantised feature vector data is processed by a CABAC encoder.

$$\hat{a} = \mathrm{round}\left(2^q \cdot \frac{a}{l}\right) \tag{2.26}$$

Input Parameters

The compression in the Coddyac algorithm is controlled by three parameters. The first one is the number of vectors N the PCA basis is truncated to. This value has a strong influence on the size of the encoded basis and also the size of the feature vector data, since each feature vector has exactly N components. The Cobra encoder takes in the second parameter, which is the quantisation fineness q used in the basis vector encoding. The last parameter is again a quantisation fineness q, which, in this case, steers the quantisation process of the feature vector encoding. While the first two parameters affect the quality of the PCA basis encoding and the third one the accuracy of the feature vectors, they all have an impact on the quality of the reconstructed vertex trajectories and thus the quality of the compressed animation.

Algorithm Discussion

The Coddyac algorithm satisfies the assumptions made in its design and in a vast majority of cases outperforms its competitors. One of its advantages lies within the distribution of error caused by the PCA-based compression, which appears as a loss of small details in the vertex trajectories rather than motion discontinuities exhibited by D3DMC and FAMC, or quantisation noise in the case of Dynapack.

However, using PCA in the space of trajectories means that most of the data (at least the whole set of feature vectors) has to be available to the decoder in order to decode any single frame of the animation making Coddyac unsuitable for use in applications where random frame access is crucial. Nevertheless, for a standard animation coding, where it is enough to have random access to the keyframes, this approach fits fine.

Recent research has also shown that there still is room for improvement of the algorithm. Using only a simplified mesh in the prediction of the feature vectors and interpolating the rest of them using Radial Basis Function improves the performance of the algorithm while also enabling for a progressive data transfer and decoding as shown in [26]. Employing a novel Laplacian-based quantisation approach by Sorkine et al. [19] might also improve the performance and the error distribution.

3

Rate-Distortion Optimisation

Rate-distortion (RD) optimisation, in terms of general lossy compression, is a technique that configures a compression algorithm for optimal performance. In most cases, this configuration depends on the input data, thus we have to perform it again for every new dataset. There is usually a set of many possible optimal configurations for each input data. RD optimisation is a process that finds a single configuration from this set that satisfies a specified constraint, e.g. a particular target bitrate.

In video compression, for example, a multiple-pass compression is often performed in order to achieve the best result. In the first pass, the input data is analysed. Based on this analysis, each subsequent pass attempts to configure the compression process for the best quality, while maintaining a user-specified overall bitrate or quality. Simultaneously, the analysis is updated based on the current compression result, thus each pass is expected to produce a result better than the previous one. Most video encoders only require the constraint value (target quality or bitrate) as an input. However, internally, they might use many different configuration parameters. The values of these parameters are either preset to defaults by the authors of the algorithm, or determined from the input data and the behaviour of the compression process.

For the dynamic mesh compressors from the previous chapter, the configuration parameters are exposed to the user and require to be set before the compression starts. To reach a state similar to video compression, where a configuration is selected automatically based on the given constraint value, we need to use a RD optimisation method.

3.1 Rate and Distortion

As was mentioned before, the performance of a lossy compression algorithm can be measured int terms of the distortion it introduces to the compressed data and the bitrate of the result. If we run the compressor on a given data with a value set P of n input parameters $p_1 \dots p_n$, it will produce compressed data of a certain bitrate with a certain distortion compared to the original data. These two values – rate, r and distortion, d – can be plotted on a rate-distortion chart H. For a single input dataset, different parameter configurations will produce different points $h = [r, d]$ in this chart. In other words, the compression can be seen as a projection $C; C(P) \mapsto H$. Assuming the parameters are continuous, there is generally an infinite number of such points. However, a lower bound can be found for each bitrate forming an envelope curve O of the chart:

$$O = \{o = (r_o, d_o) : \forall x = (r_o, d) \in H, d \geq d_o\} \tag{3.1}$$

This curve contains the parameter configurations that result in the lowest distortion for a given bitrate, i.e. optimal configurations, which is exactly what we are looking for. Experiments show, that for most dynamic mesh compression methods, including the ones we focus on, the curve O is decreasing and convex.

Note: From this point further, the term "compression" will be equivalent to dynamic mesh compression.

3.2 Principle of Equal Slopes

Now, let us choose an *optimal* configuration $P^j; C(P^j) \in O$ and change the value of a single parameter p_i while keeping the other parameters fixed. We will obtain an RD curve H_i^j. This way, we can construct curves for all the parameters. We will assume that all these curves are also decreasing and convex. As these curves are subsets of H, they lie above O – except for a single point, where they touch O and also one another. This point is the result point $C(P^j)$. Should any two of the curves *intersect* in that point, they would also intersect O meaning that a part of each of them would lie below O contradicting its definition, thus the configuration cannot be optimal.

The idea can be explained better using *slopes*. For each point h_i^j of a curve H_i^j we can calculate the slope value s_i^j of a tangent of the curve in that point. Two convex curves with a common

point either have different slope values at that point (they are intersecting), or they have equal slopes in the common point (they are touching each other). We have already defined that in an optimal configuration all the curves are touching one another, and a configuration, in which the curves are intersecting, is not optimal. Hence we can say that *a parameter configuration P^j is optimal if and only if the slopes s_i^j of all rate-distortion curves H_i^j in the result point $C(P^j)$ are equal.*

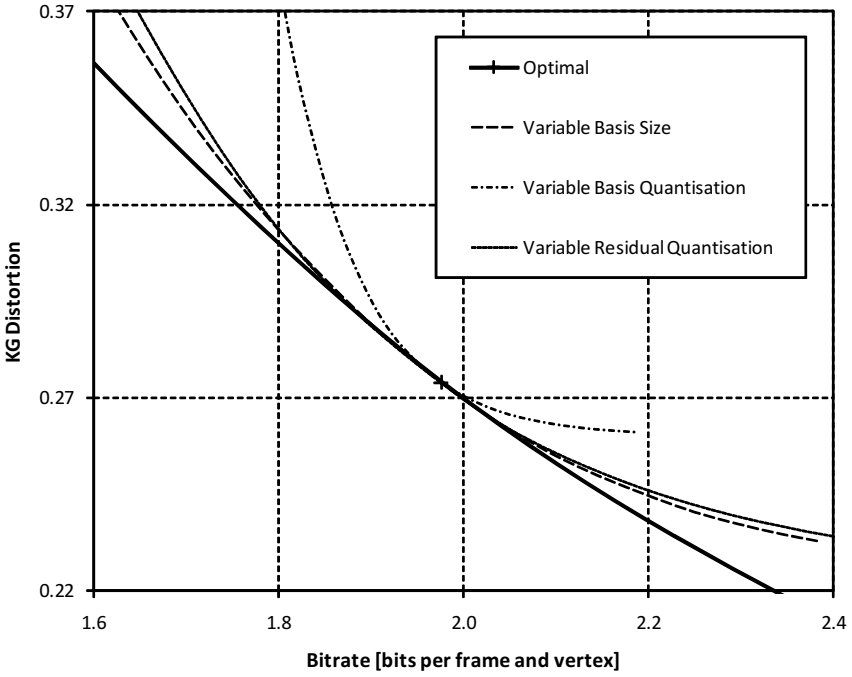

Figure 3.1: Parameter curves of the Coddyac algorithm touching the envelope curve in an optimal configuration.

3.3 Iterative Optimisation

Based on the Principle of Equal Slopes, we have designed an algorithm that iteratively refines a parameter configuration until it gets close enough to an optimum. In each iteration, the pair of r and d is evaluated by running the compression algorithm with the current configuration. Two additional pairs of values for each parameter p_i are evaluated by shifting the value of that parameter a small step downwards ($r_i^{\text{left}}, d_i^{\text{left}}$), and a small step upwards ($r_i^{\text{right}}, d_i^{\text{right}}$), as shown in equation 3.2. From these values, we can then calculate a local approximation of the left and right slope $s_i^{\text{left}}, s_i^{\text{right}}$ (equation 3.3).

$$\begin{aligned}(r, d) &= C(p_1, \ldots, p_n) \\ (r_i^{\text{left}}, d_i^{\text{left}}) &= C(p_1, \ldots, p_i - \Delta p_i, \ldots, p_n) \\ (r_i^{\text{right}}, d_i^{\text{right}}) &= C(p_1, \ldots, p_i + \Delta p_i, \ldots, p_n)\end{aligned} \tag{3.2}$$

$$s_i^{\text{left}} \approx \frac{r - r_i^{\text{left}}}{d - d_i^{\text{left}}} \qquad s_i^{\text{right}} \approx \frac{r_i^{\text{right}} - r}{d_i^{\text{right}} - d} \tag{3.3}$$

Finally, a new configuration is then determined using these values. This configuration is then used as an input of the next iteration and so on, until a stop-condition is met. For each new configuration P^j, the overall deviation e^j is calculated. Once this deviation falls under a specified threshold value, the optimisation ends.

Optimisation Criteria

The optimisation process needs to be constrained to exactly determine the resulting configuration. Therefore, a certain criterion needs to be applied. This criterion also specifies the calculation of the overall deviation for the stop-condition of the iterative configuration refinement. There are four criteria we have considered and implemented:

- **Fixed Slope**

  This criterion allows us to specify a target slope value s^* the final slopes should be equal to. The overall deviation is here defined as the maximum of the distances between the specified slope and each parameter slope:

  $$e^j = \max_{i=1\ldots n} |s^* - s_i^j| \tag{3.4}$$

  This method is not very useful most times, since we usually do not know the magnitude of the bitrate and the distortion, and thus we cannot exactly determine the desired slope.

- **Fixed Bitrate**

  With this criterion applied, the algorithm tries to find an optimal configuration which results in the specified bitrate, r^*. It can be quite effectively used if we want to compare the compression algorithm with another one for which we know exact RD values. The overall deviation depends on both bitrate and slope differences. It is evaluated using the maximum slope distance from an average slope and the absolute deviation from the target bitrate:

$$e^j = \sqrt{\left(e_s^j\right)^2 + \left(e_r^j\right)^2} \tag{3.5}$$

$$e_s^j = \max_{i=1\ldots n} \left| \left(\frac{1}{n} \sum_{k=1}^{n} s_k^j \right) - s_i^j \right| \qquad e_r^j = |r^* - r^j| \tag{3.6}$$

- **Fixed Distortion**

  The idea here is the same as in the previous case, only the output configuration is bound to have a given distortion d^* instead of bitrate. The overall deviation is also defined similarly to the fixed bitrate case:

$$e^j = \sqrt{\left(e_s^j\right)^2 + \left(e_d^j\right)^2} \tag{3.7}$$

$$e_s^j = \max_{i=1\ldots n} \left| \left(\frac{1}{n} \sum_{k=1}^{n} s_k^j \right) - s_i^j \right| \qquad e_d^j = |d^* - d^j| \tag{3.8}$$

- **Fixed Parameter**

  This criterion drives the optimisation with respect to a given value of a specified parameter. It is a very useful method for constructing the optimal RD curves, since we usually know the possible value range of the parameters, while we might not know the resulting bitrates, distortions and slopes. In each iteration, the slope s_k^j of the fixed parameter p_k is evaluated and the overall deviation is then determined in a similar manner as in the fixed slope case:

$$e^j = \max_{i=1\ldots n} |s_k^j - s_i^j| \tag{3.9}$$

Next Configuration Calculation

Once we have acquired the information about the local curve behaviours, we can determine the configuration for the next iteration. The algorithm approximates the shape of the parameter curves by functions of their known local behaviour, and then calculates the next configuration using these functions. The quality of the curve approximation influences the speed and the probability of convergence to the desired result. We have considered three different methods of determining the next configuration:

- **Linear Approximation in All Parameters**

  This method approximates the slope of a parameter curve by a linear function of the corresponding parameter. Note that we cannot use a linear function to approximate the curve

itself, since in that case, the slope of the approximation would stay constant. In each iteration, relative differences of slope (ds_i), rate (dr_i), and distortion (dd_i) are calculated for each parameter p_i, as shown in equation 3.10.

$$ds_i = \frac{s_i^{\text{right}} - s_i^{\text{left}}}{\Delta p_i} \qquad dr_i = \frac{r_i^{\text{right}} - r_i^{\text{left}}}{2\Delta p_i} \qquad dd_i = \frac{d_i^{\text{right}} - d_i^{\text{left}}}{2\Delta p_i} \tag{3.10}$$

The slope equality condition then determines $n - 1$ linear equations (3.11). One additional equation is then constructed based on the selected optimisation criterion (3.12). These equations together form a linear system with variables $\zeta_1 \dots \zeta_n$. The solution of this system defines the parameter shifts for the next configuration, i.e. the next value of each parameter p_i is calculated by adding the corresponding shift ζ_i to its current value.

$$\begin{aligned} ds_1\zeta_1 - ds_2\zeta_2 &= s_2 - s_1 \\ &\vdots \\ ds_1\zeta_1 - ds_n\zeta_n &= s_n - s_1 \end{aligned} \tag{3.11}$$

$$\begin{aligned} \text{fixed slope:} \qquad & ds_1\zeta_1 = s^* - s_1 \\ \text{fixed rate:} \qquad & dr_1\zeta_1 + \dots + dr_n\zeta_n = r^* - r \\ \text{fixed distortion:} \qquad & dd_1\zeta_1 + \dots + dd_n\zeta_n = d^* - d \\ \text{fixed parameter } p_i\text{:} \qquad & \zeta_i = 0 \end{aligned} \tag{3.12}$$

This method can be very fast, as it optimises all the parameters in each step, but it does not adapt very well to curve changes. When one of the parameters is changed, the shape of another parameter's curve may also change, especially if there are dependencies between these parameters. This method, however, expects the curves to stay invariant. This fact can lead to actually less optimal configurations and, subsequently, divergence, if the curves change too much. Moreover, experiments show, that linear approximation of slopes is not very accurate.

- **Linear Approximation in a Single Parameter**

  To improve the convergence probability of the previous method, we can limit the number

of parameters optimised at a time. By only changing one parameter in each iteration, we can eliminate the reliance on curve invariance. On the other hand, this step may significantly slow down the optimisation, especially for compression algorithms with high parameter counts.

The next configuration is here calculated as follows. First, the parameter p_k with the most deviating slope is selected. For the fixed slope case, it is the parameter, whose slope deviates the most from an average slope. For fixed rate and distortion, it is the one, which deviates the most to the wrong direction (i.e. if its slope changes towards the average, the target quantity will move towards the desired value). In the fixed parameter case, it is, naturally, the parameter with a slope most distant from the slope of the fixed parameter. Next, the shift ζ_k of the selected parameter is calculated according to equations 3.13, where $s_{\mathrm{avg}(k)}$ is the average of slopes not including s_k and the relative differences ds_i, dr_i, and dd_i are defined the same way as in the previous method (equation 3.10). The weight w determines the priority given to reaching the target rate or distortion over the slope equality. Its value is proportional to the deviation in the target quantity, i.e. the closer we get to the desired value, the more weight is given to equalising the slopes and vice versa. This accelerates the convergence in case the most deviating slope is close to the average slope.

$$\begin{aligned}
\text{fixed slope:} \qquad & \zeta_k = \frac{s^* - s_k}{ds_k} \\
\text{fixed rate:} \qquad & \zeta_k = w\frac{r^* - r}{dr_k} + (1-w)\frac{s_{\mathrm{avg}(k)} - s_k}{ds_k} \\
\text{fixed distortion:} \qquad & \zeta_k = w\frac{d^* - d}{dd_k} + (1-w)\frac{s_{\mathrm{avg}(k)} - s_k}{ds_k} \\
\text{fixed parameter } p_i\text{:} \qquad & \zeta_k = \frac{s_i - s_k}{ds_k}
\end{aligned} \tag{3.13}$$

■ Exponential Approximation in All Parameters

Another way of improving convergence probability of the first method is to better describe the shape of the curves. In this case, we are using an exponential function $s_k = a_k b_k^{p_k}$ to determine the slope. It has been experimentally found, that for the algorithms we focus on, this function is a much closer approximation of the slope than a linear function. From the known information about the parameter curve, the coefficients a_k and b_k are calculated for each parameter p_k, as shown in equations 3.14, where $p_k^{\mathrm{left}} = p_k - (\Delta p_k)/2$. The parameter shifts are then determined using this approximation (equations 3.15).

$$a_k = s_k^{\text{left}} \left(\frac{s_k^{\text{left}}}{s_k^{\text{right}}} \right)^{-\frac{p_k^{\text{left}}}{\Delta p_k}} \qquad b_k = \left(\frac{s_k^{\text{left}}}{s_k^{\text{right}}} \right)^{\frac{1}{\Delta p_k}} \tag{3.14}$$

$$\begin{aligned}
\text{fixed slope:} \qquad & \zeta_k = \frac{\ln(s^*) - \ln(a_k)}{\ln(b_k)} \\
\text{fixed rate:} \qquad & \zeta_k = \frac{(r^* - r) + \sum_{j=1}^{n} \left[\frac{dr_k \ln(-s_k)}{\ln(b_k)} \right]}{\ln(b_k) \sum_{j=1}^{n} \left[\frac{dr_k}{\ln(b_k)} \right]} - \frac{\ln(-s_k)}{\ln(b_k)} \\
\text{fixed distortion:} \qquad & \zeta_k = \frac{(d^* - d) + \sum_{j=1}^{n} \left[\frac{dd_k \ln(-s_k)}{\ln(b_k)} \right]}{\ln(b_k) \sum_{j=1}^{n} \left[\frac{dr_k}{\ln(b_k)} \right]} - \frac{\ln(-s_k)}{\ln(b_k)} \\
\text{fixed parameter } p_i\text{:} \qquad & \zeta_k = \frac{\ln(s_i) - \ln(a_k)}{\ln(b_k)}
\end{aligned} \tag{3.15}$$

Because of the more accurate approximation of the slope behaviour, this method is in most cases faster than the previous two methods and has a high probability of convergence. However, it exploits an additional knowledge about the optimised algorithms, thus it might not be as generally applicable as the linear version.

4

Implementation

In this chapter, the implementation details of the described compression algorithms and the proposed method of rate-distortion optimisation will be explained. The implementation of both parts was carried out in the form of modules on the Modular Visualization Environment 2 (MVE2) platform [23].

4.1 Modular Visualization Environment 2

It is a data-flow-based scientific data visualisation and processing tool developed at the University of West Bohemia. MVE2 consists of a graphical or command-line front-end, an execution core, and a library of *modules* and standard data types. It offers a platform for creating and running experiments and collecting results. Each experiment is defined as a *map* consisting of modules linked together in a data-flow manner and their settings. A module has an interface of input and output ports, through which it is connected to other modules in the map (i.e. an output of a module is linked to an input of another module).

The module library contains, among others, modules for loading various file formats including the most common static and dynamic triangle mesh formats, many data processing modules (e.g. normal computation, colour mapping, iso-surface extraction), and output modules, such as renderers and file savers. The platform includes intuitive API for creating custom modules and data types.

Data Types

MVE2 contains a large number of predefined data structures for various data representations. Dynamic triangle meshes are here represented as an array, where each element is a static tri-

angle mesh representing a single frame of the animation. For an array with all elements of the same type, MVE2 defines the `UniformDataArray` type. Each frame is then stored in a `TriangleMesh` structure, which is a special case of an unstructured grid – the `UnstrGrid` data type. Each unstructured grid contains an array of vertex positions (the `Point3D` type) and an array of cells, which determine the connectivity. For the `TriangleMesh` subclass, the vertices are connected through triangle cells (type `Triangle`). The `UnstrGrid` type can also contain arrays of point and cell attributes (e.g. cluster membership).

Module Design

Each module in MVE2 is created as a subclass of the `Module` API class. This class defines default implementation of the core module methods. The most important one is the `Execute` method, which is called each time a linked module requests an output of this module. The `GetInput(portName)` method retrieves data from an input port with the specified name. After the processing is done, the output data is exposed at an specified output port by calling `SetOutput(portName, data)`. Ports are added to the module using the `AddInPort(portName, dataType)` and `AddOutPort(portName, dataType)` methods, usually in the class constructor.

The settings of a module are by default represented by public read-write properties of the module class. Each such property is visible in the module setup dialog and is stored in the map file. Both the saving/loading of the settings and the setup dialog can be customised should the author of the module need it.

4.2 Compression Algorithms

The implementation of the Dynapack and Coddyac algorithms is already available as a part of *Dynamic Mesh Compression Toolkit* for MVE2, which can be obtained from the MVE2 Module Repository [23]. Details of the implementation of these two algorithms can be found in the documentation of the toolkit. The toolkit also contains implementations of the distortion metrics used in this work.

The D3DMC and FAMC algorithms had to be reimplemented, since to the day, we were unable to obtain a working implementation from the authors of these algorithms. The implementation of the algorithms was based on the available documentation and published papers. While we were able to replicate the published results for D3DMC, our implementation of the FAMC algorithm does not reach the compression ratios claimed by the authors of the algorithm.

D3DMC

The D3DMC compression algorithm is implemented as a single module, `D3DMCCompressor`. This module has three settings:

- number of bits for the keyframe quantisation
- number of bits for the motion vector quantisation
- octree split error threshold

Three input ports are attached to the module, `InAnimation` for the input mesh animation and optional `InQBits` and `InSplitThr` for externally overriding the latter two of the settings. Output ports cover the decompressed animation (`OutAnimation`) and the bitrate in bpfv (`OutBitrate`).

The compression procedure is implemented in the `Compressor` class. First, an instance of this class is created and given the input animation and the compression parameters. The constructor initialises the required data structures and the quantiser. The decoded vertex positions can be then obtained from the `DecodedPositions` property of this class. This property is lazy-initialised, thus the compression itself is performed the first time it is accessed. The first frame of the animation is compressed through the 3DMC compressor. After that, for each pair of consequent frames, the motion octree structure is built as an instance of the `Octree` class. For each tree, the root cell is initialised and inserted into a queue. A loop then removes one cell from the queue in each iteration, until the queue is empty. If the motion compensation error in this cell is higher than the threshold, the cell is split into octants and these are pushed into the queue again. Immediately after the creation of each cell, the positions of the enclosed vertices are decoded using the corner motions of the cell, and the motion compensation error is calculated. Finally, the decoded positions from leaves of the tree are collected and returned as a result.

A wrapper class, `D3DMCCompressorImpl`, is created as an interface between the module and the `Compressor` class. It helps to separate the pure compression procedure from the data encoding and the module logic. This class has two important methods. The `Compress()` method reads the decoded positions from the compressor instance, and then reconstructs and returns a data structure with the decoded animation (a `UniformDataArray` of `UnstrGrid` elements). The `WriteData(stream)` method encodes the compressed animation into the provided stream and collects statistics about the encoded data, such as the total size and bitrate of the data. These informations are then returned to the module in a `CompressorStats` structure.

FAMC

The implementation of the FAMC compression algorithm is divided into two separate modules. The motion-based segmentation is carried out by the `FAMCClusterizer` module and the `FAMCCompressor` module then performs the compression itself.

■ Clusteriser

From specification, the segmentation process of the FAMC algorithm is driven by a single parameter, which is the threshold error of the motion approximation. The magnitude of this error is usually unknown to the user. Therefore, we have added an option to specify the number of resulting clusters instead. Because of that, the `FAMCClusterizer` module has two settings: the error threshold and the number of clusters, and a switch to choose, which one of those settings will be used. Input ports include the animation input (`InAnim`) and two ports to override the settings (`InMaxError` and `InNumClusters`). The port `OutAnim` then outputs the input animation with a cluster index attached to each vertex as an point attribute, and the `OutClusters` port outputs the cluster connectivity (i.e. the remaining non-collapsed edges of the mesh).

The `Clusterizer` class contains the implementation of the main segmentation procedure. The class constructor initialises an array with one `ConnectedPoint` instance for each vertex and builds a priority queue of `ConnectedEdge` instances, one instance per mesh edge. The `ConnectedPoint` structure holds the list of ancestors of the corresponding vertex and a list of edges leading from this vertex. The `ConnectedEdge` structure represents an edge of the mesh. It stores the indices of its endpoint vertices and the cost of collapsing the edge. The edges are stored in a priority queue, where the topmost edge is the one with the lowest collapse cost. The `CreateEdge(v1, v2)` method constructs and edge between the specified vertices and calculates its collapse cost by calling a delegate cost function specified as a parameter of `Clusterizer` constructor. An array containing cluster index for each vertex is returned by the `GetClusterIndices()` method, and an array with the non-collapsed edges is returned by the `GetClusterEdges()` method. The first one called of these two methods builds the clustering by invoking the `CreateClusters()` method. This method iteratively performs the half-edge collapse of the topmost edge in the priority queue and tests for a stop-condition by calling a delegate method, again, specified through the constructor. Once the condition is met, the collapsing ends and the clustering array is returned. Using external delegate functions allows us to use the `Clusterizer` class also outside of FAMC with other definitions of cost functions and stop-conditions.

Similarly to the D3DMC implementation, a wrapper class, `FAMCClusterizerImpl`, is

created. This class contains the delegate method `CostFunction()`, which calculates the cost of collapsing the specified edge, and methods `ClustCntStopCondition` and `MaxErrStopCondition` for testing the cluster count and error threshold stop-conditions. Finally, the `GetAnimation()` and `GetConnectivity()` methods are the heart of this class. The first one acquires the clustering array from the `Clusterizer` instance, embeds it into the point attribute of each frame of the input animation, and returns the result. The second builds and returns an unstructured grid containing the non-collapsed vertices as points and the edges between them as cells.

- **Compressor**

  The implementation of the FAMC compressor shares the same structure as the above described implementations. The module is represented by the `FAMCCompressor` class. It has four settings corresponding to the quantisation parameters of the FAMC algorithm:

  - `GlobalTranslationQBits`,
  - `ClusterTransformQBits`,
  - `ClusterWeightsQBits`,
  - `PredictionErrorQBits`,

  and a fifth setting specifying the quantisation of the first frame (`FirstFrameQBits`). For each of these settings, a corresponding input port is also defined. The `InAnim` input port is defined to obtain the input animation with the clustering attribute and the `InClusters` input port takes in the cluster connectivity. Two output ports are also defined: `OutAnim`, which outputs the decompressed animation, and `OutRate`, which provides the resulting bitrate in bpfv.

  The core compression procedure is implemented in the `Compressor` class. The constructor of the class initialises the required data structures, encodes the first frame, computes and encodes the global translation vectors, and restores the clustering structure from the input data. Although the published descriptions of the algorithm does not mention the compression of the first frame, the MPEG-4 standard states that 3DMC encoder is used for this purpose. The `DecompressedPositions(frame)` method provides an array with decoded vertex positions in the specified frame. The first time this method is called, the compression process is performed. The `ConstructTransformationMatrices()` method is called to obtain the cluster transformations in each frame. The `ComputeClusterWeights()` method then determines the transformation weights for each vertex. Residual vectors are retrieved by calling `ComputeReziduals()` and finally, the decoded positions are recovered by the `Decode()` method.

The `FAMCCompressorImpl` class takes care of reconstructing the decoded animation data structure through the `GetAnimation()` method. Four classes were designed to store and encode the results of the four respective compression stages: `GMC` for the global translation, `ClusterTransform` for the cluster transformations, `ClusterWeights` for the animation weights, and `PredictionReziduals` for the residual vectors. The final encoding and arithmetic compression is then performed by the `WriteData(stream)` method. As a result, this method returns a `CompressorStats` structure containing information about the compressed dataset.

4.3 Optimisation Method

The proposed method is implemented in the `PES` module. This module has two input ports for reading the current bitrate (`InRate`) and distortion (`InDistortion`) and a variable number of output ports, which depends on the number of optimised parameters. Settings of this module include semicolon-separated sets of the initial, maximum, minimum and step values for the parameters, a switch for choosing the desired criterion, the target value for the selected criterion, optimisation core selection, and the slope equality and target value tolerances.

An interface named `IOptimizer` is created to simplify the switching between various optimisation cores and speed up the implementation process of new ones. This interface defines two methods: `Init(...)`, which initialises the optimiser with the specified setting values, and `NextStep(currentParam)` to determine the parameter shifts for the next iteration. Three properties are defined for reading the current state of the optimisation: `Bitrate`, `Distortion`, and `Slopes`.

The `Execute()` method of the `PES` module contains the main optimisation loop. First, an instance of the selected optimisation core is created and initialised. Then, the iterative configuration refinement is started. In each iteration, an optimisation step is performed by invoking the `NextStep()` method. This method updates the properties of the core instance and determines the next parameter shifts. After that, the stop-condition is tested depending on the selected criterion. If this condition is not satisfied, the calculated shifts are used to improve the current parameter configuration and a next iteration is performed.

Four optimisation core classes were created:

- `PESOptimizerA`
  – linear slope approximation and optimisation of all parameters in each step
- `PESOptimizerB`

– linear slope approximation and optimisation of only one parameter in each step

- `PESOptimizerC`
  – exponential slope approximation and optimisation of all parameters in each step
- `PESOptimizerD`
  – an improved version of `PESOptimizerC` with additional means of recovering from unexpected situations, such as bad slope shapes

Each class implements the `IOptimizer` interface and its methods and properties. The construction of all these classes follows the same pattern. When the `NextStep()` method is invoked, it first calls the `EvalParam(p)` method for each parameter to retrieve information about the parameter curve. Based on the specified optimisation constraint, one of the step methods (`StepSlope()`, `StepRate()`, `StepDistortion()`, or `StepParameter()`) is then called to calculate the parameter shifts. The resulting shifts are then handed back to the optimisation module.

5

Experimental Results

5.1 Algorithm Comparison

In this section, we will compare the four algorithms described in this work: Dynapack, D3DMC, FAMC, and Coddyac. The comparison will be performed the KG and STED metrics, and the compression performance of the algorithms in both metrics will be evaluated. We will use three dynamic mesh datasets of different properties to evaluate the performance of the algorithms:

- **Dance** is a 201 frames long sequence of meshes with 7061 vertices. Its motion is based on a bone skinning model, which results in rigid or nearly rigid affine motion in parts of the mesh, and a very limited number of principal vertex trajectories.
- **Chicken Crossing** sequence consists of 3030 vertices and 400 frames. It comprises 41 connected components. This may degrade the performance of algorithms based on a mesh traversal approach. The animation combines sequences of rapid movement with sequences of almost rigid motion.
- **Cowheavy** is a simulation of an interaction with a rubber cow model. It represents a general, difficultly skinnable, motion characteristic, which may be cause problems when using algorithms based on motion segmentation. It contains 2904 vertices in 204 frames.

Figures 5.1–5.6 show the RD curves of optimally configured algorithms. As you can see, the Coddyac algorithm outperforms the others on all the tested datasets and both metrics. It is followed by D3DMC and last comes Dynapack. Moreover, the gap between Coddyac and D3DMC is larger when the STED metric is used. This might indicate that the distortion caused by the Coddyac algorithm is less disturbing to a human viewer than the distortion caused by D3DMC. Note that we have only included the results of the FAMC algorithm in the first experiment, since it clearly produces results worse than the reference values (figure 5.2).

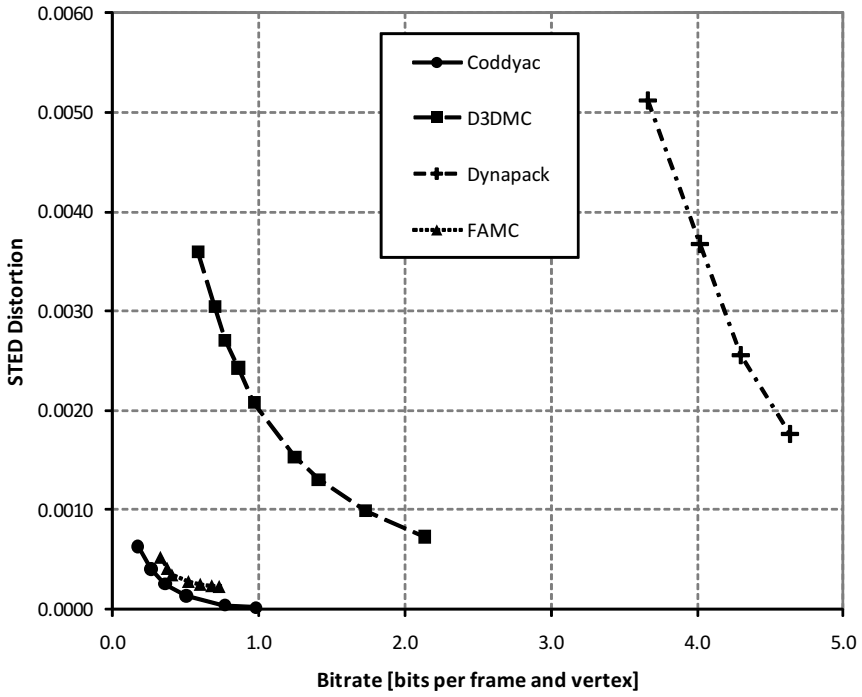

Figure 5.1: RD curves of the Dance sequence measured using the STED metric.

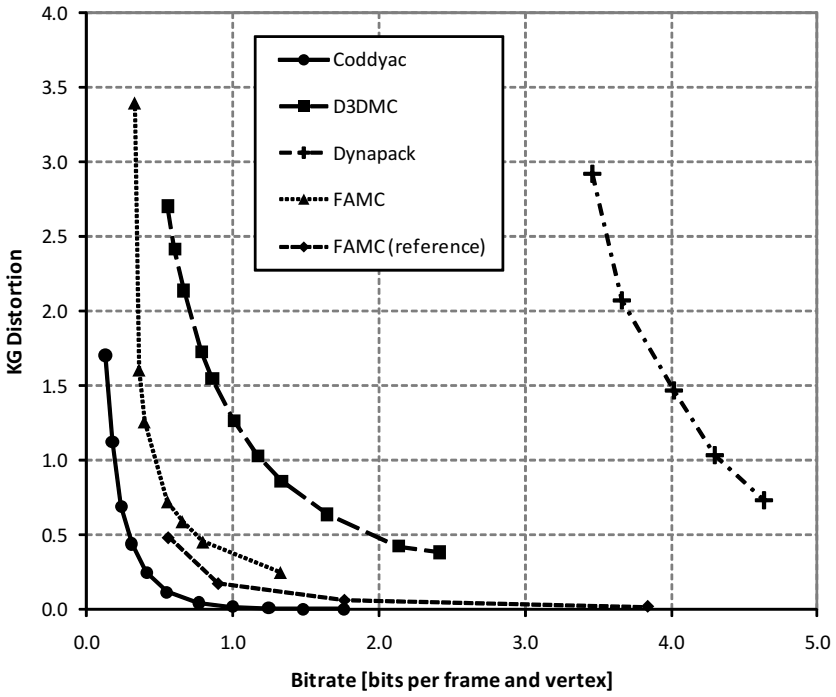

Figure 5.2: RD curves of the Dance sequence measured using the KG metric.

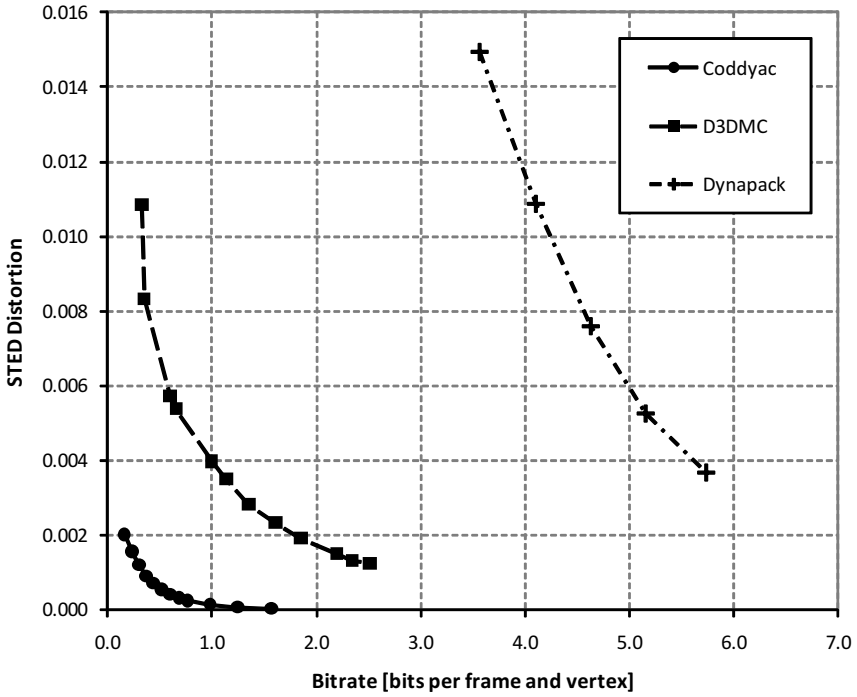

Figure 5.3: RD curves of the Chicken Crossing sequence measured using the STED metric.

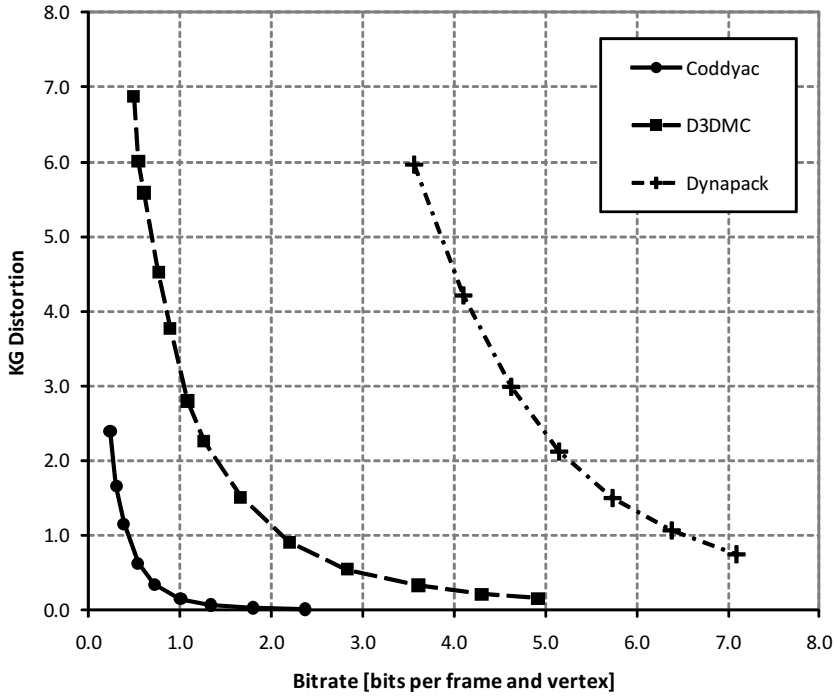

Figure 5.4: RD curves of the Chicken Crossing sequence measured using the KG metric.

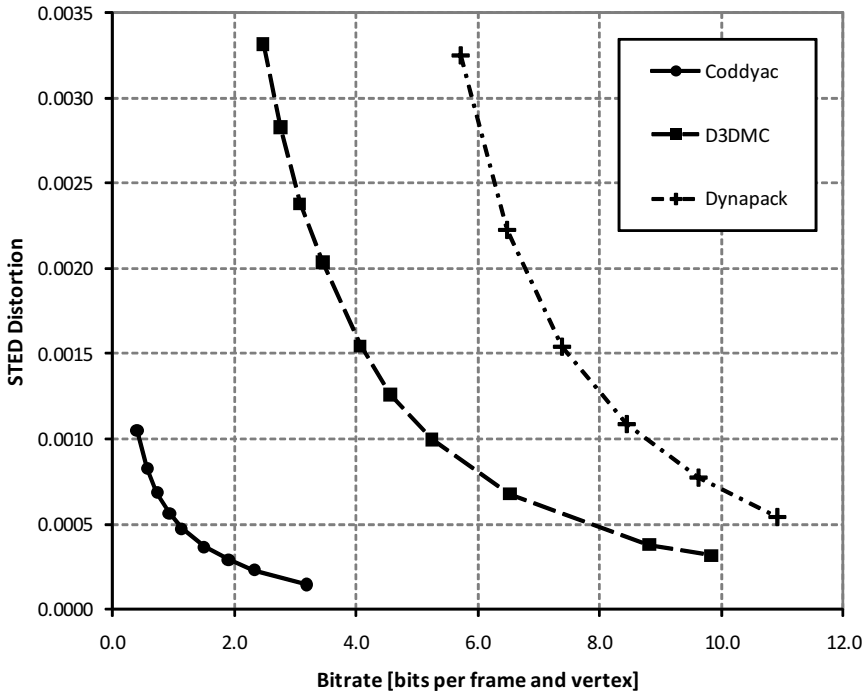

Figure 5.5: RD curves of the Cowheavy sequence measured using the STED metric.

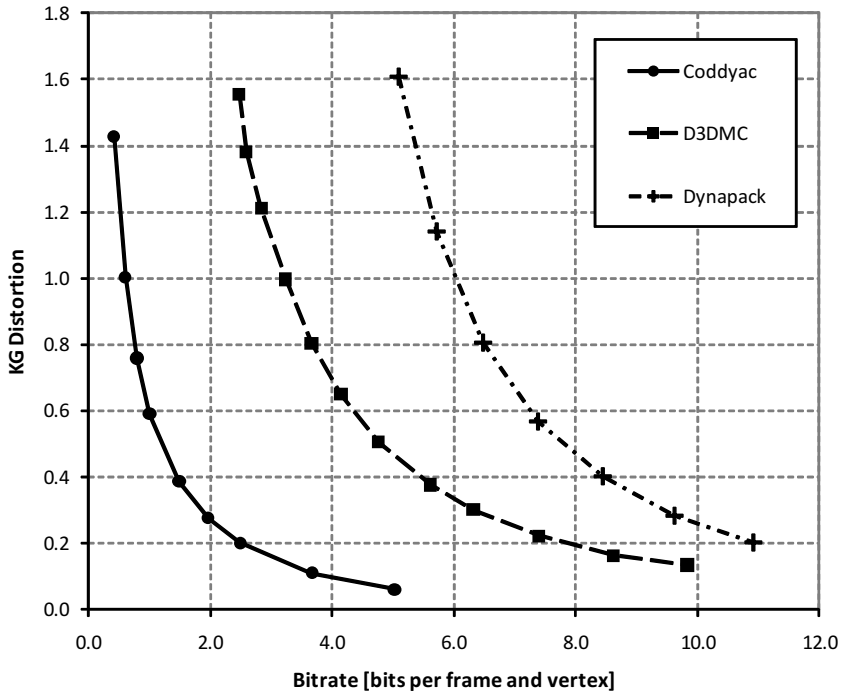

Figure 5.6: RD curves of the Cowheavy sequence measured using the KG metric.

5.2 Metric Comparison

We will now discuss the influence of the distortion metric on the optimal parameter configurations of the compression algorithms. If we compare the optimal configurations of any of the algorithms between the two distortion measurement methods, we may note that they contain different parameter values. Tables 5.1 and 5.2 show such differences for Coddyac and D3DMC algorithms between the metrics. For Coddyac, there are large differences in the prediction residue quantisation settings. At similar bitrates, STED prefers finer quantisation of the residues than KG, at the expense of decreasing the number of basis vectors and slightly coarsening the quantisation of basis vectors. Results of D3DMC show that again, in comparison with the KG metric, finer quantisation settings are selected with STED, while allowing for higher motion compensation error to reach a similar bitrate. Even our implementation of FAMC, which does not perform according to the reference results, shows changes in parameter values between the measurement methods.

STED				KG			
Basis Size	**Basis Q**	**Res. Q**	**Bitrate**	**Basis Size**	**Basis Q**	**Res. Q**	**Bitrate**
20	13.96	1.82	0.40	30	15.17	-0.30	0.41
30	14.95	2.34	0.57	40	16.04	0.41	0.60
40	15.76	2.63	0.73	50	16.66	0.92	0.79
50	16.47	3.08	0.93	60	17.18	1.38	1.00
80	17.70	3.84	1.51	80	18.00	2.18	1.49
100	18.14	4.24	1.90	100	18.56	2.75	1.96
120	18.68	4.62	2.33	120	19.03	3.27	2.49
160	19.53	5.22	3.19	160	20.02	4.20	3.67

Table 5.1: Optimal parameter values for the Cowheavy animation compressed using the Coddyac algorithm in the STED and KG metrics.

5.3 RD Optimisation Method

We compared our optimisation algorithm to the commonly used exhaustive search approach, which is considered the state of the art. In the following part, we show the results of finding the optimal envelope curve on the Cowheavy animation using Coddyac and D3DMC com-

STED			KG		
Motion Q	**Comp. Thr.**	**Bitrate**	**Motion Q**	**Comp. Thr.**	**Bitrate**
8.0	3.25	0.60	8.5	3.75	0.59
8.0	2.75	0.67	9.0	3.00	0.70
8.5	2.25	0.79	9.0	2.50	0.77
8.5	2.00	0.86	8.5	2.00	0.86
8.5	1.50	1.00	9.0	1.75	0.97
9.0	1.25	1.17	9.5	1.25	1.24
9.0	1.00	1.33	9.5	1.00	1.41
9.5	0.75	1.65	10.0	0.75	1.73
10.0	0.50	2.13	10.0	0.50	2.13
11.0	0.50	2.42	11.5	0.50	2.57

Table 5.2: Optimal parameter values for the Dance animation compressed using the D3DMC algorithm in the STED and KG metrics.

STED					KG				
Clusters	**Trans. Q**	**Weight Q**	**Res. Q**	**Bitrate**	**Clusters**	**Trans. Q**	**Weight Q**	**Res. Q**	**Bitrate**
10	12	10	0.0	0.33	10	12	10	0.0	0.33
10	12	12	0.0	0.38	10	14	10	0.0	0.36
10	14	12	0.0	0.41	10	16	10	0.0	0.40
12	14	14	0.0	0.52	18	16	10	0.0	0.56
18	14	14	0.0	0.60	18	18	10	0.5	0.66
18	16	14	0.0	0.68	18	18	10	1.0	0.80
18	16	16	0.0	0.73	18	18	10	2.0	1.32

Table 5.3: Optimal parameter values for the Dance animation compressed using the FAMC algorithm in the STED and KG metrics.

pression algorithms. For Coddyac, the fixed parameter criterion was used in our method with 15 different numbers of basis vectors. The same 15 values were set in the brute-force search together with 15 basis quantisation and 20 residual quantisation settings producing a total of 4500 configurations. With D3DMC, the optimiser was run the same way with 19 octree split threshold values, and 51 motion quantisation settings were used for the brute-force search (969 configurations in total).

The resulting rate-distortion charts are shown in figures 5.7 and 5.8 and the results are summarised in table 5.4. For the Coddyac compressor with three input parameters, the proposed algorithm is significantly faster than the exhaustive search, while for D3DMC with only two parameters, the performance is roughly equal. This shows a decrease in complexity related to the number of parameters compared to the brute-force approach. With more than three optimised parameters, our method would probably be even more efficient. Note that most configurations evaluated during the brute-force search produce higher than optimal bitrates resulting in longer encoding times. Thus, the run-time difference between the algorithms is greater than the difference in the number of compressor runs.

	Coddyac		**D3DMC**	
	Compressor Runs	Run Time	Compressor Runs	Run Time
brute-force	4500	36:06:35.4	969	9:44:33.5
our method	612	1:19:03.7	859	8:02:42.6
speedup	**7.4**	**27.4**	**1.1**	**1.2**

Table 5.4: Construction of a curve of optimal configurations for the Cowheavy animation using 15 different numbers of basis vectors for Coddyac and 19 different split thresholds for D3DMC.

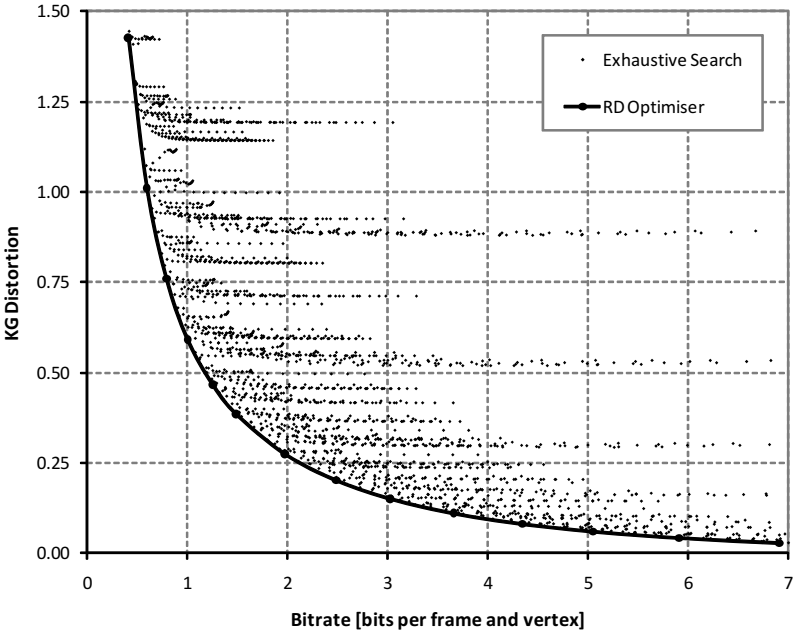

Figure 5.7: The results of the optimisation method compared to the configurations evaluated by the exhaustive search for the Cow animation compressed by the Coddyac algorithm.

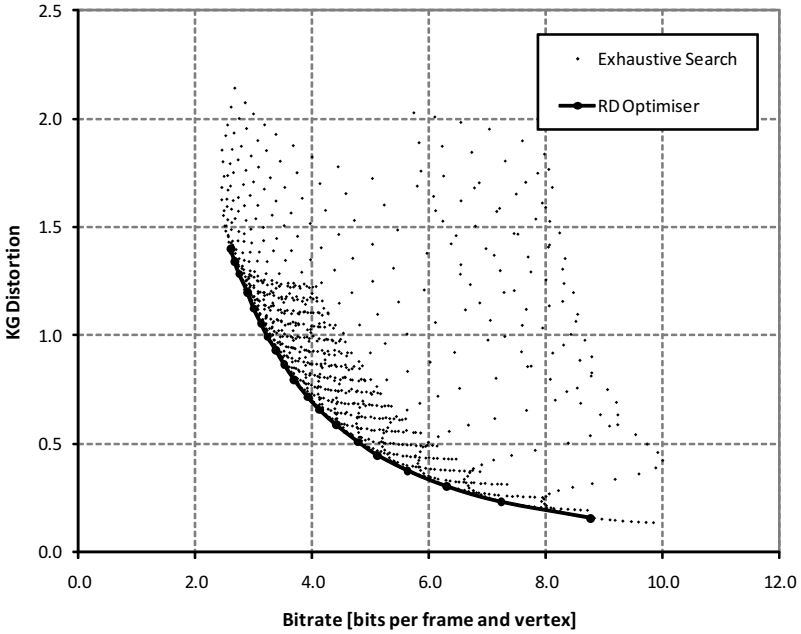

Figure 5.8: The results of the optimisation method compared to the configurations evaluated by the exhaustive search for the Cow animation compressed by the D3DMC algorithm.

6

Conclusions and Future Work

This work focused on rate-distortion optimisation in the area of compression of dynamic triangle meshes with constant connectivity. We have presented four compression algorithms, which are considered the state of the art. Each of these algorithms was described in detail, its configuration parameters were introduced and their influence on the compressed data was explained. The performance of the algorithms was then discussed, potential problems were identified, and suggestions were made to eliminate these problems.

We have introduced the problem of rate-distortion optimisation and discussed it thoroughly. Subsequently, we have proposed a novel method of rate-distortion optimisation of dynamic mesh compression algorithms, based on the Principle of Equal Slopes. For compressors with more than two input parameters, our method is able to find optimal configurations faster than the current state of the art. It reaches significant speedups with the Coddyac algorithm. However, experiments show that this optimisation approach might not be suitable for compression algorithms with compression stages compensating for the distortion caused by the preceding stages, which is the case of FAMC.

In the scope of this work, we have reimplemented two of the covered compression algorithms, FAMC and D3DMC, and implemented the proposed optimisation method. We were able to reach the published results with our implementation of the D3DMC algorithm, but the FAMC implementation produces worse results than those published by its authors. The implementation was carried out in the form of modules for the Modular Visualization Environment 2, which enables us to easily connect our optimiser to various modules for dynamic mesh compression and distortion evaluation.

A comparison of the rate-distortion performance of the described compression algorithms was presented. Their performance was evaluated using the KG and STED distortion measures. We

have found out, that the order of the algorithms according to their performance stays constant in all examined cases between the two metrics. However, the performance ratios between the algorithms have changed, which suggests that the choice of the distortion measure might influence the order. Moreover, the optimal configurations differ significantly between the used metrics. The results reveal that, in comparison with the KG method, the STED metric prefers finer quantisation settings at the expense of other parameters. That is a logical conclusion, as its design focuses on human distortion perception, which has been shown to be more sensitive to fine noise, such as the quantisation one, than to smooth deformations of much higher magnitudes [7, 19].

The possible course of future work is to focus on improving the optimisation method to make it applicable to FAMC and algorithms of similar nature. An entropy-distortion optimisation approach might be employed, where the entropy of the residues is measured and used instead of bitrate. Another room for improvement can be found in the robustness of the optimisation algorithm, since the parameter curves often contain noise and deformations. While the current implementation contains some simple measures of dealing with such situations, a more sophisticated approach would certainly improve the performance of the method.

References

[1] CALDERBANK, A. R., DAUBECHIES, I., SWELDENS, W., AND YEO, B. Wavelet transforms that map integers to integers. *Applied and Computational Harmonic Analysis 5*, 3 (1998), 332–369.

[2] HOPPE, H. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), ACM, pp. 99–108.

[3] IBARRIA, L., LINDSTROM, P., ROSSIGNAC, J., AND SZYMCZAK, A. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Proceedings of Eurographics 2003* (2003), pp. 343–348.

[4] IBARRIA, L., AND ROSSIGNAC, J. Dynapack: space-time compression of the 3D animations of triangle meshes with fixed connectivity. In *SCA 2003: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), Eurographics Association, pp. 126–135.

[5] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 14496 Part 2: Visual*. International Organization for Standardization, 2001.

[6] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 14496 Part 16: Animation Framework eXtension (AFX), amendment 2: Frame-based Animated Mesh Compression (FAMC)*. International Organization for Standardization, 2009.

[7] KARNI, Z., AND GOTSMAN, C. Spectral compression of mesh geometry. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 279–286.

[8] KARNI, Z., AND GOTSMAN, C. Compression of soft-body animation sequences. *Computers and Graphics 28* (2004), 25–34.

[9] KAVAN, L., MCDONNELL, R., DOBBYN, S., ŽÁRA, J., AND O'SULLIVAN, C. Skinning arbitrary deformations. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 53–60.

[10] KAVAN, L., SLOAN, P.-P., AND O'SULLIVAN, C. Fast and efficient skinning of animated meshes. *Computer Graphics Forum 29*, 2 (2010), 327–336.

[11] KIRCHHOFFER, H., MARPE, D., MÜLLER, K., AND WIEGAND, T. Context-adaptive binary arithmetic coding for frame-based animated mesh compression. In *ICME 2008: 2008 IEEE International Conference on Multimedia and Expo* (2008), pp. 341–344.

[12] MAMOU, K., ZAHARIA, T., AND PRÊTEUX, F. A skinning approach for dynamic 3D mesh compression. *Computer Animation and Virtual Worlds 17*, 3-4 (2006), 337–346.

[13] MAMOU, K., ZAHARIA, T., PRÊTEUX, F., STEFANOSKI, N., AND OSTERMANN, J. Frame-based compression of animated meshes in MPEG-4. In *ICME 2008: 2008 IEEE International Conference on Multimedia and Expo* (June 2008), pp. 1121–1124.

[14] MARPE, D., SCHWARZ, H., AND WIEGAND, T. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology 13*, 7 (july 2003), 620–636.

[15] MÜLLER, K., SMOLIC, A., KAUTZNER, M., EISERT, P., AND WIEGAND, T. Predictive compression of dynamic 3D meshes. In *ICIP 2005: 2005 IEEE International Conference on Image Processing* (2005).

[16] ROSSIGNAC, J. EdgeBreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5* (1998), 47–61.

[17] ROSSIGNAC, J. 3D compression made simple: Edgebreaker with zipandWrap on a corner-table. In *SMI 2001: 2001 International Conference on Shape Modeling and Applications* (may 2001), pp. 278–283.

[18] SMITH, L. I. *A tutorial on Principal Component Analysis*. University of Otago, Dunedin, New Zealand, 2002. Lecture Notes for COSC435: Computer Vision.

[19] SORKINE, O., COHEN-OR, D., AND TOLEDO, S. High-pass quantization for mesh encoding. In *SGP 2003: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (2003), Eurographics Association, pp. 42–51.

[20] STEFANOSKI, N., XIAOLIANG, L., KLIE, P., AND OSTERMANN, J. Scalable linear predictive coding of time-consistent 3D mesh sequences. In *3DTV Conference, 2007* (May 2007), pp. 1–4.

[21] TAUBIN, G., AND ROSSIGNAC, J. Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2 (1998), 84–115.

[22] TOUMA, C., AND GOTSMAN, C. Triangle mesh compression. In *Graphics Interface* (June 1998), pp. 26–34.

[23] UNIVERSITY OF WEST BOHEMIA. Modular visualisation environment 2. Online resource: http://herakles.zcu.cz/research/projects/11/index.php, April 2010.

[24] VÁŠA, L., AND SKALA, V. CoDDyAC: Connectivity driven dynamic mesh compression. In *3DTV-CON, The True Vision - Capture, Transmission and Display of 3D Video* (Kos, Greece, May 2007), IEEE Computer Society.

[25] VÁŠA, L., AND SKALA, V. Cobra: Compression of the basis for the PCA represented animations. *Computer Graphics Forum 28*, 6 (2009), 1529–1540.

[26] VÁŠA, L., AND SKALA, V. Geometry-driven local neighbourhood based predictors for dynamic mesh compression. *To appear in Computer Graphics Forum* (2010).

[27] VÁŠA, L., AND SKALA, V. A perception correlated comparison method for dynamic meshes. *To appear in IEEE Transactions on Visualization and Computer Graphics* (2010).

[28] ZHANG, J., AND OWEN, C. B. Octree-based animated geometry compression. In *DCC '04: Proceedings of the Conference on Data Compression* (Washington, DC, USA, 2004), IEEE Computer Society, p. 508.