Geometry Compression of Triangle Meshes Using a Reference Shape

Eliška Mourycová[a] and Libor Váša[b]

Department of computer science and engineering, University of West Bohemia, Univerzitní 8, Plzeň, Czech Republic
{emouryc, lvasa}@kiv.zcu.cz

Keywords: Compression, Triangle Mesh, Encoding, Geometry

Abstract: Triangle mesh compression is an established area, however, some of its special cases are yet to be investigated. This paper deals with lossy geometry compression of manifold triangle meshes based on the EdgeBreaker algorithm using a reference shape known to both the encoder and the decoder. It is assumed that the shape of the reference object is similar to the shape of the mesh to be encoded. The predictions of vertices positions are done extrinsically, i.e. outside the reference shape, and then orthogonally projected onto its surface. The corrections are encoded by two integer numbers, denoting the layer order and an index of a hexagon in a hexagonal grid generated on the surface of the reference shape centered at the prediction point. The availability of a reference mesh results in a smaller bitrate needed for comparable error when compared to a state of the art static mesh compression algorithm using weighted parallelogram prediction.

1 INTRODUCTION

While triangle mesh compression is a mature field with numerous applications, there are still certain special cases that remain unexplored. One particular open question is how to efficiently exploit a shape reference that is available at both the encoder and the decoder. This scenario occurs in practical applications, such as quality check scanning or compression of time-varying meshes.

In quality check scanning, certain item is being produced and 3D scanned at the end of a production line in order to verify its quality. Apart from direct analysis, the scans are commonly saved for further processing/reference, which may be costly with high volume production. At the same time, most of the scans represent a very similar shape, and in most practical cases, even a perfect shape reference is available in the form of a CAD model of the produced item.

A similar situation occurs when encoding a sequence of meshes representing an animation, i.e. a continuous deformation of a certain shape. When animations are created artificially, the frames usually share connectivity (dynamic mesh), and such data can be compressed very efficiently even when the animation structure (bone rig or similar) is unknown (Chen et al., 2018). When scanning a real world dynamic scene, on the other hand, a series of meshes with varying connectivity (time-varying mesh) is often obtained, making the need for efficient storage more acute, yet at the same time making the actual compression much more difficult. One way to deal with the problem is to compress the frames sequentially, using each previous frame (possibly warped in a certain way) as a reference for compression of each following frame.

The problem at hand is therefore as follows: a coder and a decoder share a shape, represented as a triangle mesh, denoted *reference mesh*. The task is to transmit another shape, again represented by a triangle mesh, from the encoder to the decoder. This *input mesh* has a shape that is very similar to the reference shape, however, it has a completely different tessellation, and possibly even different topology (genus). The objective is to encode the input mesh using as few bits as possible, using the shared knowledge of the actual shape that is being transmitted.

At first sight, this may look like a low hanging fruit: information that is available at a decoder can be omitted from the transmission, providing an improved compression performance. It may even seem that the decoder already has all the information it needs, since it has the shape available, however, additional information is certainly needed, since the mesh representation captures not only the shape of the model, but also its sampling, i.e. tessellation, since in our scenario, we wish to preserve the connectivity of the input mesh. How much of the bitrate commonly used

[a] https://orcid.org/0000-0001-9379-4097
[b] https://orcid.org/0000-0002-0213-3769

for encoding a triangle mesh is spent on the actual shape, and how much is spent on capturing the particular tessellation, is generally not known, and attempting to fruitfully exploit the shape reference in order to reduce the data rate with respect to no-reference encoder turns out to be a surprisingly difficult task.

We present an algorithm based on traversing the input mesh and predicting vertex positions one by one. In order to make the prediction, we use the reference mesh. Our algorithm works with projections of predicted and encoded vertices onto the surface of the reference mesh. The difference between the prediction and the actual position (also known as *correction*) is encoded *intrinsically*, restricting the possible locations to the 2D reference surface and, most importantly, using only two coordinates. Finally, rather than using a rectangular grid in order to quantize the coordinates, we use a hexagonal grid that has better properties in terms of quantization error.

The rest of the paper is structured as follows: Section 3 describes the overall process of encoding a mesh, including requirements imposed on the input shapes and steps taken to preprocess the data. Section 4 then in detail describes three relatively independent modules used for the assembly of the encoding algorithm. Section 5 is devoted to the evaluation of the performance of the proposed method and its comparison with an alternative static mesh encoder.

2 RELATED WORK

Compression of polygonal meshes, and of triangle meshes in particular, is a field that has been actively studied for several decades. The problem can be further split to compression of connectivity, which is always understood as lossless, and compression of geometry (vertex positions), where mostly lossy algorithms are employed, sacrificing reconstruction precision in order to achieve a more efficient compression.

For connectivity compression, it is known that assuming that every possible triangulation is equally probable, at least 3.245 bits per vertex (bpv) are needed in the limit for genus 0 triangle meshes(Tutte, 1962). A guarantee of 4 bpv is provided by the EdgeBreaker algorithm(Rossignac, 1999), which can be further improved by employing a more efficient entropy coding. Further improvement is achieved by valence based encoders(Alliez and Desbrun, 2001), assuming that regular connectivities with vertex valences close to 6 are more probable than others, reaching data rates of 1-2 bpv for common datasets.

For geometry compression, the most common approach that complements the EdgeBreaker connectivity coder well is the parallelogram prediction(Touma and Gotsman, 1998). Whenever a new vertex is encountered during the EdgeBreaker traversal, its position is predicted by forming a parallelogram from a known neighbouring triangle. Next, rather than encoding the quantized coordinate, only a correction vector which represents the difference between the actual and predicted position is stored reaching a lower entropy and thus a lower bitrate.

This approach has been further improved by encoding the geometry in a separate pass, when the full connectivity is known to both the encoder and the decoder. This allows adjusting the shape of the parallelogram stencil according to the degrees of vertices involved in the prediction(Váša and Brunnett, 2013).

Other approaches to geometry encoding have been proposed as well, building on concepts such as expressing the geometry in delta coordinates (Sorkine et al., 2003), known as high-pass coding (HPC) or expressing the shape in the frequency domain(Valette and Prost, 2004). These often lead to different character of introduced distortion, targeting at perceptual quality metrics(Corsini et al., 2013). Recently, a modification of the HPC has been proposed, which allows achieving competitive results in terms of both traditional error metrics, such as mean squared error or Hausdorff distance, as well as perceptual metrics(Váša and Dvořák, 2018).

Finally, a range of algorithms has been proposed aiming at various particular desirable properties of mesh transmission, such as the possibility of partial decoding(Hoppe, 1996), encoding of mesh sequences of shared connectivity(Chen et al., 2018) or joint encoding of meshes with color or texture information(Caillaud et al., 2016). Our paper fits into this last category, focusing on a special case scenario when a reference mesh is available.

3 ALGORITHM DESCRIPTION

This section describes the steps to encode (compress) the input mesh. The algorithm works under the assumption that both the encoder and the decoder have the same reference mesh, whose shape is similar to that of the input mesh. However, the sampling of the surfaces can be completely different.

3.1 Input Data

There are certain conditions that both the original and the reference mesh must meet, in particular:

- both meshes must be manifold,

- all triangles in both meshes must be equally oriented, i.e. all clockwise or all counterclockwise,
- the meshes cannot contain degenerate triangles, i.e. triangles with zero area.

3.2 Preprocessing

Before the encoding algorithm is launched, two preprocessing steps are done - a Bounding Volume Hierarchy (BVH) tree for the reference mesh (as described in section 4.2.4) is built, and the neighbors for each triangle in both the original and the reference mesh are found and stored.

The triangles' neighbors are stored in a hash table, where the keys are oriented edges of triangles, i.e. such structures which keep the index of the starting vertex and the index of the end vertex. The hash table is filled by iterating through all triangles of the mesh and adding all three edges as keys with the same value - the current triangle.

3.3 Encoding

To start encoding the mesh, the first triangle of the original mesh is projected onto the surface of the reference mesh (using orthogonal projection described in section 4.2.4). The vertices of the projected triangle are updated to match the projected positions.

The encoding algorithm then follows the steps of the EdgeBreaker algorithm. If the code "C" was to be encoded during the EdgeBreaker algorithm, i.e. the next vertex behind a prediction gate has not been conquered, then the encoder does the following steps:

1. project the tip vertex from the input mesh onto the reference mesh,
2. evaluate the prediction (see subsection 4.1 for details on how predictions are made),
3. construct the correction by generating a hexagonal grid on the reference surface and finding the hexagon centre that lies closest to the projected tip,
4. replace the tip vertex position in the input mesh by the nearest hexagon centre (in order to keep the encoder in sync with the decoder for following predictions),
5. save the correction (i.e. identification of the nearest hexagon) into the data stream - two values are needed, the layer order and the index in the layer, as will be described in more detail later.

The connectivity of the encoded mesh stays the same as in the input mesh.

There are two sources of precision loss in the procedure: first, the difference between the actual vertex position and its projection onto the reference mesh is neglected and not rectified in the decoder. This can be fixed using an additional correction layer, however, we choose not to include such correction in order to evaluate the algorithm assuming that this error is negligible. The other source of distortion is the quantization by the hexagon grid on the reference surface. This error can be controlled by adjusting the hexagon edge length.

4 MODULES

This section describes in detail parts of the algorithm which are then used for the mesh compression.

4.1 Prediction of the Vertex Position

The predictions are made using an *extrinsic parallelogram prediction* (see Fig. 1).

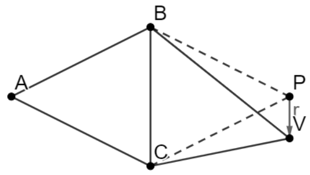


Figure 1: Paralellogram prediction.

In Fig. 1, point A is a vertex known to both encoder and decoder, points B and C form the current gate in the EdgeBreaker procedure, point P is the predicted point, point V is the actual (projected in step 1.) position of the point and vector $\vec{r}$ is the correction vector (encoded by the hexagon layer and index).

Point P is acquired as $P = B + C - A$. Such point does not always lie on the surface of the reference mesh, therefore it is projected onto it - hence the term extrinsic prediction.

4.2 Orthogonal Projection of Points onto the Surface

Another required part of the solution is a method capable of orthogonally projecting points onto the surface of a mesh. This is used for projecting vertices of the original mesh onto the surface of the reference mesh and acquiring projections of the predictions of the vertices of the input mesh.
To avoid a brute force approach, i.e. iterating through all of the triangles of the reference mesh and checking which is the closest one to the given query point, a BVH tree is constructed.

4.2.1 Building a BVH Tree

The tree is built as a binary tree. Each node of the tree holds a list of triangles it contains, references to its parent, children and their bounding box. The construction steps are:

1. Create a queue of tree nodes and enqueue the root node
   - The root node contains all triangles in the mesh, its bounding box is the same as that of the whole mesh and it has no parent.
2. If the queue is empty, then break
3. Dequeue a node into `currNode`
4. If `currNode` is not a leaf node, create its two children
   - See subsection 4.2.2 for details
5. Enqueue both children and go to 2.

4.2.2 Creating Children of a Tree Node

Children of a node are created by splitting the parent's bounding box along its longest side. The parent's list of triangles is sorted by the x, y or z position of the triangles' centroids (depending on which side of the bounding box was the longest). The first half of the sorted triangles is assigned to the first child and the second half to the second child. New (sub-)bounding boxes are calculated for both children. If the children's triangle count is less than or equal to 2, then a flag labeling them as leaf nodes is set to true for them.

4.2.3 Using the BVH Tree for Orthogonal Projection

Using the tree to locate the nearest triangle to a given query point is done as follows:

1. Start in the root node of the tree
2. Go to the child node whose bounding box centre is closer to the query point
3. When a leaf node is reached, distances to its triangles are evaluated and the minimum saved as d
   - See section 4.2.4 for details on projecting a point into a triangle
4. Traverse upwards from the leaf node and consider whether a different (unvisited) branch needs to be visited
   - A branch does not need to be visited if the query point lies outside the bounding box, further than d from at least one of its faces
5. If a leaf node is reached again, d is updated if a closer triangle is found
6. Continue until traversing leads back to the root node
7. Return the closest found triangle (with distance d from the query point)

4.2.4 Projection of a Point into a Triangle

Projecting the query point into a triangle is done by first projecting the point onto the plane of the triangle:

$$\vec{u} = V_1 - V_0, \vec{v} = V_2 - V_0$$
$$\vec{n} = \vec{u} \times \vec{v}$$
$$\vec{w} = Q - V_0$$

$$\gamma = \frac{\vec{u} \times \vec{w} \cdot \vec{n}}{\vec{n} \cdot \vec{n}}$$
$$\beta = \frac{\vec{w} \times \vec{v} \cdot \vec{n}}{\vec{n} \cdot \vec{n}}$$
$$\alpha = 1 - \gamma - \beta,$$

where V_0, V_1 and V_2 are the vertices of the triangle and Q is the query point to be projected. γ, β, α are then barycentric coordinates of the projected point with respect to the given triangle. If

$$\beta < 0 \ \vee \ \gamma < 0 \ \vee \beta + \gamma > 1, \tag{1}$$

then the projected point is outside of the triangle. In this case we project the point onto the edges:

$$\vec{ev} = V_i - V_{(i+1)\%3}$$
$$\vec{vv} = P - V_i$$
$$C = P - V_i + \frac{\vec{ev} \cdot \vec{vv}}{\vec{ev} \cdot \vec{ev}} \vec{ev},$$

where V_i is one of the vertices of the triangle, P is the point projected onto the plane given by the triangle and C is the point projected onto a line given by one of the edges.

If C lies between the two vertices of the triangle which defined the line, then the distance between P and C is computed and saved as d. d is updated if a smaller distance is found.

After all edges are checked, distances to the triangles vertices are computed individually and d is updated if necessary. A point on the triangle with found minimal distance d is finally returned.

4.3 Straight Walk on the Surface

One of the needed parts of the algorithm is a method for moving along a geodesic line on the surface of a mesh from a starting point in a given direction, until a desired distance is reached. This part of the solution is used for the hexagonal grid generation on the surface of the reference mesh.

The starting point of the walk is described in barycentric coordinates. For storing the barycentric coordinates, we only need two values. The conversion from barycentric to cartesian coordinates is done using the following formulas:

$$
\begin{aligned}
cPt^x &= V_0^x + bPt^x(V_1^x - V_0^x) + bPt^y(V_2^x - V_0^x)\\
cPt^y &= V_0^y + bPt^x(V_1^y - V_0^y) + bPt^y(V_2^y - V_0^y)\\
cPt^z &= V_0^z + bPt^x(V_1^z - V_0^z) + bPt^y(V_2^z - V_0^z),
\end{aligned}
$$

where cPt is a 3D point in cartesian coordinates, bPt is the barycentric point to convert to cartesian coordinates and V_0, V_1, V_2 are the vertices of the triangle in which the barycentric point lies.

4.3.1 Determining the Walking Direction

First, a direction in which to walk is determined. Either a vector is specified, in which case this vector is used as the direction. If none is given, then a default reference direction is chosen as a vector going from the starting point to the first vertex of the starting triangle. If such vector is a zero vector, then a vector going from the starting point to the second vertex of the starting triangle is chosen instead.
This algorithm computes the direction in the coordinate system of the triangle in which it is currently operating (analogically to barycentric coordinates of a point).

For conversion from triangle to cartesian direction coordinates, following formulas are used:

$$
\begin{aligned}
cDir^x &= tDir^x(V_1^x - V_0^x) + tDir^y(V_2^x - V_0^x)\\
cDir^y &= tDir^x(V_1^y - V_0^y) + tDir^y(V_2^y - V_0^y)\\
cDir^z &= tDir^x(V_1^z - V_0^z) + tDir^y(V_2^z - V_0^z),
\end{aligned}
$$

where $cDir$ is a direction (vector) in cartesian coordinates, $tDir$ is the direction in the triangle coordinate system to convert to cartesian coordinates and V_0, V_1, V_2 are the vertices of the triangle in which the direction is specified.

4.3.2 Finding the Intersected Edge

Next, an edge which will be intersected by walking straight to the border of the current triangle is found. This is done by calculating signed distances to all the edges using the triangle coordinate system direction.

The distances are computed as follows:

$$
d_1 = \begin{cases} \frac{-bPt^y}{tDir^y}, & \text{if } tDir^y \neq 0 \\ -1, & \text{otherwise} \end{cases}
$$

$$
d_2 = \begin{cases} \frac{1-bPt^x-bPt^y}{tDir^x+tDir^y}, & \text{if } tDir^x + tDir^y \neq 0 \\ -1, & \text{otherwise} \end{cases}
$$

$$
d_3 = \begin{cases} \frac{-bPt^x}{tDir^x}, & \text{if } tDir^x \neq 0 \\ -1, & \text{otherwise,} \end{cases}
$$

where d_1 is the signed distance to the edge between vertices V_0 and V_1, d_2 is the signed distance to the edge between vertices V_0 and V_2 and d_3 is the signed distance to the edge between vertices V_2 and V_1.

The smallest positive distance is found and the edge of this distance is identified as the edge to be intersected. If no edge was found, we check whether a vertex was hit by walking.

A barycentric point lies on a vertex of a triangle if

$$
\begin{aligned}
bPt^x = 0 \ \wedge \ bPt^y = 0 \ \vee\\
bPt^x = 1 \ \wedge \ bPt^y = 0 \ \vee\\
bPt^x = 0 \ \wedge \ bPt^y = 1
\end{aligned}
$$

Next, we move towards the found edge or vertex. We check if the travelled distance is greater or equal to the desired distance. If it is, we calculate how much farther we have walked compared to the desired distance (because we end each iteration on an edge or vertex) and we move back by the calculated difference vector to the result point.

Otherwise, if not enough distance was travelled, we rotate the walking direction (vector) over the intersected edge or vertex.

4.3.3 Rotating the Vector

Rotating the walking direction when an edge to rotate over was found is done as follows: The vector is rotated over the edge using the Rodrigues' rotation formula in a form allowing rotating a vector about an axis by a given angle (Liang, 2018).

$$
v_{rot} = \vec{v}cos(\theta) + (\vec{n} \times \vec{v})sin(\theta) + \vec{n}(\vec{n} \cdot \vec{v})(1 - cos(\theta)),
$$

where $\vec{v}$ is a vector in $\mathbb{R}^3$, $\vec{n}$ is a unit vector describing an axis of rotation about which $\vec{v}$ rotates by an angle θ and $\vec{v_{rot}}$ is the rotated vector $\vec{v}$.

We select $\vec{n}$ as the cross product of the current and the neighbouring triangle's (the one that shares the intersected edge with the current triangle) normals'. We don't calculate the angle θ directly, we consider the sine of the desired rotation angle to be the length of $\vec{n}$ (before it is normalized) and the cosine of the angle to be the dot product between the current and the neighbouring triangle's normals.

Rotating the walking direction when a vertex was hit is done as follows:

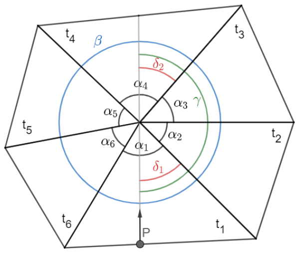


Figure 2: Illustration of angle computation after vertex hit.

Fig. 2 illustrates the calculation of the angle to go round a vertex to continue the walk after a vertex was hit. We set:

$$\beta = \sum_{i=0}^{n} \alpha_i, \gamma = \frac{\beta}{2}, \tag{2}$$

where n is the number of triangles incident with the hit vertex. In general $\beta \neq 2\pi$.

The steps of the algorithm are:

1. Calculate $\gamma = \frac{\sum_{i=0}^{n} \alpha_i}{2}$
2. Calculate δ_1
3. Go counterclockwise through the fan of triangles incident with the hit vertex until kth triangle such that $\delta_1 + \sum_{i=1}^{k} \alpha_i < \gamma < \delta_1 + \sum_{i=1}^{k+1} \alpha_i$
4. Calculate $\delta_2 = \gamma - \delta_1 - \sum_{i=1}^{k} \alpha_i$
5. Continue walking from the hit vertex to $(k+1)$th triangle at angle δ_2

After the rotation, we switch to the new triangle and continue with the next iteration.

4.3.4 Summary

To summarize, the straight walk algorithm consists of the following steps:

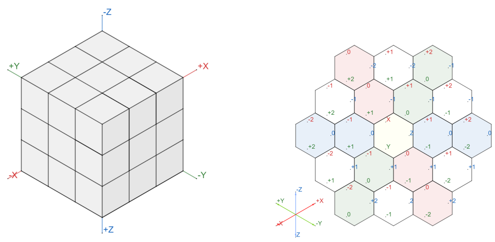
1. Get the initial direction in which to walk
2. Find the edge which will be intersected by the walk
3. Check if a vertex was hit
4. Check if desired distance was travelled
   - If (more than) enough distance travelled, backtrack and return result point. End.
5. Rotate the walk direction (vector) over the intersected edge or vertex
6. Switch to the next triangle
7. Go to 2.

4.4 Hexagonal Grid Generation on the Surface

Hexagonal grid is used for encoding the correction vector as two integer numbers - layer index and an index of a given hexagon within the specified layer. When a prediction is made, this part of the algorithm generates a hexagonal grid on the surface of the mesh (using the surface walking module described in section 4.3) from the prediction point and finds the hexagon, whose centre is closest to the correct point.

Note that rather than generating the actual hexagons, the algorithm only generates their centres by walking a certain distance in a certain direction from the prediction point. Layer index 0 is reserved for the hexagon generated centered at the prediction. The hexagons in each layer are indexed clockwise.

For determining the distance to travel to a given hexagon centre, *cube coordinates* are used (see Fig. 3). Cube coordinates assign a virtual 3D point to the centre of each hexagon. This 3D point can be understood as a vector going from point $[0,0,0]$ (i.e. the hexagon in layer 0) to said point. This vector is then multiplied by the user specified distance between two centres of two neighbouring hexagons.



(a) Cube (b) Cube coordinates for hexagonal grid

Figure 3: Cube coordinates for hexagonal grid.

The direction in which the centre of the first hexagon of layer 1 lies is selected as $\vec{v} = V_1 - V_0$, where V_0 is the first vertex of the triangle in which the grid generation starts and V_1 is the second vertex of said triangle.

The user must specify the distance between two centres of two neighbouring hexagons *hexSize*. From *hexSize* we can calculate the distance between the centre of a hex and its vertex and mark it as *centPeakDist*:

$$triHeight = \sqrt{(hexSize)^2 - (\frac{hexSize}{2})^2}$$

$$centPeakDist = (\frac{2 \cdot triHeight}{3}) \cdot \sqrt{2}.$$

The user also specifies the maximum number of layers to be generated (*maxLayers*).

The used approach for determining the closest hexagon to the end point works as follows:

1. Get the default direction vector as $\vec{v} = V_1 - V_0$
2. Calculate *centPeakDist*
3. Calculate distance from the starting point to the end point and save as d, set $i = 1$
4. Generate cube coordinates for hexagons in layer i
5. Generate hexagons in layer i. Use the cube coordinates to get the distance of each hexagon from the starting point (centre of the central hexagon)
   - For each hexagon in the current layer, check if its centre is closer to the end point than d, if it is, update d
   - If $d < centPeakDist$, return this hexagon as the closest one. End.
6. Increment the angle of direction by $60/i$
7. If $i > maxLayers$, return hexagon of distance d. End.
8. Increment i and go to 4.

5 EVALUATION

The experiments to verify the functionality of the algorithm were carried out on three different data sets (meshes), each with three different hexagon sizes and three different reference meshes. The results were compared to the performance of a reference implementation of the EdgeBreaker algorithm with weighted parallelogram prediction (Váša and Brunnett, 2013), so that similar Mean Square Error (MSE) was achieved and therefore the bitrates correspond to comparable errors.

See the original meshes in section 5.1. The different types of reference meshes will be presented in the following sections. Three different hexagon sizes were selected as follows:

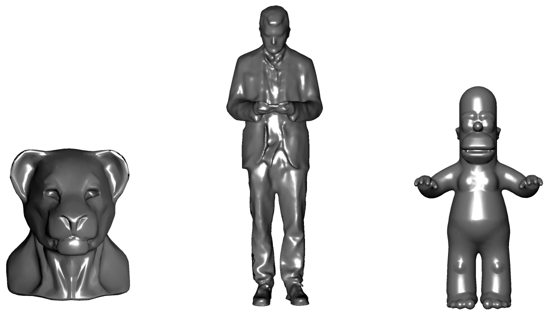
- $hexSize1 = length/1$
- $hexSize2 = length/2$
- $hexSize3 = length/3$,

where *length* is the length of the 1st edge of the 0th triangle of the input mesh and *hexSizeX* is the distance between two adjacent hexagons' centres.

Note: We will use this hexagon size designation in the following sections.

5.1 Input meshes

Fig. 4 shows the input meshes (meshes to be encoded) which were used for the experiments.



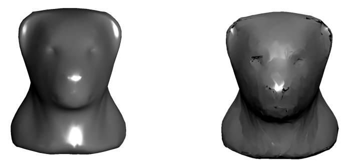
(a) Mesh 1: Lion (b) Mesh 2: Person (c) Mesh 3: Homer

Figure 4: Input meshes.

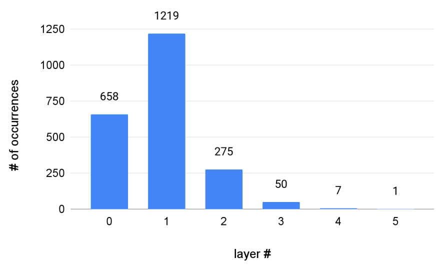
Note: This section contains charts showing the distribution of layer numbers and their occurrences. The layers (denoted on the x axis) are sorted by their number in ascending order. The maximum layers allowed to be generated for these experiments was 500. Only those layers which appeared at least once are visible in the chart.

For every encoded mesh, MSE was computed as well as bits per vertex (bpv) of the hexagon layers and indices lists using arithmetic coding, see subsection 5.2. Experiments on the input meshes were performed with unsuitable, good and very good reference meshes.

By unsuitable reference, we mean such a mesh whose surface is not very similar to the surface of the input mesh. These reference meshes were acquired by running ten iterations of HC Laplacian Smoothing on the input meshes using the `MeshLab` software. Fig. 5 shows a typical result achieved by encoding the Lion mesh using an unsuitable reference mesh.



(a) Reference mesh (b) Encoded mesh

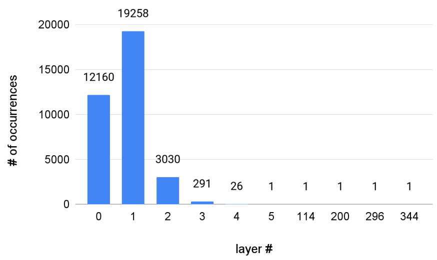


(c) Hexagon layers distribution

Figure 5: Lion mesh encoded using an unsuitable reference mesh and hexSize2.

(a) Reference mesh

(b) Encoded mesh

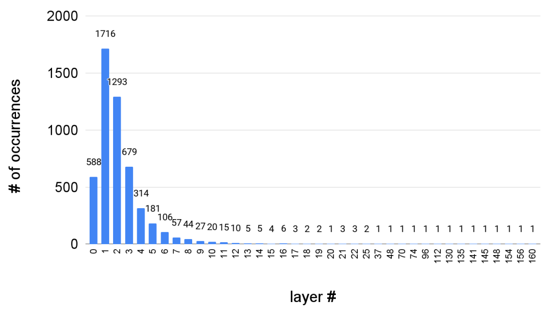


(c) Hexagon layers distribution

Figure 6: Person mesh encoded using a good reference mesh and hexSize2.

(a) Reference mesh

(b) Encoded mesh



(c) Hexagon layers distribution

Figure 7: Homer mesh encoded using a very good reference mesh and hexSize2.

Good reference meshes were acquired by running two iterations of HC Laplacian Smoothing on the original meshes using the `MeshLab` software. Fig. 6 shows a typical results achieved by encoding the Person mesh using a good reference mesh.

Very good reference meshes were acquired by running one iteration of Isotropic Explicit Remeshing on the original meshes using the `MeshLab` software. Fig. 7 shows a typical results achieved by encoding the Homer mesh using a very good reference mesh.

5.2 Mean Square Error and BPV Data

This subsection contains information about the calculated MSE and bpv for each mesh. The data is shown in tables 1, 2 and 3 and in Figures 8, 9 and 10. In the tables, the rows are coded **U**, **G** or **V** - these letters stand for **U**nsuitable, **G**ood or **V**ery good reference meshes. The numbers following the letters denote the *hexSize* used for the experiment, i.e. **1** stands for *hexSize*1, **2** for *hexSize*2 and **3** for *hexSize*3. The results were compared with the performance of the Edgbreaker algorithm, as shown in Table 4.

In the figures showing the comparison between the proposed method and the EdgeBreaker algorithm, the EdgeBreaker performance is color coded in blue. The points show the performance of the proposed method and are color coded as folows: Performances measured when using unsuitable reference meshes are displayed in red, good reference meshes in orange and very good reference meshes in green.

The shown BPV is needed for encoding only the geometry of the objects, the bitrate needed for encoding the connectivity is not taken into account, as it would be the same in both cases. The data shows that our method is able to outperform the state of the art algorithm, especially at very low bitrates, provided that a good or very good reference mesh is provided. At higher bitrates, the proposed algorithm results in a higher distortion, likely caused by projection on the reference surface.

	MSE	bpv layers	bpv indices	bpv total
U, 1	$1.51 \cdot 10^{-2}$	1.29	2.34	3.63
U, 2	$9.35 \cdot 10^{-3}$	1.64	2.96	4.60
U, 3	$8.32 \cdot 10^{-3}$	2.03	3.52	5.55
G, 1	$7.70 \cdot 10^{-3}$	1.34	2.40	3.74
G, 2	$2.37 \cdot 10^{-3}$	1.72	3.01	4.73
G, 3	$1.38 \cdot 10^{-3}$	2.13	3.58	5.71
V, 1	$6.67 \cdot 10^{-3}$	1.41	2.42	3.83
V, 2	$1.66 \cdot 10^{-3}$	1.78	3.09	4.87
V, 3	$7.17 \cdot 10^{-4}$	2.17	3.61	5.78

Table 1: Performance on the Lion mesh.

	MSE	bpv layers	bpv indices	bpv total
U, 1	$2.92 \cdot 10^{-5}$	1.19	2.10	3.29
U, 2	$1.72 \cdot 10^{-5}$	1.44	2.55	3.99
U, 3	$1.49 \cdot 10^{-5}$	1.72	2.97	4.69
G, 1	$1.68 \cdot 10^{-5}$	1.20	2.09	3.29
G, 2	$4.93 \cdot 10^{-6}$	1.42	2.51	3.93
G, 3	$2.77 \cdot 10^{-6}$	1.72	2.95	4.67
V, 1	$1.65 \cdot 10^{-5}$	1.23	2.14	3.37
V, 2	$4.98 \cdot 10^{-6}$	1.56	2.67	4.23
V, 3	$2.84 \cdot 10^{-6}$	1.93	3.19	5.12

Table 2: Performance on the Person mesh.

	MSE	bpv layers	bpv indices	bpv total
U, 1	$5.59 \cdot 10^{-2}$	1.98	3.20	5.18
U, 2	$4.90 \cdot 10^{-2}$	2.79	4.19	6.98
U, 3	$4.77 \cdot 10^{-2}$	3.29	4.85	8.14
G, 1	$1.13 \cdot 10^{-2}$	2.01	3.28	5.29
G, 2	$5.48 \cdot 10^{-3}$	2.78	4.25	7.03
G, 3	$4.36 \cdot 10^{-3}$	3.30	4.92	8.22
V, 1	$7.62 \cdot 10^{-3}$	2.02	3.27	5.29
V, 2	$2.02 \cdot 10^{-3}$	2.80	4.33	7.13
V, 3	$1.00 \cdot 10^{-3}$	3.34	4.99	8.33

Table 3: Performance on the Homer mesh.

	MSE	bpv
Lion	$2.60 \cdot 10^{-1}$	3.72
	$1.03 \cdot 10^{-2}$	4.30
	$2.26 \cdot 10^{-3}$	5.19
	$6.30 \cdot 10^{-4}$	6.74
Person	$6.41 \cdot 10^{-4}$	3.56
	$2.49 \cdot 10^{-5}$	3.67
	$6.25 \cdot 10^{-6}$	4.10
	$2.50 \cdot 10^{-7}$	7.93
Homer	$2.50 \cdot 10^{-1}$	3.86
	$1.00 \cdot 10^{-2}$	5.53
	$2.46 \cdot 10^{-3}$	7.41
	$6.25 \cdot 10^{-4}$	10.02
	$2.50 \cdot 10^{-5}$	17.13

Table 4: Edgbreaker performace.

6 CONCLUSIONS

We have presented a compression algorithm for triangle meshes that uses a reference mesh of similar

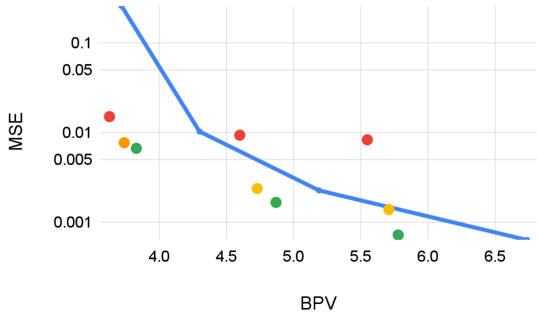


Figure 8: Performance comparison on the Lion mesh.

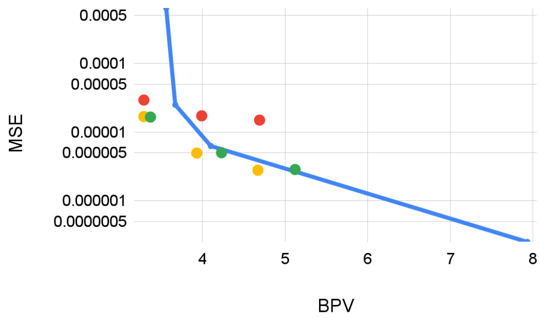


Figure 9: Performance comparison on the Person mesh.

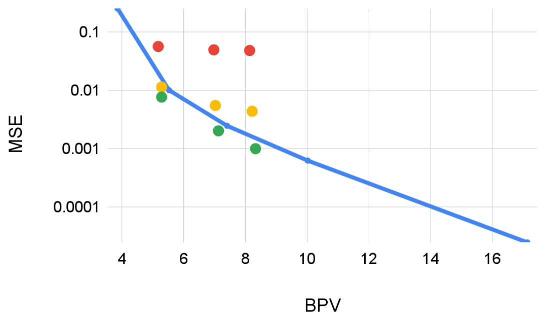


Figure 10: Performance comparison on the Homer mesh.

shape to reduce the required data rate. Although the reference shape undoubtedly provides quite a lot of information about the input mesh, exploiting it turns out to be a non-trivial task, since the particular choice of sampling of the shape represents a major portion of the data needed for defining a triangle mesh. The results, however, demonstrate that our algorithm succeeds at this using a novel combination of intrinsic encoding and hexagonal grid quantization.

In the current state, the algorithm is only practical for offline encoding, since the encoding is substantially slower than the decoding because of the exhaustive search for closest hexagon centre. This does not eliminate all practical scenarios, because often meshes are indeed encoded offline and stored for later processing, and the limiting factors are transmission time (which is improved due to better compression efficiency), and decoding time, which is much faster than the encoding.

In the future, we would like to investigate a more efficient means of finding the nearest hexagon centre. We have already performed experiments with a variant of walking algorithm with promising results, more tests are, however, still needed.

Also, we would like to explore possibilities of better mapping of local quantization areas to the curved surface of the reference mesh. It is well known that hyperbolic vertices, i.e. vertices with sum of incident angles larger than 2π, compromise the bijectivity of the exponential map, creating a certain "shadow" which cannot be reached by walking along a straight line. Overcoming this problem could lead to a further reduction of the data rate.

Finally, in the future we would like to perform experiments with an additional correction layer limiting the projection error and making it possible to reach an arbitrary coding precision.

ACKNOWLEDGEMENTS

This work was supported by the project 20-02154S of the Czech Science Foundation. Eliška Mourycová was partially supported by the University specific research project SGS-2019-016 Synthesis and Analysis of Geometric and Computing Models. The authors thank Diego Gadler from AXYZ Design, S.R.L. for providing the test data.

REFERENCES

Alliez, P. and Desbrun, M. (2001). Valence-Driven Connectivity Encoding for 3D Meshes. *Computer Graphics Forum*.

Caillaud, F., Vidal, V., Dupont, F., and Lavoué, G. (2016). Progressive compression of arbitrary textured meshes. *Computer Graphics Forum*, 35(7):475–484.

Chen, C., Xia, Q., Li, S., Qin, H., and Hao, A. (2018). High-fidelity compression of dynamic meshes with fine details using piece-wise manifold harmonic bases. In *Proceedings of Computer Graphics International 2018*, CGI 2018, page 23–32, New York, NY, USA. Association for Computing Machinery.

Corsini, M., Larabi, M. C., Lavoué, G., Petřík, O., Váša, L., and Wang, K. (2013). Perceptual Metrics for Static and Dynamic Triangle Meshes. *Computer Graphics Forum*.

Hoppe, H. (1996). Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 99–108, New York, NY, USA. Association for Computing Machinery.

Liang, K. K. (2018). Efficient conversion from rotating matrix to rotation axis and angle by extending rodrigues' formula.

Rossignac, J. (1999). Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61.

Sorkine, O., Cohen-Or, D., and Toldeo, S. (2003). High-pass quantization for mesh encoding. In *Proc. of Eurographics Symposium on Geometry Processing*, pages 42–51, Aachen, Germany. Eurographics Association.

Touma, C. and Gotsman, C. (1998). Triangle mesh compression. In *Graphics Interface*, pages 26–34.

Tutte, W. T. (1962). A census of planar triangulations. *Canadian Journal of Mathematics*, 14:21–38.

Valette, S. and Prost, R. (2004). Wavelet-based progressive compression scheme for triangle meshes: Wavemesh. *IEEE Trans. Vis. Comput. Graph.*, 10(2):123–129.

Váša, L. and Brunnett, G. (2013). Exploiting connectivity to improve the tangential part of geometry prediction. *IEEE Transactions on Visualization and Computer Graphics*, 19:1467–1475.

Váša, L. and Dvořák, J. (2018). Error Propagation Control in Laplacian Mesh Compression. *Computer Graphics Forum*.