International Journal of Shape Modeling
© World Scientific Publishing Company

SPACE-TIME SURFACE SIMPLIFICATION AND EDGEBREAKER COMPRESSION FOR 2D CEL ANIMATIONS

VIVEK KWATRA
and
JAREK ROSSIGNAC
GVU Center, College of Computing, Georgia Institute of Technology
801 Atlantic Drive, Atlanta, GA 30332, USA

Digitized cel animations are typically composed of frames containing a small number of regions; each region contains pixels of the same color and exhibits a significant level of shape coherence through time. To exploit this coherence, we treat the stack of frames as a $3D$ volume and represent the evolution of each region by the bounding surface of the $3D$ sub-volume V that it sweeps out. To reduce transmission costs, we triangulate and simplify the bounding surface and then encode it using the Edgebreaker compression scheme. To restore a close approximation of the original animation, the client player decompresses the surface and produces the successive frames by intersecting V with constant-time planes. The intersection is generated in real-time with standard graphics hardware through an improved capping (i.e. solid clipping) technique, which correctly handles overlapping facets. We have tested this approach on real and synthetic black&white animations and report compression ratios that improve upon those produced using the MPEG, MRLE, and GZIP compression standards for an equivalent quality result.

Keywords: cel animation, animation compression, geometric simplification, geometric compression

1. Introduction

In this paper, we propose a technique for compressing cel animations. Prior research on video compression has been mostly focused on videos of real scenes or on synthetic video aimed at simulating real environments, which consists of continuously varying colors and brightness levels over the image in a single frame, and also over time. In contrast, in cel animations, each frame consists of several regions of uniform color. The regions change their shape over time, but their color remains the same. We exploit this particularity of cel animations to increase compression ratios. Since the color inside each region of a cel animation remains constant, we need to store only one color per region for the entire animation. The remaining information lies in the geometric evolution of the shape of these regions. Our goal is to devise a compact representation for these evolving shapes.

One might consider several approaches for encoding the evolution of the shape of these regions over time. As a frame of reference, consider a short black and white video of 336 frames showing the evolution of a single region at a 320×240 resolution.

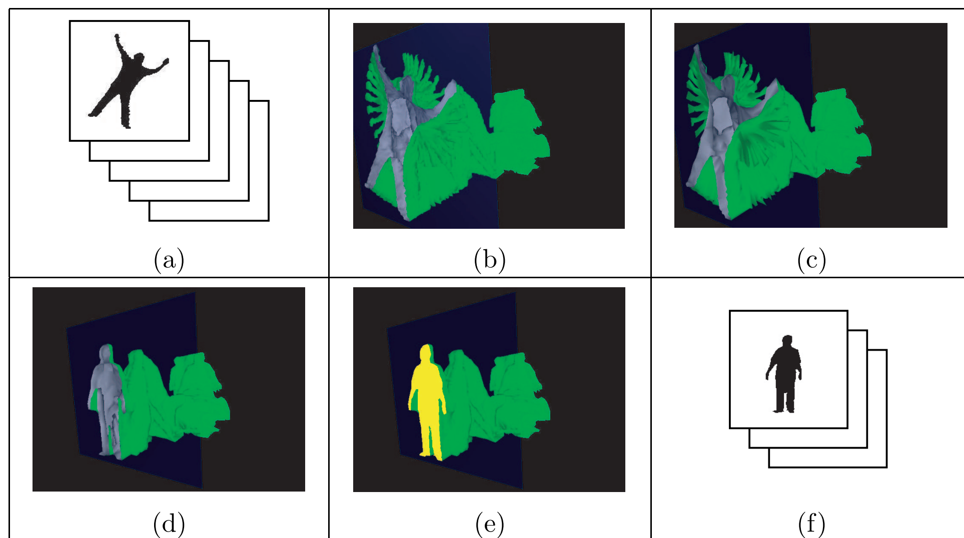

Fig. 1. (a) Consecutive frames of the animation of a uniform (black) region are shown. (b) The bounding surface of the region is produced by voxelization and iso-surface extraction. (c) Bounding surface shown after simplification; this surface is further compressed using Edgebreaker. (d) After downloading and decompressing the bounding surface, the client player clips it with a plane whose offset represents time. (e) An improved hardware-assisted capping technique is used to obtain the cross-section of the surface intersected by this plane (f) Each frame of the animation is rendered by painting these cross-sections.

It contains 25,804,800 black or white pixels and thus could be stored as a sequence of bits (0 for a white voxel and 1 for a black voxel), which without compression would occupy 3.2MB(Megabytes). Compressing the file using Microsoft's Run Length Encoding (MRLE) [1] reduces storage to 1.2MB. MPEG-1 [2] achieves a slightly better compression, reducing storage to 1.1MB (a 3:1 compression ratio over the raw one-bit-per-pixel encoding). Surprisingly, running a lossless GZIP compression [3] over the raw data produces significantly better results than MPEG and MRLE, reducing its size down to 78KB(Kilobytes). The technique described here compresses that same animation down to 80KB (a 10:1 compression ratio over MPEG-1 and MRLE) without any perceivable loss in quality. It is hence comparable to GZIP for high fidelity compression. However, when a small loss of accuracy is acceptable, our technique outperforms MPEG-1, MRLE, and GZIP. For example, we compress the original animation down to 16.8KB (thus achieving a 60:1 compression ratio over MPEG-1 and MRLE), while a GZIP compressed low-resolution version of the same video (160×120×168) requires 21KB and produces a clearly lower quality result.

In order to achieve this compression, we treat the black and white video as a $3D$ arrangement of voxels, extract the bounding surface of the three-dimensional black region, and then simplify and compress that surface, taking advantage of any spatio-temporal coherence present in the evolving region.

More precisely, we pile the frames of our $2D$ animation into a block embedded in $3D$ space. A sample point in this space has integer coordinates (x, y, t), where (x, y) are the coordinates of a pixel in the image plane, and t represents time, or equivalently, the frame number in the animation. We call this $3D$ space, the image-time space. As time evolves, each region sweeps a volume V through the image-time space. The bounding surface S of V interpolates the bounding curves of the region for each discrete value of t. We can use compact representations of S to encode the shape of V. For example, when the region is a constant radius disc that moves at constant velocity, V is an inclined cylinder and can be accurately represented by a small number of triangles, regardless of the number of frames. We compress the surface S using existing $3D$ geometric simplification [4, 5, 6, 7] and compression [8, 9, 10, 11, 12, 13, 14, 15] techniques. Our approach will benefit from any further progress in surface simplification and compression, two active fields of research.

The client player downloads the compressed representation of S and decodes it. We can play the animation directly from S in real-time, thus avoiding the need for reconstructing and storing the video. For that, we use the standard graphics hardware. We exploit the fact that each frame of the animation is a cross-section of the volume enclosed by the bounding surface. More specifically, the nth frame can be rendered by computing the cross-section of this volume with the plane $t = n$. However, we don't need to compute a geometric representation of the cross-section explicitly. Instead, we can use graphics hardware to clip away the portion of the surface for which $t < n$. The triangles of S are oriented consistently so that they face outwards from V. We use a standard capping approach [16] to paint the triangles that face the viewpoint in white (background color), and the triangles that face

away (and are seen through the cross-section as shown in Figure 1(d)) in black.

Front-facing triangles that lie behind the visible back-facing triangles could be eliminated using a standard z-buffer. However, when S is the result of a simplification process, it may contain regions where front-facing and back-facing triangles overlap, and thus have the same depth. To correctly handle such situations, we use a parity rule to decide on the color of each pixel. This decision process is supported in hardware using OpenGL's standard stencil buffers.

The above approach assumes that the animation has already been segmented into uniformly colored regions and that each region can be analyzed separately. The bounding surfaces defined by each of these regions can then be combined to obtain the representation for the entire animation. Hence, we focus our discussion on the compression and display of a single region of the animation.

We choose to represent the bounding surface as an irregular mesh composed of arbitrary vertex locations along with connectivity information. The rationale behind this choice as opposed to an axis-aligned representation (for e.g. an octree) is that the freedom of placing vertices in a continuous space allows us to capture shape-redundancy in a much stronger fashion. Further more, such a representation allows us to visualize the animation as a $3D$-surface and gives the animator a very intuitive and interactive tool for editing the animation using $3D$ deformation tools.

In the remainder of the paper, we review related work and explain the details of the various steps of our technique including bounding surface extraction, simplification, compression, and animation playback. We then present results obtained from three test animations, and finally conclude with a summary of our contributions.

2. Prior Art

A vast amount of work has been focused on image and video compression, resulting in a variety of techniques and standards, which include JPEG (for images) [17] and MPEG (for video) [2]. These handle images and videos of natural scenes, for which neighboring pixels may all have different colors. These general-purpose compression techniques may be inferior to specialized approaches when targeting a specific subclass of videos: cel animations in our case.

One of the ways to encode a cel animation consisting of multiple uniformly colored regions is to convert the animation into a set of binary digital videos, one for each color. Each video represents a constant-color region and is associated with a color descriptor. As an example, if we start with a $W \times H$ resolution color-animation with K regions and N frames, the raw data size would be $24WHN$ bits. Here it is assumed that we use 24 bits per pixel to store the color information. If we use K binary videos, each having WHN bits, we obtain a total of $WHNK$ bits and a color table of size $24K$. The compression-ratio of $24/K$ is not a large gain, even considering that K is usually small in cel animations. The compression ratios of these approaches are limited because they exploit coherence only in color, but not in shape or time. A slightly better approach would be to store the entire animation

as one video with $\log K$ bits per pixel, referencing an entry in a color look-up table. This choice would result in a storage size of $WHN \log K + 24K$ and the resulting raw data could be compressed directly using entropy-based techniques.

A binary video is highly likely to have a large number of long sequences containing only 0's or 1's, therefore run-length encoding (RLE) [1] and entropy-based compression techniques [3, 18, 19] seem appropriate. However, they only exploit coherence in one dimension. An octree encoding [20] of the binary set of black voxels may be more effective at exploiting the spatio-temporal coherence of the $3D$ representation. The octree approach recursively splits an inhomogeneous volume into octants, thus avoiding the need to represent the voxels of large uniform regions. The compression ratio depends on the alignment of the faces of V with the axis-aligned bisecting planes of the octree. If the octree has L leaf-nodes, then the storage would be $O(L \log K)$. Since the leaf nodes contain boundary voxels, L is expected to be $O((WHN)^{2/3})$ and is dominated by the number of the finest resolution boundary voxels.

A different approach would be to encode the bounding curve of the region in each frame of the video (obtained after color-segmentation of that region). This method better addresses the issue of shape-redundancy. By considering just the bounding curve, we are already removing redundant information about the interior and exterior of the region. Suppose we encode the curves losslessly; for every curve, we store the (x, y) coordinates of a seed pixel on the curve. Starting at the seed, we now move along the curve one pixel at a time, storing for each pixel its displacement from the previous pixel. Since there are eight possible displacement vectors, we require three bits to encode each move. If we have on the average, P curve-pixels in every frame, and we use 20 bits to represent the seed (which we assume is only one per frame), we would end up with a representation of size $(3P + 20)NK$ bits - for most animations P should be $O(W + H)$.

Better compression ratios should be easily achievable if lossy encoding is acceptable. A lossy compression technique would encode approximations to the actual curves by taking advantage of any smoothness present in the curve. While this approach takes into account the spatial coherence present in individual curves, it does not address the fact that they are highly correlated with each other. Our technique essentially extends this idea into $3D$, exploiting the spatio-temporal coherence present in the animation.

3. Bounding Surface Extraction

To represent an animation by a bounding surface, we need to extract the surface from the image-time space and build a representation for it.

3.1. *Definition of Bounding Surface*

The bounding surface S should separate the black voxels from the white ones. The simplest definition of such a surface is the set of square faces that lie between black

and white voxels. Such a surface, however, contains sharp edges and corners which is undesirable. In order to alleviate this problem, we define a scalar field F over the image-time volume:

$$F(p) = \begin{cases} 0 & \text{if p is at the } \textit{center} \text{ of a black voxel} \\ 1 & \text{if p is at the } \textit{center} \text{ of a white voxel} \end{cases}$$

F is initially not known at points other than these voxel centers and must be approximated by interpolation. Once F is defined everywhere, the bounding surface is defined as the set of points for which $F = 0.5$. Naturally, the smoothness of the surface depends on the smoothness of the interpolant for F, as shown in Figure 2.

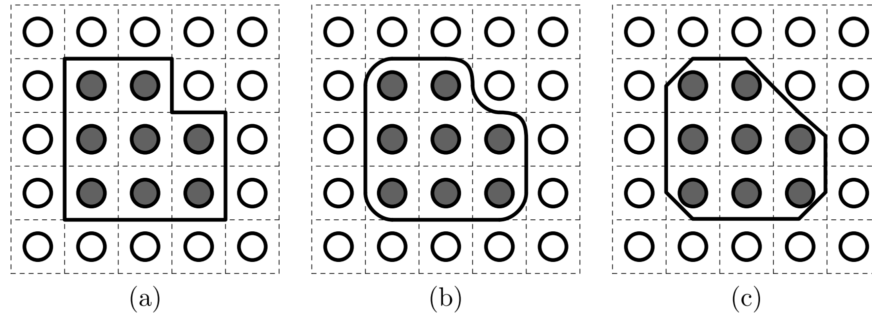

Fig. 2. The bounding surface obtained by using different interpolation schemes for scalar field F - shown for a $2D$ voxel-grid. (a) Piecewise-constant interpolation. (b) Bilinear interpolation. (c) Piecewise linear interpolation.

If we use a *piecewise-constant* interpolation for F (Figure 2(a)) using a box-shaped filter, it takes the value of 0 everywhere inside a black voxel and the value of 1 everywhere inside a white voxel. Therefore F is discontinuous along the faces where black and white voxels meet. Although $F(p) \neq 0.5$ for any p, F does change from 0 to 1 (or vice-versa) at the points of discontinuity and therefore they are the natural candidates to be a part of the bounding surface. Note that the surface thus obtained is exactly the surface we would have obtained by just considering the faces separating black and white voxels (as discussed above).

A more sophisticated interpolation scheme should definitely give a smoother surface. One such scheme is *bilinear interpolation* (Figure 2(b)). The surface obtained using bilinear interpolation is very smooth; however, it is computationally expensive to generate such a surface. Moreover, it is a quadric surface that has to be resampled in order to convert it to a triangle mesh representation which is desirable since we want to run a simplification algorithm over it. Taking middle ground, we choose a *piecewise-linear* interpolant (Figure 2(c)) that is simple to compute and still smooth enough for our purpose.

3.2. *Iso-surface Extraction*

The procedure for extracting the triangulated surface makes use of tetrahedral decomposition of each voxel. It may be viewed as a special case of the approach described in [23]. The main difference lies in the fact that we are not extracting the iso-surface from an arbitrary voxel-volume. Instead, the voxels have scalar values of 0 and 1 only, and the iso-surface has a scalar value of 0.5. Consequently, a linear interpolation between two adjacent voxel centers with different scalar values would always lie at their mid-point. We briefly describe the iso-surface extraction procedure and some of the details that were handled differently by us. For more information on this method, please read [23].

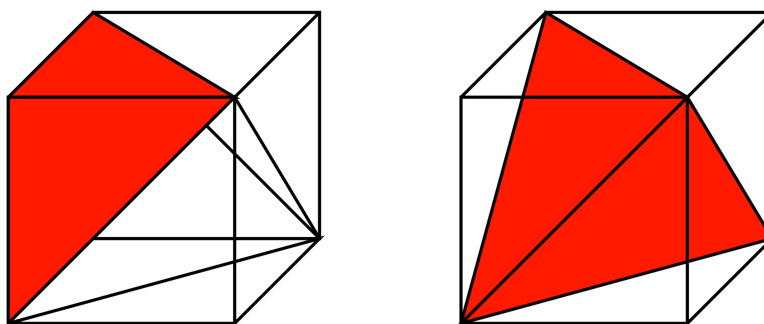

Fig. 3. A cube in the voxel volume is decomposed into five tetrahedra. Four of these tetrahedra are identical in shape (the canonical shape is shown as shaded in the left figure). The fifth tetrahedron is embedded within the first four (shown as shaded in the right figure).

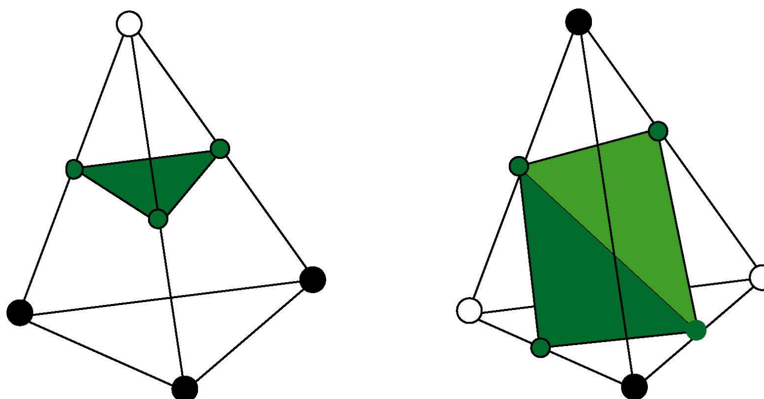

Fig. 4. Intersection of the iso-surface with a tetrahedron can either be a triangle (left) or a quadrilateral (right), depending upon the color of the voxels on its corners.

The traditional approach to iso-surface extraction is the *marching-cubes* algorithm [24]. In this algorithm, the entire voxel-volume is divided into cubes, so that

each cube has eight neighboring voxels at its corners. The intersection of the surface with each of these cubes is computed and the union of these intersections gives us the iso-surface. The tetrahedral decomposition algorithm modifies marching-cubes by further dividing each cube into five tetrahedra (see Figure 3), and computing the intersection of the surface with each tetrahedron. Note that both these algorithms - marching tetrahedra and marching cubes - implicitly assume a piecewise-linear scalar field F as is desired. However, such a linear F in $3D$ can be guaranteed to go through atmost four points. This fact is crucial because it is the sole reason for the ambiguities that arise in the marching cubes algorithm [25] - marching cubes tries to find the level-set of a function obtained through linear interpolation over eight points. Tetrahedral decomposition alleviates this problem because the interpolation happens only over four points; it also greatly reduces the number of possible outcomes of the intersection computation.

To compute a surface-tetrahedron intersection, we need to know the binary scalar values of the voxels at the corners of the tetrahedron. Only tetrahedron edges with different scalar values at their corners intersect the surface. The possible outcomes of the intersection can either be a triangle or a quadrilateral, as shown in Figure 4. If the mid-point of each edge that has different scalar values at its corners is picked as a vertex of the bounding surface, we may get undesirably sharp corners in the surface (see Figure 5). This would happen, for example, if the candidate edge is a face-diagonal of the original cube that was tetrahedralized and three of the corners of that square face have the same scalar value (Figure 5, top-left). Note that if the other diagonal had been chosen during tetrahedralization, the sharp corner wouldn't have been created (Figure 5, top-right). To avoid this effect, we check for the condition that can create the sharp corner and if it is true, we displace the vertex towards the appropriate corner (Figure 5, bottom). In [23], the displacement is computed using bilinear interpolation. In our case, because of the binary nature of our voxel-volume, the displacement is always one quarter of the diagonal. This step ensures that the bounding surface is independent of the tetrahedralization - we are referring to the tetrahedralization of a single voxel which can be done in two ways depending on which diagonal is chosen to split each face.

For simplification, compression, and even for rendering, it is important that the iso-surface extraction procedure generate a triangle mesh that has the connectivity of a watertight surface. In particular, if we consider the triangles and edges to be relatively open (i.e., not including their boundaries), then any two elements from the total set of triangles, edges and vertices must be disjoint. For example, a vertex cannot coincide with an interior point of a triangle or of an edge. Furthermore, two edges cannot intersect. Also, each edge must have an even number of incident triangles. Such meshes are simplicial complexes that represent the boundaries of solids. They can be represented as pseudo-manifolds using a simple data structure, such as the one described in [13]. Techniques for pairing triangles incident upon non-manifold edges in a manner that prevents self-crossings of the surface and that minimize the number of non-manifold vertices are discussed in [26]. They are im-

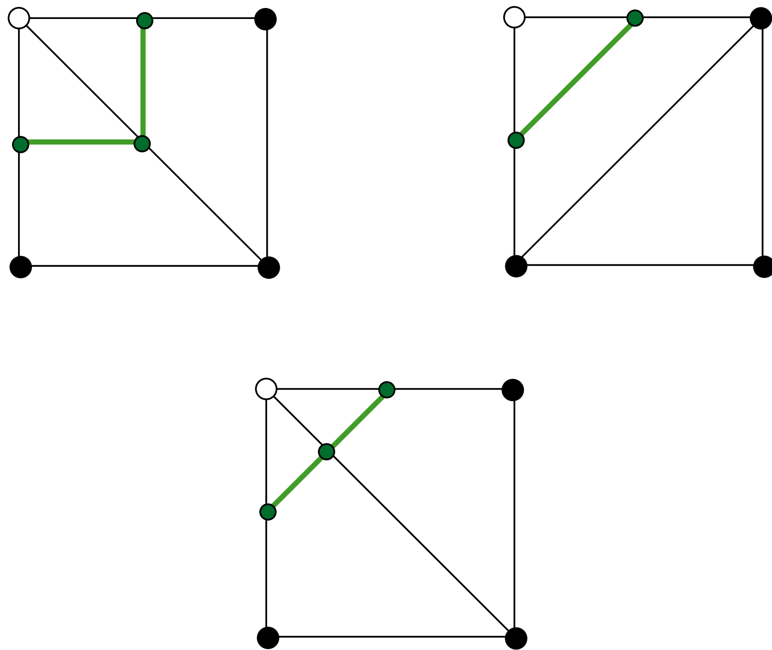

Fig. 5. For the voxel configuration shown above, the computed iso-surface depends on the diagonal chosen during tetrahedralization - if the diagonal intersects the iso-surface (top-left), a sharp corner is created. If the diagonal does not intersect the iso-surface (top-right), the iso-surface is smooth. The sharp corner created in the first case is removed by moving the vertex of the iso-surface towards the appropriate voxel (bottom). The resulting iso-surface now looks identical in both cases.

portant for reducing the storage of the mesh and for guaranteeing the compatibility of the surface orientation with the status of the volume it bounds.

We obtain a valid triangulation, as defined above, by ensuring that the decomposition of the cubes generates tetrahedra that are consistent with each other. In our situation, consistency among tetrahedra is satisfied if two triangles that bound two different tetrahedra are either identical or disjoint (remember that triangles are relatively open and do not include their edges). We achieve this by alternating tetrahedralizations, so that the tetrahedral decompositions of two adjacent cubes will be mirror images of one another with respect to the plane of symmetry between the cubes.

The algorithm described above is simple and easy to implement. It also handles the topological changes between pairs of consecutive slices automatically. This simplicity comes at a cost though. Since we are dealing with animation, there exists a correspondence between various structural elements of the object being animated which is not explicitly captured when the surface is constructed voxel by voxel. This means that the surface obtained will be sensitive to the sampling frequency in space and time, and simplification can potentially introduce significant artifacts in the animation. A possible alternative is to construct a skeleton or *Reeb graph* and use it to segment the $3D$ surface into regions that are suitable for compression [21, 22]. We, however, stick to the voxel-marching approach mainly for reasons of simplicity.

4. Simplification

We now have a triangle mesh represented as a pseudo-manifold. We simplify it using a sequence of edge-collapse operations [4, 7] that at each step minimize the error between the resulting surface and the original one. We have used the simplification technique developed by [5].

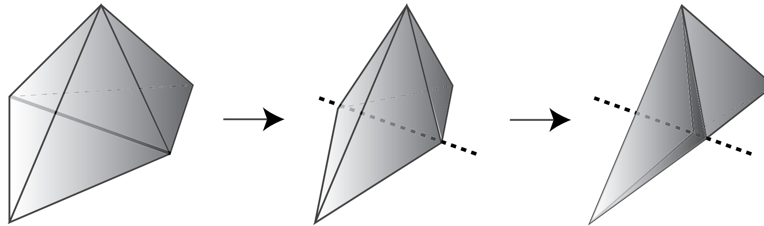

Fig. 6. Simplification with edge-collapse operations can cause triangles to fold over each other and overlap. The thick edge (solid in left and dotted in middle and right) is being collapsed here; for the configuration shown, it ultimately produces a pair of overlapping triangles.

We have found that simplification is very effective at decreasing the triangle count by 95-99% without noticeable error in the resulting animations. For instance, the iso-surface for the animation shown in Figure 1 originally involved 1,850,200

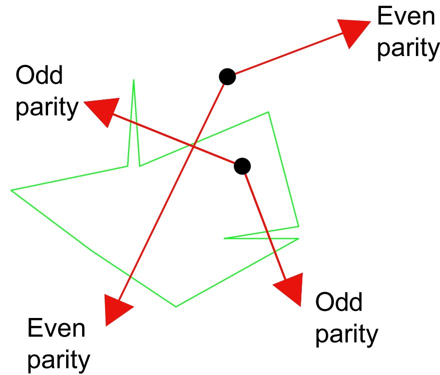

Fig. 7. The mesh obtained through simplification is watertight even if it has overlapping triangles; a parity test can be used to check if a point lies inside or outside the mesh - shown here for the $2D$ case.

triangles (top center) and was simplified to 18,502 triangles (top right) with an error of less than 0.2%, as measured with Metro [27].

The edge-collapse based simplification algorithm can produce degenerate cases where the surface folds upon itself producing overlaps between triangles (Figure 6). However, it preserves the manifold connectivity of the mesh which means that it is watertight; a point can be tested for being inside or outside the volume bounded by the surface by a simple parity checking logic (Figure 7) - the parity of the number of intersections of the surface and a ray shot from the point in question to ∞ is odd if the point is inside and even if the point is outside. As we shall see, this has implications on the rendering phase.

5. Compression

We have used the compression approach described in [10, 12, 13]. The first step of this approach is to quantize the coordinates of the vertices of the simplified surface. For this, we compute a coordinate system in which all vertex coordinates are represented as integers and span the range $[0 \;\; 2^B - 1]$, where B is the number of bits needed to represent them. Typically, B is chosen to match the original voxel resolution. For the example in Figure 1, we used $B = 9$.

We encode the connectivity of the mesh by building a spiraling triangle-spanning tree and by storing for each visited triangle, one symbol from the set $\{$**C**,**L**,**E**,**R**,**S**$\}$. A **C** corresponds to a situation where the tip of the triangle has not yet been visited. The other symbols correspond to the four possible situations where the tip vertex has been visited, and the left and right triangle neighbors exhibit different combinations of visited or not visited. The sequence of symbols may always be encoded with less than 1.8 bits per triangle and suffices to recover the connectivity

graph for all simply connected triangle meshes. It may be extended to support meshes with handles by adding two references per handle. In practice, it compresses to about 1 bit per triangle for sufficiently complex meshes.

The number of vertices is roughly half the number of triangles. We compress their coordinates by encoding the vertices in the order in which they are encountered by the connectivity compression traversal described above. Instead of encoding the location of each vertex, the encoder and the decoder both use the same prediction mechanism, which is based on the parallelogram rule suggested in [15]. We encode in a lossless fashion, the residue (i.e., corrective) vectors between the quantized version of the predicted locations and the actual quantized locations. Because the predictions are usually accurate, the corrective vectors are short and can be significantly compressed using entropy codes. For example, the storage necessary for the corrective vectors of the surface in Figure 1 was compressed to an average of 4.3 bits per coordinate.

The player would receive and decode the sequence of symbols and the corrective vectors. It would then decompress them as described in [12, 13], reconstructing a triangle mesh that is suitable for replaying the animation in real-time. The source code and compact description of the Edgebreaker compression and decompression are available [28].

6. Animation Playback

OpenGL has support for arbitrary clipping planes, besides the usual ones necessary for defining the viewing volume. Once we define a clipping plane, all surfaces are clipped with that plane before being rendered. Now consider the bounding surface of an animated region. We clip this surface with the plane $t = n$. We retain all points on the surface with $t \geq n$ and remove the rest. We call the half space containing points with $t \geq n$, the *retained half-space.* The half space represented by $t < n$ is called the *culled half-space.* We render an orthographic view of the surface onto a projection plane that is parallel to the clipping plane and lies in the *culled half-space* (Figure 1(f)). The clipped surface is no longer watertight, i.e. we can now see back-facing as well as front-facing triangles of the surface. Also, the projections of the back-facing and front-facing triangles meet at the intersection-boundary of the surface and the clipping plane. Figure 1(d) shows the clipped surface with the back and front facing polygons painted in purple and green respectively. This bounding curve partitions the projection plane into two regions - interior and exterior. The exterior contains the background and the projection of front-facing triangles, while the interior contains the projection of back-facing triangles. If we render the back-facing and front-facing polygons with different colors, the bounding curve is automatically rendered as the boundary between the these two colors (Figure 1(d,e)). In particular, if we render the front-facing triangles in background color, and the back-facing triangles in the actual color of the animated region, we get back a frame of the animation. We can do

this for all successive frames of the animation, thereby playing back the animation. Since we are actually rendering individual frames, we can capture them offline into a video and then use the video for playback.

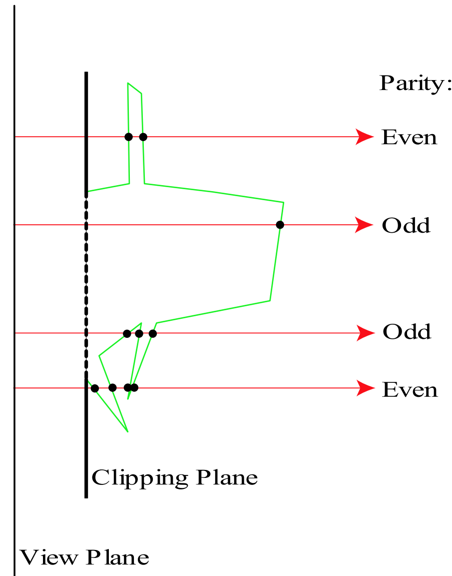

Fig. 8. Using parity-checking logic to render a frame of the animation - shown for the $2D$ case. If a ray shot from the viewing plane towards the volume intersects the surface an odd number of times, it sees a back facing triangle; conversely, if it intersects the surface an even number of times, it sees a front-facing triangle. Hence the parity of each pixel uniquely determines if it is a part of the animating region or not - the back-facing triangles project to the interior of the animating region and the front-facing ones project to its exterior (or background).

While in theory the above method should always work, there are problems that may occur in practice and need to be overcome. The simplification stage in compression involves edge-collapses that may lead to overlapping triangles (Figure 6). Hidden-surface removal using z-buffering would show one of these triangles, but which one is actually visible cannot be predicted deterministically. We can solve this problem by using the parity of the number of triangles projecting onto a pixel. Consider the surface before clipping; as mentioned before, this surface is watertight. A ray starting from a point that is inside the volume enclosed by this surface would hit the surface an odd number of times. On the contrary, a point outside the enclosed volume would hit the surface an even number of times (Figure 7). Now, consider the clipped surface; the portion of the clipping plane that lies within the volume enclosed by the original unclipped bounding surface obviously consists of points inside that volume. Therefore, any ray that starts at one of those points would hit the unclipped bounding surface an odd number of times. If the ray is directed towards the *retained half-space*, it would hit the clipped surface also an

odd number of times, because in that half space the surface has not changed. A similar argument can be used to show that a ray that starts from the portion of the clipping plane that lies outside the enclosed volume and that is directed towards the *retained half-space* would hit the clipped surface an even number of times. The two portions of the clipping plane that we have talked about are exactly the interior and exterior of the bounding curve of the animated region in that frame. Therefore, the portion inside the enclosing volume is where the back-facing triangles project and the one outside is where the front-facing triangles project. We combine all these facts to come up with the following test: To decide whether the triangle visible at a pixel is back-facing or front-facing, we shoot a ray from that pixel, orthogonal to the projection plane. If the ray pierces an odd number of triangles of the clipped surface, it should see a back-facing triangle. Otherwise, it should see a front-facing triangle, or the background (in case of no intersections). In fact, once we have this information about each pixel, we don't need to explicitly render the surface again because we have already gathered the information about what each pixel sees. So we can just paint each pixel appropriately. See Figure 8 for a schematic description.

We implement this test using the stencil-buffers in OpenGL. We first render the clipped surface, flipping the bit at a pixel in the stencil buffer whenever it sees a surface. Assuming that we initialized the buffer to 0, all pixels in the interior of the animated region will now be set to 1 *(on)*. All remaining pixels would be set to 0 *(off)*. We then use the stencil buffer as a mask, and render the *on* pixels with the color of the animated region, and the *off* pixels with the color of the background. We do this by rendering a polygon parallel to and as large as the projection plane with the color of the animated region, and with the stencil buffer as a mask.

7. Results

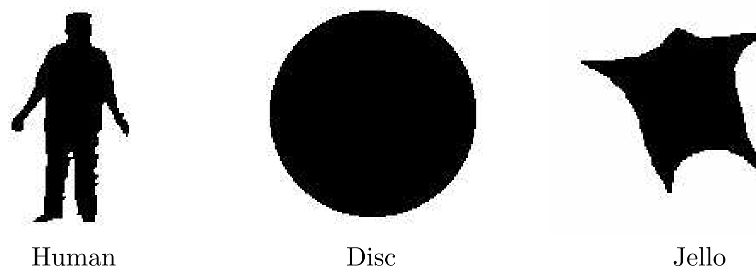

Fig. 9. Snapshots from the various datasets we used for experimentation.

We have implemented our compression technique for animations with only one animated region. It is reasonable to assume that an animation with multiple regions can be decomposed into single-region animations using color information and/or

connected component analysis. We tested our technique on three videos (Figure 9). Besides our technique, we also compressed them with GZIP, MPEG-1 and MRLE. In an attempt at lossy compression with GZIP, we used GZIP to compress a lower resolution (half of the original in all three dimensions) version of the video. Our technique outperformed each of the other techniques. The results of *lossy* GZIP had compression ratios comparable to ours but were significantly poor in quality. The results are summarized in Table 1 and Figure 10. The details are as follows:

(i) Silhouette video of a **human** performing certain actions: this video had a spatial resolution of 320×240 pixels and 336 frames. There were 359,004 boundary-faces in the video. Using our technique, we could compress it to 80KB without any perceivable loss. With more lossy compression, we could compress it down to 16.8KB. The original iso-surface had 924,778 vertices and 1,850,200 triangles. We simplified it to 1% of the original number of triangles. The final mesh had 8,907 vertices and 18,502 triangles. Predictive encoding of vertices required 13 bits per vertex. Edgebreaker (for connectivity encoding) used 1 bit per triangle. GZIP compressed the same video to 78KB losslessly; *lossy* GZIP achieved 21KB but with considerable loss in quality. MPEG-1 compressed it to 1.1MB, while MRLE compressed it to 1.2MB.

(ii) Video of a circular **disc** first increasing and then decreasing in radius: this video had 320×240 pixels per frame and 100 frames. There were 83,948 boundary-faces in the video. We compressed the video to 3.7KB without much loss in quality, while GZIP achieved 17.5KB with lossless compression and 4.1KB with lossy compression.. MPEG-1 compressed it to 206KB, while MRLE compressed it to 170KB.

(iii) Video of a **jello**-like object performing some actions: this video had 320×240 pixels per frame and 590 frames. There were 456,738 boundary-faces in the video. We compressed the video to 19.3KB without much loss in quality, while GZIP achieved 122KB with lossless compression and 25.7KB with lossy compression. MPEG-1 compressed it to 1.5MB, while MRLE compressed it to 971KB.

The disc video was relatively simple with only the radius of the disc changing over the sequence; it helped us to empirically verify the soundness of our technique. The human and the jello video, on the other hand, were relatively complex where the animated region changed its shape significantly during the course of the animation; this allowed us to test the robustness and effectiveness of the technique.

By adjusting the level of simplification and quantization of the original bounding surface, we can control the quality of compression. Figure 11 shows snapshots from the animations obtained after applying different amounts of simplification. The quality of the result deteriorates as the amount of simplification is increased. Another factor that affects the quality of compression is the initial sampling of the animation. The temporal sampling in the human video was poor (15 frames per second), causing more artifacts in regions of large motion. This highlights a

Table 1. Comparison of compression results. *Size* is the size of the compressed animation. *Ratio* is the (compression) ratio of the animation's original size to its compressed size.

Technique	Human(3.2MB) 320×240×336		Disc(957KB) 320×240×100		Jello(5.6MB) 320×240×590	
Compressed	Size(KB)	Ratio	Size(KB)	Ratio	Size(KB)	Ratio
MPEG	1102.0	2.9	206.0	4.6	1530.0	3.6
MRLE	1200.0	2.7	170.0	5.6	971.0	5.72
GZIP (Lossless)	78.0	41.8	17.5	54.7	122.0	45.6
GZIP (Lossy)	21.0	151.4	4.1	233.4	25.7	216.3
Simplification+Edgebreaker	16.8	189.2	3.7	378.0	19.3	288.1

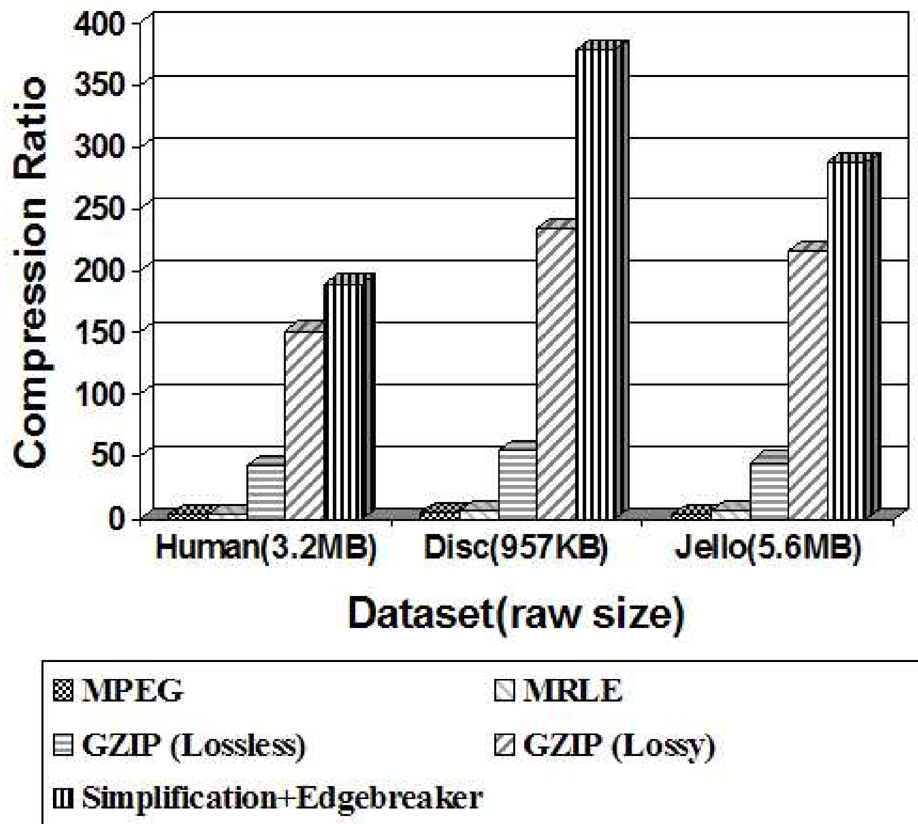

Fig. 10. These graphs show a comparison of the compression ratios (original to compressed size) obtained via different techniques. The original size of each animation is indicated below its graph.

limitation of our technique: it is sensitive to the velocity of motion in the animation. A better iso-surface extraction procedure should help in alleviating this problem.

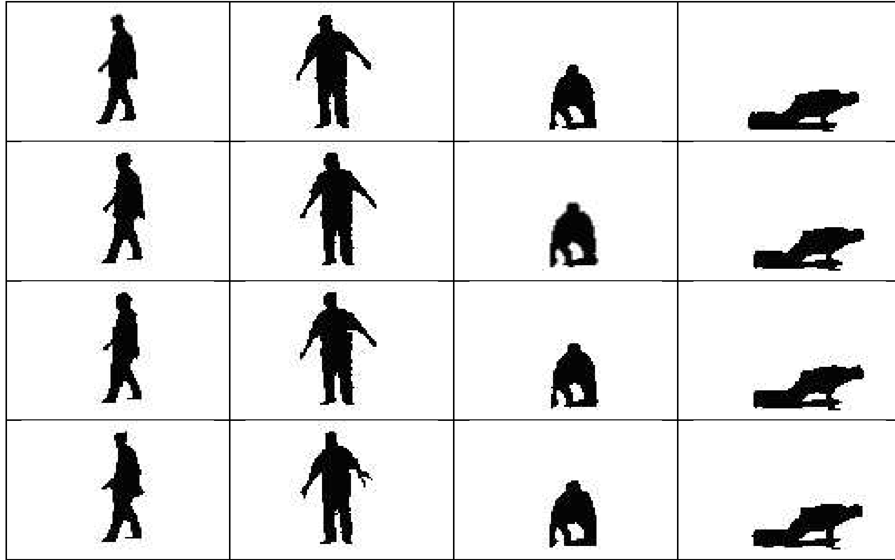

Fig. 11. Comparison of animations obtained by applying different amounts of simplification. The first row shows four frames from the original animation. The subsequent rows show the same frames from simplified versions of the animation. The simplification (expressed as the ratio of the number of triangles in the simplified mesh to that in the original mesh) is 5%, 1% and 0.5% in the 2nd, 3rd and 4th rows respectively.

We also built a tool for visualizing the bounding surface extracted from the animation. Figure 1(b,c,d,e) shows the visualization of the animation. We show the clipped bounding surface in $3D$. The interior of the surface and the clipping plane are visible. The cross-section of the surface intersecting the clipping plane is same as the bounding curve of the animated region.

8. Conclusion

We have presented a technique that leverages the power of $3D$ compression and simplification, for the task of compressing cel animations. By making use of surface simplification techniques, we have provided a multi-resolution approach to cel animation compression. The iso-surface extraction procedure demonstrates the adaptation of existing techniques to black and white voxels. We have also devised an improved capping technique for computing cross-sections of $3D$ surfaces, which successfully deals with the problem of overlapping triangles generated as a result of simplification. Additionally, it runs on standard graphics hardware, allowing real-time playback of the animation.

We believe that this technique should also be applicable to other application domains. Medical imaging and segmentation is one such area where our technique could be used to segment out and compress $3D$ shapes of organs from $2D$ slices. Other possible applications are compression of weather-visualization data, and web-based instruction manuals and tutorials containing schematic explanations.

Acknowledgements

We would like to thank Davis King and Andrzej Szymczak for helping with the simplification and compression software. This work was partly supported by the National Science Foundation under CARGO grant 0138426.

References

1. S.W. Golomb, “Run-length encodings“, *IEEE Transactions on Information Theory IT-12(3)*, 399-401, 1966.
2. D.J. Le Gall, “The MPEG video compression algorithm“, *Signal Processing: Image Communication*, 4:129-140, 1992. 67
3. J. Gailly, “GZIP program for UNIX“, 1993.
4. H. Hoppe. “Progressive Meshes“, *Proceedings of ACM SIGGRAPH*, August 1996, pages 99-108.
5. P. Lindstrom and G. Turk, “Fast and Memory Efficient Polygonal Simplification“, *Proceedings of IEEE Visualization*, pp. 279-286, October 1998.
6. J. Rossignac and P. Borrel, “Multi-resolution 3D approximations for rendering complex scenes“, in *Geometric Modeling in Computer Graphics, Springer Verlag, Eds. B. Falcidieno and T. L. Kunii*, Genova, Italy, June 28-July 2, 1993, pp. 455-465.
7. R. Ronfard and J. Rossignac, “Full-range approximations of triangulated polyhedra“, *Proceedings of Eurographics’96, Computer Graphics Forum, Vol. 15, No. 3*, August 1996, pp. C-67.
8. D. King and J. Rossignac, “Guaranteed 3.67V bit encoding of planar triangle graphs“, *11th Canadian Conference on Computational Geometry (CCCG”99)*, pp. 146-149, Vancouver, CA, August 15-18, 1999.
9. D. King, J. Rossignac and A. Szymczak, “An Edgebreaker-based efficient compression scheme for regular meshes“, *12th Canadian Conference on Computational Geometry*, Fredericton, New Brunswick, August 16-19, 2000.
10. J. Rossignac, “Edgebreaker: Connectivity compression for triangle meshes“, *IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 1*, January - March 1999, pp. 47-61.
11. J. Rossignac, “3D Compression and progressive transmission“, *Lecture at the ACM SIGGRAPH conference*, July 2-28, 2000.
12. J. Rossignac and A. Szymczak, “Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker“, *Journal of Computational Geometry, Theory and Applications, Volume 14, Issue 1-3*, November 1999, pp. 119-135.
13. J. Rossignac, A. Safanova and A. Szymczak, “3D compression made simple: Edgebreaker on a Corner Table“, *Shape Modeling International Conference*, Genoa, Italy May 2001.
14. G. Taubin and J. Rossignac, “Geometric Compression through Topological Surgery“, *ACM Transactions on Graphics, Volume 17, Number 2*, April 1998, pp. 84-115.
15. C. Touma and C. Gotsman. “Triangle Mesh Compression“, *Graphics Interface ‘98*,

1998, pp. 26-34.
16. J. Rossignac, A. Megahed, and B.O. Schneider, "Interactive Inspection of Solids: Cross-Sections and Interferences", *ACM Computer Graphics (Proc. ACM Siggraph) Vol. 26, No. 2*, July 1992, pp. 353-360.
17. G. K. Wallace, "The JPEG still picture compression standard", *Communications of the ACM, 34(4)* 1991, 30-44.
18. D. Salomon, "Data Compression: The Complete Reference", *Springer*, 1997.
19. K. Sayood, "Introduction to Data Compression", *Morgan-Kaufmann*, San Francisco, CA, 1996.
20. H. Samet, "The Design and Analysis of Spatial Data Structures", *Addison-Wesley*, Reading, MA, 1990.
21. M. Attene, S. Biasotti and M. Spagnuolo, "Re-meshing Techniques for Topological Analysis", *Proceedings of Shape Modeling International*, May 2001, 142-151.
22. S. Biasotti, M. Mortara and M. Spagnuolo, "Surface Compression and Reconstruction using Reeb graphs and Shape Analysis", *Spring Conference on Computer Graphics*, 2000, pp. 174-185.
23. A. Gueziec and R. Hummel, "Exploiting triangulated surface extraction using tetrahedral decomposition", *IEEE Transactions on Visualization and Computer Graphics*, 1:328-342, 1995.
24. W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm", *Proceedings of ACM SIGGRAPH, Vol. 21 of Annual Conference Series*, Aug. 1987, pp. 163–169.
25. G. Nielson and B. Hamann, "The asymptotic decider: Resolving the ambiguity in marching cubes", *Proceedings of Conference of Visualization, IEEE*, 1991, pages 83-91.
26. J. Rossignac and D. Cardoze, "Matchmaker: Manifold BReps for non-manifold r-sets", *Proceedings of the ACM Symposium on Solid Modeling*, June 1999, pp. 31-41.
27. P. Cignoni, C. Rocchini and R. Scopigno, "Metro: Measuring Error on Simplified Surfaces", *Computer Graphics Forum*, Vol. 17, No. 2, June 1998, pp. 167–174.
28. "Edgebreaker compression and decompression software", *http://www.gvu.gatech.edu/~jarek/edgebreaker/eb/*