Coding with ASCII: compact, yet text-based 3D content

Martin Isenburg* Jack Snoeyink†
University of North Carolina at Chapel Hill
and
INRIA Sophia-Antipolis

Abstract

Because of the convenience of a text-based format 3D content is often published in form of a gzipped file that contains an ASCII description of the scene graph. While compressed image, audio, and video data is kept in separate binary files, polygon meshes and interpolators are usually included uncompressed into the ASCII description, as there is no widely-accepted standard for such data.

In this paper we show how to use compressed polygon meshes and compressed interpolators within a purely text-based format. Our scheme codes these data-heavy nodes as ASCII strings that compress well with standard gzip compression. Specifically we demonstrate the efficiency of this technique on a sparse scene composed of many small polygon meshes. While typical for 3D content on the Web, such scenes tend to be more difficult to compress.

1. Introduction

Most 3D content found in games, in virtual reality systems, in high-end visualization applications, and in low-polygon-count interactive web-viewers is represented by some kind of hierarchical scene graph structure. In order to save and distribute the 3D content, this scene graph structure has to be mapped to a file. The most popular way for doing this is to store the scene graph hierarchy as a correctly bracketed ASCII representation (e.g. VRML and its variants). The advantage of such a description is that it is very author friendly, in the sense that it remains meaningful to the human reader. A scene graph represented in a textual format can be viewed, understood, and modified with any text editor. Most importantly, anyone can do this, even without knowledge about the specific software package that generated the 3D content.

An author editing the scene graph in its ASCII representation will usually not modify all parts of the scene. Namely, he will not modify an individual polygon or an individual vertex position of a large polygon mesh. Typical editing operations involve changing the lights in scene, modifying the material properties of a surface, or scaling/translating/rotating of entire scene graph nodes. Being able to use cut&paste to merge or to exchange components of different scene graph files is very convenient.

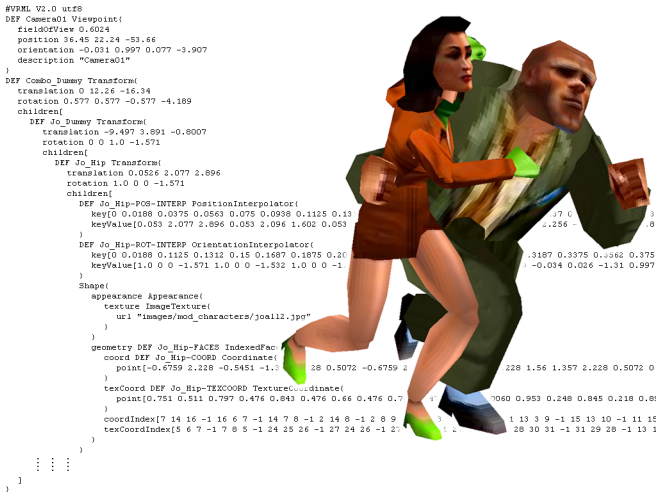

Figure 1. Our example VRML scene contains 33 indexed face sets, 5 position interpolators, and 27 orientation interpolators. The size of the gzipped VRML file is 29895 bytes. With ASCII coded indexed face sets this reduces to 18364 bytes. Additional ASCII coding of all interpolators brings it down to 12996 bytes.

However, a textual format has two disadvantages compared to a binary format: longer parse times and larger file sizes. Parsing an ASCII scene is more expensive in terms of computation because of the required string operations such as comparisons and conversions. Storing an ASCII scene is more expensive in terms of memory-usage because the textual representation of the data tends to be less compact. Most authors of 3D content seem to value the readability of a scene over the efficiency in parsing it. However, large file sizes are considered a serious problem because they hamper fast transmission of 3D content in a distributed bandwidth-limited environment such as the Internet.

*isenburg@cs.unc.edu http://www.cs.unc.edu/~isenburg/asciicoder
†snoeyink@cs.unc.edu

The size of a scene graph file is mainly determined by its *data-heavy* nodes. These are the nodes that contain texture images, audio clips, video sequences, polygon meshes, and the data for interpolation of positions or orientations. For the VRML scene in Figure 1, for example, these nodes constitute more than 96 percent of the total file size.

Widely accepted binary compression standards such as JPEG, GIF, and MPEG are available to compress image, audio, and video data. Software to read, save, create, and modify this data is plentiful and easy to use. Text-based scene graphs APIs like VRML support these compression standards. Since these formats are binary they cannot be part of the textual scene graph file. Instead they are stored as separate binary files that are referenced in the text file.

However, there are no compression standards for the other two data-heavy components typically found in a scene graph: polygon meshes and interpolation data. While recent research has aimed at developing compact representations for this kind of data, attempts to establish a standardized compressed format (e.g. MPEG-4, binary VRML, etc.) have not yet been successful.

The main obstacle seems to be the complex structure of the data. Audio data is always a sequence of numbers, image data is always a block of numbers and video data is always a block of numbers that changes over time. There are a few control parameters like sampling rate, dynamic range, or frame size, but these are easy to define. This is different for 3D data—there is no obvious format to agree upon. Users have different ideas of how, for example, a polygon mesh should be represented: Some need triangle strips and interleaved vertex arrays for fast rendering, others want high-level surface descriptions that enable collision detection, etc. Moreover a polygon mesh can contain a lot of different data besides positions and triangles. Or should I say polygons? There can be a layer of texture coordinates. Or two. Maybe three. There are normals (sometimes), colors (rarely), shading groups (one or multiple?), attached bones (a maximum of three?), and so on. Trying to establish a standard for compressed polygon meshes that everyone will accept and more importantly *use* is difficult.

Nevertheless, compressed 3D content has found its way onto the Web inside the proprietary file formats of various CAD and Web3D companies. Usually they compress the *entire* scene graph into a compact binary representation in the moment the content is ready to be published. They can afford to do so because they provide the tools needed to read, modify, and write this format to their customers. Since only their own software needs to be able to read the compressed format, they can tailor the compressor to their specific scene graph structure for maximal compression.

Since 1996 geometry compression for VRML has been an important item on the wish-list of the Web3D Consortium [2]. It was widely understood that a binary format would be required to allow compressed geometry. This lead to the formation of the Compressed Binary Format workgroup [18], which (a) created a binary format for all VRML nodes and (b) proposed new *compressed* versions of five data-heavy nodes that would only exist in binary. Despite excellent compression results, in the end the proposal was rejected. Some felt it was the sentiment against an unreadable binary format from the author-side and the reluctance to support two VRML formats from the browser company-side that influenced the decision.

We have recently proposed a mesh compression technique that does not require a binary file format. In [8] we show how to code textured polygon meshes as a sequence of ASCII symbols that compresses well with standard gzip compression. One benefit of this approach is that it eliminates the binary requirement. In order to add compressed polygon meshes to VRML (or now X3D) we no longer have to wait for a binary standard. The other benefit is that complete conformance between the ASCII version and the (eventual) binary version of a VRML scene would be possible. Translating back and forth between the two versions will not require invoking compression or decompression algorithms. Furthermore the same coding technique could be used to inflate a compressed node, no matter if it was stored in an ASCII VRML file or in a binary VRML file.

In this paper we (a) show that the concept of coding with ASCII can be extended to other data-heavy node types and (b) describe in detail the techniques we used to compress the scene shown in Figure 1. The main contents of this scene are described in Table 1, 2, and 3. They are *typical* for 3D scenes destined for instant playback in a Web-browser: many individual meshes each textured and of low-polygon count plus a lot of animation and control data.

All data-heavy nodes of this scene are replaced with ASCII coded versions that can remain inside the text-based file (see Figure 3). The intuition is that these parts of the scene are not read and edited anyways. And coding these parts with ASCII does not affect the ability to read and edit the other parts of the scene. This allows authors of 3D content to use compression without changing their work-flow: they can continue to publish gzipped ASCII scene files.

We describe proof-of-concept implementations of ASCII coding for three scene graph nodes: the indexed face set, the position interpolator, and the orientation interpolator node. However, the concept of ASCII coding is independent from the particular compression techniques used in this paper. On the contrary, many other compression algorithms could be adapted to produce ASCII instead of a binary bit-stream. Our particular choice comes from the additional requirement for extreme lightweight decoding algorithms. Our decoders have been developed to be used with the Shout3D scene graph API [14]. This API implements a plugin-less Web player for 3D content that downloads all required java

classes on demand. Therefore it is important that the additional java classes required for decoding are as small as possible, since they have to be transmitted as well. After all, reducing the amount of data that needs to be downloaded was the sole motivation for compression in the first place.

Our lightweight decoders can be distributed at minimal additional cost. The sizes of the java classes for decoding the indexed face set, the position interpolator, and the orientation interpolator node are 5420 bytes, 1578 bytes, and 1705 bytes respectively and only a total of 4924 bytes if included into a zipped archive of java class files.

2. Coding Indexed Face Sets

VRML and other scene graph APIs specify polygon meshes in form of indexed face set. In the scope of this paper we will be concerned with polygon meshes that have one (optional) layer of texture coordinates. The indexed face set representation of such a mesh contains two arrays of floats and two arrays of integers. Given a mesh with p positions, t texture coordinates, and f faces that have a total of c face corners, these arrays will contain the following:

- An array of $3p$ floats that specifies the x, y, and z coordinate for each of the p positions.
- An array of $2t$ floats that specifies the u and v coordinate for each of the t texture coordinates.
- An array of $c+f$ integers that specifies a position index for each corner of each face. The indices of the corners are listed in (usually) counterclockwise order around the face followed by a special value of -1 that acts as a face delimiter. Thus, the array contains c position indices and f face delimiters.
- An array of $c+f$ integers that specifies a texture coordinate index for each of the c face corners. The order on the corners follows that of the position index array and the face delimiters are also used the same way.

The excerpt of the VRML scene file shown in Figure 3 contains a typical example of an indexed face set.

The array of position indices and the array of texture coordinate indices can be encoded in a very compact manner without affecting the quality of the polygon mesh (e.g. lossless coding). Encoding the positions and texture coordinates, on the other hand, does affect the quality of the mesh. In order to efficiently code these floating-point values they are quantized to a user-specified number of bits, which introduces quantization error (e.g. lossy coding).

2.1. ASCII Coding of Position Indices

Many schemes were proposed to code the position indices of triangular [3, 16, 17, 4, 13, 1] or polygonal [6, 5, 11] meshes. These schemes do not code the position indices directly. Instead they code only the connectivity graph of the mesh and then change the order in which the positions are stored in the position array. The positions are arranged in the order in which their corresponding vertex is encountered during encoding and decoding of the connectivity graph.

This reduces the information needed for storing all position indices to whatever is required to code the connectivity graph of the mesh. Our ASCII coder uses the Face Fixer scheme [6] to code the connectivity graph, because it handles arbitrary polygon meshes, has a simple and lightweight implementation, and produces a symbol stream that easily maps into a compressable ASCII string.

The Face Fixer scheme encodes a polygonal connectivity graph as a sequence of labels R, L, S, E, $\mathrm{M}_{i,k,l}$, F_n, and H_n. The connectivity graph is decoded by processing the sequence of labels in *reverse*. For the details on the encoding and decoding process we refer the reader to the original references [6, 8]. The ASCII coding of the connectivity graph is any unique ASCII representation of the reversed label sequence. We choose a simple mapping from labels to white-space separated integer numbers for two reasons: On one hand it makes the conversion from the ASCII string to an array of integer numbers really simple (i.e. efficient conversion routines already exist). And on the other hand the decoder can use the integer value of a label directly for subsequent computation (i.e. as a counter).

2.2. ASCII Coding of TexCoord Indices

The indexed face set representation uses a texture coordinate index for every face corner. However, there is usually a strong correlation between position indices and texture coordinate indices. Namely there is either one or more texture coordinate index for every position index. All proposed methods for encoding texture coordinate indices [4, 15, 6, 7] exploit this correlation.

Around every vertex of the connectivity graph is an ordered cycle of face corners. We say a corner is a *smooth* corner if it uses the same texture coordinate index as the (counterclockwise) previous corner, otherwise we call it a *crease* corner. A *smooth* vertex has only smooth corners; it has one texture coordinate index that is used by all corners. A *crease* or *corner* vertex has two or more crease corners; it has two or more different texture coordinate indices each used by a set of adjacent corners.

There is a one-to-one mapping from smooth vertices and crease corners to texture coordinate indices. In order to specify all texture coordinate indices it is sufficient to code this mapping and then re-order the texture coordinates appropriately. In [6] we suggested the use of *vertex bits* and *corner bits* to code this mapping. One bit per vertex is needed to distinguish smooth vertices from crease and corner vertices. In addition we need for all corners around a

indexed face sets	content				characteristic			quantization		code words
	positions	posindices	texcoords	texindices	polygons	holes	components	pos_bits	tex_bits	
Jo_Hip	17	104	27	104	26	1	1	6	5	95
Jo_Leg1_R, Jo_Leg1_L	20	144	20	–	36	–	1	7	6	57
Jo_Leg2_R, Jo_Leg2_L	23	168	23	–	42	–	1	6	6	66
Jo_Foot_R, Jo_Foot_L	22	124	22	–	31	1	1	5	5	56
Jo_Chest	64	496	88	496	124	–	1	6	6	375
Jo_Arm1_R, Jo_Arm1_L	26	192	31	192	48	–	1	6	5	107
Jo_Arm2_R, Jo_Arm2_L	13	72	18	72	48	1	1	6	6	53
Jo_Hand_R, Jo_Hand_L	13	88	30	88	22	–	1	5	5	115
Jo_Hair	148	936	164	936	234	5	4	6	8	585
Os_Hip	18	128	26	128	36	–	1	6	6	112
Os_Leg1_R, Os_Leg1_L	26	192	26	–	48	–	1	7	6	75
Os_Leg2_R, Os_Leg2_L	20	144	20	–	36	–	1	6	6	57
Os_Foot_R, Os_Foot_L	22	124	22	–	31	1	1	6	5	56
Os_Torso	51	392	68	392	98	–	1	8	8	299
Os_Arm1_R, Os_Arm1_L	26	192	26	–	48	–	1	7	6	75
Os_Arm2_R, Os_Arm2_L	19	120	19	–	30	1	1	6	5	52
Os_Hand_R, Os_Hand_L	13	88	30	88	22	–	1	5	5	115
Os_Head	68	464	74	464	116	1	1	7	7	289
Os_Brow	18	96	18	–	24	1	1	10	6	45
Os_Jaw	96	576	113	576	144	4	4	9	8	434
Os_Eyes	12	40	12	–	40	1	1	12	6	25

Table 1. This table lists various statistics for the 33 indexed face sets of the example scene: Their content, which is number of positions, of position indices, of texture coordinates, and of texture coordinate indices. Their characteristic, which is number of polygons, of holes, and of connected components. Their quantization, which is the number of bits used to quantize positions and texture coordinates. And finally the total number of code words that encode all indices is reported. Indexed face set without texture coordinate indices have a per-vertex texture coordinate mapping.

crease or corner vertex one bit to distinguish smooth corners from crease corners. For ASCII coding we simply map vertex bits and corner bits to integer values that are already frequently used for other things. This reduces the entropy of the ASCII stream for better gzip compression results.

2.3. ASCII Coding of Positions and TexCoords

The ASCII format of an indexed face set specifies the x, y, and z coordinate of each position and the u and v component of each texture coordinate as an ASCII representation of a floating point number. Although it would be possible to represent them at the full precision of an IEEE 32 bit floating point number, in practice one finds fixed point representation that use between 3 and 6 decimal digits.

The common approach for coding positions and texture coordinates first quantizes the floating-point numbers uniformly and then applies some form of predictive coding. Quantization with k bits of precision maps each floating-point value to an integer value between 0 and $2^k - 1$. Subsequent predictive coding reduces the variation and thereby the entropy of the resulting sequence of k bit numbers. Rather than specifying each position individually, previously decoded information is used to predict the next coordinate and only a correcting term is stored. The simplest method predicts the next position as the last position and was suggested by Deering [3]. This is also known as delta coding. Other popular methods are the spanning tree predictor by Taubin and Rossignac [16] and the parallelogram predictor by Touma and Gotsman [17], which give better compression.

For the sake of simplicity our prototype implements delta coding on the two arrays of quantized positions and texture coordinates. The order in which positions and texture coordinates are stored in the array makes a difference for the delta coder. Ideally subsequent entries are close to each other so that the correction deltas are small. This will be true for the positions, since neighboring position entries usually correspond to vertices that are connected by an edge. Unfortunately this will not always be true for neighboring texture coordinate entries. Two texture coordinates used around a crease vertex are stored one after the other in the texture coordinate array. However, they can address a completely different location in the texture image.

3. Coding Position Interpolators

An interpolator node is used to drive an animation. Typically it receives a time (e.g. a fractional value between 0.0 and 1.0) from a time sensor node that specifies the current time of the animation cycle. The interpolator then determines the values corresponding to that time for the parameters it interpolates. Usually these are then routed to the appropriate fields of the nodes it animates.

Interpolators are defined by a list of k key frames, each of which consists of a key and a key value. The keys specify the times at which the interpolated parameter should have the corresponding key values. The values at all other times are computed by (usually) linear interpolation between neighboring key values.

The position interpolator node is used to move objects along a path in 3D. The interpolated parameter is a position

and each key value is a 3D coordinate. The position interpolator node contains two arrays of floats. For an animation with k key frames, these arrays will contain the following:

- An array of k strict monotonically increasing floats between 0.0 and 1.0 that specify the k keys.
- An array of $3k$ floats containing the x, y, and z coordinate for the interpolated positions that specify the k key values.

The array of keys can often be quantized in a lossless manner. Common modeling packages use an integer scale that counts in frames per second to time an animation. On export the times are converted into the VRML-style floating-point range 0.0 to 1.0. Often the original timing can be recovered. In our example scene the original integer scale was 160, which can easily be reconstructed from the spacing of subsequent keys. For all interpolators we report the minimal and maximal increase between keys together with the number of key frames in Tables 2 and 3.

We convert the keys back to their original integer scale and apply delta coding with delta prediction. We predict the keys to be evenly spaced; hence we always predict the next delta as the remaining time divided by the number of remaining keys. We then record the difference between the predicted and the actual delta. An interesting aside: converting the keys back to their original integer scale did not only allow lossless coding, in our case it actually *repaired* them. This is not a feature of our method, but points at a weakness in the specification of VRML. The requirement for keys to be on a floating-point scale invites a systematic rounding error when exporting animations to VRML.

The array of key values (e.g. positions) is uniformly quantized and then delta coded. Quantization is uniform based on the largest range using a user-defined number of bits. The ranges for the x, y, and z coordinate of each interpolated position are reported in Table 2.

position	# key	keys		positions		
interpolators	frames	min_incr	max_incr	range_x	range_y	range_z
Jo_Dummy	22	0.0063	0.1125	6.3690	–	0.6066
Jo_Hip	53	0.0063	0.0376	–	2.0616	5.8100
Jo_Hair	7	0.0063	0.6000	0.0942	0.0655	0.1270
Os_Dummy	22	0.0063	0.1125	6.3680	–	0.6065
Os_Hip	53	0.0063	0.0376	–	1.8549	5.8430

Table 2. This table lists for all position interpolators the number of key frames, the minimal and maximal increment of subsequent keys, and maximal range of x, y, and z component for the interpolated position.

4. Coding Orientation Interpolators

The orientation interpolator node is used to rotate an object in 3D. The interpolated parameter is an orientation and each key value is a 3D rotation axis and a rotation angle. The orientation interpolator node contains two arrays of floats. For an animation with k key frames, these arrays will contain the following:

- An array of k strict monotonically increasing floats between 0.0 and 1.0 that specify the k keys.
- An array of $4k$ floats containing the x, y, and z coordinate of the rotation axis and the rotation angle α for the interpolated orientations that specify the k key values.

The keys are coded the same way it was done for position interpolators. The rotation axii and the rotation angles are quantized and delta coded. However, they are treated differently. A rotation axis is a normalized direction vector. How to quantize such vectors has been treated extensively for the compression of shading normals. The most complete and sophisticated scheme has been proposed by Deering [3], but its implementation is quite complicated. He exploits the symmetries on the unit sphere to define an almost perfectly uniform sampling of all normal directions. Another approach reported Taubin et al. [15] maps normals to a discrete set of normals using an index of $3 + 2n$ bits. Each such index addresses a single triangle of an n times recursively subdivided octagon that represents a normal direction. While conceptually simple, this addressing scheme does not naturally include the most common normals (like those in unit direction, for example) and it is not obvious how to extend the concept in an elegant way. Furthermore the proposed indexing scheme does not allow efficient delta coding, as neighboring normals often have very different indices. For lightweight decoding we require a simpler compression scheme:

One approach would be to quantize and delta-code only two of the three components, for example the y and z coordinate, of every rotation axis. The absolute value of the x coordinate could then be computed as the one that completes the unit length and only its sign would need to be stored. Its simple implementation makes this quantization scheme a tempting choice, but this simplicity comes with some serious drawbacks: It results in a highly non-uniform quantization of all directions. For small values of y and z the directions are sampled dense, but as $y^2 + z^2$ approach 1 the sampling becomes very sparse.

Imagine a regular distribution of directions over the unit sphere. The above scheme projects them along the x-axis into the y-z plane where they are uniformly sampled. This sampling will have a fairly uniform density near the poles but as we come closer to the equator it becomes sparser and increasingly non-uniform. On or very near the equator we can expect numerical instabilities for this method. However, compared to other methods [3, 15] it is much simpler to implement and there is a way to fix the main problem: We simply remove the equator (or plant new poles).

Depending on the direction to be quantized we choose one of three projections: along the x-axis, the y-axis, and the z-axis. We choose the projection axis with which the considered direction encloses the smallest angle. Intuitively this selects the component with the largest absolute value as the one, which is computed from the other two. The chosen projection axis has to be stored as well. We combine the information about the chosen projection axis and about the sign of the computed component into one number between 0 and 5. The other two components are uniformly quantized and delta-coded.

The rotation angles are also uniformly quantized and delta-coded. The quantization level is chosen such that the quantization error does not succeed a user-defined tolerance. In our example scene the maximal quantization error was set to one degree for both, the axis and the angle, which seemed sufficient to avoid (almost) any visual impact.

orientation interpolators	# key frames	keys		orientation			
		min_incr	max_incr	# x	# y	# z	range_α
Jo_Hip	35	0.0063	0.2250	35	–	–	0.437
Jo_Leg1_R	33	0.0063	0.1125	33	–	–	1.108
Jo_Leg2_R	33	0.0063	0.1125	33	–	–	2.010
Jo_Foot_R	25	0.0063	0.2437	25	–	–	0.842
Jo_Leg1_L	39	0.0063	0.0563	39	–	–	1.533
Jo_Leg2_L	38	0.0063	0.0563	38	–	–	4.092
Jo_Foot_L	30	0.0063	0.1125	30	–	–	0.745
Jo_Chest	55	0.0063	0.0188	6	49	–	2.317
Jo_Arm1_L	17	0.0063	0.7313	–	17	–	1.403
Jo_Arm2_L	19	0.0063	0.6000	19	–	–	0.682
Jo_Arm1_R	55	0.0063	0.0188	19	10	26	1.441
Jo_Arm2_R	49	0.0063	0.1125	49	–	–	2.352
Jo_Hair	15	0.0063	0.2250	5	10	–	0.667
Os_Hip	35	0.0063	0.2250	34	–	–	0.437
Os_Leg1_L	37	0.0063	0.1125	37	–	–	1.655
Os_Leg2_L	51	0.0063	0.0563	47	4	–	4.048
Os_Foot_L	27	0.0063	0.2437	21	6	–	0.627
Os_Leg1_R	48	0.0063	0.0563	48	–	–	1.425
Os_Leg2_R	54	0.0063	0.0375	51	3	–	6.125
Os_Foot_R	35	0.0063	0.1125	25	10	–	0.754
Os_Torso	55	0.0063	0.0188	26	29	1	1.460
Os_Arm1_L	55	0.0063	0.0188	41	6	8	1.220
Os_Arm2_L	53	0.0063	0.0375	53	–	–	1.975
Os_Arm1_R	26	0.0063	0.3563	–	26	–	0.483
Os_Arm2_R	53	0.0063	0.4875	–	–	14	0.788
Os_Hand_R	26	0.0063	0.3563	–	7	19	0.692
Os_Head	29	0.0063	0.1125	27	2	–	0.687

Table 3. This table lists for all orientation interpolators the number of key frames, the minimal and maximal increment of subsequent keys, how many times x, y, and z projections were used to quantize the rotation axii and the maximal range of the rotation angle α over the course of the entire animation.

5. Implementation and Results

We have implemented a set of extremely lightweight decoders based on the coding schemes described in this paper. Our prototype implementation uses the Shout3D pure java API [14] and extends the nodes *IndexedFaceSet*, *PositionInterpolator*, and *Orientation Interpolator* to the nodes *CodedIndexedFaceSet*, *CodedPositionInterpolator*, and *CodedOrientationInterpolator* respectively. This

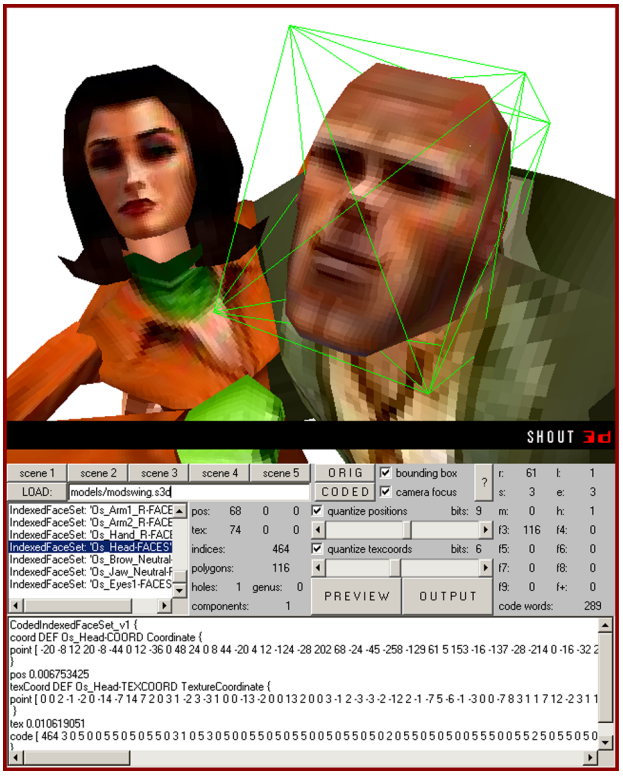

Figure 2. This is a screen-shot of the ASCII coder. It was implemented in form of a Web applet using Shout3D [14] and can be found on our project page [9]. The example scene used in this paper was ASCII coded with this applet.

way they can be used as custom nodes of the VRML-based Shout3D scene graph, which gives us the ability to download the required decoder classes *on-demand*. The nodes automatically decode themselves once loading has completed. The java class files that implement the decoders have a size of only 5420 bytes, 1578 bytes, and 1705 bytes respectively. If the decoders are included into a zipped archive of java class files their sizes reduce to roughly half this.

We ASCII coded the VRML scene shown in Figure 1, which contains 33 indexed face sets, 5 position interpolators, and 27 orientation interpolators. The size of the gzipped original VRML file is 29895 bytes. With ASCII coded indexed face sets this reduces to 18364 bytes. Additional ASCII coding of all interpolators brings it down to 12996 bytes. The size of the scene file is reduced by more than 50 percent. Even when the download of the required decoding software (4924 bytes) is taken into account the compression is significant. The time consumed for decoding is small compared to the time needed to download and parse the scene.

Coding the indexed face resulted in the biggest gain and can always be done without visible loss in quality. Coding the interpolators gave a smaller gain and the effects of quantization seem slightly noticeable. While we limited

the quantization error of each individual orientation interpolator to maximally one degree, the hierarchical structure of the interpolators probably requires a global quantization strategy. The interpolator for Jo's hand, for example, was quantized with the same precision as the interpolator of her hip. Yet, the first influences only the hand, while the other influences the entire character. In fact the final position and orientation of the hand is result of the combined animations of hand, lower arm, upper arm, chest, and hip. However, correct quantization of deep animation hierarchies is not topic of this paper.

In [8] we reported compression results for ASCII coding simple scenes each of which contained one large textured polygon model. On such scenes we were able to achieve compression by a factor of six and more. Here we demonstrated that ASCII coding is also effective for *sparse* scenes composed of many small polygon meshes (see Table 1), which are usually considered difficult to compress. You can find an interactive java implementation of the encoder and the decoder together with several example scenes (including the one used in this paper) on our project Web page [9].

6. Current Work

There is still a significant gap between the compression rates that are achievable with coders that compress into binary bit-streams and the ASCII coders we have presented here. The main reason is that binary coders use entropy coding to squeeze the produced symbol streams into as few bits as possible. Arithmetic coding, for example, will always outperform gzip coding, because it gives optimal compression in respect to the information entropy of a symbol sequence [12].

We can combine the advantages of arithmetic coding with that of non-binary coding by letting the arithmetic coder produce an ASCII string of zeros and ones instead of a binary bit-stream. Standard gzip coding is able to aggressively compress the resulting ASCII string of zeros and ones, as it only contains two different symbols. We have implemented an arithmetic coder that compresses to from ASCII and initial results are promising. The disadvantage of this approach is the additional computation and the additional java classes required for the arithmetic decoding.

However, this additional effort may be well worth it. Recently we have developed a better compression scheme for coding polygonal connectivity. Our degree duality coder [5] achieves the best compression rates for polygon mesh connectivity reported so far, but requires the use of a context-based arithmetic coder to achieve this. We have a proof-of-concept java implementation of this connectivity coder that already demonstrates the ASCII producing arithmetic coder we mentioned above. This interactive demo can also be found on the respective project Web page [10].

These results suggest that it will be possible to achieve compression rates that approach those of binary coders, while retaining the advantages of a text-based approach. Furthermore, should one day a binary standard for VRML (or X3D) be accepted to co-exist with the ASCII standard our technique will allow complete conformance between the two versions. Translating back and forth between the two versions will not require invoking compression or decompression. The same coding technique can then be used to inflate a compressed node, no matter if it was stored in an ASCII VRML file or in a binary VRML file. In this paper we do not argue against a binary version of VRML. A binary format will reduce parse time and also allow storing compressed nodes even more compact. In this paper we argue for doing compression today without waiting for its binary specification.

7. Acknowledgments

We thank Paul Isaacs for the discussion at SIGGRAPH'01, which gave us the idea of compression for text-based scenes. We also thank Shout3D for the permission to use their VRML scene in our experiments. This work was partly supported by the ARC-TeleGeo grant at INRIA Sophia-Antipolis.

References

[1] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. In *Eurographics'01*, pages 480–489, 2001.

[2] The Web3D Consortium. http://www.web3d.org/

[3] M. Deering. Geometry compression. In *SIGGRAPH'95 Conference Proceedings*, pages 13–20, 1995.

[4] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98*, pages 133–140, 1998.

[5] M. Isenburg. Compressing polygon mesh connectivity with degree duality prediction. *to appear in Graphics Interface 2002*.

[6] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. In *SIGGRAPH'00*, pages 263–270, 2000.

[7] M. Isenburg and J. Snoeyink. Compressing the property mapping of polygon meshes. In *Pacific Graphics'01*, pages 4–11, 2001.

[8] M. Isenburg and J. Snoeyink. Compressing polygon meshes as compressable ASCII. In *Web3D Symposium'02*, pages 1–10, 2002.

[9] http://www.cs.unc.edu/~isenburg/asciicoder/

[10] http://www.cs.unc.edu/~isenburg/degreedualitycoder/

[11] A. Khodakovsky, P. Alliez, M. Desbrun, and P. Schroeder. Near-optimal connectivity encoding of 2-manifold polygon meshes. *to appear in Graphical Models 2002*.

[12] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Trans. on Information Systems*, 16(3):256–294, 1998.

[13] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. on Vis. and Comp. Graph.*, 5(1):47–61, 1999.

[14] Shout3D. http://www.shout3d.com/

[15] G. Taubin, W. Horn, F. Lazarus, and J. Rossignac. Geometry coding and VRML. *Proceedings of the IEEE*, 86(6):1228–1243, 1998.

[16] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Trans. on Graphics*, 17(2):84–115, 1998.

[17] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface'98 Conference Proceedings*, pages 26–34, 1998.

[18] Compressed Binary Format Workgroup. http://www.web3d.org/workinggroups/vrml-cbf/cbfwg.html

original scene

```
#VRML V2.0 utf8
...
...
  children[
    DEF Jo_Hip-POS-INTERP PositionInterpolator {
      key[0 0.0188 0.0375 0.0563 0.075 0.0938 0.1125 0.1312 0.15 ...  0.8812 0.9 0.9187 0.9375 0.9562 0.975 0.9937 1.0]
      keyValue[0.053 2.077 2.896 0.053 2.096 1.602 0.053 2.116 ...  0.053 2.094 1.44 0.053 2.081 2.799 0.053 2.077 2.896]
    }
    DEF Jo_Hip-ROT-INTERP OrientationInterpolator {
      key[0 0.0188 0.1125 0.1312 0.15 0.1687 0.1875 0.2062 0.225 ...  0.5813 0.6 0.6187 0.6375 0.8625 0.8812 0.9937 1.0]
      keyValue[1.0 0 0 -1.571 1.0 0 0 -1.532 1.0 0 0 -1.335 ...  0 -1.396 1.0 0 0 -1.405 1.0 0 0 -1.562 1.0 0 0 -1.571]
    }
    Shape {
      appearance Appearance {
        texture ImageTexture {
          url "images/mod_characters/joall2.jpg"
        }
      }
      geometry DEF Jo_Hip-FACES IndexedFaceSet {
        coord DEF Jo_Hip-COORD Coordinate {
          point [-0.6759 2.228 -0.5451 -1.354 2.228 0.5072 ...  0.5863 0.1099 0.966 0.3132 -1.315 -0.966 0.3132 -1.315]
        }
        texCoord DEF Jo_Hip-TEXCOORD TextureCoordinate {
          point [0.751 0.511 0.797 0.476 0.843 0.476 0.66 ...  0.326 0.43 0.486 0.0040 0.463 0.247 0.125 0.43 0.392 0.43]
        }
        coordIndex [7 14 16 -1 16 6 7 -1 14 7 8 -1 2 14 8 -1 ...  -1 1 14 2 -1 4 15 5 -1 5 15 16 -1 16 1 0 -1 5 16 0 -1]
        texCoordIndex [5 6 7 -1 7 8 5 -1 24 25 26 -1 27 24 26 ...  24 27 -1 17 13 18 -1 18 13 7 -1 7 15 16 -1 18 7 16 -1]
      }
    }
...
...
```

ASCII coded scene

```
#VRML V2.0 utf8
...
...
  children[
    DEF Jo_Hip-POS-INTERP CodedPositionInterpolator {
      key[0.0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 3 1 1 1 1 1 1 1 1 1.0]
      ks 160
      keyValue[0.053 2.077 2.896 0 1 -57 0 1 -55 0 1 -28 0 5 4 0 24 ...  0 -1 -33 0 0 -11 0 -1 23 0 0 66 0 -1 60 0 0 4]
      pos 0.022784313
    }
    DEF Jo_Hip-ROT-INTERP CodedOrientationInterpolator {
      key[0.0 -1 11 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -2 -2 -3 -3 -3 -3 -4 -4 -5 -5 -6 -7 -9 22 -4 9 1.0]
      ks 160
      keyValue[0 0 0 -1.571 0 0 0 1 0 0 0 6 0 0 0 1 0 -2 1 -1 0 -1 ...  -1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -5 0 0 0 0]
      ob 5
    }
    Shape {
      appearance Appearance {
        texture ImageTexture {
          url "images/mod_characters/joall2.jpg"
        }
      }
      geometry DEF Jo_Hip-FACES CodedIndexedFaceSet {
        coord DEF Jo_Hip-COORD Coordinate {
          point [-9 -10 14 -9 10 -14 -5 52 0 29 0 0 17 0 20 -3 -30 ...  0 22 40 0 0 -7 -26 -3 -26 0 0 0.966 0.3132 -1.315]
        }
        pos 0.07298413
        texCoord DEF Jo_Hip-TEXCOORD TextureCoordinate {
          point [-5 -1 -11 0 -3 0 20 0 -4 0 -1 1 -1 14 7 0 -27 0 ...  -5 6 3 0 -3 8 2 0 -4 0 -10 0 8 -7 -6 0 0.845 0.218]
        }
        tex 0.030451613
        code [104 3 0 5 0 5 0 5 0 0 5 5 0 5 5 0 0 5 0 5 0 5 5 5 5 ...  5 5 0 0 5 0 5 5 5 0 5 0 0 5 5 5 0 5 5 0 5 0 0 0 2]
      }
    }
...
...
```

Figure 3. This figures shows a small excerpt of our example VRML scene before (top) and after (bottom) ASCII coding. The IndexedFaceSet, the PositionInterpolator, and the OrientationInterpolator node have been replaced with their ASCII coded counterpart: the CodedIndexedFaceSet, the CodedPositionInterpolator, and the CodedOrientationInterpolator node.