Binary Compression Rates for ASCII Formats

Martin Isenburg*
University of North Carolina
at Chapel Hill
isenburg@cs.unc.edu

Jack Snoeyink
University of North Carolina
at Chapel Hill
snoeyink@cs.unc.edu

ABSTRACT

Geometry compression for VRML has been an important item on the wish-list of the Web3D Consortium since 1996. It was widely understood that a binary format would be required to allow compressed geometry, which explains why there is still no geometry compression in VRML. We demonstrate here a compression technique that does not require a binary format and that is able to achieve bit-rates that are within 1 to 2 percent of those of a binary benchmark coder.

Furthermore, our technique will allow complete conformance between the current ASCII standard and the future binary standard of VRML (or X3D). Translating between the two will not require to invoke complex compression or decompression schemes. Compressed nodes have an ASCII as well as a binary representation and conversion from one to the other is a simple symbolic mapping. The same decompression algorithm can be used to inflate a compressed node, no matter whether it was stored in ASCII or in binary.

We do not argue against a binary format for VRML. A binary format will reduce parse time and might store a scene even more compactly. We argue to support geometry compression now ... without waiting for a binary specification.

Categories and Subject Descriptors

I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*surface, solid, and object representations*

Keywords

Mesh compression, arithmetic coding, base64, textual formats, ASCII, binary, VRML, X3D, BIFS, MPEG-4, XMT.

1. INTRODUCTION

The most popular way of distributing 3D content on the Web is in form of a textual representation of the scene such as VRML and its variants. The advantage of such a description is that it is very author friendly in the sense of being meaningful to the human reader. A scene 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.

*http://www.cs.unc.edu/~isenburg/ac

One disadvantage of an ASCII format is that scene files can become too large for efficient Web transmission when the scene contains many and/or detailed polygon meshes. Although the scene files are usually compressed with gzip compression, such general purpose coders do not come close to the compression rates that can be achieved with a dedicated geometry coder.

For image, audio, and video data an on-going standardization process has produced binary compression formats such as JPEG, GIF, and MPEG that are widely accepted. Software to read, create, save, and modify these formats is plentiful and easy to use. Several attempts to develop a similar standard for compressed polygonal data have so far been without success.

Geometry compression for VRML has been an important item on the wish-list of the Web3D Consortium since 1996. 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, which (a) created a binary format for all existing VRML nodes and (b) proposed new *compressed* versions of five data-heavy nodes that would only exist in binary. Despite excellent compression results [12, 11], 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 [5] we show how to code textured polygon meshes as a sequence of ASCII symbols that compresses well with standard gzip compression. The main benefit of such an approach is that it eliminates the binary requirement. In order to add compressed geometry to VRML (or now X3D) one no longer has to wait for a binary standard. Another benefit is that it makes it possible to have complete conformance between the current ASCII standard and a future binary standard of VRML—including the compressed nodes.

However, there was still a significant gap between the compression rates of the ASCII coder proposed in [5] and that of binary state-of-the-art geometry coders [13, 8]. The reason for this is that binary coders use entropy coding to squeeze the produced symbol stream into as few bits as possible. Arithmetic coding, for example, outperforms gzip coding because it gives optimal compression in respect to the (context-based) information entropy of a symbol sequence [9].

The ASCII coder proposed in [5] was specifically designed to have an extremely light-weight decoding algorithm so that it could be used with Shout3D's [10] pure java API—a plugin-less Web3D player that downloads all required classes on demand. Although using this scheme would allow com-

plete conformance between ASCII and binary VRML, it would mean to compromise the binary compression rates.

In an attempt to overcome this, we have combined the advantages of arithmetic and of non-binary coding: Our arithmetic coder stores the sequence of bits it produces as a stream of "safe" ASCII characters instead of a stream of binary bytes. This is by no means a new idea. Schemes such as *uuencode* or *base64* are standard techniques for mapping binary files to ASCII so that they can be handled by the purely text-based transfer protocols used by SMTP email and usenet groups. However, the conversion to ASCII increases the size of their files by 33 to 35 percent.

This in different for us because our files do not need to be ASCII at transmission time. Both, the protocol used for transmitting VRML content and also the browsers used for viewing it can handle binary data formats such as JPEG, GIF, MPEG, and most importantly for us—gzipped ASCII files. The key insight for achieving binary compression rates in ASCII formats is the following: the redundancy that is added when mapping the sequence of bits produced by an arithmetic coder into "safe" ASCII characters is almost completely removed by subsequent gzip compression.

We have implemented a context-based arithmetic coder that writes and reads its bits to and from the *base64* set of ASCII characters. Using this coder with our compression schemes for mesh connectivity [1], geometry [2], property mappings [4], and property values [6] results in ASCII files that are more compact than the gzipped ASCII files of the scheme we proposed last year [5]. If these files are then compressed with gzip we achieve bit-rates that are within 1 or 2 percent of those of a binary benchmark coder [8].

The paper is organized as follows: The next section discusses text-based and binary formats and how they relate to compression. Section 3 investigates how the bit sequence of an arithmetic coder can be efficiently stored as ASCII. Section 4 introduces the benchmark coder we used in our experiments and explains why context-based arithmetic coding is crucial to its success. Section 5 discusses the experimental results of our proof-of-concept implementation and concludes with a short summary of our results.

2. FILE FORMATS AND COMPRESSION

In theory the following three things are completely independent from each other.

1. an ASCII format for a VRML scene
2. a binary format for a VRML scene
3. a compressed format for some nodes of a VRML scene

In practice the feasibility of the third was thought to depend on the availability of the second, because it would not be integratable into the first. We have shown in [5] that this is not true and try here to strengthen this claim.

A binary format is a different representation for the entire VRML node set. There should exist a bijective mapping between the current ASCII format and the future binary format. This mapping would be applied to an entire scene and should neither affect the visual nor the functional quality of the scene. The only differences between the two would be file size, parse time, (un-)readability in an text-editor, and maybe some byte-format related issues with transmission (e.g. big5, utf-8, ...). The binary format might be more compact than the ASCII format, but that does not necessarily mean that the resulting binary files will always be smaller than the corresponding gzipped ASCII files.

A compressed format is a more compact representation for the data contained in a VRML node. The author of the scene selectively applies a compression process to some data-heavy nodes in the scene while trading off between its visual impact and the achieved reduction in file size. Compression can affect the contents of a node both, visually (e.g. loss of precision) and also functionally (e.g. reordering of arrays).

The trade-off between visual quality and file size can not only be balanced by compression but also by simplification. These are conceptually two different things. There is compression, which—speaking in terms of polygonal geometry—does not change the polygon, position, and texture coordinate count of a geometry node, but may affect its visual quality by quantization of position and texture coordinates. And there is simplification, which changes a geometry node by iteratively reducing its polygon, position, and texture coordinate count in a way that tries to preserve its appearance until, for example, the target polygon budget was reached. The simplified result may subsequently be compressed. Here we are only concerned with compression.

In the rejected proposal of the Compressed Binary Format workgroup compressed nodes were only supposed to exist in the binary format. We believe that this is a disadvantage. Imagine some scene in binary format that contains compressed nodes. In order to quickly change, for example, the lights in the scene, you use the imaginary "bin2asc" converter and then open the generated ASCII version of this scene in your favorite text editor. After making the desired changes, you save the modified scene and call the imaginary "asc2bin" converter to get back to the binary representation. But what happened to the compressed nodes?

The compressed nodes could have been inflated into their uncompressed counterparts by the "bin2asc" converter. But when going back to binary, how would the "asc2bin" converter know (a) which nodes used to be compressed and (b) with which compression parameters (e.g. number of bits) they had originally been compressed. Furthermore, depending on the used compression and decompression algorithms, the conversion of compressed nodes could be significantly more complex than that of uncompressed nodes. In addition, the ASCII version of a scene containing compressed nodes would be always *much* larger than the binary version.

However this dilemma was solved, compressed nodes would always be second-class citizens that are not allowed to live in both (e.g. ASCII and binary) worlds. The mechanism proposed in this paper will assure them a life in both.

3. STORING BITS AS ASCII

Sequences of bits that are output of an arithmetic coder are best represented in a binary format. The bits of such a sequence have no correlation that could be exploited to compress them further. The most compact way to store them, is to pack groups of eight bits into one byte. In order to efficiently represent this bit sequence with ASCII characters there are four goals to consider.

1. The ASCII should be safe and portable, able to be used in all editors and applications that handle text.
2. The conversion between a bit sequence and its ASCII representation should be computationally simple.

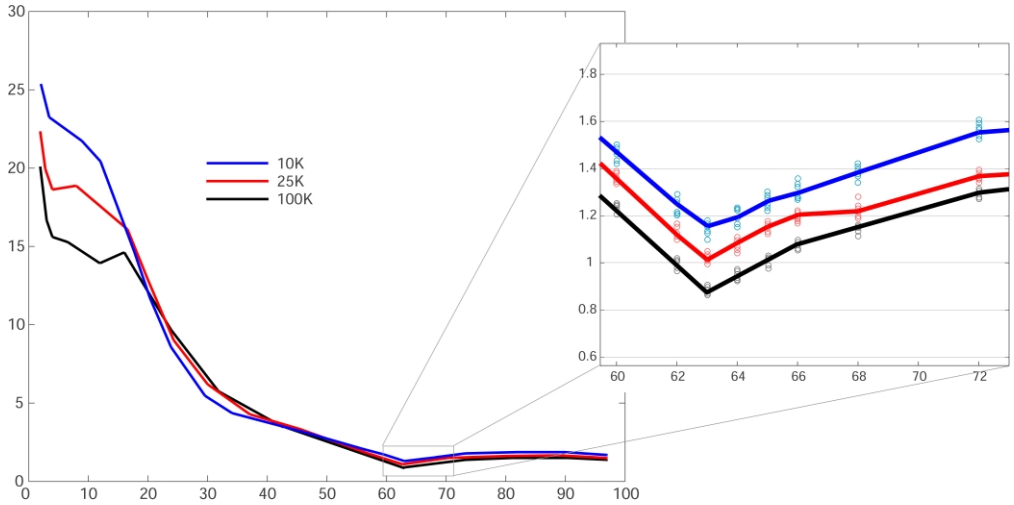


Figure 1: The size increase in percent of files that contain random bits coded as ASCII base$_n$ and compressed with "gzip -9" over files that contain the same bits in plain binary is plotted as a function of n for 10, 25, and 100 KB. Lines indicate averages of 8 separate test, each marked by "o" signs. The enlarged portion shows that base$_{63}$ gives the best performance.

3. The representation of the bits in ASCII should be as compact as possible; in particular, it should not be too much larger than their representation in binary.

4. The resulting ASCII should compress well using standard gzip compression; in particular, the size of the gzipped ASCII file should be near that of the binary.

A natural choice for satisfying the first goal are those ASCII formats that have long been used for transferring binary files in text-based transmission systems. There are *uuencode*, which has its origin in usenet groups, *base64*, which is the preferred mime-type for binary email attachments, and *hexbin*, which is mostly used by applications on Mac computers. All three formats encode binary data basically by mapping groups of six bits to a set of 64 "safe" ASCII characters. In general such mappings can use from as little as 2 all the way up to the entire set of 96 printable ASCII characters. In the following we will refer to a mapping that uses n characters as a base$_n$ mapping.

Any base that is a power two satisfies the second goal because then $n = 2^k$ and there exists a bijective mapping between the value of a group of k bits and the ASCII characters of the base. Going back and forth between the two could then be implemented with simple look-up tables.

For the third goal we calculate the expansion factor. If we map the sequence of bits into a selection of n different ASCII characters, we get an expansion factor of $8/\log_2(n)$. The expansion factor decreases as n becomes larger. For base$_{32}$, the expansion factor is 60%; for base$_{64}$, the expansion factor is 33%; if we used all 96 printable ASCII characters and encode in base$_{96}$, then the expansion factor is 21.5%.

For the fourth goal, we generated random bit sequences whose binary representation have sizes of 10, 25, and 100 Kilobytes and encoded them using all ASCII bases ranging from base$_2$ to base$_{96}$. Subsequently we compressed the resulting ASCII files using "gzip –9". Figure 1 plots the relative increase in size of the gzipped ASCII file over the corresponding binary file as a function of the number of different ASCII characters in the base. The resulting curves give us a clear winner: the actual minimum is at base$_{63}$ with base$_{64}$ coming in second place.

Overall we found that base$_{64}$ is the best choice. It is already an accepted standard, it is trivial to implement, its expansion factor is only 33% with 21.5% being the best possible, and, most importantly, after gzip compression the increase in file size over binary is only about 1%.

4. BENCHMARK MESH COMPRESSION

For several years now, the coder by Touma and Gotsman [13] has been the most widely accepted benchmark in mesh compression. Their connectivity coder tends to give the best bit-rates for compressing triangular connectivity. Also their geometry coder delivers competitive compression rates. While the simplicity of their scheme is one reason for its popularity, the achieved bit-rates are so good that it continues to be the main benchmark in geometry compression.

However, we decided not to implement this benchmark coder for our experiments. On one hand it was designed to compress purely triangular meshes and our test meshes contain many non-triangular polygons. On the other hand it was not designed with support for textures in mind and our test meshes have one layer of texture coordinates.

In order to compress polygon meshes, the Touma and Gotsman coder [13] performs an initial triangulation step. But polygon meshes can be compressed more efficiently directly in their polygonal representation [3]. Recently we have extended both the connectivity coder and the geometry coder of Touma and Gotsman to the polygonal case [1, 2]. This combination delivers the lowest reported compression rates for polygon meshes and can be seen as a natural generalization of the Touma and Gotsman benchmark coder from triangle meshes to polygon meshes.

Furthermore we have proposed a predictive technique for the compression of texture coordinate mappings that gives the best compression rates currently known [4]. Together with a scheme for predictive compression of texture coordinate values [6] we have all the components for a state-of-the-art compression engine. We have made this compression software available to serve other researchers as a benchmark coder for compressing textured polygon meshes [8].

In Table 1 we list the performance of our polygonal benchmark coder on the eight textured polygon models shown in

mesh name	number of positions	texcoords	polygons	components	mesh name	binary compression rates conn	geom	texmap	texval	total [bpv]	[KB]
lion	16302	16652	16738	120	lion	1.28	13.72	0.10	6.29	21.53	42.84
wolf	7068	7234	7454	35	wolf	1.10	13.40	0.06	6.59	21.30	18.38
raptor	7454	6984	7808	79	raptor	1.21	13.75	2.93	6.33	23.83	21.68
fish	4685	4685	4901	7	fish	1.11	14.03	0.00	7.05	22.18	12.69
snake	11137	11610	11268	6	snake	0.24	8.49	0.06	3.92	12.88	17.51
horse	9199	9988	9518	5	horse	0.65	11.86	0.17	4.90	17.99	20.21
cat	9627	10350	10340	39	cat	1.23	12.32	0.18	5.00	19.11	22.46
dog	6650	6522	9278	19	dog	1.74	16.21	1.84	6.81	26.46	21.48
					average	1.07	12.97	0.66	5.86	20.66	

Table 1: The table lists the number of positions, texture coordinates, polygons, and components for each polygon mesh shown in Figure 2. The binary benchmark compression rates for the **conn**ectivity [bpv], the **geom**etry [bpv], the **tex**ture coordinate **map**ping [bpv], and the **tex**ture coordinate **val**ues [bpt] are reported side by side in bits per vertex (bpv) or bits per texture coordinate (bpt). The level of quantization was 12 bits for positions and 10 bits for texture coordinates.

Figure 2. Compression rates are reported separately in bits per vertex (bpv) for the connectivity, for the geometry, and for the texture coordinate mapping[1] and in bits per texture coordinate (bpt) for the texture coordinate values.

In order to achieve these low bit-rates our coder makes heavy use of context-based arithmetic coding [14]. Given sufficiently long input, the compression rate of such a coder converges to the entropy of the input. The entropy for a sequence of n symbols is $-\sum_n \left(p_i \log_2(p_i)\right)$, where the ith symbol occurs with probability p_i. Roughly speaking, the less spread out the distribution of the symbol stream, the lower is its entropy. A context-based approach splits one symbol stream into two or more symbol streams in the hope that each of them has a lower dispersion.

In the following we only try to give an intuition why the use of context-based arithmetic coding is imperative for our coder's success. For the details on encoding and decoding we refer the reader to the original references [1, 2, 4, 6].

Connectivity: We represent the connectivity of a polygon mesh basically as a sequence of vertex degrees and face degrees [1]. Because low-degree vertices are more likely to be surrounded by high-degree faces and vice versa, we switch contexts based on neighboring face degrees when encoding a vertex degree and based on neighboring vertex degrees when encoding a face degree.

Geometry: After recording the floating-point bounding-box of a polygon mesh we quantize its positions using a user-defined number of precision bits. Then we represent the positions through a sequence of vectors, each correcting the prediction of a position with the parallelogram rule. Because the parallelogram predictions *within* a polygon are more successful then those *across* polygons, we switch contexts based on whether the prediction was within or across when encoding a corrective vector [2].

Texture Coordinate Mapping: Classifying vertices and corners of a mesh as smooth and not smooth is sufficient to encode a manifold texture coordinate mapping [4]. Since there is a strong correlation between neighboring corners and vertices we switch between eight different contexts when encoding whether a vertex or a corner is smooth or not.

Texture Coordinate Values: Similar to positions, the texture coordinates are first quantized and then predicted with the parallelogram rule. In the presence of discontinuities in the texture mapping we fall back to a simpler predictor [6]. When compressing the corrective vectors we again switch contexts depending on the used prediction.

[1]The reason that the texture coordinate mapping of the "raptor" and the "dog" model does not compress as well as that of other models is that these two mappings are heavily *non-manifold*. This means that texture coordinates are re-used for different parts of the model.

5. SUMMARY OF RESULTS

We have implemented a proof-of-concept prototype of the benchmark coder for polygon mesh compression [8] that uses an arithmetic coder which writes to and reads from *base64* ASCII. The implementation was integrated into the pluginless, pure java Web3D player from Shout3D [10]. Their API allows to extend the VRML-style node set by automatically downloading all java classes required for custom nodes on demand. By extending the standard *IndexedFaceSet* node to a new *CompressedIndexedFaceSet* node, we are able to present a fully functional prototype that integrates the compression scheme proposed here into a VRML scene.

The new compressed node contains the *base64* coded bits of the arithmetic coder, the floating-point bounding box for positions and texture coordinates, and the number of precision bits used for quantization. It automatically decodes itself on the fly after it was loaded by the browser. An interactive online demo of this prototype can be found at this web address: **http://www.cs.unc.edu/~isenburg/ac**

In order to simplify performance comparisons we use the same set of simple VRML scenes as in [5], each of which contains one of the polygon models shown in Figure 2. The size of the (gzipped) ASCII file for each of these scenes is listed in Table 2. Side by side we report the file size of a scene using the *plain* IndexedFaceSet node, the *old* CodedIndexedFaceSet node from [5], and the *new* CompressedIndexedFaceSet node that uses the compression scheme proposed here.

The average compression gain of our scheme over plain VRML is 1 : 25 for regular ASCII files and 1 : 11 for gzipped ASCII files. Finally, we compare the file size that our coder achieves in binary to those it achieves in gzipped ASCII and report the difference. Since the total binary file size reported in Table 1 reflects only what is required to encode the polygon mesh we added 100 bytes to the binary rate to account for the scene setup that is included in the gzipped ASCII rate. On average the gzipped ASCII rates are only 1 to 2 percent above the binary rates.

The main point we were trying to make is that compression—in particular geometry compression for VRML—does not require a binary file format. The presented technique is independent from the mesh compression scheme we used [8] and may also be combined with other binary mesh coders [13,

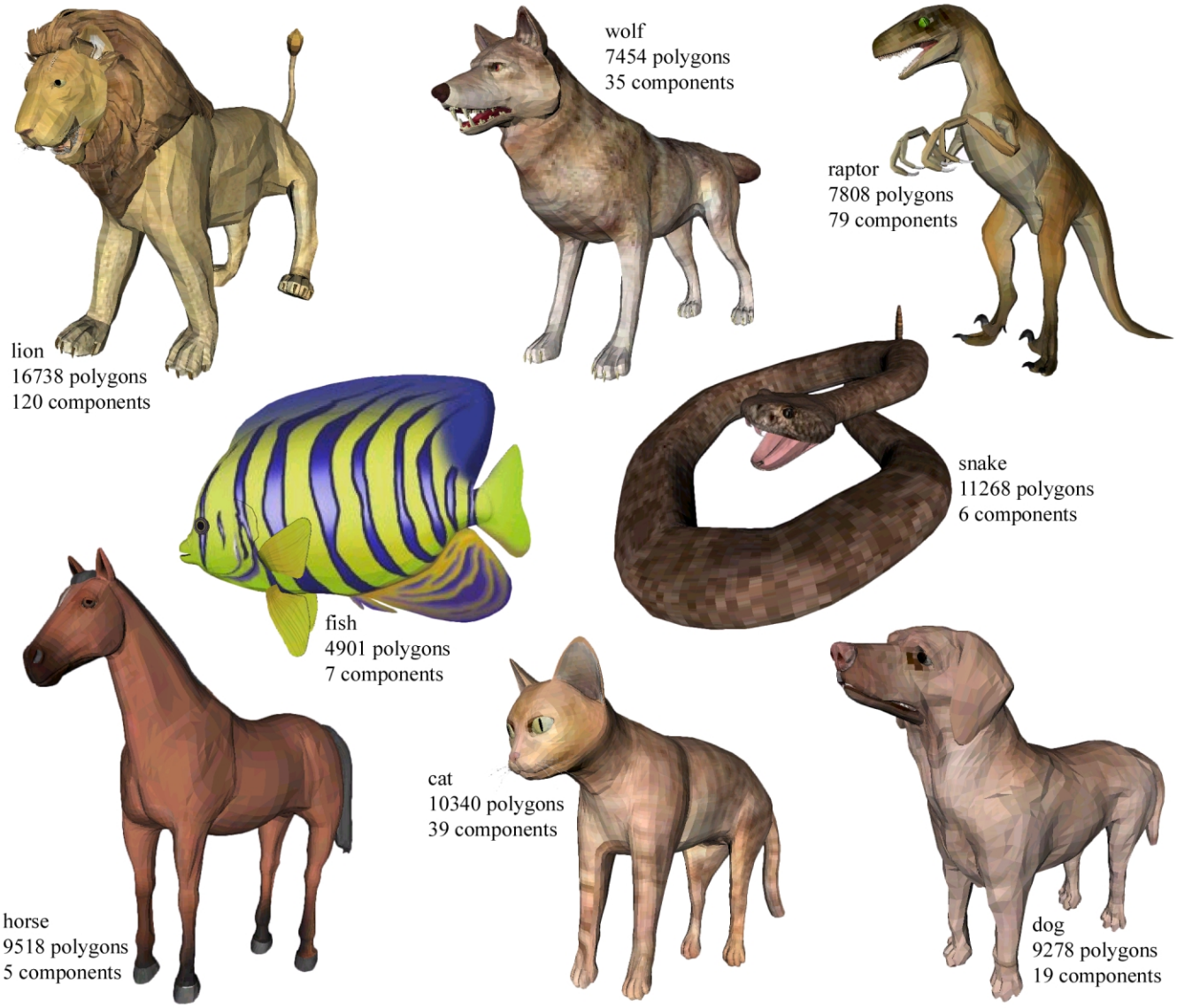


Figure 2: The eight example models used in this paper are all textured and have non-triangular connectivity.

11]. In fact, it can be applied to any binary data that needs a compact representation inside a (gzipped) text file. Therefore it could also be used to keep compressed BIFS nodes inside the extensible MPEG-4 Textual Format (XMT) [7].

6. REFERENCES

[1] M. Isenburg. Compressing polygon mesh connectivity with degree duality prediction. In *Graphics Interface'02 Conference Proceedings*, pages 161–170, 2002.

[2] M. Isenburg and P. Alliez. Compressing polygon mesh geometry with parallelogram prediction. In *Visualization'02 Conference Proceedings*, pages 141–146, 2002.

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

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

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

[6] M. Isenburg and J. Snoeyink. Compressing texture coordinates with selective linear predictions. *submitted*.

[7] M. Kim, S. Wood, and L. Cheok. The extensible MPEG-4 textual format (XMT). In *Proceedings of the ACM workshops on Multimedia*, pages 71–74, 2000.

[8] A benchmark coder for polygon mesh compression. *http://www.cs.unc.edu/~isenburg/pmc/*

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

[10] Shout3D. a pure java Web3D API, www.shout3d.com.

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

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

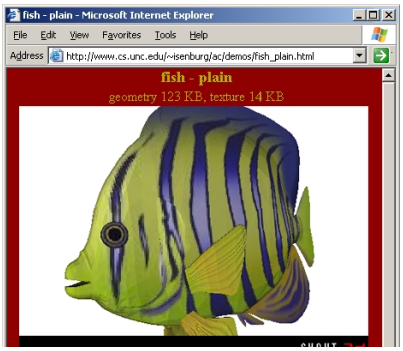
[13] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface'98 Proc.*, pages 26–34, 1998.

[14] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. In *Communications of the ACM*, 30(6):520–540, 1987.

mesh name	size of ASCII file [KB] plain	old	new	ratio	size of gzipped ASCII file [KB] plain	old	new	ratio	mesh name	size of compressed file [KB] binary	gzipped ASCII	difference
lion	1360	311	57	1:24	442	66	43	1:10	lion	42.94	43.49	1.3 %
wolf	569	135	25	1:23	183	29	19	1:10	wolf	18.47	18.80	1.8 %
raptor	586	154	29	1:20	200	35	22	1:9	raptor	21.78	22.14	1.7 %
fish	375	91	17	1:22	123	23	13	1:9	fish	12.78	13.06	2.2 %
snake	909	211	24	1:38	312	35	18	1:17	snake	17.61	17.93	1.8 %
horse	749	189	27	1:28	266	41	21	1:13	horse	20.30	20.64	1.7 %
cat	791	192	30	1:26	267	40	23	1:11	cat	22.56	22.92	1.6 %
dog	586	143	29	1:20	186	35	22	1:8	dog	21.58	21.94	1.7 %
average				1:25				1:11	**average**			1.7 %

Table 2: The tables lists the file size of (gzipped) ASCII scenes that contain a polygon model using either the *plain* IndexedFaceSet node, the *old* CodedIndexedFaceSet node proposed in [5], or the *new* CompressedIndexedFaceSet node proposed here. In addition the compression ratios between *new* and *plain* are given. Quantization of positions was to 12 and of texcoords to 10 bits of precision. The table also reports the compression difference between binary and gzipped ASCII in percent.

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      modulateTextureWithDiffuse true
      diffuseColor 1 1 1
    }
    texture ImageTexture {
      url fish.jpg
    }
  }
  geometry IndexedFaceSet {
    creaseAngle .9
    coord Coordinate {
      point [ -0.0715 4.7609 6.3930 -0.0715 ...  -0.4689 -4.4092 4.4136 ]
    }
    coordIndex [ 7 6 209 204 -1 4 217 210 5 -1 ...  -1 4577 4423 4222 -1 ]
    texCoord TextureCoordinate {
      point [ 0.3735 0.9441 0.3289 0.9315 ...  0.2666 0.4990 0.1082 ]
    }
    texCoordIndex [ 0 1 2 3 -1 4 5 6 7 -1 8 9 ...  -1 4311 4293 4683 -1 ]
  }
}
```




```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      modulateTextureWithDiffuse true
      diffuseColor 1 1 1
    }
    texture ImageTexture {
      url fish.jpg
    }
  }
  geometry CompressedIndexedFaceSet {
    creaseAngle .9
    code "CgEofTPoTnfUYfTWzzAbc9ehFji6UL0ZwL ...  PvAU34iOfyvXS7GVbHHUuX"
    box_tex [ 0.0146 0.0749 0.9896 0.9834 ]
    bits_tex 10
    box [ -1.9725 -7.6861 -10 1.9725 7.6861 10 ]
    bits 12
  }
}
```

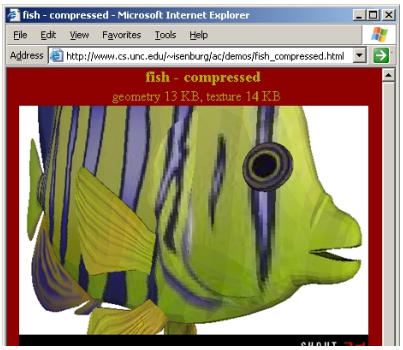


Figure 3: Two simple VRML scenes that describe a texture-mapped polygon model of a fish. The first scene uses the standard *IndexedFaceSet* node to describe this textured polygon mesh. The second scene uses our *CompressedIndexedFaceSet* node to describe the same model much more compactly. It contains the following data: the floating-precision bounding box for positions (*box*) and texture coordinates (*box_tex*), the number of precision bits used for quantization of positions (*bits*) and texture coordinates (*bits_tex*), and the *base64* coded bits of the arithmetic coder (*code*). The loss in precision resulting from quantizing the vertex positions and the textures coordinates to 12 and 10 bits respectively is not visible. The compressed scene can still be viewed and edited in any text editor to, for example, change the material or the crease angle. However, the size of the gzipped VRML file is now only 13 KB instead of 123 KB bytes.