Fast Compression of Meshes for GPU Ray-Tracing

Vasco Costa
INESC-ID/IST
Rua Alves Redol, 9
1000-029 Lisboa,
Portugal
vasco.costa@ist.utl.pt

João M. Pereira
INESC-ID/IST
Rua Alves Redol, 9
1000-029 Lisboa,
Portugal
jap@inesc-id.pt

Joaquim A. Jorge
INESC-ID/IST
Rua Alves Redol, 9
1000-029 Lisboa,
Portugal
jaj@inesc-id.pt

ABSTRACT

We present a novel and expedite way to compress triangles meshes, fans and strips for ray-tracing on a GPU. Our approach improves on the state of the art by allowing the lossless compression of all connectivity information without changing the mesh configuration, while using linear time and space with the number of primitives. Furthermore, the algorithm can be run on a stream processor and any compressed primitive can be indexed in constant time, thus allowing fast random-access to geometry data to support ray-tracing on a GPU. Furthermore, both triangle and quad meshes compress particularly well, as do many type-specialized mesh structures where all primitives have an equal number of vertexes. Our results show that the compression algorithm allows storing and ray-tracing meshes with tens of millions of triangles on commodity GPUs with only 1GB of memory.

Keywords

Ray-tracing, gpu, mesh, compression.

1 INTRODUCTION

Polygon meshes are the most common representation for scene geometry. Triangles are the most common primitive although quad meshes are also popular in some applications being particularly suitable for architectural scenes where large and flat surfaces are the most preeminent features in a scene.

Triangle meshes may also be represented with triangle fans or triangle strips. These allow additional space savings by taking advantage of the fact that there will often be common edges in a triangular mesh. The common edges in fans and strips also mean the computational costs required to compute visibility will be reduced by virtue of symmetry in the adjacent triangles.

However, meshes are a very inefficient and storage consuming way of representing complex scenes, which makes it difficult to fit even moderately complex scenes inside graphical cards with limited memory, in more complex scenes it may be required to use different kinds of primitive types to enable higher face data compression as can be observed in Figure 1.

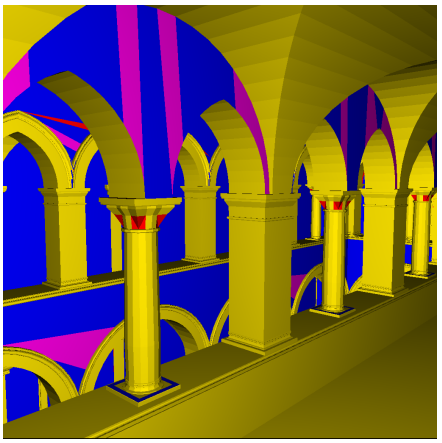


Figure 1: Sponza scene. *Triangles* are shown in red, *quads* are yellow, *triangle fans* are blue, *triangle strips* are violet. A triangle mesh representation would use 1.62 MB of space compared to the representation in the figure which only requires 1.17 MB. Thus we achieve a 72% compression ratio just by employing these more complex primitives.

A common way used to store such meshes with different primitive types is displayed in a simplified form in Figure 2. For example in PBRT [PH10] a scene is stored in a list of primitives where each primitive is subclassed from a main object class. This leads to much waste of memory storing pointers, C++ class data, etc when the scene is a mesh, so PBRT supports type specialized triangles meshes as well for improved performance in such cases as can be seen in Figure 4. Similar special-

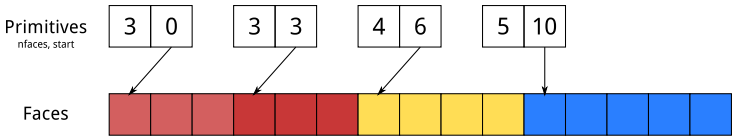


Figure 2: Regular data structure to store an n-gon mesh.

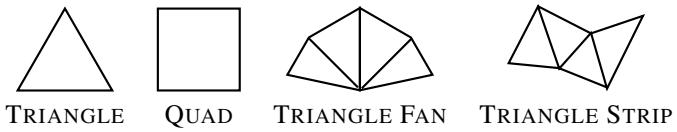


Figure 3: Primitive types.

ized quad meshes could have been implemented just as well as can be observed in Figure 5.

In this work we shall present an algorithm for mesh storage. This algorithm shall enable storing triangle meshes and quad meshes with low storage requirements, like the type specialized versions, thanks to an innovative way of storing and compressing the primitive array data using arithmetic encoding. In addition the algorithm can also store and compress n-gon meshes with triangles, quads, triangle fans, and triangle strips. Face data is stored in a compressed array with the leading zeros trimmed out.

Our algorithm thus employs non-lossy primitive compression (arithmetic encoding) and face compression (discard leading zeros). It can also quantize 32-bit vertex coordinate data down to 16-bits in a lossy fashion. In practice the lossy compression scheme seems to have little impact on final output quality for the tested scenes, as can be seen in Table 4, and enhances compression further. In order to minimize the loss of precision all coordinates are converted from world to scene coordinates prior to the quantization step.

Our algorithm *does not require expensive preprocessing*, e.g. the construction of temporary data structures for doing adjacency queries on the mesh, so it runs in $O(n)$ linear time. It produces similar compression results to other more complex hybrid geometry and acceleration structure compression schemes.

Finally we will do a performance comparison of our achieved compression ratios versus the industry standard GZIP [Deu96] compression tool which uses a Lempel-Ziv [ZL77] compression scheme. GZIP is inherently serial since it is required to read previous values to determine the next consecutive value in the stream.

We note that our algorithm features constant time $O(1)$ random access to any primitive, while GZIP works only on streaming data, thus GZIP requires $O(n)$ time in the worst case to access any random primitive. Hybrid schemes often store the scene in a tree structure in which accesses take $O(\log n)$ time to complete.

2 RELATED WORK

We are going to limit ourselves to mentioning other algorithms which work purely on scene compression first. Our algorithm is intended for generic use, specifically for meshes, and is agnostic to the kind of ray tracing acceleration scheme being used.

For those rendering algorithms which operate on large blocks of data say on a page level basis, such as out-of-core algorithms, they may still find Lempel-Ziv or other similar general purpose compression schemes worthwhile. While these algorithms are inherently serial multiple blocks can be worked in parallel with a minor compression ratio penalty. This is the approach followed by the LZSS streaming compression algorithms [OSC12]. Also essential is work on processing variable length data on streaming architectures [Bal10] in an efficient fashion with a parsimonious use of atomic operations.

Given that we are using a ray-tracer primitive intersection tests for triangles [MT97], quads [LD05], fans [GA05], and triangle meshes in general [AC97] demand being mentioned.

In the realm of geometric compression and mesh optimization several works stand out:

- Isenburg [ILS05a, ILS05b] uses arithmetic encoding to compress vertex coordinates while taking advantage of parallelogram predictors in triangular meshes by virtue of having knowledge beforehand of the mesh topology leading to improvements in the encoding predictor function.
- Yoon [YLPM05, YL06] reorders the geometry in order to increase memory coherency during the rendering pass thus improving rendering performance.

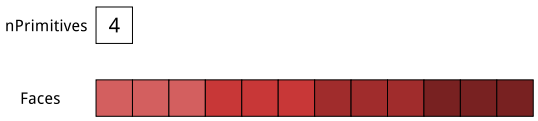


Figure 4: Specialized triangle mesh.

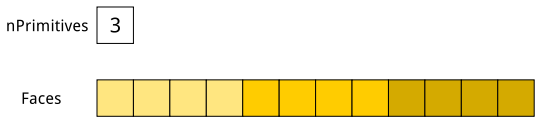


Figure 5: Specialized quad mesh.

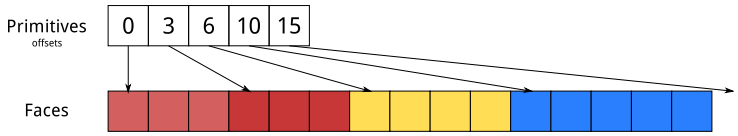


Figure 6: Compact data structure to store an n-gon mesh.

There are other interesting works on compressing ray tracing acceleration structures together with the geometry. These are not acceleration scheme agnostic and take advantage of local knowledge coming from the cells. For example, take a partition cell's bounding box, and improve compression of both the partitioning structure and the geometry contained therein since you know the vertexes range of values is constrained to the box.

Most of the interest in this sort of scheme lies with bounding volume hierarchies (BVHs) since in that scenario you do not need to store the same faces twice in different leaves of the tree.

BVHs do not have duplicated primitive instances in different cells. Yoon [YL07, KMKY10] has done a lot of work in this area of hybrid BVH and mesh storage schemes. More recently Garanzha [GBG11] has worked on a more simplified hybrid BVH and mesh storage scheme implemented for use on a streaming architecture. Another approach is that taken by Lauterbach [LYM07] where a hybrid Kd-tree and triangle strip scheme is used to represent mesh data. The main issue with all of these particular hybrid algorithms is their complexity, requiring expensive preprocessing e.g. the construction of temporary data structures for doing adjacency queries on the mesh, and their difficulty of implementation. Preprocessing takes a long time so these methods are not useful for dynamic scenes with destructible geometry. The encoding methods of Garanzha are much more amenable for GPU implementation than Yoon's more elaborate work and the algorithm provides good rendering performance due to the use of a surface area heuristic (SAH) BVH, good data locality, and aligned memory loads.

Our work is intended to be *acceleration structure agnostic* so we do not rely on any of these schemes. The algorithm we devised is also amenable to implementation on a streaming architecture and can be computed in $O(n)$ linear time. Our algorithm may be used on any object/space subdivision structure: BVHs, KD-trees, Grids. It is also applicable for other applications which do not require the use of an acceleration structure and just require random access to the mesh geometry.

3 ALGORITHM

Our algorithm compresses an n-gon scene. As an example the scene can be described in the .OBJ file format.

3.1 Scene Loading

The scene loader processes the scene data and generates a regular data structure such as that seen in Figure 2 as output.

Since .OBJ file format n-gons are not necessarily planar this means we cannot use explicit ray-polygon intersection routines safely.

So we load the scene n-gons in the following fashion:

- if the polygon has *three* faces, the output is a *triangle* which is stored in the primitive and face arrays.
- if the polygon has *four* faces, the output is a *triangle fan* which is stored in the primitive and face arrays.
- if the polygon has *five or more* faces, the output is passed through the GLUTESSELATOR which splits the n-gon into *triangles*, *triangle fans*, and *triangle strips* that are in turn stored in the primitive and face arrays.

In the next step we process an array such as the one in Figure 2 into a compact array like the one which can be seen in Figure 6. This is done with a SCAN operation:

```
def scan(uint *prims, uint nprims) {
  for (uint i=1; i<=nprims; ++i) {
    prims[i+1] += prims[i];
  }
}
```

Listing 1: SCAN.

The complexity of a SCAN also known as PREFIX-SUM operation is $O(n)$ but in a parallel processor it can be computed even faster. With such a pass we reduce the amount of memory required to store the primitive array by roughly a half.

3.2 Geometry Compression

Next to the primitive array compaction we proceeded to its compression.

We employ arithmetic encoding to compress the primitive array using the PACKPRIMITIVES function. The predictor function we are employing assumes all the primitives in the scene have the same number of faces. So for triangle meshes and quad meshes the primitive array is shrunk to nothing and the array degenerates to those seen in Figures 4 and 5 respectively. This is the optimum outcome.

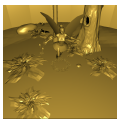
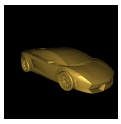
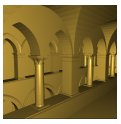
	 LADY BIRD	 BULLDOZER	 SANDAL	 FAIRY FOREST	 LAMBORGHINI	 SPONZA
num vertices	23903	105507	2636	97124	575416	39742
original						
num faces	93984	436587	11676	365949	3017847	149926
num triangles	0	145529	1197	17715	1005949	1170
num quads	23496	0	1970	78201	0	34819
num fans	0	0	29	0	0	649
num strips	0	0	0	0	0	248
triangulated						
num faces	140976	436587	15852	522351	3017847	228462
num triangles	46992	145529	5284	174117	1005949	76154

Table 1: Scene geometry data.

For more complex scenes with dissimilar primitives the delta between the predicted and actual value is stored packed tightly as a small integer of range 2^{pMSB} in a bit array.

```
def packPrimitives(uint *prims, uint nprims) {
  float avg;
  avg = float(prims[nprims+1])/nprims;

  int Min = 0;
  int Max = 0;

  min = prims[nprims+1];
  for (uint i=0; i<=nprims; i++) {
    uint predict = avg*i;
    uint actual = prims[i];
    int diff = long(actual)-predict;

    Min = min(diff, Min);
    Max = max(diff, Max);
  }

  // arithmetic encode
  pMSB = log2(Max-Min);
  hprims = callocBits(nprims+1, pMSB);

  for (uint i=0; i<=nprims; i++) {
    uint predict = avg*i;
    uint actual = prims[i];

    uint diff = actual-predict-Min;
    pack(hprims, i, diff);
  }

  uint2 params;
  params.x = as_int(avg);
  params.y = Min;
  return params, hprimis;
}
```

Listing 2: PACKPRIMITIVES.

Compression of face data proceeds in a similar fashion to the small integer packing method mentioned before. We compress away the leading zeros in the face data using the PACKFACES function.

The small integers will have a range of 2^{fMSB}. The faces of primitives other than triangles are shifted to the left one bit to accommodate a flag stating if they are a triangle fan (0) or triangle strip (1) respectively.

```
def packFaces(uint *faces, uint nfaces, uint nvertices) {
  uint fMSB = log2(nvertices*2);
  hfaces = callocBits(nfaces, fMSB);

  // compress leading zeros
  for (uint i=0; i<nfaces; ++i) {
    pack(hfaces, i, faces[i]);
  }
  return hfaces;
}
```

Listing 3: PACKFACES.

Finally we quantize the vertexes from 32-bits per x, y, z component to 16-bits per component. First we take care to ensure we do this while operating in scene bounding box space in order to reduce the range of data we will compress. Then the quantization to 16-bits per component is done. This is our only lossy compression step. This PACKVERTICES function compresses vertexes by 50%.

```
def packVertices(Axisbox bounds, uint *vertices, uint nvertices) {
  hvertices = new ushort3[nvertices];

  // transform to scene bounding box coordinates and quantize to 16-bits
  const float3 scale = 65535.0f/(bounds.Max-bounds.Min);

  for (uint i=0; i<nvertices; ++i) {
    hvertices[i] = vertices[i] - bounds.Min) * scale;
  }
  return hvertices;
}
```

Listing 4: PACKVERTICES.

The error produced by the quantization step is minimal in the scenes we tested as can be seen on the rightmost column in Table 4.

In a typical scene composed of manifolds there are more polygons than vertexes so face compression is very important contrary to what one might otherwise assume. This can easily be established empirically by looking at the scenes in Table 1.

3.3 Intersection Testing

During ray tracing scene traversal it will be required to test if a given primitive is intersected by a ray. All primitive intersection tests are done using the Möller-Trumbore [MT97] ray-triangle intersection algorithm.

This can be accomplished with the TEXTPRIMITIVE pseudo-code.

```
bool testPrimitive(Axisbox bounds, uint id, Intersection *isect) {
  const uint i = prims[id];
  const uchar nverts = prims[id+1] - i;

  switch (nverts) {
  case 3:
    {
      const uint3 idx = vload3(0, faces+i);
      return testTriangle(bounds, idx, vertices, ray, isect);
    }
    break;
  case 4:
    {
      const uint4 idx = vload4(0, faces+i);
      return testQuad(bounds, idx, vertices, ray, isect);
    }
    break;
  default:
    if ((faces[i] & 1)) {
      return testStrip(bounds, nverts, vertices, faces+i, ray, isect);
    } else {
      return testFan(bounds, nverts, vertices, faces+i, ray, isect);
    }
    break;
  }
  return false;
}
```

Listing 5: TESTPRIMITIVE.

Unfortunately, like we mentioned before, it cannot be trusted that the quads in an .OBJ file will be planar as is indeed the case in the FAIRY FOREST, SANDAL, and other test scenes. It is debatable if we should just constrain the primitives to e.g. triangles and fans in order to support n-gons since it would greatly simplify the control code and save 1 bit per face.

4 TEST METHOD

The algorithm was implemented in C++ and OpenCL on Linux. The test platform is an AMD FX 8350 8-core CPU @ 4.0 GHz powered machine with 8 GB of RAM. The graphics card includes a NVIDIA GeForce GTX 660 Ti GPU with 2 GB of RAM. All ray tracing rendering is offloaded to the GPU.

The ray-tracing engine supports hashed grids [LD08] as a spatial subdivision acceleration structure. All the compute intensive parts of the grid construction algorithm are also run on the GPU. Due to limitations of space we do not describe this engine in detail here.

All scenes were rendered at 1024×1024 resolution with one sample per pixel using Phong shading.

We selected several test scenes not just for having large amounts of geometry but for the richness of their polygon soup so to speak. In order of presentation the scenes are:

LADY BIRD	Quad mesh of an organic creature.
BULLDOZER	Triangle mesh of a construction vehicle.
SANDAL	More complex primitives. e.g. triangle fans in the sole of the shoe.
FAIRY FOREST	Standard benchmark scene which features both triangles and quads.
LAMBORGHINI	Large triangle mesh of a car with over 1M triangles.
SPONZA	Complex scene in terms of geometric detail due to the arches but also features several flat surfaces such as walls.

For more information on the scene geometry data please consult Table 1.

The following tests were considered to be interesting:
- To test our proposed compression methods vs other schemes, namely [GBG11], specific for geometry compression for GPU ray tracing purposes.

- To determine the performance of the primitive array predictor function in the arithmetic coding phase we test the misprediction residuals i.e. the delta between the predicted and actual values in the test scenes.

- Compression ratio of our proposed compression methods vs uncompressed data in terms of: space savings and rendering frame rate.

- Compression ratio of our proposed compression methods vs GZIP to determine how good our compression algorithms are at achieving space savings compared to a standard streaming compression algorithm.

5 RESULTS

As can be seen in the left of Table 2 our implementation uses less memory than the non-compressed triangle meshes of [GBG11] because we do not store duplicate triangle vertexes. Our compressed face and vertex scheme achieves similar compression results to their compressed quad mesh for the tested scenes *without* requiring any pre-processing to convert the triangle mesh to a quad mesh as they do. Our algorithm also supports the use of quad meshes but, as can be seen in the right of Table 2, this is not required to achieve good compression ratios due to our face compression algorithm and the storage of unique vertexes only.

One of the big issues with any scheme employing arithmetic coding is getting the right prediction function. In our case, since we want random access in constant time we cannot consult prior values since that would require decoding them beforehand, plus all the previous values to those, to begin with. So our predic-

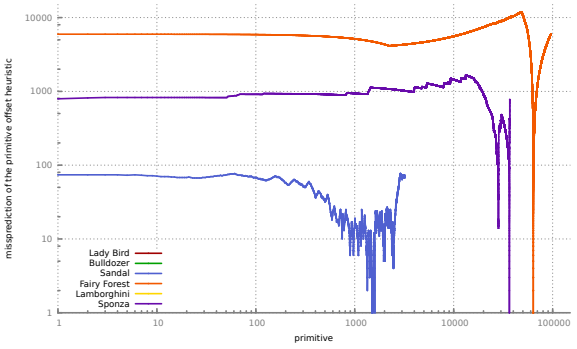


Figure 7: Delta between the predicted and actual values in the test scenes. It displays perfect prediction in the triangle and quad meshes such as Lady Bird, Bulldozer, Lamborghini scenes. For the other scenes the misprediction residuals can be quite large.

	OURS P	[GBG11] NCT	OURS PFV	[GBG11] CQ
DRAGON 7 MTRI	123.92 MB	247.85 MB	80.03 MB	82.62 MB
THAI 10 MTRI	171.66 MB	343.32 MB	114.44 MB	114.44 MB
LUCY 28 MTRI	481.61 MB	963.22 MB	331.11 MB	321.07 MB

Table 2: Comparison of the space required to store a mesh using our algorithm versus Garanzha [GBG11]. First we compare the memory usage of both our specialized triangle meshes. Next we compare the performance of our compressed face and vertex scheme versus their compressed quad mesh.

tion function must rely only on a table of values independent of the compressed array. In our case we use a linear prediction function to approximate the primitive data values. This works well when the scenes all have similar sized n-gons i.e. triangle or quad meshes where the residuals are zero. The mispredictions and residuals increase with the storage of different sized n-gons. This can be observed in Figure 7 where the LADY BIRD, BULLDOZER, LAMBORGHINI primitive arrays get compressed to 0 bits per primitive while on the other scenes namely SANDAL, FAIRY FOREST, and SPONZA this does not happen. This problem could probably be reduced by employing higher order prediction functions such as polynomial functions with more terms than our linear function.

From the extensive test results, which can be observed in Table 3, we managed to confirm several of our hypothesis as matters of fact. The support for triangle and quad mesh scenes, such as LADY BIRD, BULLDOZER, and LAMBORGHINI, is excellent with very good compression capabilities and, in some cases, we even achieve enhanced frame rates over the uncompressed baseline due to improved data locality caused by the compression. This is most obvious when compressing the **p**rimitives array and enabling the lossy **v**ertex compression together in the pure triangle and quad meshes i.e. the **PV** results. This option also enables reasonable compression ratios in the order of $\sim 70\%$.

We can also observe that for the more complex scenes using large n-gons we often get higher frame rates by triangulating the scene beforehand. This is rather evident in the SPONZA and SANDAL scenes in particular. This is due to spatial subdivision. These larger primitives will occupy more cells in the grid and hence there will be more redundant intersection calculations going on. This could perhaps be improved with the use of mailboxing. We did not attempt to use mailboxing in our implementation. Another possible improvement is better specialized ray-triangle fan, and ray-triangle strip intersection routines which reduce the amount of redundant ops such as those mentioned in Section 2. This does not apply to the FAIRY FOREST scene because of its smaller n-gons. That scene consists of only triangles and quads.

We get our worst frame rate results when **f**ace compression is enabled. This is due to the loss of use of vectorized memory loads in our implementation and the unaligned memory accesses to access elements after the compression. This can most likely be improved further by creating dedicated vectorized load operations for our bitarray structures.

Invariably the highest compression ratios happen when we enable all of our compression techniques: **p**rimitive compression, **f**ace compression and **v**ertex compression i.e. the **PFV** results. In fact with all these techniques enabled we get better compression ratios than GZIP in many scenes such as LADY BIRD, SANDAL, FAIRY FOREST. The only scenes where GZIP manages to win over our compression scheme are those scenes where the vertexes have redundant coordinates or there are redundant vertexes such as the BULLDOZER, the LAMBORGHINI, and SPONZA. This problem with vertex quantization techniques had already been identified by Isenburg et al.

We remind the reader that contrary to GZIP our algorithm allows random access to any primitive in the scene in $O(1)$ constant time which is essential for ray-tracing and other applications. This is particularly relevant for stringy kd-trees with few primitives per leaf. We get compression ratios in the order of $\sim 40-50\%$ which is commensurate with streaming lossless compression techniques which are a lot harder to parallelize on a GPU properly since they require sequential access to compress or decompress data.

6 CONCLUSIONS

We have demonstrated an n-gon (triangle, quad, triangle fan, triangle strip) scene compression algorithm which can compress such a scene in linear $O(n)$ time with constant $O(1)$ scene primitive access time. For the special case where the n-gons in the scene are all the same size like triangle or quad meshes the additional space used over a type specialized structure is essentially zero. The algorithm can optionally provide limited compression ratios with improved rendering performance, or much improved compression with worse rendering performance than in the uncompressed case. The compression ratios, in the order of $\sim 40-50\%$, are competitive with those achieved using the GZIP tool which, contrary to our algorithm, does not allow constant time random access to the data.

In the future it should be possible to significantly improve the primitive array compression level using non-linear prediction functions. We also believe that either a limit on the size of triangle fans and triangle strips is imposed, in order to improve the performance under spatial subdivision ray tracing, or there needs to be a way to more quickly detect misses for such complex primitives, use mailboxing, or more than one of these

schemes. Otherwise the rendering performance for the complex primitives will be subpar, even if the compressed sizes for scenes using these primitives would be a lot better.

7 ACKNOWLEDGMENTS

This work was supported by national funds through FCT - Fundação para a Ciência e Tecnologia, under project PEst-OE/EEI/LA0021/2013.

We would like to thank Gilles Tran (Lady Bird), Ralph Garmin (Bulldozer), John Burkardt (Sandal), Ingo Wald (Fairy Forest), Temur Kvitelashvili (Lamborghini), Marko Dabrovic (Sponza) and the Stanford 3D Scanning Repository (Dragon, Thai, Lucy) for the test scenes.

REFERENCES

[AC97] J. Amanatides and K. Choi. Ray Tracing Triangular Meshes. In *Proceedings of the Eighth Western Computer Graphics Symposium*, pages 43–52, 1997.

[Bal10] A Balevic. Parallel variable-length encoding on GPGPUs. In *Euro-Par 2009, Parallel Processing Workshops*, pages 26–35. Springer, 2010.

[Deu96] P. Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[GA05] E. Galin and S. Akkouche. Fast Processing of Triangle Meshes using Triangle Fans. In *Shape Modeling and Applications, 2005 International Conference*, pages 326–331. IEEE, 2005.

[GBG11] K. Garanzha, A. Bely, and V. Galaktionov. Simple Geometry Compression for Ray Tracing on GPU. In *GraphiCon'2011*, 2011.

[ILS05a] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless Compression of Predicted Floating-Point Geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[ILS05b] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming Compression of Triangle Meshes. In *ACM SIGGRAPH 2005 Sketches*, page 136. ACM, 2005.

[KMKY10] T. Kim, B. Moon, D. Kim, and S. Yoon. RACBVHs: Random-accessible compressed bounding volume hierarchies. *Visualization and Computer Graphics, IEEE Transactions on*, 16(2):273–286, 2010.

[LD05] A. Lagae and P. Dutré. An Efficient Ray-Quadrilateral Intersection Test. *Journal of Graphics Tools*, 10(4):23–32, 2005.

[LD08] A. Lagae and P. Dutré. Compact, Fast and Robust Grids for Ray Tracing. In *Computer Graphics Forum*, pages 1235–1244. Wiley Online Library, 2008.

[LYM07] C. Lauterbach, S. Yoon, and D. Manocha. Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 19–26. IEEE, 2007.

[MT97] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[OSC12] A. Ozsoy, M. Swany, and A. Chauhan. Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 37–44. IEEE, 2012.

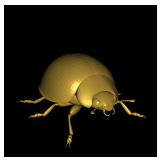
[PH10] M. Pharr and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2010.

[YL06] S. Yoon and P. Lindstrom. Mesh Layouts for Block-Based Caches. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1213–1220, 2006.

[YL07] S. Yoon and P. Lindstrom. Random-Accessible Compressed Triangle Meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1536–1543, 2007.

[YLPM05] S. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics (TOG)*, 24(3):886–893, 2005.

[ZL77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.

LADY BIRD

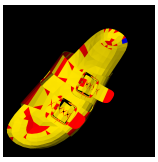
REGULAR 739.03 KB
TRIANGULATED 1014.37 KB

TECHNIQUES	GZIP	OURS	COMP RATIO	FRAME RATE
	393.12 KB	739.03 KB	100 %	102.94 FPS
V	...	598.97 KB	81.05 %	100.93 FPS
F	...	555.46 KB	75.16 %	81.44 FPS
FV	...	415.41 KB	56.21 %	78.06 FPS
P	...	647.24 KB	87.58 %	**111.16 FPS**
PV	...	507.18 KB	68.63 %	104.92 FPS
PF	...	463.68 KB	62.74 %	84.25 FPS
PFV	...	**323.62 KB**	**43.79 %**	85.53 FPS
T	465.78 KB	1014.37 KB	100 %	87.16 FPS
TV	...	874.31 KB	86.19 %	90.81 FPS
TF	...	739.03 KB	72.86 %	65.04 FPS
TFV	...	598.97 KB	59.05 %	67.92 FPS
TP	...	830.80 KB	81.9 %	97.14 FPS
TPV	...	690.74 KB	68.1 %	100.40 FPS
TPF	...	555.46 KB	54.76 %	71.23 FPS
TPFV	...	415.40 KB	**40.95 %**	73.98 FPS


BULLDOZER

REGULAR 3.43 MB
TRIANGULATED 3.43 MB

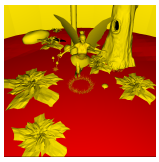
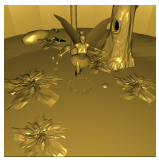
GZIP	OURS	COMP RATIO	FRAME RATE
1.28 MB	3.43 MB	100 %	88.58 FPS
...	2.82 MB	82.39 %	87.87 FPS
...	2.70 MB	78.74 %	69.24 FPS
...	2.10 MB	61.13 %	68.66 FPS
...	2.87 MB	83.81 %	94.51 FPS
...	2.27 MB	66.19 %	95.30 FPS
...	2.14 MB	62.55 %	73.26 FPS
...	**1.54 MB**	**44.94 %**	74.93 FPS
1.28 MB	3.43 MB	100 %	87.98 FPS
...	2.82 MB	82.39 %	88.40 FPS
...	2.70 MB	78.74 %	69.23 FPS
...	2.10 MB	61.13 %	68.52 FPS
...	2.87 MB	83.81 %	94.37 FPS
...	2.27 MB	66.19 %	**95.62 FPS**
...	2.14 MB	62.55 %	73.08 FPS
...	1.54 MB	**44.94 %**	74.91 FPS




SANDAL

REGULAR 88.99 KB
TRIANGULATED 113.46 KB

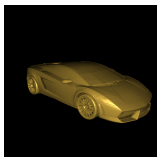
TECHNIQUES	GZIP	OURS	COMP RATIO	FRAME RATE
	54.56 KB	88.99 KB	100 %	195.99 FPS
V	...	73.55 KB	82.64 %	186.20 FPS
F	...	61.91 KB	69.57 %	157.39 FPS
FV	...	46.47 KB	52.21 %	151.69 FPS
P	...	79.23 KB	89.03 %	180.00 FPS
PV	...	63.79 KB	71.68 %	168.37 FPS
PF	...	52.15 KB	58.6 %	145.47 FPS
PFV	...	**36.71 KB**	**41.25 %**	141.51 FPS
T	61.66 KB	113.46 KB	100 %	193.17 FPS
TV	...	98.02 KB	86.39 %	183.96 FPS
TF	...	76.69 KB	67.6 %	158.19 FPS
TFV	...	61.25 KB	53.98 %	151.65 FPS
TP	...	92.81 KB	81.8 %	**196.04 FPS**
TPV	...	77.37 KB	68.19 %	188.65 FPS
TPF	...	56.05 KB	49.4 %	158.18 FPS
TPFV	...	40.60 KB	**35.78 %**	154.70 FPS

FAIRY FOREST

REGULAR 2.87 MB
TRIANGULATED 3.77 MB

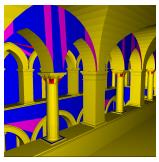
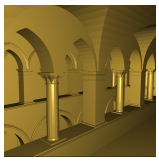
GZIP	OURS	COMP RATIO	FRAME RATE
1.54 MB	2.87 MB	100 %	**25.45 FPS**
...	2.32 MB	80.66 %	25.03 FPS
...	2.26 MB	78.74 %	19.39 FPS
...	1.71 MB	59.4 %	18.92 FPS
...	2.67 MB	92.84 %	22.52 FPS
...	2.11 MB	73.5 %	21.82 FPS
...	2.06 MB	71.58 %	17.18 FPS
...	**1.50 MB**	**52.24 %**	17.41 FPS
1.80 MB	3.77 MB	100 %	21.95 FPS
...	3.21 MB	85.25 %	21.34 FPS
...	2.90 MB	76.87 %	16.90 FPS
...	2.34 MB	62.12 %	16.49 FPS
...	3.10 MB	82.37 %	23.72 FPS
...	2.55 MB	67.63 %	23.66 FPS
...	2.23 MB	59.24 %	17.67 FPS
...	1.68 MB	**44.49 %**	17.76 FPS

LAMBORGHINI

REGULAR 21.93 MB
TRIANGULATED 21.93 MB

TECHNIQUES	GZIP	OURS	COMP RATIO	FRAME RATE
	8.87 MB	21.93 MB	100 %	53.27 FPS
V	...	18.64 MB	84.99 %	54.10 FPS
F	...	17.98 MB	81.96 %	43.74 FPS
FV	...	14.68 MB	66.95 %	44.19 FPS
P	...	18.10 MB	82.51 %	58.93 FPS
PV	...	14.80 MB	67.49 %	**61.11 FPS**
PF	...	14.14 MB	64.46 %	47.16 FPS
PFV	...	**10.85 MB**	**49.45 %**	49.03 FPS
T	8.87 MB	21.93 MB	100 %	53.02 FPS
TV	...	18.64 MB	84.99 %	53.77 FPS
TF	...	17.98 MB	81.96 %	43.84 FPS
TFV	...	14.68 MB	66.95 %	43.93 FPS
TP	...	18.10 MB	82.51 %	58.93 FPS
TPV	...	14.80 MB	67.49 %	60.80 FPS
TPF	...	14.14 MB	64.46 %	47.14 FPS
TPFV	...	10.85 MB	**49.45 %**	48.96 FPS

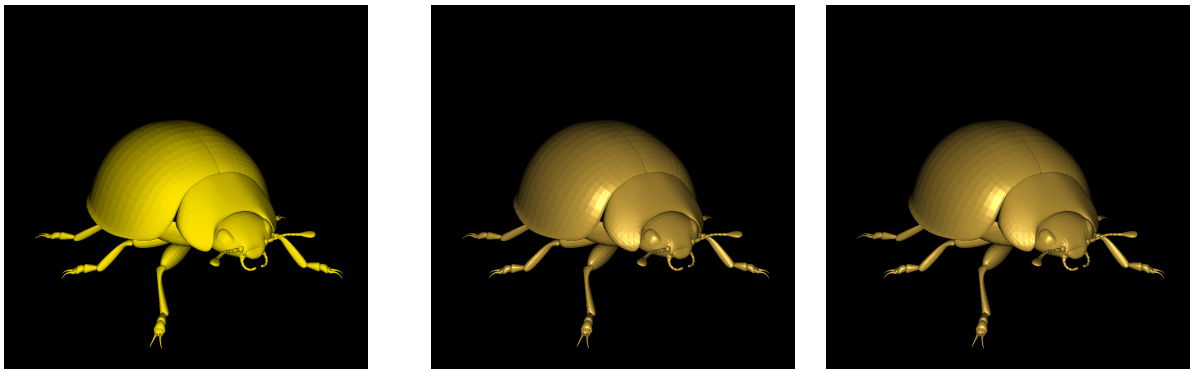



SPONZA

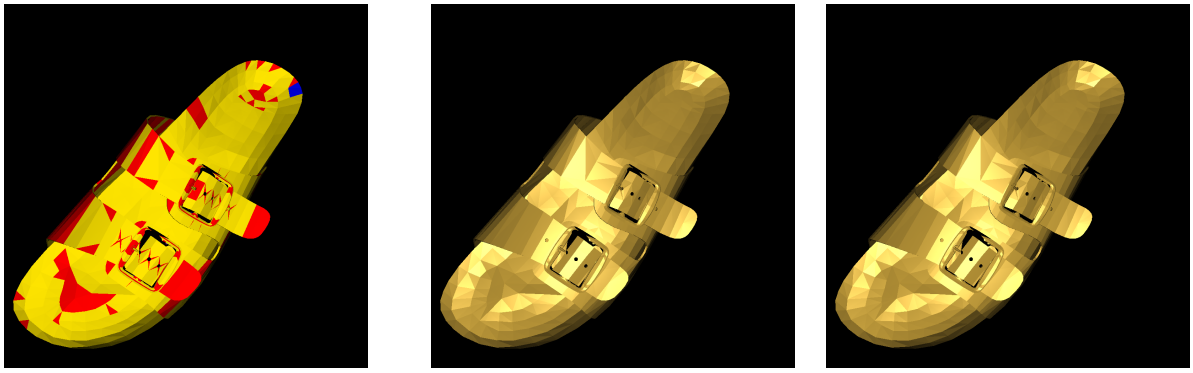
REGULAR 1.17 MB
TRIANGULATED 1.62 MB

GZIP	OURS	COMP RATIO	FRAME RATE
426.95 KB	1.17 MB	100 %	54.73 FPS
...	962.61 KB	80.52 %	52.20 FPS
...	920.95 KB	77.04 %	43.49 FPS
...	688.08 KB	57.56 %	41.24 FPS
...	1.08 MB	92.09 %	50.47 FPS
...	868.04 KB	72.61 %	47.14 FPS
...	826.38 KB	69.13 %	40.10 FPS
...	**593.52 KB**	**49.65 %**	38.38 FPS
548.34 KB	1.62 MB	100 %	**64.05 FPS**
...	1.39 MB	85.94 %	59.70 FPS
...	1.21 MB	74.73 %	48.51 FPS
...	1004.45 KB	60.67 %	45.88 FPS
...	1.33 MB	82.03 %	62.45 FPS
...	1.10 MB	67.97 %	59.45 FPS
...	939.83 KB	56.77 %	47.35 FPS
...	706.97 KB	**42.7 %**	45.95 FPS

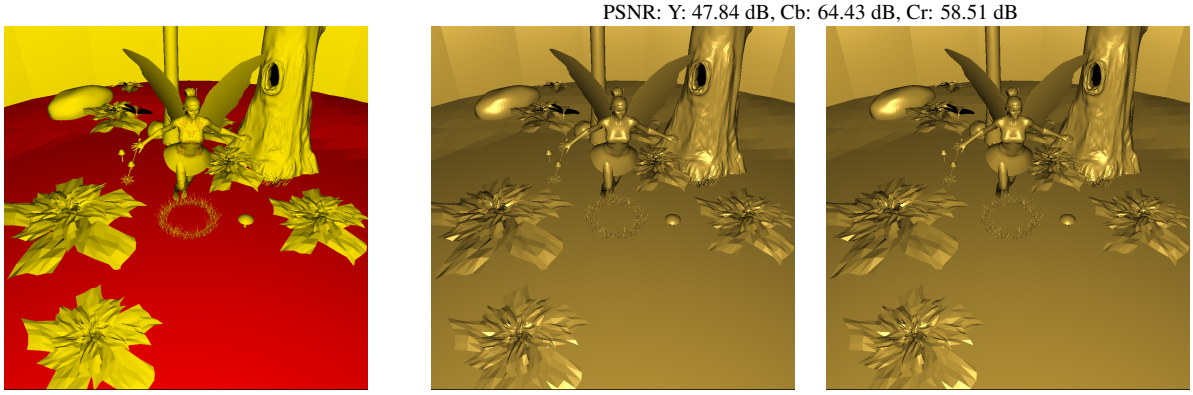
Table 3: Performance results. Includes data about total memory required without compressing scene data, the output size by compressing the scene data with gzip, the amount of the memory required to store the scene with some of our techniques enabled, the compression ratio, and finally the frame rate using any combination of the given methods. *T* triangulates the scene geometry, *P* losslessly compresses the primitive lists using arithmetic encoding, *F* losslessly compresses the faces by discarding the leading zeros of a vertex index, *V* does lossy quantization from 32-bits to 16-bits.



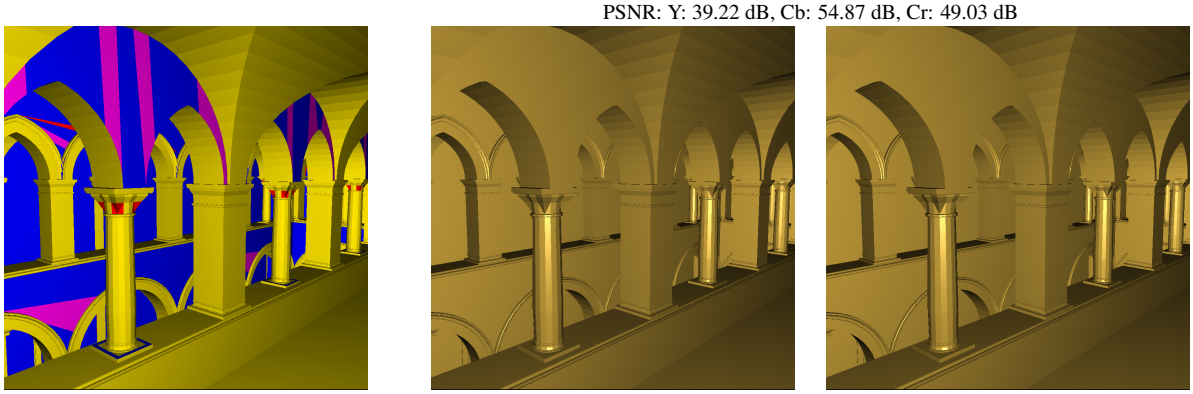
PSNR: Y: 51.72 dB, Cb: 66.10 dB, Cr: 61.41 dB



PSNR: Y: 47.84 dB, Cb: 64.43 dB, Cr: 58.51 dB



PSNR: Y: 39.22 dB, Cb: 54.87 dB, Cr: 49.03 dB



PSNR: Y: 41.23 dB, Cb: 56.71 dB, Cr: 50.69 dB

Table 4: The first column from the left allows the visualization of the primitives types in the test scenes: *triangles* are shown in red, *quads* are yellow, *triangle fans* are blue, *triangle strips* are violet. The second column shows image output without vertex compression. The third column shows image output with lossy vertex compression quantized from 32 to 16 bits.