Algorithms and Hardware for Data Compression in Point Rendering Applications

P. N. Mallón[1] M. Bóo[1] M. Amor[2] J.D. Bruguera[1]

[1]Department of Electronic and Computer Eng., University of Santiago de Compostela, Spain
E-mail: {paulanm, mboo, bruguera}@dec.usc.es
[2]Department of Electronic and Systems, University of A Coruña, Spain
E-mail: margamor@udc.es

ABSTRACT

The high storage requirements associated with point rendering applications make the utilization of data compression techniques interesting. Point rendering has been proposed only recently and, consequently, no compression strategies have yet been developed.
In this paper we present compression algorithms for two specific data distributions widely used in point rendering: a naive distribution with no specific sorting of the points and a layer distribution which is suitable for incremental algorithms. In this last case points are sorted in layers and the connectivity among them is encoded. The algorithms we propose have a high compression rate (5.0 bits/point for the naive distribution and 7.7 bits/point for the layer distribution).
Additionally we present the hardware implementation for the decompression of both algorithms. Both algorithms are implemented in a single hardware unit providing a control to select between them.

Keywords: Data compression, point rendering, graphics hardware

1 Introduction

The storage requirements associated with current 3D models are increasingly growing due to improved design and the need of higher accuracy. The management of high volumes of information involves an important bottleneck in the transmission of data from CPU to GPU. One common strategy for reducing the transmission problems is the utilization of compression techniques. In this context, different proposals have been developed for the compression of some representations commonly employed in 3D graphics, as for example triangle meshes [Gumho98] [Rossi99] [Malló02] and tetrahedral meshes [Szymc99] [Gumho99]. However, there are recently proposed representations for which no compression strategies have been developed. Among these representations, the most promising is point rendering [Alexa03].

Point based rendering techniques have recently become a promising alternative for high quality rendering of complex scenes [Pfist00] [Rusin00] [Zwick01] [Wand01] [Ren02]. Specifically, in point based applications, the objects are modelled as a dense set of surface point samples and the point rendering algorithms reconstruct a continuous image from this set of samples.

Most of the recently developed out–coming algorithms in point rendering [Pfist00] [Ren02] [Zwick01] [Amor02] are based on a regular sampling of the scene and the utilization of a hierarchical structure to store it. The scene is represented with an octree, where each node stores a cubic section of the scene. Each node stores a subsampled version of its children [Pfist00].

Basic point rendering algorithms [Wand01] [Pfist00] [Zwick01] [Ren02] [Rusin00] do not employ any specific sorting of the points inside each octree node. Incremental strategies have been suggested [Gross98] as a way to reduce the computational requirements of the algorithms to generate the final image in the GPU. An efficient implementation of these strategies requires a previous sorting of the points in the CPU [Amor02].

Complex scenes require several millions of points to be represented and a high computational

rate: up to one thousand million points per second [Wand01]. This suggests the utilization of compression techniques to reduce the transmission bandwidth between CPU and GPU. The compression is carried out in the CPU while the decompression would be implemented in specific hardware in the GPU.

In this paper we propose two compression algorithms and the corresponding hardware units for the decompression processes. The first algorithm, we call Cell Identification Algorithm (CI), is adequate for those point rendering proposals where no specific point sorting is required. Whereas the second one, Extended Cell Identification Algorithm (ECI), permits the compression of the scene for sorting proposed in [Amor02] for incremental point rendering algorithms.

These algorithms reduce the transmission requirements by about 83% for the CI and 20% for the ECI with respect to reference implementations without compression. On the other hand, both hardware decompression units can be easily integrated in a single module.

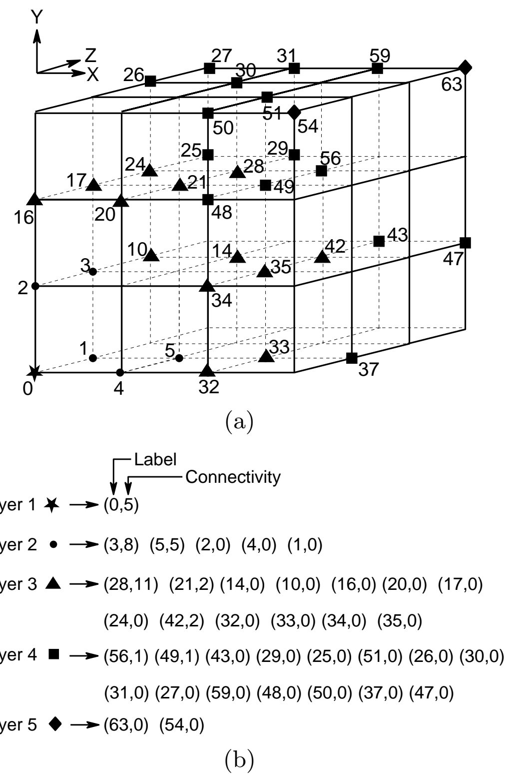


Figure 1: *Layer Distribution* (a) $4 \times 4 \times 4$ *Cube Example* (b) *Layer Representation*

2 Data Distribution in Point Rendering

In this section we summarize the data distribution usually employed in point rendering: Naive representation and layer representation. In the naive representation no specific sorting of the points inside each octree node is mandatory. In the layer representation [Amor02], the points are sorted to permit an efficient implementation of incremental algorithms. In the following we assume that the result of the regular sampling of the scene are cubes of $32 \times 32 \times 32$ positions. Extension to other sizes is straightforward.

2.1 Naive Distribution

No specific sorting of the points is predefined in naive data distributions, so the points can be organized in such a way that the compression rate can be optimized. The coordinates of each point inside the cube are (i,j,k) with $\{i,j,k\} \in [0,31]$.

2.2 Layer Distribution

Incremental algorithms exploit the regular distribution of the points in space in such a way that part of the computations can be reemployed for the processing of neighbor points. According to this, points have to be sorted before sending to the GPU. The data sorting that we summarize in this section was proposed in [Amor02]. It is based on the identification of layers of neighbor points and the specification of the connectivity among them. The layers can be iteratively built starting from a random point. The first layer is composed of its neighbor points, the second layer of the neighbor of the first layer points (excluding points already assigned to a previous layer), and so on. Figure 1(a) shows an example where each layer is marked with a different symbol. The first layer is composed of only one point, point $\{0\}$. The second layer is given by its neighbor points $\{3,5,2,4,1\}$ and so on.

A point in a given layer can be used as a "seed" for a subset of the points in the next layer; this subset is composed of the neighbors of the seed. In incremental algorithms the computations of the seed can be partially reemployed for the neighbors in the following layer. In order to optimize the incremental algorithm efficiency, a data sorting inside each layer is proposed. Specifically, seed points

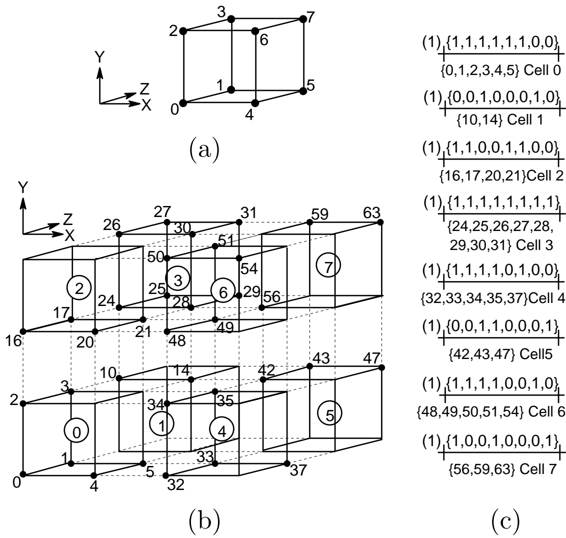


Figure 2: *Cell Identification Algorithm* (a) *Point label inside a cell* (b) $4 \times 4 \times 4$ *Cube Example* (c) *Representation*

with higher number of neighbors in the next layer are processed first. Each point is represented by two numbers (see Figure 1(b)): its label in the cube and the connectivity with next layer. Note that the connectivity of a point in a layer k is the number of neighbors in the following layer $k+1$ but still not considered as neighbor of previous a point of layer k.

3 Compression Algorithms

In this section we propose two compression algorithms for the data distribution summarized in the previous section.

3.1 Naive Distribution: Cell Identification Algorithm (CI)

The algorithm we propose is based on the representation of the cube in terms of cells (cube of 2×2×2 positions) and the identification of the points inside each cell. The cells are encoded following a specific order and empty cells are encoded with just one bit. The resulting algorithm is simple and presents a high compression rate. The cube is partitioned in non overlapping cells. As an example, in Figure 2(b) a $4 \times 4 \times 4$ cube with 36 points is presented, where cells are indicated with solid lines. Eight non overlapping cells are required for covering all positions. The cells are processed following the positive Z direction (from front to back), Y (from bottom to top) and X (from left to right) axis respectively. The numbers inside the cells indicate their processing order. On the other hand, positions inside each cell are processed according to the labelling shown in Figure 2(a).

The representation for each cell is:

$$(Cell_Filled)\{Pos_0, \ldots, Pos_7\} \tag{1}$$

where $Cell_Filled$ indicates if the cell is empty (0) or has points (1) and Pos_i indicates if there is a point in position i ($Pos_i = 1$) or not ($Pos_i = 0$). As an example, "cell 0" in Figure 2(b) is represented as (1) $\{1,1,1,1,1,1,0,0\}$ that is, there are six points in the cell in the positions labelled as {0,1,2,3,4,5}.

The $Cell_Filled$ bit allows the increase of the compression rate due to the high number of empty cells in a scene. These cells can be encoded with only one bit. Experimental results show that at least 80% of the cells, corresponding to non empty cubes, have no points. On the other hand eight bits are required for encoding the $\{Pos_0, \ldots, Pos_7\}$ list. According to this, the number of bits per point is in interval [1.125,9] where 1.125 corresponds to the case of a cell with eight points and 9 for a cell with only one point. Experimental results (Section 6) show that the average number of bits per point is around 5.0.

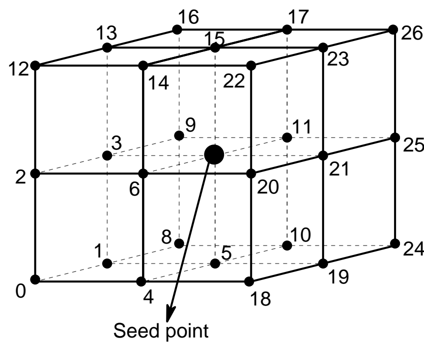
3.2 Layer Distribution: Extended Cell Identification Algorithm (ECI)

In this section we present an encoding scheme for the layer distribution described in Section 2.2. The basic idea is the encoding of the information related to each seed and its connectivity.

The encoding for each point we propose is:

$$(Position_Label, \psi, Number_Seeds) \tag{2}$$

where, for a given layer i, $Position_Label$ specifies the relative position of the point with respect to its seed, ψ is the connectivity of the point with the following layer $i+1$. Among these ψ points, some of them are also seeds for $i+2$ layer; this is indicated in $Number_Seeds$. To specify a binary encoding for the $Position_Label$ we will refer to Figure 3(a). This Figure shows a generic seed point that can be surrounded, at most, by



(a)

Point	Binary Coding	Point	Binary Coding	Point	Binary Coding
0	0,0,0	9	0,10,11	18	11,0,0
1	0,0,10	10	10,0,11	19	11,0,10
2	0,10,0	11	10,10,11	20	11,10,0
3	0,10,10	12	0,11,0	21	11,10,10
4	10,0,0	13	0,11,10	22	11,11,0
5	10,0,10	14	10,11,0	23	11,11,10
6	10,10,0	15	10,11,10	24	11,0,11
7	Seed	16	0,11,11	25	11,10,11
8	0,0,11	17	10,11,11	26	11,11,11

(b)

Figure 3: *Relative encoding* (a) *Relative positions to a seed* (b) *Binary Coding*

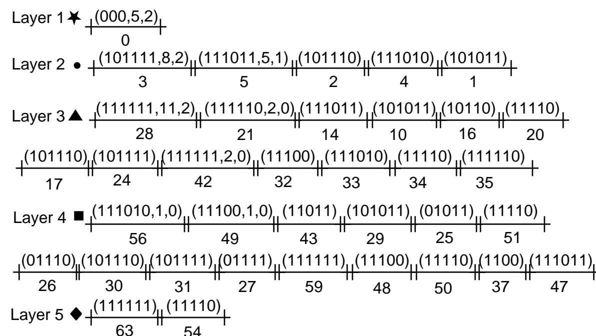


Figure 4: *Layer Encoding in ECI Algorithm*

26 points. The x,y,z coordinates of a point can be obtained from the seed point through increments/decrements of its coordinates. That is, the coordinates of the point are obtained by adding an offset of $\{+1, 0, -1\}$ to the coordinates of the seed point. This offset is represented with a variable length coding where value -1 is represented with one bit 0, value 1 is represented with two bits 11 and value 0 also with two bits 10. Figure 3(b) shows this coding. As an example, a point located in position 13 is encoded as $(-1, 1, 0) = (0, 11, 10)$. The encoding corresponding to the example of Figure 1 is shown in Figure 4. Note that those points that are not seeds are encoded specifying only the *Position_Label*.

The total number of bits needed for encoding each point is calculated considering that from 3 to 6 are required for the *Position_Label*, 5 bits for the connectivity taking into account that the maximum connectivity is 26 (see Figure 3(a)) and a maximum of 5 bits for the *Number_Seeds* field. Note that the number of bits in this last field depends on the connectivity value ψ. In fact, the number of bits required for *Number_Seeds* is $\lceil log_2\psi \rceil$. This value is in range $[1, 5]$. In summary, the number of bits per point is in range $[3, 16]$. According to our simulations an average of 7.7 bits per point are required (see Section 6).

4 Decompression Algorithms

The information, compressed in the CPU, is sent to the graphics pipeline where it has to be decompressed to be processed. In this section we present the decompression procedure for the Cell Identification (CI) and Extended Cell Identification (ECI) algorithms. As a result, the local coordinates of each point inside the cube (i,j,k) with {i,j,k} $\in$ [0,31] are obtained. Global coordinates of a point can be easily computed employing its local coordinates to its cube, the global coordinates of a reference point for the cube and the grid resolution (grid size of the cube).

4.1 CI Algorithm

The decompression algorithm processes the information in the same sequential order employed in the compression procedure. This way, a cell with points is fully decompressed before starting with the following non–empty cell. All points inside this cell are decoded sequentially.

The basic idea is the computation of the coordinates of position labelled as 0 in each cell (see Figure 2(a)) and the use of this information to obtain the coordinates of the other points in the cell by adding an offset to each coordinate. In order to obtain a unified representation with the ECI algorithm we call this point *Seed*. The basic decompression algorithm is outlined in Figure 5(a). For each non–empty cell of the cube, the Seed coordinates are computed from the index of the cell (line 4). Specifically, the local coordinates of position labelled as "0" (Figure 2(a)) in the cell are computed. To do this, the cell index is split into three fields (four bits per field for cubes of $32 \times 32 \times 32$ positions because there are

```
1 Cell_Index = 0;
2 for each CELL ∈ CUBE {
3   if (Cell_Filled = 1) {
4     Compute Seed_Coord from Cell_Index;
5     Load_List ({ Pos });
6     for each POINT ∈ CELL {
7       Position_Label = Translate ({Pos});
8       offset = Decode (Position_Label);
9       Point_Coord = Seed_Coord + offset; } }
10  Cell_Index ++;
11 }
```

(a)

```
1 k = 1; Initialize List_Seed (layer k);
2 for each LAYER k {
3   j=0; /* Index for elements in layer k + 1 */
4   for each SEED in List_Seed (layer k) {
5     Connected = Load points j to j + ψ − 1
                                of layer k + 1;
6     j = j + ψ ;
7     for (i=0 ; i < ψ ; i++){
8       offset = Decode (Connected(i));
9       Point_Coord = Seed_Coord + offset; }
10    List_Seed(layer k + 1) ← First Number_Seeds
                      points of Connected list; }
11  k ++; }
```

(b)

Figure 5: *Decompression* (a) *CI* (b) *ECI*

only $16 \times 16 \times 16$ non overlapping cells) and the coordinates of the seed are obtained by multiplying each field by two. As an example, the index of the cell 67 is 000001000011 so that the coordinates of the seed are $2 \times (0000, 0100, 0011)$, that is $(00000, 01000, 00110)$.

Next, in line 5, the $\{Pos\}$ list is loaded. For each point in a cell, its *Position_Label* is obtained from the $\{Pos\}$ list (line 7). That is, each position i ($i \in [0,7]$) with $Pos_i = 1$ is translated to its *Position_Label* in the cell. As an example cell 0 in Figure 2 is encoded as $\{Pos\}$={1,1,1,1,1,1,0,0}. The first point to consider is Pos_0 = "1" and its corresponding *Position_Label* is "000". The following point is Pos_1=1 with *Position_Label* = "001" and so on.

After this, the coordinates of this point are computed from the coordinates of the *Seed* by adding the corresponding $offset$ for each coordinate (lines 8 and 9). The offset can be "1" or "0". Specifically offsets are (0,0,1) for point {1}, (0,1,0) for point {2} and so on.

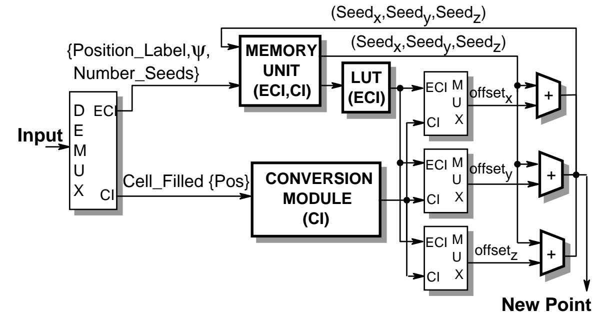


Figure 6: *Decompression Unit*

Finally, the index of the cell is updated and a new cell in the cube is processed. Note that if the cell is empty the only action is the updating of the index.

4.2 ECI Algorithm

The decompression process for the Extended Cell Identification Algorithm is carried out layer by layer. Specifically, for each *Seed* point in a layer, all its connected points in the following layer are decoded, then, the next *Seed* is considered and so on. Seed points inside each layer are processed sequentially and each layer is fully processed before starting with the following one.

The basic algorithm is outlined in Figure 5(b). The process starts with layer $k = 1$ where the only point is considered as seed (line 1). For each layer k two actions are carried out: the neighbors of each seed of the layer k are decoded and the list of seeds of layer $k + 1$ is built.

Seeds of layer k are read from the list of seeds (line 4). The ψ neighbor points of each seed are loaded in list *Connected* (line 5). The processing of the neighbors of each seed is shown in lines 7–9. The coordinates of point in the *Connected* list are computed by adding the coordinates of the Seed point and an offset. This offset is encoded in the *Position_Label* of the point and can be {−1,0,+1} depending on its relative position to the Seed. This offset, has been encoded as {0,10,11} respectively (see Figure 3). Once the points in the connected lists are decoded, the first *Number_Seed* of them are loaded into the list of seeds of layer $k + 1$ (line 10).

5 Hardware Implementation of the Decompression Algorithms

In this section we present the architecture for the decompression algorithms. Both proposals, CI and ECI algorithms, can be integrated in the same hardware unit. Figure 6 shows the block diagram of the hardware implementation for the decompression algorithms.

The input data stream is delivered towards the *Conversion Module* (for CI algorithm) or the *Memory Unit* (for ECI algorithm). The offset for each point is obtained in the *Conversion Module*, for the CI algorithm, or using the *Memory Unit* and the *LUT* (Look–Up Table) for the ECI algorithm. The coordinates of the seed for both algorithms are computed in the *Memory Unit*. Finally, once the offset and the seed coordinates are known, the coordinates of the point are computed in the final adders. In the following, these modules are described in detail.

5.1 Conversion Module

This module (see Figure 7(a)), employed only for the CI algorithm, computes the offset of each point from the $\{Pos\}$ list that represents the positions of the points inside the cell.

The $\{Pos\}$ list is stored in the 8–bits input register. Specifically, REG_i stores Pos_i. Initially, the priority encoder receives as input the content of this register and provides the position label corresponding to the first point with $Pos_i = 1$. Then, to process the next point in the list, the bit i of the register is cleared. This process is repeated until the last point in the list is detected. Finally, the offset of each coordinate is spanned to two bits by concatenating a "0".

5.2 Memory Unit

The block diagram of this module is shown in Figure 7(b). The unit computes the seed coordinates for the CI algorithm and the *Position_Label* and seed coordinates for the ECI algorithm. It is composed of two shift registers, for the seed and connected list, and a counter to obtain the seed coordinates for the CI algorithm. The data indicated in the figure corresponds to the first two layers of Figure 1 (ECI algorithm).

In the Seed List, the coordinates of each seed are stored. In case of the ECI algorithm these coordinates are obtained in the final adders (see Figure 6) and stored back. In the CI Algorithm the seed coordinates are obtained in the "Seed Calculation" unit (line 4 in Figure 5(a)). This module consists of a counter to obtain the Cell Index which is split into three fields. The coordinates of the seed are obtained concatenating a "0" as least–significant bit of each field. Each time a new cell is processed, the counter is incremented and the coordinates of the new seed are obtained.

The control unit determines if a new seed has to be loaded in the list and the shifting of the seed list.

The memory unit stores the connected list for the ECI algorithm. Each point in the connected list is specified by its position label, its connectivity (ψ) and the number of seeds in the following layer. In each cycle the *Connected_List* is shifted so that the point in the first position is read and a new point is stored. The control unit determines when a new point is loaded and the number of bits that are required for its representation[1].

As the *Position_Label* has a variable number of bits, the *Length Adapter Module* spans this field to 6 bits in order to obtain the offset in the LUT.

5.3 LUT

As shown in Figure 6, this module obtains the offset of each point from its *Position_Label* (ECI algorithm). It is implemented as a 64×6 table. The position label is employed to address the LUT and the corresponding offset per coordinate is read. This offset inputs the adders.

5.4 Hardware Requirements

As was shown in the previous sections, the hardware is simple. The main components are a storage system for the seed and connected list, register and priority encoder for the decoding of the CI algorithm, a set of adders to compute the coordinates of each point, and the required control for the global system. We have shown that both algorithms can be implemented in a single hardware unit providing the control to select between the CI or the ECI algorithm.

[1] Note that the *Position_Label* and the *Number_Seeds* have been encoded using a variable length code.

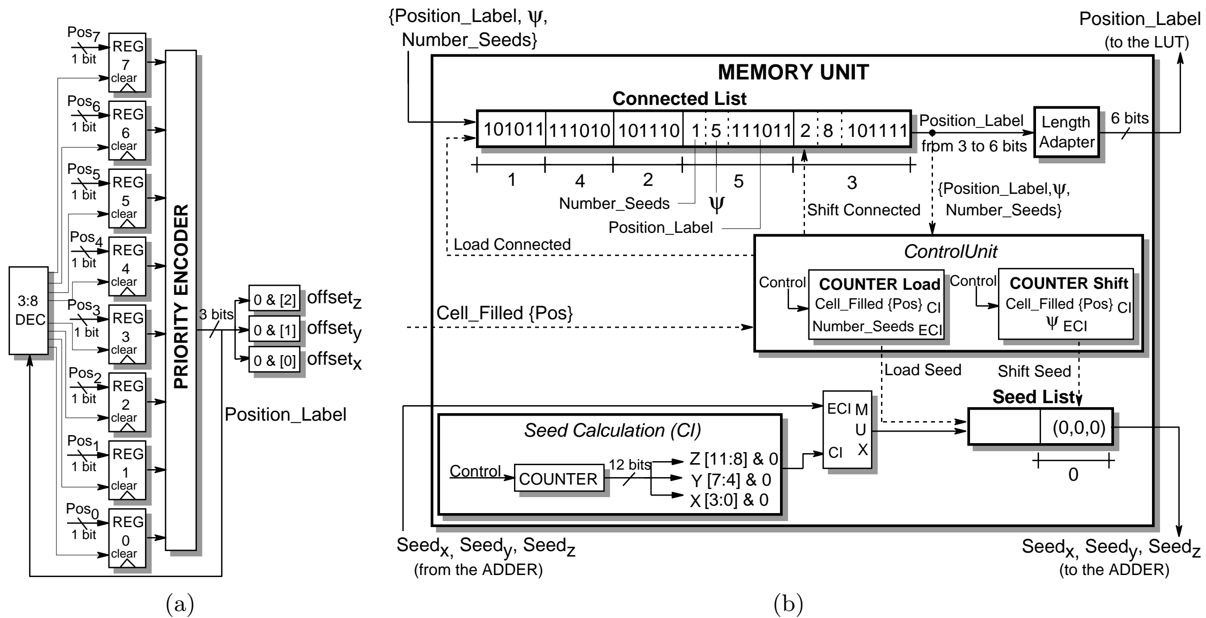


Figure 7: (a) *Conversion Module* (b) *Memory Unit*

6 Evaluation of the Algorithms: Compression Ratio

The compression ratio we summarize in this section has been obtained after a software simulation of both algorithms using a set of images. For each algorithm we compute the number of bits required to encode the points in the cube and to evaluate the benefits of our proposals, we compare the results with two reference implementations.

In both cases we compare the results with a basic implementation without compression: 1 bit per position in the naive distribution and 10 bits per position[2] (5 for the position label and 5 for the connectivity) in the layer distribution.

CI Algorithm As explained in Section 3.1, empty cells are encoded with just 1 bit, and non–empty cells with 9 bits. This means that, for non–empty cells, the number of bits per point is in $[1.125, 9]$, where 9 bits per point are required when the cell has only one point, and 1.125 bits per point when the cell has 8 points.

Our experimental results show that an average 80% of the cells are empty (for non–empty cubes). These cells can be encoded with only one bit (*Cell_Filled* bit). On the other hand, the number of points in each non–empty cell is variable and, consequently, the number of bits per point is also variable: Cells with more points require less bits per point. On average and for the different images employed, 5.0 bits per point are used in the CI algorithm.

To evaluate the benefits of our proposal, a reference algorithm is employed. In this case, a basic representation with only 1 bit per position is employed where a 1 indicates that the position has a point, and 0 otherwise. This means that a cube requires $32 \times 32 \times 32$ bits. On average, the reference algorithm needs 29.5 bits per point. These numbers show that the CI algorithm reduces the communication requirements by about 83%.

ECI Algorithm. As explained in Section 3.2, the number of bits per point is in the range $[3, 16]$. This is due to the fact that not all points are seeds, and that a variable length code is associated with the *Position_Label* and *Number_Seeds* fields. 3 bits is the minimum number of bits required for a non–seed point, whereas 16 bits is the maximum number of bits required for a seed point. Our simulations show that the average number of bits per point is 7.7.

The reference algorithm employed for comparison employs 10 bits per point, 5 bits to specify the relative position of the point

[2] A point is surrounded by 26 points at most.

to the seed and 5 bits for the connectivity. Then, the ECI algorithm reduces the communication requirements by about 20%.

7 Conclusions

We have presented two encoding schemes for two different data distributions in point rendering. For the naive distribution we propose the CI algorithm. This algorithm is based on the splitting of the cube in non overlapping cells and the efficient encoding of the points inside each cell. A specific encoding is proposed for the empty cells, in such a way that just 1 bit is enough for their representation. With this technique we obtain a high compression ratio (5.0 bits/point), which reduces the transmission and storage requirements by about 83% with respect to a reference algorithm proposed for the evaluation.

For a layer distribution, we propose the ECI algorithm. It is based on the identification of layers of neighboring points and the efficient encoding of the connectivity among them. The points inside a cube are encoded differently depending on whether they are seeds or not. Moreover, variable length codes are employed to specify the positions of the points with respect to their seed. With this algorithm 7.7 bits per point are required; which represents a reduction of 20% with respect to a reference implementation.

We present a unique hardware unit for the decompression of both proposals. The resulting architecture is simple, regular and suitable for its hardware implementation.

8 Acknowledgements

This work was supported in part by the Ministry of Science and Technology of Spain under contract MCYT–FEDER TIC2001–3694–C02–01 and by the Secretaria Xeral I+D of Galicia (Spain) under contract PGIDIT03TIC10502PR.

REFERENCES

[Alexa03] M. Alexa, M. Gross, M. Pauly, M. Zwicker, H. Pfister, M. Stamminger, and C. Dachsbacher. Point–Based Computer Graphics. In *Eurographics Tutorial*, 2003.

[Amor02] M. Amor, M. Bóo, A. del Río, M. Wand, and W. Straßer. A New Algorithm for High Speed Projection in Point Rendering Application. In *Euromicro 2003. Workshop on DSD*, pages 136–149, 2002.

[Gross98] J. P. Grossman. Point Sample Rendering. In *Master's thesis, Dept. of Electrical Engineering and Computer Science*, 1998.

[Gumho98] S. Gumhold and W. Straßer. Real–Time Compression of Triangle Mesh Connectivity. In *Siggraph'98*, pages 133–140, 1998.

[Gumho99] S. Gumhold, S. Guthe, and W. Straßer. Tetrahedral Mesh Compression with the Cut–Border Machine. In *IEEE Visualization '99*, pages 51–58, 1999.

[Malló02] P.N. Mallón, M. Bóo, M. Amor, and J.D. Bruguera. Concentric Strips: Algorithms and Architecture for the Compression/Decompression of Triangle Meshes. In *3D Data Processing Visualization Transmission*, pages 380–383, 2002.

[Pfist00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface Elements as Rendering Primitives. In *Siggraph 2000*, pages 335–342, 2000.

[Ren02] L. Ren, H. Pfister, and M. Zwicker. Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. In *Eurographics*, 2002.

[Rossi99] J. Rossignac. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Trans. on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[Rusin00] S. Rusinkeiwicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Siggraph 2000*, pages 343–352, 2000.

[Szymc99] A. Szymczak and J. Rossignac. Grow & Fold: Compression of Tetrahedral Meshes. In *ACM Symp. on Solid Modeling and Applications*, pages 54–64, 1999.

[Wand01] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The Randomized z–Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Siggraph 2001*, pages 361–370, 2001.

[Zwick01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface Splatting. In Eugene Fiume, editor, *Siggraph 2001*, pages 371–378, 2001.