Point Cloud Rendering System

Technical report

Lukáš Maršík

Brno University of Technology
Faculty of Information Technology

Brno, 2010

Contents

1. INTRODUCTION

The approach described in this report is based on the idea that direct point cloud rendering, which is in the principle not too complicated, can be efficiently implemented in modern graphics cards, programmable or custom hardware. Such implementation can be useful not only for its performance but especially for the possibility to include it into solutions that require 3D graphics output also in non PC environments and in embedded solutions with low power consumption, etc. The point cloud model rendering is very interesting alternative to the most common polygon rendering (for example OpenGL), especially when scanned point cloud model is directly available.

The principles of point cloud rendering are step-by-step described in second section. The third section contains description of demo software programmed for PC and also photographs of system output and experimental system as whole are included. Last a few sections contain published papers and conclusion of stated work.

2. POINT CLOUD RENDERING FUNDAMENTALS

The point cloud rendering algorithms described in this report relies on rendering of the single oriented points forming the point cloud.

2.1 Point Element Shape

One of the feasible geometrical representations of the scene element – oriented point – is an oriented circle whose projection is a general ellipsis. See Figure 1.

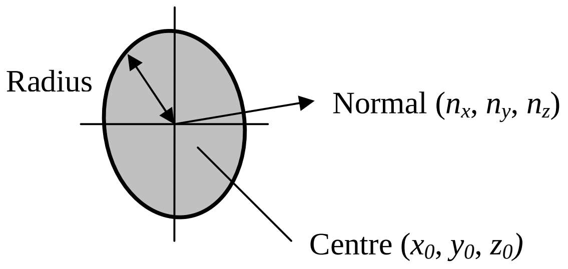

Figure 1: Point element projection

The rendering algorithm can be subdivided into several principal parts [Zemc04][Hero05][Tisn02][Mars08]:

1. Projection of the elements' positions into 2D screen and Z space and computation of the corresponding elements' projected normal and radius.

2. Evaluation of the elements' color (lightness) based on the projected normal vector, element local lighting model (material), and the light sources' and observer's parameters.

3. Rendering of the elements (ellipses) into the image frame buffer (visibility solved using Z-buffer).

In the proposed approach, all of the parts are performed on regular PC. However hardware acceleration is possible using DPS and FPGA chips (see [Mars08]). Then all the part 1 can also be performed through the host processor (DSP), part 2 can also be performed partially in the DSP through access to the pre-calculated reflection and diffusion tables and the final color evaluation is done in the FPGA, and the part 3 is performed in the FPGA.

2.2 Representation of Shapes

The point shapes are pre-calculated. The idea of pre-calculation is based on the fact that the normal vector can be converted into two dimensions according to the below equation and quantized and thanks to the fact that radius can be quantized as well.

$$(1) \quad \vec{n} = (n_x, n_y, n_z) = k(n'_x, n'_y, 1) = k\vec{n}'$$

The shapes of ellipses (circle projections) can then be stored in tables, indexed by quantized point size and normal vector, the size of the table is 17x61^2 (n'_x, n'_y are quantized as 61 values, the particle size as 17).

An efficient method is used to compress the point bitmap, which uses namely the fact that the point shape is convex and symmetrical. Based on experiments on practical data and hardware implementation issues, the bitmap size is defined to be 8×8 pixels. For small points (whose shape can fit into the 8×8 raster), symmetry is exploited to minimize the representation size. Experiments show that majority of points rendered by the system meets the extent of small points and their processing is thus efficient. However, even larger point shapes appear in the rendered set, whose encoding takes larger number (inherently not limited, but practically 2×2) of fractional bitmaps. See Figure 2 and Figure 3 for examples of small and large point shapes.

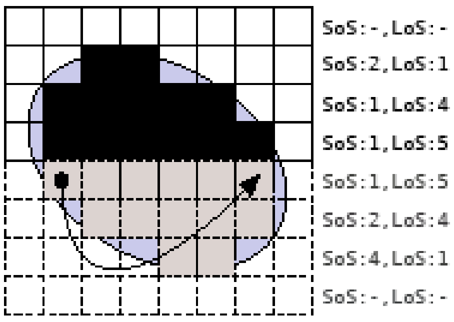

Figure 2: Small point shape evaluation

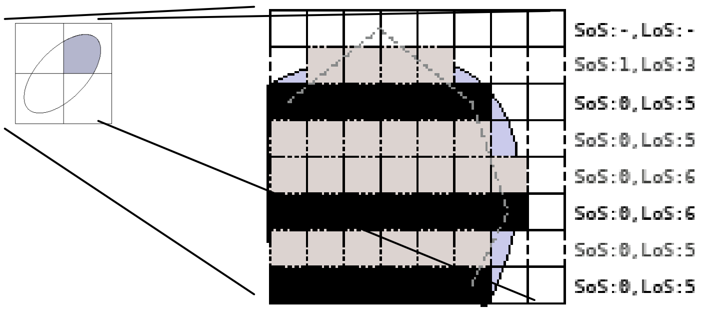

Figure 3: Large point shape evaluation

2.3 Representation of Colors

As for the color model, the proposed solution allows for Phong model with diffuse and specular parts but limited to a single level of specular reflection and single color of light sources presumably white.

$$(2)\ \hat{I} = \hat{I}_0 + \hat{k}_{d,Material} I_{Diffuse} + \hat{k}_{s,Material} I_{Specular}$$

where *Diffuse*, *Specular*, and *Material* are parameters sent to the rendering engine for every element while color $\hat{I}_0$ and both $\hat{k}_d$ and $\hat{k}_s$ color tables are stored in the rendering engine.

The elements' properties, as evaluated by steps 1. and 2. described above through the pre-calculated table, form a code-word of 64 bits which enters the rendering Engine. See Figure 4 for the code-word's structure. Note, please, that the *y* co-ordinate is missing from the code word as the engine generates it implicitly. Detailed description of the rendering machine is in the following section of the report.

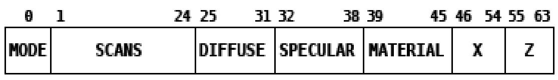

Figure 4: Point code word

2.4 Rendering Engine

Each of the point elements could be rendered very quickly, just through interpreting the key word that is simple and the interpretation can easily be done in parallel (using graphics card with CUDA/OpenCL or FPGA) except for the natural bottleneck lying in the number of pixels affected by each point element that can be up to 64 (8x8 blocks).

In our approach, however, this bottleneck is overcome through subdividing of the raster image memory and Z-buffer memory, used for rendering output, into 8 parts so that all the lines of the element image (8x8 pixels) can be rendered in parallel. As the shape of the element is encoded in very simple way (see above), the rendering lies merely in updating the Z-buffer and conditional writing of a color value into the output raster. As the Z-buffer and raster memory have 4 pixels per word (oriented horizontally), 8 pixels can be updated within 3 accesses in the memory as the data is not always word aligned.

Because the size of the raster and Z-buffer memory especially in the FPGA is limited, our implementation can slides a narrow window (e.g. 16 pixels high and as wide as the image) across the output image line by line so that only that narrow window of raster and Z-buffer are present in the on-chip memory in the FPGA. The already processed part of the image is flushed out of the rendering engine. This approach allows for evaluating only those elements whose *y* co-ordinate is in the centre of the sliding window. This fact enforces sorting of the elements according to the *y* co-ordinate and sending them into the rendering engine in the *y* order. While the sorting operation seems to cause high computational complexity, the fact is that the *y* range is limited and as the co-ordinates are anyway integer, the sorting can be seen as merely subdividing the point cloud (set) into several subsets with linear complexity.

See Figure 5 for the illustration of sliding window. Note, please, that 8 pixels high window would be sufficient for rendering but the extra lines are suitable for buffered flushing of the output and buffered loading of the input (initialized raster and Z-buffer).

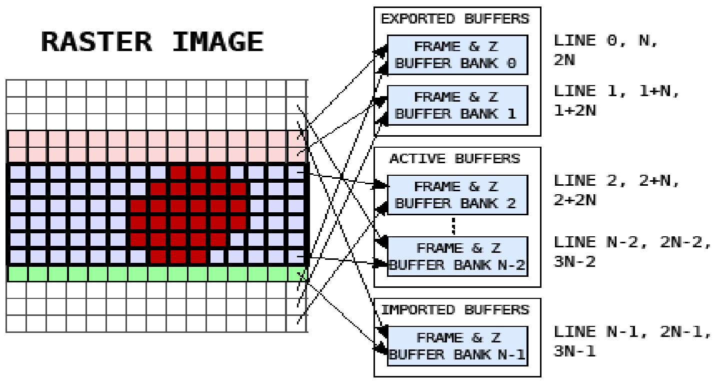

Figure 5: Sliding window over the raster

3. SOFTWARE DEMO

The software demo is programmed in C++ using the OpenCV library for simple and multiplatform window creation, image showing and handling its real-time animation. Software simply reads point cloud from model file, store them and for each of frame of animation perform particle projection, encoding and rendering as described above. Moreover particle size and color can be also defined, so is possible to shade (with Phong model as described above) appropriately large particles using various materials (with red, green and blue components). There can also be optionally set initial background of each frame of scene (see Figure 6).

Software running on the regular laptop is able to perform rendering with rate 30 and more FPS (the actual frame rate is printed out to the console). Using the CUDA or OpenCL for programming the rendering part of software and running rendering algorithm on graphics card can rapidly increase frame rate. Another very powerful option is usage of a specialized hardware as a combination of DSP (projecting particles and encoding into the codeword) and FPGA chip (rendering). Representative of such system can be PC with CAMEA Uni1p PCI board complemented with DX64 acceleration modules (see Figure 7), as stated in [Mars08].

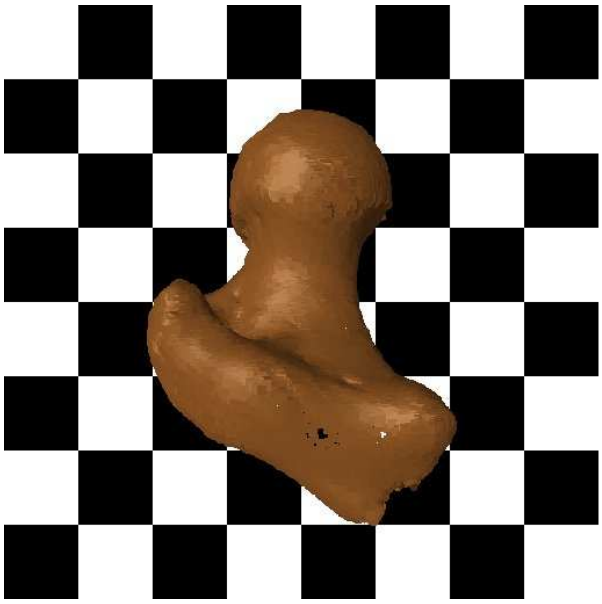

Figure 6: Software demo output (point cloud model of bone with 25816 elements)

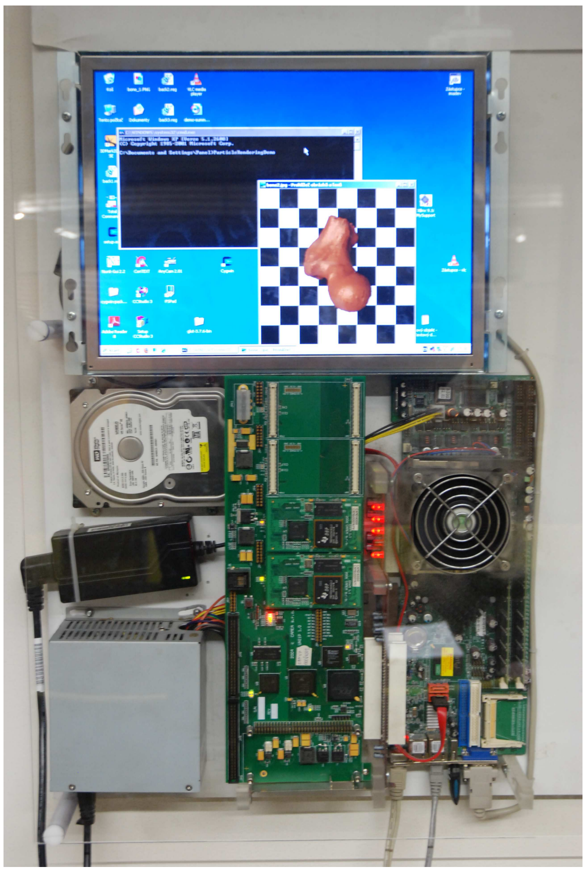

Figure 7: PC system with Uni1p board and DX64 modules

4. CONCLUSION

The presented demo software shows alternative approach to scene rendering. Instead of using triangles as a primitive we decided to use oriented ellipses as representatives of particular points. This is advantage especially if we directly have list of scanned points, then no conversion into the polygonal mesh is necessary. The rendered scene can be realistic enough thanks to the shading model (including materials) and point cloud resolution. It has been shown that rendering of point cloud can be well paralleled and hardware accelerated using modern chips.

5. PUBLICATIONS

Published:

- Zemčík, P., Maršík, L., Herout, A.: Point Cloud Rendering in FPGA, Proceedings of WSCG 2009, Plzeň, Czech Republic, 2009

6. REFERENCES

[Gros02] Gross, M.: Point Based Computer Graphics, Proceedings of SCCG 2002, Budmerice, Slovakia, 2002

[Hero05] Herout, A., Zemčík, P.: Hardware Pipeline for Rendering Clouds of Circular Points, In: Proceedings of WSCG 2005, Plzeň, Czech Republic, 2005

[Mars08] Maršík, L.: Image processing in FPGA, Bc. Thesis, Brno University of Technology, Brno, Czech Republic, 2008 (in Czech)

[Rusi01] Rusinkiewicz, S.: Surface splatting, Proceedings of SIGGRAPH 2001, USA, 2001

[Tisn02] Herout, A., Tišnovský, P.: Vector Field Calculations on a Special Hardware Architecture, In: East-West-Vision 2002 Proceedings, Graz, TUV, Austria, 2002

[Zemc03] Zemčík, P., Tišnovský, P., Herout, A.: Particle Rendering Pipeline, Proceedings of SCCG 2003, Budmerice, Slovakia, 2003

[Zemc04] Zemčík, P., Herout, A., Crha, L. Tupec, P. Fučík, O.: Particle rendering pipeline in DSP and FPGA, In: Proceedings of Engineering of Computer-Based Systems, Los Alamitos, USA, IEEE CS, 2004