Point Cloud Rendering in FPGA

Pavel Zemčík
Faculty of Information Technology
Brno University of Technology
Božetěchova 2
Czech Republic, 612 66, Brno
zemcik@fit.vutbr.cz

Lukáš Maršík
Faculty of Information Technology
Brno University of Technology
Božetěchova 2
Czech Republic, 612 66, Brno
xmarsi03@stud.fit.vutbr.cz

Adam Herout
Faculty of Information Technology
Brno University of Technology
Božetěchova 2
Czech Republic, 612 66, Brno
herout@stud.fit.vutbr.cz

ABSTRACT
This contribution describes recent development in ongoing work focused on point cloud rendering algorithm implementation usable in environments containing programmable or custom hardware. The approach described in this paper is based on the idea that direct point cloud rendering, which is in the principle not too complicated, can be efficiently implemented in 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 in non PC environments and in embedded solutions with low power consumption, etc. This contribution describes the overall approach and the current results.

Keywords
Point cloud, programmable hardware, FPGA, hardware rendering.

1. INTRODUCTION
3D point-clouds [Rusi01][Gros02] represent a modern trend in computer graphics. They can be used for an alternative representation of graphics scenes instead of traditional entities, such as planar polygons or triangles, surface patches, etc. Geometrically, point-clouds are "internally unorganized" sets of points and as such they are very easy to handle during various graphics operations. On the other hand, point-clouds require large amount of memory to represent common objects and that is the reason why they have not been widely used historically. In recent few years, the situation changed. Not only the recent computer systems, including embedded and DSP systems, can be are equipped with cheap and large memory, but also the today affordable 3D scanning devices can create object models based on point clouds directly. To visualize these point clouds, however, methods to convert them into surface patches are still often used and/or GPUs are mostly used to render these structures although for direct point cloud rendering, the GPUs are often inefficient. The proposed approach offers an alternative based on a simple engine for direct point cloud rendering based on custom hardware circuits in the form of programmable hardware or custom chip. This alternative will most probably not be of wide use in contemporary PCs whose graphics performance, thanks to the modern GPUs, is very high and at the same time the PCs are relatively cheap. However, the approach, if successful, can serve as a model for future high performance implementation, e.g. as a module in some future piece of hardware or it can be used in embedded systems where exploitation of GPUs is not feasible at all and also in systems with limited power consumption, where exploitation of GPUs is also very problematic.

The recent development in programmable hardware, specifically Field Programmable Gate Arrays (FPGAs) [Curd07][Hite05] offers a good platform for power efficient, cost efficient, and also high performance implementation of point cloud rendering, which is not quite supported by the traditional computer graphics manufacturers

The proposed solution is characterized by using low amount hardware resources (system logic and fast memory) – offering possibility to embed numerous rendering engines on a chip, low power consumption, low heat emission and other features important for non-desktop applications.

The solution describer in this paper is an extension to previous work on FPGA accelerated point cloud rendering and adds higher performance due to both

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

technology and solution itself, parallelism, and extended color model [Zemc03][Zemc04].

2. POINT ELEMENT

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

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.

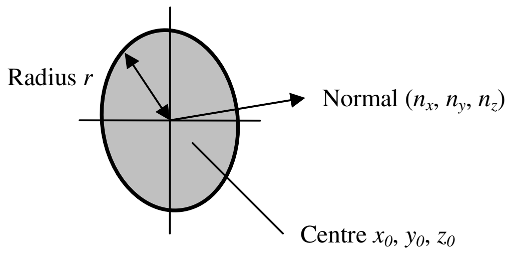


Figure 1: Point element projection

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 the part 1 is performed through the host processor (DSP), part 2 is 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.

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.

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

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 17x64^2 (n'_x, n'_y are quantized as 64 values, the particle size as 17).

An efficient method is used to compress the point bitmaps is used, 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.

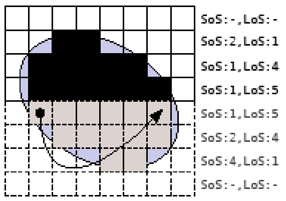


Figure 2: Small point shape evaluation

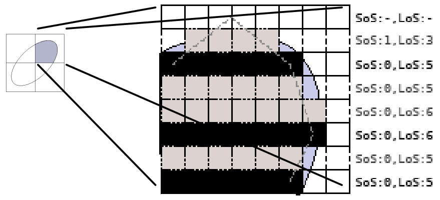


Figure 3: Large point shape evaluation

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 paper.

0	1 24	25 31	32 38	39 45	46 54	55 63
MODE	SCANS	DIFFUSE	SPECULAR	MATERIAL	X	Z

Figure 4: Point code word

3. SYSTEM ARCHITECTURE

The rendering engine is experimentally implemented on a PCI board placed in a host PC.

System board

The board (UNI1P-VUT manufactured by CAMEA, Brno) contains PCI controller, DRAM memory, and programmable hardware including FPGAs to control the data flow. The board is capable of carrying up to four DSP/FPGA computational engines (DX64), each with a DSP (TI C6416), FPGA (Xilinx VIrtex II), and DRAM (128MB). Block diagram is shown in the Figure 6.

Each of the DSP/FPGA boards implements single rendering engine.

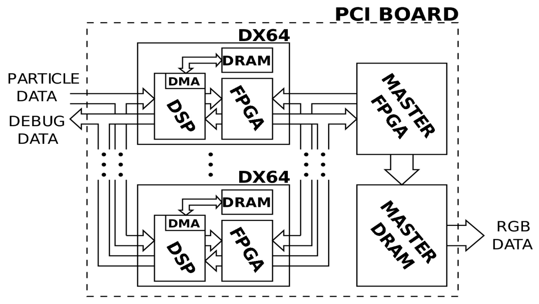


Figure 6: System architecture block diagram

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 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 in the FPGA is limited, our implementation 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 seem 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 7 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).

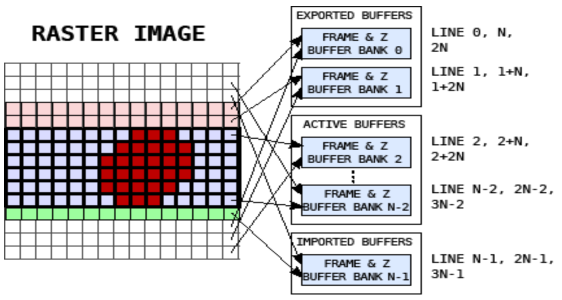


Figure 7: Sliding window over the raster

Rendering chain

The above described approach is well parallelizable. As the rendering engine is capable of uploading of the initialized image and Z-buffer and flushing the processed output with relatively low delay (processing of 16 lines of the window), it is possible to chain the rendering engines as shown in Figure 8 and when randomly data distributed among the engines lead into nearly linear speedup.

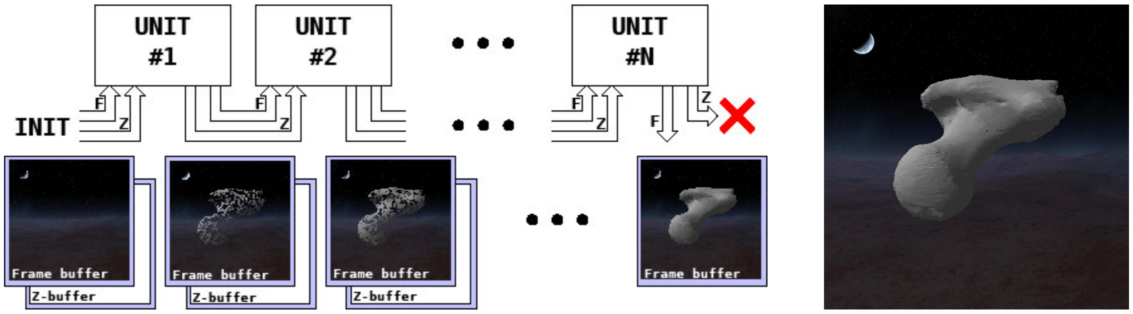


Figure 8: Processing chain block diagram and the rendering output

The block diagram of the engine itself is shown below. Note, please, that the "writer block" contains the raster and Z-buffer on-chip memory.

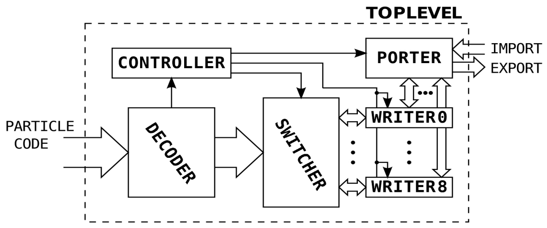


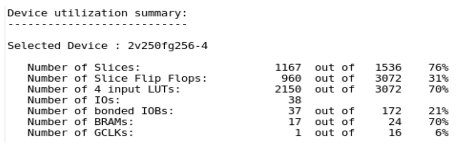
Figure 9: Single engine block diagram

4. PERFORMANCE

The performance of the approach can be evaluated in terms of the point elements rendering speed and in terms of the resource consumption.

Resource consumption

The consumption of FPGA resources on the Virtex II chip is shown in Table 1 below. Overall, the design consumes approximately 60 percent of the small Virtex II-250 chip.



```
Device utilization summary:
---------------------------

Selected Device : 2v250fg256-4

   Number of Slices:                    1167  out of   1536    76%
   Number of Slice Flip Flops:           960  out of   3072    31%
   Number of 4 input LUTs:              2150  out of   3072    70%
   Number of IOs:                         38
   Number of bonded IOBs:                 37  out of    172    21%
   Number of BRAMs:                       17  out of     24    70%
   Number of GCLKs:                        1  out of     16     6%
```

Table 1: Consumption of the FPGA chip resources

Rendering speed

The achievable rendering speed is affected by the clock speed of the FPGA structure. Currently, the achieved clock speed is 100 MHz and 4 clock cycles are needed to evaluate single element. At the same time, 4 rendering engines run in parallel so the raw rendering speed of 100 million (10^8) elements per second. The useful rendering speed is affected by the idle times. If e.g. video is generated, the worst idle time is 0.02 s. The rendering speed for *N* elements is:

$$(3)\ t_{\mathrm{Im}age}(s) = \max\left(0.02, N/10^8\right)$$

5. CONCLUSIONS

It has been demonstrated that the direct point cloud rendering engine in FPGA and DSP is feasible and that it can be built using relatively simple resources in programmable hardware. While the raw speed of the single engine does not reach extremely high figures, the approach is easily scalable and parallelizable and can be also used in embedded environments.

6. ACKNOWLEDGMENTS

This work has is supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics), and by the research project Security-Oriented Research in Information Technology, MSM0021630528.

7. REFERENCES

[Curd07] Curd, D.: Power Consumption in 65 nm FPGAs, White Paper: Virtex-5 FPGAs, February 1, 2007, XILINX, San Jose, CA, USA

[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

[Hite05] Hitesh, P.: Synthesis and Implementation Strategies to Accelerate Design Performance , White Paper: Virtex-4 and Spartan-3 Families, July 6, 2005, XILINX, San Jose, CA, USA

[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