EG UK Theory and Practice of Computer Graphics (2010)
John Collomosse, Ian Grimstead (Editors)

Rendering Large Point Datasets with GPU Shaders

Hugo Aguirre[1], Diego Gutierrez[1] and James S. Perrin[2]

[1]Universidad de Zaragoza, Spain
[2]University of Manchester, UK

Abstract

This paper demonstrates how programmable GPUs are a powerful tool to display large point datasets at an interactive frame rate. Point datasets are commonly used to analyse and solve complex problems, but rendering them is always an expensive task in computational terms. This paper researches the possibilities that GPUs and shading languages offer for rendering large datasets on modest computers and the improvements in speed and quality. GPU techniques to represent scalar and vector glyph are also described. The GPU method is compared with common methods, such as using polygons or textures. The shader glyphs are drawn onto planar primitives using equations to generate surface, lighting and depth information. The improved computational efficiency allows the display of larger datasets with a simultaneous increase in visual quality.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture

1. Introduction

Computer graphics have made significant progress for scientific research in several branches, such as biology, physics, astronomy, chemistry and medicine. Many researchers study their datasets in a visual way, which allows them to have a clearer analysis of individual data relations and the importance of them when analyzed together. The larger the datasets are the harder it is to handle them and obtain results. Scientists usually need huge datasets of point data to understand the behaviour of phenomena. Nevertheless, the computational cost of visualizing such huge datasets is high and scientists have to work with samples or fragments of the datasets. Hence, it is important to make improvements in this area in order to obtain more efficient methods, consequently making it possible to use bigger datasets.

Point data's main characteristic is usually its position in space. In addition to this, point data can have more additional attributes which can be shown in visual terms such as size, colour, direction or shape. Therefore, point data needs to be represented visually by geometrical shapes. These can be two-dimensional, such as circles and polygons, or three-dimensional, such as spheres, polyhedrons, cylinders, cones, and arrows. The latter have a larger computational cost but they are usually more visually attractive and widely used in scientific study. Several of the included references show the use of three-dimensional shapes.

This research is based on methods proposed in the papers referred to below. It is remarkable to state that the papers and documentation found only declare success, but they do not explain how they developed their methods. Although our proposal is not entirely new, there are meaningful differences with former papers since it is not only important how the sphere is built but also how a large number of spheres are rendered in real time and how they are passed data. This paper compares several options to render large datasets in order to determine future work. An additional aim of this paper not discussed in the previous work is the rendering of cones, which are used for spatially oriented data i.e. vectors. The majority of point data have scalar variables but there are many cases in scientific visualization where vector data needs to be analysed; flows and forces. Scalar values can also be abstractly mapped to direction when looking for correlations in the data.

The final goal of this paper is to show how utilizing the GPU via Shader programming can improve performance over established techniques (for spheres and cones). The performance tests use a randomly generated dataset within a specified region of space to avoid characteristics of a partic-

© The Eurographics Association 2010.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG10/031-037

ular dataset affecting the results. Results are compared with the other usual graphic methods. The shader method was tested with the GLSL 1.1 but GLSL 1.3 can also be used.

2. Previous work

Research with three-dimensional model simulations has been carried out for a long time. For example, Krogh, Painter and Hansen [KPH97] examine performance improvements in the visualization of large datasets in three-dimensional simulations. In more recent works, e.g., Liang et al. [LMC05] use parallel rendering for visualizing molecular dynamics data on supercomputers. Gribble and Parker [GP07] apply lazy evaluation to avoid expensive lighting effects. The paper presented by Zemcik et al. [ZTH03] is based on a custom algorithm design for hardware. While Zemcik et al. want a specific hardware with high efficiency but low portability, this paper studies how programmable GPUs allow different algorithms to work on different hardware. The result is that the same algorithm can be both executed or updated on improved GPU hardware.

There are several papers that treat the topic of making geometrical shapes with shading languages. To implement a sphere, Toledo and Levy [TL04] suggest using a ray tracing algorithm over polyhedrons including cubes, icosahedrons, or points. Taylor [Tay04] achieves good results rendering non-polygonal objects with GPUs. He performed his research in a similar way to Ranta, Singh, and Narayanan [RSN06], where each sphere is rendered onto a quad.

The direct use of GPU shaders for certain rendering techniques allow programmers to maximise the size of the datasets that their graphic applications can handle. The use of shading languages is not always an advantage; they can work very well for some applications, primarily graphic ones [Pau07]. GPU applications usually offer better performances for high intensity computations [JZC*08]. It is hard to debug errors in GPU programming as is pointed out in numerous articles [DHH05]. Although there are new proposals in this area [Hil06], the main method is called visual style. This means the output colour for the fragment shader is set with a value in order to flag the content of the variable.

3. Efficient Rendering of Spheres using Shaders

We first consider rendering point datasets without orientation using the sphere glyph. Although conceptually simple it is an expensive glyph because of the high number of vertices needed to approximate the smooth surface. Nevertheless, spheres are the usual choice due to some advantages. It the most natural representation of atoms, particles and other data sample, which means a greater acceptance and understanding in scientific research. The fact that it is rotationally invariant is most useful.

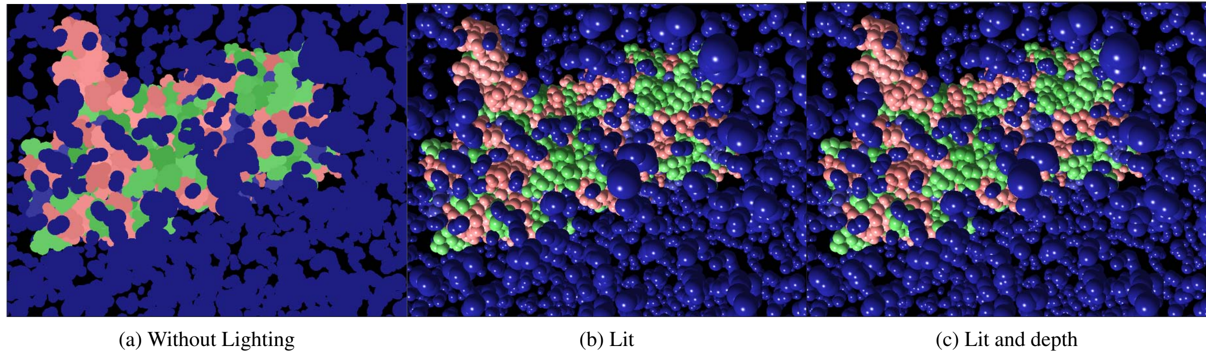
The approach here does not use a polygonal approximation but computes the surface of the sphere directly. Ranta, Singh, and Narayanan [RSN06] present a method which uses a dummy quad to create geometrical shapes using shading languages. One of the most efficient ways to show a fake sphere is to draw it onto a plane. However, it presents a series of difficulties such as lighting and depth computation since a plane has a flat surface. These problems are solved within the shader code.

In the vertex shader several operations are carried out. The quad must maintain its normal pointing at the view plane the camera moves around, or else the quad will appear edge on. So billboarding is applied before the position is set. In the fragment shader the sphere is simulated. In reality, it is a circle which is lit to look like a sphere giving a sensation of depth. By default, it is not possible to know where the fragment within the quad is, nor the order in which each fragment is processed. Therefore, assuming an origin at the centre of the quad, each vertex of the quad is assigned to a local coordinate, from -1 to 1. These values are interpolated for each fragment when passed to the fragment shader. A simple operation gives the distance of the quad fragment from the centre, fragments further than the radius are eliminated. This can be done in two ways, setting the opacity to zero or issuing a *discard* instruction. If available, the *discard* instruction is faster since the shader jumps to the next fragment immediately. Otherwise, the fragment's colour is set, which is a necessary output.

At this point, the silhouette is achieved but it needs further work to simulate a sphere (Figure 1a). Lighting is fundamental to achieve realistic effects and programmable GPUs allow using per-fragment lighting to increase visual quality. The Blinn-Phong lighting model [Bli77] is implemented in the shader. This model takes advantage of using directional lights since the half vector, which is needed for lighting operations, is only calculated once per frame. The half vector is provided by the GLSL core. A new normal is still needed because it is intended to light the quad as a sphere. In figure 2, the 'x' and 'y' normal coordinates are the fragment position but the 'z' is still unknown. The line from the centre to the fragment point, the quad normal and the radius form a triangle where the length of the base and the radius (the hypotenuse) are known. Therefore, the height is found and realistic lighting is created (Figures 1b and 3).

Although fake spheres have been obtained, there is still a defect in the visualization. There are no intersections among the glyphs. Spheres are drawn over a flat quad, so the depth buffer only takes into account whether primitives are in front or behind each other. This is a problem which appears if spheres are drawn onto textures too, but for textures, it is not possible to solve with the fixed pipeline. The depth buffer of the OpenGL fixed pipeline can not be replaced by an exact formula in the shading code. Furthermore, it is recommended to set the depth buffer with the same equation for all fragments. This is because it is not possible to guarantee the same depth results mixing different formulae including the

© The Eurographics Association 2010.



(a) Without Lighting (b) Lit (c) Lit and depth

Figure 1: On the left is an example of the technique without lighting. Fake spheres are achieved applying Billboarding to the quad and eliminating the fragments out of the circle. In the middle lighting is added so a depth sensation is obtained. On the right the depth is fixed so occlusions are calculated properly.

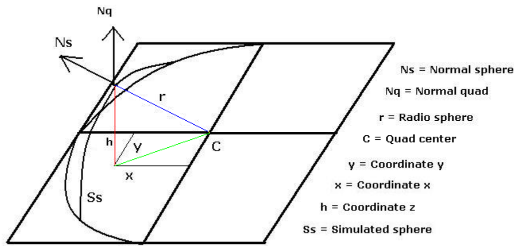


Figure 2: A new normal (Ns) for the simulated sphere is calculated in order to replace the normal quad (Nq). Thanks to the 'x' and 'y' coordinate of the fragment position, a point can be found where a triangle can be made with two known sides: the distance from the centre to the fragment point (green line) and the radius of the sphere (blue line). So the 'z' coordinate is known (red line)

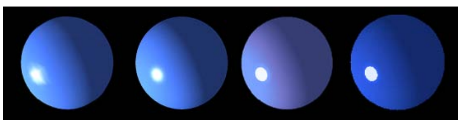


Figure 3: Spheres created by different methods. From left to right: polygons, shaders and two texture methods.

fixed pipeline one [Ros04]. There is not a fixed depth function to use. Therefore, several functions have been tested and studied to observe their results. The formulae give similar visual results. The following equation, appearing in a paper written by Yuan [Yua08], is chosen due to a slight performance benefit.

$$Z = \frac{Z\int_f}{Z\int_f - Z\int_n}\left(1 - \frac{Z\int_n}{d}\right) \quad (1)$$

After fixing the lighting and depth, the simulated sphere is achieved (Figure 1c). More investigation is carried out in order to optimize the performance and at the same time, maintain the visual outcome.

- GLSL variables can be sent either to the vertex shader or the fragment shader in several ways: lists, arrays or buffer objects. The results show that the main advantages are different depending on the graphic card used. On the one hand, the GeForce 6 card offers good results for lists and variables sent to vertex shaders. On the other hand, series 7 and 8 can make most of compression and communication methods. These graphic cards utilise PCI Express which improves CPU to GPU communications. This, along with the increase in fragment processors, means that variables can be sent to the fragment shader if necessary. Moreover, buffer compression can raise performance up to fifty percent depending on the series.
- It requires more computation to declare variables which may not be used than to load programs. The performance can be raised up to five times by using two short scripts, one with lighting, one without, rather than a single script which may be loading unnecessary variables.
- Using different libraries, e.g., ARB or OpenGL 2.0, has a very little effect on performance.
- Other geometrical shapes can contain the sphere, e.g., a regular hexagon may be used instead of a quad. Although this shape has two more vertices, it has a lesser area. Thus, less fragments are processed which usually assures a better performance. Nonetheless, as long as the *discard* instruction is applied performance does not vary. *Discard* is executed when the fragment is outside of the radius. As a consequence, discarded fragments are not taken into account as a potential bottleneck.
- Swapping CPU load to the GPU in GeForce 6 makes no improvements. However, on GeForce 7 and 8 cards there

© The Eurographics Association 2010.

is a doubling in performance. Specially, rotations and Billboarding perform faster when they are carried out in the vertex shader rather than in the OpenGL code. In general terms, any vector and matrix operations are favoured. Preparing specific codes for different graphic card generations can be useful due to the changing nature of shaders.

3.1. Results

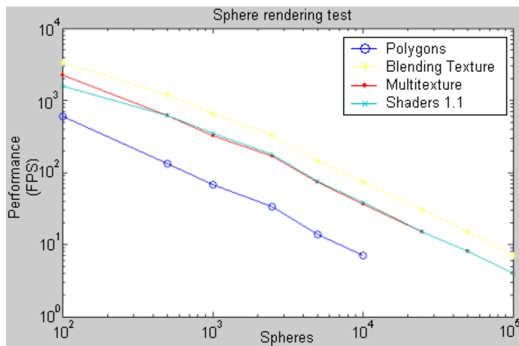


Figure 4: Logarithmic diagram (point data/fps) carried out with a GeForce 6800 Ultra. The test is done for the techniques: polygons (blue), blending texture (yellow), multitexture (red) and shaders 1.1 (clear blue).

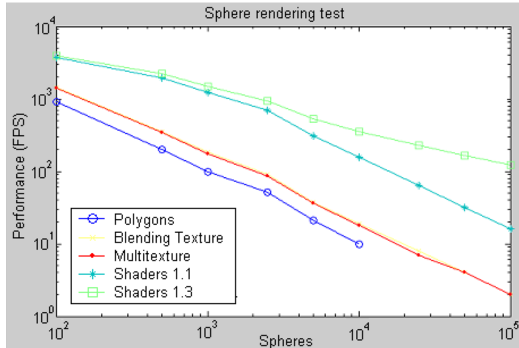


Figure 5: Logarithmic diagram (point data/fps) carried out with a GeForce 8800 GTS SLI. The test is done for the techniques: polygons (blue), blending texture (yellow), multitexture (red), shaders 1.1 (clear blue) and shaders 1.3 (green).

A performance test, measured in frames per second, is done with two different graphic cards: GeForce 6800 Ultra and GeForce 8800 GTS SLI. Four techniques, including shaders, are tested with datasets which are generated randomly and are formed from 100 to 100000 spheres. Results are shown in the figures 4 and 5. The polygonal mode has the worst result in performance. It involves a larger load on the CPU than the other techniques. When datasets are above 10000 spheres, it is not even possible to execute them. Apart from polygons, shaders have been compared with two texture approximations. Blending texture mode uses two modulating images, one for the circle and the other for the lighting. Colour is provided by the quad onto which the textures are applied. Multi-texture mode is focused on directly representing the colour on the textures so this technique needs an image for each desired colour. Both texture modes compete with shaders in GeForce 6 where the GPU does not have the latest technology. The visualization with textures is better when they are not close to the camera and pixilation comes apparent. Since textures are reloaded frequently, a resolution of 256 px^2 is chosen to avoid stuttering. The Blending texture method shows good performance and suits real time purposes. Using textures is an excellent option for GeForce 6 and older hardware as their capabilities are suited very well this task. There is a visual disadvantage since lighting is the same for all the spheres. The Multi-texture mode not only has a significant fall in performance, but also has a high reload time when the camera rotates. It is not a good option for dynamic environments. If the method were to apply a texture for each point data, the former effects would be more profound. A large use of textures should be avoided.

The shaders method proves to be a good way for current and future graphics hardware. Shading language for 1.3 shows significant improvements compared with the former 1.1 version. The latest GLSL version used allows the GeForce 8 to reach one million spheres with more than 20 fps. Shaders also show good results with GeForce 8, and newer graphic cards could manage huge datasets with millions of data points.

4. Rendering of Spatially Oriented Glyphs

Unfortunately, the sphere cannot be used to represent directional data. This section examines how to use shaders to render vector data using cones, which are conveniently available as an OpenGL primitive. A cone is suitable because it is able to show the same attributes as the sphere and it adds direction. A cone is relatively easy to calculate due to its rotational symmetry.

The shader method draws a cone on a flat primitive (a quad) and applies lighting to simulate the geometry. In the vertex shader the camera rotation is applied to the point's direction vector. This result is given in three dimensions but it must be transformed into two (Cartesian to Polar coordinates) to allow the cones to be drawn onto a plane using two angles, theta and phi (respectively 'x' and 'y' axes). In the fragment shader the creation of the cone only depends on theta since phi is used to turn the vertices. Billboarding is applied before any rotation as before.

Inside the fragment shader it is necessary to know which fragments from the glyph are kept and which are discarded. The orientation of the cone is determined from the theta angle, which has been passed from the vertex shader. A cone

© The Eurographics Association 2010.

can be thought of as a geometrical shape composed of an ellipse and two tangents, which join in the vertex. Both parts are calculated independently. The base of the cone is defined through the ellipse canonical equation where the theta value is shown as the variable 'd' and determines its angle. If the result is less than the radius, the fragment belongs to the base. To obtain the tangent point, equation 2 is used where 'h' is the 'y' coordinate of the cone vertex. The 'x' coordinate is found using the canonical equation again (Figure 6).

$$\frac{x^2}{1^2} + \frac{y^2}{d^2} = 1 \quad y_t = \frac{d^2}{h} \tag{2}$$

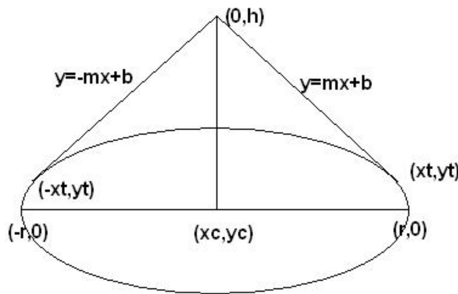


Figure 6: This is a drawing representing the cone we aim to create. The radius, the height and the quad centre are known. On the other hand, it is necessary to achieve the equation of the lines to know what fragments, which belong to the cone's side, are inside. A tangent needs two points: one is given, but the tangent point that touches the ellipse has to be calculated.

The tangents are found once the points are calculated, so it is then possible to determine if the fragment should be discarded. Given the 'y' component of the fragment's position, the component 'y' belonging to the tangent is the same. The line equation is useful to find the 'x' component of the tangent point. The fragment point is inside the lateral surface of the cone if it obeys the following rules. The 'y' fragment must be above the tangent point and beneath the vertex. The 'x' fragment must be between the negative and positive 'x' point coordinate belonging the tangent line. Conditions are summarized in the equation 3. The final result is shown in figure 7 on the left.

$$y_l < y_f < h \quad -x_l < x_f < x_l \tag{3}$$

Lighting is applied with different equations depending on if the fragment is located on the base or the conic surface. This task can be solved with the former equations. Once known, a new normal is set to define lighting. Rotations have been split between the vertex and fragment shaders. This alters the usual OpenGL way. Thus, the solution is not guaranteed to be invariant respecting the fixed pipeline and lighting is an approximation due to the difficulty of determining the normals of a cone, which is drawn onto a plane. The result can be observed in figure 7 on the right. Quality should be enough for visual purposes (figure 8).

In this rendering, occlusion has not been implemented, which can also be added to future work. Nonetheless, this problem may be solved following similar steps to the former occasions. An ideal case may be calculated. If fragments that belong to the base and surface are achieved separately, the problem is simplified. After that, it may be possible to calculate a new depth using the former equations presented in this paper and several trigonometric operations in order to simulate the rotations.

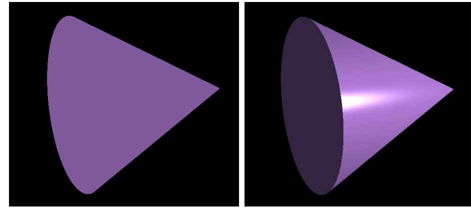


Figure 7: A cone is the final outcome of the spatially oriented glyph. On the left the cone without lighting and on the right the glyph is lit. The images represent the same cone but the first one does not allow the direction to be determined. Therefore, lighting gives precious information.

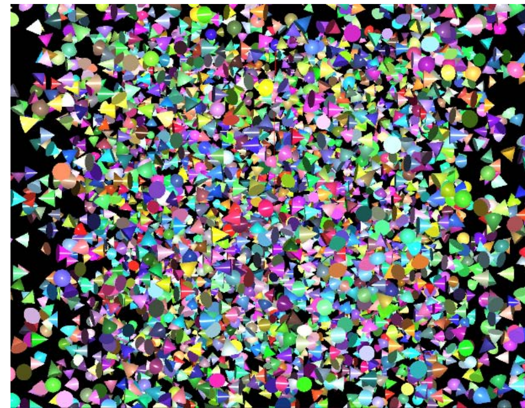


Figure 8: Rendering thousands of random cones thanks to the GPU. Despite the disorder, lighting helps to distinguish where the cones are pointing.

4.1. Results

The performance test is similar to the previous one. It is measured in frames per second and applied with three different graphic cards: GeForce 6800 Ultra, GeForce 7900 and GeForce 8800 GTS SLI. Two techniques, shaders and polygons, are tested with datasets which are generated randomly and are formed from 100 to 100000 cones. The figure 9 shows the result of the performance test. The polygonal mode offers much better results than sphere rendering

© The Eurographics Association 2010.

since cones are easier to build with vertices. It is surprising that polygons have better performance in GeForce 6800 than shaders. Nevertheless, the mode tends to break down if datasets are above 25000 cones. The other graphic cards allow shaders to obtain higher performances and to out perform the polygonal method. Unlike sphere rendering, the 1.3 version of GLSL does not demonstrate a difference in the frame rate. Therefore, it is not a fact that the 1.3 version always improves performance compared to older versions. Since operations are similar between sphere and cone rendering, it is believed that the bottleneck is generated by the uniform variables, which are sent by the CPU. Using vertex buffer may improve the performance but then, 1.4 version is required. In general, exploiting shader performance requires a specific study about the program type and the hardware where it is executed. Most operations are executed in the vertex and fragment shaders unloading work from the CPU, which becomes a bottleneck as GPU power grows.

It is remarkable that there is a clear division between GeForce 6 and the newer graphic cards used. GeForce 6 works very well with OpenGL lists as is seen in sphere rendering but shaders are not very powerful. Series 7 and 8 focus their efforts on Vertex Buffer Objects and other new ways to improve performance. However, they give poor results with older techniques (like OpenGL lists). Moreover, these cards are able to take advantage of better GPU capabilities. Shaders present a good perspective because for each series, its performance is increased at least up to sixty per cent.

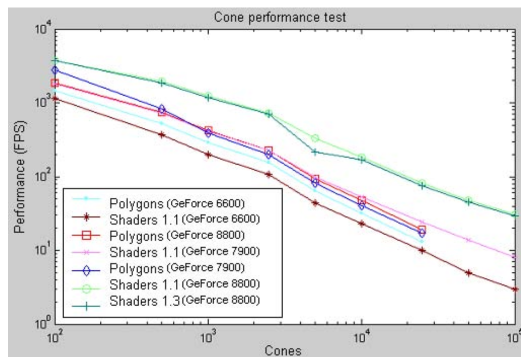


Figure 9: A cone performance test which is performed on several GPUs (6800, 7900 and 8800 GeForce). The 'x' axis is the dataset sizes and the 'y' axis is the frames per second. The test is done for the techniques: polygons (clear blue, red and blue), shaders 1.1 (brown, rose, pale green) and shaders 1.3 (green).

5. Conclusions

New GPUs are developed almost every year, e.g., Geforce 9 and 200 in 2008 and GeForce 300 in 2009. These graphic cards contain new hardware features that require new versions of shading languages to be designed in order to exploit their capabilities. For these reasons most work based on graphic card performance comparisons soon obsolete. Thus, this paper focuses on the techniques. It is expected that newer research can be derived from this paper as the principles remain intact. Performance comparisons are only a performance measure and, if the shown growth among series continues, graphic cards can be replaced with new ones to obtain better performance results. If necessary, shaders scripts can be easily updated since they are loaded by applications at runtime. Therefore, larger datasets can be examined with the subsequent card generations. Although the hardware tested can quickly become out-of-date, this work provides a better understanding of programming shaders and it serves as a foundation for new research as presented in *future work*.

GLSL is meant to guarantee a cross-platform compatibility for any graphic card which support this shading language. This should be true, but in reality it is not. The reason is that *the need of an implementation is greater than the independent hardware matters* [KBR04]. Furthermore, each vendor includes their own GLSL compiler and driver and they are allowed to create optimized code and to include their own functions. As a consequence, performance results are highly variable among graphic card generations, the versions of the drivers and the version of the GLSL specification supported.

Shader programs (sphere and cone rendering) have serious problems with ATI configuration (HD 4850 graphic card) which is not able to carry out either 1.1 or 1.3 GLSL versions. These problems obligate the developer to make specific shader programs for ATI cards. The 1.3 version does not offer greater compatibility with a program built for NVIDIA cards, which is more worrying. A standard should be set which assures that a code can be compiled correctly in any card if it supports the version (similar to XML verifications). Support for GLSL often lags behind the vendors proprietary shader languages.

Newer GLSL versions are focusing on a greater use of uniform variables in a similar way as CG does. Almost all built-in variables and matrix functions are effectively removed in GLSL 1.4 version and OpenGL 3 (they are still available through an extension). The main purpose is that these operations can be implemented in the shaders speeding up the graphic applications.

At the same time, this philosophy allows programmers to develop work further and quicker as new features and versions appear. GeForce 8 shows excellent results for all the methods presented and suits real time purposes. These results prove that performance can greatly benefit from shaders if exploited properly. The visual output is not an aim; nevertheless, programming pixels gives a better chance to improve visualization by means of a more accurate resolution. Buffer Objects probably become the best way to swap variables be-

© The Eurographics Association 2010.

tween the CPU and GPU since performance is actively increasing. Version 1.4 at GLSL allows them to be used for any variable; so far they have been restricted to vertex variables.

In conclusion, shader technology needs to improve its standards in order to be a powerful tool independent of hardware. On the other hand, it is shown that shaders offer multiple choices to solve problems and performances can be great as the code is not fixed, but entirely designed by the programmer.

6. Future work

The work contained in this paper has raised a number of questions and there is a lot of scope for further research and development.

Shaders are used to develop spheres and cones in a cheap way, but they are not the only useful glyphs. Future work on GPU glyphs could focus on developing more efficient algorithms. Although a hundred thousand point data have been represented for this research many scientists need millions of them to analyse their samples. Therefore, it is interesting to investigate the possibilities that OpenGL 3 and GLSL 1.5 (and later specifications) can offer to geometry rendering and new hardware architectures. In this work, the sphere rendering using version 1.3, has been a great step forward in performance.

GLSL 1.5 version can not only involve greater efficiency, but it also allows the use of geometry shader that means an extra functionality which raises the work research possibilities. This shader can add or remove vertices which can be useful, for example, in a method where point lines are used (one vertex each point). The geometry shader could add (curves) or eliminate (straight lines) vertices depending on the level of detail required.

Another problem to deal with is the depth buffer. It is not possible to exactly replicate the computations, and so if the depth is to be changed, it is necessary to make a new depth function. Depth is not frequently changed in shader programs and there are many proposals on how to modify it, but there is a lack of serious research in this area. Thus, greater efforts are needed in order to clarify this unclear element of GLSL. This becomes increasingly important if these rendering methods are combined with others, e.g., geometry rendered by the fixed pipeline. Once a better understanding is achieved; cone occlusion can be added to future work using shaders to solve fragment depth.

References

[Bli77] BLINN J. F.: Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics 11*, 2 (Sept. 1977), C219–C231. 2

[DHH05] DOKKEN T., HAGEN T. R., HJELMERVIK J. M.: The gpu as a high performance computational resource. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics* (New York, NY, USA, 2005), ACM, pp. 21–26. 2

[GP07] GRIBBLE C. P., PARKER S. G.: Interactive particle visualization with advanced shading models using lazy evaluation. *Eurographics Symposium on Parallel Graphics and Visualization* (May 2007). 2

[Hil06] HILGART M.: Step-through debugging of glsl shaders. *School of Computer Science, DePaul University, Chicago, USA* (2006). 2

[JZC*08] JOSELLI M., ZAMITH M., CLUA E., MONTENEGRO A., CONCI A., LEAL-TOLEDO R., VALENTE L., FEIJÓ B. O. M. D., POZZER, C.: Automatic dynamic task distribution between cpu and gpu for real-time systems. In *CSE '08: Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 48–55. 2

[KBR04] KESSENICH J., BALDWIN D., ROST R.: *The OpenGL Shading Language*. 3Dlabs, Apr. 2004. GLSLangSpec.Full.1.10.59.pdf. 6

[KPH97] KROGH M., PAINTER J., HANSEN C.: Parallel sphere rendering. *Parallel Comput. 23*, 7 (1997), 961–974. 2

[LMC05] LIANG K., MONGER P., COUCHMAN H.: Interactive parallel visualization of large particle datasets. *Parallel Comput. 31*, 2 (2005), 243–260. 2

[Pau07] PAUL B.: Characterizing scientific application performance on gpus. *The university of Wisconsin Madison* (2007). 2

[Ros04] ROST R. J.: *OpenGL(R) Shading Language (Orange Book)*. Addison-Wesley Professional, Feb. 2004. 3

[RSN06] RANTA S. M., SINGH J. M., NARAYANAN P. J.: Gpu objects. *Indian Conference on Computer Vision Graphics and Image Processing* (2006). 2

[Tay04] TAYLOR R. M.: Directly rendering non-polygonal objects on graphics hardware using vertex and fragment programs. *UNC Technical Report TR04-023* (2004). 2

[TL04] TOLEDO R., LEVY B.: Extending the graphic pipeline with new gpu-accelerated primitives. *24th International Gocad Meeting* (2004). 2

[Yua08] YUAN F.: An iterative programmable graphics process unit based on ray casting approach for virtual endoscopy system. *Optica Applicata 2008 38*, 3 (2008), 519–530. 3

[ZTH03] ZEMCIK P., TISNOVSKY P., HEROUT A.: Particle rendering pipeline. *Spring Conference on Computer Graphics 2003 Proceedings* (2003). 2

© The Eurographics Association 2010.