Optimized Data Transfer for Time-dependent, GPU-based Glyphs

S. Grottel, G. Reina, and T. Ertl

Institute for Visualization and Interactive Systems, Universität Stuttgart

Abstract

Particle-based simulations are a popular tool for researchers in various sciences. In combination with the availability of ever larger COTS clusters and the consequently increasing number of simulated particles the resulting datasets pose a challenge for real-time visualization. Additionally the semantic density of the particles exceeds the possibilities of basic glyphs, like splats or spheres and results in dataset sizes larger by at least an order of magnitude. Interactive visualization on common workstations requires a careful optimization of the data management, especially of the transfer between CPU and GPU. We propose a flexible benchmarking tool along with a series of tests to allow the evaluation of the performance of different CPU/GPU combinations in relation to a particular implementation. We evaluate different uploading strategies and rendering methods for point-based compound glyphs suitable for representing the aforementioned datasets. CPU and GPU-based approaches are compared with respect to their rendering and storage efficiency to point out the optimal solution when dealing with time-dependent datasets. The results of our research are of general interest since they can be transferred to other applications where CPU-GPU bandwidth and a high number of graphical primitives per dataset pose a problem. The employed tool set for streamlining the measurement process is made publicly available.

Index Terms: I.3.6 [Computer Graphics]: Methodology and Techniques I.3.6 [Computer Graphics]: Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1 Introduction

The performance-optimized rendering of points or splats has been investigated for some time now. The applications of these techniques can be roughly divided into two main topics. The first relates to point set surface rendering, where the geometry of a single point is usually a very simple surface (circular or elliptic splats [3]). The rendering quality of such splats has been steadily improved over the years to yield high surface quality (see [2] and [14]). The other main area employs different kinds of glyphs with higher semantic density. This includes rendering of such glyphs on the GPU using point billboards for particle datasets (e.g. see figure 1, [21], or [11]) and even more complex glyphs for information visualization purposes [5].

To obtain interactive performance, much time has been dedicated to develop efficient storage, like in-core representations and hierarchical data structures (for example in [18] or [19], among many others). Linear memory layouts have been appreciated not only for their benefits for rendering performance, but also for the advantages when rendering out-of-core-data [13], which is why we employ this approach in our visualization system as well. However, in all the related work we know of, the authors often make simplifying assumptions regarding first-level data storage and transfer to the GPU. One assumption states that the visualized data is stored in the GPU memory and read from static vertex buffer objects, which of course ensures optimal performance. However, this is not possible when handling time-dependent data. The other assumption regards the best-performing upload techniques that need to be employed to cope with such dynamic data, which at first glance also seem an obvious choice. A software capable of handling such data has been shown in [8], however, there are many factors that can potentially influence the resulting performance, starting from the hardware choice and system architecture, over driver issues to implementation issues (in the application as well as in drivers and hardware). To our knowledge, these well-informed choices are very rarely supported by hard numbers and direct comparison of the many alternatives, so we want to fill this gap. Examples for performance analyses exist for the uploading and downloading of texture data as well as for shader arithmetic [7]. A more generic benchmarking tool exists [4], but it does not cover the aspects we are interested in, so we are trying to provide more detailed data on the available vertex upload mechanisms.

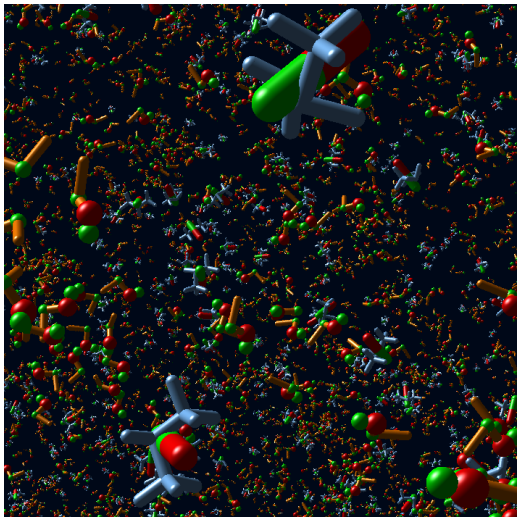

Figure 1: A typical real-world particle dataset from the field of molecular dynamics, containing a mixture of ethanol and heptafluoropropane, 1,000,000 molecules altogether, represented by GPU-based compound glyphs.

The main contributions of this paper are the provision of a benchmarking tool as well as the subsequent investigation of the performance impact of different vertex upload strategies and silhouette calculations for raycast GPU glyphs. We put these figures into context by applying the findings to a concrete visualization example from molecular dynamics simulations from the area of thermodynamics. Widespread programs for molecular visualization exist, for example VMD or Pymol, however their performance is not satisfactory when working with current datasets, which consist of several hundreds of thousands of molecules. One flaw is the lack of proper out-of-core rendering support for time-dependent datasets, and the other is insufficient optimization and visual quality, as the capabilities of current GPUs are not significantly harnessed. Approaches

have been published which remedy the latter issue, such as employing non-perspectively correct texture-based primitives [1]. Perspective correctness as well as higher visual quality through ambient occlusion calculations has been added in [20], while higher performance is obtained by aggregating atoms and generating their bounding geometry from a single uploaded point [12], which is similar to the approach we chose when utilizing the geometry shader (see below).

We deduce different strategies for generating more complex compound glyphs suited to represent current molecular models from our analysis and evaluate the resulting performance with real-world datasets. This approach can also be transferred to any other field that makes use of particle-based simulations, such as computational physics, biochemistry, etc.

The remainder of this work is structured as follows: Section 2 describes different data upload strategies and their evaluation on different machines with our tool for the automation of the benchmarking process. In section 3 we present the approaches for rendering compound glyphs along with performance figures. Section 4 concludes this work with a discussion of the presented results.

2 Data Upload

The datasets currently available to us require interactive rendering and navigation of up to several million atomic representations, in our case still mostly simple geometric primitives. This is relatively easy to achieve for static datasets since current graphics cards offer memory in about the same order of magnitude as workstation PCs and considerable processing power for rendering the primitives we use. For time-based datasets one of the most important aspects to optimize is the data transfer between CPU and GPU.

2.1 Setup

The available optimizations unfortunately might depend on the employed platform and graphics library. We have therefore cross-checked a random subset of our results from our OpenGL-based visualization tool with a rudimentary DirectX 9 implementation and found the performance nearly equal (within a margin of 3%), so no further investigation was conducted for the time being. All considerations in the following will thus only refer to using OpenGL as the graphics library. Some of the tests have been conducted under Linux as well, but they are also on par with the performance values reported for Windows.

All of our tests have been performed with production software that is used also by our project partners in natural and engineering science for visualizing the output of their simulation runs. Additionally, we used synthetic datasets of varying sizes that are at least on par with or slightly larger than the currently available average simulation.

Since the available PCs nowadays also differ slightly in some architectural choices (memory controller location, etc.) as well as in processing power, we chose a small number of systems and GPUs and tested the viable combinations of those. Two fast machines (Intel Core 2 Duo 6600, AMD Phenom 9600) and a slower one (AMD Athlon 64 X2 4400+) were used, all running Windows XP x64. We also included an AGP-based system (Intel P4 2.4Ghz) running 32-bit XP; for comparability all machines used the same 32bit binaries. The PCIe graphics cards we rotated through the machines were an Nvidia GeForce 6800 GS, a 7900 GT, an 8600 GT and a 8800 GTX. Additionally we tested a GTX280 in the Core2 machine (The other computers did not have a adequate power supply). All cards except the GTX280 use driver version 169.21. Since the GTX280 is not supported by the old driver, we had to use version 177.51 (which in turn does not support the older cards).

No AMD cards were used since their OpenGL support is currently insufficient for any of our algorithms. Immediate mode and static VBOs (Vertex Buffer Objects) were always included in all measurements as reference for the highest possible load on the CPU and the highest possible framerate achievable with the shaders used, even though the latter cannot be employed for dynamic datasets. All diagrams include error bars to highlight rendering modes with highly fluctuating frame rates.

We used the default driver settings, but programmatically turned *vsync* off for all tests. This makes measurements less error-prone but seems disadvantageous for the Intel-specific display driver path: by default the drivers are in 'multiple display performance' mode, which causes at least 10% performance loss with respect to 'single display performance mode' (only on our Intel machine).

When working with time-dependent data sets the visualization process can be understood as a pipeline with the following major steps: loading and preparing a time step in main memory, uploading the data from the main memory onto the GPU, and rendering the final image. The first point, loading and preparing the data, heavily depends on the application at hand. Data might be loaded from secondary storage, received over a network, or even computed on the local machine. Technologies like RAID configurations, fast networks like infiniband, and multicore CPUs allow many optimizations of this aspect. Preprocessing can also be applied to reduce the processing workload while loading the data. We therefore decided to not discuss this first point in our work at hand.

It is difficult to handle and observe the second and the third stages of this simplified pipeline independently. One of our assumptions is that every frame renders a different time step and therefore needs different data. So keeping data in the GPU-side memory for several frames is not an option. As we will discuss later (Section 3), the layout and amount of the data to be uploaded differs depending on the applied rendering technique. Using a more sophisticated rendering approach often results in additional data to be uploaded. So we understand these two interdependent stages of the pipeline as a single optimization problem. Because current graphics drivers almost always use asynchronous data transfer, it is not meaningful to estimate individual times for these two stages. Instead we decided to test different combinations of uploading strategies and rendering methods which result in different processing loads for the GPU. We believe that we get more meaningful results this way.

However, by doing so we have to face two drawbacks: our performance tests are now close to *black box* tests, so we cannot get detailed information on which part of the rendering code forms the bottle neck. To overcome this we decided to write rather small and clean rendering codes (e. g. only about 3 to 5 OpenGL calls for uploading data with vertex arrays). The second drawback is that we cannot identify the sheer uploading time in milliseconds. Instead we show the overall rendering performance in *frames per second*. To exclude time required for data upload, we only use data which fits into the main memory and which is loaded before our tests.

To understand the resulting performance we need an upper bound: a maximum frame rate which could be reached by the rendering stage alone. For defining this value, we included rendering using *static vertex buffer objects* in all of our tests. These reference values are calculated by uploading the data once into the static VBO before the tests starts and disregards upload completely. Although static VBOs are not viable when visualizing time-dependent data, this gives us a reference value of the rendering load and allows us to put the upload part into context.

This approach results in a huge amount of different measurements to be performed, much more than one would want to handle manually (For the work at hand we performed 155 tests per hardware combination; 2170 test altogether). We therefore created a tool that automatically performs all these tests, collects and aggregates the results, and even performs some basic calculations (see figure 2). Since we decided to focus on the windows platform we chose the .NET framework as basis for this tool. By doing so the tests can be controlled by code, written in any high-level lan-

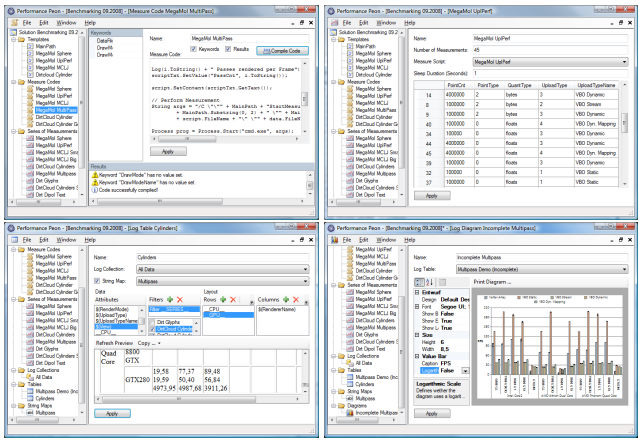

Figure 2: User interface of our performance measuring tool. From upper left to lower right: editing of .NET code controlling the measurement process, instantiation of measuring code using parameter tables, managing results and creation of performance tables and diagrams from performance results.

Name	OpenGL Calls Description	Main Parameter

Immediate	`glBegin`
	`glVertex*`
	manual upload of individual data points
		`GL_POINTS`

Vertex Array	`glVertexPointer`
	`glDrawArrays`
	direct array data upload
		`GL_POINTS`

VBO static	`glBufferData`
	`glDrawArrays`
	reference rendering with only *one* upload (not time-dependent)
		`GL_STATIC_DRAW`
		`GL_POINTS`

VBO stream	`glBufferData` buffer object upload meant for data "modified once and used at most a few times"	`GL_STREAM_DRAW`
VBO dynamic	`glBufferData` buffer object upload meant for data "modified repeatedly and used many times"	`GL_DYNAMIC_DRAW`
VBO dynmapping	`glMapBuffer` buffer object memory mapping when CPU memory layout is not optimal	`GL_WRITE_ONLY`
VBO dynmC	Same as *VBO dynmapping* but includes a color array	
VBO pre-int	Same as *VBO dynmC* but CPU-side memory layout is already optimal, by using interleaved attributes per point	

Table 1: Explanation of the different uploading mechanisms used in the first tests.

guage supported by the .NET framework, which can then be instantiated using parameters from a table with all tests to be performed. This includes the creation of temporary files and data for input and output, the spawning of processes and the capture of their output streams, which can be conveniently processed exploiting the whole .NET functionality. For additional sanity checks, a screenshot of the measured application can be taken and stored with the results, such that the rendinger output of suspicious measurements can be double-checked.

The performance results are stored in text files, easy to parse, and can be imported into a local database of our tool. Latter allows custom queries, like filtering for specific configurations or parameters, to generate tables and diagrams with complex layouts like multi-level categories. Tables and diagrams can be exported in different formats (HTML, CSV, LaTeX, PDF) for ease of use. The generic approach of our tool allows for using it with any kind of application, like 3d model rendering or volume rendering. The application to be tested just needs to be configurable through an input file or its command line and must be able to output its performance to a file or to its output stream. This tool is publicly available from our web site[1] and will probably be further extended.

Only excerpts of all the benchmarks performed can be presented here, but the full results, including all diagrams in full page size, are also available on our web site.

2.2 Upload Strategy

The upload mechanisms available in OpenGL range from the CPU-intensive *immediate mode* (an exclusive to OpenGL) over *vertex arrays* to different kinds of *vertex buffer objects*. Details and the names we use to reference these mechanisms can be found in table 1. Many publications proposing radically different visualization approaches compare one specific mode to another or just plainly advocate the use of one over all others – obviously any optimized method works better than the immediate mode. However there are many differences in data size and organization as well as side-effects. So we wanted to take a much closer look at the whole range of available options and compare them, keeping the particular problem in mind that we need to transport a high number of objects with a small number of parameters to the graphics card to generate glyphs directly on the GPU.

[1] http://www.vis.uni-stuttgart.de/eng/research/fields/perf/

In general it can be said that with the available diagrams it is quite easy to distinguish effects of the GPU choice (similarities among the three machine-dependent measurement groups) from platform problems, like lacking CPU power or bandwidth (similarities inside one group). Furthermore, high fluctuation can be observed across all diagrams to be caused by overstrained GPUs (mainly too high a shader load in our case).

The first batch of tests was aimed at finding the optimal upload strategy among vertex arrays and the different options for vertex buffer objects. These benchmarks are especially important when looking at the specification of OpenGL 3.0 [15], which deprecates vertex array support. The tests are run with varying rasterization load, either single-pixel points, raycast spheres on a regular grid or overlapping raycast spheres (to cause z-replacing more frequently).

The first effect that can be observed is that vertex arrays are always at least twice as fast as any of the VBO modes when the fragment processing load is negligible. It can also be observed that GeForce 7 GPUs suffer from less overhead as for small batches of spheres the mid-range card performs even better than the current high-end card (see the 100k spheres diagrams in the full results). One possible explanation is that the shader units are statically assigned to either vertex or fragment processing, while the drivers seem to have to balance the use of processing units of the GeForce 8 to match the current load presented by the activated shaders. This overhead is not present when the fragment load is minimal, so pixel-sized points can be rendered minimally faster on the newer GPUs (see upper diagram in figure 3).

2.3 Quantization

We also tested how much performance can be gained when uploading quantized data. For example [10] suggests the use of byte quantization to increase performance by reducing the upload. However the performance gain was not set in relation to the loss of resolution. We therefore tested quantization with floats, shorts, and bytes in our framework. The results show very clearly that shorts perform as expected: they are nearly twice as fast as floats and thus directly benefit from the halved data size. Even with the reduced bandwidth requirements, vertex arrays are still at least 50% faster than the best VBO variant, and interestingly even faster than static VBOs on an 8600GT since the GPU memory seems to be more of a limit than the system bandwidth. A powerful host system is required to obtain optimal performance from an 8800GTX, since only the Core2

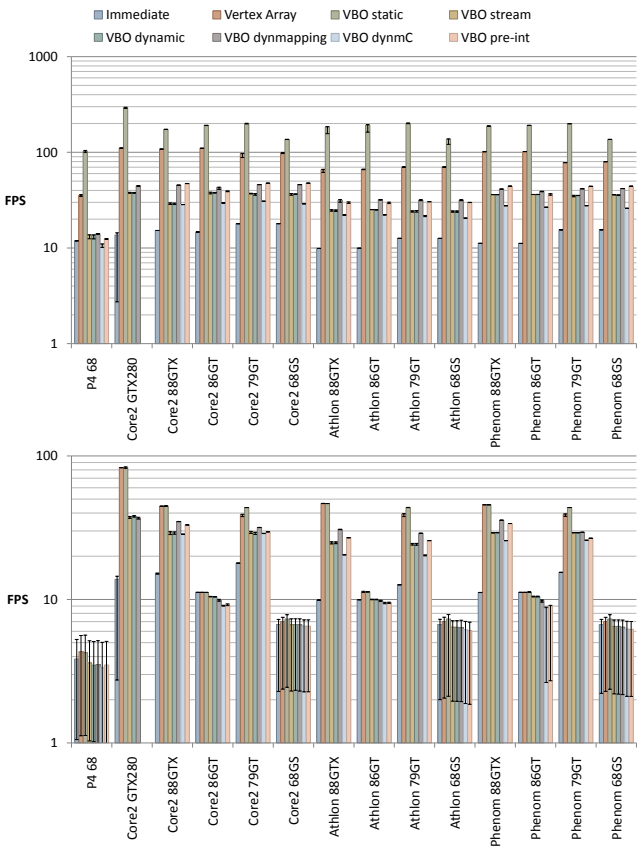

Figure 3: Upload performance for 1M 1-pixel points (top) and 1M touching spheres (bottom) on a regular 3D grid covering 80% of the viewport. This diagram uses logarithmic scale for better depiction, since the data values vary by orders of magnitude. Positions are defined using 3D float vectors. View port size is 512^2. Since our focus lies on the data upload, we accept the overdraw due to our large particle counts. See table 1 for a description of the shown methods. VBO dynamic mapping means that the buffer is locked and then `memcpy`'d into. The last two measurements compare dynamic mapping including one color per vertex. For the first a high number of copy operations is needed (and as such is slower as dynamic mapping only), while the second makes use of a pre-interleaved client-side array of positions and colors, which is consistently faster even than dynamic mapping only for points, but slower for spheres.

system can offer a significant performance increase when changing the 8600GT for an 8800GTX. The upload performance of AMD systems does not benefit particularly from this high-end card (see figure 4). The bandwidth limitation of the Core2 system can be seen when uploading floats (see the 'floats' diagrams in the full results linked previously). Switching from shorts to bytes, however, does not yield a significant performance increase, but rather a slight to quite marked (in the Phenom case) decrease that might be due to alignment problems – assuming the hardware is optimized for handling dwords. Using the generally less-advisable VBO upload, the performance gains are of at least 25%, however still not large enough to compensate the advantage of vertex arrays.

2.4 Billboard Geometry and Geometry Shader

As discussed earlier we understand data upload and rendering to be interdependent, since the way of upload and the layout of the data needs to be adapted to different rendering algorithms, while the possible choices of these algorithms depend on the available uploading

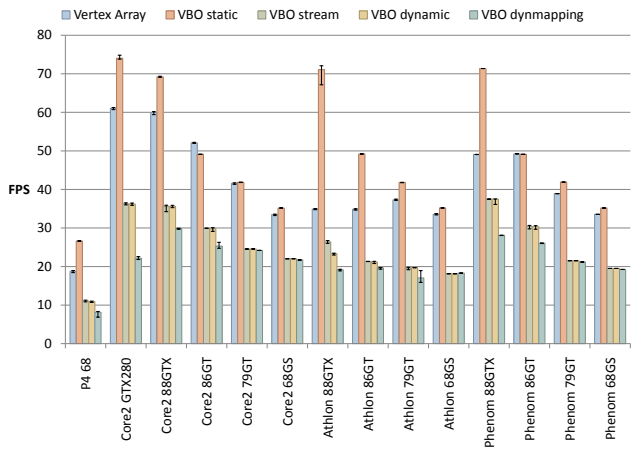

Figure 4: Upload performance for 4M `short`-quantized points. Only 1 fragment is rasterized per vertex.

mechanisms. So, another series of tests was targeted at finding out how much the raycast primitives would benefit from tightly-fitting bounding geometry to reduce the discarded fragments outside the glyph silhouette. This is an optimization problem where more complex silhouette approximations decrease fragment processing load, but increase vertex processing load and also might increase the data that needs to be transferred.

As example we chose a cylinder glyph with 2:1 aspect ratio. Three silhouette variants have been tested: a single point (thus a screen-space axis-aligned bounding square), an object-aligned quad uploaded as vertex array, and the corners positioned in the vertex shader, thus trading fragment load for bus load. Finally points were uploaded and expanded into the same quads by use of a geometry shader. It should be noted that the geometry shader has to output a much higher number of attributes per vertex than in comparable approaches (e.g. the billboard generation in [12]) thus putting a significant load on the GPU. We need a transformed camera and light position passed to the fragment shader as well as the primitive parameters for raycasting the glyph surface. Unfortunately, series 8 Nvidia GPUs are extremely sensitive to the total number of attributes emitted, degrading the resulting performance.

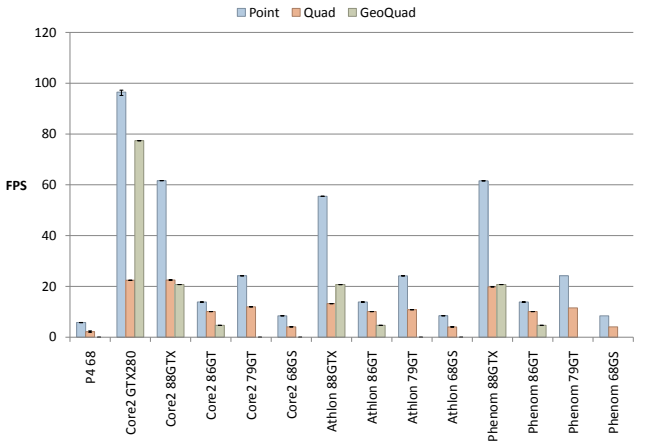

Figure 5: Rendering performance for 500K raycast 2:1 cylinders in a 512^2 viewport with varying bounding geometry. The geometry-shader constructed quad is only available for GeForce 8 cards and newer.

Because of the aspect ratio, a fragment processing overhead of 50% when using points as bounding square is fairly common. Since the glyph is relatively cheap to raycast, this test focuses on finding the additional cost of integrating an improved bounding geometry calculation into the rendering engine, as the calculation itself is also simple. Of course with very expensive glyphs a better bounding geometry becomes more beneficial, but also more expensive to calculate. The results can be seen in figure 5. It is obvious that the use of a geometry shader is extremely costly without a high-end card.

For series 8 Nvidia GPUs the brute-force approach with quads offers a comparable performance on the fast systems, as the available bandwidth allows it. Only the old Athlon system benefits from the employment of the geometry shader as it takes enough load off the system. The current GeForce GTX280 does not lose that much performance any more, however the single point billboard still performs significantly better. From these experiments we draw the conclusion that the current fragment processing is so carefully optimized that a significant overhead and the ensuing high number of fragment kills are not an issue and thus 'suboptimal' point primitives are still a reasonable approach. On the flip side it is obvious that Nvidia's implementation of the geometry shader is expensive to use and does not provide significant benefits when used to reduce bus load only (even as significantly as the 75% as in our tests). The current generation of GPUs does improve the situation much, but not enough. Of course a very interesting alternative would be a variant of *EXT_draw_instanced* where the primitives would be multiplicated in an inner loop (see [6]) and could be reinterpreted as another primitive (points as quads in this case), but such an extension does not (yet?) exist.

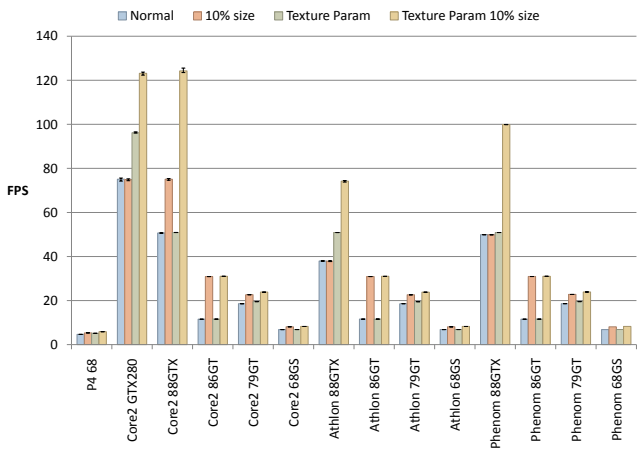

Figure 6: Rendering performance for 500K raycast dipoles (complex glyph consisting of two spheres and one cylinder, raycast in one shader; see [17]) in a 512^2 viewport. The 'normal' benchmarks employ directly uploaded parameters per primitive, while the others get them from a texture. Also the glyphs are scaled to only 10% of their size to show texture access cost with less dependency on the high fragment shader load.

The last tests were used to investigate the effect of textures to specify per-primitive-type parameters (radii, distances, colors etc.) and access them in the vertex shader instead of uploading all parameters with every glyph. The reduced bandwidth should have a significant impact on performance, however tests that were conducted when the vertex shader could first access textures (GeForce 5 onwards) were not convincing since it incurred a significant cost when compared to texture accesses in the fragment stage. These results can still be reproduced with the older cards (figure 6), but with the newer generations of graphics cards, things have changed. Parameters stored in textures never cause performance loss. The current generation actually can benefit from local parameters in every situation, while the 8800GTX only benefits in light shader load situations or on machines with limited CPU resources (especially on the Athlon). In the next section we will apply our findings to a real-world problem and discuss the data upload optimizations.

3 Compound Glyphs

In many scientific areas, such as mechanics, thermodynamics, materials, and biotechnology, classical simulation methods based on continuous data structures still fail to produce satisfactory results or even fail to correctly model the situation to be studied. Molecular dynamics simulations are rapidly becoming a wide-spread alternative, and due to the cost-effectiveness of ever larger COTS clusters, their processing power, and their availability as client-site infrastructure, the necessary computations can be performed without heavy effort and within acceptable timeframes.

While simple molecular models can use a single mass center, more complex ones may consist of multiple mass centers and multiple and optionally directed charges. When analyzing such datasets visualization allows the scientists to observe the interactions of individual molecules and helps to understand the simulation itself. Not only are the positions and energy levels of interest, but also the orientations and the distances between their mass centers and charged elements. A crucial problem is introduced by the large amounts of particles which need to be rendered interactively. This problem gets even worse when the complexity of the molecular model is increased. Accordingly complex glyphs must thus be used to represent multiple mass centers and charges. While using GPU-based raycasting or texture-based approaches for rendering geometric primitives is common practice, these techniques cannot be easily applied to complex or compound glyphs, that is glyphs consisting of multiple graphical primitives like spheres or cylinders. The straightforward approach of rendering these graphical primitives, which is also addressed in this paper, easily increases the number of objects to be rendered by at least one order of magnitude.

3.1 Modeling

To create a meaningful visualization the molecules' visual representation must match the elements of the molecular model employed in the simulation. However the visual model for the molecules must also be chosen reasonably, as we cannot put an arbitrarily high load on the GPU. Usually the mass centers of a molecule are shown as spheres. The van-der-Waals radius is often displayed, since it is a good representation of the influence range of the mass element. This, however, results in rather dense representations occluding potentially interesting features like localized, directed charges. We therefore propose a more sparse representation, showing the structure of the molecule and emphasizing the charges, which is based on the classical (ball-and-)stick metaphor, as known from the fields of chemistry and biology.

These glyphs are constructed out of several spheres and cylinders with appropriate parameters. The principal structure of the molecules is conveyed by a stick representation using spheres and cylinders with the same radius. To easier distinguish molecule types, all uncharged elements in one molecule share the same color. Additional elements represent the charges: spheres indicate point charges, and two spheres with two cylinders show a directed charge using the metaphor of a bar magnet. The radii of these elements are chosen proportionally to the strength of the charges they represent, and the type is shown by the color (green for positive charges, red for negative ones). The ethanol molecule (figure 7, left) consists of four spheres and two cylinders (a third cylinder, located between the two upper spheres, is removed in an optimization step because it is completely occluded). The R227ea molecule (heptafluoropropane; figure 7, right) is constructed from twelve spheres and eleven cylinders.

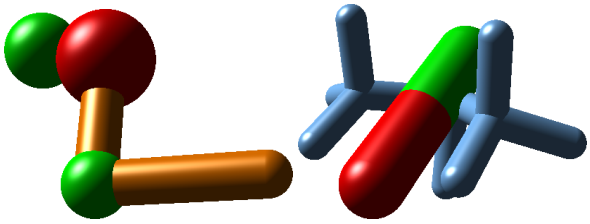

Figure 7: Two complex molecules modeled with spheres and cylinders. Left: an ethanol molecule with the orange stick representing the carbon backbone and three spheres showing point charges; Right: a heptafluoropropane molecule with a blue stick representation of the carbon and flourine atoms and a bar magnet showing a directed quadrupolar charge.

Of course, other representations for these molecules could also be applied. Our framework is flexible in the number of elements a glyph is composed of and in the way these elements are placed and parameterized. Other graphical primitives, for example cones or ellipsoids, could be used as well. The primitives are placed and oriented in a particle-centered coordinate system. The center position and orientation as the only parameters of each particle are then used to construct its final visual representation.

3.2 Rendering

All graphical primitives are rendered using GPU-based raycasting of implicit surfaces as presented in [9] and [11]. [17] showed that it is possible to construct even more complex glyphs in a single raycasting shader, however, this approach is very limited and cannot be generalized to arbitrary compound glyphs without serious performance drawbacks. Since we work on time-dependent data and use interpolation of positions and orientations to generate a smooth animation, the necessary calculations for each image must be limited to achieve interactive frame rates.

The naïve approach is to transfer all graphical primitives from their local glyph-centric coordinate systems into a common world coordinate system and then render them. This recalculation is performed on the CPU to keep the raycasting shaders as simple as possible to set a baseline reference for upcoming optimized approaches. The additional data can be directly sent to the graphics card by using *immediate mode* functions or it can be stored linearly in main memory for *vertex array* transfer. As the results section 3.3 shows, the rendering performance of this approach is quite unacceptable.

Therefore, we moved these calculations to the graphics card. The mechanisms of *instancing* seemed suitable. However, hardware support for OpenGL instancing requires *shader-model-4*-capable hardware. To be able to use this approach on older cards too, we used *vertex buffer objects* to emulate instancing (as is also suggested in [16]). The idea is to upload all particle data once per frame and re-use it once per graphical primitive needed for the specific molecule glyph. We change only a single uniform value as primitive index (replacing *gl_InstanceID*, see below). The parameters of a primitive (such as relative coordinates, radius, and color) are loaded from two parameter textures into the vertex shader. This shader will then recalculate the primitives' positions using the orientation quaternion and the world positions of the molecules. The results show that this approach performs very well with our molecule models consisting of 6 and 21 graphical primitives.

However, the question remains whether hardware-supported instancing or geometry shaders could do an even better job on current graphics cards. The latter approach uploads only one point per molecule and then emits multiple graphical primitives from the geometry shader unit. The parameters for these glyph elements are again retrieved from textures. However, this requires a fragment shader capable of rendering all graphical primitives. Although some elements have some aspects in common and a combined shader could be optimized, this is not true for the generic approach of combining arbitrary glyph elements. To keep the maximum flexibility, the shader must be some sort of concatenation of the primitive shaders enclosed by program flow control, which is still expensive in the fragment processing stage. When using a geometry shader to produce the individual elements, this flow control is also needed for the code partition of the geometry shader which performs the silhouette approximation. So using this approach with our particular glyph comes with the overhead of two large *ifs* (one per stage).

Name	Description
*	Entries from table 1; All these modes use simple shaders drawing one graphical primitive each.
Geo combo	Uses vertex array upload (one vertex per molecule) and one geometry shader for all graphical primitives.
Geo primitive	Uses one geometry shader per primitive type and uploads (one vertex per molecule) once per shader using the vertex array mechanism.
Geo VBO static	Works like *Geo primitive* but uses the described VBO upload with `GL_STATIC_DRAW`
Geo VBO stream	Works like *Geo primitive* but uses the described VBO upload with `GL_STREAM_DRAW`
Geo VBO dynamic	Works like *Geo primitive* but uses the described VBO upload with `GL_DYNAMIC_DRAW`
Instancing combo	Uses the extension `GL_EXT_draw_instanced` and `glDrawArraysInstancedEXT` for data upload, and uses one geometry shader for all graphical primitives, analogous to *Geo combo*
Instancing primitive	Works like *Instancing combo* but uses simple shaders (one for each graphical primitive) and uploads the data multiple times, analogous to *Geo primitive*.

Table 2: Explanation of the different uploading mechanisms in addition to the ones described in table 1.

To avoid this overhead, which is fatal as our results show (see figure 8), a separate geometry shader for each graphical primitive is employed. Since the glyph elements do not require alpha blending, we only need one shader switch per primitive type, and all the shaders get rid of the branching. However, this creates overhead again since all the molecule data needs to be uploaded multiple times per frame (one time for each shader, if at least one molecule contains all element types) when using vertex arrays. This again is a fitting application for vertex buffer objects: upload the data only once, but use it twice (in case of two element types). However, section 3.3 demonstrates the cost of employing a geometry shader is still too high to be compensated by optimized upload. This could change with future graphics card generations, if the penalty for outputting many attributes is reduced.

The second alternative on current graphics cards would be instancing [6]. This OpenGL extension allows to input the same data (a *vertex array* or *vertex buffer object*) several times into the OpenGL pipeline with just a single call. We use this mechanism to multiply the input molecules by the number of their primitive elements and employ the built-in instance index to look up the per-primitive parameters from the texture. Analogously to the two approaches using vertex arrays, instancing can use one complex shader capable of raycasting all graphical primitives using *if* clauses or the calls can be separated to use cheaper shaders. This approach results in performance values very similar to the results of the emulated instancing using VBOs. It therefore is currently unclear what the advantages of using this extension could be.

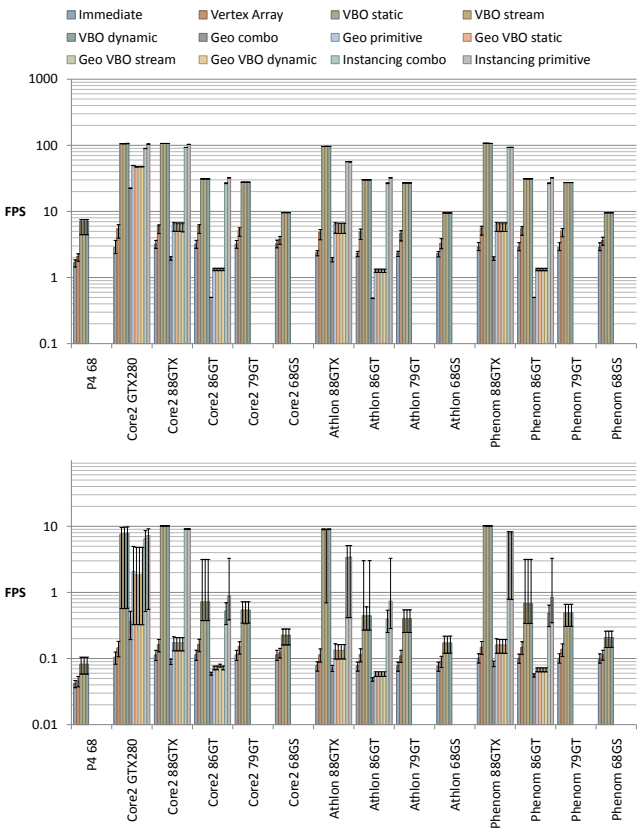

Figure 8: Performance for two compound glyph datasets (top: 100,000 Molecules; bottom: 1,000,000 Molecules). This diagram uses logarithmic scale for better depiction, since the data values vary by orders of magnitude. See table 2 for descriptions on the different uploading methods.

3.3 Rendering Performance

The different methods of creating compound glyphs for complex molecules described above were tested on the same machines used for the preliminary measurements in section 2. We used two datasets from molecular dynamics simulations employing the Lennard-Jones model. The first dataset contains 85,000 ethanol molecules and 15,000 heptafluoropropane molecules. The second dataset uses the same molecule mixture under the same thermodynamical conditions but uses ten times more molecules of each type. Using the same molecule representation as described in section 3 this results in a total number of 825,000 and 8,250,000 graphical primitives ($6 * 850,000 + 21 * 150,000$).

Table 3 and figure 8 show the performance values of both datasets with our approaches. On the high-end cards the usage of vertex buffer objects to emulate instancing results in the best performance, although the instancing extension is just slightly slower.

It is obvious that shader programs should be kept as simple as possible (branching should still mostly be avoided). On cards prior to the GTX280 even the *immediate mode* rendering is faster than using the combined geometry shader. The two methods using the primitive shaders are either much faster or at least not significantly slower than the ones with the complex shaders.

Another interesting fact is that the frame rates of the instancing-based rendering modes are quite constant for the 100K dataset but strongly varying for the 1M dataset (on all cards except for the GeForce 8800) – see the error bars in the lower diagram of figure 8. We interpret this as an indication that the overall GPU load is at its limit for the older cards. The current high-end card also exhibits extremely unstable frame rates with large data sets, which is probably due to the immature driver code.

4 Conclusion and Future Work

In this work we presented an automated, flexible tool for performance measurement series. We employed it to produce concrete values for all the different uploading mechanisms OpenGL offers for time-dependent point-based data. Based on that data we also demonstrated a way of representing molecules as compound glyphs showing backbone atoms as well as point and directed charges. To be able to render huge time-dependent datasets we also evaluated different strategies of constructing these compound glyphs for optimal performance for a wide range of workstation computers using current graphics cards. Our contribution is a detailed quantification of the performance factors that affect GPU-based glyph and splat rendering. We believe that the findings presented can be applied to a wide range of applications beyond than the ones presented here.

For the datasets we used vertex arrays still are the best uploading mechanism available for basic glyphs. We attribute this to the fact that it comes with the least overhead compared to dynamic or streaming vertex buffer objects. Another interpretation would be that vertex arrays allow the GPU to start rendering immediately after the data transfer begins, while VBOs need to be transferred entirely before rendering, resulting in a marked performance disadvantage when no re-use takes place. The deprecation of vertex arrays in OpenGL 3.0 is very disappointing, since there is no obvious successor. No vertex buffer type has clearly superior performance over the others: on the *Athlon GeForce 8800* system streaming vertex buffer objects are faster, but on the *Core2, GeForce 8600* system dynamic vertex buffer object are faster, for example. A part of the 'common knowledge' about the OpenGL is confirmed: mapping a dynamic VBO is advantageous if the data layout in memory is suboptimal, offering an ideal alternative to the costly immediate mode. When linearly organized data is available, the direct-upload VBOs have less overhead and result in better performance.

Our measurements showed that the highly situational performance of geometry shaders was much improved with the GeForce GTX280, offering a viable option for the construction of bounding geometry that more tightly fits a glyph than basic points.

For compound but rigid glyphs the best option overall is to use any of the instancing approaches. Additionally the parameters of the primitives should be placed in parameter textures, since this has at least potentially positive impact on the performance on all hardware combinations we tested.

We want to add further measurement series to extend our findings to ATI graphics cards and especially NVIDIA Quadro cards. An explicit test of the whole range of DirectX upload strategies is also planned. After making our measurement tool publicly available, we hope to collect performance results for an even wider range of systems and applications with the support of other users. Our measuring tool will be extended and improved to further streamline the workflow of such performance evaluations.

Acknowledgements

This work is partially funded by Deutsche Forschungsgemeinschaft (DFG) as part of Collaborative Research Centre SFB 716.

References

[1] C. Bajaj, P. Djeu, V. Siddavanahalli, and A. Thane. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 243–250, 2004.

[2] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Proceedings of Point-Based Graphics '05*, pages 17–141, 2005.

machine	Im-mediate	Vertex Array	VBO static	VBO stream	VBO dynamic	Geo combo	Geo primitive	Geo VBO static	Geo VBO stream	Geo VBO dynamic	Instancing combo	Instancing primitive
100,000 Molecules												
P4 68	1.69	2.08	7.14	7.14	7.14	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Core2 GTX280	2.87	5.52	105.99	105.61	106.25	22.46	49.53	47.30	47.32	47.32	89.66	104.20
Core2 88GTX	3.19	5.94	106.73	106.73	106.74	1.98	6.41	6.37	6.37	6.36	92.88	103.30
Core2 86GT	3.20	5.96	31.06	31.06	31.06	0.50	1.34	1.34	1.34	1.34	26.73	32.23
Core2 79GT	3.19	5.29	27.72	27.72	27.73	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Core2 68GS	3.24	3.75	9.59	9.59	9.59	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Athlon 88GTX	2.40	4.88	96.00	96.38	96.19	1.90	6.35	6.27	6.27	6.27	56.67	56.32
Athlon 86GT	2.33	4.95	30.07	30.06	30.06	0.49	1.29	1.29	1.29	1.29	26.74	32.23
Athlon 79GT	2.35	4.64	27.02	27.00	27.00	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Athlon 68GS	2.32	3.31	9.51	9.51	9.51	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Phenom 88GTX	2.91	5.59	107.57	107.48	106.80	1.97	6.40	6.36	6.37	6.37	93.03	93.03
Phenom 86GT	2.90	5.49	31.12	31.12	31.12	0.50	1.34	1.34	1.34	1.34	26.74	32.24
Phenom 79GT	2.89	5.12	27.34	27.34	27.34	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Phenom 68GS	2.88	3.63	9.55	9.55	9.55	n/a	n/a	n/a	n/a	n/a	n/a	n/a
1,000,000 Molecules												
P4 68	0.04	0.05	0.08	0.08	0.08	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Core2 GTX280	0.10	0.14	7.63	7.78	7.88	0.36	2.07	1.84	1.84	1.87	6.48	7.24
Core2 88GTX	0.11	0.16	10.12	10.12	10.12	0.09	0.17	0.17	0.17	0.17	9.10	9.12
Core2 86GT	0.11	0.16	0.72	0.72	0.72	0.06	0.07	0.07	0.08	0.07	0.52	0.89
Core2 79GT	0.11	0.15	0.54	0.54	0.54	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Core2 68GS	0.11	0.12	0.23	0.23	0.23	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Athlon 88GTX	0.08	0.11	9.05	8.99	9.05	0.07	0.13	0.13	0.13	0.13	3.37	3.44
Athlon 86GT	0.08	0.11	0.45	0.44	0.45	0.05	0.06	0.06	0.06	0.06	0.40	0.74
Athlon 79GT	0.08	0.11	0.40	0.40	0.40	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Athlon 68GS	0.08	0.09	0.17	0.17	0.17	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Phenom 88GTX	0.10	0.15	10.15	10.12	10.15	0.08	0.16	0.16	0.16	0.16	7.99	7.92
Phenom 86GT	0.10	0.15	0.68	0.67	0.67	0.06	0.07	0.07	0.07	0.07	0.48	0.84
Phenom 79GT	0.10	0.14	0.49	0.49	0.50	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Phenom 68GS	0.10	0.11	0.21	0.21	0.21	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 3: The frames per second performance values for the two real-world datasets (upper part: 100,000 molecules; lower part: 1,000,000 molecules).

[3] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *Pacific Graphics'03*, pages 335–343, 2003.

[4] I. Buck, K. Fatahalian, and P. Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *Poster Session at GP2 Workshop on General Purpose Computing on Graphics Processors*, 2004. `http://gpubench.sourceforge.net/`.

[5] M. Chuah and S. Eick. Glyphs for software visualization. *Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on*, pages 183–191, Mar 1997.

[6] GL_EXT_draw_instanced Specification. `http://opengl.org/registry/specs/EXT/draw_instanced.txt`.

[7] M. Eissele and J. Diepstraten. *GPU Performance of DirectX 9 Per-Fragment Operations Revisited*, pages 541–560. Shader X4: Advanced Rendering with DirectX and OpenGL. Charles River Media, 2006.

[8] S. Grottel, G. Reina, J. Vrabec, and T. Ertl. Visual Verification and Analysis of Cluster Detection for Molecular Dynamics. In *Proceedings of IEEE Visualization '07*, pages 1624–1631, 2007.

[9] S. Gumhold. Splatting illuminated ellipsoids with depth correction. In *Proceedings of 8th International Fall Workshop on Vision, Modelling and Visualization*, pages 245–252, 2003.

[10] M. Hopf and T. Ertl. Hierarchical Splatting of Scattered Data. In *Proceedings of IEEE Visualization '03*. IEEE, 2003.

[11] T. Klein and T. Ertl. Illustrating Magnetic Field Lines using a Discrete Particle Model. In *Workshop on Vision, Modelling, and Visualization VMV '04*, 2004.

[12] O. D. Lampe, I. Viola, N. Reuter, and H. Hauser. Two-level approach to efficient visualization of protein dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1616–1623, Nov./Dec. 2007.

[13] Markus Gross and Hanspeter Pfister, editor. *Point-Based Graphics*. Morgan Kaufmann Publishers, 2007.

[14] T. Ochotta, S. Hiller, and D. Saupe. Single-pass high-quality splatting. Technical report, University of Konstanz, 2006. Konstanzer Schriften in Mathematik und Informatik 219.

[15] OpenGL 3.0 Specification. `http://www.opengl.org/registry/doc/glspec30.20080811.pdf`.

[16] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[17] G. Reina and T. Ertl. Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization Eurovis '05*, 2005.

[18] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, 2000.

[19] M. Sainz, R. Pajarola, and R. Lario. Points Reloaded: Point-Based Rendering Revisited . In *SPBG'04 Symposium on Point - Based Graphics 2004*, pages 121–128, 2004.

[20] M. Tarini, P. Cignoni, and C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–1244, 2006.

[21] R. Toledo and B. Lévy. Extending the graphic pipeline with new gpu-accelerated primitives. Tech report, INRIA Lorraine, 2004.