GPU-based Interactive Visualization

Nicolas Cuntz
Martin Lambers
Christof Rezk-Salama
Andreas Kolb

Abstract This chapter introduces basic visualization techniques conforming to the parallel architecture of graphics processing units (GPUs). The efficient use of modern graphics hardware is the key to interactivity, which is one of the main requirements of visualization applications. Due to the peculiarities of GPU programming, one must carefully design algorithms in order to exploit the GPU's potential. The visualization method also strongly depends on the visualized data set. Accordingly, the choice of suitable data structures is an important aspect. In our overview, we concentrate on space-discrete structures, namely 2D/3D grids and particle systems.
The presented techniques are demonstrated by means of two examples. The first one addresses the visualization of SAR (Synthetic Aperture Radar) images and thus makes use of 2D grid techniques, the second one covers flow visualization using particle- and grid-based time surfaces. The latter example also shows the benefit of the combination of both (grid and particle) structures.

Key words: ACM Categories: I.3.1 Hardware Architecture (Graphics processors), I.3.6 Methodology and Techniques (Graphics data structures and data types)

1 Introduction

The mission of any type of visualization process is finding answers to very specific questions about the underlying data. The vast amount of information contained in a typical data set, however, cannot be displayed in a static image. In order to find an answer to his question, the domain scientist must analyze

Institute for Vision and Graphics, University of Siegen, Germany `http://www.ivg.informatik.uni-siegen.de`

the data set by interactively exploring it. He wants to have full control over the visualization system in real-time. The application should allow him to navigate through the data and to modify any visual parameter at runtime, while being provided with an immediate visual feedback of his operations.

Ever since the field of visualization emerged as a scientific discipline, the community has made efforts to exploit the high computational power of graphics workstations in order to visually display scientific data. Driven by the mass market for 3D computer games, consumer graphics hardware has become comparable to dedicated workstations in terms of performance in the late 90ies. The high level of programmability soon lead to off-the-shelf graphics boards which would easily outperform most of the expensive graphics workstations of that time.

The speed at which new and more powerful computer hardware emerges is breathtaking. If we cannot solve a problem today, due to a lack of computational power, bandwidth or storage capacity, we may be enticed to say we only have to wait a few months until the required capabilities exist. This, however, is naive thinking, neglecting the fact, that the size of the problem will scale as fast or even faster than the resources available for solving it. Not in spite of, but due to the fast evolution of graphics hardware we must continue developing efficient algorithms to exploit the available resources.

Probably the first rendering method for visualizing 3D scalar data, which can be considered a GPU-based technique was proposed in a technical report by Wilson et al. [28]. The authors utilize hardware-accelerated 3D textures for fast spatial interpolation. At SIGGRAPH 1999, Westermann and Ertl showed how to efficiently exploit graphics workstations for volume visualization [26]. The first volume rendering approach tailored to consumer-level graphics hardware was presented by Rezk-Salama et al. in 2000 [18]. At that time, flow visualization was difficult to perform on graphics hardware due to the lack of floating point operations. Texture hardware was utilized for texture advection [25] and image-based flow visualization [23]. Today, the GPU has evolved into a flexible general purpose stream processor which has successfully overcome almost all of the restrictions on programmability from the past.

Today's GPUs consist of multiple parallel processing pipelines and are programmable with high level languages. Research results of the recent years have shown that they are ideal for implementing real-time visualization systems. The programming model of a parallel GPU, however, is fundamentally different from the common CPU programming model. Data structures and algorithms have to be carefully designed.

Parallel computing is not a new concept. It was there long before the first graphics processor. Before programmable graphics processors became popular, however, parallel programming always was a difficult task, mastered only by a few people who had access to parallel computers. Although modern GPUs are often applied to non-graphics computing today, we believe the programming of the graphics pipeline with its vertices and pixels still

is the easiest access and the key to understanding parallel processing. This chapter explains fundamental techniques to leverage the power of GPUs for interactive visualization, focussing on general data structures and processing paradigms, and examples.

2 GPU Programming

The GPU has been designed to allow fast image synthesis for complex scenes including large polygonal geometries and complex illumination models. The pipeline architecture of the GPU with its dedicated memory of high bandwidth makes it ideal for computations on regular grids or discrete sets such as particle systems. However, attaining maximum performance requires understanding of the underlying architecture. For microprocessors the cost of communication usually is considerably higher than the cost for computation. When programming GPUs, it is thus mandatory to be aware of the streaming model, the functional units and the memory cache hierarchy. In this section, we will focus on the main GPU aspects related to interactive visualization of large or computationally expensive data sets.

2.1 The Graphics Pipeline

The GPU pipeline is outlined in Figure 1. The two most important stages of the classic pipeline are vertex processing and fragment processing. In order to visualize a given data set, it must be converted into a scene description, which comprises vertex data and texture images.

The first stage in the pipeline after the GPU receives input data is the vertex stage. The vertex processor traditionally calculates linear transforma-

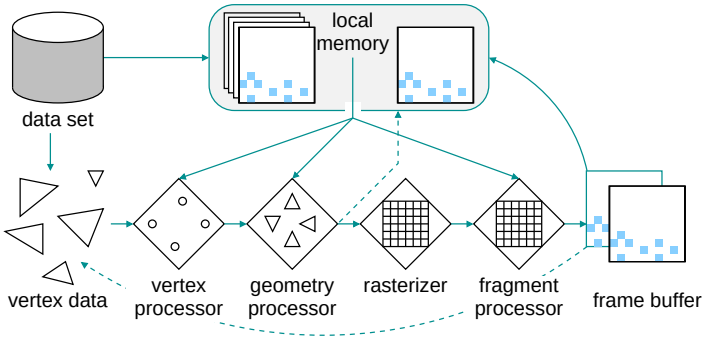

Fig. 1 The GPU pipeline

tions of the individual vertices, such as the well-known mapping from model coordinates to camera coordinates, and finally the projective transformation of the viewing frustum into the canonical view volume. The programmable vertex processor allows us to replace this fixed-function computation by a customized vertex shader which may include memory access to additional data through texture lookups. At the end of vertex processing, the vertices are assembled into geometric primitives, such as triangles, lines, or points and handed to the rasterizer.

Afterwards, the rasterizer decomposes the geometric primitives into fragments with the size of a pixel. The fragment program is executed once for each individual fragment and may access additional information from texture memory. Modern vertex and fragment processors support dynamic flow control, which means that the respective programs may contain loops and conditional branches. After the fragment stage, the shaded fragments are sent to the frame buffer, where raster operations ensure that only visible fragments are drawn and transparent fragments are blended with existing frame buffer pixels.

In order to attain maximum performance, all stages of the pipeline must be kept busy. This leads us to one of the main drawbacks of the classical GPU pipeline: Effective load-balancing must be ensured by the programmer. For scenes with a large overdraw due to a high depth complexity, the fragment stage will limit the overall performance. For geometry with a huge number of small triangles, the vertex processor will most likely be the bottleneck. To tackle the problem of load-balancing, the new generation of graphics hardware abandons the classic GPU pipeline and replaces it by a *unified shader model*. Fragment and vertex processors are now functionally identical and a scheduler takes care of distributing the computational load among the available processing units.

Another significant improvement of the new generation of GPUs is an additional pipeline stage, which is called the *geometry shader*. This processing unit is inserted between the vertex processor and the rasterizer. It receives one geometric primitive at a time, but may output several primitives simultaneously. This geometry shader removes the inability of the classical GPU pipeline to generate a variable set of geometric primitives on the fly. In combination with the unified shader model the stream output of the geometry shader may be written back to local memory without passing through the rasterization stage. With the geometry shader adaptive tessellation techniques and subdivision surfaces can be implemented efficiently and completely on the GPU.

2.2 Data Streaming and Multipass Rendering

For practical implementations, the computational power of the GPU may be utilized for many tasks that are not related to image synthesis, such as numerical integration of particle velocities, collision and intersection calculation. The result of such computation will not be directly displayed on the screen and will thus be rendered into off-screen targets. Such GPU-based non-graphics related methods are classified as *general-purpose* GPU programming, and are pushed by a large number of works and a constantly growing community known as GPGPU [17].

An important aspect of GPGPU is the use of textures as equivalent to arrays in CPU programming languages. Current GPUs support 1D, 2D and 3D textures. The task of updating a 2D texture is usually done by rasterizing a primitive that covers the texture dimensions. A fragment program defines the actions that are necessary to compute an output pixel and writes the result to an off-screen render target. This way, complex operations can be realized on arbitrary data stored in textures [21].

It is not unusual to perform multiple off-screen passes through the graphics pipeline before the final image is generated. Each render pass has a designated portion of local memory to write to, which are called *render targets*. Off-screen render targets can be created as so-called *frame buffer objects*. They can contain multiple color layers to which the fragment processor writes simultaneously. The render targets written to in previous passes may be accessed as texture images from within the current fragment or vertex program. The flexible setup of off-screen render targets allows us, for example, to directly render into the faces of an environment cube or into slices of a 3D texture. Sophisticated applications may re-interpret the content of the render target and use it as vertex data in a subsequent pass.

The parallel (streaming) architecture of the GPU yields some important restrictions for the programmer: input and output data are strictly separated, i.e. reading a data element that is written during a render pass leads to undefined behavior. This input-output disjunction is a typical requirement for streaming algorithms. The computation which is performed on all distinct output elements is called a *kernel*. The number of texture fetches within one kernel is limited by bandwidth and speed.

2.3 Scatter/Gather

GPU programmers distinguish between two types of indirect memory access operations, *gather* and *scatter* [17]. According to this scheme, a *gather* operation reads multiple values at variable locations in order to create a single output element. In contrast, a scatter operation writes multiple values at variable locations.

In principle, GPUs offer convenient support for the gather scheme: A gather operation can be implemented by performing dependent texture fetches, i.e. by reading the texture information at arbitrary texture coordinates which can be the result of prior computations. For the counterpart, scattering, one would like to use a dependent texture write operation. Unfortunately, an output element (the current fragment) has a fix location predetermined by earlier stages in the graphics pipeline, i.e. the vertex and the geometry program. There is no way on today's GPUs to change the output location within a fragment program. Instead, programmers must circumvent this limitation in order to achieve data scattering:

- Rearrange programs in order to solve the problem in terms of gather
- Use multiple render targets (MRT) in order to *scatter* the output into a restricted number of output arrays (8 at the time of writing this book)
- Use the vertex processor in order to move the output location – following this approach, simple marker geometries (e.g. point sprites) which enclose a certain set of fragments are rendered.

Each of these methods yields its own advantages and drawbacks, thus a programmer must carefully decide which technique is appropriate for a given problem. Rearrangement usually results in splitting into several rendering passes, which can be a difficult task regarding code clarity and performance. Multiple render targets require a very strict memory model consisting of a stack of equally-sized 2D textures. The output is written into all textures simultaneously, but at one common location. This restriction can result in inefficient data management. Using marker geometries splits the control flow into independent program instances for all fragments, thus making data interchange difficult.

The restrictions regarding scatter operations make general tasks such as sorting or the use of advanced data structures difficult, thus visualization systems must cope with these difficulties as well. A detailed description of aspects related to GPU memory access can be found in [22].

3 Data Management

In general, visualization applications require means for fast data traversal and large data sets, e.g. for load balancing if the data does not fit into graphics memory or even main memory (out-of-core). The data storage is strongly application dependent. In the following, the most important discrete structures, uniform 2D/3D grids and particle sets, are presented.

3.1 Resolution Hierarchies

When visualizing a large 2D data set, it is not necessary (and often impossible) to hold the complete data set in graphics memory. For example, if the user visualizes a small part of a large 2D data set, it is often sufficient to store and process only the essential part of the data on the graphics hardware. In other cases where a large region of the data has to be visualized, the processing of a downscaled version might suffice because of the fixed resolution of the display.

Tiling pyramids provide a resolution hierarchy that allows to keep only the currently necessary subset of the data in graphics memory and load other parts quickly on demand. In order to build a tiling pyramid, the data is divided into tiles of a fixed size (see Fig. 2). At the lowest pyramid level, the data set is stored in its original resolution. Each higher pyramid level halves the resolution in both directions, and therefore contains one quarter the number of tiles of the lower level. Using tiling pyramids, the application can decide which subset of tiles matches the current visualization requirements best, depending on the zoom level and the region of interest. This way, only the corresponding subset is stored and processed on the graphics hardware.

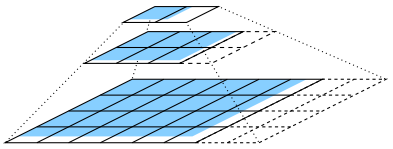

Fig. 2 A tiling pyramid for a 2D data set.

A tiling pyramid needs to be built only once and can be stored on hard disk for later reuse. Tiles can be cached in main memory. Both the loading of tiles from hard disk to main memory and from main memory to graphics memory can be done asynchronously, i.e. without the need to halt the interactive visualization interface. This allows proactive tile loading based on assumptions about the users next actions, so that data is readily available when needed.

Tiles can be extended with additional information from neighboring tiles to allow data processing techniques on local neighborhoods (see Sec. 4.1). In addition to visualization of 2D data sets, applications of tiling pyramids include various level of detail (LOD) techniques, e.g. for terrain visualization [30].

Similar approaches can be applied to 3D data sets.

3.2 Spatial Hierarchies

Hierarchical data structures are frequently used to implement fast data traversal and multiresolution techniques, e.g. for volume rendering [3]. Octrees can be used to accelerate iso-surface reconstruction by skipping regions

which are not intersected by the iso-surface [27]. Advanced techniques for texture-based volume rendering can also use octrees for level of detail techniques [12, 24].

Octrees are ideal for implementing effective empty-space leaping and adaptive sampling techniques on the GPU with both raycasting and slicing approaches. GPU-based raycasting of iso-surfaces inside large volume data sets can clearly benefit from octrees [5]. The usage of octrees for texture slicing can cause a significant bandwidth load, due to the large amount of vertex data which must be updated and transferred for each frame. This load can significantly be reduced by implementing the intersection calculation as a vertex program [19].

For interactive rendering of large data sets, compressed 3D data or time series can be decoded by the GPU at run time. Besides the hardware native texture compression schemes such as S3TC, a number of different techniques have been proposed including wavelet compression [7], packing techniques [15], and vector quantization [20].

3.3 Particle Sets

Particle sets have many applications in computer graphics. Especially in flow visualization, particle trajectories give an intuitive impression of the motion within the visualized medium. Furthermore, particles are frequently used to model physical behavior like fluid dynamics.

The GPU is a very well-suited tool for particle processing. This has been shown by numerous works in the past. For example, Krüger et al. have demonstrated a completely GPU-based flow visualization system [11]. The basic technique for GPU-based particle tracing has been presented in 2004 [8, 10].

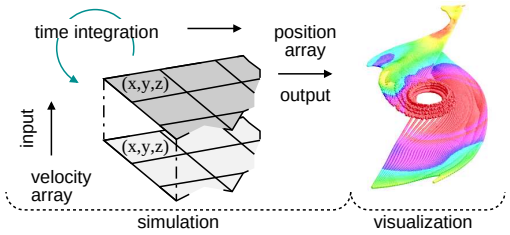

Fig. 3 Parallel processing of a particle set

The main idea of GPU-based particle systems is to represent the particle system by a 2D texture with every texel corresponding to one particle location in 3D space. Particles are moved by processing all texels during one single render pass. This is done as described in Sec. 2.2. The rasterization initiates a

fragment program which performs a time integration (e.g. using a high order Runge-Kutta scheme) and other application specific tasks. See Fig. 3 for an overview of GPU particle tracing.

In contrast to grid structures, there is no spatial coherence between one texel and its neighborhood. Techniques for particle coupling have been introduced for fluid dynamics [6, 9], but it is still a difficult task which typically involves a hierarchical structure or particle sorting.

4 2D and 3D Data Processing

This section presents general processing techniques that can be used in the context of GPU-based interactive visualization. This includes techniques for data preprocessing with interactive parameter adjustment, e.g. denoising or smoothing, as well as techniques that manipulate objects that are to be visualized, e.g. particle or surface manipulation techniques. The presented techniques make heavy use of the data structures described in Sec. 3.

4.1 Tile-based 2D Data Processing

In the 2D case, data is usually divided into tiles, stored as textures in graphics memory (see Sec. 2). Resolution hierarchies as presented in Sec. 3.1 can be used to manage the set of tiles.

Data preprocessing tasks such as denoising or smoothing can be implemented on the GPU. This allows to integrate the adjustment of preprocessing parameters into the interactive interface, thereby making the visualization system more flexible and powerful.

Processing the image data can be performed by a fragment program as described in Sec. 2.2. With tile-based data processing, all operations must be local, i.e. the fragment program can only access the subset of data that is stored in the current tile. This has two consequences.

First, in order to allow fragment programs to work on local neighborhoods even at border pixels, each tile needs to be extended with a sufficiently wide border that contains information from neighboring tiles. This border is only used to provide neighborhood information and not for visualization. For example, to be able to compute the 3×3 Gauss filter accurately without using special border handling, the tiles need to be extended with a 1 pixel wide border.

Second, special arrangements are necessary to allow the fragment programs to use global information about the data set, e.g. its minimum or maximum value. Generally, there are three possibilities, the choice of which depends on the individual application:

1. Compute the global information in a preprocessing step before the interactive visualization, e.g. while building a tiling pyramid.
2. Guess the global information based on a local neighborhood.
3. Introduce an additional user-controlled parameter that replaces the global information.

4.2 Sliced 3D Grids

Processing 3D textures requires render-to-texture functionality. Current GPUs support rendering to a fixed set of slices, however, there is no way to update the whole texture in one single pass. A frequent solution for this problem is the mapping of the 3D grid to a stack of tiled 2D textures. Using this structure, any part of the texture can be written by rendering an appropriate primitive.

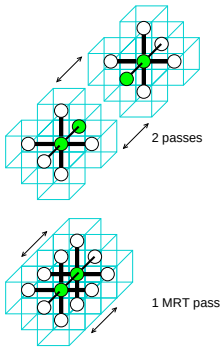

Fig. 4 Usage of MRT for a 3D kernel

It is possible to use MRT (see Sec. 2.3) in order to write multiple fragments simultaneously at a specific location. This MRT scheme can be used for optimizing the number of texture fetches within a kernel in cases where multiple fragments share common input data (see [9]). Fig. 4 demonstrates the advantage of this approach on a simple 3D grid kernel evaluated at two adjacent positions (marked in green). On the top, two (adjacent) kernel instances are processed in two independent passes (14 texture fetches); on the bottom, both instances are combined within a single pass with two render targets (12 texture fetches). Using the MRT kernel, less texture fetches are necessary. This benefit becomes more relevant the more render targets are used.

When surfaces are represented in a uniform 3D grid, narrow band approaches [14], e.g. similar to traditional virtual memory schemes, can significantly reduce the memory consumption and the run time.

4.3 Hybrid Eulerian-Lagrangian Techniques

A quite new and challenging approach is the coupling of grid (Eulerian) and particle (Lagrangian) approaches. The main challenge is the data transfer between both representations, which requires specific algorithmic approaches.

One important example for this scheme is the particle level set (PLS) approach [4] for surface representation, which is frequently used for fluid dynamics. This method is also well-suited for an arbitrary flow visualization

Fig. 5 Zalesak's sphere, 360° rotation. From left to right: Initial, rotated without and with PLS correction. Grid resolution: 128^3, 524288 particles, > 10 FPS (source: [2])

by means of time surfaces. It has been shown that the PLS technique can be ported to the GPU in a very efficient way [2]. In the following, the main ideas are outlined.

The key idea of the level set method in general is the representation of an object's boundary, the interface I, by the iso-contour $I(\phi) := \{x \in D | \phi(x) = 0\}$ of a level set function $\phi : D \rightarrow \mathbb{R}$ defined in a higher dimensional domain D. Usually, the level set is initialized to be a signed distance function. The motion of the interface is performed by evolving ϕ within a velocity field.

Enright et al. [4] use a fast first order accurate semi-Lagrangian method in their PLS method in order to evolve ϕ. Without correction, this leads to high inaccuracies already after few time steps, because the committed errors quickly accumulate, generally resulting in a significant volume loss. The PLS method aims to prevent the numerical diffusion caused by the advection by adding a particle tracing approach. Particle tracing is much more accurate than the grid-based advection of the level set, especially if a high order Runge-Kutta integration is used. Thus, it is reasonable to rely on the particles to correct the interface representation. For this purpose, (signed) particles are placed near to the interface. The correction step selects a set of particles and propagates a local level set surrounding each particle into a new (corrected) level set. The result is known to be comparable or even superior to high order grid advection schemes when using a large set of particles. Fig. 5 shows an example geometry (Zalesak's sphere) frequently used for the demonstration of PLS.

The GPU-based PLS system presented by Cuntz et al. [2] uses a distance transform in order to perform an efficient and easy reseeding of particle locations around the interface. This step can be seen as a *grid-to-particle* interchange. On the other hand, the correction step involves scattering marker geometries into the level set by making use of min-max blending, thus constituting *particle-to-grid* interchange. This GPU interpretation of PLS turned out to run about 10× faster than the CPU-based `PLSlib` by Mokberi and Faloutsos [16] while achieving a slightly higher precision.

Another example for hybrid particle-grid systems is neighborhood searching in particle sets as it is presented in [9] and [6], where data interchange between particles and the grid resembles the mentioned GPU-based PLS technique by Cuntz et. al.

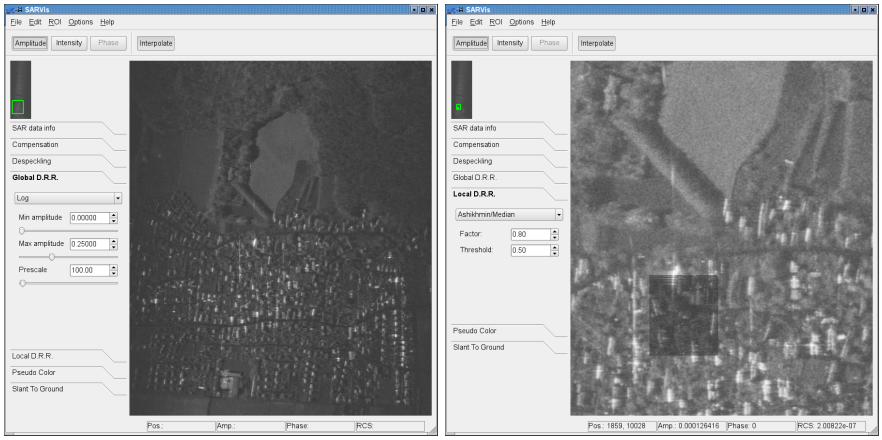

Fig. 6 Interactive visualization of SAR data with interactive parameter adjustment (raw data courtesy of FGAN, processed at ZESS/IPP).

5 Example: SAR Data Visualization

Synthetic Aperture Radar (SAR) is a technique to acquire ground images independent of weather and daylight conditions, using airborne or spaceborne radar platforms. Challenges of interactive SAR data visualization include efficient memory management, reduction of multiplicative speckle noise, and mapping of high dynamic range SAR data to the lower dynamic range gray levels of typical display devices. In this section, we show how tiling pyramids and 2D data processing techniques can be combined to solve these problems.

5.1 Data Management

A SAR image pixel consists of a 32bit floating point value that stores the amplitude of the complex reflectivity information (the phase is not used for 2D visualization). Typical images contain 16384×4096 pixels, which results in a raw data size of at least 256MB, but much larger images are common.

Tiling pyramids (see Sec. 3.1) are well suited to manage this amount of data, because they allow to store only the currently required data subset in graphics memory, process this data on the GPU, and load other data parts quickly on demand. The tiling pyramid is built in a non-interactive preprocessing step and stored on hard disk. Various global values are computed from the data during this step, for example the minimum and maximum amplitude value. These values are needed later by a variety of processing methods, including statistic approaches such as the Xiao despeckling filter [29].

The currently required subset of tiles is deduced from the current region of interest (ROI), which is the area of the data that the user wants to display (see Sec. 3.1). This subset of tiles is loaded into graphics memory, processed according to the currently selected methods and their parameters, and finally displayed on the screen. This process is continuously repeated to acknowledge user interactions.

5.2 Data Processing

The selected subset of tiles is processed using a tile-based 2D data processing chain as described in Sec. 4.1. The processing chain is determined by the currently selected processing methods. Its most important components are despeckling and dynamic range reduction.

Most despeckling filters for SAR images manipulate a pixel based on statistical properties of a local neighborhood. Such filters usually need two processing steps to compute the local mean and variance in a separable manner and to manipulate the current pixel based on this information. Other filter types may require more steps.

Dynamic range reduction methods often use a fixed function to transform amplitude values to gray levels. Such global methods are implementable using a single processing step. More complex local methods adapt themselves to the properties of local neighborhoods to achieve higher local contrast and better preservation of details. This usually requires more processing steps.

The fragment programs that implement the individual processing steps are provided with user-adjusted processing parameters. In addition, they need to know the pyramid level from which the tiles were taken, so that they can adapt the size of local neighborhoods to the different resolution levels if necessary. Furthermore, global information that was computed while building the tiling pyramid can be used.

Each processed tile is finally displayed at the appropriate position.

5.3 Results

The SAR visualization application shown in Fig. 6 allows both interactive navigation through the data and interactive adjustment of processing parameters [13]. Interactive frame rates can be achieved even for complex processing methods and large display resolutions.

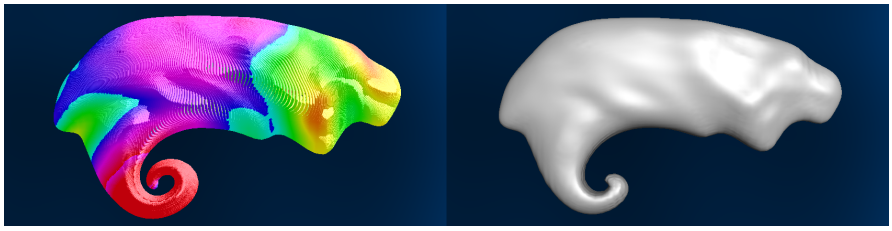

Fig. 7 Left: Purely particle-based time surface. The particles are rainbow-colored according to their velocity. The red color denotes a high magnitude. Right: Purely grid-based time surface. Both: A spherical shape is moved in an unsteady flow (raw data: courtesy of DKRZ Hamburg).

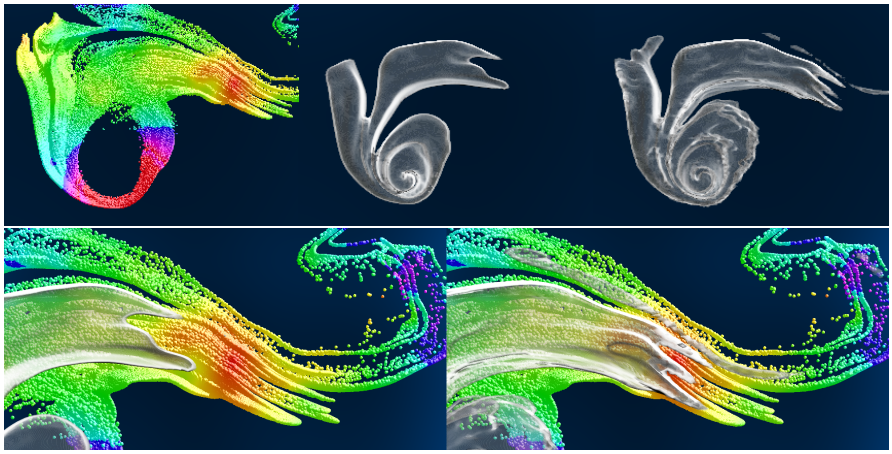

Fig. 8 First row: Purely particle-based time surface, level set time surface, PLS-based time surface. Second row: Direct comparison of particles and level set (left), of particles and PLS (right). (raw data: courtesy of DKRZ Hamburg)

6 Example: Particles, Flow Visualization

Flow visualization has various applications. For example, in the context of climate research, geophysical flows (e.g. atmospheric or oceanic circulations) are simulated and then visualized in order to analyze the result [1]. Interactive flow exploration is an intuitive way to understand climate phenomena, very different to statistical methods or precomputed and thus uncontrollable animations.

The application discussed in this section demonstrates the particle- and grid-based techniques discussed in Sec. 3.3 and Sec. 4.2. A flow volume, more precisely, a time surface is moved within an unsteady flow. The visualization starts with an initial shape (here, a sphere is used). Its surface is given as a closed boundary. Then, the object is distorted during multiple animation steps. Fig. 7 shows two differently generated time surfaces. The left side has

been obtained by an accurate second order Runge-Kutta time integration of particles located on the initial surface. On the right side, one can see the result of a grid-based level set advection after the same number of animation steps.

The difference between both approaches is pointed out in Fig. 8. The first image shows that particle integration tends to produce sparsely populated regions. In the example shown, the particles are pushed away from the twirl's center, resulting in a hole without particles. The second image shows a problem inherent to level set advection. Surface features are blurry and a volume loss occurs due to numerical diffusion. The effect is clear when comparing the particle trail in the upper right part with the corresponding level set part. This is shown in a rendering of both techniques in the second row (left image).

The advantages/disadvantages of purely grid-based and purely particle-based techniques for flow visualization are listed in the following overview:

Purely particle-based

+ The system is very fast due to parallel particle processing. A high number of particles can be traced at interactive frame rates (over 1000,000 at > 60 FPS)
− Moving the particles can result in inhomogeneous distribution. It is necessary to reposition the particles during visualization in order to focus interesting features of the flow. When visualizing a time surface, the topology is not necessarily maintained due to unspecified regions.

Purely grid-based

+ The implementation is very straight forward, and fast if combined with a narrow band approach for surface representation.
− Advection suffers from numerical diffusion. This turns into an accumulating volume loss in the level set representation.
− The visualization of the voxel grid is difficult and it generally requires a volume rendering approach.

The PLS approach described in Sec. 4.3 combines both, the particles and the grid, in order to reduce topological problems *and* numerical errors. One can observe on the third image in Fig. 8, that the volume loss of the level set volume is compensated. In direct comparison with the particle-based approach (second row), the PLS volume shows more surface details.

6.1 Results

Both particle and grid approaches can be efficiently ported to the GPU. This opens the door to various interactive flow applications, including particle

tracing techniques and volume-based visualization. Combining particles and a grid according to the PLS technique unifies the advantages of both structures.

7 Summary

Interactive visualization applications require high computing power and a large memory bandwidth to process large amounts of data. Today's GPUs can be used to meet this challenge. It is, however, mandatory to be aware of the underlying architecture to make full use of the GPU's potential.

We have examined the different functional units of the GPU which allow complex and flexible general purpose processing of the data. The data stream model of the GPU imposes some restrictions on data structures and algorithms. Most notably, input and output data are strictly separated, and scatter operations are not easily implemented.

Data management techniques for fast traversal of large data sets must be designed to fit the GPU's concepts. We have explained techniques for the most important discrete structures, uniform 2D/3D grids and particle sets, as well as data processing methods that use and manipulate such structures. This includes preprocessing tasks like denoising or smoothing as well as particle and surface manipulation techniques.

We have presented two real world examples of visualization systems. These examples combine the techniques presented in this chapter to visualize complex and large data sets with a high degree of interactivity.

References

1. N. Cuntz, M. Leidl, A. Kolb, C. Rezk-Salama, and M. Böttinger. GPU-based dynamic flow visualization for climate research applications. In *Proc. Simulation and Visualization*, 2007.
2. N. Cuntz, R. Strzodka, and A. Kolb. Parallel particle level set method on the GPU. Sym. on Interactive 3D Graphics & Games, Seattle, Poster-Session, 2007.
3. K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. AK Peters, 2006.
4. D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comp. Phys.*, 183(1):83–116, 2002.
5. M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proc. Eurographics*, 2005.
6. T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. Computer Graphics International, 2007.
7. I. Ihm and S. Park. Wavelet-based 3D compression scheme for very large volume data. In *Proc. Graphics Interface*, 1998.
8. P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based particle engine. In *Proc. Graphics Hardware*, pages 115–122, 2004.
9. A. Kolb and N. Cuntz. Dynamic particle coupling for GPU-based fluid simulation. In *Proc. 18th Symp. on Simulation Technique*, pages 722–727, 2005.

10. A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proc. Graphics Hardware*, 2004.
11. J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3D flows. *IEEE Trans. on Vis. and Comp. Graph.*, 11(6), 2005.
12. E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of IEEE Visualization*, 1999.
13. M. Lambers, A. Kolb, H. Nies, and M. Kalkuhl. GPU-based framework for interactive visualization of SAR data. In *Proc. Int. Geoscience and Remote Sensing Symposium (IGARSS)*, 2007.
14. A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.
15. W. Li and A. Kaufman. Texture partitioning and packing for accelerating texture-based volume rendering. In *Graphics Interface*, pages 81–88, 2003.
16. E. Mokberi and P. Faloutsos. A particle level set library. *Technical Report*, 2006. http://www.magix.ucla.edu/software/levelSetLibrary/.
17. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
18. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. Graphics Hardware*, 2000.
19. C. Rezk-Salama and A. Kolb. A vertex program for efficient box-plane intersection. In *Proc. Vision, Modeling and Visualization*, 2005.
20. J. Schneider and R. Westermann. Compression domain volume rendering. In *Proc. IEEE Visualization*, 2003.
21. R. Strzodka. *Hardware Efficient PDE Solvers in Quantized Image Processing*. PhD thesis, University Duisburg-Essen, 2004.
22. R. Strzodka, M. Doggett, and A. Kolb. Scientific computation for simulations on programmable graphics hardware. *Simulation Practice & Theory*, 13(8), 2005.
23. J. J. van Wijk. Image based flow visualization. In *Proc. ACM SIGGRAPH*, pages 745–754, 2002.
24. M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-Of-Detail Volume Rendering via 3D Textures. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 7–13, 2000.
25. D. Weiskopf, T. Schafhitzel, and T. Ertl. Texture-based visualization of unsteady 3D flow by real-time advection and volumetric illumination. *IEEE Trans. on Vis. and Comp. Graph.*, 2007.
26. R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proc. ACM SIGGRAPH*, 1998.
27. J. Wilhelms and A. van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.
28. O. Wilson, A. V. Gelder, and J. Wilhelms. Direct Volume Rendering via 3D-textures. Technical Report UCSC-CRL-94-19, Univ. of Cal., Santa Cruz, 1994.
29. J. Xiao, J. Li, and A. Moody. A detail-preserving and flexible adaptive filter for speckle suppression in SAR imagery. *Int. J. Remote Sensing*, 24(12), 2003.
30. P. Xuexian, Y. Xudong, L. Sikun, and S. Junqiang. High-performance navigation and rendering of very-large scale landscape and seascape. In *Proc. CAD-CG*, pages 377–384, 2005.