Vragments - Relocatability as an Extension to Programmable Rasterization Hardware

Joachim Diepstraten[†], Daniel Weiskopf[†], Martin Kraus*, Thomas Ertl[†]

[†]Visualization and Interactive Systems Institute, University Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany

{diepstraten,weiskopf,ertl}@informatik.uni-stuttgart.de

*Purdue University Rendering and Perception Lab, Purdue Unversity, West Lafayette, IN 47907, USA

kraus@purdue.edu

ABSTRACT

We propose an extension to the pixel pipeline of current programmable rasterization hardware to include the possibility to freely locate the rasterization position of fragments and pixels. The corresponding new primitive name is Vragment (variable fragment). We show how this new functionality could lead to new and wider classes of algorithms in computer graphics, especially in image processing, scientific visualization, geometric modeling, and rendering. A GPU-assisted simulator for programs running on the proposed architecture is presented.

Keywords
Rasterization hardware, programmability, shading language

1. INTRODUCTION

Tremendous changes and improvements have been seen in the field of rasterization hardware since 3D accelerators became a standard configuration for today's PCs. In the last three years a transition from a fixed-function pipeline to a configurable pipeline, and eventually to a programmable pipeline has taken place both for rasterization and vertex processing; and there is still a race going on between different graphics chip manufactures to include additional features to make their hardware even more flexible. When looking at the DirectX9 specification for pixel shader version 3.0 [MS02] and vertex shader version 3.0 [MS02], or the OpenGL 2.0 shading language proposals [Kes03], it is possible to get a glimpse on what lies ahead for the upcoming or maybe even for the next two upcoming generations of graphics chips.

Although past desktop computer graphics innovations were mostly driven by the computer games industry and their special demands [Kirk98], it is likely that this will, in parts, change in the future as manufactures are looking into different markets to sell their hardware. These markets might have different demands compared to the ones of the gaming industry. Therefore, we propose a simple but effective extension to current programmable rasterization hardware to broaden the spectrum of supported algorithms. Programmable relocation of a fragment in the fragment processor makes possible the implementation of a completely new class of algorithms, or alternative implementations of currently used algorithms. Furthermore, we provide a simulator to the research community and graphics chip manufactures to explore today the potential that might be available tomorrow. Experiences of the recent past show that increasing functionality of graphics hardware stimulates researchers in the visualization and computer graphics community to exploit hardware in ways that were never considered by the manufactures in the first place. An early example is the work by Heidrich et al. [Hei99]; recent examples are works by Carr et al. [Carr02], Purcell et al. [Pur02], Krüger and Westerman [Krü03], and Hillesland et al. [Hil03]. Although we present a number of useful algorithms in this paper that become feasible with the proposed extension, we believe that our simulator is even more important because it could inspire other researchers to develop more sophisticated hardware-based techniques.

The remainder of this paper is organized as follows: In the next section a short overview of related work is presented. Section 3 describes our extension in more detail, Section 4 introduces the simulator, the framework, and the extension to hardware shaders we used to simulate the algorithms presented in Section 5, specifically for binning, displacement mapping, particle systems, flow visualization, fur rendering, rendering of arbitrary curves, and special effects. Finally, the paper closes with some results and a discussion of open issues.

2. PREVIOUS WORK

Numerous proposals for extending graphics hardware were made in the past, and some of these eventually made it to products on the market. As a good example, the proposal for the F-buffer by Mark and Proudfoot [Mar01] has been implemented on the R350 graphics processor by ATI. Another good example of proposed extensions which made it actually to product level are texture compression algorithms, e.g. vector quantization in the PowerVR architecture and the S3 texture compression [S398]. Although these two examples focus on the fragment processing part of the graphics pipeline, other papers describe extensions to different parts of the pipeline, for example the vertex processing [Lin01] or tessellation unit [Bóo01].

Another related field are shading languages. Our framework relies on existing programming languages to include only minor extensions for relocating fragment positions. At the moment, the development of shading languages is mainly divided between the two most important rendering APIs: Direct3D with its High-Level Shading Language (HLSL) [MS02] for both fragment and vertex processing, the assembler languages pixel shader version 2.0, 2.0+ (or extended), and 3.0 for fragment processing, and the assembler-like vertex shader 2.0 and 3.0 for vertex processing; OpenGL with ARB_Fragmentprogram and ARB_Vertexprogram extensions for assembler languages and the OpenGL 2.0 shading language proposal on a more abstract level. Other work in this area targets API-independence, such as Cg [Mar03] and the work by McCool et al. [Cool02].

3. ARCHITECTURE

The vragment extension fits very well into the structure of the rendering pipeline on current graphics hardware. Figure 1 schematically shows our proposal for a modified pipeline. Green and blue boxes show elements that are already present on today's GPUs and that are (almost) not changed by our extension. The green boxes indicate parts that are related to vertex and primitive processing; the blue boxes concern operations on fragment and pixel level. The additional parts for the vragment extension are marked red. The “vragment relocation” module is an add-on to the existing fragment processing unit. It only slightly modifies this unit: A readable and writable register that describes the *x* and *y* coordinates of the current fragment in window coordinates is added, along with all the functionality that is available for other registers. Since GPUs already provide numerical operations with floating-point accuracy for a large number of registers, this modification nicely fits into the existing architecture

The second change concerns the “vragment write” module. This module actually writes the vragment information from the fragment processing at the position previously computed through the vragment relocation module. Before writing, another viewport clipping has to be executed as vragments can be freely moved by a fragment program and maybe even outside the viewport, otherwise memory page faults would occur.

Writing pixels in a random order will definitely not come without a penalty because writing vragments to the framebuffer would require a random write access to memory. An exact calculation of the costs for a random write access would require a complete emulation of today's memory technology and caching strategies. Unfortunately most of them are unknown to outsiders due to intellectual property of different companies. But estimates can be extracted from texture-indirection or dependent texture reads, which are already possible with today’s graphics hardware. They provide a random read access to memory and several sources show that dependent texture fetches lead to a significant performance decrease [Rig03,Spi01]. On the other hand new memory technology like DDRII, Rambus and an increasing popularity of dependent texture reads for example in shading calculations will hopefully help to decrease this bottleneck in the future. The vragments extension might also benefit from this as the same strategies to speed up random memory reads could also be used for writing. A special treatment could be beneficial when fragment positions are not changed at all (i.e., standard render code) because the same access mechanisms as used in today's graphics boards can be applied. If fragments positions are changed in a coherent manner (e.g. moving in similar directions), write-caching strategies will greatly reduce memory accessing penalties.

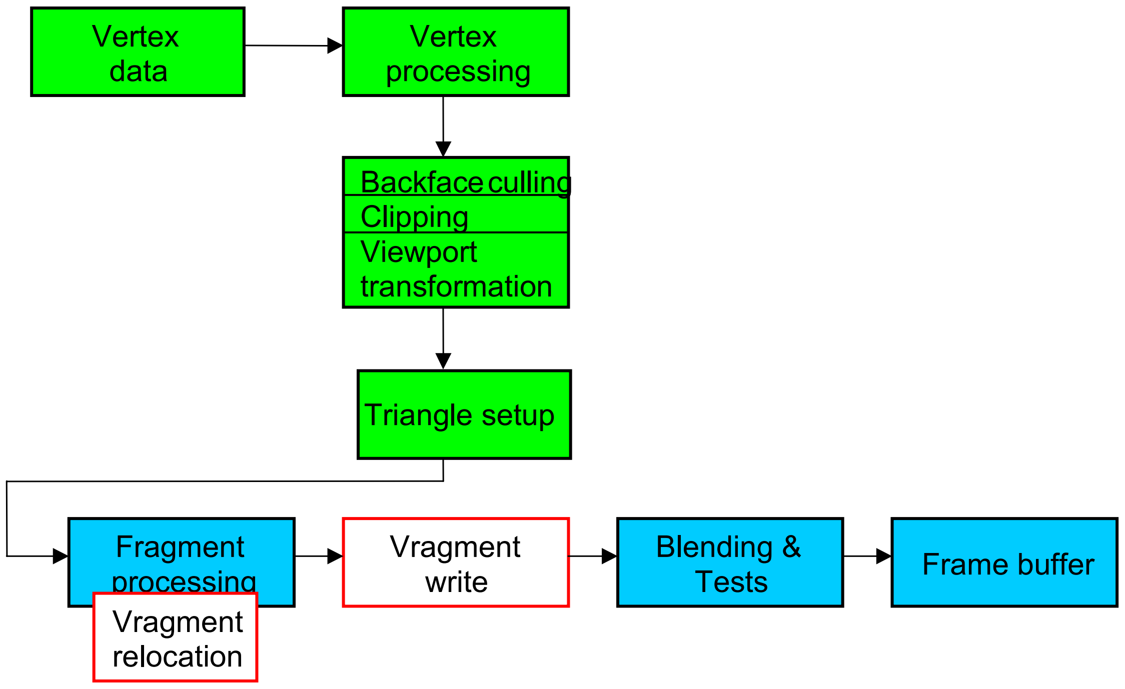

Figure 1. Structure of the rendering pipeline. Green boxes indicate parts that are related to vertex and primitive processing, blue boxes to fragment operations. The additional parts for the vragment extension are marked red.

4. A SIMULATOR FOR VRAGMENTS

Our simulator for the programmable relocation of fragments utilizes both hardware and software support. It is based on DirectX 9. This API was chosen in contrast to OpenGL or a completely self-written software rasterization, because:

- Our extension has only little impact on the rest of the graphics pipeline, and therefore only slight changes to existing GPU support should be introduced, ruling out a self-written software rasterization.
- Only minimal changes to standard pixel shader code is required; thus, our extension does not require modifications of existing fragment programs.
- All features of current graphics chips can be used in the simulator.
- High resolution and / or floating-point framebuffers formats and textures can be accessed without using any special extensions.
- It is easy to switch between a completely software-based and a hybrid software / hardware-based simulation.
- Features of upcoming graphics chips can be simulated through the provided software rasterizer in DirectX 9. This makes it possible to test our extension even with functionality from pixel shader version 3.0 and vertex shader version 3.0, which is not yet available in hardware.

The simulator affects the use of the Direct3D API in two ways: First, a parser translates pixel shader code with our extensions to standard DirectX 9 in order to generate a hardware-supported pixel shader program. Second, the actual rendering of primitives has to be called through special routines of the simulator.

4.1. Extension to pixel shaders

A few extensions are added to pixel shader versions 2.0 (ps2.0) and 3.0 (ps3.0) to provide control over rasterization positions to a shader programmer. Ps3.0 already provides read access to a fragment's raster position through the register *vPos*. For consistency, this register name is used in our extensions as well. Note that *vPos* only stores the *x* and *y* coordinates, the *z* coordinate is stored in *oDepth*. New version commands – *vss.2.0* and *vss.3.0* – are introduced to distinguish our shaders from the standard pixel shader versions.

The final position of a fragment is set by a *mov* instruction applied to the *vPos* register. Also, the *vPos* register has to be explicitly declared at the beginning of the shader. A valid vragment shader only allows one *mov* to *vPos*, which is consistent with the usage of all other output registers *oCn* (*n* describes the output target) and *oDepth* in ps2.0 and ps3.0. The write operation acting on *vPos* can be freely placed within the shader code, with one exception: It has to be located before the output to the color buffer, i.e., before *mov oCn*.

A tiny vragment shader code in assembler language looks like this:

```
// shader begin
vss.2.0          // shader version 2.0
dcl v0           // declare input color
def c0, 10.0, 0.0, 0.0, 0.0
// constant describing a shift
dcl vPos.xy      //declare fragment position
                 //register
add r0, vPos, c0 // add 10 pixels to the
                 // x-pos
mov vPos, r0     // set new fragment
                 // position
mov oC0, v0      //set fragment output
                // color
// shader end
```

The changes to a conventional DirectX 9 pixel shader program are minimal; a programmer can built on her or his knowledge of GPU programming.

4.2. Constraints

The current simulator is subject to some restrictions and constraints concerning shaders. First, the number of available instruction slots is decreased by two for both pixel shader versions because additional operations are required to communicate fragment positions to the simulator. One instruction slot is used for a *mov* instruction that writes the fragment position to a render target. We use 16 bit integer targets to temporarily store these positions. Since the supported range of values is restricted to the interval *[0,1]*, the coordinates are mapped to this range before they are written to the render target. This operation uses the second additional instruction slot. Moreover, the number of available constant registers is decreased for both pixel shader versions because one constant register is needed to store the user-specified multiplication factor for the above mapping. Finally, one additional texture register is lost for vss.2.0 because ps.2.0 does not provide a readable rasterization position. Therefore, this position register is emulated by a vertex shader program that transfers the window coordinate position of each vertex to texture coordinates. Corresponding positions are obtained for each fragment by linear interpolation during scan conversion. So far, the implementation of the simulator supports only a limited number of possible render target formats, and no multiple render targets at all.

5. BENEFITS OF VRAGMENTS

An extension to graphics hardware definitely stands or falls with its usefulness for programs running on this hardware. A change of the structure of the underlying hardware is only justified if the benefits of the additionally possible applications excel the cost of chip design. Therefore, the simulator presented in this paper is an important contribution that helps to judge the usefulness of our extension before any effort is spent on chip design.
This section describes and discusses different scenarios and algorithms in which this extension is valuable. This collection of algorithms is by no means exhaustive, but rather presents a small glimpse the possibilities offered by our extension.

5.1. Binning algorithms

Collecting and counting algorithms, or binning algorithms, are a completely new class of algorithms that become possible. This class plays a fundamental role for many tasks in image analysis, image manipulation, and computer vision [Seul00]. Basically, all these algorithms take information given on an input texture and reorder this data into bins. Since the number of bins usually is smaller than the number of input texels, a compactification is achieved.

As an example, a histogram of gray values in an image can be computed by a vragment shader program. The image is represented by an input texture. A quadrilateral is rendered with the same size as the input texture, establishing a one-to-one mapping between generated fragments and input texture. Without the vragment extension, these fragments would be rendered at the same position as the input texels. A vragment program, however, allows us to move the fragment to another location that corresponds to the gray value of the image texel. For example, the *n* bins can be organized linearly in the first column of the output image. Here, the mapping takes the gray value $v \in [0,1]$ and yields the output position $(x,y) = (0, v \bullet n)$. The output color is set to a constant value that represents the integer value one for the output render target. A histogram counts the number of elements belonging to a bin. This counting operation is implemented by additive blending. The vragment shader for this task looks like this:

```
// shader begin
vss.2.0       // shader version 2.0
dcl_2d s0     // define sampler
dcl t0        // define texture coordinate
dcl vPos.xy   // declare fragment position
             // register
def c2, 0.0, 0.0, 0.0, 0.0 // set some
                            // constants
texld r0,t0,s0          // load texel
mul r1, r0.x, c0.x      // compute luminance
mad r1, r0.y, c0.y, r1  // of texel
mad r1, r0.z, c0.z, r1
```

```
mul r1, r1, c1.x  // map to corresponding
                  // bin
mov r1.yzw, c2.x  // zero y position
mov vPos, r1    // set new fragment position
mov r2, c1.y    // set counter increase
mov oC0, r2     // write counter increase
// shader end
```

A more sophisticated counting algorithm allows for the computation of the entropy of an image. The image entropy can be deduced from the entropy of a discrete random variable, which is defined as

$$H(x) = -E_x[\log P(x)]$$
$$= -\sum_{x_i \in \Omega_x} \log(P(X = x_i))P(X = x_i),$$

where *P(X)* can be derived from a previously computed histogram. Again, the summation over several inputs is supported by the vragment extension.

5.2. Displacements

Other techniques that become possible with freely relocatable fragments are displacement mapping [Cook84] and procedural texture synthesis for hypertexture [Per89].

For doing displacements a vragment shader simply maps each input fragment to a new output coordinate by adding a directional offset to its 3D world coordinate. Afterwards this new world coordinate is transformed back to the window coordinate system. The 3D coordinates of a fragment can be accessed for example by storing the 3D world coordinates at each vertex as texture coordinates.

The advantage compared to normal triangle-based displacement mapping approaches is: No time-consuming retesselation is required. The necessary triangle setup stays the same as no new triangles have to be sent down the pipeline. This also means that the necessary workload does no longer depend on the complexity of the displacement input map but rather on the projection of the displaced object.

But not only full 3D displacements are possible; also 2D displacements in window coordinates can be implemented or even a combination of both. For example, we used a combination of 2D and 3D displacements to simulate stains on paper. Figure 3(a) and 3(b) show the displacement of the 2D stain geometry which is rendered to a texture and then used again as input for a 3D displacement map on the paper surface.

Unfortunately besides these advantages there are also some drawbacks. The vragment approach works only well with 3D displacements that have a rather small magnitude otherwise the newly calculated distance between pixels will be too large and holes are starting to pop up. When trying to close these holes a certain overdraw has to be ensured on the input fragment side. At some point the expected cost of tessellation and vertex processing will be lower than the cost of overdrawing. But for displacements with small magnitudes, for example crumpled paper or objects covered by a thin snow layer, our approach works well.

5.3. Particle systems

Vragments could also be exploited for simulating and rendering a system of particles [Ree83]. Today, particles are managed and simulated by the CPU and transferred through the entire rendering pipeline. A vragment program offers the advantage of computing the motion of particles and the final rendering only within the fragment processing stage. Therefore, data transfer between CPU and GPU is obsolete, and the vertex processing is unburdened.

In a vragment approach, the particle system is represented by textures that store the current position and any other property of each particle required for simulation and rendering (e.g., color, age, mass, momentum). One part of the vragment program computes the animation by solving the equations of motion on a per-texel basis. This task does not require the vragment extension because each texel is mapped to the very same element of the texture representing the subsequent time step. In the second part, particles are rendered to screen. Rendering requires the extension to move the fragment to the current position of the particle, as previously computed by the simulation step.

5.4. Flow visualization

The study of flow fields plays a decisive role in many scientific disciplines, such as computational fluid dynamics (CFD) or meteorological and oceanographic simulation. An important class of visualization techniques computes the motion of massless particles transported along the velocity field to obtain characteristic structures like streamlines, pathlines, or streaklines. Often a dense representation by these lines is chosen. These dense representations are typically implemented by textures [San00]. Texture advection is a most popular way of implementing this approach on today's graphics hardware: Dense texture representations are moved along the velocity field of the flow. Due to the very nature of graphics hardware, the texture representation for a new time step is based on lookups in the texture of the previous time step. Stated differently, a backward mapping along the velocity field has to be applied. In some implementations, such as Lagrangian-Eulerian advection [Job00], this causes problems because input noise might be duplicated into several texels of subsequent time steps, leading to a degradation of image quality.

A vragment approach makes possible a forward mapping: Texels of the dense representation can actively move to their position at the following time step. This process can be regarded as a special case of a particle system in which the motion of massless particles is governed by the velocity of the flow. Forward mapping avoids texel duplication because a single texel is only mapped to another single texel. Typically, not all texels are touched by forward mapping; however, these gaps can be filled with new input noise by a second rendering pass.

5.5. Automatic LOD for fur and fur-like materials

Rendering fur can be very time-consuming, as a lot of individual primitives have to be sent down the pipeline and transformed and finally rendered. Therefore Level of Detail (LOD) algorithms for fur and other fur-like materials can release some of the strains lying on the graphics pipeline. One approach would be to use volume textures [Len01]. Although this approach can solve some of the problems, it unfortunately introduces new ones. It is well known that rendering volume textures forces a very high memory demand and a high rasterization workload on the graphics board. Using vragments could solve both problems and even provide an automatic LOD mechanism for free. The idea is closely related to the previously mentioned particle systems. As with particles the geometry of one single hair, brin, blade of grass, etc. is stored in a texture. The size of this texture defines the maximal detail of a single element.

While rendering, a furry surface contains a repeated pattern of this texture together with a second offset texture which displaces each individual element to its final position in 3D world coordinates. Whenever the viewer gets closer to the furry surface more vragments will automatically be spent for each individual hair because more texels from the hair primitive are read. In contrast, when the viewer moves away from the surface only a few vragments will be spent for each individual hair because less texels are read.

5.6. Arbitrary curves

Rendering curves, for example Bézier, Hermite, and arbitrary spline curves, is still an essential task in Computer Aided Design. In today's commercial CAD packages curves are either rendered in software or split into line segments and then rendered as individual lines. With the help of vragments the rendering of simple line primitives can be individually changed to rendering directly any arbitrary curve.

5.7. Special effects

An interesting class of applications for vragments could be special effects, in particular for games. Game applications do not have time to spend a lot of computation power on special effects. The goal is visual plausibility, not physical accuracy. In this case vragments are especially well suited for effects where the connectivity of surface is extremely complicated and time-varying. Examples for these class are explosions, implosion, arbitrary object deformations, cracks, object melting and flowing, etc. With vragments no connectivity is necessary as each vragment can be treated as an particle and can therefore individually be moved without influencing any neighboring points or primitive groups. If necessary, new neighboring information besides the automatically defined connectivity through the object definition can be built by grouping fragments with special textures. These are simply mapped onto the object surface. Additionally, it is still possible to use connected and fast renderable primitive groupings like triangle strips or triangle fans.

6. RESULTS AND OPEN ISSUES

Figures 2(a)-(c) and 3(a)-(c) show result pictures created for some algorithms mentioned in Section 5. Figure 3(a)-(c) are a combination of 2D and 3D displacements to simulate coffee or similar stains on a paper sheet. A circular 2D stain geometry is first rendered by using a 2D displacement shader to receive a more irregular shape. The resulting image is afterwards stored as a 3D heightfield texture and then used as input texture for displacing a 3D plane. Figure 2(a) shows a 3D displacement of a crumpled paper sheet. The height information in the heightfield is also taken for darkening the areas of the paper texture. Notice that the 3D plane consists of only one single quadrilateral. Figure 2(b) and (c) show an example of the automatic LOD rendering of a furry surface by using vragments. Figure 2(b) is a close top view on the surface while (c) is a more distant view. All the images where rendered using the hybrid mode of the simulator together with an ATI Radeon 9700 graphics board.

Although most of the algorithms we mentioned earlier are working nicely, there are still some more general open issues of the vragments extensions. One problem is the order of writing into the framebuffer because the blend/test unit supports both read and write from/to the framebuffer, if the order is changed the final result might also change as well. This problem does not show up in the simulator because the blending has to be done in software, but that might become a problem in pure hardware rendering. Another issue is the support for different point sizes. Should the vragment write module also handle different point sizes? Or should it only allow single pixels? Different point sizes might be very useful to solve some of the overdraw issues but would also increase the complexity of the unit, as now complete memory blocks might have to be accessed. This question is very closely related to the next problem of how to treat holes that can occur in some of the previously mentioned algorithms. Right now most of them are solved by overdrawing. A better idea would be an optional final linear-interpolation unit between different vragments to close holes between two individual vragments. The final question is how should multiple render targets be treated? Should every target have its own fragment relocation or should every target map to the same location? All these open questions should be subject of further research.

ACKNOWLEDGMENTS

In parts, this project has been supported by the "Landesstiftung Baden-Württemberg".

7. REFERENCES

[Bóo01] Bóo, M. , Amor, M. , Doggett, M. , Hirche, J. , and Strasser, W. , Hardware support for adaptive subdivison surface rendering. In Proceedings of EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware, pp. 33-40, 2001.

[Carr02] Carr, N. A. , Hall, J. D. , and Hart J.C. , The ray engine. In Proceedings of EUROGRAPHICS/SIGGRAPH Conference on Graphics Hardware. pp. 37-46, 2002.

[Cook84] Cook R. L. , Shade trees. In Proceedings of ACM SIGGRAPH. pp. 223-231, 1984.

[Cool02] McCool, M. D. , Qin, Z. , and Popa, T. S. , Shader metaprogramming. In Proceedings of EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware, pp. 57-68, 2002.

[Hei99] Heidrich, W. and Seidel, H.-P. , Realistic, hardware-accelerated shading and lighting. In Proceedings of ACM SIGGRAPH, pp. 171-178, 1999.

[Hil03] Hillesland, K. E. , Molinov, S. , and Grzeszczuk R. , Nonlinear optimization framework for image-based modeling on programmable graphics hardware. ACM Transaction on Graphics Vol. 22 (3), pp. 925-934, 2003.

[Job00] Jobard, B. , Erlebacher G. , and Hussaini, M. Y. , Hardware-accelerated texture advection for unsteady flow visualization. In Proc. IEEE Visualization' 00, pp. 155-162, 2000.

[Kes03] Kessenich, J. , Baldwin, D. , and Rost, R. , The OpenGL shading language version 1.05, 2003.

[Kirk98] Kirk, D. B. , Unsolved problems and opportunities for high-quality, high-performance 3D graphics on a PC platform. In Proceedings of the EUGROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware, pp. 11-13, 1998.

[Krü03] Krüger, J. and Westermann, R. , A framework for numerical simulation techniques on graphics hardware. ACM Transaction on Graphics Vol. 22 (3), pp. 908-916, 2003.

[Len01] Lengyel, J. , Praun, E. , Finkelstein, A. , and Hoppe H. , Real-time fur over arbitary surfaces. In Proceedings of the 2001 Symposium on Interactive 3D Graphics, pp. 227-232, 2001.

[Lin01] Lindholm, E. , Kligard, M. J. , and Moreton H. , A user-programmable vertex engine. In Proceedings of ACM SIGGRAPH, pp. 149-158, 2001.

[Mar03] Mark, W. R. , Glanville, S. , and Akeley, K., Cg: A system for programming graphics hardware in a C-like language. ACM Transaction on Graphics Vol. 22 (3), pp. 896-907, 2003.

[Mar01] Mark, W. R. and Proudfoot K. , The F-buffer: A rasterization-order fifo buffer for multipass rendering. In Proceedings of EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware, pp. 57-64, 2001.

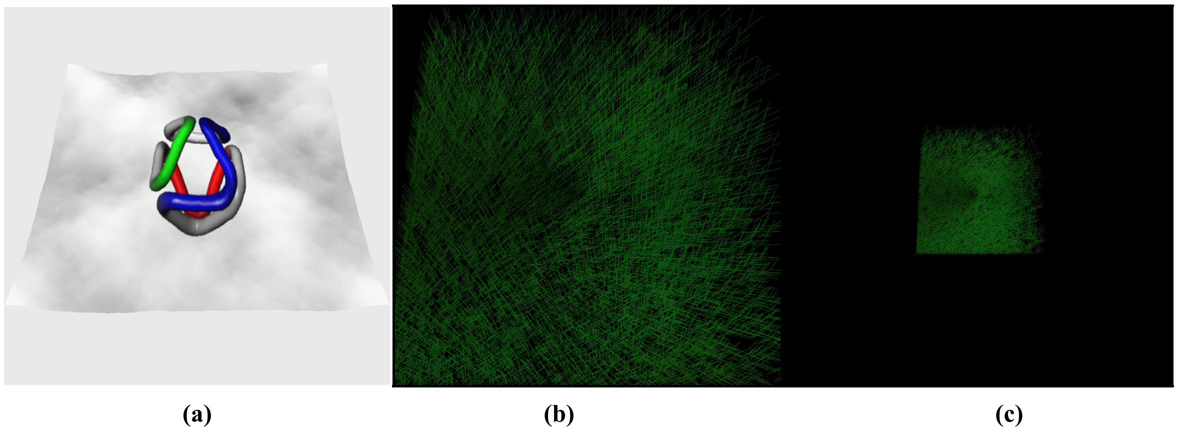

Figure 2. Example pictures for vragment shader usage: (a) A crumpled sheet of paper, (b) grassy surface in close-up view, (c) same grassy surface from far away.

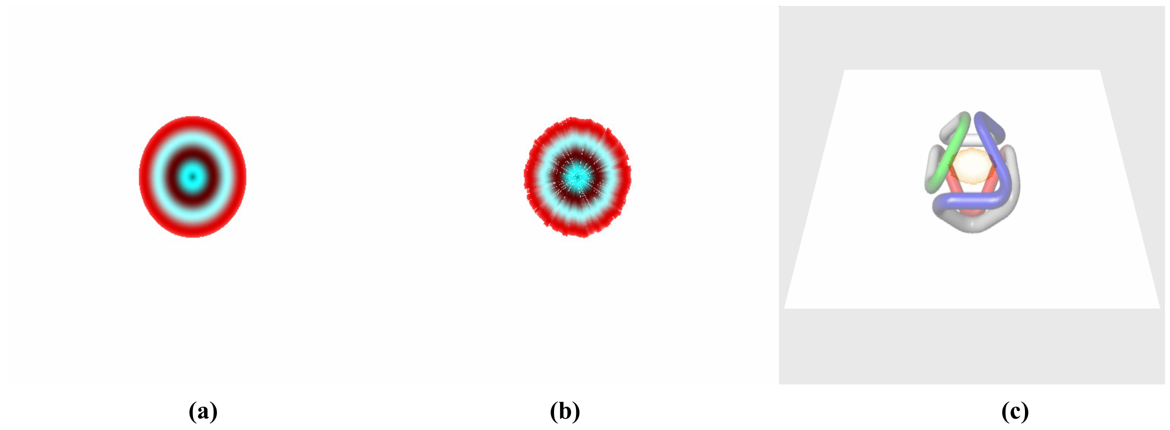

Figure 3. Stains on paper as an example of a combined 2D and 3D displacement: (a) The undisplaced 2D stain geometry, (b) the 2D displaced 2D stain geometry, and (c) the final result.

[MS02] Microsoft Corporation, DirectX 9 SDK documentation, 2002.

[Per89] Perlin, K. and Hoffert, E. M. , Hypertexture. In Proceedings of ACM SIGGRAPH, pp. 253-262, 1989.

[Pur02] Purcell, T. J. , Buck, I. , Mark, W. R. , and Hanrahan, P. , Ray tracing on programmable graphics hardware. In Proceedings of ACM SIGGRAPH, pp. 703-712, 2002.

[Ree83] Reeves, W. T. , Particle systems – A technique for modeling a class of fuzzy objects. Computer Graphics (Proc. ACM SIGGRAPH 83), 17(3), pp. 359-376, 1983.

[Rig03] Riguer, G. , Performance optimization techniques for ATI graphics hardware with DirectX 9.0. White paper, ATI, 2003.

[S398] S3 Incorporated, S3TC DirectX 6.0 standard texture compression, 1998.

[San00] Sanna, A. , Montrucchio, B. , and Montuschi, P. , A survey on visualization of vector fields by texture-based methods. Recent Res. Devel. Pattern Rec., 1, pp. 13-27, 2000.

[Seul00] Seul, M. , O'Gordman, L. , and Sammon, M. J. , Practical Algorithms for Image Analysis: Descriptions, Examples and Code. Cambridge Unversity Press. Cambridge. 1st edition, 2000.

[Spi01] Spitz, J. , GeForce 3 OpenGL performance. White paper, Nvidia, 2001.