Image Space Advection on Graphics Hardware

Markus Grabner*
Graz University of Technology
Graz, Austria
www.icg.tu-graz.ac.at

Robert S. Laramee†
VRVis Research Center
Vienna, Austria
www.VRVis.at

Abstract

The scientific visualization and computer graphics communities have witnessed a tremendous rise in graphics processing unit (GPU) related literature and methodology recently. This is due in part to the rapidly increasing processing speed offered by graphics cards. Parallel to this, we have seen several advances made in the area of texture-based flow visualization. We present a texture-based flow visualization technique, Image Space Advection (ISA), that takes advantage of the computing power offered by recent, state-of-the-art GPUs. We have implemented a completely GPU-based version of the ISA algorithm. Here we describe our implementation in detail, including both the advantages and disadvantages of implementing ISA on the GPU. The result is state-of-the-art technique that demonstrates the latest in terms of both flow visualization methodology and GPU programming.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism–Color, shading, shadowing, and texture; I.6.6 [Simulation and Modeling]: Simulation Output Analysis

Keywords: flow visualization, vector field visualization, graphics hardware, GPU programming, textures

1 Introduction

Recently, we have witnessed a rapid increase in the amount of research literature related to GPU-based computer graphics and visualization methodology. This is due, in part, to the rapidly increasing processing power offered by commodity graphics hardware. In general, the speed at which graphics hardware is developing currently exceeds that of the CPU. The fast increase in GPU processing speed along with the introduction of several new programming features has prompted many researchers to invest considerably more resources in this area than previously. A recent example in the area of flow visualization is the work of Stegmaier and Ertl [Stegmaier and Ertl 2004], who present a vortex detection algorithm implemented using GPU programming.

Parallel to the rise in GPU-based research literature, we have witnessed several advances in texture-based flow visualization methodology [Laramee et al. 2004a]. This holds true across each spatial dimension, 2D, 2.5D (surfaces), and 3D and for both steady-state and unsteady flow. With texture-based flow visualization, a texture is computed that is used to generate a dense representation of the flow. A notion of where the flow moves is incorporated through co-related texture values along the vector field. In most cases this effect is achieved through filtering of texture values according to the local flow vector. The property of texture-based methods that makes them attractive is their complete depiction of the vector field, as opposed to geometric methods like streamlines, which suffer from the seeding problem [Post et al. 2002].

What we present is a GPU-based version of a recent texture-based flow visualization algorithm for the visualization of unsteady flow on surfaces. The algorithm, called Image Space Advection (ISA [Laramee et al. 2004c]) synthesizes textures that depict unsteady fluid flow on surfaces (sometimes referred to as 2.5D) such as boundary surfaces and isosurfaces [Laramee et al. 2004b] at fast frame rates. The novelty of ISA is its speed, the fact that it applies to surfaces, and that it handles unsteady flow. There has been relatively little work done in this area, especially compared to the case of 2D flow [Laramee et al. 2004a].

The original ISA implementation is a mixture of both CPU and GPU-based computing. What we present here is a completely GPU-based implementation of ISA which takes advantage of the features of modern programmable graphics hardware. One of the goals of our work is to target Virtual Reality applications that impose real-time frame rate constraints. The main features of our algorithm implementation include:

- per-pixel calculation (in parallel) at interactive frame rates
- arbitrary view projection options, i.e, support for both orthographic and perspective projection
- no extra overhead due to changes to the view point or the case of unsteady flow

In other words, we eliminate the need for a distinction between the static case, which involves no changes to the view point and a steady-state flow field and the dynamic case (see Section 3.1). The dynamic case handles changes to the view point changed or unsteady flow [Laramee et al. 2004c]. Of course the GPU-based implementation presented here also inherits the original benefits of ISA including:

- the generation of a dense representation of unsteady flow on surfaces
- visualizes flow on complex surfaces composed of polygons whose number is on the order of 250,000 or more
- does not require a parameterization of the surface mesh or the surface mesh topology information
- supports user-interaction such as rotation, translation, and zooming while maintaining a constant high spatial resolution

We present the details of our implementation as well as the advantages and disadvantages of our GPU-based algorithm.

*e-mail: grabner@icg.tu-graz.ac.at
†e-mail: Laramee@VRVis.at

The rest of this paper is organized as follows: Section 2 presents related literature in the area of texture-based flow visualization with an emphasis on GPU-based implementations. Section 3 presents the details of our method starting with an overview in Section 3.1. Results are presented in Section 4 followed by conclusions and future work in Sections 5 and 6 respectively.

2 Related work

The large body of GPU-based literature in computer graphics and visualization is beyond the scope of this paper, as well as the large number of texture-based flow visualization methodology [Laramee et al. 2004a]. Here we summarize the texture-based flow visualization research that focuses on GPU programming.

Heidrich et al. [Heidrich et al. 1999] exploit pixel textures to accelerate LIC (line integral convolution [Cabral and Leedom 1993]) computation. Pixel textures are an OpenGL extension by SGI that provides the functionality of dependent textures in combination with multi-pass rendering. At this time, dependent textures were a brand new graphics card feature. Heidrich et al.'s implementation supports 2D, steady-state vector fields only and achieves subsecond computation times for LIC image generation.

Jobard et al. [Jobard et al. 2000] introduce a GPU-assisted texture advection technique for the dense visualization of 2D, unsteady flow. While the method of Max and Becker [Max and Becker 1995; Max and Becker 1999] advects textures based on coarse triangular meshes, Jobard et al. advect textures on a per-pixel basis by means of pixel textures, which are used in a similar way as by Heidrich et al [Heidrich et al. 1999]. The gray-scale texture from the previous time step is dragged along the flow field by modifying the texture coordinates for the dependent texture lookup according to the flow data. Nearest-neighbor sampling is combined with an update of fractional texture coordinates to represent subtexel motion and, at the same time, maintain a high contrast. An iterative injection of additional noise is used to compensate for a possible loss of contrast over time. Because of the limited functionality of the graphics hardware that supports pixel textures, the implementation requires many rendering passes and advects a texture of size 256^2 at approximately two frames per second. Moreover, the maximum resolution of textures is restricted to 256^2.

Weiskopf et al. [Weiskopf et al. 2001] present a GPU-accelerated version of the Lagrangian-Eulerian advection (LEA) algorithm using per-fragment operations. The GPU-based texture advection by Weiskopf et al. [Weiskopf et al. 2002] supports bilinear dependent texture lookups without taking into account the update of fractional coordinates. Therefore, this approach is mainly suitable for dye advection at high frame rates. Weiskopf et al. also demonstrate how GPU accelerated visualization of unsteady, 3D flows can be implemented with pixel textures.

Weiskopf et al. [Weiskopf et al. 2003] introduce a generic texture-based framework for visualizing 2D, time-dependent vector fields. They propose Unsteady Flow Advection-Convolution (UFAC) as an application of the framework for visualizing unsteady fluid flow. Also, their approach can reproduce other techniques such as LEA [Jobard et al. 2002], IBFV [van Wijk 2002], UFLIC [Shen and Kao 1997], and DLIC [Sundquist 2003]. Weiskopf et al. describe a GPU-accelerated implementation that, among other things, allows the user to trade-off quality for speed.

Until now, we have only mentioned previous work in 2D. The amount of related literature in 2.5D is small, especially in the area of texture-based flow visualization. In fact, the only other related literature we are aware of that addresses the combination of GPU programming and texture-based flow visualization in 2.5D is the work of Weiskopf and Ertl [Weiskopf and Ertl 2004]. Their method is a hybrid approach which combines properties from both object space and image space computation. The aim of their work differs from ours in that they focus more conceptually on the goal of assigning different aspects of the algorithm to object space and image space, attempting to maximize the advantages offered by both approaches, and focus less on the GPU-implementation. They also propose a color scheme targeted at effectively combining surface shape perception and flow visualization. Here we stay more inline with the original image space approach and focus on presenting the details of the GPU implementation such that other developers and researchers may reproduce the work, thus taking full advantage of our implementation.

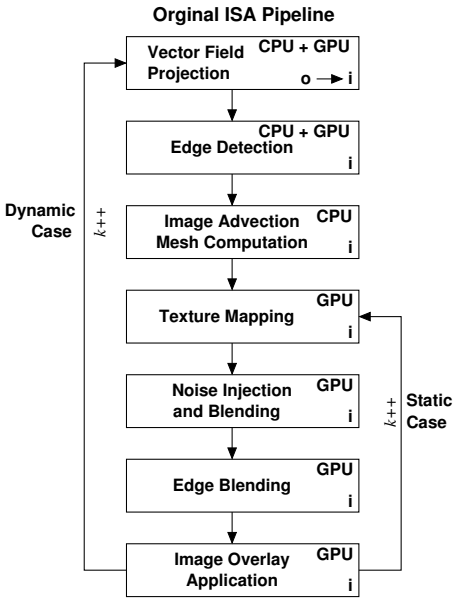


Figure 1: The major components of the original ISA implementation. Each component is indicated with **CPU** if it was originally implemented on the CPU and likewise with the GPU. Repeating the steps for the next frame is indicated by *k*++ (in the notation of the programming language C).

3 ISA on the GPU

We start with an overview of our implementation beginning with an examination of the original implementation before going into the details of the GPU-based three pass algorithm.

3.1 Method Overview

We compare our GPU-based implementation with the original ISA pipeline shown in Figure 1. Each stage of the pipeline is labeled with a **CPU** if it was originally implemented on the CPU, likewise with the GPU. If a stage of the original implementation used a combination of both the CPU and GPU this is also indicated (**CPU + GPU**). Each stage is also indicated with an **i** or **o** if it was originally implemented in image space or object space respectively. If there's a transition between the two spaces within a stage, this is also indicated. *k* represents time as a frame number during animation. We note that the original implementation exploited no explicit GPU-programming although some stages are labeled with GPU. The original implementation utilized OpenGL 1.1 only, not all of which is necessarily implemented in hardware.

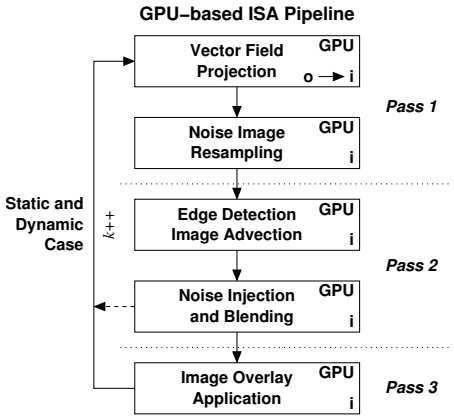


Figure 2: The major components of the GPU-based ISA implementation. While the output of the noise injection and blending stage is used as input for the next frame (dashed arrow), the output of the final stage is only displayed to the user (see Section 3.5).

The first difference to note is that the GPU implementation makes no distinction between the dynamic case and static cases. We employ a three-pass rendering algorithm, where intermediate results are stored in texture memory on the graphics board. The input is an arbitrary triangle (or quad) mesh with flow vectors defined at each vertex. Flow vectors are treated by the GPU in the same way as texture coordinates or colors, i.e., interpolated values are calculated at each pixel by the (parallel) rasterizer stage(s). The convolution is computed for a fixed number of steps (five turned out to be a reasonable number) since the loop (Figure 8, lines 12 to 21) has to be unrolled by the compiler on the platform we were using.

More details on the individual stages are given in the remainder of this section. At each stage, both a vertex program and a fragment program are executed. The corresponding Cg code is shown throughout the discussion of the method.

3.2 Data flow

Table 1 illustrates the inputs and outputs of the vertex and fragment programs at the three stages. For brevity, only *varying inputs* and *uniform sampler inputs* (i.e., textures) are shown (see [NVidia 2004] for an explanation of Cg program input and output types). Other uniform inputs, which are used to pass transformation matrices and configuration parameters to the Cg programs, are not shown. Moreover, vertex positions and normal vectors have also been omitted from Table 1 since they are treated in exactly the same way as in the standard graphics pipeline.

Hardware registers are identified by their associated *binding semantics* [NVidia 2004] `TEXCOORD0` (texture coordinates of first texture unit), `COLOR0` (diffuse color), and `COLOR` (color to be written to frame buffer). Textures are implicitly accessed in order (starting at the first texture unit), therefore the OpenGL constants `GL_TEXTURE0` and `GL_TEXTURE1` are used to specify the first and second texture unit, respectively.

Note that Cg (and the underlying graphics hardware) is flexible enough to assign quantities different from the original meaning to hardware registers (e.g., the flow direction is given as texture coordinates and then stored in the red/green components of a texture image). The remaining entries in Table 1 are discussed in sections 3.4 to 3.6.

```
1  struct ApplicationVertex
2  {
3    float4 position:  POSITION;
4    float4 normal:    NORMAL;
5    float4 texCoord0: TEXCOORD0;
6  };
7
8  struct VertexFragment
9  {
10   float4 position:  POSITION;
11   float4 wpos:      WPOS;
12   float4 color0:    COLOR0;
13   float4 texCoord0: TEXCOORD0;
14 };
```

Figure 3: Common data structures for our Cg programs.

3.3 Noise texture initialization

Similar to the approach by [Jobard et al. 2001], additive noise is defined by a function $n(x,y,k)$, where x and y are screen space coordinates, and k is the frame number since the start of the application. The noise range are the two integers $\{1,255\}$, which easily map to texture hardware (note that 0 is reserved as a background identifier, see Section 3.5). This function has the following properties:

- For a given pixel (x,y), the noise function should be piecewise constant over several frames to achieve higher spatial coherence.
- Noise covers the entire vector field over all frames k.
- Since noise can efficiently be injected by texture mapping, it is useful to precompute a set of *noise textures* and cycle through them periodically, i.e., $n(x,y,k+K) = n(x,y,k)$ for a given period size K.

The first criterion is easily achieved by keeping a large percentage of two consecutive noise images constant. To address the other two criteria, we select for each pixel an even number of random frames at which the pixel is inverted (i.e., $n(x,y,k+1) = 255 - n(x,y,k)$). This guarantees that the number of pixels that change between two consecutive noise images is roughly constant, which also holds for the transition from the last to the first image at the end of the period. Since each pixel is changed a constant number of times, noise is also well distributed in image space.

3.4 Pass 1: Projection and Resampling

We begin by defining two common data structures (Figure 3) that are shared by all Cg programs discussed throughout this section. In particular, the `VertexFragment` structure is used to connect the output of a vertex program to the input of the corresponding fragment program. See [NVidia 2004] for details on input and output parameters in Cg programs.

The vertex program at stage one is responsible for transforming the flow vectors into the screen coordinate system, creating what we call a *velocity image*. The velocity image encodes the flow vectors in the frame buffer. We choose to apply the formula

$$\mathbf{f}' = \mathbf{V}_k \mathbf{M}_k \mathbf{f}, \tag{1}$$

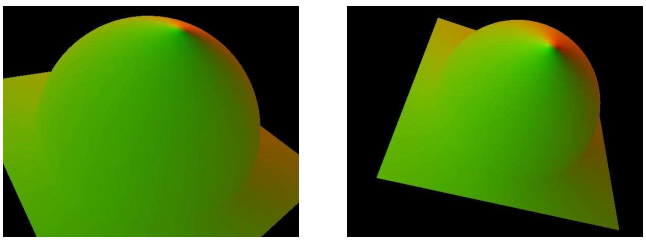
where $\mathbf{f}$ is the flow vector in object coordinates, and $\mathbf{M}_k$ and $\mathbf{V}_k$ represent the modeling and viewing transformations, respectively, at the current frame k. Flow visualization is performed according to the projected vector $\mathbf{f}'$. Since the flow vector $(f_x, f_y, f_z)^T$ needs to be interpreted as a direction vector, the application has to explicitly pass a 4-dimensional vector $(f_x, f_y, f_z, 0)^T$ (i.e., with the homogeneous coordinate w set to zero).

pass	VP input	VP output = FP input		textures								FP output			
	TEXCOORD0	TEXCOORD0	COLOR0	GL_TEXTURE0				GL_TEXTURE1				COLOR			
				r	g	b	a	r	g	b	a	r	g	b	a
1	flow	flow	–	–				–				flow		conv.	depth
2	–	–	–	flow		conv.	depth	noise			–	luminance			–
3	flow	flow	shading	color lookup			–	–				color			–

Table 1: Inputs and outputs of the vertex program (VP) and fragment program (FP) at the three stages, "conv." refers to the gray image used as input for the convolution computation (see section 3.2).

```
1  VertexFragment
2  main(ApplicationVertex av,
3       uniform float4x4 modelViewMatrix,
4       uniform float4x4 modelViewProjMatrix)
5  {
6    VertexFragment vf;
7    vf.position =
8      mul(modelViewProjMatrix, av.position);
9    vf.texCoord0 =
10     mul(modelViewMatrix, av.texCoord0);
11   return vf;
12 }
```

Figure 4: Cg code for pass 1 (vertex program): projecting geometry and flow vectors into screen space.



(a) flow field at frame $k-1$ (b) flow field at frame k

Figure 5: Velocity images: flow vectors are encoded in the red and green color channels for two successive frames.

Note that we intentionally omit the projection matrix $\mathbf{P}_k$, which is used to calculate screen coordinates

$$\mathbf{x}'_k = \mathbf{P}_k \mathbf{V}_k \mathbf{M}_k \mathbf{x} \tag{2}$$

from coordinates $\mathbf{x}$ in object space. Since we only consider static objects, the vertex location $\mathbf{x}$ in object coordinates does not depend on the frame index k. Matrix $\mathbf{P}$ is responsible for perspective foreshortening (unless parallel projection is used), equation (2), applied to the flow vectors, would therefore result in reduced perceived flow velocity with increasing distance of the object from the virtual camera. While this is physically correct, we get a better impression of the flow field if the velocity is independent from the distance to the view point, therefore we employ equation (1). However, ultimately the user may choose whether or not to use a perspectively shortened vector field. See Figure 5 for two example velocity images at two consecutive frames.

The task of the fragment program is to resample the image of the previous frame $k-1$ to the current frame k under possibly changed viewing parameters. In contrast to a previous approach using 3D noise functions [Weiskopf and Ertl 2004], the resampling procedure allows to maintain coherent images with 2D noise only. Since

```
1  float4
2  main(VertexFragment vf,
3       uniform float2 vpsize,
4       uniform float4x4 A,
5       uniform samplerRECT prev_img): COLOR
6  {
7    vf.wpos.w = 1;
8    float4 prev_hom = mul(A, vf.wpos);
9    float3 prev = prev_hom.xyz / prev_hom.w;
10   float lum;
11
12   if((prev.x < 0) || (prev.x > vpsize.x) ||
13      (prev.y < 0) || (prev.y > vpsize.y) ||
14      (prev.z < 0) || (prev.z > 1))
15     lum = 0;
16   else
17     lum = texRECT(prev_img, prev.xy).r;
18
19   return
20     float4(vf.texCoord0.xy, lum, vf.wpos.z);
21 }
```

Figure 6: Cg code for pass 1 (fragment program): resampling the velocity image from the previous to the current frame.

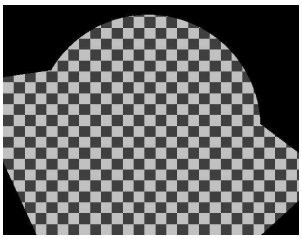
the homogeneous screen space coordinates $\mathbf{x}'_k = (x'_k, y'_k, z'_k, w'_k)^T$ at each fragment position $(x'_k, y'_k)^T$ are available to the fragment program, we can reconstruct the location of each currently processed fragment in world coordinates by applying the inverse projection. In turn we can also compute the screen space coordinates of the same fragment in the previous frame. We therefore can track pixels under modified viewpoint and viewing direction as follows:

$$\mathbf{x}'_{k-1} = \underbrace{(\mathbf{P}_{k-1}\mathbf{V}_{k-1}\mathbf{M}_{k-1})(\mathbf{P}_k\mathbf{V}_k\mathbf{M}_k)^{-1}}_{\mathbf{A}} \mathbf{x}'_k \tag{3}$$

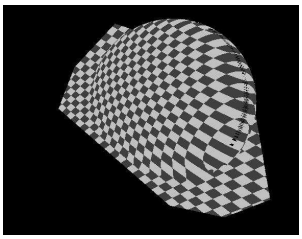
Matrix $\mathbf{A}$ is constant for each frame, it is therefore computed by the CPU and passed as a constant to the GPU (Figure 6 line 4).

We have to verify that the corresponding pixel in the previous frame is contained within the view-port. If it is not, we cannot use it for the convolution computation and therefore mark it as invalid (i.e., assign zero luminance, see Section 3.5 for more details).

The resampling process is illustrated in Figure 7. Figure 7(a) corresponds to the geometry of Figure 5(a). For illustrative purposes, we use a checkerboard pattern instead of the actual output at frame $k-1$. The pattern is resampled to the geometry of Figure 5(b), the results are shown in Figure 7(b). Note that regions of the mesh that have been outside the viewport in frame $k-1$ are rendered with the background color in frame k. Self-occlusions of the object are not accurately resolved since the frame buffer can only hold a single color value per pixel. However, this is not a problem since sufficient noise is injected to remove the resulting artefacts in short time, e.g., a fraction of a second.



(a) image at frame $k-1$



(b) image at frame $k-1$ resampled to frame k

Figure 7: Resampling the velocity image of the previous frame to the geometry of the current frame. Normally the camera does not move this much between frames, this is for illustration.

The final image encodes the screen space flow direction (red and green channel, Figure 5(b)), the luminance for the convolution computation (blue channel, Figure 7(b)), and the depth for detecting depth discontinuities (alpha channel, not shown). These values correspond to "flow/conv./depth" in Table 1.

Due to the limitation to four values in a texture image, the z-coordinate of the projected flow vector can not be stored at this stage. Therefore it has to be recalculated by the hardware in the third pass, where a color value corresponding to flow velocity is computed.

3.5 Pass 2: Image Space Advection

The vertex program at this stage performs the standard model-view-projection transformation as given in equation (2). The Cg code is not shown since it is basically a subset of the code in Figure 4.

The fragment program at pass two is the core of the advection implementation. Starting at the pixel coordinates generated by the rasterizer, a fixed number of pixels is visited according to the local flow direction. These computations are carried out in floating point accuracy, however, the noise textures are accessed by integer coordinates. It would be possible to perform bilinear interpolation at each convolution step, but this is a costly solution in terms of performance.

We empirically adjusted some parameters of the advection algorithm. As few a five iteration steps turned out to be sufficient to compensate for the effects of coordinate quantization and to provide a good result. Moreover, we are using a constant convolution kernel. More precisely, it is a box function $f(x)$ with $f(x) = 1$ for $x \in [0; N-1]$ and $f(x) = 0$ elsewhere. This is implied by the iteration algorithm performing N steps. We also tried an exponentially decreasing kernel, but didn't observe improvements. More sophisticated kernels (such as those proposed by [Cabral and Leedom 1993]) need to be precomputed and stored in a one-dimensional texture (similar to the color map approach explained in Section 3.6) since performance is critical at this stage.

Examples of texture advection and noise injection are displayed in Figure 9. The texture advection algorithm creates the image in Figure 9(a). Note that regions that have been outside the view-port in the previous frame are immediately replaced by the current noise texture (Figure 9(b)) instead of being blended with noise over several frames. The combination of both images is shown in Figure 9(c). The amount of injected noise is controlled by the parameter "`noise_blend`" (Figure 8).

```
1  float4
2  main(VertexFragment vf,
3       uniform float ztol,
4       uniform float noise_blend,
5       uniform samplerRECT flow,
6       uniform sampler2D noise): COLOR
7  {
8    float2 pos = vf.wpos.xy;
9    float zprev = texRECT(flow, pos).w;
10   float color = 0, denom = 0, real_blend;
11
12   for(int i = 0; i < 5; ++i) {
13     float4 flowval = texRECT(flow, pos);
14
15     if((abs(flowval.w - zprev) < ztol) &&
16        (flowval.b != 0)) {
17       color += flowval.b;
18       ++denom;
19       pos -= flowval.xy;
20     }
21   }
22
23   if(denom == 0) {
24     denom = 1;
25     real_blend = 1;
26   }
27   else
28     real_blend = noise_blend;
29
30   float n = tex2D(noise, vf.wpos.xy);
31   color = lerp(color / denom, n, real_blend);
32   return float4(color, color, color, 1);
33 }
```

Figure 8: Cg code for pass 2 (fragment program): image space advection is performed by the loop from lines 12 to 21, noise is injected in lines 30 and 31.

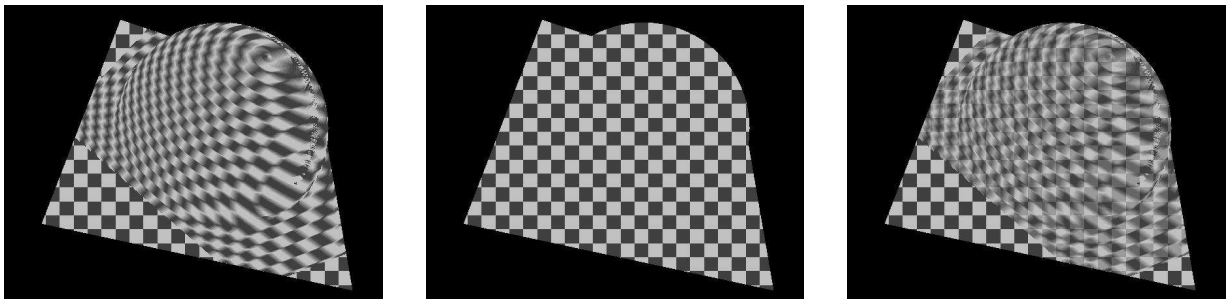
Another task performed at this stage is the detection of depth discontinuities. This can easily be done since depth values have been computed and stored in the alpha channel at pass one. This is important for two reasons:

- If the depth discontinuity is caused by one object (partly) occluding another, we want to prevent texture flow across object boundaries.
- If an object is displayed against the background, we do not want the background to "smear" into the interior of the object's projection.

The convolution computation can only use pixels corresponding to the screen-space projected surface in the previous frame. Since both background pixels and pixels outside of the view-port are assigned zero luminance, they can easily be detected and ignored. Since it is not possible to terminate a loop in a fragment program, we simply test the above conditions at each iteration and update the weighted sum only for continuous regions.

Finally, noise is injected into the image by blending the intensity found by convolution with a pixel from the noise image. The resulting image is used for two purposes:

- as the input of the resampling stage at the next frame
- as one contribution to the final image, that is improved in the rendering pass that follows



(a) advection applied to Figure 7(b) (b) noise texture at frame k (c) noise added to advected image

Figure 9: Image space advection and noise injection: (a) advection is performed on the resampled image, (b) the noise image (replaced by a checkerboard pattern) is extracted from texture memory, (c) both images are superimposed according to a user-defined ratio.

```
1  VertexFragment
2  main(ApplicationVertex av,
3       uniform float4x4 modelViewMatrix,
4       uniform float4x4 modelViewProjMatrix,
5       uniform float shading_blend,
6       uniform float4x4 modelViewIT)
7  {
8    VertexFragment vf;
9    vf.position =
10     mul(modelViewProjMatrix, av.position);
11   vf.texCoord0 =
12     mul(modelViewMatrix, av.texCoord0);
13   float3 normal =
14     mul(modelViewIT, av.normal).xyz;
15   normal = normalize(normal);
16   float c =
17     lerp(1, abs(normal.z), shading_blend);
18   vf.color0 = float4(c, c, c, 1);
19   return vf;
20 }
```

Figure 10: Cg code for pass 3 (vertex program): recomputation of flow vector projection and evaluation of GOURAUD illumination model.

3.6 Pass 3: Shading

So far we have only dealt with monochrome images under ambient illumination. To add realism to the scene, the third pass performs lighting operations and applies a color-mapping corresponding to velocity magnitude.

As mentioned above, the vertex program at this stage recalculates the transformed flow vectors in the same way as in the first pass. Moreover, a simple lighting model with LAMBERT illumination is applied. If the modelview transformation is described by the matrix $\mathbf{MV}$, the normal vectors $\mathbf{n}$ need to be transformed according to

$$\mathbf{n}' = ((\mathbf{MV})^{-1})^T$$

to obtain correct illumination of the transformed object. Under the assumption of directional light parallel to the viewing direction, the z-coordinate of the transformed normal vector $\mathbf{n}'$ is proportional to the intensity of the reflected light. For correct illumination of backfacing polygons we take the absolute value of this quantity. The user can select the desired ratio between ambient and diffuse reflection by the parameter "`shading_blend`" (see Figure 10). The result of these computations is shown in Figure 12(a).

```
1  float4
2  main(VertexFragment vf,
3       uniform float lookup_offset,
4       uniform float lookup_scale,
5       uniform float color_blend,
6       uniform samplerRECT lookup): COLOR
7  {
8    float len = length(vf.texCoord0.xyz);
9    float arg = (len + lookup_offset) *
10                lookup_scale;
11   return vf.color0 *
12     lerp(float4(1, 1, 1, 1),
13          texRECT(lookup, float2(arg, 0)),
14          color_blend);
15 }
```

Figure 11: Cg code for pass 3 (fragment program): color mapping according to flow velocity magnitude is applied to the GOURAUD-shaded image.

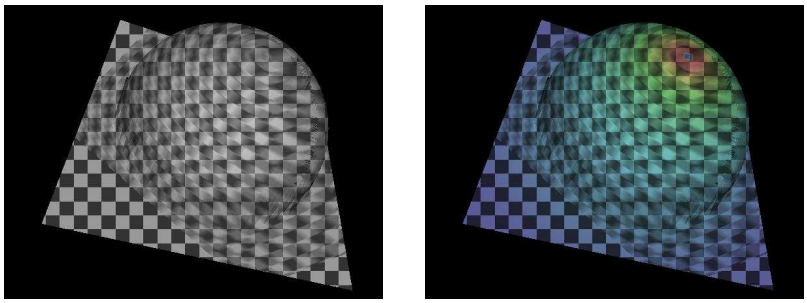
In the fragment program, the magnitude of the flow velocity is calculated at each pixel and used as an index into a color lookup table (i.e., a 256×1 texture image). Once again, the ratio between this color and the shaded image can be selected by the user (parameter "`color_blend`"). Figure 11 shows the corresponding code.

The output of pass 3 is not directly stored in the framebuffer, but combined with its previous content (which is the result of pass 2) according to the OpenGL blending mode `GL_DST_COLOR`, which is a component-wise multiplication of the three color channels. The final image resulting from this operation is presented in Figure 12(b).

4 Results

We tested our system on a number of different machines to evaluate the impact of hardware components (CPU and GPU) to overall performance (see Table 2). We observe large differences between the two GPU models in our experiments. Moreover, even the same GPU model gives different results on different platforms. We do not have a detailed explanation for this behaviour since it involves internals of the proprietary graphics driver as well as bus system implementation. It reflects the rate at which the CPU can provide data to the GPU.

The influence of different numbers of convolution steps is shown in Figure 14. Figure 14(a) is not satisfactory, quantization arte-



(a) LAMBERT light model applied (b) color overlay applied

Figure 12: Shading and color overlay applied to the output of the advection and noise injection stage (Figure 9(c)).

method	CPU (Intel/AMD)	GPU (NVidia)	main memory	graphics memory	frames/second hemisphere (2.5kQ)	ring (10.6kT)	cooling jacket (228kT)
ISA-GPU	Pentium III 600MHz	GeForce FX 5900XT	512MB	128MB	30	16	1
ISA-GPU	Pentium IV 2.4GHz	GeForce FX 5900XT	1GB	128MB	46	28	3.3
ISA-GPU	Athlon 64 1.8GHz	GeForce 6800	1GB	128MB	115	65	3.5
ISA-GPU	Pentium IV 3GHz	GeForce 6800	1GB	128MB	191	112	7.7
IBFVS	2×Pentium IV 2.8GHz	980 XGL Quadro	1GB	128MB	n/a	49	2.7
ISA	2×Pentium IV 2.8GHz	980 XGL Quadro	1GB	128MB	n/a	2.7	3.1

Table 2: Frame rates achieved with our method (ISA-GPU) on several hardware platforms for different models at an image size of 512^2 pixels, and frame rates for the IBFVS and ISA (512^2 advection mesh size) methods as reported by [Laramee et al. 2004c]. The primitive count (kT for 1000 triangles, kQ for 1000 quadrilaterals) of each model is given in parentheses. The hemisphere has been used throughout Section 3, see Figures 13 and 14 for the cooling jacket and ring data sets, respectively.

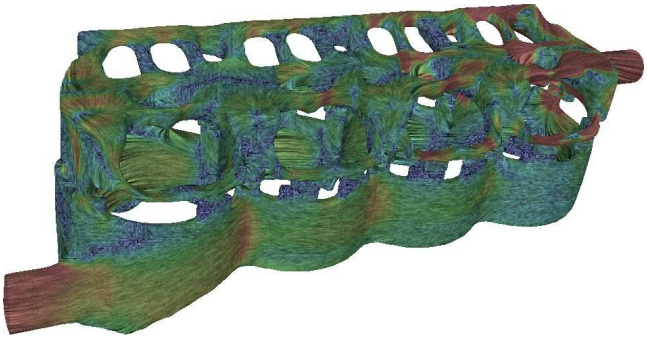


Figure 13: Visualization of flow at the complex boundary surface of a cooling jacket. Color is mapped to velocity magnitude.

facts are clearly visible. While one might prefer static images with 10 or more convolution steps (Figures 14(c) and 14(d)), those tend to overly blur the injected noise for animation purposes. Best animation results were found with 5 convolution steps (Figure 14(b)), which was also used for the accompanying video (`http://www.VRVis.at/scivis/laramee/ISAonGPU`).

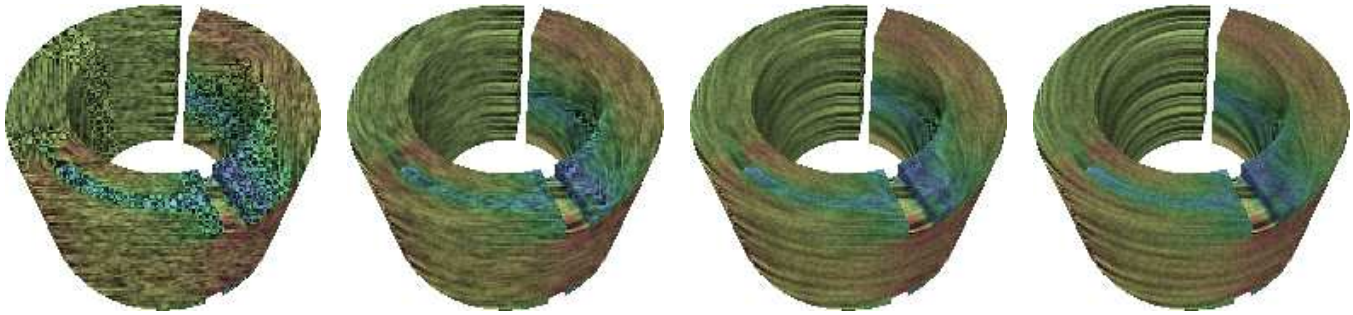
5 Conclusions

The experiments reported in Section 4 demonstrate the high efficiency of the GPU-based Image Space Advection algorithm. It can improve previous results [Laramee et al. 2004c] by a factor of up to 40 (comparing ISA-GPU and ISA on the ring data set, see Table 2). These results have been achieved with arbitrary perspective projection without any coherence constraints between successive frames, i.e., the user can freely explore the scene with a 3D navigation system. The main limitation of the current approach is that it does not take into account occlusions, i.e., advection is performed for every fragment, regardless if it will ultimately become visible or occluded by other fragments. This is a sub-optimal use of resources, in particular for objects with high depth complexity, i.e., many (self-) occlusions, such as the cooling jacket.

6 Future work

The NVidia GeForce 6800 graphics card became available to us only at the end of this work, we therefore didn't yet make use of its advanced features to further improve performance. We expect a significant performance boost by using multiple render targets to overcome the depth complexity problem. Moreover, utilizing floating point textures will resolve some issues related to accuracy and lead to a cleaner implementation.

The models we have been working with exhibit large regions of relatively constant flow direction and some smaller more turbulent areas. The presence of information at different scales lends itself to a multiresolution approach, where unnecessary (i.e., redundant) details are replaced by a simplified representation under controlled approximation error. Existing multiresolution methods (such as quadric error metrics [Garland and Heckbert 1997]) can be extended to incorporate flow direction into the set of simplification criteria.



(a) 2 steps (b) 5 steps (c) 10 steps (d) 20 steps

Figure 14: Flow at the boundary surface of the ring data set visualized using different numbers of convolution steps. Low resolution images were intentionally selected for better observation of the resulting artefacts.

Our partners in automotive engineering have confirmed the usefulness of Virtual Reality (VR) applications in a previous project involving static scalar fields [Grabner et al. 2004]. We plan to integrate the present method for visualization of dynamic vector fields into the existing VR prototype.

7 Acknowledgments

The authors thank all those who have contributed to this research including AVL (`www.avl.com`), the Austrian national research program called Kplus (`www.kplus.at`), and the VRVis Research Center (`www.VRVis.at`). CFD simulation data courtesy of AVL.

References

CABRAL, B., AND LEEDOM, L. C. 1993. Imaging Vector Fields Using Line Integral Convolution. In *Poceedings of ACM SIGGRAPH 1993*, ACM Press / ACM SIGGRAPH, Annual Conference Series, 263–272.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Conference Proceedings*, Addison Wesley, T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, 209–216. ISBN 0-89791-896-7.

GRABNER, M., BORNIK, A., SCHMIDT, S., REITINGER, B., SCHLÖGL, O., AND GARRIDO, L. 2004. Exploration of CFD data in a Virtual Reality setup. In *Proceedings Virtual Product Development in Automotive Engineering*.

HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. 1999. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. In *ACM Symposium on Interactive 3D Graphics*, 127–134.

JOBARD, B., ERLEBACHER, G., AND HUSSAINI, M. Y. 2000. Hardware-Accelerated Texture Advection. In *Proceedings IEEE Visualization 2000*, IEEE Computer Society, 155–162.

JOBARD, B., ERLEBACHER, G., AND HUSSAINI, M. Y. 2001. Lagrangian-Eulerian Advection for Unsteady Flow Visualization. In *Proceedings IEEE Visualization '01*, IEEE.

JOBARD, B., ERLEBACHER, G., AND HUSSAINI, Y. 2002. Lagrangian-Eulerian Advection of Noise and Dye Textures for Unsteady Flow Visualization. *IEEE Transactions on Visualization and Computer Graphics 8(3)*, 211–222.

LARAMEE, R. S., HAUSER, H., DOLEISCH, H., POST, F. H., VROLIJK, B., AND WEISKOPF, D. 2004. The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. *Computer Graphics Forum 23*, 2 (June), 203–221.

LARAMEE, R. S., SCHNEIDER, J., AND HAUSER, H. 2004. Texture-Based Flow Visualization on Isosurfaces from Computational Fluid Dynamics. In *Data Visualization, The Joint Eurographics-IEEE TVCG Symposium on Visualization (VisSym '04)*, Eurographics Association, 85–90,342.

LARAMEE, R. S., VAN WIJK, J. J., JOBARD, B., AND HAUSER, H. 2004. ISA and IBFVS: Image Space Based Visualization of Flow on Surfaces. *IEEE Transactions on Visualization and Computer Graphics 10*, 6 (Nov.), 637–648.

MAX, N., AND BECKER, B. 1995. Flow Visualization Using Moving Textures. In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data*, 77–87.

MAX, N., AND BECKER, B. 1999. Flow Visualization Using Moving Textures. In *Data Visualization Techniques*, 99–105.

NVIDIA. 2004. *Cg Toolkit User's Manual - A Developer's Guide to Programmable Graphics*. NVidia corporation, Jan.

POST, F. H., VROLIJK, B., HAUSER, H., LARAMEE, R. S., AND DOLEISCH, H. 2002. Feature Extraction and Visualization of Flow Fields. In *Eurographics 2002 State-of-the-Art Reports*, The Eurographics Association, 69–100.

SHEN, H. W., AND KAO, D. L. 1997. UFLIC: A Line Integral Convolution Algorithm for Visualizing Unsteady Flows. In *Proceedings IEEE Visualization '97*, IEEE Computer Society, 317–323.

STEGMAIER, S., AND ERTL, T. 2004. A Graphics Hardware-based Vortex Detection and Visualization System. In *Proceedings IEEE Visualization 2004*, IEEE, 195–202.

SUNDQUIST, A. 2003. Dynamic Line Iintegral Convolution for Visualizing Streamline Evolution. In *IEEE Transactions on Visualization and Computer Graphics*, vol. 9(3), 273–282.

VAN WIJK, J. J. 2002. Image Based Flow Visualization. *ACM Transactions on Graphics 21*, 3, 745–754.

WEISKOPF, D., AND ERTL, T. 2004. A Hybrid Physical/Device-Space Approach for Spatio-Temporally Coherent Interactive Texture Advection on Curved Surfaces. In *Proceedings of Graphics Interface*, 263–270.

WEISKOPF, D., HOPF, M., AND ERTL, T. 2001. Hardware-Accelerated Visualization of Time-Varying 2D and 3D Vector Fields by Texture Advection via Programmable Per-Pixel Operations. In *Proceedings of the Vision Modeling and Visualization Conference 2001 (VMV 01)*, 439–446.

WEISKOPF, D., ERLEBACHER, G., HOPF, M., AND ERTL, T. 2002. Hardware-Accelerated Lagrangian-Eulerian Texture Advection for 2D Flow Visualizations. In *Proceedings of the Vision Modeling and Visualization Conference 2002 (VMV-01)*, 439–446.

WEISKOPF, D., ERLEBACHER, G., AND ERTL, T. 2003. A Texture-Based Framework for Spacetime-Coherent Visualization of Time-Dependent Vector Fields. In *Proceedings IEEE Visualization '03*, IEEE Computer Society, 107–114.