Volume Graphics (2006)
T. Möller, R. Machiraju, T. Ertl, M. Chen (Editors)

GPU-assisted Multi-field Video Volume Visualization

Ralf P. Botchen[1], Min Chen[2], Daniel Weiskopf [3], and Thomas Ertl[1]

[1]Visualization and Interactive Systems, University of Stuttgart, Germany, {botchen|ertl}@vis.uni-stuttgart.de
[2]Department of Computer Science, University of Wales Swansea, UK, m.chen@swansea.ac.uk
[3]Graphics, Visualization, and Usability Lab (GrUVi), Simon Fraser University, Canada, weiskopf@cs.sfu.ca

Abstract

GPU-assisted multi-field rendering provides a means of generating effective video volume visualization that can convey both the objects in a spatiotemporal domain as well as the motion status of these objects. In this paper, we present a technical framework that enables combined volume and flow visualization of a video to be synthesized using GPU-based techniques. A bricking-based volume rendering method is deployed for handling large video datasets in a scalable manner, which is particularly useful for synthesizing a dynamic visualization of a video stream. We have implemented a number of image processing filters, and in particular, we employ an optical flow filter for estimating motion flows in a video. We have devised mechanisms for combining volume objects in a scalar field with glyph and streamline geometry from an optical flow. We demonstrate the effectiveness of our approach with example visualizations constructed from two benchmarking problems in computer vision.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture / Image Generation I.3.6 [Computer Graphics]: Methodology and Techniques I.3.m [Computer Graphics]: Video Visualization

1. Introduction

Viewing videos is a time-consuming and resource-intensive process. Even viewing in the fast-forward mode, it takes about 4 hours to watch a day's recording by a CCTV (closed-circuit TV) camera. In other words, if the footage of all estimated 25 million CCTV cameras in operation worldwide were to be examined, we would need 10 million people employed to just watch videos, in the fast-forward mode, for 10 hours every day. Video visualization is a computation process that extracts meaningful information from original video datasets and conveys the extracted information to users by appropriate visual representations. Although this technology is very much in its infancy, its potential benefits in terms of time and resource saving cannot be overestimated.

Video data can be considered as 3D volume data, with one temporal and two spatial dimensions. One major difficulty is that the geometrical objects in each video frame are the projective representations of the original 3D spatial objects. Thus, a video volume is a 3D projection of a 4D spatiotemporal description of a moving scene. Because the third dimension of a video volume is the temporal dimension, simply visualizing a video volume using traditional volume rendering techniques is often inadequate in terms of extracting and conveying the most meaningful information in a video. For example, consider the video clip 'LeftBox' (Figure 1), which is one of the benchmarking problems collected by the CAVIAR project [Fis04]. Figure 2(a) shows a visualization of the video volume using a technique similar to [DC03]. Although the visualization adequately represents the objects extracted from the background scene, it does not provide sufficient motion features to allow the user to recognize that a moving object (i.e., a person) left a stationary object (i.e., a box) in the scene.

Although it is possible to estimate and visualize the optical flow in a video as shown in Figure 2(b), the motion on its own cannot adequately convey the presence of objects in the scene. These observations indicate that the combined use of a volumetric scalar field (for the video data) and a vector field (describing the motion) might result in an effective video visualization. We thereby face the issue of multi-field visualization of 3D scalar and vector fields.

The combined visualization shown in Figure 2(c) separates four stages of the video. In stage one, the person enters the scene with a box, i.e., the person is moving. In stage two, the person stops to deposit the box on the floor. This fact is clearly conveyed through a lack of flow glyphs. In the

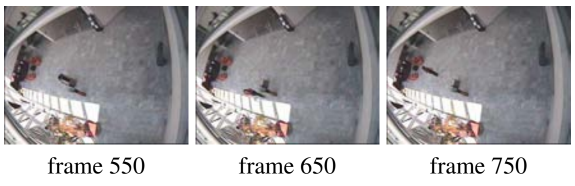


Figure 1: *Frames selected from the video clip 'LeftBox'. A woman deposits a box in the scene and leaves. The frames relate to stages 2, 3, and 4 in Figure 2.*

next stage, the person moves around the box. In stage four, the person exits the scene, but leaves the motionless box on the floor. The combination of volume and flow visualization gives the viewer a better understanding of both the information on location and motion of objects.

Due to the need for the simultaneous visualization of two datasets for scalar and vector information, the challenge of handling large 3D data is more pronounced than in traditional volume rendering. In general, it is necessary to handle a large amount of data in many applications of video visualization. For example, in scientific experiments that involve a high-speed camera, an experiment of a few seconds could result in a video of thousands of frames. For processing video archives in applications such as video segmentation and geo-biological monitoring, one may need to create a visual representation for a video of hours, days, or sometimes even longer periods of time. Finally, in many surveillance-related applications, one needs to handle large real-time video streams.

In this paper, we address technical problems associated with the fast rendering of large video data as a volume. In particular, our objective is to generate an effective multi-field visualization by combining volumetric scalar and vector data in order to extract and convey the most meaningful information in a video. Our strategy is to use the capabilities of modern GPUs (graphics processing units) to synthesize interactive multi-field visualization (see Section 5). A complementary strategy is to design a scalable rendering method for video datasets of varying size. We employ bricking techniques to overcome the difficulties of accommodating large video streams in GPU memory (see Section 6). The basic software architecture of our visualization system is discussed in Section 3, the preprocessing stages in Section 4.

Part of the work described in this paper was used to support a major user study on visual signatures in video visualization [CBH*06]. In this paper, we focus on the technical and algorithmic development of a system, called VVR (*Video Volume Renderer*), which provides interactive rendering of video volumes and extracts visual signatures for analysis. Technically, VVR represents a major leap from previous video volume visualization in terms of rendering speed, visualization features, and the scalability of data size.

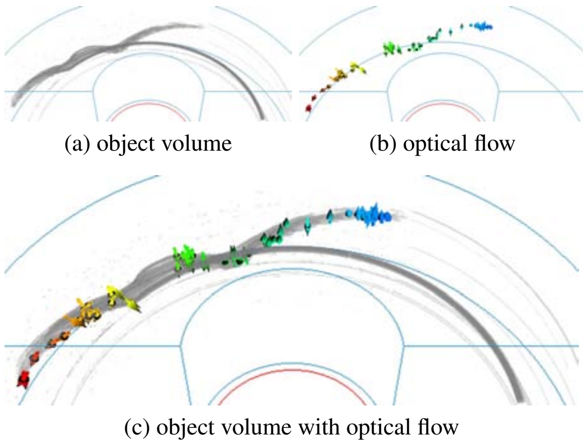


Figure 2: *Volume visualization of extracted objects in a video in (a) and flow visualization of an estimated optical flow of the same dataset in (b). Image (c) shows a combination of both visualizations.*

2. Related Work

Video visualization was introduced by Daniel and Chen [DC03] as a means of processing large volumes of video data. We adopt their idea of a horseshoe layout for video rendering because the horseshoe geometry has a number of merits, including a cost-effective space utilization and a provision of four visible sides of a video volume. However, the horseshoe layout requires the rendering of a deformed video volume. A generic way of rendering deformed volumes is to use ray casting (e.g., ray reflectors [KY97]). The original implementation of horseshoe volume rendering is based on a related implementation by CPU ray casting, which is not interactive [DC03]. In this paper, we use 3D texture slicing for interactive volume rendering [CCF94]. We adopt the approach by Rezk-Salama *et al.* [RSSSG01], which utilizes texture slicing to render deformed volumes in real time. In their approach, a backward mapping is employed to modify the texture coordinates that address the dataset.

One part of our approach is to include the visualization of the optical flow in the video volume visualization. General flow visualization methods have a long tradition in scientific visualization [WE05]. There exist several different strategies to display a vector field associated with a flow. One visual representation used in this work relies on glyphs to show the direction of a vector field at a collection of sample positions, e.g., by employing arrows or hedgehogs to visually encode direction [KH91, Dov95]. Another visual representation relies on the characteristic lines, such as streamlines, obtained by particle tracing. A major problem of 3D flow visualization is the potential loss of visual information due to mutual occlusion. This problem can be addressed by improving the perception of streamline structures [IG98] or by appropriate seeding [GGS02].

3. VVR System Architecture

The flow chart in Figure 3 shows the overall system architecture of VVR, which includes two major functional subsystems, namely *video processing* and *video rendering*. The video processing sub-system consists of a collection of filters for generating a variety of 3D scalar fields and vector fields that highlight different features of a video. Many of these filters designed for volume rendering are discussed in [CBH*05]. In the following section, we will concentrate on the computation of a flow field from a video volume, and volumetric seeds for flow geometry.

The video rendering sub-system is the main focus of this paper. We adapt volume bricking to handle large volume and flow datasets. One modification is that we partition data only in the temporal dimension instead of the spatial partitioning commonly used in traditional volume rendering. As shown in Figure 3, the bricking process affects most modules in the rendering sub-system through a loop that triggers a dynamic update within each module. Because of the existence of this loop and the logical brick structure, our bricking mechanism supports scalable multi-field visualization, including video spans, glyph geometry for flow visualization, and dynamic streamlines. The rendering framework will be detailed in Section 5 and the bricking strategy in Section 6.

4. Video Processing

4.1. Optical Flow

One ingredient of our approach is the optical flow of the video. To compute the optical flow, we adopt a gradient-based differential method [HS81]. Our implementation is based on a modified version of the gradient-based differential method [BFB94].

Let us consider an image sequence as an intensity function $I(\mathbf{p},t)$, where $\mathbf{p}=(x,y)$ is a position on an object in motion, and t is the time variable. The translation of $\mathbf{p}$ with velocity $\mathbf{v}=(\mathrm{d}x/\mathrm{d}t, \mathrm{d}y/\mathrm{d}t)=(u,v)$ is thus:

$$I(\mathbf{p},t) = I(\mathbf{p}-\mathbf{v}t, 0)\ .$$

A Taylor expansion of the above expression results in

$$I_x(\mathbf{p},t)u + I_y(\mathbf{p},t)v + I_t(\mathbf{p},t) = 0\ ,$$

where I_x, I_y, and I_t are the partial derivatives of $I(\mathbf{p},t)$. This problem is not well posed with two unknown variables (u,v). It is common to introduce further constraints in order to solve for (u,v). Many proposed methods including [HS81] associate the above equation with a global smoothness term, and perform a cost minimization over a defined domain D:

$$\int_D (I_x u + I_y v + I_t)^2 + \lambda^2 \left[\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \right] \mathrm{d}\mathbf{p}\ ,$$

where λ indicates the influence of the smoothness term, which, as suggested in [HS81], is set to 100 in our implementation. The velocity $\mathbf{v}=(u,v)$ is estimated by minimizing the above integral using an iteration process:

$$\begin{aligned} u^0 &= v^0 = 0 \\ u^{k+1} &= \bar{u}^k - \frac{I_x(I_x\bar{u}^k + I_y\bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2} \\ v^{k+1} &= \bar{v}^k - \frac{I_y(I_x\bar{u}^k + I_y\bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2}\ , \end{aligned}$$

where k is the iteration number, $\bar{u}^k$ and $\bar{v}^k$ are the averages of u^k and v^k, respectively, in a neighborhood domain. We use 60 iteration steps for the results reported in this paper, which is sufficient for the low resolution videos considered.

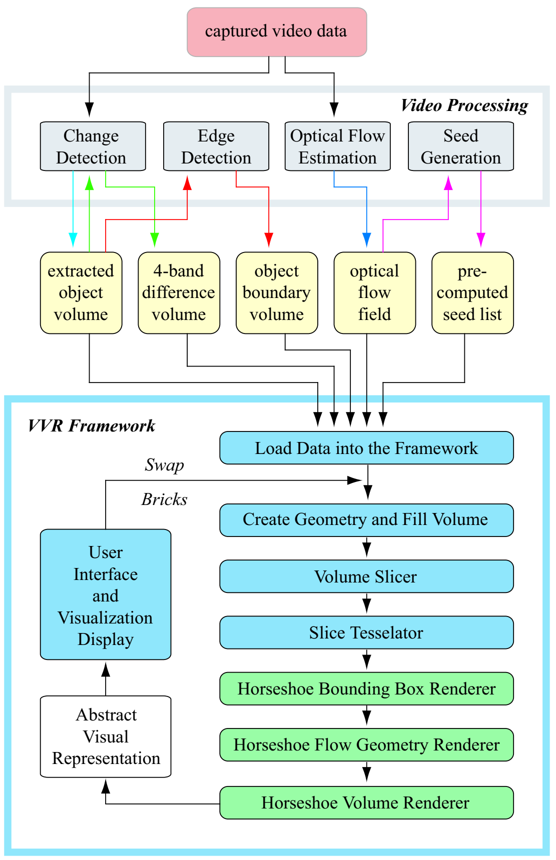


Figure 3: *The technical pipeline for processing and visualizing video data. Data files are shown in yellow, pre-processing modules in grey, software modules in blue, and GPU modules in green.*

4.2. Seed Point Generation

To facilitate the visualization of the optical flow, we need to determine a set of seed points for particle tracing or for positioning flow glyphs. The filtering stage that generates seed points is implemented as a CPU program outside the ac-

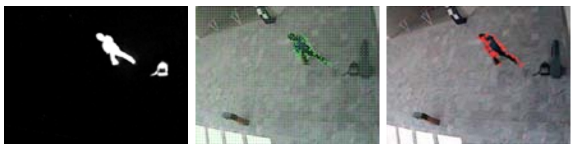


Figure 4: *Image (a) shows the difference object in the scene, computed from an empty reference frame. In (b), the optical flow of the frame is shown with green lines. In (c), seeds are generated based on the optical flow shown in image (b).*

tual rendering framework in order to provide most flexibility in designing the seeding algorithms. Typically, the seeding stage uses the optical flow and the difference object to determine the seed points. Figure 4 or Color Plate I show example frames for seeding. In addition to this external filter, some basic seeding functionality is also implemented in a CPU module in the rendering framework for handling cases where an externally generated seed list is not available.

As detailed in Section 4.1, the 2D vector fields $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ are computed based on the intensity object fields $\{I_1, I_2, \ldots, I_n\}$. Then, the filter stage generates a seed list for every frame in the form of a sequence of text files $\{S_1, S_2, \ldots, S_n\}$. We have designed the following 3-phase algorithm for seeding:

1. The algorithm determines a list of all eligible points in $\mathbf{v}_i$, with two control parameters: *grid interval* and *magnitude threshold*. With the *grid interval* parameter, the user can superimpose a grid on all the 2D vector fields and only grid points are eligible to be selected as seed points. With the *magnitude threshold* parameter, insignificant motion with a magnitude less than the threshold is filtered out.
2. The algorithm sorts the list of eligible seed points according to some criteria of visual importance, typically for instance, the magnitude of the motion vector at each point.
3. Finally, the algorithm selects a set of seeds from the sorted list. The user has the option to select *all* points, to select *the first N* points, or to select *randomly N* points in the list. As the first phase usually produces a large list of seed points, which could lead to slow rendering as well as cluttering the visualization, this selection process allows the list to be trimmed down based on importance.

Figure 4(b) shows an optical flow field estimated for a typical video frame. Figure 4(c) shows an example of a created seed list that was generated from the optical flow in Figure 4(b), using the above algorithm.

5. Rendering Framework

For real-time rendering of large video volumes, GPU methods are employed to achieve high frame rates. The visualization framework is built upon an existing slice-based volume renderer [VWE05]. An advantage of this framework is its separation of different visualization aspects into different software components. The framework is implemented in

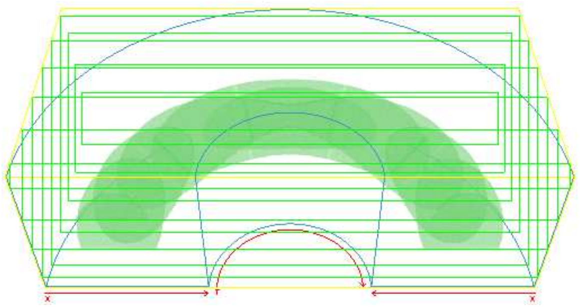


Figure 5: *Bounding boxes of the P-space (blue) and the actually rendered volume (yellow). The volume slice planes (green) are mapped to C-space in the fragment shader.*

C++, using the Direct3D graphics API and HLSL as shader programming language.

In this section, we discuss technical details of video volume rendering and optical flow visualization. The starting point for visualization is volume rendering that shows a scalar field associated with the 3D spacetime video volume. In combination with appropriate transfer functions, relevant information of the video volume can be emphasized and uninteresting regions can be made transparent. A challenge for video volume visualization is the interactive rendering of large datasets (see Sections 6), possibly using a distorted horseshoe geometry (see Section 5.1).

The second part of the visualization system provides a representation of optical flow by glyphs or streamlines constructed by particle tracing (see Section 5.2).

5.1. Distorted Video Volumes

The visible video volume might need to be distorted during rendering. Our primary example is the bending into a horseshoe shape [DC03], as shown in Figure 5. We use a backward-mapping approach for rendering such deformed volumes: instead of deforming the geometry of the volume, we distort the associated texture coordinates to obtain the same result [RSSSG01]. Therefore, planar and view-aligned slices are rendered with modified 3D texture coordinates.

We describe texture coordinates in a computation space (C) by (x_C, y_C, t_C) in the range $[0,1]^3$. Here, x and y denote the spatial dimensions of a video slice and t denotes the temporal dimension. In contrast, the coordinates in the physical space (P) — the object space of the distorted volume — are given by (x_P, y_P, z_P).

For the case of the horseshoe volume, we assume a transformation according to cylindrical coordinates,

$$(x_P, y_P, z_P) = (-r\cos(\pi t_C), y_s\, y_C, r\sin(\pi t_C)) , \qquad (1)$$
$$\text{with} \qquad r = r_{\min} + \Delta r\, x_C \quad \text{and} \quad \Delta r = r_{\max} - r_{\min} .$$

Here, $r_{\max}$ and $r_{\min}$ describe the inner and outer radius of the horseshoe, respectively. The parameter y_s provides a scaling factor for the y dimension. Figure 6 illustrates the different coordinate systems.

The inverse mapping of Eq. (1) is used to transform the physical coordinates of the slices to texture coordinates (x_C, y_C, t_C) that address the video volume. The inverted mapping involves inverse trigonometric functions, which are available in GPU fragment programs. Therefore, the volume deformation can be implemented by computing texture coordinates in a fragment program during texture slicing. An example of such a fragment program is provided in Section 6. Since the video volume is not illuminated, we can omit the transformation of volume gradients for appropriate volume shading (see [RSSSG01] for a description of this type of transformation).

5.2. Integrating Optical Flow in Volume Visualization

To combine an optical flow field with the distorted scalar field for the horseshoe video volume, the VVR system allows opaque flow geometry to be added into the scene. The geometry, in the form of arrow glyphs or traced lines, is created on-the-fly by the module *FlowGeometryRenderer* and stored in the according geometry buffers before the actual rendering takes place.

Building the arrow geometry requires two pieces of information: a point $\mathbf{p}$ and a direction $\mathbf{v}$, which are given by the pre-computed seed points S_i and the optical flow vectors $\mathbf{v}_i$, as described in Sections 4.1 and 4.2. In fact, we extend the original optical flow field from a 2D spatial vector field described by (u, v) to a 3D spacetime vector field with an additional component along the temporal dimension: $\mathbf{v} = (u, v, v_t)$. The temporal vector component v_t describes the "velocity" along the time axis of the video volume. So far, we only use a temporally equidistant sampling of the video volume. Therefore, v_t is constant for the whole volume and represents the relative speed along the time axis. We allow the user to define the relative speed v_t. The example images of this paper use $v_t = 0$ in order to focus on the motion within individual frames. Based on this 3D optical flow, for each seed point a reference geometry for glyphs can be copied to the geometry buffer, and shifted and rotated into the proper position and orientation.

As an alternative, particle tracing is used to visualize the trajectory of particles along the flow and to provide information of longer moving structures inside a frame. These lines not only emphasize the distance of a movement but also can indicate a change in direction. Particle tracing needs more processing steps and is implemented using Euler integration,

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \Delta t \mathbf{v}(\mathbf{p}_i) , \tag{2}$$

where $\mathbf{p}_i$ are positions along the particle trace, $\mathbf{v}$ is the optical flow field, and Δt the integration step size. The tracing procedure can be described as follows. From a given starting point $\mathbf{p}_0$, which is chosen out of the seed point list S_i, a

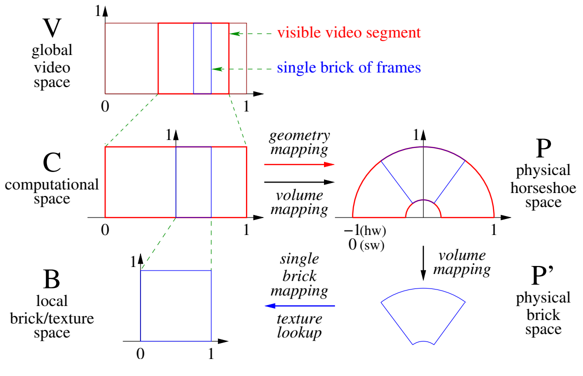


Figure 6: *Mapping between coordinate systems.*

forward integration is applied according to Eq. (2). Here, we use trilinear interpolation as reconstruction filter for the vector field. The number of computed integration steps is chosen by the user, manipulating the length of the traced lines. The rendering of those lines with dynamic texture mapping is detailed in Section 6.2.1.

One additional issue occurs when the video volume is distorted. In this case, the original vector field data, which is given in C space, needs to be transformed into the physical space P in order to obtain correct particle traces or glyph orientations. Similar to the coordinate transformation for the scalar field as discussed in Section 5.1, we also need a transformation rule for vector fields. In general, vectors can be defined as differentials according to

$$\mathrm{d}\mathbf{y} = \sum_{i=0}^{2} \frac{\partial \mathbf{y}}{\partial x_i} \mathrm{d}x_i = \sum_{i=0}^{2} \mathbf{e}_i \mathrm{d}x_i .$$

Here, the $\mathbf{e}_i$ serve as basis for the vectors in the space associated with x_i. In the case of the horseshoe, we have

$$\begin{aligned} \mathbf{e}_x &= \frac{\partial \mathbf{x}_P}{\partial x_C} = \Delta r(-\cos(\pi t_C), 0, \sin(\pi t_C)) \\ \mathbf{e}_y &= \frac{\partial \mathbf{x}_P}{\partial y_C} = (0, y_s, 0) \\ \mathbf{e}_t &= \frac{\partial \mathbf{x}_P}{\partial t_C} = \pi\, \Delta r(\sin(\pi t_C), 0, \cos(\pi t_C)) , \end{aligned}$$

with $\mathbf{x}_P = (x_P, y_P, z_P)$. With these basis vectors, a vector field $\mathbf{v}_C = (v_x, v_y, v_t)$ given in the coordinate system C is transformed to the coordinate system P by

$$\mathbf{v}_P = \sum_{i=x,y,t} \mathbf{e}_i v_i .$$

6. Scalable Multi-field Bricking

To visualize a large video dataset that cannot be loaded to GPU memory *en bloc*, it is necessary to subdivide the whole domain into smaller sections that can be handled and processed by the GPU. We introduce a generic implementation that combines volume visualization and the rendering of flow geometry in scalable user-defined bricks.

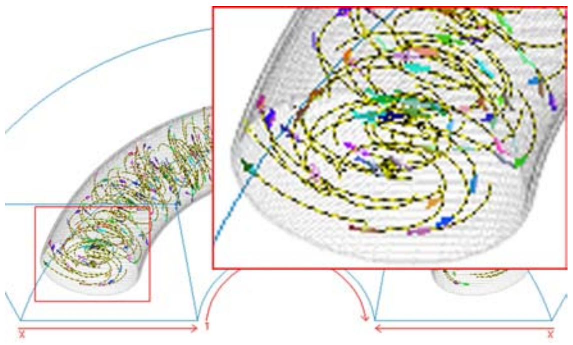


Figure 7: *Directional textured tracelines in combination with arrow glyphs.*

Let the video V be a set of consecutive 2D image frames $I_i, i \in \{1,..,N\}$, where N is the total number of frames. We divide the volume into $K \geq 1$ video bricks, where $1 \leq k \leq K$ bricks are rendered at a time. Each brick, $B_j \subseteq V$, contains m image frames, with $B_j \cap B_l = \emptyset$, where $j,l \in \{1,..,k\} \wedge j \neq l$ and the condition $k \cdot m = n$, with $n \leq N$.

When the GPU memory cannot handle the data size of N frames, we have the condition $n < N$. Thus, dynamic bricking needs to be applied to process the data. Each logical brick is described by two integer values: the number of the starting frame and the number of frames in the brick. Furthermore, we have a brick-based filter for seed generation, which is a modified version of that described in Section 4.2. It enable frames in different bricks to share a pre-processed seed point list. The input of this shared list is used by all bricks and a flag indicates for each point whether this seed point is used for constructing geometry for that brick or not. Starting from this logical entity, we build the whole dynamic bricking structure that consists of k 3D volume textures that are shifted through the horseshoe.

6.1. Bricked Video Spans

The video bricks are represented as a pointer structure that contains k 3D texture objects. Based on the information given by the logical brick structure, the memory for k texture objects is allocated and each single volume brick is filled with its corresponding video frames. Dynamic bricking is realized by reassigning the pointers in a cyclic way, forming a ring-buffer data structure. Thus, the last texture object contains information that can be overwritten and filled with the frames that newly enter the horseshoe.

The fragment program that renders a single volume brick is given in Figure 8. The first line of code scales the texture coordinates to a range of $[-1,1]$, because this permits us to map the cylindrical horseshoe coordinates between $-\pi$ and π. This mapping leads to a half circle in the xz plane, as required by the bent horseshoe (see Figure 6). The following four lines realize the inverse of the mapping in Eq. (1),

```
float  volData, tmpZ; float2 rp;
# horseshoe coordinates
float3 lkup;
# Cartesian coordinates
float3 txCrd = In.TextureCoord0;
# transform to coordinates in P space
txCrd.x = ((txCrd.x*2.0) - 1.0f) * (-1.0f);
# map from P to C; compute radius & angle
rp.x    = sqrt(pow(txCrd.x,2) + pow(txCrd.z, 2));
rp.y    = atan2(txCrd.z,txCrd.x);
lkup.x  = (rp.x - g_fInRad) / (g_fOutRad - g_fInRad);
lkup.y  = txCrd.y; tmpZ = rp.y/g_PI;
# map from C to B
lkup.z  = (tmpZ - g_vScaleCrd.x) * g_vScaleCrd.y;
# perform 3D texture lookup
volData = tex3D(VOLsmp, lkup);
# apply color values and write to buffer
Output.RGBColor = tex1D(TFsmp, volData.x);
return Output;
```

Figure 8: *The complete code of an HLSL fragment program for the bricked, dynamic video spans.*

by first computing the radius and angle of the intermediate cylindrical coordinate system, and then mapping them to the coordinate system C, which represents the visible part of the video volume. The final mapping takes the coordinates into the local coordinate system of the brick B, which is a subset of the visible video volume C. With these brick-related coordinates, a 3D texture lookup is performed and a final RGBα value is assigned according to the transfer function.

6.2. Flow Geometry Bricks

The geometry bricks are similarly to the volume bricks held in a pointer structure that eases the swapping of the bricks for the dynamic rendering of large video data. Unlike the volume bricks, a geometry brick only consists of the logical structure that holds the range information of the currently visible region. The render geometry for arrows and streamlines is constructed for the whole visible horseshoe region (Section 5.2) only when needed and directly mapped from C to P (Figure 6). All points that result from particle tracing are stored in a single vertex buffer and rendered as line strip. The arrow geometry is stored in an indexed vertex buffer to avoid redundant vertices. All buffers are rendered as opaque geometry before the semi-transparent volume is displayed with back to front blending. This rendering order allows us to accurately mix geometry and volume information by means of the depth test.

6.2.1. Directional Textured Tracelines

Lines are 1D primitives that convey information about the orientation and extent of a trace along the flow, but fail to indicate the flow direction. Therefore, we add animation to highlight the direction of flow. The idea is to attach an animated 1D texture that moves into the direction of the flow. The texture needs to have some kind of visual structure so that its motion can be perceived. In this paper, we use a zebra-like texture, as shown in Figure 7. This image illustrates a spinning sphere lying in the xy-plane and rotating around the z-axis. The arrow glyphs rendered at seed-point locations show the flow direction at these certain locations. In contrast, the traced lines provide flow information along a longer distance, covering more locations of the domain.

For texture mapping, each vertex of a line is assigned a texture coordinate, with a range between $[0,1]$ from the first to the last vertex, respectively. By shifting the local texture coordinate of each vertex with a global parameter Δt, the texture moves along the line, in direction of the underlying flow field. The 1D texture does not need to be changed for the animation and can thus be computed on the CPU and downloaded to the GPU once.

7. Results and Analysis

The VVR system is capable of visualizing video streams in real time. With the bricking mechanism, a video stream can be segmented into small time spans, each of which is processed in the video processing sub-system and pushed to the rendering sub-system. The processed multi-field data are then used to update the visualization. In this way, a continuous video stream can be visualized as either a series of horseshoe images, or one dynamically updated image.

The three images in the bottom row of Figure 9 show the snapshots of three time steps of the 'LeftBag' video. From the upper to the middle image, the horseshoe has been updated four times, i.e., moved by four bricks. From the visualization, we can see a moving object (i.e., a person) that entered the scene and then left an object (i.e., a bag) in the scene before exiting. By observing the glyphs associated with the two objects, we can recognize that the object being left in the scene remained stationary until a moving object (in fact the same person) returned and took it away.

Let us consider the visualization of another video clip shown in the bottom row of Figure 10. In both upper horse-

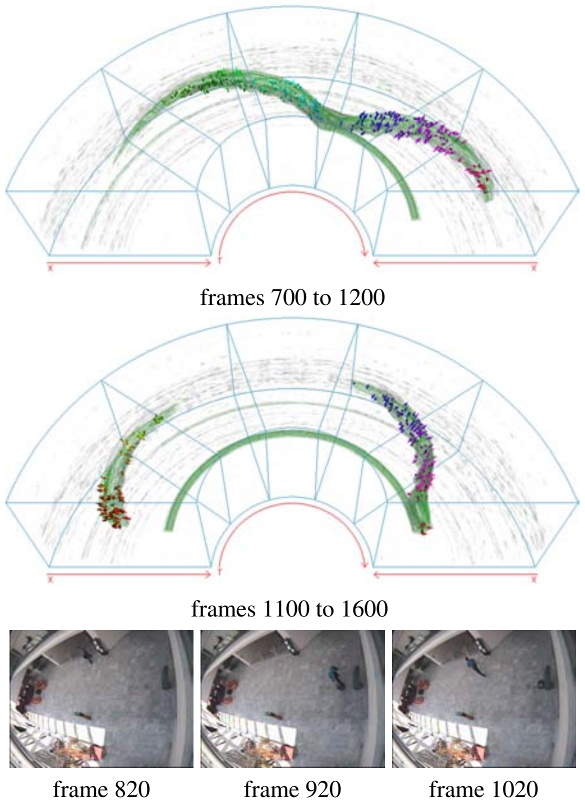


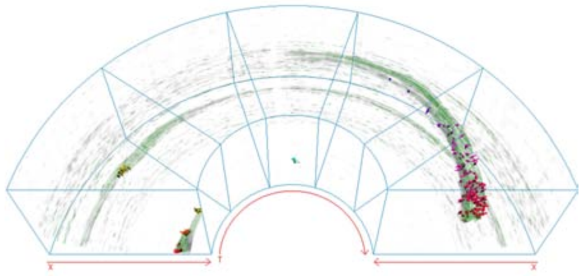
Figure 9: *The two upper images show the 'LeftBag' video rendered as bricked volume horseshoe. The three frames in the lower row present the stages* entering, depositing, *and* leaving. *Ensuing reentering and picking up the box can only be seen in the horseshoe visualization, or the color plate.*

shoe images, a moving object entered the scene and then remained almost motionless for a while before moving again. In comparison with the 'LeftBag' video clip, we can clearly recognize that there was only one object. In fact, this particular video shows a drunken man falling on the floor.

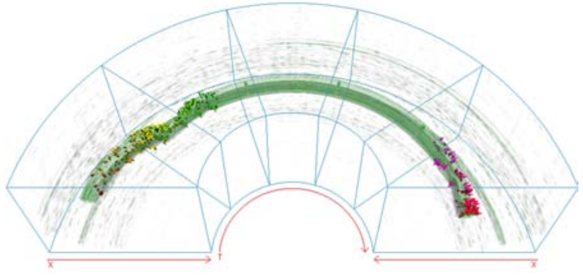
In both video clips, each brick covers a time span of about 3 seconds. With the GPU-assisted techniques described above, VVR can update the dynamic image for each new brick well below one second. The exact timing for different rendering features is given in Table 1. The table demonstrates that flow visualization does not reduce the overall rendering performance: the video span (i.e., volume) with geometry (i.e., flow) is rendered at the same speed as video span only. This behavior can be explained by the fact that the rendering pipeline of VVR renders the opaque geometry prior to the translucent volume. With depth testing activated, the system makes up for the lost speed for rendering geometry by neglecting parts of the volume occluded by the opaque geometry. The results in Table 1 also indicate that the rendering costs are proportional to the viewport size.

Table 1: *Performance results, in fps, for the 'LeftBag' dataset with a resolution of 384 × 288 × 1600 pixels. All timings were measured on a PC with 3.4 GHz Pentium 4 and NVIDIA GeForce 7800 GTX (256MB). The table shows six different types of rendering styles: volume without video span (V-S), volume with video span (V+S), volume with video span and geometry (V+S+G), volume with dynamic video span (V+DS), and all rendering features combined.*

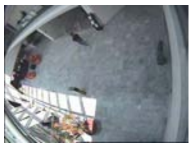
Viewport	800×600	1024×768	1280×1024
V-S	11.04	10.20	8.64
V+S	9.63	7.83	5.47
V+S+G	9.63	7.83	5.47
V+DS	7.40	6.80	5.13
All	7.20	6.65	4.56



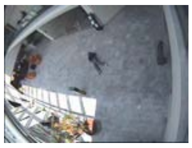
frames 100 to 600



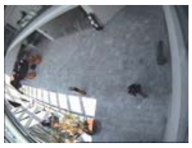
frames 500 to 1000



frame 615



frame 780



frame 995

Figure 10: *Visualization of the 'Rest_FallOnFloor' video. The three frames in the lower row show the stages* entering, lying, *and* leaving, *which can be clearly seen in the upper horseshoe images.*

8. Conclusion and Future Work

In this paper, we have described a system designed specifically for real-time video volume visualization. In fact, most of CCTV cameras provide a video stream with an average of 10 fps or less. Therefore, our basic system is appropriate for pre-processing and visualizing such a data stream in real time. Furthermore, our system is capable of handling multi-field datasets and rendering combined volume and flow visualization. Our bricking approach has been found to play a critical role in delivering this technology. Not only does it enable large multi-field datasets to be accommodated in memory-restricted graphics hardware, but it also provides a practical mechanism for visualizing real-time video streams.

A restriction of the system is the size of the streamable video volume, which is limited by GPU memory. Furthermore, all filters underly the typical problems of image processing algorithms, such as effects of changing lighting conditions or background noise produced by the recording device. One area for future work is to provide a close coupling between video processing filters and the rendering framework for realizing a full pipeline at an interactive rate. In addition, direct streaming should be supported; at this stage of development the video data was streamed from disk, not from a camera.

Acknowledgements

This work was partly supported by Royal Society UK for collaboration between Swansea and Stuttgart, and by an NSERC Discovery Grant to the third author.

References

[BFB94] Barron J. L., Fleet D. J., Beauchemin S. S.: Performance of optical flow techniques. *International Journal of Computer Vision 12*, 1 (1994), 43–77. 3

[CBH*05] Chen M., Botchen R. P., Hashim R. R., Weiskopf D., Ertl T., Thornton. I.: *Visual Signatures in Video Visualization*. Technical Report CSR–19–2005, Department of Computer Science, University of Wales Swansea, November 2005. 3

[CBH*06] Chen M., Botchen R. P., Hashim R. R., Weiskopf D., Ertl T., Thornton I. M.: Visual signatures in video visualization. In *Proc. IEEE Visualization* (2006). 2

[CCF94] Cabral B., Cam N., Foran J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proc. IEEE Symposium on Volume Visualization* (1994), pp. 91–98. 2

[DC03] Daniel G. W., Chen M.: Video visualization. In *Proc. IEEE Visualization* (2003), pp. 409–416. 1, 2, 4

[Dov95] Dovey D.: Vector plots for irregular grids. In *IEEE Visualization* (1995), pp. 248–253. 2

[Fis04] Fisher R. B.: The PETS04 surveillance ground-truth data sets. In *Proc. 6th IEEE International Workshop on Performance Evaluation of Tracking and Surveillance* (2004), pp. 1–5. 1

[GGS02] Guthe S., Gumhold S., Strasser W.: Interactive visualization of volumetric vector fields using texture based particles. In *WSCG 2002 Conference Proc.* (2002), pp. 33–41. 2

[HS81] Horn B. K. P., Schunk B. G.: Determining optical flow. *Artificial Intelligence 17* (1981), 185–201. 3

[IG98] Interrante V., Grosch C.: Visualizing 3D flow. *IEEE Computer Graphics & Applications 18*, 4 (1998), 49–53. 2

[KH91] Klassen R. V., Harrington S. J.: Shadowed hedgehogs: A technique for visualizing 2D slices of 3D vector fields. In *IEEE Visualization* (1991), pp. 148–153. 2

[KY97] Kurzion Y., Yagel R.: Interactive space deformation with hardware-assisted rendering. *IEEE Computer Graphics & Applications 17*, 5 (1997), 66–77. 2

[RSSSG01] Rezk-Salama C., Scheuering M., Soza G., Greiner G.: Fast volumetric deformation on general purpose hardware. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2001), pp. 17–24. 2, 4, 5

[VWE05] Vollrath J. E., Weiskopf D., Ertl T.: A generic software framework for the GPU volume rendering pipeline. In *Proc. Vision, Modeling, and Visualization* (2005), pp. 391–398. 4

[WE05] Weiskopf D., Erlebacher G.: Overview of flow visualization. In *The Visualization Handbook*, Hansen C. D., Johnson C. R., (Eds.). Elsevier, Amsterdam, 2005, pp. 261–278. 2