Real-Time Tracking with Non-Rigid Geometric Templates Using the GPU

Julius Fabian Ohmer, Frederic Maire, Ross Brown
Queensland University of Technology, Brisbane, Australia
j.ohmer@student.qut.edu.au, f.maire@qut.edu.au, r.brown@qut.edu.au

Abstract

The tracking of features in real-time video streams forms the integral part of many important applications in human-computer interaction and computer vision. Unfortunately tracking is a computationally intensive task, since the video information used by the tracker is usually prepared by applying a series of image processing filters. Thus it is difficult to realize a real-time tracker using only the CPU of a standard PC. Over the last few years, commodity Graphics Processing Units (GPU) have evolved from fixed graphics pipeline processors into more flexible and powerful data-parallel processors. These stream processors are capable of sustaining computation rates of greater than ten times that of a single CPU. GPUs are inexpensive and are becoming ubiquitous (desktops, laptops, PDAs, cell phones). They are now capable to greatly relieve the CPU especially for large-scale parallel processing tasks, which map well to the architecture of the GPU. In this paper, we present a system, which uses a gradient vector field to track features with flexible geometric templates. Our implementation is specifically designed to suit the parallel processing architecture of the GPU. It is capable to achieve real-time performance with framerates of around 30 frames per second.

1 Introduction

GPU are the first commodity, programmable parallel architecture that can take advantage of data-parallelism. Remarkably, GPU performance is increasing much faster than CPU performance. GPU evolution and low cost are driven by the computer game market [7]. However, harnessing the power of a GPU is hard because the data-parallel algorithms' mapping to graphics primitives presents many performance pitfalls. GPUs were designed for graphics applications. These are characterized by a lot of arithmetic, intrinsic parallelism, simple control, multiple stages, and feed forward pipelines. Computer vision and computer graphics are both characterized by a high degree of local processing which is performed per pixel (or in a small region, achieved perhaps by filtering) and can be considered (to a certain extent) as inverse processes. Basic image manipulation operations, such as convolution filters and color transformations, are well suited for GPU processing.

Real-time computer vision has many important real-world applications such as video surveillance, human-computer interaction and autonomous vehicle control. Realizing real-time computer vision on the GPU creates new opportunities by drastically lowering the cost of computer vision systems and thus making them available for new applications, such as interactive computer games. Especially 2D games are well suited since they do not use the GPU to its full capacity.

One very important area within the field of computer vision is the tracking problem. The aim is to follow the pose of specific objects such as a finger or a head in a series of videoframes.

In this paper, we present a GPU implementation of a system, that is able to detect and follow objects thanks to a non-rigid geometric template model. The video image frames are transformed into a gradient vector fields, which are then used to manipulate the template. The whole implementation is specifically designed to suit the parallel processing architecture of the GPU. Our unoptimized implementation is running at 30 frames per second on a Nvidia GeForce6800 graphics processor.

Section 2 gives a brief overview of general purpose computation on the GPU. In Section 3, we provide an overview of our implementation, which includes the processing modules and the general data-flow. In Section 4, we present our approach to construct the gradient vector field approximation by processing the image stream at different resolutions. Section 5 explains the non-rigid geometric template model in detail. Section 6 discusses some experimental results. Future work, possible improvements, as well as applications of the existing model are considered in Section 7.

2 General Purpose Computation on Graphics Hardware

Three key factors make the GPU an attractive platform for general purpose computation:

Speed - The computational capacity of GPU increases at an astonishing rate. The rate of growth outpaces Moore's Law. Modern GPUs offer a tremendous

Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation (CGIV'06)
0-7695-2606-3/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

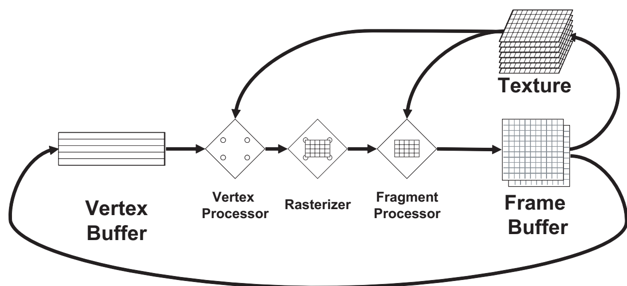

Figure 1: The modern graphics hardware pipeline. The vertex and fragment processor stages are both programmable by the user. Source:[7]

memory bandwidth compared to CPUs. The CPU is optimized for high performance on sequential code and many of their transistors are dedicated to support non-computational tasks, like branch prediction and caching. The highly parallel nature of the GPU in contrast allows us to use more transistors for computation, and achieve a higher arithmetic intensity with the same transistor count.

Value for money - GPUs can be found in off-the-shelf graphic cards which are built for the PC video game market. Their price drops significantly each time a new generation is released.

Technical maturity - Modern GPUs have reached an advanced state of maturity. Today, two processors are fully programmable on the graphic hardware; the vertex and the fragment processor. They are placed in a hardware pipeline as illustrated in Figure 1. The Nvidia GeForce6800 fully supports IEEE 32-bit floating point numbers (single precision). The number of inputs, outputs and instructions for each of the processors has been significantly increased. GPUs support full branching in the vertex pipeline and limited branching in the fragment pipeline.

But modern GPUs present serious challenges. The fact that they adopt the SIMD / MIMD parallel processing scheme means that algorithms only benefit from the architectural properties if they can be adapted to that scheme. Moreover, important fundamental constructs such as integer data operands are missing and their associated operations such as bit-shifts and bitwise logical operations. The GPU lacks 64-bit double precision number formats. The communication between a program running on the CPU and the graphics hardware has to be established by using a graphics API such as *OpenGL*. To perform computations on graphics hardware, the programmer must use graphic primitives such as lines or triangles and stores the data in texture maps. An alternative way we have not pursued is

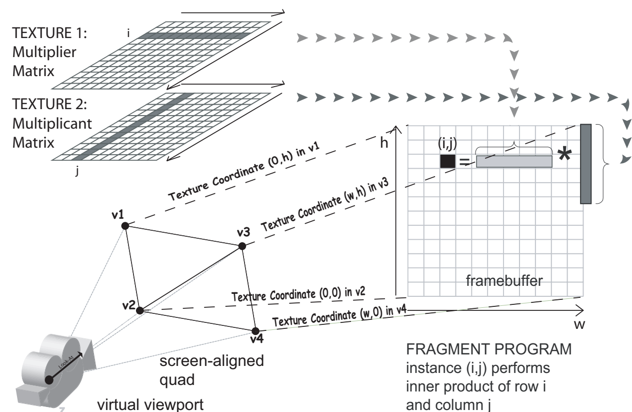

Figure 2: Matrix multiplication on the GPU.

to use one of the stream computing languages for the GPU that are in development.

For further information, the reader is referred to the comprehensive survey of general purpose computation on the GPU conducted by Owens et al. [7].

Next (and in Figure 2), we describe a sequence of steps for performing a matrix-matrix multiplication on the GPU (following a method which was pioneered by Larsen and McAllister [3]).

Preparation - The data arrays are loaded into the GPU memory. These data fields are called *textures*. Most often they are two-dimensional and each entry in the array can hold up to four values (red, green, blue and alpha components). In this example, *texture 1* contains the data coefficients of the multiplier matrix and *texture 2* the data of the multiplicand matrix. A virtual viewport is defined, which must match the dimension of the resulting matrix. Activate a previously defined shader program.

Processing - A screen aligned quad is defined, which must exactly match the viewport dimension. The fragment program performs the computation by rendering the quad. One *processing instance* of the program runs for each pixel covered by the quad. The programs have read-only access to the data resource (textures). The information about the location (the values of i and j) of the $(i, j)^{th}$ fragment processing instance is provided by interpolating the *texture coordinate* values. The texture coordinates are defined in the vertices, which bind the quad. Here, the processing instance (i, j) of the fragment program performs the inner product between the ith row of *texture 1* and the jth column of *texture 2*. The processing instance uses the values i and j to look up the proper locations in the textures. The result of each fragment program in-

Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation (CGIV'06)
0-7695-2606-3/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

stance is written at location (i, j) to the *framebuffer* which is essentially a two-dimensional data array very similar to a texture. The process described here is known as a *rendering pass*.

Results - It is possible to retrieve the content of the framebuffer by performing a read back from GPU memory to CPU memory. Alternatively the content of the framebuffer is used as a texture for a subsequent GPU computation. We use the *Framebuffer Objects* in OpenGL, which support this functionality.

An eclectic range of applications in non-graphics areas for the GPU has already been explored; image processing methods such as convolution filters, color conversion have been efficiently implemented [1]. Fast implementations on the GPU of more complex algorithms include the Fast Fourier [5] and Wavelet transforms [8]. For more information please refer to [7].

It is worthwhile to explore possible ways to implement subsequent computer vision tasks on the GPU after basic image processing operations, since:

- the computational capacity of modern GPUs is far from exhausted with image processing operations;
- the regular transfer of a complete video frame from GPU memory to CPU memory by performing a readback through the *PCI express* bus imposes high cost and is thus not desirable;
- the transfer stalls the GPU pipeline through the readback call, since all computation currently performed by the GPU must finish first before the transfer can be performed;
- some computer vision tasks may need intermediate results from the chain of image processing filters. Those intermediate results are just available on GPU memory.

3 Overview of the Implementation Layout

In our implementation, video image frames are regularly transferred from CPU to GPU memory into a *dynamic texture*. Once the memory space is allocated on the GPU at the initialization phase, the color information of this texture is frequently updated with new images from the video stream. A series of image processing filters is applied to the video frame in a filter graph. The output of the previous filter module is written to a *Framebuffer Object*, which is then used as the two-dimensional input texture of the subsequent processing module. This model fits well within the pipeline model implemented by modern graphics hardware.

In our implementation, which is illustrated in Figure 3, the incoming video frame is first processed by a blur filter to attenuate noise artifacts in the frame. The goal in our demonstration is to track a finger with a geometric template. To mask non-skin color values, we apply a skin tone segmentation filter, which first maps the *RGB* video data into the *YCbCr* color space, which is more suitable for skin detection. Adjustable threshold parameters can be set in the shader program during runtime to allow a robust segmentation. The final image processing step is performed by the *Canny* edge detection filter. It constructs an edge image from the segmented video data. Two specialized filter graph modules follow the general image processing modules. The first one is responsible for constructing the gradient vector field approximation, and the last module performs the tracking of the flexible template. These modules are discussed in further details in the sections 4 and 5 respectively.

4 Fast Gradient Vector Field Approximation Module

A gradient vector field (GVF) constructed from the edge image allows geometric templates (known as *snakes*) to converge towards feature boundaries [9]. Xu and Prince propose to build the field through a diffusion of the gradient vectors derived from the edge image . They describe the GVF as a field consisting of a two dimensional vector $g(x, y) = [u(x, y), v(x, y)]$. At each pixel location $g(x, y)$ points towards the nearest edge. It is constructed through minimization of the energy functional

$$\mathcal{E} = \int\int \lambda\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + ||\nabla f||^2||g - \nabla f||^2 dxdy$$

where λ represents a regularization parameter. We did not follow the numerical implementation that Xu and Prince proposed. The reason is that the computational times presented in their work prohibits real-time performance. Instead we evaluated ways to approximate such a field. We therefore process the image gradients of the edge maps in different resolutions.

Graphics hardware have a feature called *mipmap generation* to automatically construct a *image pyramid* from an input image. Experiments using the automatic mipmap generation mechanism provided unsatisfactory results. We evaluated possible ways to use the automatic mipmap generation to produce a gradient vector field approximation, but the results were of poor quality, because the *box filter* used for the mipmap generation is not adequate. Furthermore was the performance of the mipmap generation slower than anticipated.

IEEE COMPUTER SOCIETY

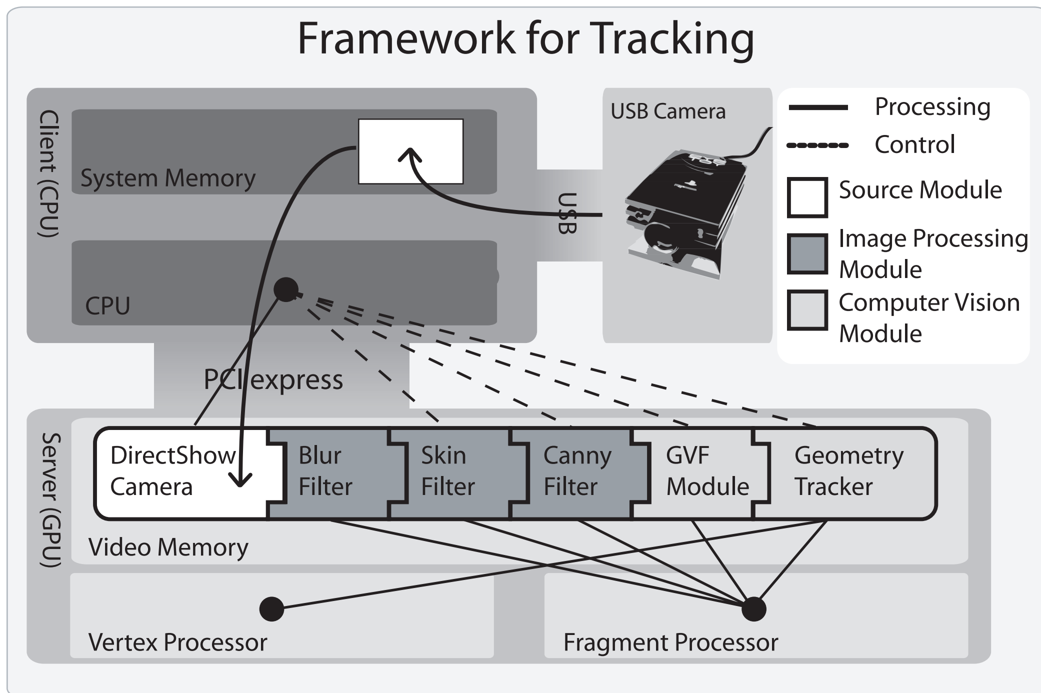

Figure 3: Architecture of the geometry tracker.

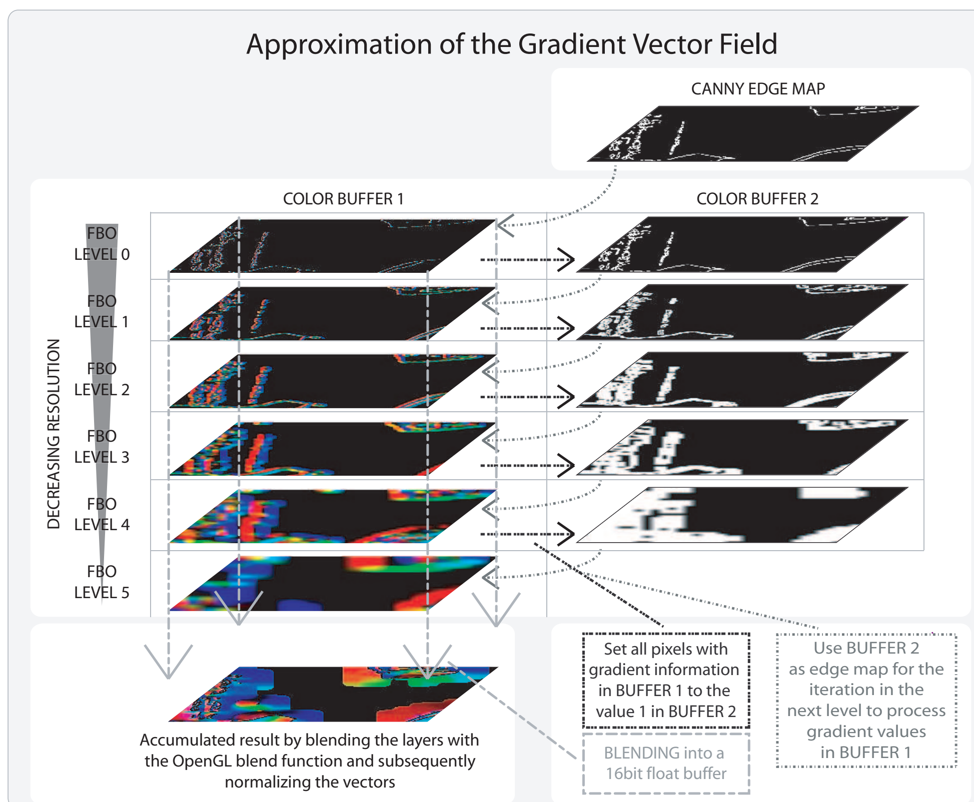

Figure 4: The different levels are constructed by using the edge image from the previous level and computing the gradients on it. Finally the different levels in the image pyramid get accumulated into one buffer using a OpenGL blending operation.

We developed a multipass approach, where the gradient calculation of an edge map is performed at different resolutions. We use several *Framebuffer Objects*, where each buffer represents a level in the *image pyramid*. Each buffer is furthermore equipped with two draw buffers. At level i, the fragment program receives the edge map from level $i-1$, which was produced in the previous rendering pass. The output of the fragment program are the edge gradients, which are written to the draw buffer 1. We compute the gradient image I_g by convolving the edge image I_e with following two masks:

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \text{(horizontal derivatives)}$$
$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \text{(vertical derivatives).}$$

The resulting components u and v of the gradient vector $g = [u, v]$ are stored as a difference of two non-negative numbers. This way, the subsequently applied *max* blending operator can accumulate negative u and v values as well and thus delivers the desired result. Positive u and v values are stored in the red respectively green color channel of the buffer. The absolute value of a negative u or a negative v component is stored in the blue respectively alpha channel.

For each pixel with a non-zero gradient vector in I_g, the value 1 is written to the draw buffer 2. In the next iteration, draw buffer 2 will be used as a binary edge map to process level $i+1$.

This process can be repeated for an arbitrary number of levels. Once all levels are processed, the gradient images with different resolutions are accumulated in one buffer in the next *rendering pass*. Smaller gradient images get scaled up to the original resolution simply by manipulating the *texture coordinates* accordingly.

We give the gradient information provided by a high resolution gradient image more weight, since it contains more details. We slightly decrease the magnitude of the gradients between level i and level $i-1$. To do so, the computed gradient vector at level i is multiplied by a factor p_i, where $p_0 = 1$ and $p_{i+1} = p_i \cdot 0.95$.

For the accumulation, we use a *framebuffer object* with 16 bit floating point precision. In this specific format the current generation of graphics hardware allows to perform hardware accelerated blending of floating point values. The blending function compares an incoming pixel value with the value already stored in the buffer and only keeps the higher value (*max* operator).

After the blending was performed, we can return to the original representation of u and v in the last rendering pass by computing the resulting vectors from the four color channels as

$$u = u_{\text{positive}} - u_{\text{negative}}$$
$$v = v_{\text{positive}} - v_{\text{negative}}$$

Finally the vectors are normalized.

5 Geometry Tracking Module

Kass introduced the classical term of a *snake* model, which describes an energy-minimizing spline guided by external constrain forces and influenced by image forces [4]. The spline is attracted toward feature such as lines and edges. We found that the model is difficult to adapt to the architecture of the GPU and switched to the non-rigid geometry template model that consists of vertices and connecting edges. This model fits well to the nature of the GPU, as these are the native primitives processed by the vertex processor in the rendering pipeline.

Furthermore we were looking for an approach, which is well suited for parallel processing. We have chosen a physically based dynamics model consisting of three forces to track features, which can be processed per vertex independently. The forces are explained in the following subsections in more detail.

5.1 Spring Force

The template must have a force, which restores it to its original shape. We use the proposed GPU implementation of a mass-spring system, which was presented by Georgii and Westermann in [2]. Here , the spring force F_{ij} that exert on connected vertices i and j with their associated current positions p_i and p_j is calculated using *Hooke's law*.

$$F_{ij}(p_i, p_j) = \frac{p_i - p_j}{||p_i - p_j||} \cdot (l_{r_{ij}} - ||p_i - p_j||) \qquad (1)$$

where $l_{r_{ij}}$ is the rest length of the spring between vertex i and j. For the implementation, we chose the *point centric approach* described by Georgii and Westermann, where for each vertex i all the spring forces of the m connected vertices are evaluated. The forces calculated in (1) are accumulated to obtain the resulting total spring force:

$$F_{i_{\text{spring}}} = \frac{1}{m} \cdot \sum_{w=1}^{m} F_{iw}$$

The drawback of the *point centric approach* is however that each spring has to be calculated twice (once per vertex). Alternatively, Georgii et al. proposed an *edge centric approach*. Here, each spring is calculated once and transfers the result to the vertices that are connected to it. This transfer requires to perform a scatter operation, which is not supported by the fragment processor. An additional rendering pass is necessary to circumvent this restriction. Please refer to [2] for more details.

IEEE COMPUTER SOCIETY

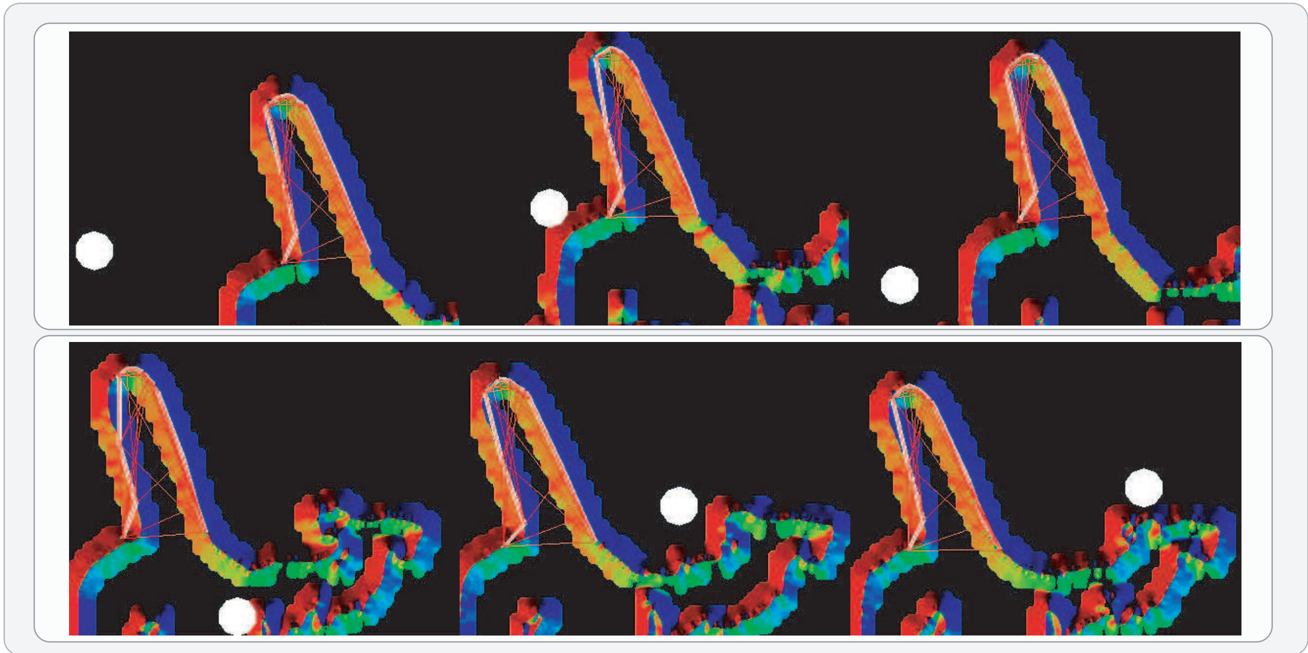

Figure 5: In this sequence (left-to-right, top-to-bottom) the system is tested on a finger. The geometry is used in a *Pong*-like game to interact with the ball. The ball bounces from the geometry around the finger (top left to top right) and the screen borders (bottom left to bottom right). Please note that the screenshots have been cropped for this illustration.

In our proposed implementation, the drawback having springs calculated twice does not have a considerable effect, since we only have relatively few vertices in one template.

5.2 Gradient Vector Force

The force vector can be directly obtained from the edge gradient vector field I_g presented in Section 4 at the current location p_i of the vertex i.

$$F_{i_{\text{gvf}}} = I_g(p_i)$$

5.3 Viscosity Force

To allow stable tracking a term is introduced to reduce the motion of the template.

$$F^t_{i_{\text{viscosity}}} = p_i^{t-1} - p_i^t$$

where p_i^{t-1} and p_i^t is the location of vertex i at the last and the current time step respectively.

5.4 Total Force

A new position p_i for timestep $t+1$ of the vertex i is calculated by:

$$p_i^{t+1} = k_s \cdot F_{i_{\text{spring}}} + k_e \cdot F_{i_{\text{gvf}}} + k_v \cdot F^t_{i_{\text{viscosity}}}$$

where k_s, k_e, and k_v are parameters to balance the forces.

6 Results

We tested our system with a simple geometry template which represents the outline of a finger. We were able to run the complete graph of rendering modules at about 30 frames per second with a video resolution of 640 by 480 pixels using a Nvidia GeForce6800 graphics hardware, a current middle class standard GPU. The results are illustrated in Figure 5.

7 Conclusions

We presented a new approach for template based tracking, which requires just the GPU for processing. We found that a module based approach adapts very well to the GPU pipeline architecture and is capable to have a flexible workload for the different GPU types available. The method to perform diffusion approximation by calculating the gradients at different resolutions on the GPU allows a rapid construction of a gradient vector field approximation, which can be used as a force field. The geometric templates using graphics primitives are well suited for fast and flexible modelling of shapes and furthermore maps naturally to GPU processing. The templates are non-rigid and the spring forces can be processed in parallel. The template is capable of tracking features using a physically based simulation model. The proposed method of tracking can be used:

- to add new gameplay elements to video games;

IEEE COMPUTER SOCIETY

- for flexible human-computer interaction;
- general tracking problems.

An extended version of this paper will be available soon as a technical report via *ePrints* [6].

References

[1] P. Colantoni, N. Boukala, and J. Da Rugna. Fast and accurate color image processing using 3d graphics cards. In *8th International Fall Workshop - Vision Modeling and Visualization 2003, Proceedings November 19-21, 2003 , München, Germany*, pages 383–390. IOS Press, 2003.

[2] Joachim Georgii and Rüdiger Westermann. Mass-spring systems on the gpu. *Simulation Modelling Practice and Theory*, 13:693–702, November 2005.

[3] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC'01)*, pages 55–55, Denver, Colorado, USA, November 10–16 2001. ACM Press.

[4] A. Witkin M. Kass and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1987.

[5] Kenneth Moreland and Edward Angel. The fft on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, San Diego, California, USA, July 26 - 27 2003. Eurographics Association.

[6] Queensland University of Technology. QUT *ePrints* archive, available online: *http://eprints.qut.edu.au/*.

[7] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August, 2005.

[8] Jianqing Wang, Tien-Tsin Wong, Pheng-Ann Heng, and Chi-Sing Leung. Discrete wavelet transform on GPU. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, Los Angeles, California, USA, August 7–8 2004. ACM Press.

[9] C. Xu and J. Prince. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing*, 7(3):359–369, March 1998.