Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Image and Signal Processing (CIISP 2007)

Real-Time Image-Based Stylistic Rendering Using Graphics Hardware Acceleration

Chun-Chung Chiang, Yu-Hung Hsueh and Damon Shing-Min Liu
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan
{ ccc93, hyh94, damon } @cs.ccu.edu.tw

***Abstract*—Silhouette is an important research issue in the field of Non-Photorealistic Rendering (NPR) and it is also a popular drawing feature in illustrations and line-drawing artworks. In this paper, we present a real-time image-based stylized rendering system. First, we project a 3D model to image-space. Then we extract edges in the image-space data. We perform edge-detection algorithms on GPU (Graphics Processing Units) for speedup. GPU is good at floating-points calculating and processing with parallelism. Both features match the property of most image processing tasks. Our system can run at an interactive frame rate when combining our edge-detection algorithms with graphic hardware architecture. We demonstrate that this system performance can reach real-time and render images in good NPR style.**

I. Introduction

SKETCHING is important to express the preliminary state of a draft, concept, and idea. It describes a drawing that is not a final, perfect result. In contrast, common photorealistic renderings cannot communicate ideas, outlines, and proposals. These photorealistic renderings reduce one's ability to rethink enhancement and modifications.

Image-based processing is very popular in recent NPR algorithms. The advantage of image-based processing is that the number of data calculation is less than 3D's. It can also take advantage of hardware acceleration. The cost of model projection is only several read/write operators. The performance of our image-based system solely depends on the size of viewport. 3D geometry complexity takes a negligible effect.

In our system, we first compute the normal buffer of the 3D scene object. Color of each pixel in the normal buffer represents the normal value of the vertex in the 3D model. Abrupt changes in normal buffer usually occur at crease edges. We then evaluate the depth buffer (z-buffer) of the 3D scene object. Similarly, color of each pixel in the depth buffer represents the depth value of the vertex in the 3D model. Abrupt changes in the depth buffer often occur at silhouette edges.

This paper presents a new general purpose rendering system to perform NPR in real-time. We integrate image processing technique with a new hardware platform (called "GPU") and a stylized rendering.

We use an edge-detection algorithm to find out the abrupt changes in both of the normal buffer and depth buffer. We implement the edge-detection algorithm on GPU for speedup. The edge-detection algorithm has a property of data independent, and it can calculate faster in graphic hardware pipeline because GPU is parallel processing.

The performance of graphics hardware increases rapidly so that the computing power of GPU measured by Giga Floating Point Operations Per Second (GFLOPS) doubles almost every year, see as Figure 1. The architectures of modern graphics hardware allow tremendous memory bandwidth and provide fast computational power. For example, ATI X800XT can sustain over 63 GFLOPS (compared to 14.8 GFLOPS theoretical peak for a 3.7 GHz Intel Pentium4 SE unit [1]). Due to the extensive capabilities of the hardware, GPU programming gains lots of interest. Moreover, high level languages have emerged to support the new programmability of the vertex and pixel pipelines. Cg, HLSL, and OpenGL Shading Language are three most common GPU programming languages which allow programmer to write GPU programs in a more C-like language. Many GPU shader programs and techniques are also available on the Internet. Here we use OpenGL Shading Language to develop our application.

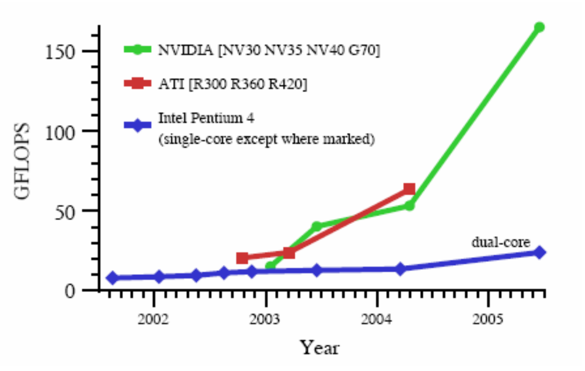

Figure 1. The programmable floating-point performance of GPU (measured on the multiply-add instruction as 2 floating-point operations per MAD) has increased dramatically over the last four years when compared to CPUs. Figure courtesy Ian Buck, Stanford University [10].

II. Related Works

A. NPR

The number of literatures on NPR drawing algorithms is large including issues of stroke placement and orientation [8], silhouette tracing [9], and simulations of particular media [11].

Point-based representation is one of the most used in NPR drawing algorithms. The representation is used for models of very high geometric complexity. With a point-based representation, the surface of a 3D object is described by a set of sample points without further topological information such as triangle mesh connectivity. The lack of topological information leads to simpler and more efficient rendering, simplification, level-of detail control, and texturing for very complex models [18].

Recent research in the field of real-time NPR exploits current graphics hardware. Praun et al. [19] implemented a technique for real-time hatching of 3D shapes. Freudenberg et al. [3] developed a similar technique for real-time halftoning. They made extensive use of the programmable rendering pipeline and texture blending capabilities of current graphics hardware. Nvidia presented an image-space technique to render edges of 3D shapes onto a screen-aligned quad [2]. It samples adjacent texture values to process encoded normals and detects discontinuities in the normal-buffer. ATI extended this approach by detecting discontinuities in the normal-buffer, z-buffer, and Id-buffer [15]. This way, edges of 3D shapes, regions in shadow, and texture boundaries can be outlined [16].

The last decade has seen a blossoming of work on NPR algorithms in a variety of styles [4]. Much of this work addresses the production of still images, while some systems for rendering 3D scenes have addressed the challenge of providing temporal coherence for animations [17].

Most work in NPR has focused on algorithms that are controlled by parameter setting or scripting the designer has no direct control over where marks are made. An inspiration for our work is the technique for direct WYSIWYG (what you see is what you get) painting on 3D surfaces proposed by Hanrahan and Haeberli [6], which is now available in various commercial modeling systems. These tools let the designer paint texture maps directly onto 3D models by projecting screen-space paint strokes onto the 3D surface and then into texture space, where they are composed with other strokes. Strokes then remain fixed on the surface and do not adapt to changes in lighting or viewpoint. In contrast, strokes in our system are automatically added, removed, or modified in response to changes in lighting or viewing conditions [21].

B. GPU-based applications

Recently, programmable GPU has been explored to enhance the performance. Harris et al. presented a physically-based, visually-realistic interactive cloud simulation using GPU [7]. The clouds were modeled using partial differential equations describing fluid motion, thermodynamic processes, buoyant forces, and water phase transitions. Kruger and Westermann introduced a framework for the implementation of linear algebra operators on GPU, thus providing the building blocks for the design of more complex numerical algorithms [13]. Purcell et al. implemented a modified photon mapping technique which uses breadth-first photon tracing to distribute photons using the GPU [20].

In the field of particle system, Kipfer et al. presented a method for simulating particle systems on GPU and implemented inter-particle collision by quickly sorting the particles on GPU to determine potential colliding pairs. Kolb et al. constructed a GPU particle system simulator that supports accurate collisions of particles with scene geometry. In it, depth maps were used to detect the penetrations or collisions by the computation with GPU. Kruger created a particle system for interactive visualization of steady 3D flow fields on uniform grids [12]. The entire process, from vector field interpolation, integration to curl computation, and finally geometry generation and rendering of the stream ribbons, is performed by GPU. Work more related to our system is cloth simulation. Green [5] implemented a simple cloth simulation that executes on GPU using fragment programs and float point buffer. Zeller [22] presented a robust method to simulate cloth on GPU that includes mesh-cutting techniques. A more complete survey on a set of GPU-assisted algorithms and systems is given by Owens [10].

III. Method

Our system consists of the following steps.

1. Tracing silhouette edges using depth map.
2. Tracing crease edges using normal map.
3. Using GPU to speedup edge-detection algorithm.

We can consider edges into two kinds.

Silhouette edges: edges adjacent to a polygon which faces towards the camera and another polygon which faces backward the camera.

Crease edges: edges between two front-facing or back-facing polygons whose dihedral angle is above a threshold. The threshold value defines the number of crease edges.

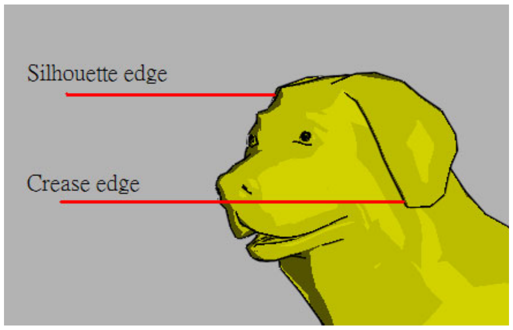

Figure 2. Silhouette edges and crease edges in a model. All of the two edges consist of contour.

A. *Tracing silhouette edges using depth map*

There are many ways to find out edges. We can roughly divide them into two kinds: 3D based and 2D (image) based. Aiming to achieve real-time performance, we conduct both silhouette and crease edges tracing using image based approach.

In computer graphics, silhouette edges in image plane is the collection of points whose outward surface normal are perpendicular to the view vector. A silhouette edge is also an edge which separates a front facing face from a back facing face.

Therefore, traditional method to find out silhouette edges is to trace all edges in a model to find out edges which separate a front facing face from a back facing face and that edges are exactly silhouette edges. This algorithm is not suitable for real-time system because this algorithm uses much time to trace each edge in 3D scene and to compare them. Instead, we capture z-values of the 3D scene into a high precision depth texture which is called a depth map. In OpenGL, for example, the depth map can be extracted by calling glReadPixels() with the GL_DEPTH_COMPONENT argument. Different gray values in the depth map represent positions of different depth. The depth value of background is highest, so that we can distinguish silhouette edges using abrupt changes in the depth map.

Access time of a depth map depends on size of viewport. Smaller size of viewport accesses faster, although larger size of viewport gets better presentation of the scene. We choose a 512 * 512 viewport in the experiments.

B. *Tracing crease edges using normal map*

A problem with using the depth map is that it does not detect the crease edge. To remedy this shortage, we can use another method that is similar to the finding of silhouette edges using depth map. The method uses surface normals to construct a normal map, which is an image that represents the surface normal at each point on an object. The values in each R, G, and B color components of a point on the normal map correspond to the X, Y, and Z surface normal at that point.

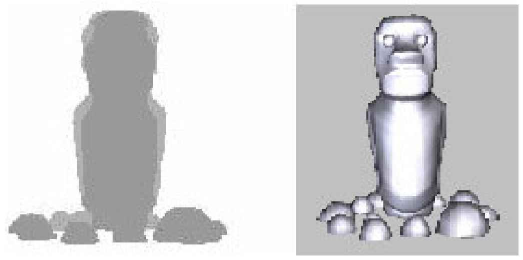

Figure 3. An example of depth map. Right is the original model (Moai of Easter Island). Left is the depth map of Moai.

We first initialize the object color to white, and the material property to diffuse reflection. We then place a red light on the X axis, a green light on the Y axis, and a blue light on the Z axis, all facing the object. Additionally, we put lights with negative intensity on the opposite side of each axis. We subsequently render the scene to produce the normal map. As a result, each light will illuminate a point on the object with intensity proportional to the dot product of the surface normal and the light's axis. An example is shown in Figure 4.

We can therefore detect edges in the normal map. These edges are obtained by detecting changes in surface orientation, and can be combined with the edges from the depth map method to produce a reasonably good silhouette image.

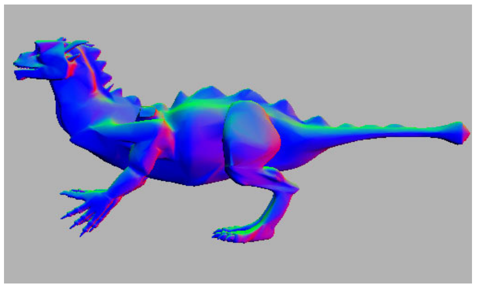

Figure 4. An example of normal map.

C. *Using GPU to speedup edge-detection algorithm*

We present the edge-detection algorithm in our system as follows, and describe how to perform this edge-detection algorithm on GPU.

1) Edge-detection algorithm

Suppose we have two 2D textures, depth map and normal map.

We use Sobel filter to find the edges in the normal map. The Sobel kernels are:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Let *I(x, y)* be the image of normal map or depth map. Vertical and horizontal edge images of *I* are both computed using discrete 2D convolution:

$$I_x(x, y) = I(x, y) \otimes S_x$$

$$I_y(x, y) = I(x, y) \otimes S_y$$

Finally, we use the Sobel filter to find out the edges using a threshold *T*:

Let $I_x(x, y) + I_y(x, y) = I_{mage}(x, y)$,

$$Edge(x, y) = \begin{Bmatrix} 1 & ; \text{ if } & I_{mage}(x, y) \geq T \\ 0 & ; \text{ if } & I_{mage}(x, y) < T \end{Bmatrix}$$

We exploit both normal map and depth map to find crease edges and silhouette edges. Combining the two kinds of edges yields a final edge map.

Although we can get a complete edge map, we observe that there are too many line segments existing in edge map. User becomes more difficult to concentrate on the contour of the scene due to those small fragments. Besides, we notice that all these redundant edges are mostly crease edges, since they often occur in arc surfaces. Face normals in arc surface usually change abruptly; making our system detects many crease edges there. We therefore add an extra filter to remove these redundant lines after getting *Edge(x, y)*.The objective of filtering is that we want to remove the edges which are crease edges with depth values close to their neighbors. This filtering traces the depth map in a way that if a pixel is considered a component of an edge, filter calculates the difference of the depth value between this pixel and its neighbors. If the difference is below a threshold *Tz*, we remove this pixel from edge map, which implies this pixel is no longer a component of the edges.

Entries in extra filter are:

Z_1	Z_2	Z_3
Z_4	Z_0	Z_5
Z_6	Z_7	Z_8

Z_1 - Z_8 are neighboring pixels of Z_0 (Z_0 is the same as $I_z(x, y)$).

$$I_z(x, y) = (|Z_1 - Z_0| + 2|Z_2 - Z_0| + |Z_3 - Z_0| + 2|Z_4 - Z_0|$$

$$+ 2|Z_5 - Z_0| + |Z_6 - Z_0| + 2|Z_7 - Z_0| + |Z_8 - Z_0|)$$

$$Edge(x, y) = \begin{Bmatrix} 1 & ; \text{ if } & I_z(x, y) \geq T_Z \\ 0 & ; \text{ if } & I_z(x, y) < T_Z \end{Bmatrix}$$

The results are shown in Figure 5. We can see there are many redundant lines in Figure 5(right). The result in Figure 5(left) is cleaner. Therefore, users can focus on the important contours using the extra filtering.

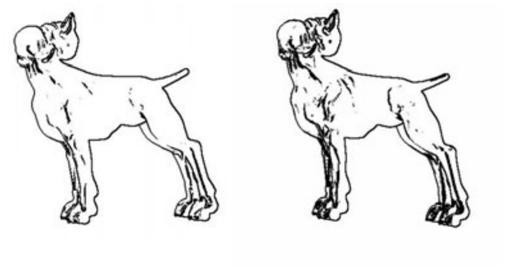

Figure 5. Right is the result without extra filtering. Left is the result with extra filtering.

2) GPU application

GPU is different from the traditional CPU sequential programming model. The graphics pipeline can be illustrated as in Figure 6. The programmable components of graphics hardware are vertex and fragment processor. In GPU, We use the Frame Buffer Object (FBO), which is an OpenGL extension that allows rendering results to a special frame buffer which can be directly read in as texture. Since we want to use fragment processor to compute the physical simulation data, the results need to be sent to frame buffer after fragment program terminates. Using FBO, we can store the computational result to FBO and later retrieve it back as fetching textures. Since the input data and computational result retain in FBO, it achieves a better performance because it avoids extra data copying from frame buffer to texture by calling glCopyTexSubImage2D OpenGL function. Furthermore, less memory is required because there is only one instance of the image [14].

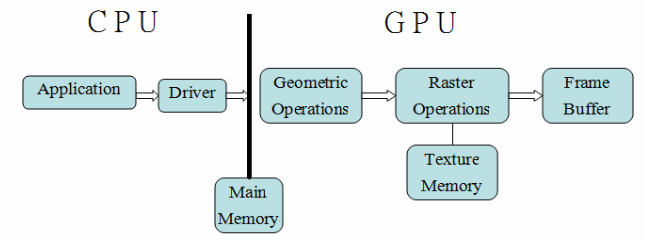

Figure 6. Graphics pipeline

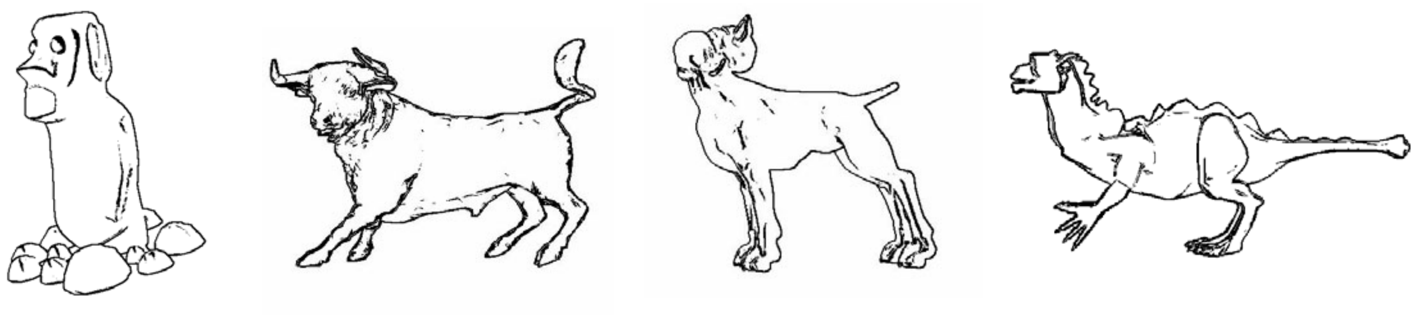

Figure 7. Resulting images. From left to right is “Moai”, “Bull”, “Pitbull”, “Dragon”.

IV. Result

Our system is implemented using OpenGL with C++ on PC with AMD Athlon64 3000+ CPU and 512MB RAM and ATI X1600Pro graphics acceleration chip. Besides using GPU, we built another system which is running on CPU only for comparison as in Table I.

We performed simulations on four different sizes of models. The “Moai” model run on CPU achieved a frame rate of 10 FPS (frame rates per second), and this same model run on GPU obtained 114 FPS. The speedup factor is over 11 times. The largest size Dragon model run on CPU achieved 8 FPS. Same model run on GPU achieved up to 64 FPS. The speedup factor is 8 times. We therefore consider our system can run at real-time frame rates because human eyes can recognize well at approximately 20 FPS.

Assume we define the real-time frame rate as 30 FPS. Our system thus must spend less than 0.03 second to render a scene. In CPU, our system spent almost 0.1sec to render the smallest size model “Moai”, the rendering time is too much to achieve real-time interactivity. In GPU, the speedup ratio is amazing. Speed-up with ten times enables our system easy to achieve real-time interactive frame rates.

Table I

Compare the performance for run on CPU and on GPU.

Model	Triangle Number	CPU (sec/frame)	GPU (sec/frame)	Speedup ratio
Moai	8956	0.098117	0.008739	11.22
Bull	12398	0.099837	0.009015	11.07
Pitbull	25,030	0.105496	0.010680	9.87
Dragon	57532	0.114573	0.014172	8.08

The performance of our system heavily depends on the size of viewport. Consequently, even the number of triangles of the model “Dragon” is more than that of model “Moai”, the rendering time is still the same. As shown in Table I, a more complex model, like “Dragon”, takes more time to render due to its preprocessing steps, like loading model.

Several resulting images are shown in Figure 7. Our system renders scenes with a pen-and-ink style. The images are actually like human-drawing.

V. Conclusion

How to speed up NPR is a topic of research challenge. Many researchers have been addressing this issue. Some may focus on NPR algorithm, hardware, reduced model, etc. If we aim at the performance, we have to reluctantly degrade the NPR quality. In contrast, if we aim at the NPR quality, we have to sacrifice the system performance. Our main contribution in this work is that our system can accomplish real-time interactive frame rates. Reallocating a great part of computing to GPU accounts for achieving such speed-up.

Another technical contribution is that we provide robust edge detector. Users simply need to select a 3D model, our system can provide its sketchy rendering. Users can move, shift, rotate, and resize the model at will and the system can render the varying scene. Besides, the contour of the model is accurate and concisely depicted.

VI. Future Work

Our system may improve in three different aspects.

First is we can transfer much calculation to GPU for faster rendering. We may place more independent data on the powerful GPU because GPU can be executed in parallelism.

Second is we can explore better methods to find silhouette and crease edges. Recent researchers brought up many good new edge-detection algorithms. Although our edge detector perform well, it still can be better.

Another possible enhancement is to add more stylistic strokes in our system to make it even more like human drawing, such as pen-and-ink, watercolor, oil painting, etc. NPR is a very popularly addressed issue in computer graphics. Our research effort is to integrate it with contemporary GPUs to improve the overall performance of NPR.

VII. References

[1] I. Buck, "GPGPU: General-purpose computation on graphics hardware—GPU computation strategies & tricks." *ACM SIGGRAPH Course Notes*, 2004.

[2] S. Dominé, A. Rege, and C. Cebenoyan, "Real-time hatching", *Game Developers Conference 2002*.

[3] B. Freudenberg, M. Masuch, and T. Strothotte, "Real-time halftoning: a primitive for non-photorealistic shading." *13th Eurographics Workshop on Rendering.* Pisa, Italy, June 2002, pp. 1–4.

[4] B. Gooch, P.-P. J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld, "Interactive technical illustration." *1999 ACM Symposium on Interactive 3D Graphics*, 1999, pp. 31–38.

[5] S. Green, NVIDIA particle system sample. http://developer.nvidia.com/object/demo_cloth_simulation.html, 2004.

[6] P. Hanrahan, P. Haeberli, "Direct WYSIWYG painting and texturing on 3D shapes." *Proc. of SIGGRAPH 90*, 1990, pp. 215–223.

[7] M. Harris, W. Baxter, T. Scheuermann, and A. Lastra, "Simulation of cloud dynamics on graphics hardware." *Proceedings ACM SIGGRAPH / Eurographics Workshop on Graphics Hardware 2003*, pp. 92–101.

[8] A. Hertzmann, "Painterly rendering with curved brush strokes of multiple sizes", *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, Orlando, Florida, 1998, pp. 453–460.

[9] T. Isenberg, N. Halper, and T. Strothotte, "Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes." *Computer Graphics Forum 21*, 3 Sept., 2002.

[10] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell, "A survey of general-purpose computation on graphics hardware", *Eurographics 2005*, State of the Art Reports, pp. 21–51.

[11] J.P. Lewis, N. Fong, X. X. Xiang, S.H. Soon, and T. Feng, "More optimal strokes for NPR sketching", *Proceedings of the 3rd international conference on Computer graphics and interactive techniques*, 2005, p. 47.

[12] J. Kruger, P. Kipfer, P. Kondratieva, and R. Westermann, "A particle system for interactive visualization of 3D flows." *IEEE Transactions on Visualization and Computer Graphics*, 2005.

[13] J. Kruger, and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms." *ACM Transactions on Graphics 22*, 2003, pp. 908–916.

[14] L.F. Chang and D.S.M. Liu, "Real-time deformable object simulation using the GPU", *Proceedings of the 2005 Computer Graphics Workshop*, 2005, p. 29.

[15] J. L. Mitchell, C. Brennan, and D. Card, "Real-time image-space outlining for non-photorealistic rendering." *SIGGRAPH 2002 Sketch*, San Antonio, July 2002.

[16] M. Nienhaus and J. Doellner, "Edge-enhancement – an algorithm for real-time non-photorealistic rendering", *Journal of WSCG*, 2003, pp. 346–353.

[17] M. Nienhaus, and J. Döllner, "Sketchy drawings", *Proceedings of the 3rd international conference on Computer graphics*, 2004, pp. 73-81.

[18] N. Zakaria and H. Seidel, "Interactive stylized silhouette for point-sampled geometry", *Proceedings of the 2nd international conference on Computer graphics and interactive techniques*, 2004, pp. 242–249.

[19] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein "Real-Time Hatching." *In Computer Graphics (Proceedings of SIGGRAPH'01)*, 2001, pp.579–584.

[20] T. Purcell, C. Donner, M. Camarano, H. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware." *In Proceedings ACM SIGGRAPH / Eurographics Workshop on Graphics Hardware 2003*, pp. 41–50.

[21] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, A. Finkelstein, "WYSIWYG NPR: drawing strokes directly on 3D models", *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 755- 762.

[22] C. Zeller, "Cloth simulation on the GPU." *In ACM SIGGRAPH 2005 Conference Abstracts and Applications*, 2005.