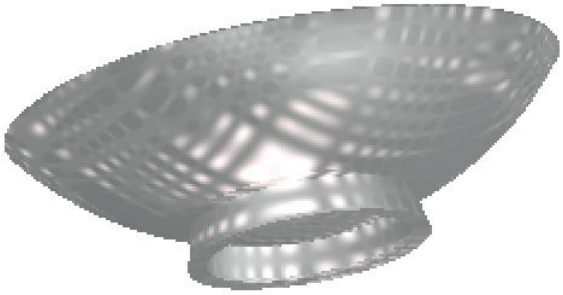
GPU-based real time FRep ray casting

Oleg Fryazinov, Alexander Pasko
The National Centre for Computer Animation
Bournemouth University, UK
ofryazinov@bournemouth.ac.uk, pasko@acm.org



Abstract

A new method is presented for rendering general FRep (functionally represented) models using GPU-accelerated ray casting. We use the GPU acceleration for all computations in the rendering algorithm: ray-surface intersection calculation, function evaluation, and normal vector computation. Performing geometric intersection calculations in parallel with shading allows us to combine the whole process of rendering within one fragment program on GPU. The algorithm is well-suited for modern GPU and provides good performance with good quality of results; it is practically memoryless and does not require a powerful CPU.

Keywords: Ray Casting, Real Time, FRep, implicit models, rendering, visualization, GPU.

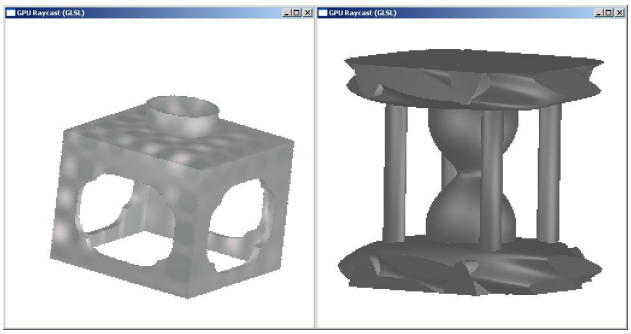
1. INTRODUCTION

The function representation (FRep) defines a geometric object by a single continuous real function of point coordinates as: $F(X) \geq 0$ [15], where the function is evaluated while tracing an underlying tree structure or by running a "black box" evaluation procedure. Functionally based models are also called implicit models or implicit surfaces. Methods of constructing implicit models are developed well enough; however, rendering of these models with interactive rates remains an open problem.

At present, there are two ways to render FRep models. The first one is the polygonization, where the surface of a FRep object is represented approximately with a set of polygons. The second one is ray-tracing. The polygonization has become popular due to the intensive development of software and hardware for polygonal mesh visualization. However, it is memory- and computationally expensive to generate in real time and moreover it is not robust, because features like spikes and sharp edges can be lost during the polygonal mesh generation. Ray-tracing is regarded as more precise method to visualize FRep models; however it is computationally expensive to perform in real time too.

In this paper, we present a method of ray-casting (ray-tracing primary rays only) accelerated using GPUs (graphics processing units) and specialized for rendering FRep models with interactive rates. In recent years, the evolution of graphics hardware has resulted in using graphics cards not only for rendering polygons, but for solving more general problems. It is due to introduction of shaders, which are GPU programs for processing vertices and fragments. Although shaders basically allow for the parallel calculations of positions of vertices or colors of pixels, they can be used as programs for more general computations. These computations can be performed faster on GPUs than similar computations on CPUs because of using multi-core parallel processing in modern GPUs.

The ray-tracing algorithm computes the ray-surface intersection for the particular pixel independently from other pixels; therefore we can accelerate these computations with parallel processing on a GPU. In our method, we use GPU for all the necessary computations: ray-surface intersection calculation, function evaluation, and calculation of the normal. By performing shading computations in a shader program we avoid superfluous computations on CPU and process all the data in one shader pass. Computation of normals of the FRep surface allows us to use per-pixel lighting, leading to better surface rendering. Moreover, we only need to store ray data (two vectors) for each pixel, so our method is practically memoryless, thereby alleviating the large memory consumption problems essential to polygonization based rendering.



a. b.

Figure 1: Rendering of some FRep objects using our method. a) stand (virtual shikki), b) sandbox (Hyperfun gallery)

By using acceleration on GPU, we achieve ray-tracing performance competitive with real-time. We also present techniques for additional accelerations of the ray-tracing algorithm that allow for further improving its performance.

2. PREVIOUS WORKS

Ray-tracing of functionally based models and especially skeletal implicit surfaces was examined by many researchers in recent years. Classical methods of ray-tracing were summarized in [9]. These methods generally are very slow even on modern hardware and they were improved for different special cases. Thus, in [17] ray-surface intersection was accelerated using polynomial approximations of implicit surfaces. The authors of [5] deal with implicit models represented by tree data structures; rendering was accelerated due to this restriction on models. For models represented by tree structures, acceleration was achieved in [10] because of analytical root finding in the tree leaves with implicit surface primitives. Although these methods provide good performance, they are not appropriate for general case objects, because not each procedural model can be easily represented with a tree-like structure. In [4] acceleration was achieved for dynamic scenes, where the previous frame was used as input data for the current frame. However, rendering of the first frame and finding the difference between frames remains computationally expensive.

Ray-tracing on GPU is also a well researched area. However, most papers have focused on polygonal meshes and parametric surfaces. Thus, GPU-accelerated ray-tracing for triangle meshes was introduced in [16, 1], where computations were divided between GPU and CPU because of limitations of graphics hardware. Further development of GPU-based ray-tracing of scenes composed of triangle meshes was considered in [2], where classical recursive ray-tracing was implemented for GPU. In [6], performance of ray-tracing was improved using kd-trees, in [20] acceleration of ray-tracing was achieved using threaded bounding box hierarchy stored as a geometry image. GPU ray-tracing of volumetric data using graphic hardware was introduced in [12]. The work [14] presented a method of GPU rendering of piecewise algebraic surfaces.

GPU-accelerated ray-tracing of implicit surfaces was introduced only for several particular types of surfaces. The work [3] considered ray-tracing of implicit surfaces defined by radial based functions. Rendering of quadratic implicit surfaces on GPU was reviewed in [13]; in [19] effective ray-tracing on GPU was implemented for objects represented by CSG-trees with pre-defined primitives; and [8] introduced ray-tracing of discrete isosurfaces.

3. ALGORITHM DETAILS

The main idea of our algorithm is using GPU for most calculations in classical ray-tracing algorithms adapted to FRep objects. As it was mentioned above, the calculation of the ray-surface intersection for each pixel does not depend on other pixels. Therefore, we can use parallel calculations in GPU to accelerate rendering. We use a program called a fragment shader to perform computations on pixels. We use the fragment shader program for performing the following:

- Find the ray-surface intersection
- Calculate the surface normal
- Compute shading

The contents of the fragment shader are shown in Figure 2.

The main calculation load is accounted for the function value calculation at the given point. If we use any classic method such as Newton root-finding or even dichotomy, we have to calculate function values for one ray many times. Therefore the main computations can be divided into two parts:

- Calculation of the function value
- Root finding when the function value is known

The first part concerns the internal model representation. The second part concerns the ray-surface intersection algorithm. The part of the shader that calculate the function value depends on the model and should be re-generated for each new model; the root-finding part is similar for all the models and depends only on the selected method of the ray-surface intersection.

For our implementation, we use OpenGL and the shader language GLSL. Note that we should bare in mind GPU restrictions such as inability to use recursion or early breaks in functions. Hardware restrictions depend on current graphics hardware, in this paper we mention restrictions that we met during the implementation.

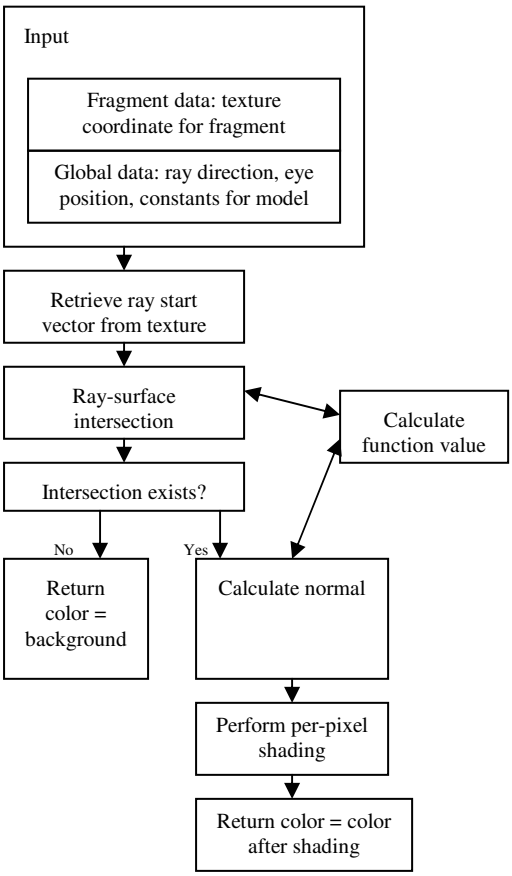


Figure 2: Scheme of the fragment shader.

3.1 Model representation

We briefly review how the model can be represented by a fragment program in GPU. In the function representation any object can be described by a real value function with real value

arguments. This function can be either given by a text file describing a tree structure (as in BlobTree [5]) or by an evaluation procedure (HyperFun [11]). In this work, we use HyperFun objects, because both HyperFun and GLSL are C-alike languages and the conversion between them can be done easily. Thus, the model representation in the form of a HyperFun file is converted to a shader program in GLSL.

The problem in the conversion from HyperFun could be with the library of primitives and operations, but all the library functions can be implemented in GLSL without any problem. For example, we show below how a HyperFun model of a sphere can be represented with GLSL functions:

HyperFun:

```
my_model(x[3], a[1])
{
my_model = 9 - x[1]* x[1] - x[2]* x[2] - x[3]* x[3];
}
```

GLSL:

```
bool my_model(in vec3 vecX, in vec3 vecA, inout float fValue)
{
fMyModel = 9-vecX.x*vecX.x-vecX.y* vecX.y-vecX.z*vecX.z;
bool bResult = (fMyModel>=0);
return bResult;
}
```

Thus, the function related part of the shader can be generated as follows:

- Check the input file for using library functions
- For the used library functions, include their implementations into the shader
- Convert objects from the input model to the shader language

In the similar way, practically all HyperFun programs can be converted to the GLSL representation. As the result we have the function in the shader that has point coordinates as the input and the function value as the output.

3.2 Ray-surface intersection

For a FRep model, ray-surface intersection means the search of zero roots of the defining function along a ray. This process can be done using either

- Approximate methods, or
- Methods with exact root search or with localization of several roots and approximate search of the others.

Approximate methods were discussed in detail in [9]. In our method, we use interval analysis combined with the Newton method. This method was selected as the easiest one to implement and relatively robust. Thus, during the generation of the fragment shader, we add the ray-surface intersection part built with the following algorithm:

- calculate function value at the first point of the ray
- subdivide the ray into intervals
- for each interval
    - calculate the function value at the end of the interval
    - compare signs of the function at the beginning of the interval and at the end
    - if signs are different, set the flag of the found root as true
- if the interval with a root is not found, return the no-intersection flag
- depending on the interval tolerance calculate the number of iterations for the Newton method
- at each iteration refine the root with the Newton method
- return the intersection point coordinates

The length of the interval and all needed tolerances are set manually by the user. Input data for the ray-surface intersection are given for each pixel and include the ray beginning vector of coordinates and the ray direction. Moreover, input data can be reduced up to just the ray beginning vector, because the ray direction is the same for all primary rays. Thus, in our algorithm we have only one vector as the input data for each pixel. We store these data in texture and pass this data to the fragment shader using rendering a single polygonal primitive with 1:1 pixel-texel mapping.

The performance of the ray-surface intersection algorithm can be increased using exact root search. Unfortunately, in general case we cannot find exact roots, but we can use it as preprocessed data for some case studies. In section 4.2 we consider details of these modifications.

3.3 Calculating the surface normal and shading

The result of the fragment shader is the colour of the pixel. Many applications use graphics hardware to perform advanced per-pixel shading. We use it also in the generated fragment shader in such way that shading computations take place together with other computations considered above.

In addition to global light parameters such as light position and color, eye position and view direction, we have to calculate the surface normal vector at the found ray-surface intersection point. We use an approximate method to calculate the normal locally as:

$$n(x) \approx -(f(x+\varepsilon, y, z) - f(x), f(x, y+\varepsilon, z) - f(x), f(x, y, z+\varepsilon) - f(x))$$

Note that we have to calculate additional function values for the normal calculation; therefore the normal calculation procedure should be inserted after the defining function in the generated shader. The shading is performed using the Phong method or similar. In the shading step we can also add procedural texture using methods, described in [18].

4. RENDERING

In this section we show how rendering of a functionally based model is performed using the fragment shader generated by our algorithm. Because of restrictions of modern graphics hardware, we have to use auxiliary polygonal data for rendering. However, we can render only a single fit-to-viewport polygon and it is sufficient for performing the shader calculations.

The standard pipeline for polygonal object processing is shown in Figure 3.

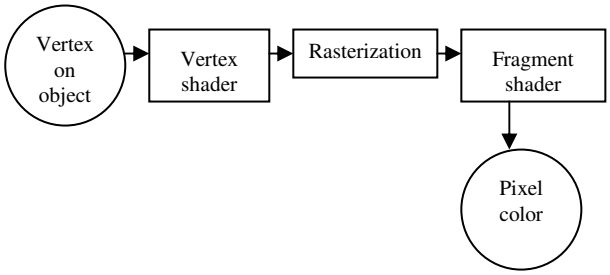


Figure 3: Standard rendering pipeline

There are two programs in the pipeline; one is applied to vertices and another one – to fragments. In our algorithm we do not use a vertex program at all because all the calculations are performed in the fragment shader. As mentioned above, we use texture as the container for input data and we apply this texture to the fit-to-viewport polygon using 1:1 (one pixel – one texel) mapping.

In this work we use the main algorithm for general models and its modification with pre-processing on CPU for better speed and quality for case studies. Below we describe these rendering algorithms in detail.

4.1 Rendering using the fragment shader

The implementation of the rendering algorithm on GPU is a classical approach from the point of view of general-purpose calculations technology on GPU, also known as GPGPU [7]. As it was mentioned above, the auxiliary model for rendering is the viewport-sized polygon. We also use viewport-sized texture that we map to this polygon, so we have 1:1 mapping from texel to pixel. This texture is the data source for our method, and we store point coordinates in it. The viewing direction and the bounding box are defined as global variables that we pass to the fragment shader.

In the fragment shader, we obtain the ray position coordinates from input data stored in texture, and then calculate the ray-surface intersection and the normal, and after that in the same fragment shader we make shading. Return data is pixel color in output area. If there is no intersection, we shade the pixel to the background color. As we make shading inside the fragment shader along with other calculations, we do not have to return anything to the CPU programs unlike general GPGPU programs. The rendering process in this case looks as follows (Figure 4):

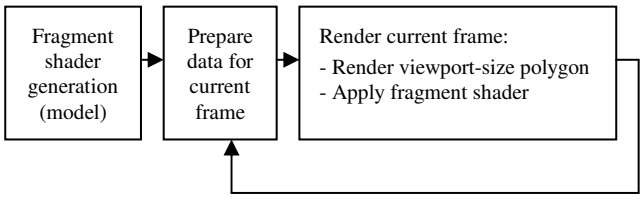


Figure 4: Rendering using the fragment shader.

Advantages of such a process are:

1. All calculations performed only in the fragment shader, there are no suspicious calculations. Moreover, we do not encounter unnecessary rasterization issues, because there is only one quad (two triangles) as the input data.
2. We can get the best image quality for general functionally represented models. However, the better quality-lower speed law is applicable.
3. Shading is performed along with other calculations, so there are no unnecessary data transfers.

Disadvantages of the described approach:

1. For general models we use interval analysis, so the object has thin or sharp features, we can skip an interval, where the root is located. The solution is to decrease the interval length, but in this case the number of intervals increases and the speed is reduced.
2. Some GPU cannot handle fragment programs with many instructions, therefore some complicated models cannot be rendered with our method.

4.2 Rendering with CPU pre-processing

We use this modification of the general algorithm when the ray-surface intersections can be calculated at the pre-processing step. This technology was introduced in [10] for CPU-based rendering. The pre-processing step on CPU can include:

1) General procedural solution for the ray-surface intersection points. In this case we just need to substitute initial data such as ray origin and direction to the provided solution and find the function root. These calculations can be executed on GPU and the rendering process looks as following (Figure 5):

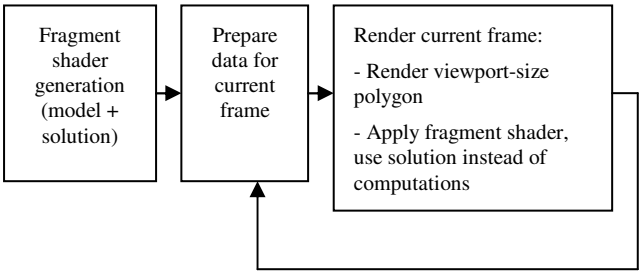


Figure 5: Rendering with CPU preprocessing (procedural solution).

2) Exact roots. In this case we calculate the roots on CPU, then calculate normals and perform shading on GPU (Figure 6):

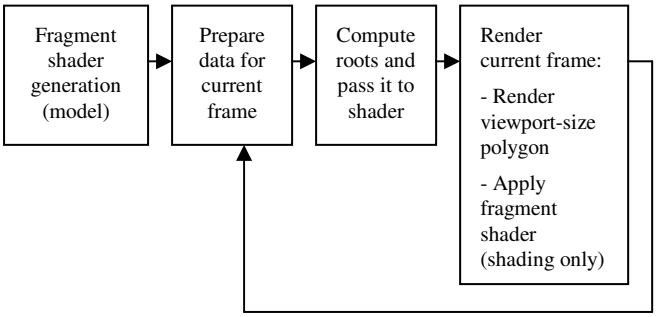


Figure 6: Rendering with CPU preprocessing (exact roots).

Advantages of this modification:

1) If the exact root finding is possible, the speed of the rendering and the quality is the best between all these methods.

2) The number of operations on GPU is less than in other modifications, so we can use more complicated models.

Disadvantages:

1) Unfortunately, exact function roots for the ray-surface intersection cannot be found for an arbitrary model. An even relatively simple object such as blended union between two cylinders leads to the root search in polynomials of degree of 5. In such cases we have to use approximate methods and the speed with quality can decrease.

2) If there are many possible roots, problems can occur with transferring these data from CPU to GPU because of limits of the data which can be transferred within one pass.

5. RESULTS

We tested our algorithm implementation with rendering several relatively simple and more complicated functionally based models (Figure 1, 7, 8). In the performance results shown below, we use a standard torus primitive as a simple model. We also used complicated models from the Virtual Shikki project [11]. For computations in general methods we use the tolerance value that produces minimum of artifacts.

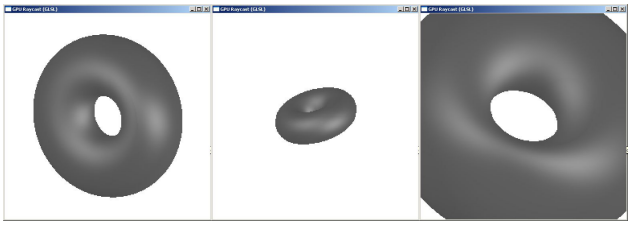


Figure 7: Rendering of a torus primitive: real time rotating and zoom, Phong shading (2 light sources).

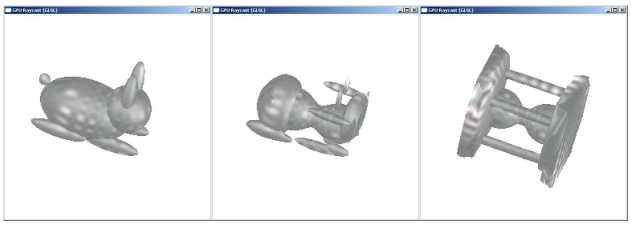
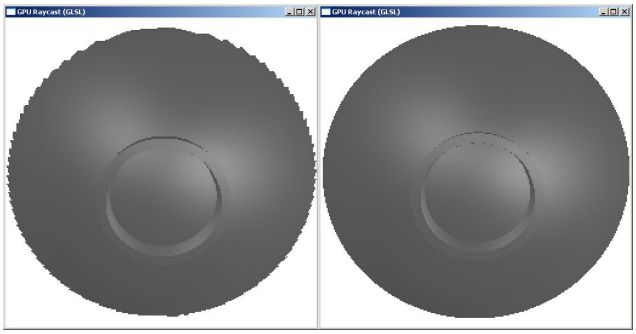


Figure 8: Rendering of dynamic model with procedural texturing: metamorphosis from a rabbit (Hyperfun gallery) to a sandbox (Hyperfun gallery)

Performance characteristics of our implementation were measured on a PC with single NVIDIA GeForce 6800 card and Intel Pentium 4 3.20GHz CPU. According to specifications, this graphics card can process up to 16 pixels per clock. All models were rendered on a 256x256 pixels viewport. For comparison with CPU, we also measure speed characteristics of a CPU-based ray-tracer implemented in PovRay (namely CPU general) and a CPU-based ray-tracer with root solving implemented in [10] (namely CPU, root solving). We provide the result in the following table, where speed is measured in frames per second (bigger fps means higher speed).

	CPU, general	CPU, root solving	Fragment	CPU/Fragment
Torus	0.16	6	30	100
Cup	0.25	N/A	20	60
Rabbit	0.03	4.7	12	50
Sandbox	0.015	N/A	10	N/A
Stand	0.02	N/A	8.56	N/A

It can be seen from the table that with the proposed GPU-based algorithm we can achieve the substantial acceleration of rendering (up to five times) compared with the fastest CPU-based method.



a b

Figure 9: Rendering of a cup model. a) Fragment b) CPU/Fragment. Note that because of interval method errors we have bowed edge in the first case.

Adding pre-processing on CPU, when the model allows, brings further increase in speed with the factor up to three and, in some cases, improves quality (Figure 9). Another shown advantage of the GPU-based solution is that it can process general procedural models, while other methods have limitations on model structure or complexity.

6. CONCLUSION

In this paper we presented the method of real-time rendering of general procedural FRep models with GPU-accelerated ray casting. As it was shown, with this method good image quality can be obtained along with high rendering speed providing interactive frame rates. This has been achieved because of:

- Parallel calculations on GPU of intersection points of cast rays with functionally represented models.
- Depending on the algorithm modification, we can obtain higher speed along with lower quality for approximate visualization; also we can use modifications to fit the algorithm implementation to a concrete graphic card.
- We perform shading in the GPU shader, so we obtain a model image with per-pixel lighting.
- If the exact root finding is possible, we can increase the speed of rendering with pre-processing on CPU.

However, our method has some limitations:

- In our method, we find only the first intersection of the ray with the surface, so it is not currently possible to render functionally based models with transparent materials.
- Traditional methods of texturing will not work in the fragment shader and in CPU/fragment modifications, because of binding of texture coordinates to vertices, but we do not have vertices as the direct ray casting is performed.
- It is impossible to represent recursively defined models, because recursion is not supported by modern GPUs. Probably this possibility will appear in the next-generation GPUs.
- Modern GPUs can handle many instructions per shader. However, it still can be insufficient for very complicated models.

The removal of these limitations and further optimization of the proposed method is the subject for future research and development.

7. REFERENCES

[1] N. A. Carr, J. D. Hall, J. C. Hart. GPU Algorithms for Radiosity and Subsurface Scattering. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, July 26-27, 2003, San Diego, California

[2] M. Christen. Ray Tracing on GPU. Master's thesis, Univ. of Applied Sciences Basel (FHBB), 2005

[3] A. Corrigan, H. Quynh Dinh. Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU. International Workshop on Volume Graphics, June 2005

[4] E. de Groot, B. Wyvill. Rayskip: Faster Ray Tracing of Implicit Surface Animations. International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia - Graphite 2005, pp 31-37.

[5] M. Fox, C. Galbraith, B. Wyvill. Efficient Use of the BlobTree for Rendering Purposes. Proceedings of the International Conference on Shape Modelling & Applications, 2001, p 306.

[6] T. Foley, J. Sugerman. KD-tree acceleration structures for a GPU raytracer. Proc. SIGGRAPH/Eurographics Workshop on graphics hardware 2005, pp. 15-2

[7] Mark Harris. GPGPU: Beyond Graphics. NVIDIA GDC Presentations, 2004

[8] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In Eurographics, Blackwell Publishing, M. Alexa and J. Marks, Eds., vol. 24.

[9] J. C. Hart. Ray Tracing Implicit Surfaces. Siggraph 93 Course Notes No 25, pp 1-15.

[10] M. Hašan. An Efficient F-rep Visualization Framework. Master thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia, August 2003

[11] V. Adzhiev, R. Catwright, E. Fausett, A. Ossipov, A. Pasko, V. Savchenko. HyperFun project: Language and Software tools for F-rep Shape Modelling. Computer Graphics & Geometry, vol. 1, No 10, 1999

[12] J. Kruger, R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. Proceedings of the 14th IEEE Visualization 2003, pp 38.

[13] C. Lessig. Interaktives Ray-Tracing und Ray-Casting auf programmierbaren Grafikkarten. Bachelor Thesis, Bauhaus University Weimar, December 2004

[14] C. Loop, J. Blinn. Real-Time GPU Rendering of Piecewise Algebraic Surfaces. Proceedings of Siggraph 2006. pp 664-670.

[15] A. Pasko, V. Adzhiev, A. Sourin, V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications, The Visual Computer, vol.11, No.8, 1995, pp. 429-446.

[16] T. J. Purcell, I. Buck, W. R. Mark, P. Hanraham. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, Vol.21, Issue 3 (July 2002), pp 703-712.

[17] A. Sherstyuk. Fast Ray Tracing of Implicit Surfaces. Computer Graphics Forum, 18(2):139-147, 1999

[18] B. Shmitt, A.Pasko, V.Adzhiev, C.Schlick. Constructive texturing based on hypervolume modelling. Journal of Visualization and Computer Animation, John Wiley & Sons, Vol. 12, No. 5, 2001, pp. 297-310

[19] R. Toledo, B. Levy. Extending the graphic pipeline with new GPU-accelerated primitives. Tech report, INRIA (2004)

[20] N. A. Carr, J. Hoberock, K. Crane, J. C. Hart. Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. ACM International Conference Proceeding Series; Vol. 137, pp 203-209