Interactive ray shading of FRep objects

Oleg Fryazinov
National Centre for Computer Animation
Bournemouth University
BH12 5BB, Poole, UK
ofryazinov@bournemouth.ac.uk

Alexander Pasko
National Centre for Computer Animation
Bournemouth University
BH12 5BB, Poole, UK
apasko@bournemouth.ac.uk

ABSTRACT

In this paper we present a method for interactive rendering general procedurally defined functionally represented (FRep) objects using the acceleration with graphics hardware, namely Graphics Processing Units (GPU). We obtain interactive rates by using GPU acceleration for all computations in rendering algorithm, such as ray-surface intersection, function evaluation and normal computations. We compute primary rays as well as secondary rays for shadows, reflection and refraction for obtaining high quality of the output visualization and further extension to ray-tracing of FRep objects. The algorithm is well-suited for modern GPUs and provides acceptable interactive rates with good quality of the results. A wide range of objects can be rendered including traditional skeletal implicit surfaces, constructive solids, and purely procedural objects such as 3D fractals.

Keywords

Functional Representation, FRep, Implicit Models, Interactive, Visualization, Ray tracing, Real Time, Rendering, GPU.

1. INTRODUCTION

In this paper we deal with the most general form of function-based (implicit) geometric models called the function representation (FRep). FRep defines a geometric object by a single continuous real function of point coordinates as: $F(X) \geq 0$ [Pas95a], where the function is evaluated while traversing an underlying tree structure or by running a "black box" evaluation procedure, which makes this model cardinally different from purely analytically defined implicit surfaces. Methods of constructing such models are developed well enough; however, rendering of these models with interactive rates remains an open problem, leading to the lack of real-time modeling tools for FRep objects.

In this paper, we present a method of ray shading accelerated using graphics hardware and specialized for rendering implicit models with interactive rates. We use the term ray shading to denote the technique of rendering based on ray-casting running on GPU and extended by processing secondary rays, shadow generation, reflection and refraction with environmental mapping. The computations take part in a special GPU programs called shaders, which allows us to change models on-the-fly during the rendering process and does not limit the CPU we use. 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.

By using the acceleration on GPU, we achieve ray-tracing performance acceptable for the real-time user interaction. We do not make assumptions and do not use a priori knowledge on the object's defining function in our basic algorithm. Therefore, it can render a wide range of objects including algebraic and skeletal implicit surfaces, constructive solids, and purely procedural objects such as 3D fractals. We also present techniques for additional accelerations of the ray-tracing algorithm that allow for further improving its performance.

2. RELATED WORK

At present, there are two ways to render general implicit models. The first one is the approximation of the surface by the set of polygons, namely polygonization [Blo87a], or by a set of other easy to render primitives. However, it is memory- and computationally expensive to generate polygonal meshes in real time and moreover it is not robust, because features like spikes and sharp edges can be lost during the polygonal mesh generation.

The second way is ray-tracing which is regarded as more precise method to visualize functionally represented models, but it is even more computationally expensive to perform in real time. Ray-tracing for visualization of functionally represented models is also a well-researched area. Traditional methods of ray-tracing implicit surfaces were summarized in [Har93a]. These methods can be applied to most of functionally represented objects, but they are generally very slow even on modern hardware. Recently a number of works appear with different approaches to acceleration, such as reducing the model complexity, reducing the number of processed rays and increasing the speed of calculations.

The model complexity can be reduced by limiting the considered set of implicit surfaces by a particular type such as quadratic or higher degree surfaces and their piecewise combinations [Woo86a, Gol89a, Kan06a, Loo06a, Sto06a], blobby and other skeletal surfaces [Fox01a], or arbitrary implicit surfaces with known analytical definitions [Kno07a]. For constructive models, the complexity can be reduced by limiting a set of available operations, for example, by set-theoretic (Boolean) operations and linear transformations in Constructive Solid Geometry (CSG) models [Woo86a, Gol89a]. Another way of reducing complexity for constructive models is the simplification of the internal constructive tree structure or tree pruning [Woo80a, Fox01a].

The number of processed rays intersecting function-based models can be reduced by the adaptive subdivision in the image plain as proposed in [Has03a] or progressive refinement [Gam06a].

The speed of calculations can be increased by using specialized hardware or additional general processing or graphics processing units [Ben06a, Kno07a]. Wide development of the graphics hardware in the recent years leads to higher speed of traditional algorithms using programs for GPU. Ray-tracing on GPU is quite well investigated; however, most of papers have been focused on polygonal meshes and parametric surfaces [Pur02a, Chr05a] and volumetric data [Kru03a, Ste05a]. GPU-accelerated ray-tracing for implicit surfaces was introduced only for several particular types of surfaces. The work [Cor05a] considered ray-tracing implicit surfaces defined by radial basis functions. Rendering of quadratic implicit surfaces on GPU was reviewed in [Les04a] and later in [Sto06a], and ray-tracing of discrete isosurfaces was introduced in [Had05a].

Recently, in [Fry07a] GPU accelerated ray-casting of general function-based models was introduced, where only primary rays were processed and higher quality effects such as shadows, reflections and refractions, environmental mappings were not considered. In this work, we present ray-tracing techniques for the most general type of procedural function-based objects where primitives, operations, and the entire model are considered "black boxes" with unknown specific properties. We also consider both primary and secondary rays to achieve higher quality of rendering.

3. ALGORITHMIC BACKGROUND

In this section we briefly describe theoretical principles related to function-based geometric models and methods for ray-tracing such models.

Function representation

Geometric objects are defined in the function representation (FRep) as closed subsets of n-dimensional Euclidean space E^n with the definition

$$f(x_1, x_2, \ldots, x_n) \geq 0$$

where f is a real continuous function defined in E^n. The function can have one of several possible definitions: analytical equation, function evaluation procedure, sampled function values at regular grid nodes or scattered points and an appropriate interpolation procedure. The only requirement to the function is to have at least C^0 continuity. In 3D space, the boundary of such an object, where the function takes zero value, is a so-called implicit surface.

For application software, an FRep object is given as a “black box” procedure for the function evaluation at the given point. In the extreme case, such a procedure can be implemented from the scratch in some programming language. A procedure generating fractal objects is a good example. Another approach is to build the procedure using provided library functions for simple geometric objects (primitives) and geometric operations. Each geometric primitive is described by a concrete type of a function chosen from the finite set of such types. Some examples of primitives are: quadratic and other algebraic primitives; skeleton-based primitives; voxel array with the trilinear or higher order interpolation; solid noise; objects reconstructed from scattered surface points using radial-basis functions.

A complex geometric object is a result of applying operations to primitives. There is a rich set of operations taking functions of arguments as input and resulting in a new continuous real-valued function as output. Unary operations include space mappings - transformations of point coordinates and function mappings (offsetting, solid sweeping, and projection). Binary operations include set-theoretic operations and their blending versions, Cartesian product, metamorphosis and others.

The basic set-theoretic operations (union, intersection, difference) are implemented using Rvachev's R-functions (see [Pas95a]), which allow to represent an arbitrary constructive object by a single function. The key point in constructive modeling is that the final object is internally represented by a tree structure with primitives as leaves and operations as nodes of the tree. An FRep modeling system provides a procedure which traverses this tree structure to calculate the function value at the given point.

Ray-surface intersection for function-based models

The intersection test between a ray and an object surface is the core of the ray-tracing algorithm. The problem here is to find a ray-surface intersection point, which is nearest to the viewpoint. This problem can be reduced to zero-root finding for the function along the ray. We consider below methods that we have included in our implementation.

3.1.1 Analytical methods

The most common type of functions for analytical root finding is the polynomial function and CSG models built on polynomial primitives. For solving the equation for the defining function that represents the model, we should turn to the ray parameter space from the modeling space:

$$f(X) = 0,\ X = X_0 + t(X_1 - X_0) \Rightarrow g(t) = 0$$

Polynomial equations of degree one can be solved using the laws of elementary algebra; for polynomials of degree two the roots of the quadratic equation are known; we solve polynomials of degree three using the Cardano's method and polynomials of degree four using the Ferrari's method. If the polynomial has degree higher than four, we cannot solve it analytically and need to use approximate methods. Once the polynomial solving procedure returns all the roots including those which are negative, duplicate, and beyond the bounds of the ray, additional filtering is usually needed. If the model is represented as a CSG-tree that is built from simple primitives (i.e., the roots for them can be found analytically), we take all the roots for the leaves of the CSG-tree and select the root that corresponds to the intersection point closest to the viewer and placed on the surface of the CSG solid.

3.1.2 Interval analysis

Interval analysis for ray-tracing was introduced in [Mit90a]. The function is extended to operate on intervals for input variables using the rules of interval arithmetic. As for analytical methods we turn to the ray parameter space from modeling space. The function representation in the ray parameter space is the base for the extension to the interval function. Moore showed in [Moo66a] that the result interval F includes f results. The root finding algorithm consists in the recursive search of the interval [a, b] with different signs of a and b. This method is considered robust; however the main problem with this method is the over-conservatism as the estimated intervals are usually much wider than actual function range. Another known problem of this method is the problem with non-arithmetic operators such as conditional operators and procedural loops.

3.1.3 Approximate numerical methods

In ray-tracing of general procedural functionally represented models, which may contain conditional operators, loops, recursive calls etc, we can not use analytical methods and interval analysis is hard to implement. However, the ray-surface intersection can be found using an approximate search. First, we split up the domain into chunks and find the first one which contains at least one root, i.e., the sign of the function differs at its ends. After that, we refine the root using the regula-falsi or the Newton's method.

4. IMPLEMENTATION

In our work we employ two main features in rendering of functionally represented models. First of all, we represent a complex object by a single function and second, we perform all the computations on the GPU.

Model representation

In FRep, any object can be described by a real-valued function with real-valued arguments. A complex scene consisting of several models also can be described by a single function that describes the union of these models. This function can be either given by a text file describing a tree structure (as in BlobTree [Fox01a]) or by an evaluation procedure in a universal or a special-purpose language (HyperFun [Hf]). In this work, we use HyperFun objects as the source models, because this language can describe arbitrarily complicated FRep models. The object definition in the HyperFun language is presented as a function with input of an array of coordinate variables, an array of model parameters, and an array of attribute variables. The output of the object definition is the value of the function. Moreover, the HyperFun language allows defining the color and other photometric characteristics procedurally through the attribute variables. In fact, the model is described by a vector-function.

In our system we use the functions in the OpenGL shading language (GLSL), which we obtain using the conversion from HyperFun models. The object geometry definition in GLSL is a function with input of a vector of coordinate variables, a vector of free variables and a vector of attribute variables. Also, for shading we use the procedural color definition, which is represented as a function with the same parameters as the object geometry function, but returns color vector instead of the real value for geometry.

As HyperFun and GLSL are both C-like languages, the conversion between them can be done easily. We leave the details of the conversion between the languages beyond this paper.

Visualization process

We use GPU for the most of calculations in the ray-tracing algorithms adapted to function-based objects. As in the most of GPU-based ray-tracing methods, all the computations take place in the fragment shaders and data transfers from and to a graphics card through the textures. The main advantage of the GPU is the possibility of the shader modifying on-the-fly. Therefore it can be used for interactive rendering. The scheme of our system is shown in Fig. 1.

For our implementation of rendering, we also use GLSL. Note that we should bear in mind current GPU restrictions such as inability to use recursion or early breaks in functions, and the limit on the number of operations within one shader. Hardware restrictions depend on currently available graphics hardware and in this paper we mention restrictions that we have met during the implementation.

The conversion from HyperFun to GLSL is a part of pre-procession stage, which also includes the generation of the set of shaders based on an initial model and setting of values to the parameters to the shader, such as bounding box of the scene, time-dependent parameters and additional information if required.

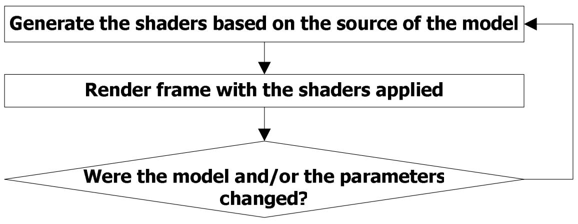

Generate the shaders based on the source of the model
Render frame with the shaders applied
Were the model and/or the parameters changed?

Figure 1. Visualization scheme.

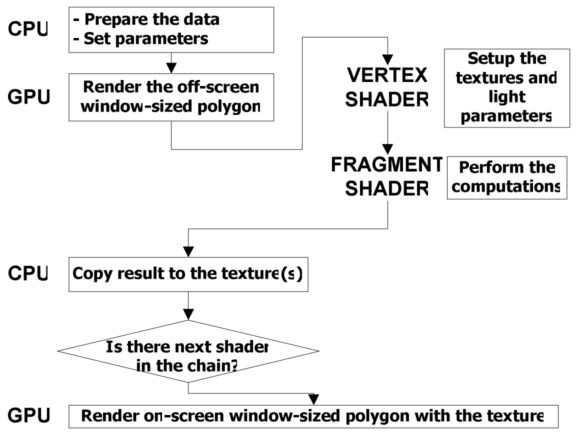

CPU
- Prepare the data
- Set parameters
GPU
Render the off-screen window-sized polygon
VERTEX SHADER
Setup the textures and light parameters
FRAGMENT SHADER
Perform the computations
CPU
Copy result to the texture(s)
Is there next shader in the chain?
GPU
Render on-screen window-sized polygon with the texture

Figure 2. Rendering process diagram.

At the stage of rendering a frame, we take the set of shaders and apply one after another to the window-sized polygon. The data is transferred between shaders through the texture memory. The first shader should provide the search for the intersection point and the last shader should have the color of the pixel as the output. The process of rendering is shown in Fig. 2.

Implementation of the ray-surface intersection on GPU

Depending on the model type we can use different methods for calculation of the ray-surface intersection point. In our implementation we use analytical methods for objects that can be represented as CSG-trees of polynomial primitives with maximum degree of four and approximate numerical methods in other cases. Also, we use interval analysis for several models.

4.1.1 Approximate root search

We use this method for complicated models, when the speed is more preferable than the quality of the image. In our current implementation we use an iterative search of the interval with different signs of the function combined with the Newton method for refining the root estimation. Thus, during the generation of the fragment shader, we add the ray-surface intersection part that finds the interval where the sign of the function differs at the ends and then refine the solution with the Newton method using the following algorithm:

- calculate the 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 origin vector of coordinates and the ray direction. However, for the primary rays input data can be reduced up to just the ray beginning vector, because the ray direction is the same for all primary rays.

4.1.2 Analytical root search

In the general case of purely procedural models exact roots cannot be found. Even a relatively simple object such as blended union between two cylinders leads to the root search for polynomials of the degree five. However, if we have a model defined as a CSG-tree over polynomial primitives of degree four and lower, we can find exact roots using analytical methods. On the pre-processing stage we generate polynomial functions for each leaf in the CSG-tree and insert this information in the shader. In this case, the root search algorithm is as follows:

- set the root found flag to false
- for each polynomial
    - calculate the roots using analytical polynomial solving
    - if there are roots in the search area $0 \le t \le 1$, select the minimal one, and set the root found flag to true
- calculate the intersection point based on the found *t* value and return the intersection point coordinates.

4.1.3 Root search using interval analysis

Although interval analysis is the most accurate way to calculate ray-surface intersection points, it is very computationally expensive even for current graphics hardware. In our implementation we use the search for the interval including a function root using dichotomy. As we cannot use recursion on GPU, we have to use a loop with a stack or a similar data structure. Moreover, at the pre-procession stage we have to include the implementation of interval arithmetic functions and interval version of the source function. In our work we use the following algorithm:

- Calculate the interval function for interval [0, 1], check the signs of the interval function, return no root if the signs are the same at the ends of the interval.
- Calculate interval functions for [0, 0.5] and [0.5, 1].
- If the signs differ for the first interval, push it to the stack with the first part flag; if the signs differ for the second interval, push it to the stack with the second part flag, otherwise return no root.
- While (stack depth more than maximum or stack is empty)
    - Pop interval and its flag from the stack
    - If interval has the first part flag, calculate the second part and push it to the stack
    - Split interval into two, calculate interval functions for both parts
    - If the signs differ for the first interval, push it to the stack, if the signs differ for the second interval, push it to the stack.
- If the stack is not empty, pop an interval from the stack, return the middle of the interval as the root.

Shading

For shading we need to have the color function applicable to any visible surface point. It means that this function should return color value for any point in the modelling space. In the general case we can evaluate the color function along with the shape defining function using a point attribute model. The methods of modeling procedural textures as point attributes are described in detail in [Shm01a]. After defining the color for the model, the shading is performed using the Phong method or a similar one. In our implementation we use the Blinn-Phong shading model.

Prior to performing shading operations, we should calculate the normal vector at the ray-surface intersection point on the surface of the FRep object. In the general case, when we consider the function of the object as the "black box", we can calculate the normal only approximately. The simplest way to obtain the normal is to apply the finite differences.

Secondary rays

To increase the quality of the visualization, we need to calculate the secondary rays for ray-tracing of the model. Due to limitation of graphics hardware such as inability to perform a recursion, we cannot use the classical recursive ray tracing algorithm. However, we can set the fixed depth for the ray processing and calculate secondary rays within this depth. For simple models, several rays can be calculated within one shader. In our implementation, we separated the rendering tasks and implemented them as several shaders. The first shader should calculate the first intersection. After that, we apply secondary ray shaders and in the last shader, that should perform surface shading, we sum up all the information. Depending on the shader type, information between shader passes is transferred either as one four-component vector (three components for the new ray direction and one for the root from the previous ray-surface intersection in the ray parameter space) or as two four-component vectors with entire information from the previous intersection.

5. EXPERIMENTAL RESULTS

We tested our method by rendering several function-based models with different degrees of complexity (Fig. 3). In the performance results shown below, we use models from the Virtual Shikki project and models from the HyperFun gallery including procedural fractal models.

Performance characteristics of our implementation were measured on a PC with a two SLI-combined NVIDIA GeForce 7900 cards and an Intel Pentium 4 3.20GHz CPU. All models were rendered on a 256X256 viewport. For comparison with CPU-only methods, we have also measured speed characteristics of a software implementation of our method on the same processor. We provide the results in the following table, where the performance is measured in frames per second (bigger fps means higher rendering speed). The Cup model was rendered using the analytical ray-surface intersection method; the Noise model was rendered using interval analysis. Other models were rendered using the approximate ray-surface intersection method.

	GPU (P)	GPU (S)	CPU
Cup	120	60	0.41
Metamorphosis	30	20	0.3
3D fractal	17	8	0.12
Noise with CSG	60	30	0.35
Virtual shikki	4	3	<0.01

Table 1. Performance characteristics for selected models. GPU(p) denotes using of only primary rays on GPU, GPU(s) denotes using of secondary rays as well as primary rays, CPU denotes implementation on CPU.

It can be seen from the table that with GPU-based rendering we can achieve interactive visualization of functionally based models and scenes. By defining the function in the shader we can change the body of the function in real-time. We can visualize interactively not only simple implicit objects, but also the procedural objects, such as procedurally defined 3D fractals and dynamic objects in real-time (Fig. 4). In the case when the speed is more important that the quality, we can use approximate methods and obtain interactive rates even for complicated scenes. Also, if the model represents a CSG-tree with polynomial primitives of degree four and less, we can use analytical methods for rendering, and obtain better quality of the visualization with better speed.

In our tests we use not only primary rays for rendering, but secondary rays also. This allowed us to obtain interactive rates for functional-based scenes with such effects as shadows, reflection and refraction (Fig. 5). Also, the procedural shading and texturing can be used for functionally-based scenes (Fig. 6).

6. CONCLUSION

In this paper we presented a method of high-quality visualization of general procedural function-based (implicit) models with GPU-accelerated ray tracing. It was shown that we obtain the good quality of visualization along with good interactive rates by representing scene by a single function and by processing it using programs for GPU. In our work we presented different algorithms for intersecting a ray with a function-based model, and in our experiments different methods were tested for different objects. Depending on the algorithm, we can obtain higher speed with lower quality, however for several models higher speed can be obtained using analytical methods. In addition, in this work we process secondary rays in the ray-tracing procedure that allows us to obtain better image quality than by pure ray-casting.

However, our method has some limitations, most of them related to restrictions of the current graphics hardware. The first limitation is restriction on the program size and instructions number. We decrease the quantity of calculations using separation over the several shaders. However, sometimes the problem of precision appeared, because there is the lack of precision during data transfers from and to the texture memory. Also, recursively defined models and models that require dynamic arrays can not be converted to current graphics hardware without the substitution to conditional operators and static arrays that is not always can be made easily. The removal of these limitations and further optimization of the proposed method are the subjects for future research and development.

7. REFERENCES

[Adz99a] Adzhiev, V., Catwright, R., Fausett, E., Ossipov, A., Pasko, A., Savchenko, V. HyperFun project: Language and Software tools for F-rep Shape Modelling. Computer Graphics & Geometry, vol. 1, No 10, 1999.

[Ben06a] Benthin, C., Scherbaum, M., and Friedrich, H. Ray Tracing on the CELL Processor. In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp 15–23.

[Blo87a] Bloomenthal, J. Polygonization of Implicit Surfaces. Computer Aided Geometric Design, vol. 5, pp 341-355, 1988.

[Chr05a] Christen, M. Ray Tracing on GPU. Master's thesis, Univ. of Applied Sciences Basel (FHBB), 2005.

[Cor05a] Corrigan, A., Quynh Dinh, H. Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU. International Workshop on Volume Graphics, June 2005.

[Fox01a] Fox, M., Galbraith, C., Wyvill, B.. Efficient Use of the BlobTree for Rendering Purposes. Proceedings of the International Conference on Shape Modelling & Applications, 2001, p 306.

[Fry07a] Fryazinov, O., Pasko, A. GPU-based real time FRep ray casting. In Proceedings of Graphicon-2007, pp. 69-74.

[Gam06a] Gamito, M., Maddock, S. A Progressive Refinement Approach for the Visualisation of Implicit Surfaces. Proc. 1st International Conference on Computer Graphics Theory and Applications (GRAPP 2006), Setúbal, Portugal, 25-28 February 2006, pp. 26-33

[Gol89a] Goldfeather, J., Monar, S., Turk, G., Fuchs, H.. Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. IEEE Comput. Graph. Appl. Vol. 9, No 3, 1989, pp 20-28.

[Had05a] Hadwiger, M., Sigg, C., Scharsach, H., Bühler, K., Gross, M. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In Eurographics, Blackwell Publishing, M. Alexa and J. Marks, Eds., vol. 24.

[Har93a] Hart, J. C. Ray Tracing Implicit Surfaces. Siggraph 93 Course Notes No 25, pp 1-15.

[Has03a] Hašan, M. An Efficient F-rep Visualization Framework. Master thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia, August 2003.

[Kan06a] Kanai, T., Ohtake, Y., Kawata, H., Kase, K. GPU-based rendering of sparse low-degree implicit surfaces. In Proceedings of the 4th international Conference GRAPHITE '06. ACM Press, New York, NY, 165-171.

[Kno07a] Knoll, A., Hijazi, Y., Hansen, C., Wald, I., Hagen, H. Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic. Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing, Ulm, Germany, 2007, pp. 11-17.

[Kru03a] Kruger, J., Westermann, R. Acceleration Techniques for GPU-based Volume Rendering. Proceedings of the 14th IEEE Visualization 2003, pp 38.

[Les04a] Lessig, C. Interactive Ray Tracing and Ray Casting on Programmable Graphics Hardware. Bachelor Thesis, Bauhaus University Weimar, December 2004.

[Loo06a] Loop, C., Blinn, J. Real-Time GPU Rendering of Piecewise Algebraic Surfaces. Proceedings of Siggraph 2006. pp 664-670.

[Mit90a] Mitchell, D. P. Three applications of interval analysis in computer graphics. In Frontiers in Rendering course notes, pages 14-1 - 14-13. SIGGRAPH'91, July 1991.

[Moo66a] Moore, R.E., Interval Analysis, Prentice-Hall, New York, 1966.

[Pas95a] Pasko, A., Adzhiev, V., Sourin, A., Savchenko, V. Function representation in geometric modeling: concepts, implementation and applications, The Visual Computer, vol.11, No.8, 1995, pp. 429-446.

[Pur02a] Purcell, T. J., Buck, I., Mark, W. R., Hanraham, P. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, Vol.21, Issue 3 (July 2002), pp 703-712.

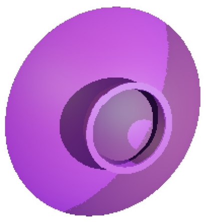
[Shm01a] Shmitt, B., Pasko, A., Adzhiev, V., Schlick, C. Constructive texturing based on hypervolume modelling. Journal of Visualization and Computer Animation, John Wiley & Sons, Vol. 12, No. 5, 2001, pp. 297-310.

[Ste05a] Stegmaier, S., Strengert, M., Klein, T., and Ertl, T. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In Volume Graphics, pages 187–195, 2005.

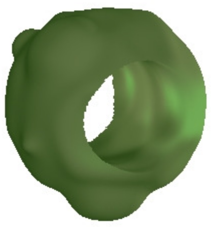
[Sto06a] Stoll, C., Gumhold, S., Seidel, H. Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU. In: Proc. IEEE Symposium on Interactive Raytracing, 141-150, 2006

[Woo80a] Woodwark, J., Quinlan, K. The derivation of graphics from volume models by recursive division of the object space, Proceedings of the Computer Graphics 80 Conference, Brighton, UK, pp 335-343.

[Woo86a] Woodwark, J., Bowyer, A. Better and faster pictures from solid models, Computer-Aided Engineering Journal 3,1, pp 17-24, February 1986.



$$\begin{aligned} f &= ((\mathit{sphere}\ 1 \setminus \mathit{sphere}\ 2) \setminus y) \mid \\ &((\mathit{conus}\ 1\ \&\ (-y-1.95)\ \&\ (y+2.6)) \\ &\setminus \mathit{conus}\ 2), \\ \mathit{sphere}\ 1 &= 33.18 - x^2 - y^2 - z^2 \\ \mathit{sphere}\ 2 &= 30.91 - x^2 - y^2 - z^2 \\ \mathit{conus}\ 1 &= \frac{(y-5.4)^2}{4} - x^2 - z^2 \\ \mathit{conus}\ 2 &= \frac{(y-4.2)^2}{4} - x^2 - z^2 \end{aligned}$$



$$\begin{aligned} f &= (81 - x^2 - y^2 - z^2 + \mathit{noise}) \setminus (1 - y^2 - z^2), \\ \mathit{noise} &= (1.8 * \sin(x) + sx) * \\ &(1.8 * \sin(0.9 * y) + sy) * (1.8 * \sin(0.9 * z) + sz), \\ sx &= 1.8 * \sin(\sin(x)/1.35 + 1.4 * \sin(0.9 * z)), \\ sy &= 1.8 * \sin(\sin(0.9 * y)/1.35 + 1.4 * \sin(x)), \\ sz &= 1.8 * \sin(\sin(0.9 * z)/1.35 + 1.4 * \sin(0.9 * y)) \end{aligned}$$

Figure 3. Various FRep models rendered with our method w, from left to right: cup (from virtual shikki project) with self-shadows, solid noise with CSG-hole (formula that describes the object is given at the right side).

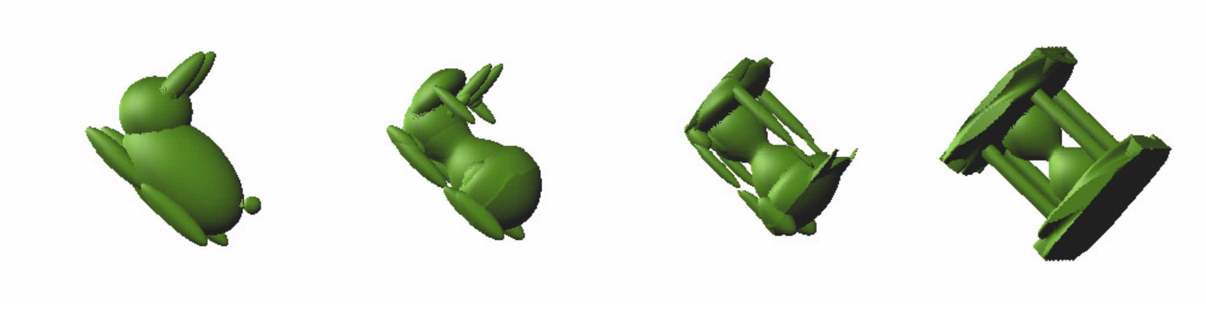


Figure 4. Rendering a dynamic model: metamorphosis from a rabbit to a sandbox (models from HyperFun gallery)

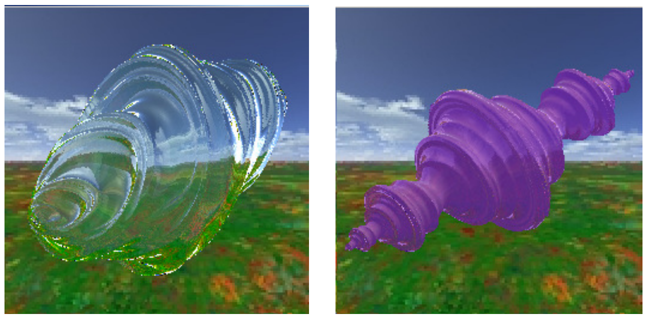


Figure 5. Procedural 3D fractals (models available in the HyperFun gallery, courtesy of F. Delhoume) rendered with reflection and refraction (the textured box is used as an environment).

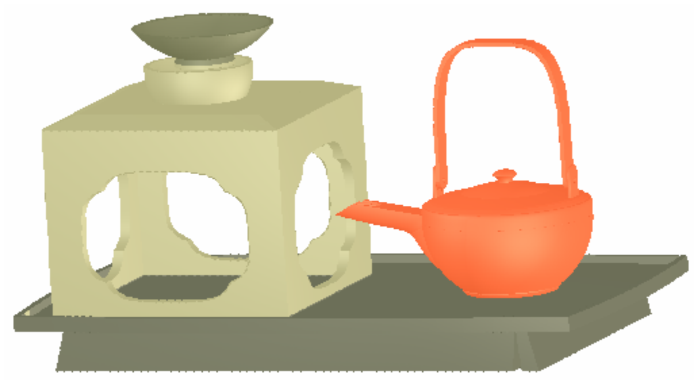


Figure 6. Virtual Shikki: real-time (4 frames per second) rendering of a functionally based scene with procedural shading.