Thickness computation of trimmed B-Rep model using GPU ray tracing

G. Lemasson[1,2,3], J.C. Iehl[2], F. Zara[1], B. Shariat[1], V. Baudet[3], P. Arthaud [3]
[1]Université de Lyon, CNRS, Université Lyon 1, LIRIS, SAARA team, UMR5205, F-69622, French
[2]Université de Lyon, CNRS, Université Lyon 1, LIRIS, R3AM team, UMR5205, F-69622, French
[3]CT CoreTechnologie, F-69007, French

Abstract

This paper demonstrates the use of direct ray tracing of large industrial CAD models on the GPU. The ray tracing kernel is a building block used to compute and assess the validity of the mechanical design of CAD models. A high precision and efficient solution to handle trimmed surfaces with holes is discussed. The central idea leverages Bézier curve properties to build a fast, robust and stable Newton iteration. Moreover, two GPU implementations are presented: one using a BVH, the other using a "Divide and Conquer" method. We compare both methods with a reference implementation running on the CPU. We demonstrate that the new methods use less memory than the current reference method, and that the method using BVH on the GPU is faster and more accurate.

CR Categories: I.3.7 [Computer Graphics]: Raytracing— [I.3.1]: Computer Graphics—Parallel processing I.3.1 [Computer Graphics]: Graphics processor— [I.3.5]: Computer Graphics—Curves, surface, solid, and object representation;

Keywords: Thickness, B-Rep, Ray tracing, Parametric surface, Trimming, Newton iteration

1 Introduction

Computer Aided Design (CAD) is very important in the prototyping process in mechanical industries such as automotive industry or aeronautics. Generally, engineers use specialized softwares (Catia, PRO Eng, NX, ...) to design the manufactured objects shape, using frequently the B-Rep model. Indeed, this model offers the best compromise between the design and mechanical constraints.

The main entities of B-Reps are faces composed by a surface defined by parametric equation. These faces are bounded by trimming curves including loops defined in the parametric domain of the surface. These loops are divided into two categories: external loops describing shape, and internal loops describing holes. The trimmed parametric surfaces could be planar, spherical, or usually NURBS.

Many model interrogation tools of CAD softwares require a high precision of the computations. One of these interrogation tools is the thickness computation, used to detect errors such as large differences of thicknesses between several branches of the B-Rep model. Indeed, these differences can cause weakness or fragility. Usually the current implementation ray traces a triangular tessellation of NURBS surfaces potentially leading to important precision losses. Thus, the triangular tessellation must be refined, increasing considerably the data size and the computation time. Moreover, most of interrogation tools do not benefit yet the possibilities offered by new massively parallel hardware architectures (GPU). Consequently, our aim is to adapt the thickness checker tool on the GPU, to obtain high precision results at an acceptable computation time.

In this paper, we present a highly efficient thickness computation of parametric B-Rep models. Our main idea is to directly ray trace the parametric surfaces to reduce the memory consumption and to improve the precision of distance computations. Surface patches and trimming curves are converted to bicubic patches and cubic Bézier curves to get a GPU friendly data layout. This conversion also helps in computing robust intersections and trimming tests. In this context, we apply and compare two methods: one using a BVH acceleration structure, which uses more memory and incoherent traversal; and the other one using the "Divide and Conquer" paradigm, which uses more memory but a more regular execution on the GPU.

Our main contributions are: a new method for direct trimming of parametric surfaces on the GPU using Newton iterations and a comparative study of two implementations for ray tracing of parametric surfaces on GPU in the context of an industrial CAD modeling tool.

2 Related work

There are many approaches to compute a thickness as ray tracing or skeletonization. But, as this latter consumes too memory, we will focus only on the ray tracing approach in our work.

Ray tracing on tessellation. The thickness can be approximated by a dense discretization of the surfaces and using a ray tracing per vertex to evaluate the distance to the opposite side.

Ray tracing triangle meshes is a well studied problem as surveyed by Wald *et al.* [2001; 2009]. Efficient implementations on the GPU are also largely discussed [Aila and Karras 2010; Hapala et al. 2011]. But for accurate thickness computation, a dense discretization is required making this approach impractical.

Ray tracing parametric patches. Direct ray tracing of parametric surfaces appears to be a better approach. This is the most accurate way to obtain information on parametric surfaces, and with less memory usage because several hundreds of triangles are reduced to one parametric patch. Kajiya [1982] was the first to introduce parametric surface ray tracing. The main idea is to convert the bivariate equations of parametric surfaces into uni-variate polynomial of higher degree. For example, a bi-cubic Bézier surface becomes a polynomial of degree 18. But computing roots of high degree polynomials can be numerically instable.

Nishita [1990] presented the Bézier clipping method as a robust root-finding algorithm. Bézier clipping can be used for both ray-surface and ray-curve intersection computation. But, the recursive

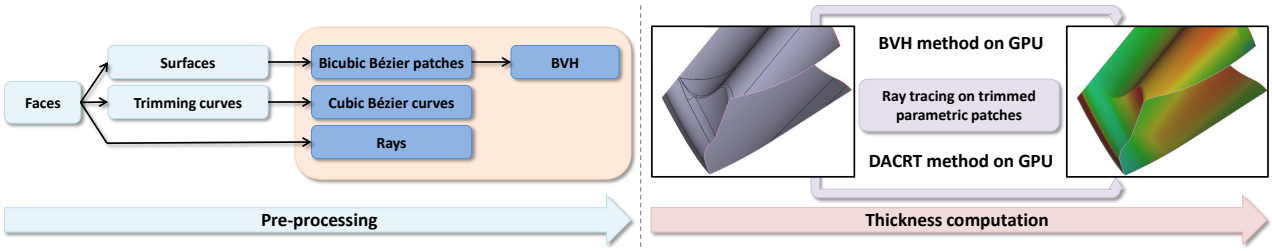

Figure 1: *Overview of our method. The pre-processing step is the same for the two methods implemented (BVH and DACRT).*

nature of the method makes is implementation difficult on the GPU. Efremov [2005] performed a robust and numerically stable Bézier clipping on NURBS. Similarly, Benthin [2004] used the recursive Loop subdivision for computing a ray/surface intersection.

Toth [1985] introduced a new approach, using Newton iteration to find a ray/surface intersection which has the advantage of being general enough to handle any parametric surface. However, it requires a good initial value to ensure correctness and fast convergence. Geimer *et al.* [2005; 2006] performed a ray tracing of bicubic Bézier patches and NURBS on CPU. They subdivide surfaces into mostly flat patches and ensure a fast and correct convergence of the Newton iteration. Pabst *et. al.* [2006; 2009] performed a ray tracing of NURBS on the GPU. They used the graphics pipeline to generate rays and compute intersection in a fragment shader.

Trimming. Surfaces are trimmed in their parametric space by curves. Consequently an intersection can be found outside the surface or inside a hole. Discretizing trimming curves [Balázs et al. 2004] or their parametric domain [Guthe et al. 2005] requires a dense sampling to be precise and thus, high memory consumption.

Claux *et al.* [2012] were interested in ray / surface intersection and offered a solution to the trimming problem by transforming a cubic Bézier curve into an implicit form [Loop and Blinn 2005] and stored it in a KD-tree. As this method is direct, *i.e.* does not need iterations or recursions to determine the orientation of the curve, it can be implemented on the GPU. But the transformation of a bicubic Bézier surface to an implicit form provides an approximation of the initial shape.

The Bézier clipping method for the trimming curve is widely used in the literature [Nishita et al. 1990; Geimer and Abert 2005; Abert et al. 2006; Schollmeyer and Fröhlich 2009]. But this technique remains a recursive method. Pabst [2006] provided an iterative formulation of the originally recursive Bézier clipping and executed it on the GPU. Schollmeyer *et al.* [2009] improve on Pabst's method by adding a better management of trimming curves. They split the curves into monotonic segments and use an acceleration structure. But this method needs a large pre-processing step and creates a large number of curves.

Acceleration structures. To reduce the number of ray / surfaces intersection, an acceleration structure can be used. As a construction of BVH does not need splitting the patches, the BVH is the more suitable to handle patches. [Aila and Karras 2010] proposed a hierarchical treelet subdivision of the acceleration structure for a massively parallel hardware architecture. [Wald 2007; Stich et al. 2009] used the SAH heuristic to build a BVH. Lauterbach [2009] offered a fast construction of BVH using GPU. Karras *et al.* [2012] maximized the parallel construction for several acceleration structures (BVH, KD-tree and Octree). By construction, the traversal of BVH is recursive, but [Hapala et al. 2011] proposed a simple stack-less traversal.

We can note that [Áfra 2012; Mora 2011] offered a ray casting without an acceleration structure, but using the "Divide and Conquer" paradigm. We can note that this method uses more regular execution on the GPU, but requires more memory than the BVH.

3 Our approach

We present an overview of the different steps of our approach in Fig. 1. We start by a pre-processing each face of our parametric model:

- First, we subdivide each surface represented by NURBS into several C^2 continuous bicubic Bézier subpatches, and due to the convex hull property of Bézier surfaces, we use their control points to determine an axis-aligned bounding box.
- Then, we convert the trimming curves, originally represented as NURBS, into piecewise cubic Bézier curves for our trimming test.
- We compute a BVH of bicubic Bézier patches using a Morton key, according to the centroid of axis-aligned bounding box of patches [Lauterbach et al. 2009; Karras 2012].
- Rays are sampled on the surface point and to the opposite of the normal surface.

After this pre-processing step, we compute the thickness of the B-Rep model, by performing rays / patches intersection on the GPU. We have implemented two methods: the first one uses the BVH directly on the GPU, and the second one uses the "Divide and Conquer" paradigm inspired from [Áfra 2012; Mora 2011] (denoted DACRT).

Moreover, as Newton iteration have been used to compute the intersection between a ray and a patch, and trimming is managed using the Newton iteration, we will also discuss about these two methods in the following sections.

3.1 Pre-processing

The purpose of our pre-processing step is to prepare data for a better convergence of the Newton iteration and to memory layout. We can note that Newton iteration has quadratic convergence if the initial values are near a root.

Surfaces. We first convert the surfaces represented by NURBS into several bicubic Bézier patches with a tolerance of 10^{-4}. This conversion has two goals: (1) The former is to reduce a parametric subspace into several smaller parametric spaces and to ensure a better convergence for Newton iteration. (2) The second is to homogenize the data for the GPU. This homogenization ensure an similar execution on each thread on the GPU and a regular access of the memory.

To perform this conversion, the aligned-axis bounding box of each bicubic Bézier patch has been computed using their control points. Thereafter, the bounding boxes will be used to build a BVH, or to check if a ray can cross a patch.

Trimming curves. The trimming curves are also converted. Initially, they are represented as NURBS. We split the curve to ensures better convergence, and we convert these NURBS into piecewise cubic Bézier curves according to the inflection points. All curves are converted in cubic Bézier curves to homogenize the execution and the access of memory on the GPU.

We can note that the subdivision of NURBS surfaces generates several Bézier patches, but the trimming does not affect all the patches. To improve preprocessing time and memory consumption, we cull subpatches located in the holes of the surface and outside the surface.

3.2 Thickness computation

BVH method on the GPU. As we build the BVH only for one thickness computation and not for rendering, we use a fast construction on the GPU [Lauterbach et al. 2009; Karras 2012] in despite of the SAH heuristic [Wald 2007; Stich et al. 2009] or the treelet subdivision [Aila and Karras 2010].

The BVH is computed with a Morton key according to the center of axis-aligned bounding box of patches [Lauterbach et al. 2009; Karras 2012]. For each patch, we compute in parallel a Morton key and we sort them according to the value of the Morton key: that is each left child corresponds to bit 0 and right child to bit 1.

At the end of the construction of the BVH, each node contains the bounding box of two children and each leaf contains the index of the first and the last patch.

Moreover, the traversal of a BVH is a recursive algorithm, we use a simple stack-less BVH traversal [Hapala et al. 2011].

Then, to manage the ray tracing using the BVH, we proceed as following. When a ray crosses a node, it first checks the bounding box of the nearest child and continues until the ray crosses a BBox or when it reaches a leaf. When a ray crosses a leaf, it checks for a potential intersection and also test equally the trimming curves of for all patches referenced by leaf.

DACRT method on the GPU. We have also implemented a "Divide and Conquer" ray tracing inspired by [Áfra 2012; Mora 2011]. For this, we consider two sets: the set of patches and the set of rays. The set of patches is split in two according to the maximal axis of the bounding box.

Then, all rays are checked with one subset and we use an in-place partition of the set of rays. The rays are organized according to the crossing bounding box of patches subset: the rays that cross bounding box of this subset are stored at the begin of set, the other at the end.

This subdivision is repeated until the size of the set of patches and the set of rays are too large. When these two subset are small enough, a *brute force* method is applied, *i.e.* each ray checks an intersection, and the trimming with each patch. After this, we go back and process another subset of patches.

3.3 Ray / patch intersection

The core of the intersection test is similar to the approach presented by Geimer *et al.* [Geimer and Abert 2005].

We represent a ray by two arbitrary chosen orthogonal planes $P_1 = (N_1, d_1)$ and $P_2 = (N_2, d_2)$, with N_1 a normal to P_1 and N_2 a normal to P_2, and d_1, d_2 the fourth parameter of plane equation.

The intersection between a ray and a patch is the set of points belonging to the patch, which also verify the plan equations P_1 and P_2. Consequently, to find the intersection point between the ray and a parametric surface $S(u, v)$ defines a Bézier patch, it defined by

$$S(u,v) = \sum_{i=0}^{3} B_i(u)^3 B_j(v)^3 p_{ij}, \tag{1}$$

with $B_i(u)^3, B_j(v)^3$ the Bernstein polynomials, p_{ij} the controls points, we have to solve

$$R(u,v) = \begin{bmatrix} N_1.S(u,v) + d_1 \\ N_2.S(u,v) + d_2 \end{bmatrix}. \tag{2}$$

The Newton iteration is used to find the u, v parameters. If we note S_u, S_v the partial derivative of the parametric surface in the corresponding parametric directions, and u_n, v_n the parametric results for the n^{th} iteration of Newton, we have

$$\begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} u_n \\ v_n \end{bmatrix} - J^{-1} \cdot R(u_n, v_n) \tag{3}$$

where J is the Jacobian matrix defined by

$$J = \begin{bmatrix} N_1 \cdot S_u(u,v) & N_1 \cdot S_v(u,v) \\ N_2 \cdot S_u(u,v) & N_2 \cdot S_v(u,v) \end{bmatrix}. \tag{4}$$

We can note that the getting of the parameters u, v enables to compute the real intersection point, and the distance between the origin of the ray and the parametric surface.

The use of 32 bits float on GPU can cause a numerical instability. To prevent this, we choose to use the De Casteljau algorithm rather than Bézier formula to compute a point on curves or on a patch. Indeed, the De Casteljau algorithm uses divisions by two making this operation exact for the float. Moreover, to expand the margin of the error, we apply a scale factor of 10,000 to the parametric space of the patches. On average a 32 bits float has 7 decimal digits, but using a scale of 10,000 for values in $[0, 1]$ can add 5 more decimal digits.

Moreover, to ensure a fast and correct convergence, we must initialize correctly u_0, v_0 of the Newton iteration. Before compute the intersection of the surface, we compute the intersection with their control polyhedron. The intersection is determined on a square of control point. For example, if the intersection is discovered between the controles points $p_{1,2}$, $p_{2,2}$, $p_{1,3}$ and $p_{2,2}$, the parameter u_0 will be interpolated between $\frac{1}{3}$ and $\frac{2}{3}$, and the parameter v_0 will be interpolated between $\frac{2}{3}$ and 1.

To ensure the same execution on each thread on the GPU, we apply the same number of iterations: seven iterations are required for a good convergence. Then, there are three potential outcomes: (1) no-convergence, (2) convergence out of parametric space, (3) convergence. A no-convergence is detected when $\|R(u,v)\| > \epsilon$, where ϵ is an user defined threshold.

We can note that the Newton iteration have a quadratic convergence. At the end of the iterations, the precision of the calculated intersection joint is limited by the precision of float numbers, *i.e.* approximatively 10^{-5} [Dammertz and Keller 2006].

3.4 Trimming

When a ray find an intersection with the surface, it must check if it occurs in the interior of the trimming curves defined by this test reduces to check if the intersection is on the left side of the curve. We note p the intersection between a ray and the parametric surface, and $p_c = c(t_p)$ the projection of p on the nearest trimming curve $c(t)$. If $(\overrightarrow{pp_c} \times c'(t_p))_z < 0$, where $c'(t)$ is the first derivative of $c(t)$, and p_c the nearest point on $c(t)$, the intersection is in the interior of the trimming (see Fig. 2). We can note that, due to the orientation of curves, this approach is correct both for external and internal curves.

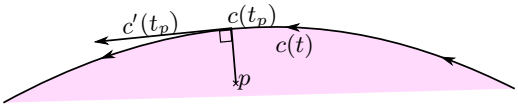

Figure 2: *Intersection point p inside trimming with $(\overrightarrow{pp_c} \times c'(t_p))_z < 0$.*

To determine the projection p_c, we write that it is the nearest point to $c(t)$ when $\overrightarrow{pp_c} \cdot c'(t_p) = 0$. For this, we use Newton iteration with [Schneider 1990]:

$$t_{n+1} = t_n - \frac{(c(t_n) - p) \cdot c'(t_n)}{c'(t_n)^2 + c''(t_n) \cdot (c(t_n) - p)} \quad (5)$$

where t_n is the approximation of the parametric coordinate t_p at the n^{th} iteration, and $c''(t)$ is the second derivative of $c(t)$. Only three iterations are required to ensure a good convergence with cubic Bézier curves.

Nearest Curve. Processing all trimming curves is time consuming and not accurate. Consequently, we propose to find the nearest curve before running the Newton iteration to find the nearest point. For this, we cut the curve $c(t)$ in three linear segments, and we look for the nearest. The nearest curve $c(t)$ is the curve whose segment $\overline{c(t)}$ is the nearest of p. Fig. 3 illustrates the computation of the nearest curve: the nearest curve is $c_a(t)$, because the segment $\overline{c_a(t)}$ is the nearest to p.

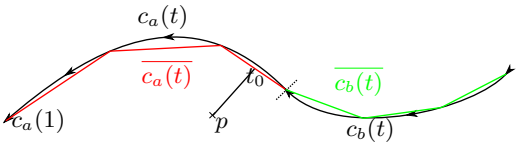

Figure 3: *Two cubic Bézier curve $c_a(t)$ and $c_b(t)$, and respectively the sets of segments $\overline{c_a(t)}$ and $\overline{c_b(t)}$. The projection of p on $\overline{c_a(t)}$ defines the initial value t_0 of Newton iteration parameter.*

Initial value. The segmentation of curves also ensure a correct and fast convergence. Indeed, if a bad initial value is chosen, Newton iteration will not converge. The initial value t_0 should be as close as possible to the real root. We choose t_0 according to $\overline{c(t)}$. Consequently, at the same time that we search the nearest curve using $\overline{c(t)}$. A linear interpolation is used to compute t_0.

Dead angle. Consecutive trimming curves along the same loop are generally C^1 or C^2 continuous, but a continuity break can appears (see Fig. 4). This case appears when, for two consecutive trimming curves $c_a(t)$ and $c_b(t)$, their first derivative at their junction points are not collinear. We call this zone a *dead angle*.

If the intersection p is in dead angle, p is nearest to a curve $c_a(t)$ at the point $c_a(1)$, and p is nearest to the curve $c_b(t)$ at the point $c_b(0)$, but there is no possible convergence for Newton iteration.

To solve this problem, two categories of dead angles exist: concave dead angle and convex dead angle (see Fig. 4). When $c'_a(1) \times c'_b(0) > 0$, p is inside a concave dead angle and consequently within the trimmed zone (Fig 4-a). Reciprocally, if $c'_a(1) \times c'_b(0) < 0$, p is inside a convex dead angle and consequently outside the trimmed patch (Fig 4-b).

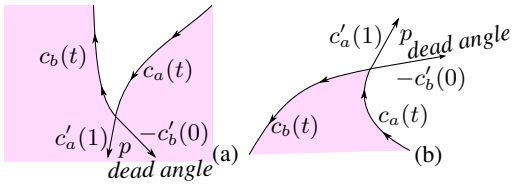

Figure 4: *Illustration of dead angle: a) Concave or b) Convex. The colored zone is located within the trimmed patch.*

4 Results

Thereafter, we present the performance of our ray tracing system for two different techniques (BVH and "Divide and Conquer") measured on a Nvidia GeForce GTX 580 M implement with OpenCL. We compare our results with the software thickness analysis tool. The software platform uses CPU ray tracing of a tessellated geometry. A second method improves the method on CPU and consists on a parallelized ray tracing on CPU. This latter use the Intel Threading Build Block library and tests are performed on Intel core i7-280Qm 2.30 Ghz using double precision numbers.

Our tests are carried out on a part of the geometrical model of the rear light of a car, called *Lens* (Fig. 5), as well as on a model called *Mask* (Fig. 6). Respectively, these models contain 88,581 patches and 27,188 trimming curves; 29,379 patches and 12,793 trimming curves (Fig. 7).

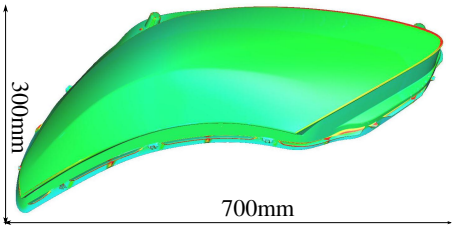

Figure 5: *The Lens model after thickness computing. Blue represents small distances, and red for large thicknesses.*

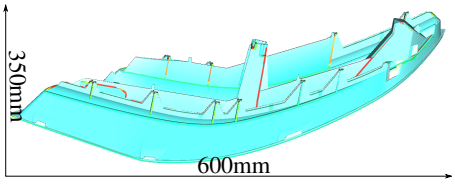

Figure 6: *The Mask model after thickness computing. Blue represents small distances, and red for large thicknesses.*

We ran some experiments to choose the stop criterion for DACRT. Fig. 8 and 9 show the different computation times according to the size of the both buffer for the Lens model.

Fig. 10 shows the performance of different methods including pre-processing time. We can observe that the BVH method is the fastest, especially when the sampling of rays is large. However the ratio between the different methods depends of the model. On average the CPU parallelized version is 4.25 times faster than the serial method on CPU for the Lens model. It is 4.8 faster for the Mask model. The method using the BVH is the fastest method. This speed is not significant when the number of rays is low, but it increases significantly when the number of rays is large.

	Lens	Mask
# bicubic patches	88,581	27,188
# cubic trimming curves	27,793	12,793
# intersections tests / rays (BVH)	23	47
# intersections tests / sec (BVH)	2,874,122	3,312,760
Time intersections (ns)	0.348	0.301
Time pre-processing (s)	69.5	43.8

Figure 7: *Information about model. The intersection time includes the trimming test.*

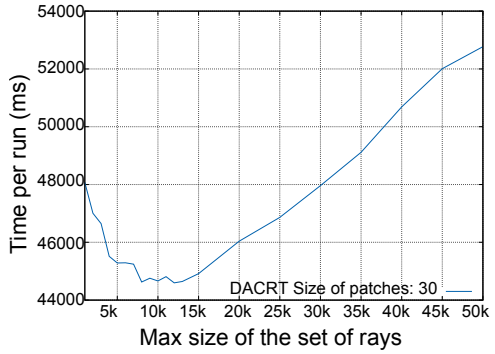

Figure 8: *Computing time for the "Divide and Conquer" method according to the maximum size of rays set for the Lens model. The size of patches set is fixed to* 30.

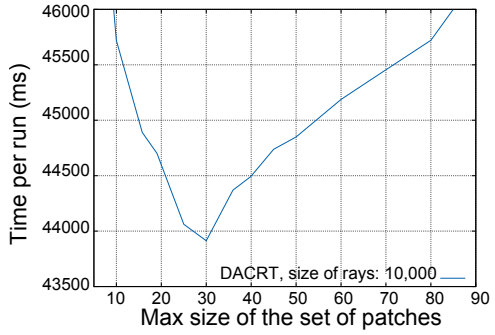

Figure 9: *Computing time for the "Divide and Conquer" method according to the maximum size of patches set for the Lens model. The size of the rays set is fixed to* 10,000.

Moreover, the memory consumption of BVH method (Fig.11) is less than the memory consumption of the actual implemented software (Fig. 12). With the BVH method, the rays are generated on the fly, but for the DACRT all rays must be stored. Fig. 12 shows the memory consumption of the DACRT method. Although both GPU methods consume less memory than CPU methods, the method with the BVH is even more interesting because it uses less memory and this consumption of memory is constant whatever the number of rays.

# rays Lens	BVH	DACRT	# rays Mask	BVH	DACRT
2.11 M	4.52	3.42	1.75 M	3.94	3.26
3.78 M	11.45	7.71	2.23 M	8.93	6.93
4.02 M	11.77	7.89	3.29 M	9.43	7.61
4.84 M	14.17	9.09	4.10 M	12.88	9.98
7.50 M	19.06	10.79	8.20 M	27.89	17.81

Figure 10: *Performance comparison for Lens and Mask models with pre-processing. Ratio between CPU and GPU based methods.*

Model	BVH	Patches	Trimming curves	Total
Lens	12.94	23.50	1.45	37.90
Mask	5.38	9.69	0.69	15.76

Figure 11: *Memory consumption in Mb for the method with BVH.*

# rays Lens	CPU Mem.	DACRT Mem.	# rays Mask	CPU Mem.	DACRT Mem.
2,11 M	1615.0	773.1	1.75 M	1341.6	642.3
3.78 M	2898.0	1386.1	2.23 M	1708.3	817.7
4.02 M	3079.2	1473.1	3.29 M	2519.7	1205.4
4.84 M	3712.4	1775.8	4.10 M	31368.3	1501.8
7.50 M	5749.6	2749.4	8.20 M	6286.3	3005.7

Figure 12: *Memory consumption in Mb for both methods.*

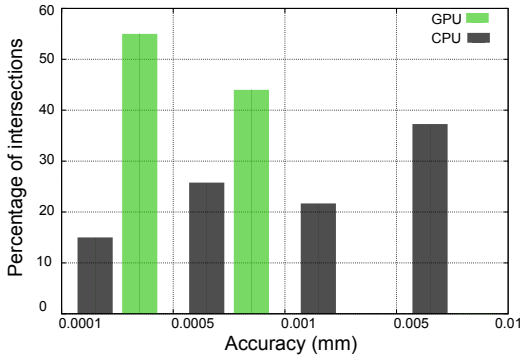

Figure 13: *Comparison of the accuracy in mm between the both GPU implementations and the CPU implementations.*

Our new methods are thirty times more accurate than the current implementation. Fig. 13 illustrates the precision histogram. We obtain these results by comparing GPU and CPU method with the same rays. For each method, we use the theoretical u and v parameters to compute the real intersection with the original NURBS surface. With the CPU method only 15 percent of the points are computed at a precision of one thousandth of millimeter, despite of 64 bits floats used. Although with the GPU method 55 percent of the points are calculated with the same precision, using only 32 bits floats.

5 Conclusions and perspective

In this paper, we have presented a ray casting, usually used for the rendering, to compute a thickness within B-Rep model. We improved the current methods used in our software platform and proposed a new method to manage trimming curves.

Compared to the current CPU method, we are faster, more accurate and we consume less memory. We are more accurate because the thickness is computed on parametric patches and not on an approximation using tesselation resulting in several millions triangles. Consequently, the memory consumption is also reduced. The performance is also increased by the use of the GPU. However, at the first step of our approach a sampling of the parametric patches is used to create rays.

We also presented a novel method for curved regions with holes and applied this technical for direct trimming of parametric surfaces. We used this method on cubic Bézier to ensure a good convergence and a homogeneous process on the GPU. This method can be applied to all parametric curves and it is fast and does not need a complex pre-computation.

In future work, we will reduce the time of preprocessing and it would be interesting to use this work to visualize big assembly of models using ray tracing on parametric surfaces using also trimming method. We can improve our method to compute another kind of distance between two faces, which is closer to the notion of the skeleton and the spherical distances.

References

ABERT, O., GEIMER, M., AND MULLER, S. 2006. Direct and fast ray tracing of nurbs surfaces. *Symposium on Interactive Ray Tracing 0*, 161–168.

ÁFRA, A. T. 2012. Incoherent ray tracing without acceleration structures. In *Eurographics*, Eurographics Association, 97–100.

AILA, T., AND KARRAS, T. 2010. Architecture considerations for tracing incoherent rays. In *Proc. of the Conf. on High Performance Graphics*, Eurographics Association, HPG '10, 113–122.

BALÁZS, Á., GUTHE, M., AND KLEIN, R. 2004. Efficient trimmed nurbs tessellation. *Journal of WSCG 12*, 1, 27–33.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2004. Interactive ray tracing of free-form surfaces. In *Proc. of AFRIGRAPH '04*, ACM, New York, NY, USA, 99–106.

CLAUX, F., VANDERHAEGHE, D., BARTHE, L., PAULIN, M., JESSEL, J.-P., AND CROENNE, D. 2012. An Efficient Trim Structure for Rendering Large B-Rep Models. 31–38.

DAMMERTZ, H., AND KELLER, A. 2006. Improving ray tracing precision by object space intersection computation. In *Interactive Ray Tracing 2006, IEEE Symposium on*, 25–31.

EFREMOV, A., HAVRAN, V., AND SEIDEL, H.-P. 2005. Robust and numerically stable bézier clipping method for ray tracing nurbs surfaces. In *Proc. of the 21st spring conference on Computer graphics*, ACM, NY, USA, SCCG '05, 127–135.

GEIMER, M., AND ABERT, O. 2005. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *Proc. of WSCG*, 71–78.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2005. Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph. 24*, 3, 1016–1023.

HAPALA, M., DAVIDOVIC, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. 2011. Efficient stack-less bvh traversal for ray tracing. In *27th Spring Conference on Computer Graphics*.

KAJIYA, J. T. 1982. Ray tracing parametric patches. In *Proc. of the 9th annual conference on Computer graphics and interactive techniques*, ACM, NY, USA, SIGGRAPH '82, 245–254.

KARRAS, T. 2012. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proc. of the conf. on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 33–37.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 1000–1009.

MORA, B. 2011. Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph. 30*, 5, 117:1–117:12.

NISHITA, T., SEDERBERG, T. W., AND KAKIMOTO, M. 1990. Ray tracing trimmed rational surface patches. *SIGGRAPH Comput. Graph. 24*, 4, 337–345.

PABST, H.-F., SPRINGER, J., SCHOLLMEYER, A., LENHARDT, R., LESSIG, C., AND FROEHLICH, B. 2006. Ray casting of trimmed nurbs surfaces on the gpu. *Symposium on Interactive Ray Tracing 0*, 151–160.

SCHNEIDER, P. J. 1990. Graphics gems. Academic Press Professional, Inc., San Diego, CA, USA, ch. An algorithm for automatically fitting digitized curves, 612–626.

SCHOLLMEYER, A., AND FRÖHLICH, B. 2009. Direct trimming of nurbs surfaces on the gpu. In *ACM SIGGRAPH'09*, ACM, New York, NY, USA, 47:1–47:9.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009*.

TOTH, D. L. 1985. On ray tracing parametric surfaces. In *Proc. of the 12th annual conference on Computer graphics and interactive techniques*, ACM, NY, USA, SIGGRAPH '85, 171–179.

WALD, I., AND SLUSALLEK, P., 2001. State of the art in interactive ray tracing.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2009. State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6, 1691–1722.

WALD, I. 2007. On fast construction of sah-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 33–40.