Parallel GPU Algorithms for Mechanical CAD

by

Adarsh Krishnamurthy

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering - Mechanical Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sara McMains, Chair
Professor David Dornfeld
Professor Carlo Séquin

Fall 2010

The dissertation of Adarsh Krishnamurthy, titled Parallel GPU Algorithms for Mechanical CAD, is approved:

Professor Sara McMains, Chair Date

Professor David Dornfeld Date

Professor Carlo Séquin Date

University of California, Berkeley

Parallel GPU Algorithms for Mechanical CAD

by

Adarsh Krishnamurthy

Abstract

Parallel GPU Algorithms for Mechanical CAD

by

Adarsh Krishnamurthy

Doctor of Philosophy in Mechanical Engineering

University of California, Berkeley

Professor Sara McMains, Chair

This thesis describes new parallel GPU algorithms that accelerate fundamental CAD operations such as spline evaluations, surface-surface intersections, minimum distance computations, moment computations, etc., thereby improving the interactivity of a CAD system.

CAD systems (such as SolidWorks, AutoCAD, ProE, etc.) create graphical user interfaces for solid modeling, which build on fundamental CAD operations that are performed by a modeling kernel. However, since many of these fundamental operations are compute-intensive, the CAD systems make the designer wait until a particular operation is completed before providing visual feedback and allowing new operations to be performed, reducing interactivity. The broad objective of this research is to develop new parallel algorithms for CAD that run on Graphics Processing Units to provide order-of-magnitude better performance than current CPU implementations.

A critical operation that all CAD systems have to perform is the evaluation of Non-Uniform Rational B-Splines (NURBS) surfaces. We developed a unified parallel algorithm to evaluate and render a NURBS surface directly using the GPU. The GPU algorithm can render over 100 NURBS surfaces at 30 frames per second, significantly enhancing interactivity.

Fundamental modeling operations (such as surface intersections, separation distance computations, etc.) are typically performed repeatedly in a CAD system during modeling. We have developed GPU-accelerated algorithms that perform surface-surface intersections more than 50 times faster than the commercial solid modeling kernel ACIS. We have also developed GPU algorithms to perform minimum distance computations, which have applications in multi-axis machining, path planning, and clearance analysis. These algorithms are not only more than two orders of magnitude faster than the CPU implementations, they often have much tighter error bounds.

We have also developed algorithms for computing accurate geometric moments of solid models that are represented using multiple trimmed-NURBS surfaces. We have developed a framework that makes use of NURBS surface data to evaluate surface integrals of trimmed NURBS surfaces in real time. With our framework, we can compute volume and moments of solid models with error

estimates. The framework also supports local geometry changes, which is useful for providing interactive feedback to the designer while the solid model is being designed.

Finally, the ultimate objective of this research is to provide a generalized framework to overcome some of the GPU programming challenges in CAD. Using this framework, a programmer could easily develop complex CAD algorithms that utilize the GPU to improve the performance of CAD systems.

Acknowledgments

I would like to thank my advisor, Prof. Sara McMains, for providing me sound guidance and motivation throughout my PhD. She was always supportive of my research and helped me develop several skills that are invaluable for a researcher. I am always amazed at her sense of precision in framing and presenting research ideas and problems. I immensely enjoyed working with her throughout my stay at Berkeley.

I would like to thank my dissertation and qualifying exam committee members Prof. David Dornfeld and Prof. Carlo Séquin for their guidance. Prof. Dornfeld's courses on manufacturing helped me understand the application area where my research could be usefully applied. Prof. Séquin's course on Solid Modeling introduced me to the field of CAD and helped me get an idea of all the interesting unsolved problems in the field. I always enjoyed the invigorating and thought provoking discussions I had with them.

I would also like to thank the other members of my qualifying exam committee, Prof. Paul Wright and Prof. Alice Agogino. They provided me invaluable guidance while writing my research proposal and helped me develop a coherent narrative for my ideas.

I benefited immensely from my two internships at SolidWorks Corporation. I am grateful to my mentor Kirk Haller for being really helpful and supportive of my research. He introduced me to many current research challenges in the CAD industry. I immensely enjoyed working with my colleagues at SolidWorks, especially Xiaobin Wu, and Prof. Stephen Mann.

I would also like to acknowledge my collaborators Prof. Gershon Elber and Prof. Iddo Hanniel for all the interesting discussions we have had regarding CAD research. Prof. Elber was especially helpful in making sure that our algorithms are mathematically sound.

I would like to thank all the members of Computer Aided Design and Manufacturing Lab for providing me a secure environment for performing research. I would especially like to thank Rahul Khardekar for providing me insights on GPU programming and helping me out whenever I was stuck. Xiaorui, Youngung, Wei, Yusuke, and Sushrut made sure that working in the lab was a fun experience.

A special thanks to Kranthi Kiran Mandadapu, my friend and roommate during my stay at Berkeley, for all the thought-provoking conversations we have had on diverse topics. I would also like to thank Athulan Vijayaraghavan for all the discussions we have had in the corridors of Etcheverry hall. I especially enjoyed our long research conversations during our afternoon cookie breaks.

My stay at Berkeley would have been extremely boring without my friends. Sriram, Karthik, Dilip, Pannag, Subbu, Mary, and Praveena made sure that I would feel right at home during my initial years at Berkeley. The last few years at Berkeley would not have been so much fun if not for Debanjan, Aditya, Yasaswini, Sharanya, Anuj, and Vinay. I would also like to thank my friends from my undergraduate years at IIT Madras, Vaidehi, Sethu, Subash, Bharat, Radha, Prashant, and Prathusha, for their support throughout my PhD.

My doctoral work would not have been possible if not for my parents' immutable support for every endeavor of mine. I am also grateful to my brother, uncle, and grandmother for their love and support. They have always been a great source of inspiration for all my accomplishments.

Dedicated to my family and friends.

Contents

Chapter 1

Introduction

Designers are increasingly relying on commercial CAD systems like SolidWorks, ProEngineer etc. for communicating as well as fine-tuning their designs. However, these systems are not being used to verify the validity of the design or to check if the design is optimal for further processes like manufacturing, cleaning, assembly etc. This is because of the lack of efficient tools that can give interactive feedback to the designer about design and manufacturing. One of the main hurdles in developing functional feedback in CAD software is that some of the core modeling operations required to perform these analyses are computationally intensive and cannot be performed fast enough to give real-time feedback. One way to improve the performance of these fundamental operations is to parallelize them so that they can run efficiently on current generation hardware like GPUs and multi-core CPUs.

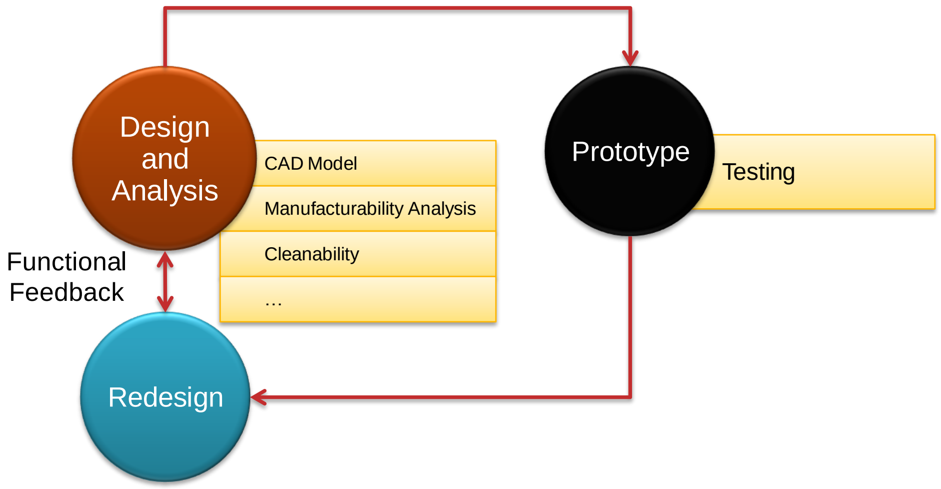
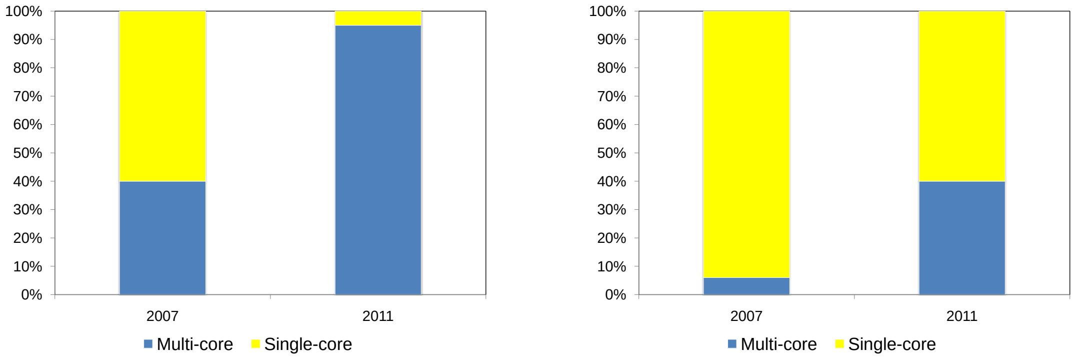


Figure 1.1: *A design process coupled with analysis and interactive functional feedback will reduce the number of design cycles.*

Improving the performance of fundamental modeling operations is essential for providing a good design experience. Such interactive operations will help integrate analysis with design by providing interactive functional feedback to the designer while the part is being designed. As shown in Figure (1.1), this would reduce the time taken for the design process by reducing the number of design cycles and would ultimately result in reduced time to market for the product.

An important motivation for parallelizing existing CAD algorithms is the growth of multi-core CPUs. Figure (1.2(a)) shows the projected growth of multi-core processors shipped by Intel, according to which 95% of the total CPUs shipped by Intel in 2011 will be multi-core. In addition, the number of cores in a CPU is expected to be greater than 100 in a few years extrapolating from the current technological growth rate. Furthermore, a survey by the Venture Development Corporation projects that the percentage of software tools capable of taking advantage of such multi-core processors will constitute only 40% of the total software in 2011 even if current development rates are maintained. These statistics provide compelling motivation for the need to develop new parallel algorithms for some of the fundamental CAD operations.



(a) *Intel's multi-core processor production* **(b)** *Software tools for multi-core processors*

Figure 1.2: *Projected growth of multi-core hardware and software.*

Graphics Processing Units (GPUs) have recently evolved into programmable parallel processors capable of performing general-purpose computational tasks. Figure (1.3) shows the peak floating-point performance of AMD and NVIDIA GPUs compared to current generation CPUs. GPUs are more than an order of magnitude faster than existing CPUs and the rate of increase of their speed over time has also been higher. GPUs have multi-core programmable units that can execute a user-defined set of instructions; current GPUs have more than 100 programmable cores. Because multiple operations are performed in parallel, and operands are four-component vectors, GPUs can achieve much higher computational speeds than conventional CPUs on arithmetically intensive operations. This high performance has been exploited for performing different general-purpose computations and has developed into a separate field called General-Purpose computations on the GPU (GPGPU) with many applications [Kipfer *et al.*, 2004; Pharr, 2005; Guthe *et al.*, 2005; Loop and Blinn, 2005; Guthe *et al.*, 2006; Carr *et al.*, 2006; Greß *et al.*, 2006; Kanai, 2007; Sengupta *et al.*, 2007].

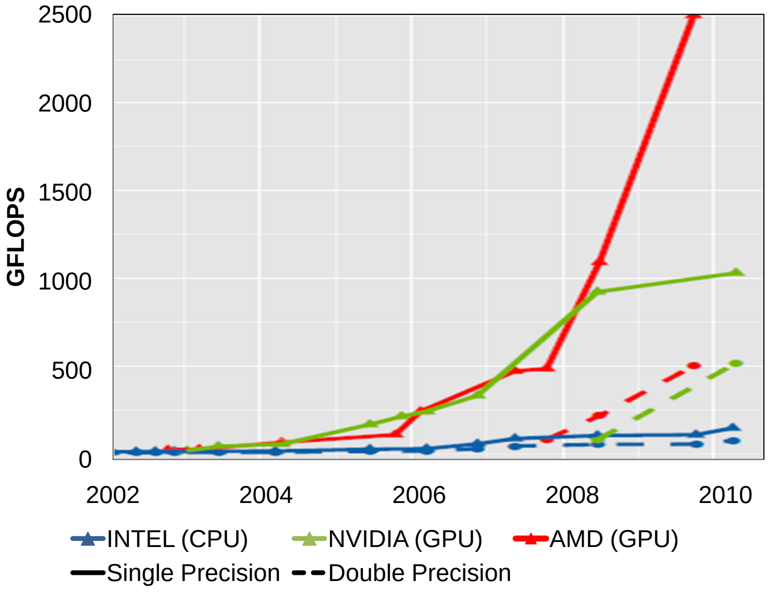


Figure 1.3: *Graph comparing the floating point performance (in terms of billion floating points operations per second) of current generation GPUs with CPUs (figure from John Owen, UC Davis).*

Until recently, using the GPU to perform the computations involved casting the computational problem as a graphics problem; this necessitated the programmer to know the details of graphics programming to utilize the GPU effectively. Hence, GPGPU programming remained a niche field; programmers had to come up with novel algorithms to cast the computations as graphics processes. Early GPGPU applications made use of shaders–specialized graphics programming kernels–to perform the computations. The shaders were written in high-level languages (Cg, BROOK, etc.) and used the OpenGL graphics API to invoke them for performing the computations. This research tries to build a framework that can be effectively used to develop GPU algorithms for CAD. Such a framework will help reduce development time and will help in developing optimal implementations of GPU algorithms.

1.1 Research Outline

The main objective of this research is to develop GPU algorithms for fundamental CAD operations. Since many new commercial products are made of smooth surfaces for better aesthetics, the focus is on developing algorithms that can efficiently operate on curved free-form surfaces. Non-Uniform Rational B-Spline (NURBS) surfaces are the most general spline representation for curved surfaces in CAD systems. We have developed parallel GPU algorithms to render as well as perform modeling operations on NURBS surfaces. These can be easily extended to actual 3D solid models since the boundary of these solid models are represented using NURBS surfaces.

NURBS are the most general type of spline curve, encompassing B-splines and Bezier curves as special cases ([Piegl, 1991]). NURBS offer a way to represent almost arbitrary shapes while maintaining mathematical exactness. NURBS control points can have non-unit weights, which make them rational. Rational curves have the advantage that they can represent conic sections, allowing the exact representation of features with circular cross sections such as rolling ball blends and surfaces of revolution. Evaluating NUBRS surfaces is one of the fundamental algorithms required for for interactive feedback in models with curved free-form surfaces. We have developed a GPU algorithm to evaluate a NURBS surfaces and its normals that is more than 40 times faster than our optimized CPU implementation. We present the details of this algorithm in Chapter (4).

Expanding further the role of GPUs in CAD, we have developed algorithms for performing surface interrogations. Surface interrogations, which include inverse evaluation and ray intersections, are fundamental operations required for interacting with NURBS surfaces. These are required to interactively modify a NURBS surfaces in real-time. Another fundamental operation that has many applications like Boolean tree to b-rep conversion is evaluation of intersection curves or surface-surface intersections. We have developed parallel algorithms that run on the GPU and accelerate both these operations that are explained in detail in Chapter (5).

Computing minimum distances and clearance between solid models is essential for performing accessibility analysis and path planning in CAD. We have developed GPU-accelerated algorithms for performing minimum distance computations between NURBS surfaces and models made up of multiple NURBS surfaces that we explain in detail in Chapter (6). Our algorithms are not only faster, but also more accurate than the commercial solid-modeling kernels. Our algorithms also have theoretical error bounds for the results of the queries that are directly applicable to current CAD systems.

Finally, we have developed surface integration techniques on NURBS surfaces that can be used to compute accurate moments of solid models (Chapter (7)). We have developed GPU algorithms that can compute volume, center of mass, and moment of inertia interactively while a designer is changing a model. Providing feedback about the center of mass informs designers about the stability of the part while designing. Volume computations are also important in injection molding to estimate the amount of raw material required to manufacture a part.

In Chapter (2), we present the mathematical formulation and the background for NURBS surfaces that are requied to understand our GPU algorithms. We also give a introduction to programmable GPUs and the GPU framework we use to develop algorithms in Chapter (3).

Chapter 2

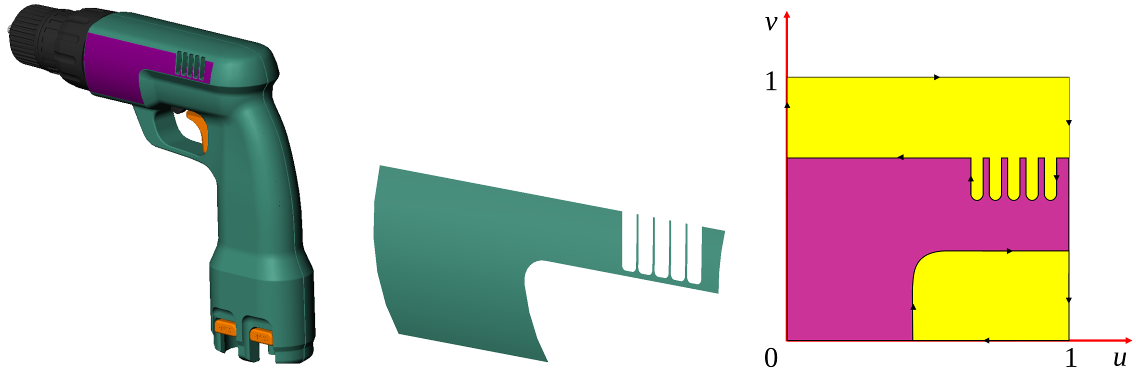
Background and Mathematical Formulation

2.1 NURBS Surface Models

A CAD model is usually represented by the CAD system using the faces that form the boundary of the model. This method of representation, called Boundary-Representation(B-Rep), is the standard representation used in commercial systems. The boundary of a CAD model is usually represented using tensor product NURBS surface patches. Hence, a model is fully defined by defining the NURBS representation of the faces of the model.

NURBS are the most general type of spline curve, encompassing B-splines and Bezier curves as special cases [Piegl, 1991]. NURBS offer a way to represent almost arbitrary shapes while maintaining mathematical exactness. NURBS control points can have non-unit weights, which make them rational. Rational curves have the advantage that they can represent conic sections, allowing the exact representation of features with circular cross sections such as rolling ball blends and surfaces of revolution. Evaluating NUBRS surfaces is one of the fundamental algorithms in a CAD system.

These NURBS surfaces are rectangular sheets; therefore they are not very flexible, especially when it comes to representing surfaces that are not rectangular or those with holes or complex local geometries that arise due to Boolean operations. Therefore, many NURBS patches are trimmed, discarding a part of the surface portion defined in the parametric domain. An example of a trimmed NURBS surface in a CAD model is shown in Figure (2.1). The trimming information is defined in the 2D parametric domain of the surface (Figure (2.1(c))). Typically, trim curves are represented as directed closed loops; the direction of the loop determines which side of the trim curve to cut away. There can also be multiple loops per surface, one defining the boundary and others defining interior holes, or even holes within holes.



(a) *Cordless Drill model* **(b)** *Trimmed NURBS surface* **(c)** *Parametric trim curves*

Figure 2.1: *Cordless drill modeled using trimmed NURBS surfaces.*

2.2 NURBS Curve and Surface Definitions

In this section, we briefly review the mathematical notation we use for defining NURBS curves and surfaces adapted from [Piegl and Tiller, 1997]. Equation (2.1) gives the definition of a NURBS curve C as a function of the parameter u, where the P_is are the control points and N_i^ps are the B-spline basis function of degree p given by Equation (2.2). Since the NURBS curve can have repeated knot values, the special case of $0/0$ that may arise in either of the terms in Equation (2.2) is taken to be 0. For concreteness, we consider a NURBS curve of order k with n control points, which has a knot vector of length $n+k$ in all the examples. Although a spline curve may have hundreds of control points, the local support property guarantees that in a B-spline curve of order k, the curve evaluation point at any given parameter location is controlled only by the k (parametrically) nearest control points. This simplifies evaluation as well as curve editing and optimization.

$$C(u) = \frac{\sum_{i=0}^{n} N_i^p(u) w_i P_i}{\sum_{j=0}^{n} N_i^p(u) w_i} \tag{2.1}$$

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u) \tag{2.2}$$

$$N_i^0(u) = \begin{cases} 1 & \text{if } u_i \le u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

The tensor-product NURBS surface definition (Equation (2.4)) is extended directly from that of a NURBS curve. The parameter values (u, v) are the 2D evaluation points; the basis functions N_i^ps are the same B-spline basis functions of degree p defined by Equation (2.2); and the P_i^js are the NURBS control points defined as a quadrilateral mesh. The NURBS surface is fully defined

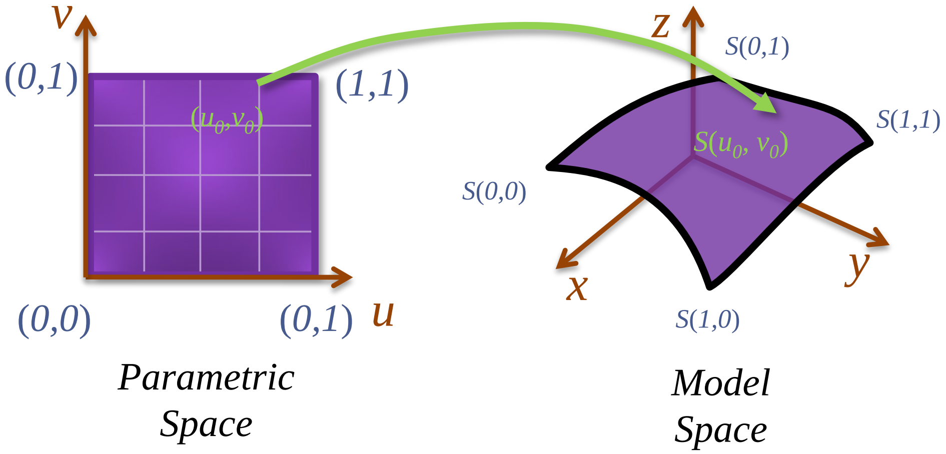


Figure 2.2: *A point* (u_0, v_0) *in the parametric space is mapped to* $S(u_0, v_0)$ *in the model space.*

by a control point mesh and the two independent arbitrary degree u and v parametric direction knot vectors (Figure (2.2)). As in the case of curves, a NURBS surface point is influenced only by a small sub-mesh of control points of size $k_u \times k_v$.

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{n} N_i^p(u)N_j^p(v)w_{ij}P_{ij}}{\sum_{i=0}^{n}\sum_{j=0}^{n} N_i^p(u)N_j^p(v)w_{ij}} \tag{2.4}$$

2.3 Differential Geometry for B-Spline Surfaces

In this section, we present a concise version of the equations that are required for computing derivatives of NURBS surfaces, adapted from [Piegl and Tiller, 1997]. We present the exact equations for a Non-Uniform B-Spline (NUBS) surface first and then extend the derivation to include rational surfaces. For a NUBS surface, $S(u,v)$, given by Equation (2.5), the derivatives can be computed by multiplying the control points (P_{ij}s) with the derivatives of the basis functions. The N_i^ps and N_j^qs are the B-spline basis functions of degree p and q respectively, as a function of the knots u_is and v_is respectively (Equations (2.6) and (2.7)); the P_{ij}s are the NUBS control points defined as a quadrilateral mesh.

$$S(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u)N_j^q(v)P_{ij} \tag{2.5}$$

$$N_i^p(u) = \frac{u-u_i}{u_{i+p}-u_i}N_i^{p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}}N_{i+1}^{p-1}(u) \tag{2.6}$$

$$N_i^0(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

The derivative of the basis function of degree p with respect to u is given by Equation (2.8). To evaluate the derivative of a basis function of degree p, the basis function of degree $p-1$ needs to be computed. We use the indicial notation $N_{,u}$ to denote the derivative with respect to u. Note that the $p-1$ in the numerator of Equation (2.8) arises due to the fact that the B-spline basis function of degree p that we are differentiating is a piecewise polynomial of degree p in u.

$$N_{i,u}^p(u) = \frac{p-1}{u_{i+p} - u_i} N_i^{p-1}(u) - \frac{p-1}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u) \tag{2.8}$$

The derivatives of the B-spline basis functions, $N_{,u}$ and $N_{,v}$, are then multiplied by the control points P_{ij} to get the derivative along the u or v parametric direction on the surface as given by Equations (2.9) and (2.10) respectively. We can then calculate the surface normal $N(u,v)$ of the NUBS surface (Figure (2.3)) by taking the cross product of the u and v partial derivatives (Equation (2.11)). It should be noted that $N(u,v)$ is not a unit vector field but it is well defined as long as S is a regular surface.

$$S_{,u}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) P_{ij} \tag{2.9}$$

$$S_{,v}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_{j,v}^q(v) P_{ij} \tag{2.10}$$

$$N(u,v) = S_{,u}(u,v) \times S_{,v}(u,v) \tag{2.11}$$

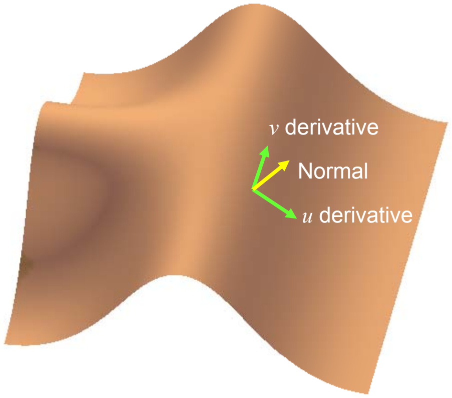


Figure 2.3: *Calculation of surface normal from the u and v partial derivatives.*

2.3.1 Rational Derivatives

The derivatives of NURBS surfaces are not as straightforward to evaluate as in the NUBS case [Abi-Ezzi and Wozny, 1990]. This is because the derivatives have to be evaluated using the chain rule due of the existence of the rational component. The NURBS surface coordinates are evaluated as the 4-component vector shown in Equations (2.12) and (2.13). Since we evaluate the 4-component vectors without performing the rational division on the GPU, we can effectively use this data to evaluate the surface derivatives.

$$S(u,v) = \frac{\bar{X}}{w}, \bar{X} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{2.12}$$

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) x_{ij} \\ \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) y_{ij} \\ \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) z_{ij} \\ \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij} \end{pmatrix} \tag{2.13}$$

$$S_{,u}(u,v) = \frac{\bar{X}_{,u} w - \bar{X} w_{,u}}{w^2} \tag{2.14}$$

$$\begin{pmatrix} x_{,u} \\ y_{,u} \\ z_{,u} \\ w_{,u} \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) x_{ij} \\ \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) y_{ij} \\ \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) z_{ij} \\ \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) w_{ij} \end{pmatrix} \tag{2.15}$$

The partial derivative with respect to u (Equation (2.14)) is derived using the quotient rule (in turn derived using the chain rule). It can be calculated by first evaluating the product of the derivatives of the basis functions and the corresponding control points as a 4-component vector (Equation (2.15)) and then performing the required rational division operations. The partial derivative of the surface with respect to v can also be evaluated in a similar manner. In this work, we assume all the weights (w) are positive and hence no poles can occur in S or its partial derivatives.

2.4 Curvature of NURBS Surfaces

Evaluating the exact curvature of the surfaces along the two parameter directions can be performed in a similar manner to evaluating the first derivatives. However, the number of additional calculation steps (16 passes for a bi-cubic surface) required for this operation is prohibitively many and therefore cannot be completed in a real-time setting. Nevertheless, since we have exact derivatives along the two parameter directions, we can approximate the second derivatives to a reasonable accuracy (error $< O(1/n^2)$ for n evaluation points) by evaluating them using central differencing.

The central differencing formula for second derivatives is given in Equation (2.16). The value of h is $1/n$ for the u direction and $1/m$ in the v direction since the surface is evaluated on a $(n+1) \times (m+1)$ grid of evaluation points. There are three second-derivative values for each surface point: the second derivatives with respect to each parameter direction ($\partial^2 S/\partial u^2$ and $\partial^2 S/\partial v^2$) and one mixed second derivative ($\partial^2 S/\partial u \partial v$). However, we can use the same program written to perform the central differencing operation to evaluate the second derivatives with different first derivative values as input. For example, Equation (2.17) shows how to calculate the second derivative with respect to u using the first derivative as input using central differencing.

$$\frac{\partial F(x)}{\partial x} = \frac{F(x+h) - F(x-h)}{2h} \tag{2.16}$$

$$\frac{\partial^2 S}{\partial u^2} = \frac{\partial \left(\frac{\partial S(u,v)}{\partial u}\right)}{\partial u} = \frac{\frac{\partial S(u+h,v)}{\partial u} - \frac{\partial S(u-h,v)}{\partial u}}{2h}, h = \frac{1}{n} \tag{2.17}$$

2.5 Bounding-Boxes for NURBS Surfaces

We make use of axis-aligned bounding-boxes (AABB) for the NURBS surfaces to perform modeling operations using the GPU. With the help of such bounding-boxes, several queries such as ray-surface intersections and surface-surface intersections can be efficiently answered, which then form the building blocks for more complex operations like sketching on the surface and intersection curve calculations. There are different methods to construct bounding-boxes for free-form surfaces. One method is to fit bounding-boxes that enclose the control-points that define the surface. This method however does not produce very tight bounding-boxes and makes the bounding-boxes independent of the user-defined tolerance values. Another approximate method is to construct bounding-boxes enclosing sets of four adjacent points evaluated on the surface. In [Greß *et al.*, 2006], the bounding-boxes for use in collision detection were constructed from sets of four adjacent points on a parameterized surface, after ensuring that their approximation of the surface is within the given tolerance by very finely subdividing the surface. However, this method does not guarantee that the surface will be completely enclosed by the bounding-box and it can potentially miss some intersections. We overcome these issues by evaluating the NURBS surface in a regular grid and then expand the bounding-boxes based on the curvature of the surface so that they are guaranteed to enclose the surface. Another advantage of this method is that the bounding-boxes automatically become tighter when we evaluate the surface at a finer resolution.

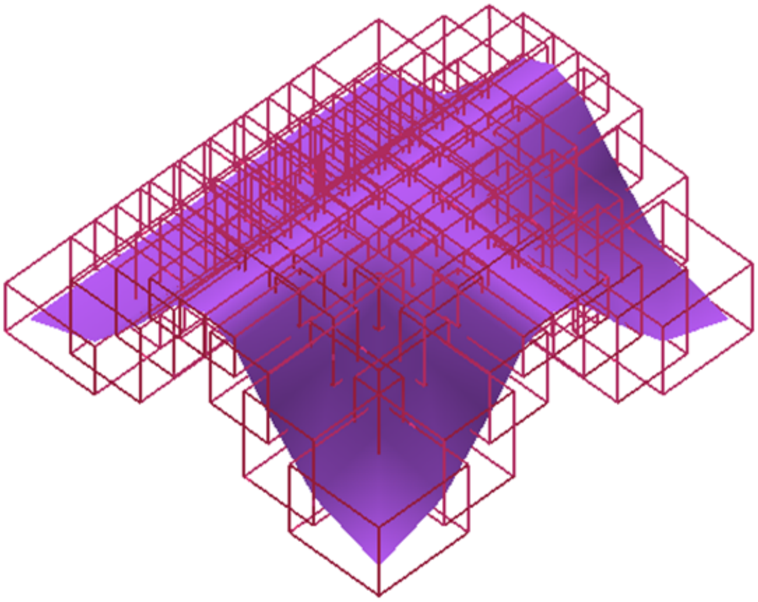


Figure 2.4: *Surface bounding-boxes constructed from points evaluated on a NURBS surface.*

The analytical expression for the factor that can be used to expand the bounding-boxes based on the surface curvature is given in [Filip *et al.*, 1987]. They show that if a parametric C^2 surface is evaluated at $(n+1)\times(m+1)$ grid of points, the deviation of the surface from the piecewise linear approximation cannot exceed a constant K defined by Equations (2.18)-(2.21).

$$K = \frac{1}{8}\left(\frac{1}{n^2}M_1 + \frac{2}{nm}M_2 + \frac{1}{m^2}M_3\right) \tag{2.18}$$

$$M_1 = \max_{\forall(u,v)}\left[\max\left(\left|\frac{\partial^2 x}{\partial u^2}\right|, \left|\frac{\partial^2 y}{\partial u^2}\right|, \left|\frac{\partial^2 z}{\partial u^2}\right|\right)\right] \tag{2.19}$$

$$M_2 = \max_{\forall(u,v)}\left[\max\left(\left|\frac{\partial^2 x}{\partial u \partial v}\right|, \left|\frac{\partial^2 y}{\partial u \partial v}\right|, \left|\frac{\partial^2 z}{\partial u \partial v}\right|\right)\right] \tag{2.20}$$

$$M_3 = \max_{\forall(u,v)}\left[\max\left(\left|\frac{\partial^2 x}{\partial v^2}\right|, \left|\frac{\partial^2 y}{\partial v^2}\right|, \left|\frac{\partial^2 z}{\partial v^2}\right|\right)\right] \tag{2.21}$$

To compute the bounding-boxes for a NURBS surfaces, we first evaluate the surface $S(u,v)$ in a grid of points using our NURBS evaluator. We also evaluate the precise first derivatives of the surface, $\partial S/\partial u$ and $\partial S/\partial v$, at these points as explained in Section (2.3.1). We approximate the

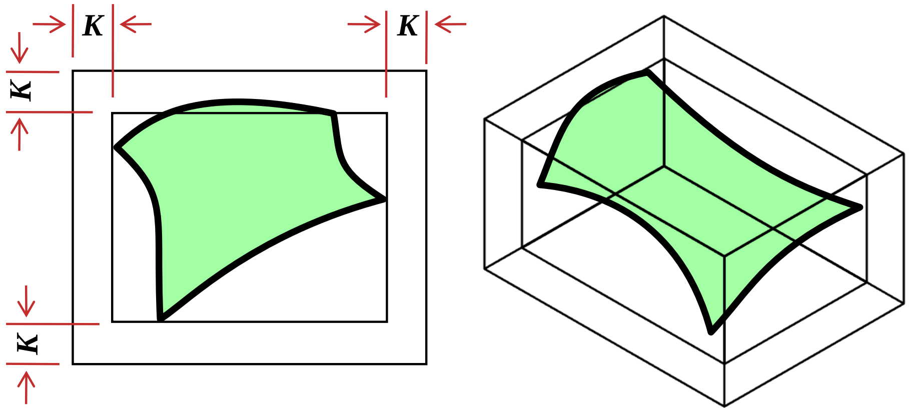


Figure 2.5: *We expand the AABBs by K in all three dimensions to guarantee that the surface patch is completely enclosed.*

second partial derivatives of the surface by central differencing (explained in Section (2.4)). We then find the value of K for the surface using Equation (2.18). The bounding-boxes themselves are constructed by constructing boxes that enclose sets of four adjacent surface points and then expanding this box by K, which ensures that no part of the surface penetrates out of the bounding-box.

2.6 Summary

In this chapter we have presented the mathematical formulations and background that are used in our NURBS algorithms. We make use of the surface bounding-boxes as a accelerating structure for our geometric algorithms such as surface-surface intersection evaluation and minimum distance computations. In addition, we make use of the exact derivatives to calculate the geometric bounds for these algorithms. In the next chapter, we present the basics of programmable GPUs and the GPU framework for accelerating CAD algorithms that we have developed.

Chapter 3

GPU Framework

3.1 Programmable GPUs

Graphics processing units (GPUs) have evolved into programmable parallel processors capable of performing general-purpose computational tasks [Kilgariff and Fernando, 2005; Owens *et al.*, 2007]. Initially, the GPU was programmed by modifying the graphics pipeline with the help of user-defined set of instructions. These instructions were used to either modify the vertices of the geometry being rendered using the vertex program, or the fragments (potential pixels) before rendering using the fragment program. These instructions were executed in the place of a fixed sequence of geometric transformations, lighting operations (per-vertex operations), and texturing operations (per-fragment operations). Geometric primitives (triangles generally) assembled from the vertex data then get rasterized into fragments that pass through the Fragment Processing Unit (FPU).

Vertex programs can obtain the geometry and attribute (color, texture coordinates, etc.) data stored in the GPU memory via traditional display lists or Vertex Buffer Objects (VBOs). Vertex and fragment programs can access data stored in textures that can have full 32-bit floating-point precision. Usually the output of the FPU goes into a framebuffer, which is a two dimensional block of memory with four attributes at each location. In modern GPUs, the FPU can also output directly to a floating-point texture (render-to-texture) using off-screen render targets called Frame-Buffer Objects (FBOs). This allows the use of the output of a first pass through the rendering pipeline as input texture data for the second pass. FBOs can also be used to render into a Vertex Buffer Object (VBO) so that the output can be used as vertex data for the next rendering pass. Because multiple vertices and pixels are processed in parallel, and operands are four-component vectors, GPUs can achieve much higher computational speeds than conventional CPUs on arithmetically intensive operations.

With the evolution of programmable GPUs, general-purpose computations using the graphics pipeline could be performed relatively easily. NVIDIA developed a high-level language called Cg, which stands for “C for graphics,” for writing the vertex and fragment programs in a C-like language with extensions, which could then be compiled into GPU assembly instructions during runtime [Mark *et al.*, 2003]. The OpenGL 2.0 specification introduced a similar high-level language called GLSL (OpenGL Shading Language) for GPU programming. We make use of Cg in most of our implementations of our GPU algorithms.

More recently, there has been support to perform general-purpose computations directly without using the graphics pipeline or OpenGL. NVIDIA introduced CUDA (Code-Unified Device Architecture) that uses C with extensions to utilize the GPU for computations directly. CUDA provides the advantage of simplifying the programming by separating computing operations from graphics operations. On the other hand, it is a closed system and runs only on NVIDIA hardware. To make GPU computing more widespread, an open specification standard, OpenCL, is being developed by the Khronos group to standardize computing across different platforms, including multi-core CPUs and GPUs.

3.2 GPU Programming Challenges

Even though general purpose computations using the GPU has started being more widespread, we had to overcome some key challenges posed by the GPU architecture and computations. We have identified some of the main challenges that we have tried to address in our GPU algorithms.

The first main challenge is to identify the distribution of work between the CPU and GPU. Some operations are inherently serial and are better suited to be performed on the CPU. In addition, proper distribution of work between the CPU and GPU will lead to a balanced algorithm that provides optimal performance.

The second main challenge is to overcome the limitations of the GPU architecture. Many common features such as dynamic loops and double precision arithmetic were not available in older GPUs. Even though newer generation GPUs have these features, the use of these features usually results in a performance drop. One of the main limitations that we had to overcome in our GPU CAD algorithms was the lack of scatter operation (random writes to different memory locations)in traditional GPU programming. We overcame this limitation by using a combination of a vertex program and a VBO, which is explained in more detail in Section (3.5.2). CUDA, on the other hand, supports scatter but has a performance drop associated with its use.

In order to develop optimally performing GPU algorithms, we also had to follow some performance guidelines. Operations such as coherent memory reads, where data from consecutive memory locations are read as a single block and used by different threads, drastically improve the performance. In addition, the GPU architecture is designed for pure Single Program Multiple Data (SPMD) operations. Hence operations that introduce branching, such as "if" statements, do not run as efficiently as branchless kernels. The GPU in such cases waits for the slowest branch to finish computing before proceeding to the next GPU computation. Finally, we have to reduce the amount of data that is read back from the GPU. This is because the GPU to CPU data bandwidth is much lower for read back compared to the bandwidth from CPU to GPU. If possible, we directly display the output or perform reduction operations so that we can read back only the required data.

Finally, one of the most significant challenges is to develop algorithms and implementations that are not vendor specific. Even though newer GPU programming environments such as CUDA are easier to develop in, they can run only on NVIDIA hardware. In our case, we implemented our algorithms using OpenGL function calls and used Cg for the GPU fragment programs. This makes our implementation cross-platform and it can run on both NVIDIA and AMD (ATi) GPUs.

3.3 Hybrid CPU/GPU Algorithms

We present a hybrid framework that can use both the CPU and GPU to perform geometric computations. The main idea is to split the computations into serial and parallel stages as shown in Figure (3.1). To perform the parallel operations on the GPU, we make use of the map-reduce parallelism pattern that consists of assigning the computations to separate non-communicating parallel threads [Mattson *et al.*, 2004]. The inter-communication between the CPU and GPU is shown in Figure (3.2).

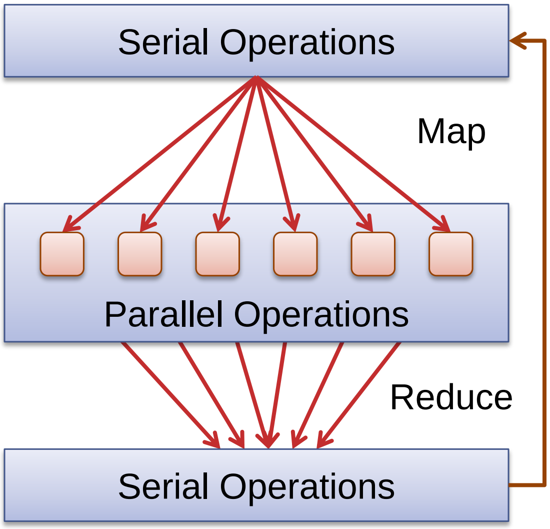


Figure 3.1: *Operation flow for performing geometric computations. The parallel operations are mapped and performed on the GPU while serial operations are performed on the CPU. The intermediate parallel output is reduced and read back to the CPU.*

Once the computations are performed, the computed result can be used by the modeling system in the three different ways shown. Read-back is important for integrating the GPU algorithms with traditional modeling systems. In addition, since GPUs are designed for pipelining the data only in one direction from the CPU to the GPU for display, the method of read-back significantly affects the performance of hybrid algorithms. The most efficient method of read-back is reducing the results to a smaller set of values by using operations such as finding the maximum, minimum, sum, or by using non-uniform stream reductions ([Sengupta *et al.*, 2007; Blelloch, 1990]). The second method is to directly display the output on the screen using the GPU. This is ideal for certain operations that require only visual outputs; for example displaying the evaluated NURBS surface directly. The last and the most expensive method is to read-back all the results from the GPU to the CPU; this might be required for certain computations where the result of a computation is required for further processing on the CPU.

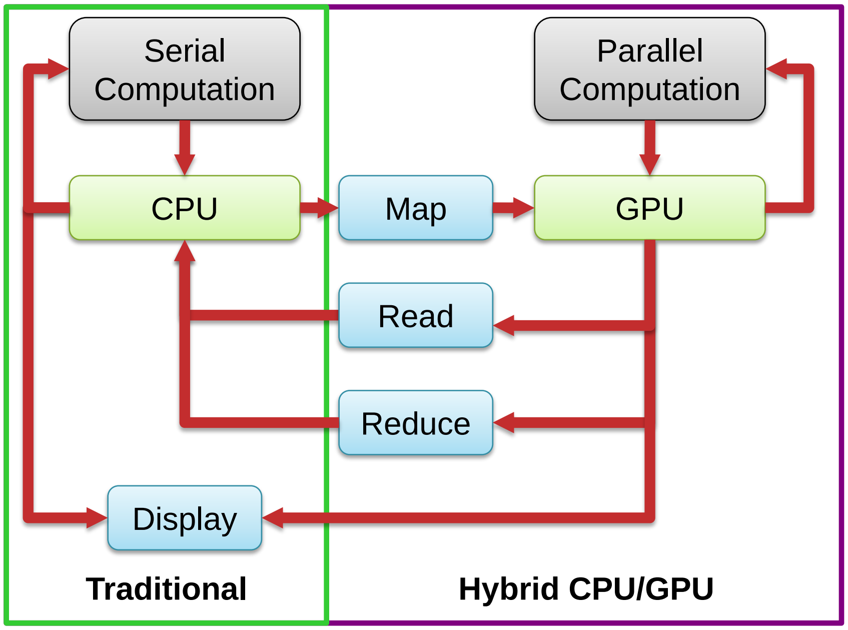


Figure 3.2: *Schematic showing our hybrid framework that extends traditional geometric computations to use the GPU as a co-processor to perform some parts of the computations in parallel.*

Our operations on the GPU fall into three main types. The first type includes parallel geometric computations that can be performed efficiently on the GPU. The outputs of such operations are usually numeric values that are then stored in the GPU as textures. If an operation produces more than one output value for each parallel operation, we can store those using separate channels of the same texture or using different textures. The second GPU operation type is parallel search operations that give a binary output of 0 or 1 based on the type of search; these include operations such as bounding-box intersection tests, finding if a value lies within a given range, etc. The third GPU operation type is reduction that is performed using multiple passes on the GPU. GPU reductions can in turn be classified into two types. The first type, called standard reductions, include reducing the given input to a single value such as computing the sum, min, max, etc. Standard reduction operations are usually performed in $O(\log n)$ passes and hence are very efficient. The second type of reductions, called non-uniform stream reductions, reduces the input to a smaller set of values.

Non-uniform stream reduction operations are particularly important when the result of a reduction operation is not a single value but multiple values that satisfy a particular criterion. Since the positions of the output elements do not have any fixed correspondence with the positions of the input, the stream-reduction process is considered non-uniform. We make use of an $O(n)$ GPU stream-reduction algorithm (Section (3.5.2)) to perform non-uniform stream reductions; please refer to Section (3.5.2) for details of this algorithm.

3.4 Computations Using Shader Programs

We perform standard geometric computations using Shader Programs. These are fragment program kernels written in Cg and then compiled during run time to GPU assembly code. We make use of OpenGL rendering primitives to invoke these kernels. If we want to perform some geometric computation on a rectangular data of size $n \times m$, we transfer the data first to an $n \times m$ texture. We then use OpenGL to draw a rectangle of size $n \times m$ to an off-screen render buffer using a FBO. The kernel is then invoked on each of the $n \times m$ pieces of data by the GPU and the final computation is written to the framebuffer. The framebuffer is attached to a texture which resides in the texture memory that can be used to access the results or as input for further processing.

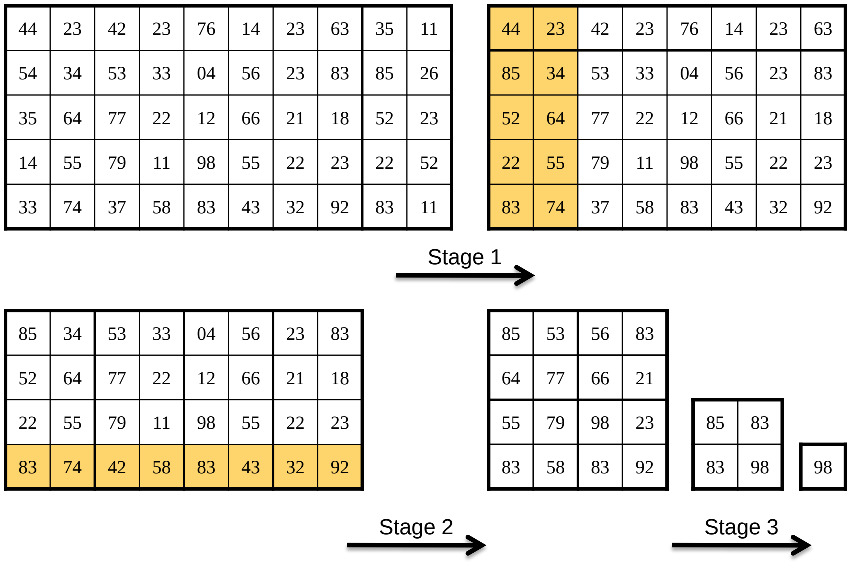
The use of texture memory for transferring data has both advantages and disadvantages. The advantage is that texture memory is cached by the GPU, which hides the latency in accessing the data. In addition, if the data is stored coherently in the texture, the latency of the read operation is very efficiently hidden. It is also random-access and can be used to store any kind of data such as integer or floating-point. On the other hand, texture memory is read-only for the GPU fragment programs. There is also an overhead incurred while setting up the texture memory.

3.5 Reductions

3.5.1 Standard Reductions

Standard reductions include reducing the given input to a single value such as computing the sum, min, max, etc. If the given input is a square texture with a size that is a power of two, then we reduce four adjacent values (for example, the sum) to a single value in a given pass. Thus the total reduction operation can be performed in $O(\log n)$ passes and hence, it is very efficient.

On the other hand, if the input is not a square texture, then we perform the reduction in three stages. In the first stage, we reduce only two values along the height or the width direction until we reach a power-of-two texture. In the second stage, we reduce along the larger dimension until we reach a square texture. Finally, we perform the normal reduction for square power-of-two textures in the third stage.



44	23	42	23	76	14	23	63	35	11
54	34	53	33	04	56	23	83	85	26
35	64	77	22	12	66	21	18	52	23
14	55	79	11	98	55	22	23	22	52
33	74	37	58	83	43	32	92	83	11

44	23	42	23	76	14	23	63
85	34	53	33	04	56	23	83
52	64	77	22	12	66	21	18
22	55	79	11	98	55	22	23
83	74	37	58	83	43	32	92

85	34	53	33	04	56	23	83
52	64	77	22	12	66	21	18
22	55	79	11	98	55	22	23
83	74	42	58	83	43	32	92

85	53	56	83
64	77	66	21
55	79	98	23
83	58	83	92

85	83
83	98

98

Figure 3.3: *Example showing the different stages in reduction for computing the maximum of a non-square non-power-of-two texture. Stage 3 is the standard reduction for square power-of-two texture.*

3.5.2 Non-Uniform Stream Reductions

One of the most essential operations in our CAD algorithms is to find the addresses or the indices (location) of the texels in a texture that have a given value. This operation is usually performed after a parallel search operation where the texels are marked as either 0 or 1 based on the result of the search. This sub-problem falls under the class of stream-reduction, the process of removing unwanted elements from a stream of values and reducing it to a smaller list containing the required output. GPGPU uses stream reduction to remove defunct elements from the output of a previous pass before sending it as input for the next pass. Since the positions of the output elements do not have any fixed correspondence with the positions of the input, the stream-reduction process is considered non-uniform. Stream reduction is usually considered a serial operation since the number of elements in the output is not known and hence the whole input has to be operated upon to output the correct result. We build on previous work that developed parallel algorithms based on parallel prefix sum for this operation.

Carr et al. [Carr *et al.*, 2006] also presented a GPU algorithm to find the indices of the rendered texels in a texture. A parallel $O(k+\log n)$ algorithm, where k is the output size, for non-uniform stream reduction based on prefix sums was given in [Blelloch, 1990]. However, standard graphics cards do not have the capability to perform the scatter operation, which was an essential step in the algorithm given in [Blelloch, 1990]. Another algorithm has been presented in [Horn, 2005] for non-uniform stream-reduction on the GPU that runs in $O(n\log n)$, not as efficient due to workarounds required because of lack of scatter. A stream reduction algorithm specifically for 2D textures on the GPU was proposed in [Greß *et al.*, 2006], which used the fragment processor to perform other operations while performing the scatter operation, thereby hiding the latency. Recently, an $O(n)$ GPU stream-reduction algorithm was proposed in [Sengupta *et al.*, 2007], also using prefix sums, that relies on the latest NVIDIA CUDA architecture for its scatter functionality. We propose a similar $O(n)$ stream-reduction algorithm based on computing a parallel prefix sum, but implement it using the standard GPGPU framework so that it is both compatible with older hardware and not limited to a single brand of GPU. Developing algorithms and implementations that are not restricted to a particular hardware or manufacturer is essential for their wide adoption.

We first explain briefly the parallel stream reduction operation described in [Blelloch, 1990]. It consists of three main steps: up-sweep, down-sweep, and scatter. The up-sweep operation computes a hierarchy of $\log n$ levels where each element at a higher-level is obtained as a sum of two elements in the lower-level (Algorithm (3.1)). An example of the up-sweep operation is shown using an 8-element 1D array (Figure (3.4)). The last element at the end of the operation gives the total number of elements with the value 1 in the input array. After performing this operation, we obtain a binary tree with the last element as the root node and the original array as the leaf nodes; each node of this tree represents the sum of all the values in the sub-tree of that node.

```
for $d = 0$ to $\log_2 n - 1$ do
    forall $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
        $x[k + 2^{d+1} - 1] \longleftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$;
    end
end
```

Algorithm 3.1: *The up-sweep algorithm to construct a hierarchy of the input.*

The down-sweep operation given by Algorithm (3.2), performed on the array resulting from Algorithm (3.1), computes the exclusive prefix sum of the original input array. The exclusive prefix sum of an array is defined as the sum of all the values preceding a particular position in the array not including the value in the position itself. Figure (3.5) gives an example of the down-sweep operation performed on the output shown in Figure (3.4) in order to calculate the exclusive prefix sum for the original input given in Figure (3.4). The first step of the down-sweep operation is to replace the last element (root element) in the array obtained after the up-sweep operation with the value 0. Then in the consecutive steps, the parent element at each sub-array is copied to the left element of the child array and the right element of the child array is calculated as the sum of the old left element and the parent element. In effect, every element now contains the sum of all the elements to the left of itself in the tree structure.

The value of the exclusive prefix sum at the positions where the value of the input array is 1 gives the address to which that particular input value has to be scattered to perform the stream reduction. The final step, after the up-sweep and down-sweep are completed, is the scatter operation

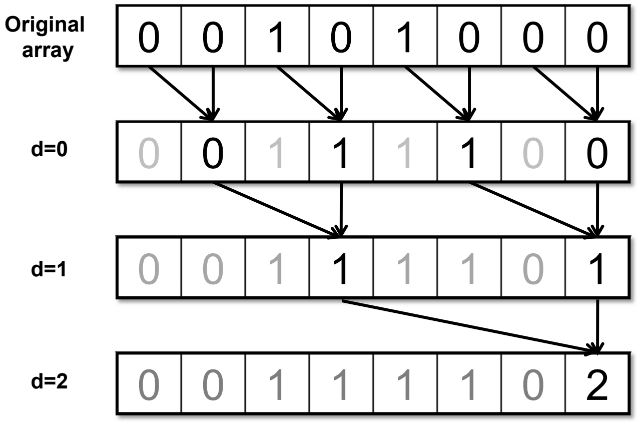


Figure 3.4: *Example of the up-sweep operation performed on an 1D array given in the first row. The inputs indicated are summed at each step.*

in which this address is used to reduce the input stream such that the elements with value 1 are collected at the front of the array.

```
$x[n-1] \longleftarrow 0$;
**for** $d = \log_2 n - 1$ *down to 0* **do**
    **forall** $k = 0$ *to* $n-1$ *by* $2^{d+1}$ **in parallel do**
        $t \longleftarrow x[k+2^d-1]$;
        $x[k+2^d-1] \longleftarrow x[k+2^{d+1}-1]$;
        $x[k+2^{d+1}-1] \longleftarrow t + x[k+2^{d+1}-1]$;
    **end**
**end**
```

Algorithm 3.2: *The down-sweep algorithm to construct the inclusive prefix sum.*

However, we cannot directly use this stream reduction algorithm on the GPU due to three issues. The first issue is that the original algorithm was developed for 1-dimensional arrays and hence has to be adapted to operate on a 2-dimensional texture. The second issue is that the traditional GPGPU model based on OpenGL or DirectX does not allow the scatter operation, which is the last step of the stream reduction algorithm. Finally, the original formulation in [Blelloch, 1990] computed the prefix sum in situ by modifying the input array. This is not possible using the standard GPGPU framework since we cannot read and write to the same location simultaneously.

We solve the first problem by first assuming that each row of the texture is a separate array and compute the first part of the up-sweep operation until each row array is reduced to a single element. Now we again perform the up-sweep operation on the array formed by concatenating all the single elements in a column along the column direction. In the example shown in Figure (3.6(b)), we

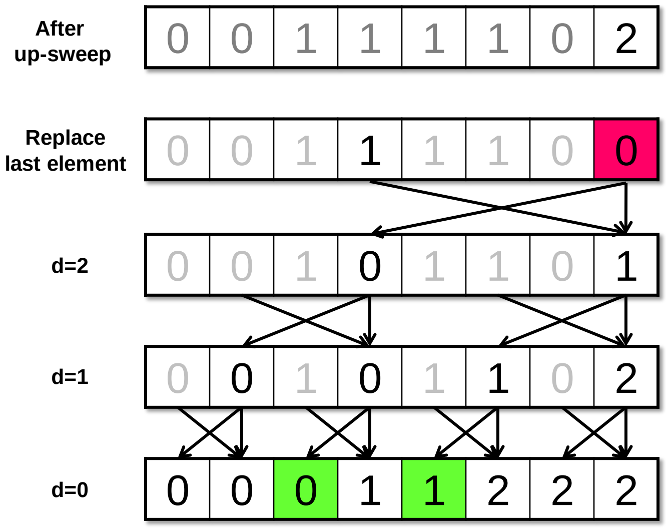
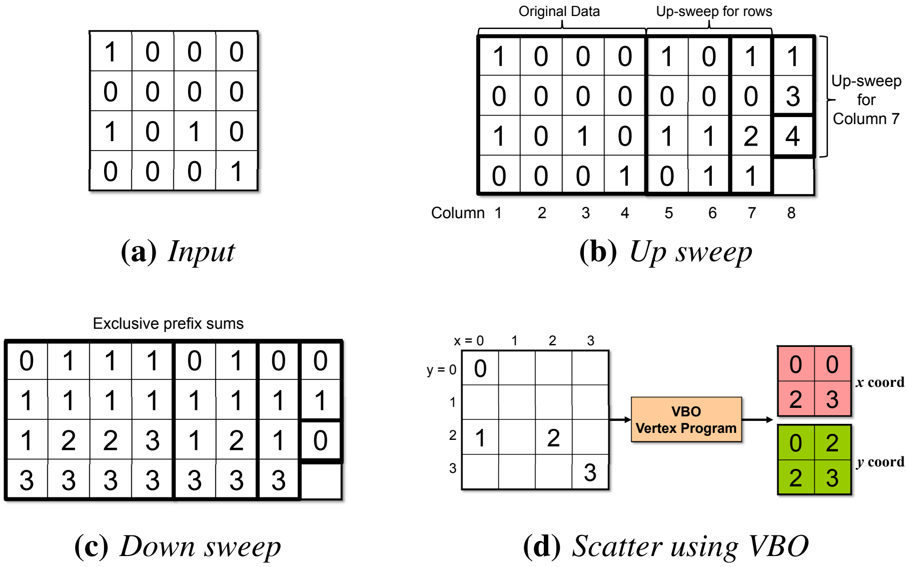


Figure 3.5: *Example of the down-sweep operation performed on the original 1D array given in Figure (3.4). The elements corresponding to the values of 1 in the original input are highlighted in the result; these are the addresses where those values are to be scattered.*

perform the up-sweep operation on each row until we end up with the values in column 7. Then we perform the up-sweep operation on column 7 and output the results to column 8. As shown in the example, to overcome the restriction of reading and writing to the same memory location, we maintain a hierarchy of the input texture. This method uses only twice the storage as the original texture used, and a single fragment program written to perform the summation can be repeatedly used. We compute the up-sweep operation in $O(\log n)$ passes.

We then perform the down-sweep operation in a similar manner but in reverse order, by first performing the operation along the columns and then extending it to the rows to obtain the exclusive prefix sum of the input. In the example shown in Figure (3.6(c)), each bold box contains the exclusive prefix sum of the corresponding bold box in Figure (3.6(b)).

Once we have the output from the down-sweep operation we extract the address of only those texels which have the value 1 in the input texture (Figure (3.6(d))). We reinterpret this texture as a VBO and use a vertex program written to output the addresses of the input values with value 1 as (x,y) coordinates, to write to two separate channels of the output texture. The size of the output texture varies based on the number of elements with value 1 in the input texture; it is equal to the first square number larger than the number of elements with value 1 in the input. This output texture is then directly used by the inverse evaluation and the surface-surface intersection applications for further processing.



(a) *Input*

(b) *Up sweep*

(c) *Down sweep*

(d) *Scatter using VBO*

Figure 3.6: *Different steps of the GPU stream reduction algorithm.*

3.6 Summary

In this chapter we have presented our framework for developing GPU algorithms for CAD. In the following chapters, we will explain how this framework can be effectively used to accelerate different CAD operations.

Chapter 4

NURBS Evaluation

4.1 Introduction

In this chapter, we present a new unified and optimized method for evaluating and displaying trimmed NURBS surfaces using the GPU. Trimmed NURBS surfaces are currently being tessellated into triangles before being sent to the graphics card for display since there is no native hardware support for NURBS. Other GPU-based NURBS evaluation and display methods either approximated the NURBS patches with lower degree patches or relied on specific hard-coded programs for evaluating NURBS surfaces of different degrees. Our method uses a unified GPU fragment program to evaluate the surface point coordinates of any arbitrary degree NURBS patch directly, from the control points and knot vectors stored as textures in graphics memory. This evaluated surface is trimmed during display using a dynamically generated trim-texture calculated via alpha blending. The display also incorporates dynamic Level of Detail (LOD) for real-time interaction at different resolutions of the NURBS surfaces. Different data representations and access patterns are compared for efficiency and the optimized evaluation method is chosen. Our GPU evaluation and rendering speeds are 40 times faster than evaluation using the CPU.

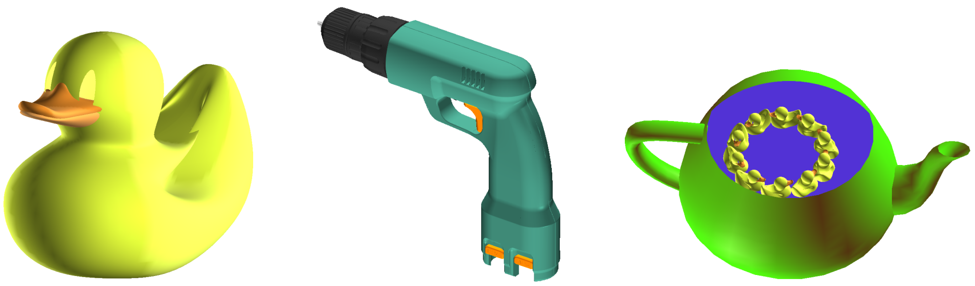


Figure 4.1: *NURBS models constructed from trimmed NURBS surfaces evaluated and rendered on the GPU.*

There is currently no built-in hardware support for displaying NURBS surfaces even though they are ubiquitous in the CAD industry. OpenGL provides a software NURBS solution; however, the implementation is not fast enough for evaluating large surfaces interactively, and in our experience it often renders trimmed NURBS surfaces incorrectly. Because surface evaluation is a computationally intensive operation, the common practice in CAD systems is to preprocess the NURBS surfaces by evaluating and tessellating them into triangles, and then using the standard graphics pipeline to display them.

Using a preprocessing technique not only leads to very high memory usage, but also restricts the surface evaluation to a particular Level of Detail (LOD). Hence, a highly enlarged view of the surface may not be tessellated sufficiently, whereas a distant view may render an excessive number of triangles. In this chapter, we describe a method by which we evaluate and display a trimmed NURBS surface directly, without approximating it by simpler surfaces, using a programmable graphics card. Using the GPU's computational power not only speeds up the surface evaluation significantly but also reduces the CPU memory usage, eliminating the need to calculate and store the tessellation data or simplified surface information that is typically used only for visualization purposes.

Previous GPU methods like [Guthe *et al.*, 2005; Guthe *et al.*, 2006] focused mainly on rendering NURBS surfaces rather than exact evaluation. Hence, they approximated higher degree NURBS surfaces by lower degree Bezier surfaces that closely resembled the original surface while rendering. The closeness was measured using pixel location error. Even though such approximations are good enough for rendering, they cannot be extended to a general-purpose NURBS evaluator capable of handling arbitrary degree NURBS surfaces. We introduce a unified method to evaluate arbitrary degree NURBS surfaces on the GPU without making any approximations in this chapter. The contemporaneous work by [Kanai, 2007] for evaluating NURBS surfaces also did not use any approximations, but required different GPU programs for evaluating NURBS surfaces of different degrees. However, having multiple GPU programs make their implementation tedious, since specific new programs have to be written for surfaces of different degrees. Moreover, since standard CAD models can be made of surfaces of widely varying degrees, with surfaces up to degree 100 occurring in many complex models [Haller, 2006], a unified NURBS evaluation algorithm will be a more practical solution.

4.1.1 NURBS Evaluation Techniques

Many early high-quality renderings of curved surfaces used ray tracing. [Toth, 1985] and [Nishita *et al.*, 1990] perform ray tracing on parametric and rational surfaces by solving for the ray-surface intersection point using numerical methods. [Martin *et al.*, 2000] gives a complete algorithm for ray tracing trimmed NURBS. [Pabst *et al.*, 2006] used ray casting on the GPU to render trimmed NURBS surfaces.

To take advantage of graphics hardware, parametric surfaces tend to be tessellated before display. Much work on trimmed NURBS focuses on the trimming aspect. The OpenGL version

1.1 implementation renders trimmed NURBS surfaces using the method presented in [Rockwood *et al.*, 1989] for trimmed parametric surfaces, which divides the parametric domain into patches based on the trim curves. These patches are then tessellated in the 2D domain and then evaluated to find the surface point coordinates. This algorithm is still used in the current version of OpenGL. However, in our experience the OpenGL implementation tessellates trimmed NURBS surfaces incorrectly at trim curve concavities. In addition, being a CPU evaluator, it is not fast enough to render large numbers of trimmed NURBS surfaces at interactive rates.

Previous work such as [Kumar and Manocha, 1995; Kumar *et al.*, 1996; Kahlesz *et al.*, 2002] displayed NURBS after first converting them to Bezier patches and converting the trimming curves to Bezier segments, since Bezier evaluation is less computationally demanding. These patches were then triangulated and sent to the graphics card for display. [Guthe *et al.*, 2005; Guthe *et al.*, 2006] approximate each NURBS surface with lower degree Bezier patches, but they then evaluate the Bezier patches on the GPU after the CPU approximation step. They also introduced a LOD system for choosing the appropriate approximation patch decomposition and the sampling density. Since in general no Bezier surface of lower degree can exactly match an arbitrary degree NURBS surface, a disadvantage of this approach is that the final surface may not achieve sufficient accuracy unless it is split into many Bezier patches, increasing the number of patches by up to two orders of magnitude in their examples.

Subdivision surfaces, which have largely replaced tensor-product patches in entertainment applications where mathematical exactness is not required, have also been directly evaluated on the GPU. Prior work by [Bolz and Schröder, 2002; Shiue *et al.*, 2005] focused on using a fragment program to compute the limit points of Catmull-Clark subdivision meshes. These methods can be extended to evaluate uniform B-spline surfaces; the limit surface of a Catmull-Clark subdivision in the absence of extraordinary points is the bi-cubic B-spline surface. However, they cannot be extended to evaluate NURBS because they do not have a subdivision scheme with stationary rules [Sederberg *et al.*, 2003; Sederberg *et al.*, 1998]. [Loop and Blinn, 2006] used the GPU to render piecewise algebraic surfaces of lower degrees. However, it is difficult to extend the method to evaluate arbitrary-degree NURBS surfaces.

These fragment-program implementations of surface evaluation of subdivisions were not fast enough for real-time interaction with a large number of surfaces because the evaluated surface coordinates had to be read back from an off-screen pixel buffer using an expensive p-buffer switch for each surface. [Guthe *et al.*, 2005] overcomes this issue by using a vertex program, but this method is not as flexible because the number of parameters that can be passed to a vertex program is quite limited, and vertex texture fetches are possible only on more recent graphic cards. Thus, they approximated the original input by a hierarchy of bi-cubic Bezier patches to limit the amount of data that needed to be transferred per patch. In our approach, we use a fragment program but get around the p-buffer switch issue by using a frame buffer object, which renders directly to a texture, and a vertex buffer object, which takes this texture as input coordinates for a subsequent rendering pass.

Recently, [Kanai, 2007] developed a fragment program based NURBS evaluation that closely resembles our method. However, their implementation required different fragment programs for surfaces of different degrees. While this method is theoretically capable of evaluating any NURBS surface, its implementation becomes tedious since different fragment programs have to be written specifically for each possible degree of a NURBS surface that may be present in a model. Hence a unified evaluation method that can be used to evaluate arbitrary degree NURBS surfaces is preferred.

4.2 GPU Evaluation and Rendering Algorithm

Our NURBS evaluation algorithm consists of two steps: the first step is to evaluate the NURBS basis functions and the second step is to multiply these basis function values with the control points to get the curve or surface point coordinates. This is a multi-pass algorithm that uses fragment programs to evaluate the surface point coordinates without any approximations. For rendering trimmed-NURBS surfaces, we make use of our evaluation algorithm to evaluate points on the surface uniformly spaced in the parametric domain. We then use the GPU to trim the unwanted parts of the surface while rendering. The density of the mesh or the number of points at which the surface is evaluated is based on the view position (Section (4.6.2)). The trimming operation is directly adapted from the approach by [Guthe *et al.*, 2005]. In our implementation, the trimming curves are evaluated and the trim-texture is generated using alpha blending in the graphics card. Finally, while rendering the surface, the actual trimming of the surface is performed on the GPU using another fragment program. Thus, trimming is completely decoupled from surface evaluation. The flow of the different operations, some of which are performed on the CPU, is shown in Figure (4.2).

To obtain optimum performance, we distribute the different operations to be performed either on the CPU or on the GPU. Inherently serial operations, such as calculation of the knot array, are better suited to be performed on the CPU. Operations like basis function evaluation and NURBS surface point evaluation are numerically intensive operations well suited for the better floating-point performance of the GPU. Hence, we parallelize these operations and perform them on the GPU. However, even though curve evaluation can be performed on the GPU, the performance gains, if any, are small (see Section 4.4.3). Hence we perform curve evaluation on the CPU itself.

4.3 NURBS Basis Function Evaluation

The first step in NURBS curve or surface evaluation is the calculation of the B-spline basis functions, which are dependent only on the knot vector and the parameter value. We need to transfer the information corresponding to the knot values to the GPU in order to calculate the basis function values. For this purpose, we generate a knot array texture on the CPU. The algorithm by [Kanai, 2007] on the other hand, performs this operation using binary-search on the GPU. We perform this on the CPU since the operation does not involve numerically intensive calculations; performing it

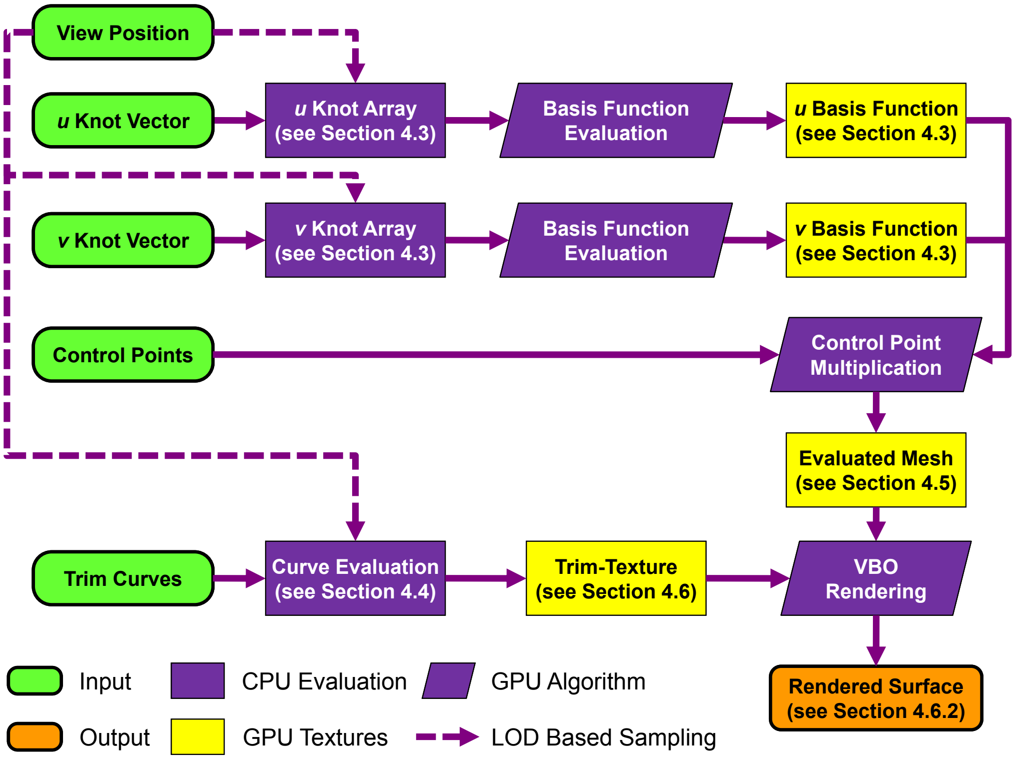

View Position
u Knot Vector
u Knot Array (see Section 4.3)
Basis Function Evaluation
u Basis Function (see Section 4.3)
v Knot Vector
v Knot Array (see Section 4.3)
Basis Function Evaluation
v Basis Function (see Section 4.3)
Control Points
Control Point Multiplication
Evaluated Mesh (see Section 4.5)
Trim Curves
Curve Evaluation (see Section 4.4)
Trim-Texture (see Section 4.6)
VBO Rendering
Rendered Surface (see Section 4.6.2)
Input
CPU Evaluation
GPU Algorithm
Output
GPU Textures
LOD Based Sampling

Figure 4.2: *Algorithm for rendering trimmed NURBS surface.*

on the CPU will make the algorithm balanced in terms of CPU/GPU workload.

The knot array texture has the value of the parameter u in the first column; it has dimensions of width $2k+1$, where k is the order of the NURBS curve, and height equal to the number of evaluation points. The remaining columns have the $2k$ knot values for the evaluation of the corresponding non-zero basis function values for a particular evaluation point. An example of such a knot array is shown in Figure (4.3), where the values are depicted as a color plot for clarity. This is a sample knot array for evaluating a cubic NURBS curve at 100 evaluation points with equally spaced parameter values u from 0 to 1. For this example, the knot vector from which the eight relevant knot values for each value of u are taken is
$[0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.1 \quad 0.1 \quad 0.5 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0]$.

Calculation of the basis function (Figure (4.4)) is done by constructing the higher order basis functions from the lower order basis functions on the GPU. The first-order (zero-degree) basis function, being the digital impulse function, is common for all evaluation points. It is a vector of size $k+1$ and is of the form shown in Equation (4.1).

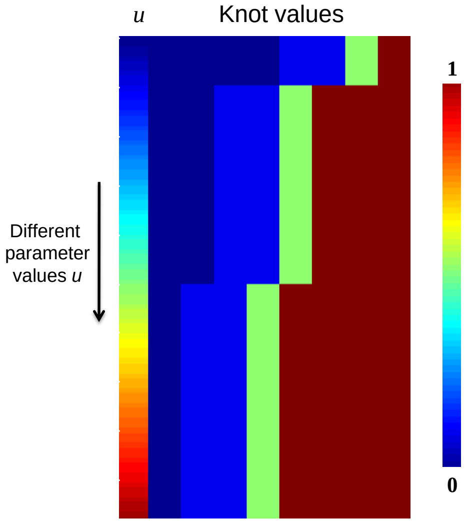


Figure 4.3: *Knot array; knot values that are transferred to the GPU as a texture depicted as a color plot.*

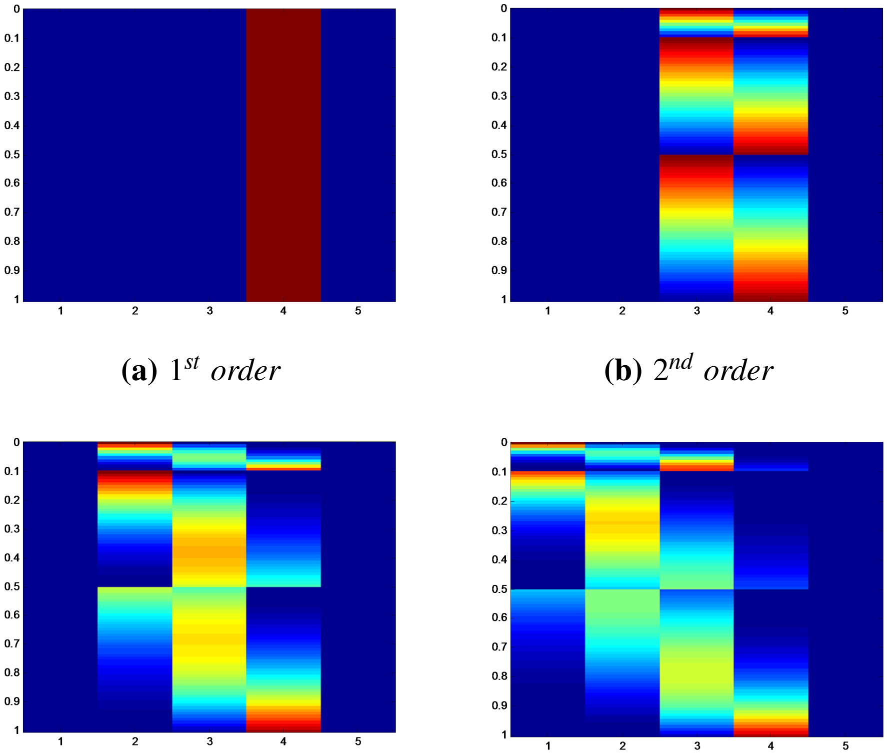
$$\begin{bmatrix} \underbrace{0 \quad 0 \quad \ldots \quad 0}_{k-1} & 1 & 0 \end{bmatrix} \tag{4.1}$$

This vector is generated on the CPU; Figure (4.4(a)) shows the generated first-order basis functions for the cubic NURBS curve (order 4) for 100 evaluation points. The generated first-order basis function is then transferred to the graphics card and stored there as a texture, call it $tex1$. The second-order basis function is computed from $tex1$ and the knot array using a fragment program and is directly rendered to another texture, call it $tex2$, using the frame buffer object. The third-order basis function is then similarly computed using $tex2$ as input and rendering back to $tex1$. Thus by alternatively using $tex1$ and $tex2$, the higher order basis functions are calculated; a fourth-order basis function is calculated at the end of the third pass. In general, a k^{th}-order basis function is computed in $k-1$ passes. Algorithm (4.1) gives the algorithm for computing the higher order basis function values from the lower order basis function values and Figure (4.4) shows the output during intermediate passes while computing a fourth-order basis function. This “ping-pong” technique of computing back and forth between two textures is commonly used in GPU programming to deal with cases where the output from an intermediate computation is required at a later stage. The last column is always 0 during the evaluation; however we still store the values in the texture to prevent introducing a branch in the code for evaluation. The additional 0 column unifies the code for evaluation since the access pattern is the same for evaluating all higher-order basis functions.

```
Input  : $k$: Order, $n$: Number of evaluation points,
         $B_1(n \times (k+1))$:Basis array, and $K(n \times (2k+1))$: Knot array
Output : $B_k(n \times (k+1))$: Basis function values.

1. For p = 2 to k
  2. For each $(i, j)$ in parallel
    3. $u \longleftarrow K(0, j)$.
    4. $B_p(i, j) \longleftarrow B_{p-1}(i, j)\frac{u-K(i,j)}{K(i+k-1,j)-K(i,j)} + B_{p-1}(i+1, j)\frac{K(i+k,j)-u}{K(i+k,j)-K(i+1,j)}$.
```

Algorithm 4.1: *Algorithm to evaluate basis function values in parallel using the GPU.*



(a) 1^{st} *order*

(b) 2^{nd} *order*

(c) 3^{rd} *order*

(d) 4^{th} *order*

Figure 4.4: *Intermediate basis function values depicted as a color plot while computing a cubic basis function on the GPU. The color values correspond to basis function values between 0 and 1.*

4.4 Curve Evaluation

Following [Piegl and Tiller, 1997], we can break computing the coordinates of a point on a NURBS curve given a parameter value u into these three steps:

1. Find the knot span $[u_i, u_{i+1})$ in which u lies, i.e. $u \in [u_i, u_{i+1})$.
2. Compute the corresponding non-zero basis function values $N^p_{i-p}(u), \ldots, N^p_i(u)$.
3. Multiply the non-zero basis function values with the corresponding control points and sum the results.

The first step, finding the knot span in which u lies, is performed on the CPU; this operation is essentially performed while generating the knot array on the CPU. The basis function values corresponding to each control point are then evaluated using a fragment program on the GPU. Finally, the actual curve points are evaluated by multiplying out the values of the basis functions and the corresponding control points, and then adding them together using another fragment program. For clarity, we first describe our procedure for calculating a NURBS curve point without any packing of data or optimization in the following two sections. Details of our data packing and optimizations are presented separately in Section (4.4.2).

4.4.1 Basic Algorithm

We first compute the basis function values using the GPU evaluation method described in Section (4.3). Once the basis function values are calculated, the next step is to multiply these values with their corresponding control points. For this, another array with the corresponding control points for each parameter value to be evaluated is created on the CPU. This control point array is an array of width k, with the x, y, z and w values stored in the *RGBA* channels.

```
Input  : $B_k(n \times k)$:Basis array, and $P(n \times k \times 4)$: Control point array
Output : $M(n \times k \times 4))$: Multiplied values.

For each $(i, j)$ in parallel
    $M(i, j, 0) \longleftarrow B_k(i, j) * P(i, k, 0)$
    $M(i, j, 1) \longleftarrow B_k(i, j) * P(i, k, 1)$
    $M(i, j, 2) \longleftarrow B_k(i, j) * P(i, k, 2)$
    $M(i, j, 3) \longleftarrow B_k(i, j) * P(i, k, 3)$
```

Algorithm 4.2: *Algorithm to multiply the basis function values with the control points in parallel using the GPU.*

The control point array is multiplied with the basis function array calculated in the previous step as shown in Algorithm (4.2). A fragment program then multiplies all the four channels of the control point array simultaneously with the basis function values. The resulting array is then "reduced" (summed) along the width direction to its per-row sum to obtain the actual curve positions using a different fragment program. The sequence of steps for calculating the final point coordinates is shown graphically in Figure (4.5).

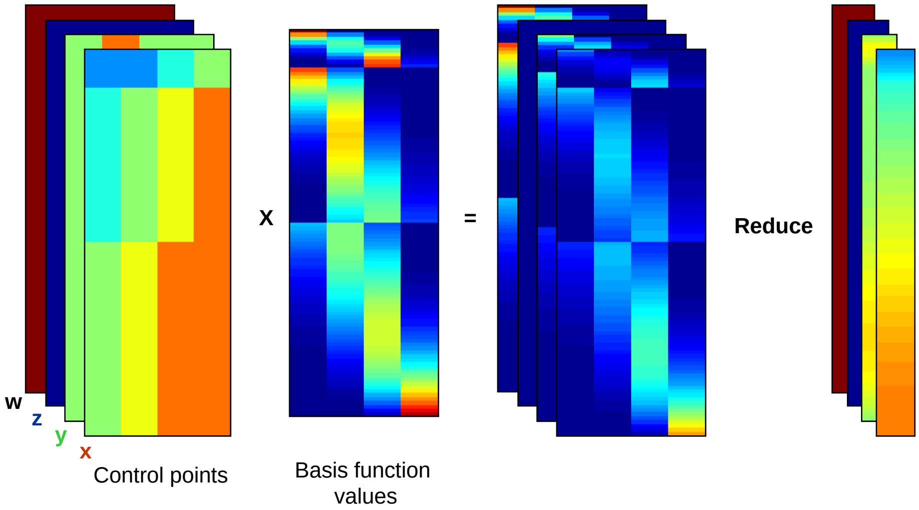


Figure 4.5: *Sequence of steps for curve point evaluation. The control points are first multiplied with the corresponding basis function values and then summed to get the curve coordinates.*

4.4.2 Optimization and Packing of Data

The previous section described our method for curve evaluation without any packing of data or optimization. We now describe two techniques that reduce the evaluation time, data packing and index arrays.

GPU calculations are performed simultaneously on all four channels (RGBA); therefore using only one channel for the calculations leads to wasted resources. Packing of data refers to using the four channels to store and process the data instead of using just a single channel. By packing the data in the knot array in an intelligent manner, we can save storage space as well as speed up the computations. The data can be packed either in the width direction or in the height direction. However, since the width of the array is dependent on the order of the basis function being evaluated, packing it in the width direction will necessitate the use of different fragment programs for different degrees of the curves being evaluated. This will make the implementation tedious because the program for the packed version cannot be directly extended from the non-packed version. It is also not practical because different programs have to be developed, one each for each different degree of curve being evaluated.

The data required for the calculation of the B-spline basis function is completely contained in each row of the knot array. Hence, it will be simpler to pack the data along the height direction with each channel corresponding to different evaluation points as shown in Figure (4.6). The first entry of each channel in the row specifies the parameter value at which the basis functions are to be evaluated. This kind of packing is also easy to implement since it directly extends from the non-packed version, requiring only minor changes to the fragment program. In addition, the data access from lower degree basis function to evaluate higher degree basis functions in the fragment program remain the same for a particular evaluation point. It is also not required to have different programs based on the order of the curve being evaluated; the same program generalizes to any order.

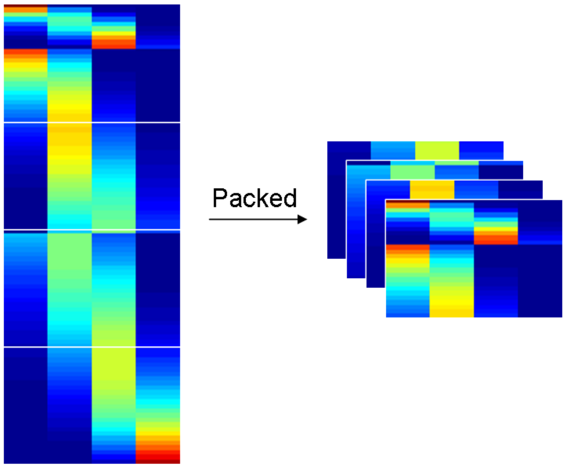


Figure 4.6: *Packing of the knot and basis function data reduces data transfer and GPU computations.*

However, there is a disadvantage in packing the data for basis function evaluation. NURBS curves with repeated knot values give rise to the special 0/0 case in their evaluation, which we need to yield a result of 0 rather than the NaN specified by IEEE standards. Although many older GPUs we tested return the non-IEEE-compliant 0 that we desire, for greater portability and forward-compatibility we explicitly check for these special cases. Moreover, since the current generation GPUs are moving towards IEEE-compliance, they return a NaN value. Since these 0/0 cases have to be separately handled for each channel, it leads to numerous *if* statements in the fragment program, increasing its length. Older graphics cards evaluate both branches of if statements and hence they can slow down computation. However, the performance drop due to these statements in our implementation is negligible if any. The difference in the total evaluation timings even in older cards like the ATi Mobility Radeon 9700 is less than 5% for the largest evaluation size. Newer

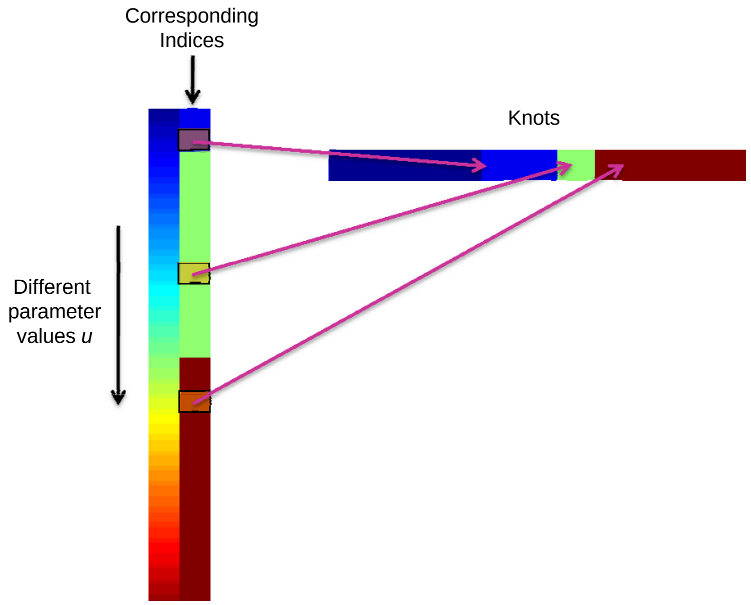


Figure 4.7: *Using an index array to prevent data duplication.*

graphics cards have hardware support (dynamic flow control) for branching and hence this is not a major problem.

We now describe the second, alternative optimization technique we implemented. In the evaluation of the basis function in the example given in the previous section, many knot values were repeatedly used. For example, the knot values required for the computation of the first 10 parameter values shown in Figure (4.3) use the same knot values. One method to reduce the amount of data transfer in such cases is to use an index array, which contains indices pointing to the knot values needed for the basis function evaluation. The knot values are stored separately in another array and are transferred directly from the CPU to the GPU. The knot array will then only contain the parameter value and the index of the first element in the knot vector required for the evaluation of the basis functions (Figure (4.7)).

Using an index array also has its advantages and disadvantages. There is an obvious reduction in data transfer. On the other hand, the GPU architecture is not optimized for such texture indirections or nested texture fetches. The cache is optimized to retrieve data quickly from nearby memory locations; the cache misses are presumably the reason that too many texture indirections can significantly slow performance by introducing too much latency (latency that can no longer be hidden by the parallel nature of fragment processing). In addition, the indexed data cannot be packed anymore because the different channels will point to different knot positions. Hence even if the data is packed, it will require four texture fetches that offset the advantage gained by packing. Therefore, we cannot combine our two techniques (data packing and index based).

4.4.3 Curve Evaluation Timings

Using the above variations of the GPU algorithm, we timed the evaluation of NURBS curves on different GPUs. Timings were done on four different implementations: CPU, GPU packed, GPU non-packed, and GPU index-based. The non-packed implementation is the regular implementation without any packing or indexing as described in Section (4.4.1).

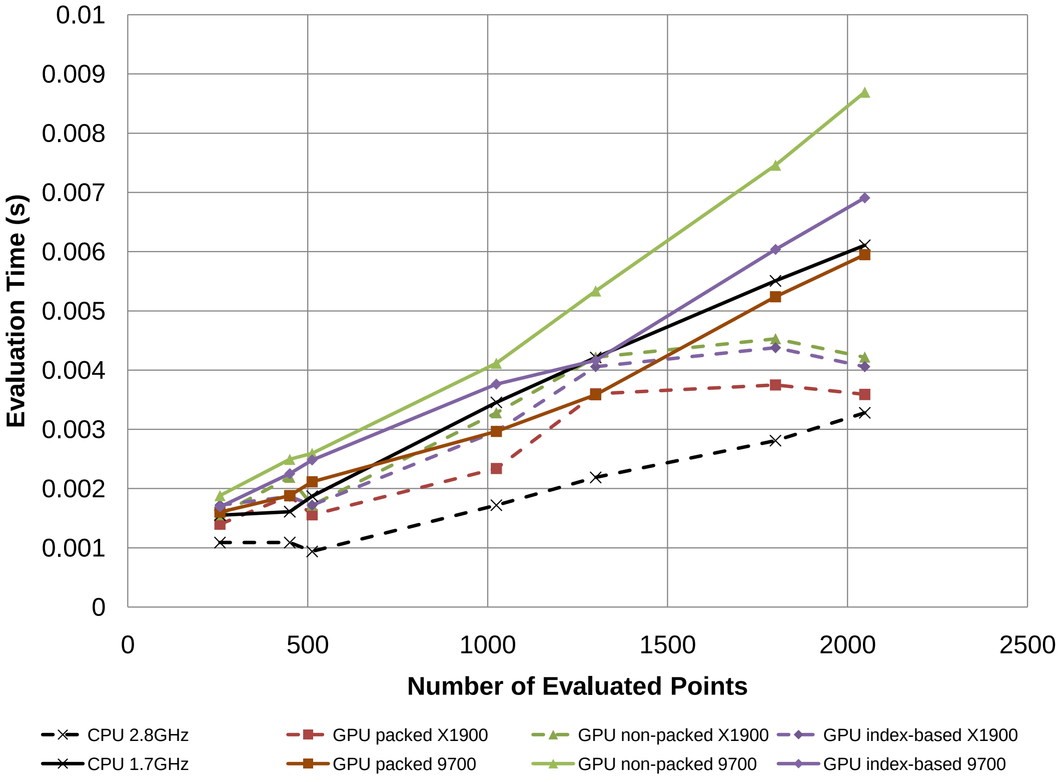


Figure 4.8: *Time for evaluating a cubic NURBS curve on two different GPUs.*

Figure (4.8) shows the curve evaluation timings for a cubic NURBS curve with different numbers of evaluation points evaluated on ATi Mobility Radeon 9700 and ATi Radeon X1900 graphics cards. The CPUs used for the evaluation were Intel Centrino 1.7GHz and Intel Pentium-4 2.8GHz processors respectively. As expected, the evaluation time increases roughly linearly with the number of points evaluated. It can be seen that the packed method is a bit faster than the 1.7GHz CPU evaluation. However, the other methods are slower than the CPU method on both platforms, either due to the amount of data transferred in the case of the unpacked implementation or due to the texture indirection in the case of the index-based implementation. Evaluation timings on other GPUs also followed the same qualitative trend, with the packed version always the fastest of the GPU methods.

From these results for 2D NURBS curves, it is not immediately clear that a GPU implementation for NURBS surface evaluation will be enough of an improvement over CPU evaluation to justify the development effort. However, in the case of surface evaluation, with its higher arithmetic intensity, the GPU win over CPU is far more pronounced, as described in the later results section. Since we found the GPU packed method of evaluating the basis functions to be the fastest of the three different techniques we developed, we use this method in the surface evaluation algorithm. Since the surface control points used for surface evaluation are already 4-component vectors (XYZW), additional data packing is not required for surface evaluation.

4.5 NURBS Surface Evaluation

Given all the data for a NURBS surface, our surface evaluation algorithm computes the surface point coordinates at parametric coordinates (u,v) in the following manner.

1. Locate the lower-left corner of the sub-mesh of control points that influence the evaluation point coordinates.
2. Compute the non-zero basis functions along the two parameter directions.
   (a) Compute the non-zero u basis functions using the u direction knot vector.
   (b) Compute the non-zero v basis functions using the v direction knot vector.
3. Multiply the non-zero basis functions with their corresponding control points from the sub-mesh and sum the results.

The first step of computing the lower left corner control point that influences the current surface point coordinate is equivalent to the first step in the curve evaluation; it is done on the CPU and transferred as a 1D texture to the graphics card. The two substeps of the second step are each performed in the same manner as computing the basis functions for curve evaluation explained in Section (4.3). Finally, the evaluated basis functions are multiplied with the corresponding control points and added together, as explained in detail below.

Figure (4.9) represents the surface evaluation process pictorially. We specify the parametric u and v coordinates of the points required to be evaluated on the CPU. We then calculate the basis functions corresponding to these coordinates on the GPU using the basis function evaluation algorithm defined in Section (4.3) and generate the two textures for u and v having the basis function values at the required parameter coordinates. We implemented the packed version of the basis function evaluation algorithm because it was the fastest among the different methods discussed in Section (4.4.2).

Once the basis functions are evaluated, we again alternate (ping-pong) between output textures to evaluate the final surface coordinates. We store the control point data in a texture of size $n \times m$

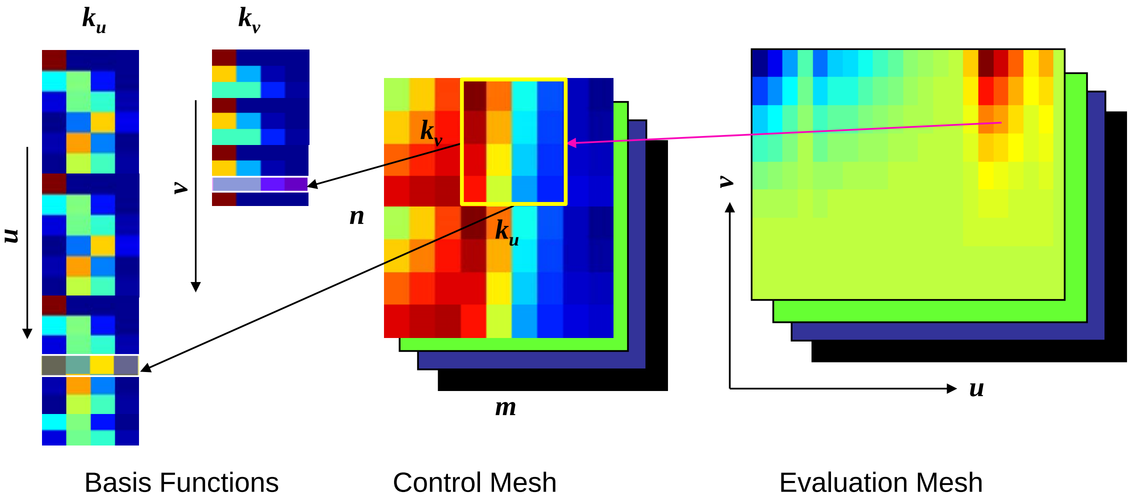


Figure 4.9: *Graphical representation of the surface evaluation algorithm.*

in the GPU memory. We also have a texture of size equal to the evaluation mesh, call it $tex1$, which is initialized to zero. Given a particular u and v coordinate, we look up the coordinates of the control point that influences the current evaluation point using the index values stored in the 1D textures calculated in step 1. We then multiply this control point with its corresponding u and v basis function values and add it to the corresponding pixel in $tex1$ using a fragment program. This fragment program directly renders the multiplied result to another texture, call it $tex2$. In the next pass, the newly multiplied values of this pass are added to $tex2$ and rendered directly back to $tex1$. Thus, the final curve point is evaluated in $k_u \times k_v$ passes; for example, a bi-cubic NURBS surface point is evaluated in 16 passes. In our current implementation, since we evaluate each surface separately, it does not matter if the processed surfaces have different degrees.

4.5.1 Dynamic LOD

The NURBS patches that make up a particular model or a scene are usually of different sizes and at different magnification levels. In such cases, it would be inefficient to evaluate all the surfaces at the same level of detail. Therefore, we use different evaluation grids for different surfaces based on the size of the surface and the distance of the surface from the eye point. Older graphic cards were optimized to only work with square power-of-2 textures. Hence, the transitions between the different LODs are not smooth, leading to popping artifacts between them. Furthermore, it was not possible to have different number of evaluation points along the u and v directions. However, newer graphic cards support rectangular textures of any size. Thus, for the different LODs, the number of evaluation points change continuously from the minimum to the maximum value in our implementation. In addition, the number of evaluation points are different for the u and v directions. This leads to a better rendering of dynamic scenes encountered in interactive environments like solid modeling. Figure (4.10) shows a duck model rendered at different zoom levels. The LOD varies continuously between the different levels, resulting in smooth transitions.

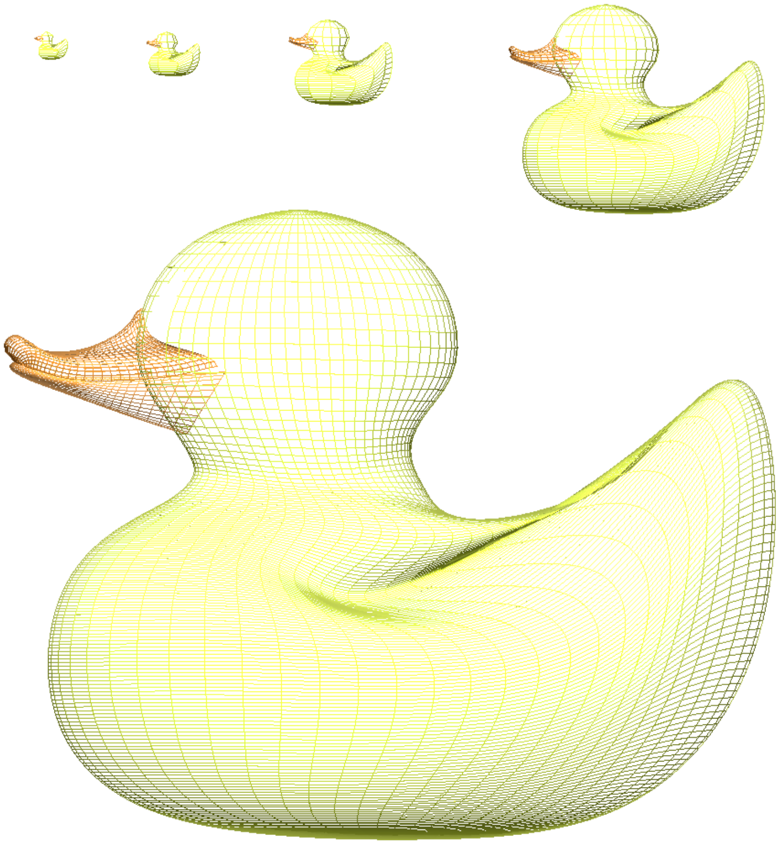


Figure 4.10: *Dynamic LOD: Duck rendered at different resolutions based on the required LOD.*

We compute the required height and width of the evaluation mesh by finding the distance of the object from the eye point as well as the size of the object. Then the connectivity of the points is generated on the CPU using the selected size. We make use of the fact that the connectivity of a 2D mesh in the parametric domain is the same as the connectivity of the final NURBS surface. This index information is sent to the graphics card and the surface is rendered by using the corresponding point coordinate data taken directly from a texture using a vertex buffer object. This way we eliminate the redundant and costly operation of reading back the evaluated point coordinates from the GPU and then sending them back as vertex coordinates.

4.6 Trimming

] For efficient rendering of a trimmed NURBS surface, the surface evaluation should be decoupled from trimming. Instead trimming can be performed with the help of texture mapping using a trim-texture, a trimming technique first applied to trimmed spline surfaces by [Guthe *et al.*, 2005].

The trim-texture is generated by evaluating and rendering the trim curves in the 2D parametric domain. Even though NURBS curves can theoretically be used for trim curves, most of the trim curves in practice are piecewise linear segments. This is because a space curve on a 3D NURBS surface is usually approximated by linear segments in the 2D parametric domain. If the trim curves are described by splines, they can be evaluated and converted to piecewise linear segments. In our implementation, the trim curves are evaluated and rendered directly to a trim-texture.

4.6.1 Trim Texture Generation

As described in [Woo *et al.*, 2004], arbitrary concave polygons (possibly even including holes) do not need to be tessellated for rendering. Instead, triangles connecting a common origin to each polygon edge in turn are rasterized, but only those regions that are filled an odd number of times are finally rendered. This is shown in Figure (4.11), where only parts of the domain that are rendered once or thrice are considered to be the part of the surface that is to be finally rendered. Another advantage of using such an algorithm is that the orientation of the holes and holes within holes need not be explicitly dealt with as separate cases.

The above algorithm can be implemented either by using the stencil buffer or by alpha blending. Using the stencil buffer is sufficient to trim surfaces that are parallel to the view plane; implementation details for using the stencil buffer are given in [Woo *et al.*, 2004]. However, we use an alternate implementation based on the alpha blending functionality of graphics cards to generate the trim-texture because the trimmed surfaces may be arbitrarily oriented or curved.

Some basic preprocessing is required for using alpha blending, as explained below. The viewport is set up to match the size of the trim-texture, which is determined based on the required LOD, as in [Guthe *et al.*, 2005]. The Model View matrix is set to 2D mode with view area from [0 1] in both width and height. For planar faces, the two directions correspond to the two orthogonal directions defining the coordinate system in the plane of the face; for non-planar faces, the parametric u and v directions that define the texture coordinate system are used. The background color is cleared to (0, 0, 0, 0). The required blending factors are chosen to perform an odd/even count. This can be done by toggling the existing value from 0 to 1 or 1 to 0 whenever a new fragment is drawn over it. Once all the parameters are set up, a triangle fan is drawn with color (1, 1, 1, 1). Thus, the algorithm can be easily extended to complex shapes like fonts or irregular holes.

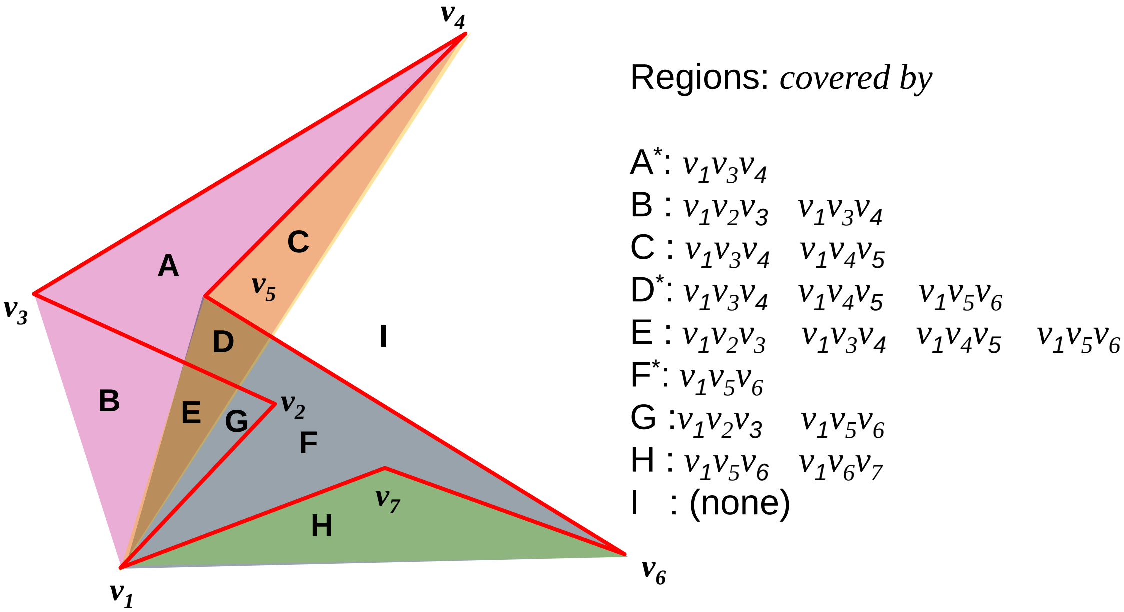
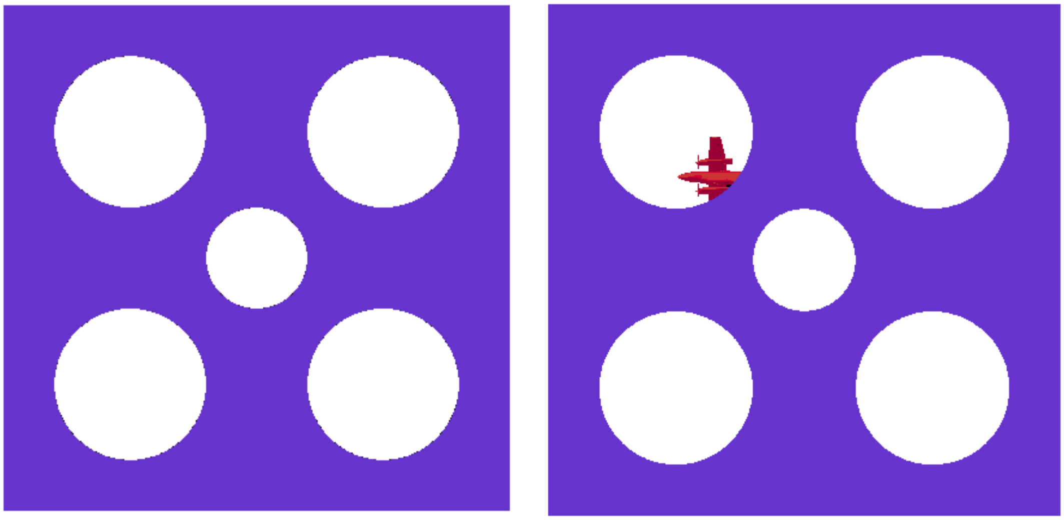

v_4
v_3
v_5
v_2
v_7
v_1
v_6
A
B
C
D
E
G
F
H
I
Regions: *covered by*
A*: $v_1v_3v_4$
B : $v_1v_2v_3$ $v_1v_3v_4$
C : $v_1v_3v_4$ $v_1v_4v_5$
D*: $v_1v_3v_4$ $v_1v_4v_5$ $v_1v_5v_6$
E : $v_1v_2v_3$ $v_1v_3v_4$ $v_1v_4v_5$ $v_1v_5v_6$
F*: $v_1v_5v_6$
G :$v_1v_2v_3$ $v_1v_5v_6$
H : $v_1v_5v_6$ $v_1v_6v_7$
I : (none)

Figure 4.11: *Adapted from the OpenGL Programming Guide [Woo et al., 2004]: Example of a trim-texture. Each of the region names is followed by a list of the triangles that cover it. The uniformly shaded regions A, D, and F make up the original polygon; note that these three regions are covered by an odd number of triangles. Every other region is covered by an even number of triangles (possibly zero). Thus, only the regions that are rendered an odd number of times (starred regions) are finally displayed.*

4.6.2 Rendering

The trim-texture is then used to mask parts of the surface using a fragment program during the rendering pass. Even though the trim-texture has alpha values that can be mapped directly to the surface by using alpha blending, this may lead to incorrect results. One such example is shown in Figure (4.12(a)), where alpha blending is used to cut the holes for a scene with an airplane inside a box. The correct rendering in seen in Figure (4.12(b)). Unless all the objects are rendered in back-to-front order, the blending will not be correct; the objects behind discarded trim portions will not be rendered. The problem becomes even more pronounced in the case of curved surfaces, where the surface itself may be self-occluding. In this case, since the order in which the fragments are processed by the graphics card is not defined, the final surface will be rendered incorrectly and may even have artifacts similar to self-shadowing.

To overcome this problem, only the parts of the surface that lie outside the trim curves are rendered (Figure (4.12(b))). The advantage of such a method is that the lighting calculations need not be done to those fragments that are discarded. However, this implementation uses branching and may lead to a performance drop in older graphic cards. Our fragment program used for the



(a) *Using alpha blending* **(b)** *Using fragment program*

Figure 4.12: *Difference in trimming with using alpha blending and fragment program. Alpha blending produces incorrect results.*

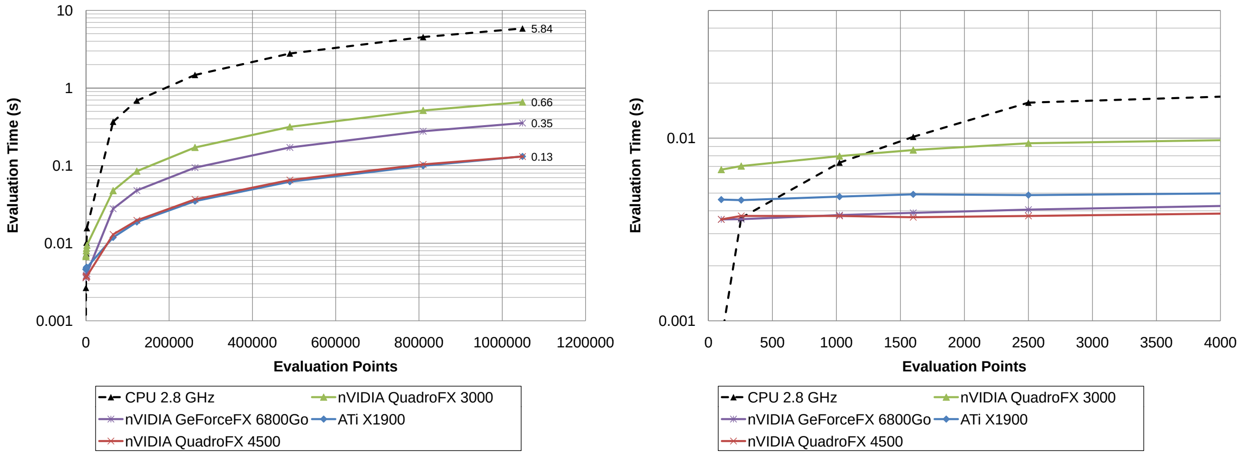
trimming operation, written in Cg [Mark *et al.*, 2003; Fernando and Kilgard, 2003], makes use of the *discard* command that kills the fragment when the value of the particular color channel used to trim is 0. To save memory we store different trim-textures in different color channels of the same texture. We then switch between the different channels while rendering different trimmed surfaces.

4.7 Results

We tested our evaluation method on the different GPU platforms listed in Table (4.1).

GPU	VRAM	CPU	RAM
ATi X1900	512 MB	2.8 GHz	512 MB
nVIDIA Quadro FX4500	512 MB	3.00 GHz	2048 MB
nVIDIA Quadro FX3000	256 MB	1.88 GHz	1024 MB
nVIDIA GeForce FX6800Go	256 MB	1.60 GHz	512 MB

Table 4.1: *Different GPU platforms tested.*



(a) *Evaluation timings* **(b)** *Timings for small evaluation grids*

Figure 4.13: *Log-scale comparison of evaluation timings for a bi-cubic NURBS surface with increasing number of evaluation points.*

Figure (4.13(a)) compares the evaluation timing alone of a single bi-cubic NURBS patch defined by 144 control points when increasing the density of the evaluation grid. The evaluation time includes the time taken to generate the knot array and control-point array on the CPU; the timings will remain the same even if the user interactively changes the knot values or the control points. The GPU-based evaluation is faster than the CPU-based evaluation by a factor of about 50 when evaluated at a large numbers of evaluation points. However, the GPU evaluation has more overhead for very small patches and hence is not suitable for evaluating surfaces having less than 16×16 evaluation points (Figure (4.13(b)). The nVIDIA QuadroFX 3000 is an older graphics card and uses AGP8x bus architecture. Hence, the data bandwidth is not as high as the other PCI-e graphics cards tested. As a result, the timings are somewhat slower but still about 10 times faster than on a CPU. The high end PCI-e 16x graphics cards from both ATi and nVIDIA produced almost identical results.

The duck model shown in Figure (4.1) consists of three NURBS surfaces with both non-uniform knots and non-unity weights for the control points. One of the three surfaces in the model is also trimmed. Figure (4.1) is rendered using an evaluation grid of 64×64 points for each surface on a window of size 1280×1024. Note that the trimmed yellow patch representing the duck's body fills most of this window, but has no obvious tessellation artifacts with this sampling density. This evaluation grid is similar to the one shown for the largest duck in Figure (4.10). In addition, the model can be interactively displayed with varying LODs without re-sending the data to the GPU repeatedly. Similarly, any changes to the model will necessitate transferring only the control points to the GPU.

Figure (4.14) compares the frame rates for an animated scene containing many such ducks

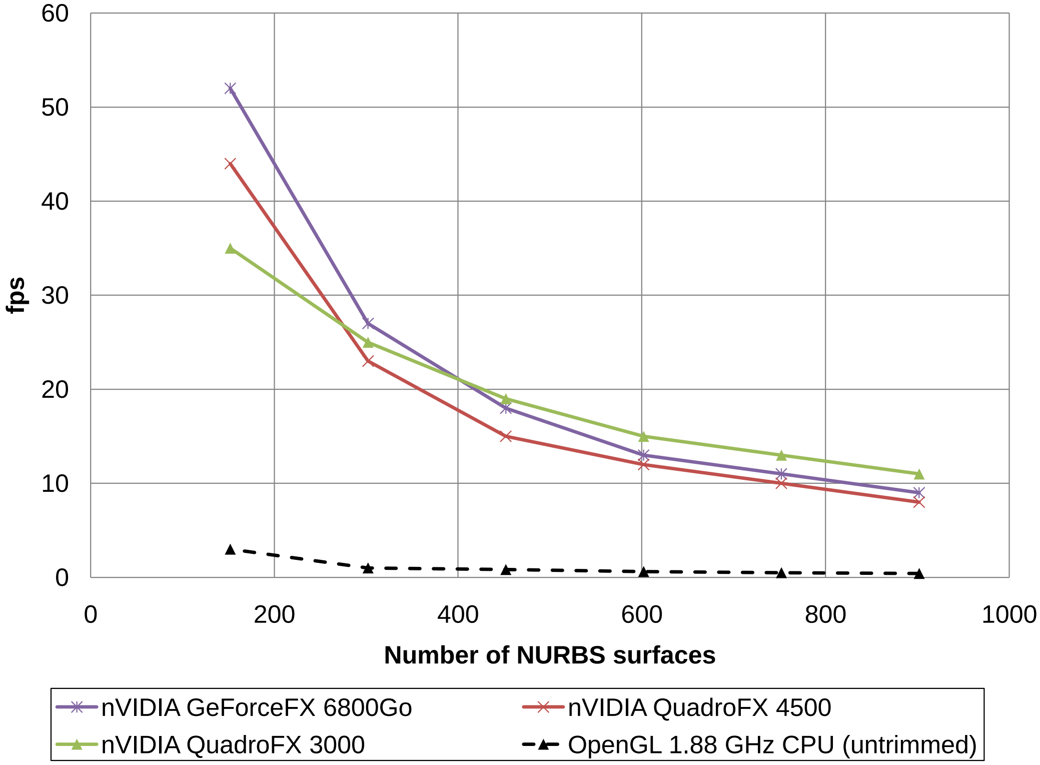


Figure 4.14: *Comparison of frame rates with different nVIDIA graphics cards. One-third of the total NURBS surfaces are non-trivially trimmed.*

swimming in a (tessellated) teapot, similar to Figure (4.1), using our GPU implementation and with the CPU OpenGL implementation. The scene is again rendered in a window of size 1280 × 1024; the individual NURBS surfaces, being smaller than the full screen area, were evaluated on a 16 × 16 grid of evaluation points. One-third of the NURBS surfaces were non-trivially trimmed. As expected, the frame rate decreases with the increase in the number of surfaces. However, the decrease in frame rate is not linear in the number of surfaces. This may be due to the extra overhead of transferring the control points data for a large number of surfaces to the graphics card and some overhead in switching between the VBO of different surfaces. Even though trimming was not performed while obtaining the OpenGL-rendered timings, its frame rates are unacceptably slow for more than about 100 NURBS surfaces, consistently 40-50 times slower than our GPU-based implementation. In addition, the OpenGL implementation had rendering artifacts at trim curve concavities while rendering trimmed NURBS surfaces (Figure (4.15)).

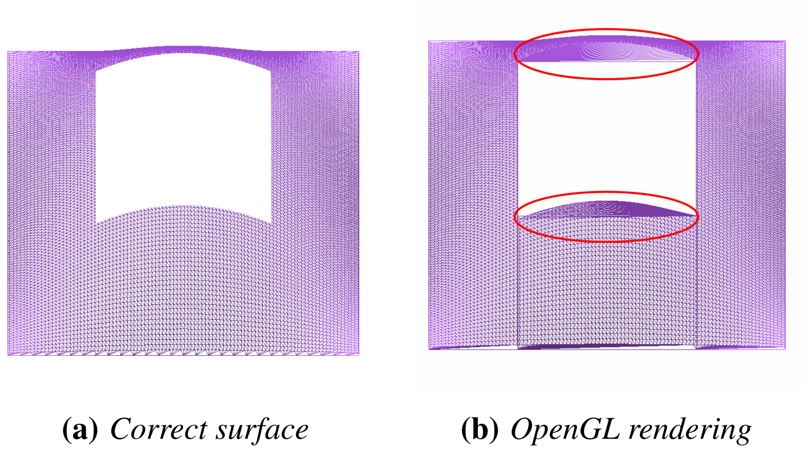

(a) *Correct surface* (b) *OpenGL rendering*

Figure 4.15: *Trimmed NURBS surface rendered incorrectly by OpenGL. The figure on the left shows the correct trimming.*

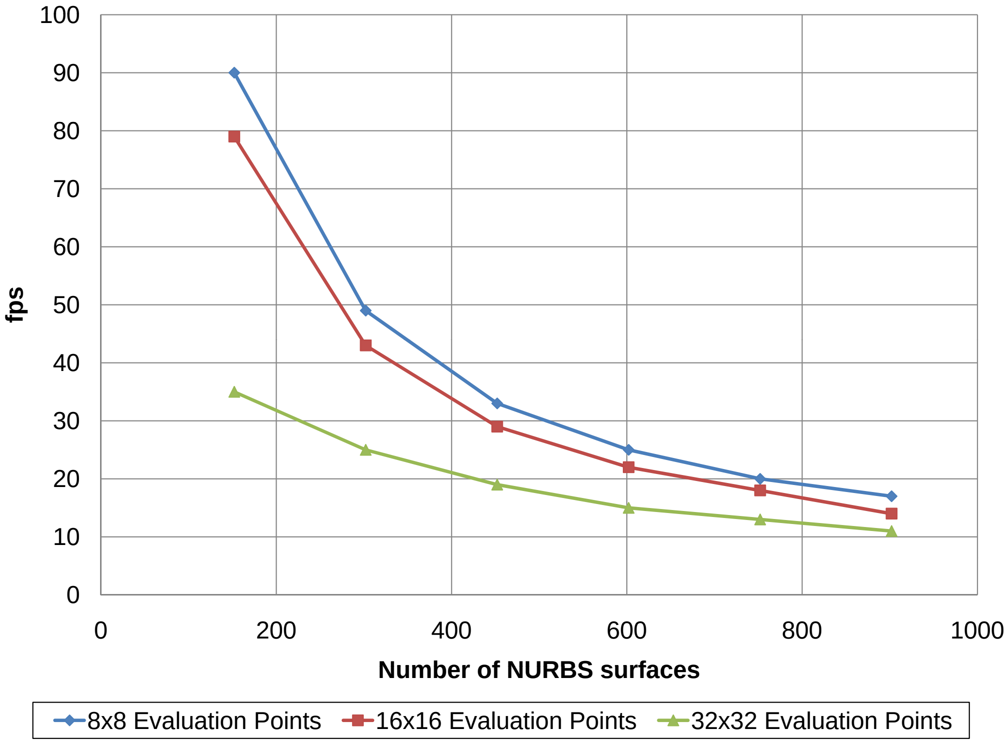


Figure 4.16: *Comparison of frame rates with varying per-patch evaluation grid size on nVIDIA Quadro FX3000 graphics card.*

Figure (4.16) shows the frame rates for animating the same scene as the above example but varying the per patch evaluation grid size as well as the number of ducks. The frame rates were timed on the nVIDIA Quadro FX3000 graphics card. The NURBS surfaces evaluated on a 32×32 grid of evaluation points was the slowest, but for a larger number of surfaces the rates start to converge.

4.8 CUDA Implementation

In order to assess the performance of our NURBS evaluation algorithm using newer GPU programming techniques, we developed a CUDA implementation. CUDA, as mentioned in Section (3.1), is NVIDIA's GPU computing architecture that allows direct general-purpose computations on the GPU without resorting to graphics primitives.

Our CUDA implementation is very similar to our GPGPU implementation described in detail in Section (4.5). CUDA supports the use of two kinds of memory on the GPU, called global memory and shared memory that is shared between a smaller set of CUDA threads. However, since we do not reuse any data in our evaluation algorithm, we make use of the global memory. CUDA also supports the use of texture memory on the GPU that is cached. On the other hand, texture memory is read-only and requires the use of the ping-pong technique to perform multi-pass computations. The advantage of using global memory is that, since it is read-write, we do not need to ping-pong to perform the operations but can modify the data in-situ.

To perform parallel computations using CUDA, the programmer has to divide the computations in to threads that are then executed as blocks by the GPU. However, there can only be a maximum of 512 threads per block. Since the majority of the surfaces we evaluated were cubic, we found that a block size of 64×4 to be optimal basis function evaluation. However, to be unbiased with respect to the parametric directions in surface evaluation, we used a 16×16 block size for the control point multiplication, since it is the largest square block size that can be evaluated using CUDA. We tried different square block sizes for the control point multiplication but found the 16×16 block size to be the optimum size for NURBS evaluation in the graphics cards we tested.

4.8.1 CUDA Evaluation Timings

We first compared the standard CUDA implementation with both our CPU and our GPGPU implementation on four different CUDA-capable NVIDIA graphics cards. Then we compared the performance of the different variations of our CUDA and GPGPU implementations on two specific cards. The specifications of the different graphics cards are given in Table (4.2) along with the details of the CPU systems.

Figure (4.17) shows the evaluation time in seconds with a varying number of evaluation points. For large numbers of evaluations points, the CUDA implementation is 3 to 4 times faster than the CPU implementation; however, the GPGPU implementation is more than 30 times faster than

GPU	Release Date	VRAM	CPU	Clock Speed	RAM
GeForce 9600GT	Feb 2008	512 MB	Intel Pentium 4	3.00 GHz	2048 MB
GeForce 9600M GT	Oct 2008	256 MB	Intel Core2Duo	2.40 GHz	4096 MB
Quadro FX5600	Sep 2007	1536 MB	Intel Core2Quad	2.66 GHz	4096 MB
Quadro FX5800	Nov 2008	4096 MB	Intel Core2Quad	2.66 GHz	4096 MB

Table 4.2: *Different GPU platforms tested.*

the CPU implementation. CUDA is slower because there is an overhead in accessing the global memory. In addition, the texture memory used in the GPGPU implementation is read-only and cached, which leads to a well-hidden memory latency in the case of texture fetches. Moreover, in the GPGPU implementation, since the output of the kernel is to a specific pixel location in the framebuffer instead of a random write, the memory writes are fast.

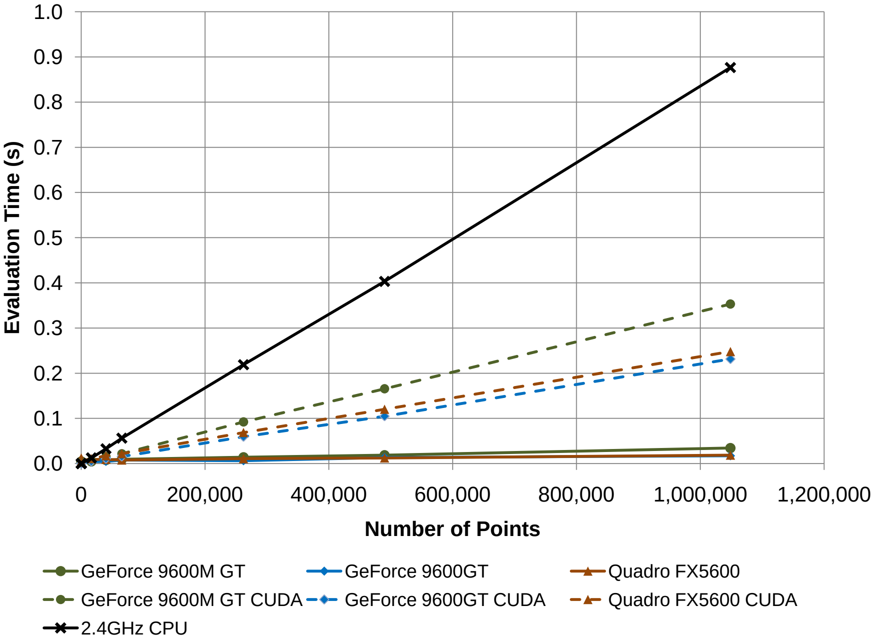


Figure 4.17: *Evaluation timings with increasing number of evaluation points for evaluating a bi-cubic NURBS surface using the CPU, CUDA and GPGPU algorithms.*

However, if we focus on smaller evaluation sizes (Figure (4.18)), the GPGPU implementation is slower than CUDA. This is due to the overhead involved in setting up the texture memory in OpenGL, which is lower in the case of CUDA. Moreover, the transition point where the OpenGL implementation is faster changes based on the type of the graphics card used. This is due the

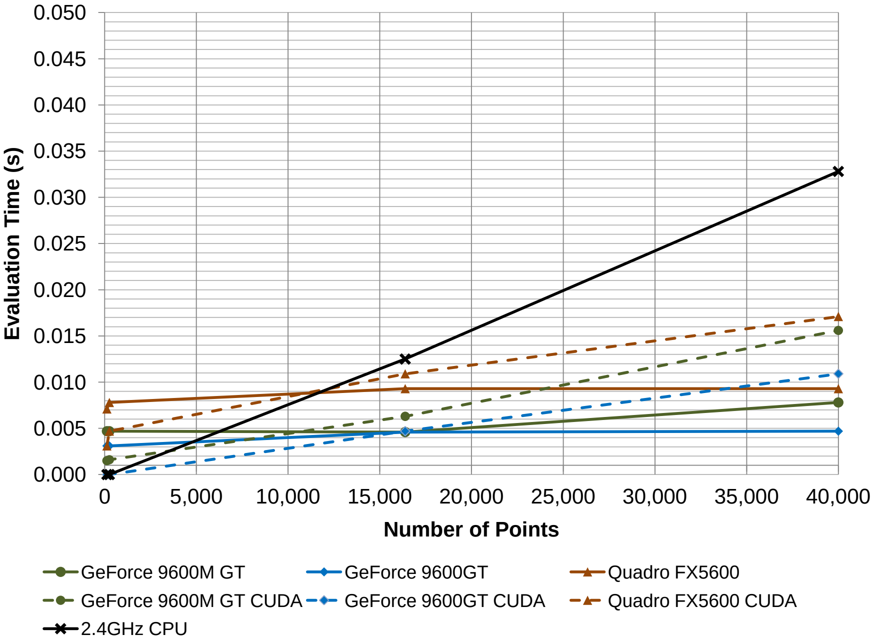


Figure 4.18: *Evaluation timings for smaller number of evaluation points. The transition point where OpenGL evaluation is faster than CUDA varies for different graphics cards.*

variation in the architecture of these graphics cards; the graphics cards tested were released to the market at different times.

We profiled the CUDA code to understand the discrepancy in the timings between the CUDA and the GPGPU implementation at higher evaluation resolution. Figure (4.19) shows the main reasons for the performance drop in CUDA; they can be classified as uncoalesced memory access and divergent branching. Coalesced access refers to sequential threads accessing adjacent memory locations in the global memory. In such cases, the memory is read as a single block and the bandwidth for memory access is high. Uncoalesced or misaligned accesses on the other hand have an effective bandwidth of only one-eighth of the coalesced access case.

As it can be seen, the percentage of uncoalesced memory load is high in basis function evaluation but is negligible in the case of control point multiplication. The uncoalesced memory access could be the primary reason why the CUDA code is slow compared to the GPGPU implementation. The percentage of uncoalesced store is also high in both steps since we output one value for each kernel call and they are all calculated independently.

Another reason for the performance drop in the case of CUDA is the large number of divergent branches in the evaluation. Divergent branches in a GPU code leads to a performance drop since the GPU has to wait until all branches of code are executed before proceeding to the next step. In CUDA, these branches are required to correctly deal with the border elements that arise due to mismatch between the block size and the evaluation grid size (Figure (4.20)). These branches are

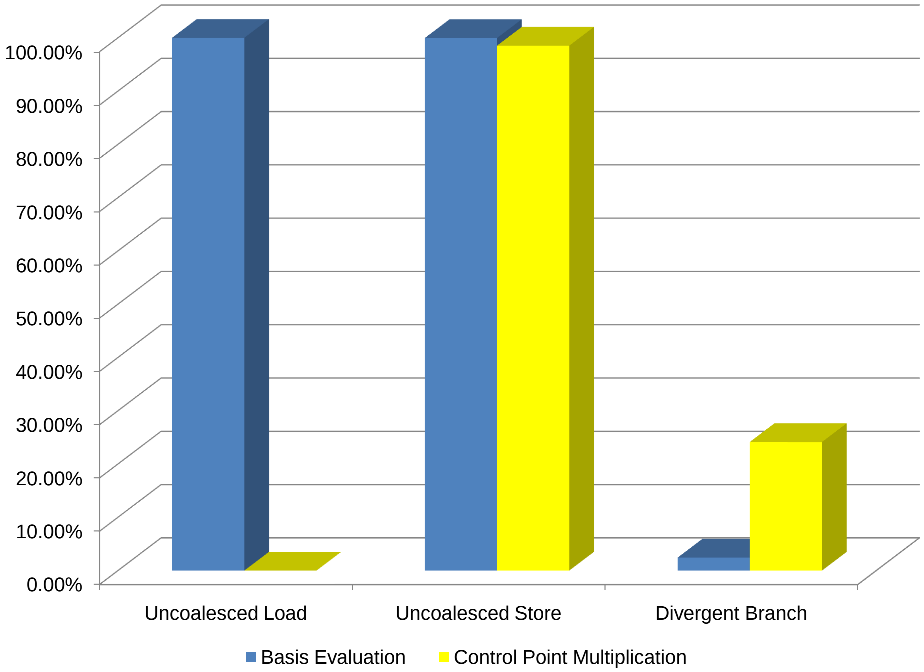


Figure 4.19: *The main reasons for the performance drop while using CUDA. High uncoalesced memory accesses and divergent branching reduce the performance of the CUDA implementation.*

absent in the case of the GPGPU implementation since there is no blocking of threads. This can lead to huge performance drops in the CUDA implementation.

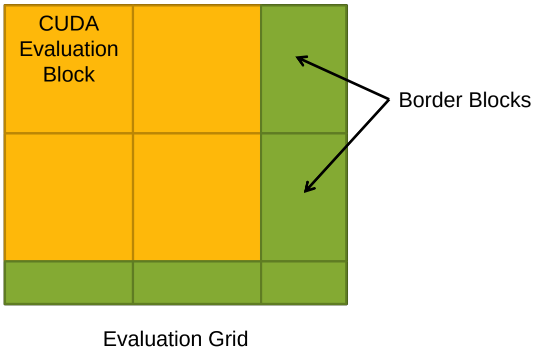


Figure 4.20: *Evaluation of the border elements results in divergent branches in the CUDA implementation.*

4.8.2 Comparison of Different Implementations

We performed detailed timings of different CUDA and OpenGL GPGPU implementations using the GeForce 9600M GT and Quadro FX5800 graphics cards. These two cards both support dynamic loops in kernels. We compared the performance of three different CUDA and GPGPU implementations. They were the basic CUDA implementation, the second one that used texture memory and the third using dynamic loops without ping-pong. The implementation not using ping-pong requires loops whose sizes vary based on the degree of NURBS surfaces being evaluated. Since the degree of NURBS surfaces can vary, the GPU has to support dynamic loops. Similarly, we also timed three different GPGPU implementations. These include a basic GPGPU implementation, the second that uses dynamic loops without ping-pong, and a third implementation that does not pack the data in textures as explained in Section (4.4.2).

Figure (4.21) compares the timings of the six different implementations between the two graphics cards. It can be seen that there is an order of magnitude difference in the CUDA performance between the two cards compared to only a factor of two performance difference in case of the GPGPU implementation. One possible reason for the performance difference could be the fact that the Quadro FX5800 is a newer card that has built in optimizations for uncoalesced memory reads.

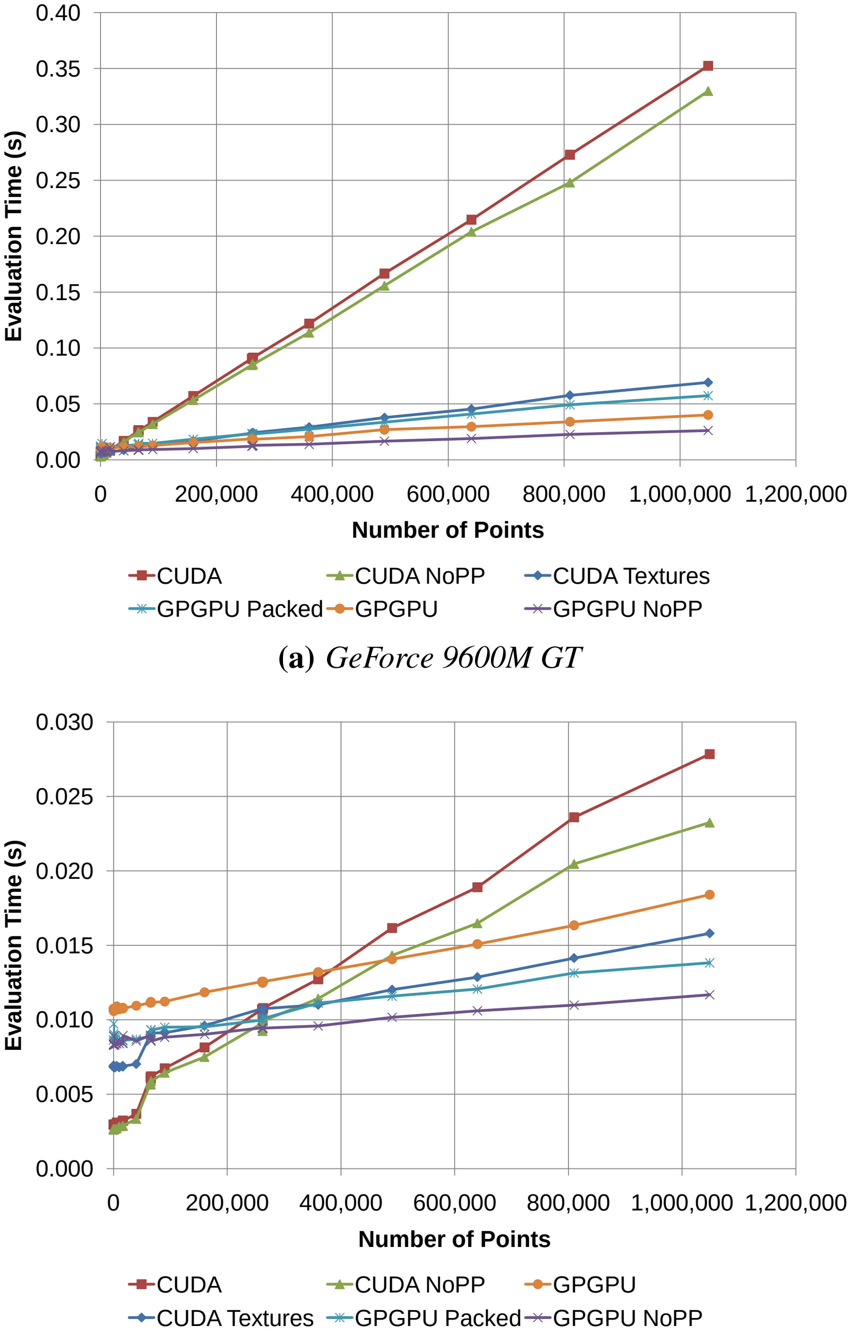
In the GeForce 9600M GT, the use of texture read-only memory improves CUDA performance when there are large numbers of evaluation points; the textured implementation being almost seven times faster at the largest evaluation grid size. Even though we do not need such large grids for display, we require them to meet the tolerances in our modeling operations, which will be discussed in more detail in Chapter (5). However, all the GPGPU variants are faster than CUDA for evaluating a large number of points, with the GPGPU implementation without ping-pong being the fastest.

On the other hand, the initialization times are higher for both GPGPU and the textured implementations. In the Quadro FX5800, the difference is much more pronounced between the different implementations for smaller and larger evaluation grid sizes. It can be seen that the initialization time is much lower for using CUDA global memory. Hence, as a result, the CUDA implementation is better suited for evaluating smaller number of points and the GPGPU version is better suited for evaluating large grids.

4.9 Summary and Conclusions

We have presented a new method to evaluate and display trimmed NURBS surfaces on the GPU. Our algorithm evaluates the NURBS surface point coordinates directly, without resorting to approximations, using a unified evaluation framework that uses the same fragment program to evaluate arbitrary degree NURBS surfaces. Our evaluation framework, which calculates all the basis function values in parallel, can be extended to calculate derivatives and normals, serving as a foundation for modeling operations as well (Chapter (5)). We show that packing the basis function arrays into the four color channels (along their height dimension to preserve the unified, degree-independent property of the implementation) yields a more efficient algorithm than unpacked or index-array based approaches for NURBS curve evaluations. The method shows great promise for real-time interaction with exact NURBS models, as seen from the frame rates we achieved even on older graphics cards. The evaluation timings show more than 40 times improvement over evaluation on the CPU for large inputs, and a similar improvement in overall frame rate compared to the OpenGL implementation. However, this method is still not optimal for a small number of evaluation points since the overhead of setting up the GPU for performing the computations is relatively high in this case. The number of surfaces that can be evaluated and displayed is primarily limited by texture memory on the GPU that is used to store the evaluated surface points and the trim data. We found our method to be capable of interactively evaluating and rendering up to 300 NURBS surfaces. For interactive display of a large number of trimmed NURBS surface patches, we have demonstrated that GPU-based evaluation of the exact surfaces is a viable option.

We have compared the NURBS evaluation time using both GPGPU and CUDA; both implementations were faster than the CPU. Even though the GPGPU implementation is faster than the CUDA implementation for high-resolution grids, it is slower than the CUDA implementation for smaller grids. This is due to the higher overhead in setting up the texture memory using OpenGL in the GPGPU implementation than using global memory or texture memory in CUDA. However, the CUDA implementation is slower than the GPGPU implementation at higher resolutions due to lack of caching of the global memory. Another main reason for the poor performance of the CUDA at the higher resolution is due to the presence of divergent branching while handling border elements that arise due to CUDA execution using blocks. This suggests that for optimal performance, we have to make use of a hybrid approach for NURBS evaluation. We have to use CUDA if the evaluation resolution is small and then switch to the GPGPU implementation for evaluating large resolution grids. In addition, the transition point at which the GPGPU performance gets better than CUDA changes based on the graphics card used. Hence, the algorithm has to be tuned for different kinds of graphics cards to get the optimum performance under different conditions.



(a) *GeForce 9600M GT*

(b) *Quadro FX5800*

Figure 4.21: *Comparison of evaluation timings of different CUDA and GPGPU algorithms.*

Chapter 5

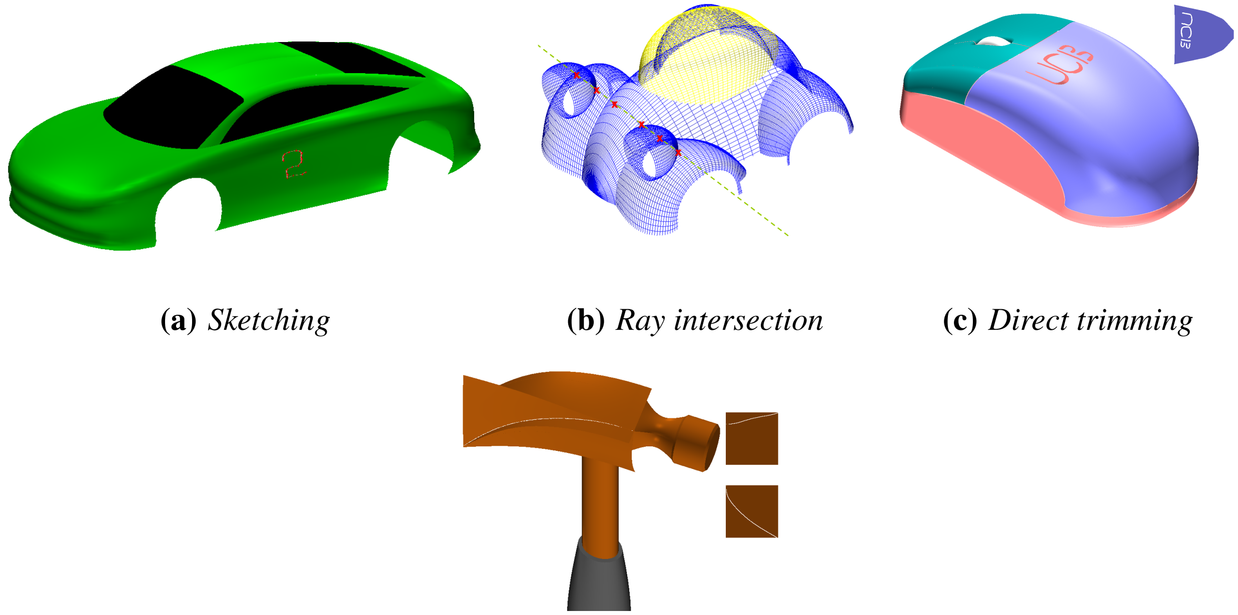
NURBS Modeling Operations

5.1 Introduction

In this chapter, we present algorithms for performing modeling operations such as inverse evaluations, ray intersections, and surface-surface intersections on NURBS surfaces. Our modeling algorithms run in real time, enabling the user to sketch on the actual surface to create new features. In addition, the designer can edit the surface by interactively trimming it without the need for re-tessellation. Our GPU-accelerated algorithm to perform surface-surface intersection operations with NURBS surfaces can output intersection curves in the model space as well as in the parametric spaces of both the intersecting surfaces at interactive rates. We also extend our surface-surface intersection algorithm to evaluate self-intersections in NURBS surfaces.

With the advent of programmable graphics hardware, the need for tessellating the NURBS surface in the CPU for display was obviated, since the GPU can be used for the evaluation and direct display of the surfaces [Krishnamurthy *et al.*, 2007; Kanai, 2007; Guthe *et al.*, 2005; Pabst *et al.*, 2006]. However, CAD packages still perform modeling operations using the CPU with either the tessellated surfaces or analytically using NURBS definitions. In addition, the tessellation is also performed only using the CPU. This reduces the interactivity for the user when designing these free-form surfaces, since operations such as sketching on the NURBS surface and fast intersection curves evaluation are not possible. Leading commercial CAD packages do not allow the designer to sketch directly on the NURBS surface; instead, they restrict the user to sketching on a tangent plane. Because of this, the designer has to wait until the operation is completed to get visual feedback.

The process of finding the surface coordinates (x,y,z) for given parameter values (u,v) is called evaluation. Inverse evaluation is the process of finding the parameter value (u,v) given any point on the surface. We have developed a parallel algorithm for fast inverse evaluations of NURBS surfaces on the GPU. This algorithm forms the basis of many modeling operations like selection (ray-surface intersection), sketching on the surface, and interactive trimming. Moreover, since these algorithms exploit the parallelism of the GPU, these operations can now be performed at



(a) *Sketching*

(b) *Ray intersection*

(c) *Direct trimming*

(d) *Surface intersection*

Figure 5.1: *Modeling operations like sketching, ray intersection, trimming and surface-surface intersection performed directly on trimmed NURBS models.*

interactive speeds, making immediate visual feedback to the designer possible for the first time. We demonstrate the use of our fast inverse evaluation algorithm to directly sketch on the surface, which makes certain operations like interactive trimming intuitive to the designer.

Designers are usually trained to work with curves on surfaces, such as silhouette curves and intersection curves. Thus, they would like to see real-time changes in these curves as the underlying surfaces are edited, which requires an efficient algorithm to compute intersection curves of free-form surfaces. Finding the intersection curve is in general a very complex operation, since two NURBS surface equations of arbitrary degree have to be solved simultaneously. Many commercial CAD packages use marching methods, where the algorithm uses a numerical root-finding technique to first find a single intersection point. The algorithm then finds another point along the intersection curve that is close to the first intersection point. This process is repeated and ultimately a complete piecewise linear approximation of the intersection curve is calculated. However, since this technique is inherently serial it cannot be parallelized for efficient evaluation on the GPU. We have developed a GPU-accelerated parallel algorithm to evaluate the intersection curves using bounds on the evaluated surface points. This algorithm is both fast and guaranteed to find the intersection curves within a user-defined tolerance.

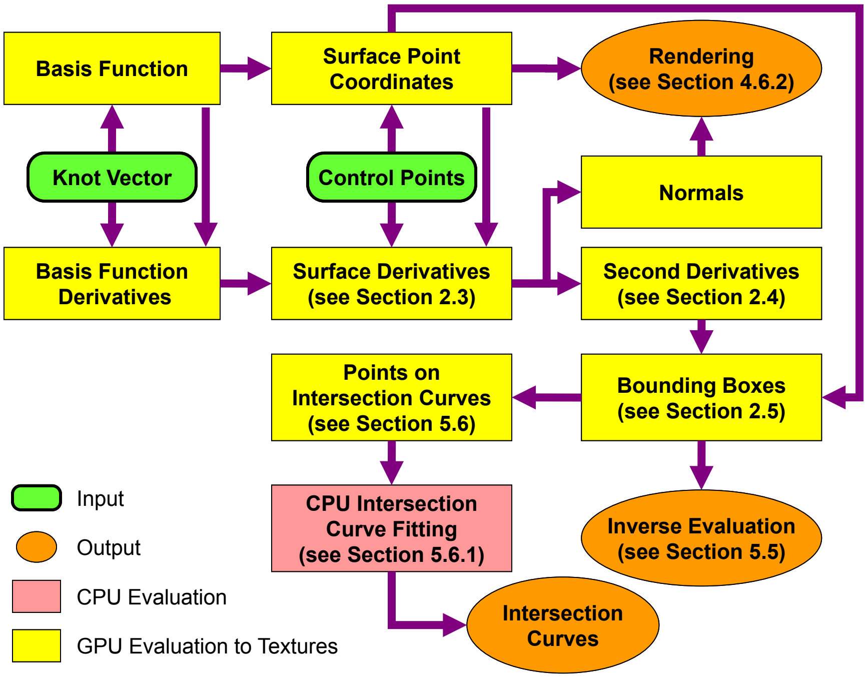

Basis Function
Surface Point Coordinates
Rendering (see Section 4.6.2)
Knot Vector
Control Points
Normals
Basis Function Derivatives
Surface Derivatives (see Section 2.3)
Second Derivatives (see Section 2.4)
Points on Intersection Curves (see Section 5.6)
Bounding Boxes (see Section 2.5)
Input
Output
CPU Evaluation
GPU Evaluation to Textures
CPU Intersection Curve Fitting (see Section 5.6.1)
Inverse Evaluation (see Section 5.5)
Intersection Curves

Figure 5.2: *Graphic showing the links between different parts of our modeling algorithms. The results of the GPU evaluations are stored in separate textures.*

We extend of our GPU NURBS evaluator (see Chapter (4)) to evaluate the first and second derivatives of the NURBS surfaces and then use these to compute bounding-boxes for NURBS surfaces (Section (2.5)). We describe how these bounding-boxes are used to perform inverse evaluations (Section (5.5)) and to compute intersection curves (Section (5.6)) in this chapter. Figure (5.2) shows the connections between the different parts of our algorithms; each of these operations are described in detail in the sections indicated.

5.2 Related Work

Previous work that used GPUs to render NURBS curves or surfaces focused only on efficient evaluation of the surface coordinates and/or normals [Guthe *et al.*, 2005; Loop and Blinn, 2005; Guthe *et al.*, 2006; Kanai, 2007]. They did not use GPUs to perform modeling operations like inverse evaluations and intersection curve evaluations. Previous work on inverse evaluation of NURBS surfaces mainly focused on ray tracing NURBS surfaces. Ray tracing was performed on parametric and rational surfaces by solving for the ray-surface intersection point using numerical methods [Toth, 1985; Nishita *et al.*, 1990; Martin *et al.*, 2000]. There has also been previous work on ray tracing using the GPU, which include [Purcell *et al.*, 2002; Purcell *et al.*, 2003; Carr *et al.*, 2006; Pabst *et al.*, 2006]. Another application of inverse evaluation of NURBS is solving for geometric constraints. A method to solve geometric constraints by using multivariate splines was given in [Elber and Kim, 2001], which can be used to solve several related problems like ray traps and sweep envelopes. Inverse evaluation has also been used for haptic rendering to find the parametric (u,v) coordinates of a given point on a NURBS surface [Thompson and Cohen, 1999]. Inverse evaluation was used in this case to solve for the contact point of a haptic probe with trimmed NURBS surfaces in a virtual environment.

Several approaches to collision detection on the GPU have been proposed. Occlusion queries on graphics hardware were used in [Govindaraju *et al.*, 2003] to detect collisions of polygonal meshes in large environments. Collisions between particles were calculated in [Kipfer *et al.*, 2004; Kolb *et al.*, 2004] to simulate large scale particle systems on the GPU. Recently, a method to detect collisions between deformable parameterized surfaces using GPUs was presented in [Greß *et al.*, 2006]. They solve the collision detection problem by generating a bounding-box hierarchy for the surface and then detect collisions by checking overlap between the bounding-boxes.

Evaluation of intersection curves is a fundamental operation in computer aided geometric design and solid modeling [Requicha and Rossignac, 1992; Hoffmann, 1989]. There have been several attempts to solve the problem, since it is hard to achieve all the desired characteristics of robustness, accuracy, and efficiency. A comprehensive survey of surface-surface intersection algorithms was summarized in [Patrikalakis, 1993]. A more recent algebraic algorithm for efficient surface intersection using lower dimensional formulations was given in [Krishnan and Manocha, 1997]. They also classified the conventional methods for evaluating the intersection curves as analytical methods, lattice evaluations, subdivision methods, and marching methods. Many commercial CAD software packages use the numerical marching method outlined in [Barnhill and Kersey, 1990; Kriezis *et al.*, 1990] to evaluate intersection curves.

5.3 Derivatives of NURBS Surfaces

To perform geometric operations on NURBS surfaces, we not only require the surface point coordinates themselves but also the first and second partial derivatives with respect to the two parameter directions u and v at the surface points. As a very fast first-degree approximation, we can use the evaluated point coordinates to estimate the first derivatives using central differencing. However, this approach gives rise to artificial discontinuities at patch boundaries and at rational parts of the surface. Moreover, second derivatives estimated from these first derivatives in the same manner have larger errors associated with them. One way to overcome this issue is to evaluate the normals of the surface exactly at each surface point, similar to the evaluation of the surface coordinates. Since we already evaluate the higher order basis functions from lower order basis-functions, we can directly calculate the derivatives of the basis-functions within the same framework as our basis-function evaluation algorithm, and then use the basis function derivatives to evaluate the derivatives of the NURBS surface precisely, to within machine precision. However, in our implementation, we estimate the second derivatives using central differencing since exact evaluation of second derivatives is very compute intensive and is not possible in an interactive setting. However, the errors in second derivative are very small since we accurately calculate the first derivatives and use a fine resolution (1024×1024) for the evaluation grid.

5.3.1 GPU Implementation

The GPU implementation of the evaluation of surface derivatives is a direct extension of the evaluation of the surface coordinates as explained in Chapter (4). The mathematical formulation for the evaluation of the derivatives was given in detail in Section (2.3.1). The GPU evaluation consists of four steps as given below. The first three steps are similar to the method for evaluation of the surface coordinates. We give the steps for evaluating the surface derivatives with respect to u; the steps for finding the derivative with respect to v are similar, exchanging u and v in step 2:

1. Locate the sub-mesh of control points that influence the evaluation point coordinates.
2. Compute the basis functions and their derivatives along the two-parameter directions respectively.
   (a) Compute the non-zero basis function derivatives with respect to u.
   (b) Compute the non-zero basis functions with respect to v.
3. Multiply the non-zero basis functions and the basis function derivatives with their corresponding control points from the sub-mesh and sum the results.
4. Evaluate the rational derivatives as given by Equation (2.14) using the evaluated surface coordinates and surface derivatives from the previous step.

One notable feature of this algorithm is that step 1 and step 2(b) are already performed while evaluating the surface coordinates using our NURBS evaluation algorithm. Moreover, computing the u derivative in step 2(a) is different from evaluating the B-spline basis function only in the final step of the evaluation. Since we are using the de Boor evaluation algorithm, evaluating the B-spline basis function of order k as well as its derivative requires the evaluation of the B-spline basis function of order $k-1$. In practice, since we are already computing the B-spline basis function of order $k-1$, we store this intermediate result as a texture on the GPU. We then use this as input for evaluating both the B-spline basis function of order k as well as its derivative.

We evaluate the derivatives of the basis functions with respect to each parameter direction separately and store them in separate textures on the GPU. Once the derivatives with respect to the u and v directions are calculated as 4-component vectors, the surface normals are calculated. This is performed using a separate fragment program that takes the rational surface derivatives as input and then evaluates their cross product to calculate the surface normal (Equation (2.11)). Thus, the process of evaluating the NURBS surfaces as well as their normals can be performed efficiently within a single framework using our method.

5.4 Bounding-Boxes for NURBS Surfaces

We make use of axis-aligned bounding-boxes (AABBs) for the NURBS surfaces to perform modeling operations using the GPU. With the help of such bounding-boxes, several queries such as ray-surface intersections and surface-surface intersections can be efficiently answered, which then form the building blocks for more complex operations like sketching on the surface and intersection curve calculations. We make use of the GPU to evaluate the derivatives and curvature of the NURBS surfaces. We then construct the bounding-boxes using the method explained in Section (2.5). Figure (5.3) shows pictorially the different GPU operations that are performed to construct the bounding boxes.

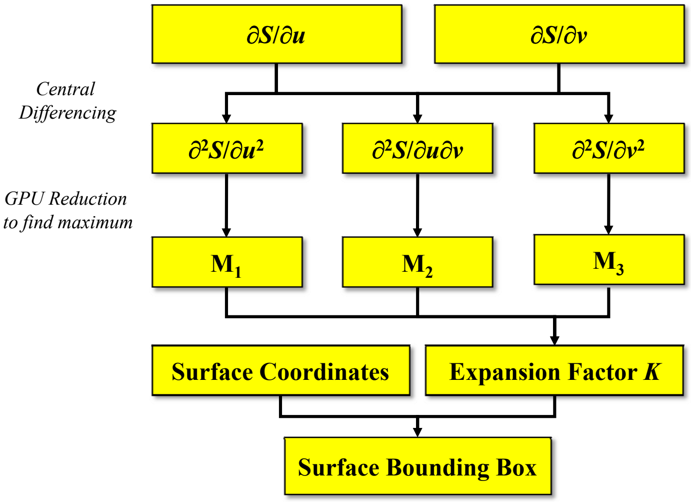


Figure 5.3: *Flow of algorithm to evaluate bounding-boxes of a NURBS surface on the GPU.*

5.5 Inverse Evaluation of NURBS Surfaces

Given a point that lies on the NURBS surface, inverse evaluation is the process of finding the parameter values corresponding to that point. Since the B-spline basis functions are non-linear, theoretical expressions for the inverse evaluation are very complex and differ based on the degree of the surfaces. Therefore, inverse evaluations are usually performed numerically to find a solution within a desired tolerance.

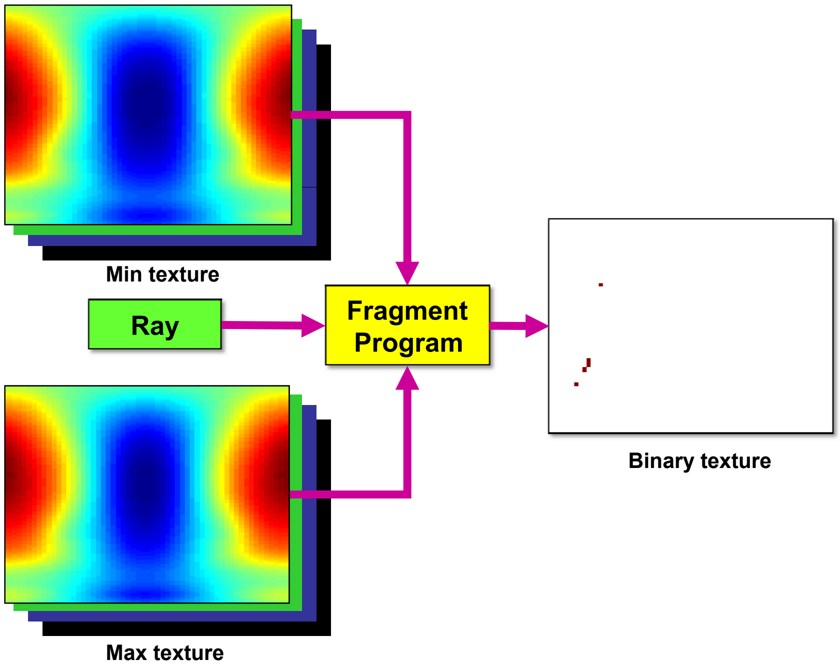


Figure 5.4: *Bounding-boxes stored as min and max textures are tested with the ray using a fragment program; its output is a binary texture indicating the intersection.*

The standard numerical approaches based on solving the NURBS equations for inverse evaluation are not easily parallelizable so as to be performed efficiently on the GPU. Therefore, we chose a method based on axis-aligned bounding-boxes. The AABBs for the NURBS surface are constructed using the method outlined in Section (5.4). In the case of selection and directly drawing on the surface, the AABBs are aligned parallel to the ray cast in the viewing direction, through the current location of the mouse. We then check for intersection between the ray and all the AABBs simultaneously using a fragment program written to perform this intersection test. The output of this program is a two-dimensional array of binary values with the value 1 indicating the intersection of the ray with the corresponding AABB (Figure (5.4)). In addition, the intersecting AABB also contains information about the minimum and maximum parameter values of the surface sub-patch enclosed by the AABB. Using this correspondence, we can efficiently find the parametric (u, v) value corresponding to the ray intersection point on the surface.

Since the NURBS surfaces are usually curved, there can be many surface sub-patches intersecting the given ray. We find the addresses of all the intersecting bounding-boxes (locations with the value 1 in the binary texture) by using the GPU stream reduction operation explained in Section (3.5.2). We use this address to access information about the intersecting bounding-box as well as the parametric ranges of the surface sub-patch enclosed by the bounding-box. Using the bounding-box information, we get bounds on the location of the intersection point of the ray with the surface in both the model space as well as in the parametric space simultaneously. If the bounding-boxes are smaller than the required tolerance, we can take the midpoint of the bounding-box as the intersection point of the ray with the surface. Once all the ray intersection points on the surface are found, we output only the point that is closest to the view-plane by evaluating the distance of all the ray intersection points from the view-plane on the CPU and choosing the point with the smallest distance value.

5.5.1 GPU Implementation of Inverse Evaluation

The algorithm used for performing the full inverse evaluation is given pictorially in Figure (5.5). The three steps in the top row of Figure (5.5)—evaluating the surface, constructing bounding-boxes, and finding intersecting boxes—are performed on the GPU. The data corresponding to the selected bounding-box is read back from the GPU. We then check on the CPU whether the ranges in the parametric domain of the surface as well as the size of the bounding-box are within the required tolerance; for example, we can use an absolute tolerance of 10^{-6} in the parametric space and a relative tolerance of 10^{-3} in the model space. If the tolerance conditions are met, we output the midpoint of the parametric range as the output of the inverse evaluation. If not, we re-evaluate the NURBS surface at a finer resolution within the previously output parametric range(s). These tolerances are usually met within two or three iterations since we evaluate the surface at a high resolution (1024 × 1024) during each iteration.

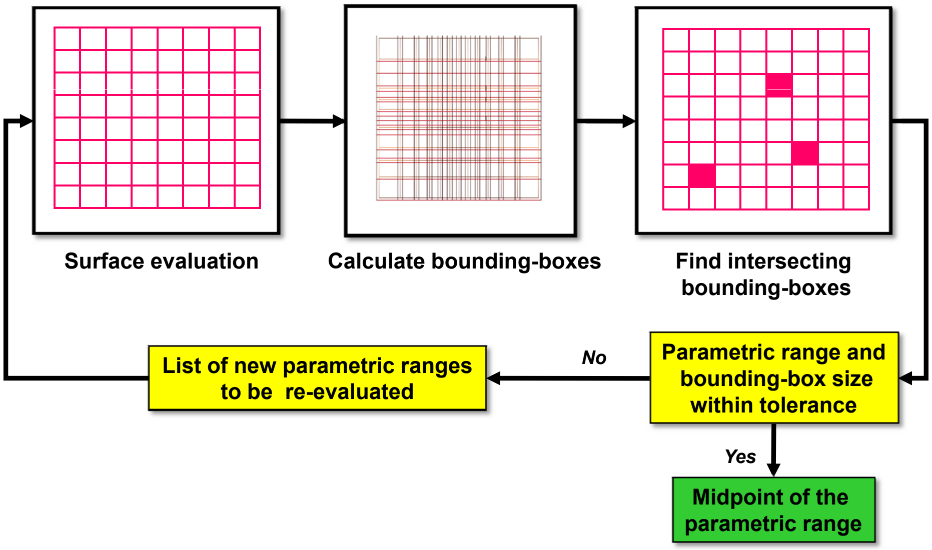
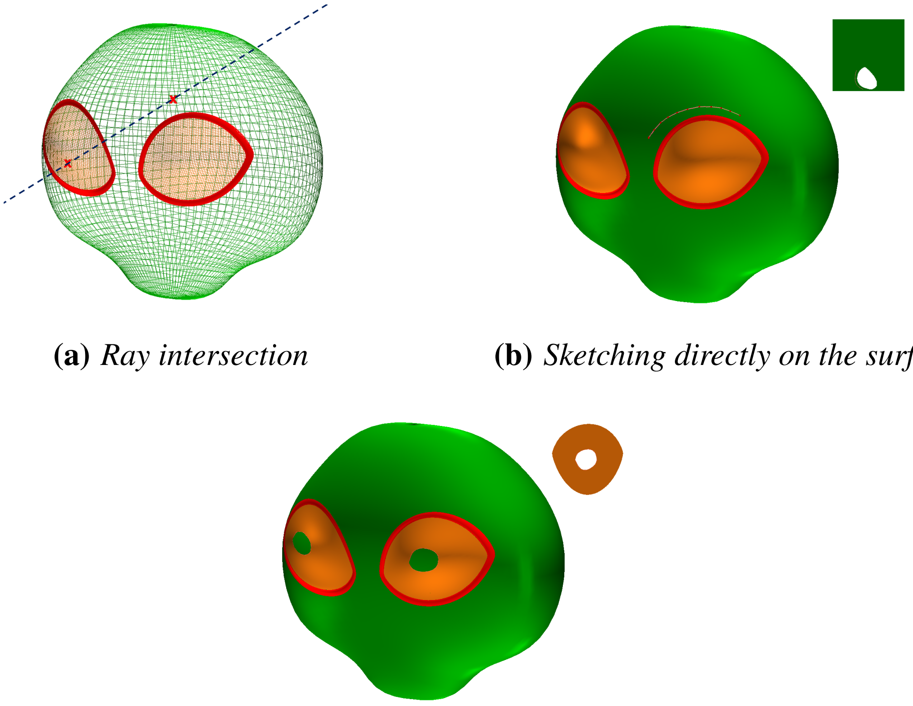

Surface evaluation
Calculate bounding-boxes
Find intersecting bounding-boxes
Parametric range and bounding-box size within tolerance
No
List of new parametric ranges to be re-evaluated
Yes
Midpoint of the parametric range

Figure 5.5: *Algorithm for inverse evaluation of NURBS surfaces.*

5.5.2 Applications of Inverse Evaluation

We can build different modeling operations using the inverse evaluation algorithm as the basic module. These operations include ray intersections, direct sketching on NURBS surfaces, and interactive trimming. Figure (5.6(a)) shows an example where we compute all the intersection points (two in this case, marked with red crosses) of a particular ray with the surfaces of a model of a toy. By aligning the ray direction perpendicular to the view plane, we can use the same algorithm for selecting a particular surface from a given set of NURBS surfaces.

One of the most important advantages of a real-time algorithm to perform inverse evaluation is the ability to sketch directly on the NURBS surface. The advantage comes from the fact that the curve is simultaneously sketched both in the 3-dimensional model space as well as in the 2-dimensional parameter space. This helps in performing modeling operations like extrusions and trimming, where the parameter space sketches are typically used for defining these operations. Figure (5.6(b)) shows a curve sketched on a NURBS model and the curve in the parametric domain is shown in the inset.



(a) *Ray intersection*

(b) *Sketching directly on the surface*

(c) *Interactive trimming: the eyes of the model were trimmed interactively*

Figure 5.6: *Different NURBS modeling applications using inverse evaluation.*

By combining our sketching interface with the algorithm that renders trimmed NURBS surfaces in real-time, we can perform interactive trimming operations (Figure (5.6(c))). Using our interactive trimming application, the designer gets immediate feedback on the result of the trimming operation, unlike current commercial CAD systems.

5.6 NURBS Intersection Curve Evaluation

Calculating the intersection curve of a surface-surface intersection is a frequently encountered operation in CAD systems. It forms an essential part of important CAD operations like trimming, filleting, and b-rep generation from Boolean operations. However, since it is a slow operation, it is usually performed in the background and thus the user does not get real-time feedback except in the simplest of cases. We present a GPU-accelerated surface-surface intersection algorithm to calculate intersection curves both in the model space as well as in the parametric spaces of both the surfaces.

We now give a broad overview of our surface-surface intersection algorithm. Our algorithm makes use of bounding-box hierarchies to accelerate the intersection operation. We evaluate both intersecting surfaces using the GPU and then use the method described in Section (5.4) to construct the AABBs for the surfaces, using the same coordinate frame. We construct a hierarchy of bounding-boxes by combining four bounding-boxes at a finer level to construct a single bounding-box in the next coarser level. To find the intersection curve, we traverse along the hierarchy for both the surfaces simultaneously and find the intersecting bounding-boxes in the finest level using the GPU. At the same time, we also get the ranges in the parametric domain corresponding to the intersecting surface patches. We then check if the size of the bounding-boxes as well the parametric ranges are within a user-defined tolerance. Once the tolerance conditions are met, we get a better estimate of the point on the intersection curve by intersecting the linearized surface patch within the intersecting bounding-boxes.

We will explain the details of our surface-surface intersection algorithm with an example (Figure (5.7)). Given two surfaces, S_1 and S_2, we evaluate them and construct their bounding-boxes as explained in previous sections. We also construct the bounding-box hierarchies for both the surfaces and store them on the GPU as textures. Once we have the hierarchies, we check whether the top-level (level 1) bounding-box of S_1 intersects with the top-level bounding-box of S_2. We perform this test on the CPU since it is a very simple test. If the bounding-boxes intersect, we then test the bounding-boxes from the next level (level 2) onwards on the GPU, using one pass per level. We perform the intersection tests for all the bounding-boxes in a level in parallel using a fragment program written to perform the bounding-box intersection test. The input to the fragment program is a texture called the address texture that contains the address of the bounding-boxes in the hierarchy (also stored as textures). For example, to test for intersection in the second level, we make use of a 4×4 address texture on the GPU, where we test for intersection of four bounding-boxes of S_1 with all the four bounding-boxes of S_2. In Figure (5.7), the rows of the address texture

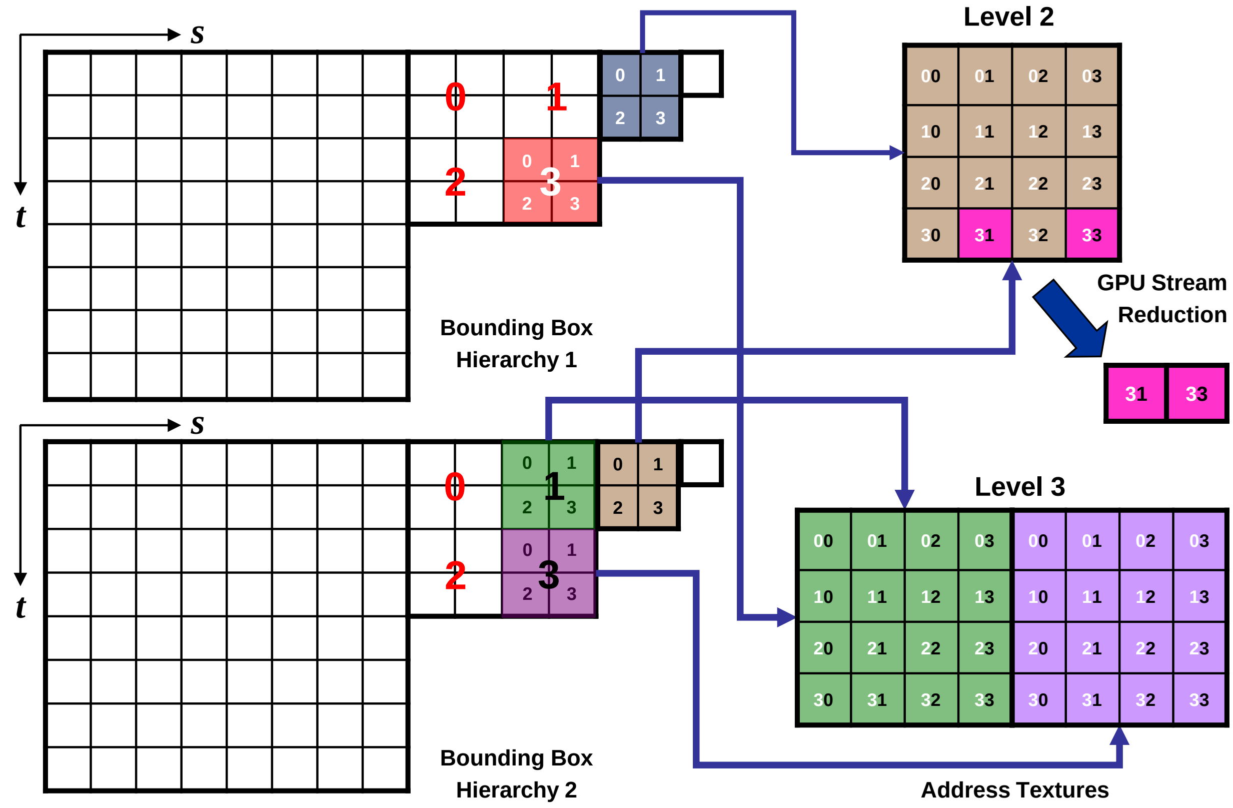


Figure 5.7: *Example hierarchical bounding-box comparison in the surface-surface intersection algorithm.*

(Level 2) corresponds to bounding-boxes from S_1 and the columns correspond to bounding-boxes from S_2. The address texture is a 4-component texture consisting of the address corresponding to bounding-boxes of S_1 and S_2 in the bounding-box hierarchy textures ((s_1,t_1,s_2,t_2) stored using RGBA channels). The intersection test is performed on the GPU using a fragment program, which uses the address information to retrieve the data for the bounding-boxes from the bounding-box hierarchy and subsequently tests them for intersection. The output of the fragment program is a binary texture with a value of 1 indicating an intersection. We use the stream reduction algorithm explained in Section (3.5.2) to find the address of the intersecting bounding-boxes. In the example shown, we find that bounding-box 3 of S_1 intersects with bounding-boxes 1 and 3 of S_2 at level 2.

In the next level (pass), we test for the intersection of the children of the intersecting bounding-box pairs of the previous coarser level simultaneously on the GPU. Thus, the size of the address texture varies dynamically based on the number of intersections in the previous levels. The size of the address texture is always a multiple of four since we test for intersection between S_1 and S_2 in blocks of 4 × 4 intersection tests. However, to perform the stream reduction operation we zero-

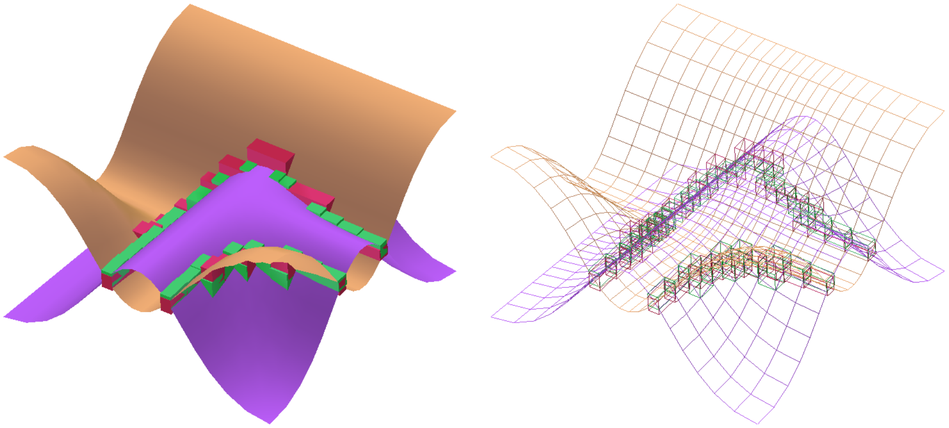


Figure 5.8: *Intersecting bounding-boxes of two NURBS surfaces.*

pad this rectangular texture to make it a square texture with a power-of-2 size. The parallelism of the GPU is exploited in checking for intersection of all the intersecting bounding-box pairs at any given level and this helps in accelerating the intersection algorithm as the address texture grows in size. Once we reach the finest level of the bounding-box hierarchy, we get a list of the bounding-boxes that intersect at this level (Figure (5.8)). This list can then be used for further processing on the CPU to get the actual intersection curve.

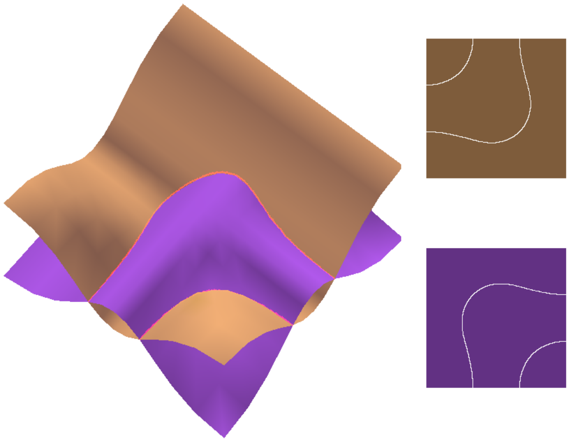


Figure 5.9: *Intersection curves of two NURBS surfaces plotted both in the model space as well as in their corresponding parametric spaces.*

In addition, we use this list to render the points on the intersection curve of each surface to a dynamic texture in the parametric domain. We map this texture back onto each surface, providing real-time feedback to the designer about the shape of the intersection curve (Figure (5.9)).

5.6.1 Fitting an Intersection Curve

To get a better estimate of the intersection point lying on the intersection curve of two surfaces, we intersect the surface sub-patches enclosed by the intersecting bounding-boxes on the CPU. We approximate each surface sub-patch inside the bounding-box with two triangles that share an edge. We intersect these two triangles contained inside the bounding-box of the first surface with the two other triangles contained in the bounding-box of the second surface. This gives rise to four pairs of intersection tests between the triangles of the two surfaces; each intersection test can be true or false, generating 16 different cases. We show the most common case in Figure (5.10), where one triangle of surface $S1$ intersects with another triangle of surface $S2$. The four triangles are denoted as $A_0A_1A_2$ and $A_1A_2A_3$ for surface S_1, and $B_0B_1B_2$ and $B_1B_2B_3$ for surface S_2 in the figure. We find the midpoint of the intersection line-segment and use this midpoint as a point on the intersection curve if it lies within the intersecting region of the bounding-boxes. The intersecting region of the bounding-boxes is denoted by $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ in the figure. In the case of multiple intersections, we take the centroid of the midpoints of the intersection line-segments computed for each intersecting triangle pair as a point on the intersection curve.

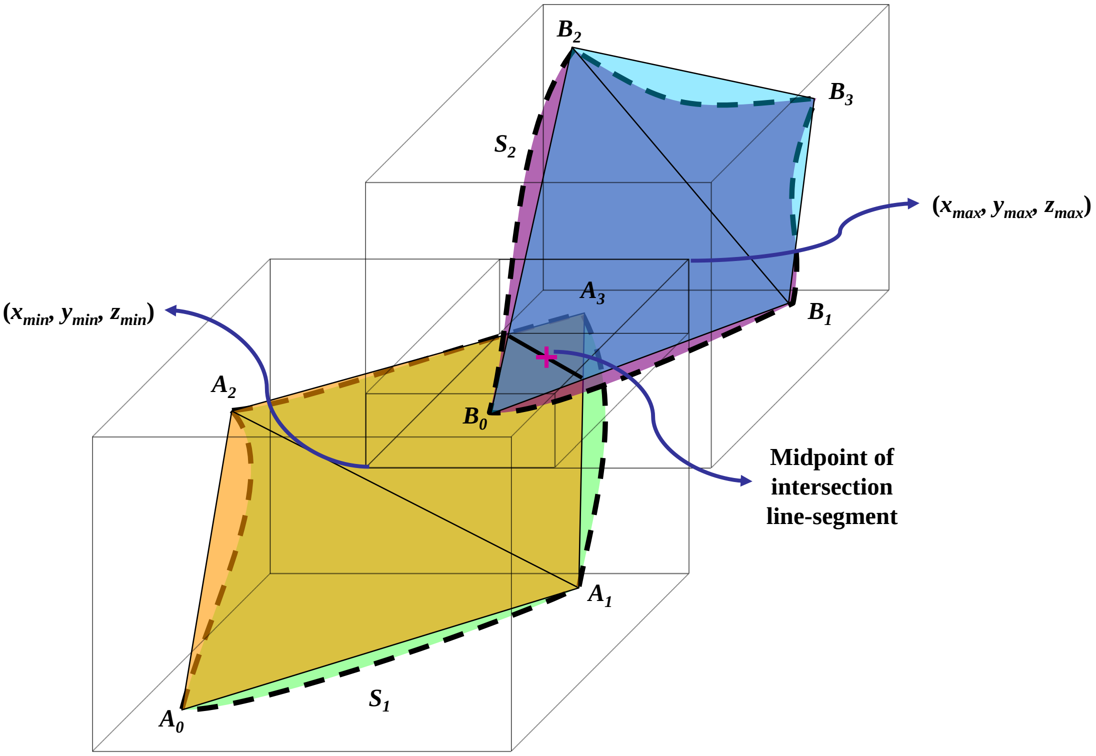


Figure 5.10: *Intersecting triangles inside overlapping bounding-box pairs to get a better estimate of the point on the intersection curve.*

We work in the 7-dimensional space $\Re^7$ for curve fitting, integrating the data from both the model space as well as the two parametric spaces. Performing the curve fitting in $\Re^7$ is more robust since different components of intersection curves that might be close in a particular geometric or parametric space are less likely to be simultaneously close in all three spaces. We extract the 7-tuple (x,y,z,u_1,v_1,u_2,v_2) for each point found on the intersection curve, where (x,y,z) is the point on the intersection curve in 3D space, (u_1,v_1) and (u_2,v_2) are the corresponding points in the parametric space of surface S_1 and surface S_2 respectively. The parametric points are found by computing the barycentric coordinate of the (x,y,z) intersection point in each of the corresponding intersecting triangles and then interpolating the parametric coordinates at the three vertices of the triangle linearly using the barycentric coordinates.

Finally, to compute the actual intersection curves themselves from the list of points, we compared two different algorithms. The first one is a greedy algorithm (Algorithm 5.1) that computes the intersection curves by successively merging polylines that are close to each other.

```
Input  : List of points on the intersection curves in $\Re^7$.
Output : Polyline list $L$, corresponding to the intersection curves
         (an ordered list of connected edges).

1. Make all points into a polyline of length 0; add to $L$.
2. For all polylines in $L$, find the pair, $P_1$ and $P_2$, that is the closest (between two end
points of $P_1/P_2$ in $\Re^7$).
3. If distance greater than maximal distance to merge
    Quit;
    Otherwise,
        (a) Merge $P_1$ and $P_2$ into a new polyline $P$.
        (b) Replace $P_1$ and $P_2$ by $P$ in $L$.
        (c) Goto 2
```

Algorithm 5.1: *Algorithm to fit polylines to the points on the intersection curves.*

The second algorithm (Algorithm 5.2) uses the fact that the intersection points we find are enclosed by AABBs that are part of a regular grid. We can thus fit a polyline by connecting a point to the closest point whose enclosing bounding-box is a neighbor to the enclosing bounding-box of the current point, limiting our search to the 1-ring neighborhood of bounding-boxes. After adding the closest point in the 1-ring neighborhood to the polyline, we repeat our search to find another point that is the closest to the point just added. Since the starting point can be in the middle of a intersection curve, we have to grow the polyline in both directions. This algorithm can be compared to a depth-first search on a list to find all the connected components and hence takes $O(n)$ time. However, if there is more than one remaining adjacent bounding-box with unmerged intersection points, some points may not merged into the polyline and will be output as polylines of length 1 (Figure (5.11)). These polylines can then either be discarded or merged at the correct position of the longer polylines by making an additional pass.

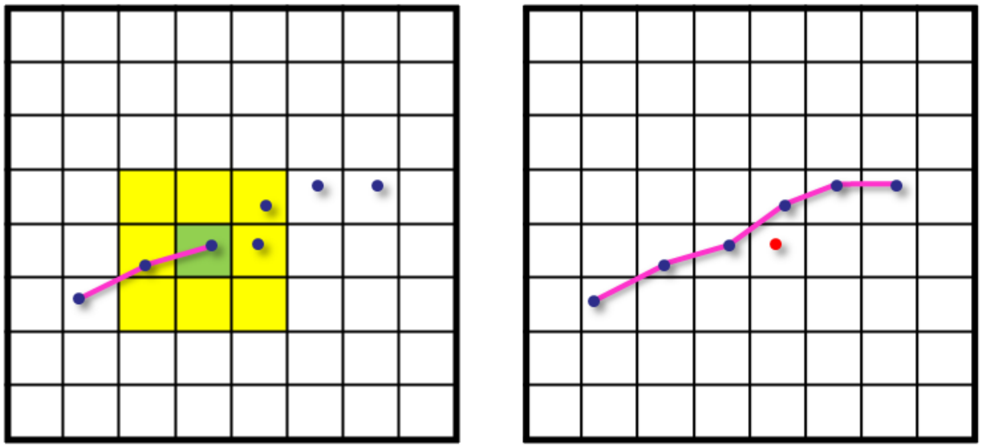


Figure 5.11: *Example showing the possible generation of single-point polyline by Algorithm 5.2. The figure on the left shows the 1-ring of bounding-boxes (shaded). The point marked in red on the right is not merged and is output as a single-point polyline if it is not the closest point in* $\Re^7$.

The time taken to fit a polyline using Algorithm (5.1) depends on an efficient closest neighbor query. Currently, we perform this operation through an exhaustive search that takes $O(n^2)$ time, which could be optimized by using more efficient search techniques, but we would still expect it to be slower than the $O(n)$ time Algorithm (5.2). For the example shown in Figure (5.9), the polyline fitting for over 7000 points takes 0.20 seconds on a 2GHz PC for a tolerance value of 2×10^{-3}. On the other hand, the time taken by a single pass of Algorithm (5.2) was 0.02 seconds for the same input and tolerance value. However, 320 single point polylines were also produced by Algorithm (5.2) which were discarded. From a tolerant geometry point of view, discarding these points does not reduce the overall tolerance achieved compared to Algorithm (5.1). Since the intersection points obtained from our algorithm are enclosed within their corresponding bounding boxes both in the model space and in the parametric space, we can guarantee a required bound on the results.

A polyline that passes through our input list of points on the intersection curve can be directly used for further modeling operations, since these points are sufficiently dense. If a more compact representation is required, we can fit a NURBS curve of any required order that approximates the points on the intersection curve using standard curve fitting techniques. We can guarantee an arbitrary bound in both the model space and parametric spaces. In addition, if the user-defined bounds are small enough, we are guaranteed not to miss any portion of the intersection curve. Since we also give instantaneous visual feedback to the user, the user will immediately know if there are any features missing and can reduce the tolerance to obtain the desired result.

```
Input   : List of points on the intersection curves in $\Re^7$.
Output  : Polyline list $L$, corresponding to the intersection curves
          (an ordered list of connected edges).

1. Add all the points to unmerged points list $M$.
2. Add $S$, the first point in $M$ to a new polyline $P$.
3. $A \longleftarrow S$
4. While there are unmerged points in the 1-ring of $A$ (the point added last to $P$)
   (a) Find the closest point $B$ in the 1-ring of $A$.
   (b) Add $B$ to tail of $P$ and remove it from $M$.
   (c) $A \longleftarrow B$
5. $A \longleftarrow S$
6. While there are unmerged points in the 1-ring of $A$
   (a) Find the closest point $B$ in the 1-ring of $A$.
   (b) Add $B$ to head of $P$ and remove it from $M$.
   (c) $A \longleftarrow B$
7. If $M$ is empty
      Quit;
   Otherwise,
      (a) Add $P$ to $L$
      (b) Goto 2.
```

Algorithm 5.2: *Faster algorithm to fit polylines to the points on the intersection curves.*

One of the main limitations of both our algorithms for fitting a polyline is that they will fail to recreate the correct topology when two unrelated intersection curves are very close on both surfaces. This can happen when an intersection curve splits into two branches or when the two surfaces are locally flat and are touching each other. A method that ensures the topology of the intersection set is to be sought as future work, possibly at the CPU level, using the GPU only to find the simple intersection curves. Such a method will also help in balancing the load between the CPU and the GPU.

5.6.2 Self Intersection Evaluation

We extended our surface-surface intersection algorithm to detect and evaluate self-intersections in NURBS surfaces. To perform the self-intersection test, we create two instances of the bounding-box hierarchy for the surface on the GPU. We then test for intersection between these two surface instances using the same GPU algorithm we use to perform surface-surface intersections. The output of this algorithm is a list of bounding-box pairs at the lowest level of the hierarchy that overlap each other. We then remove from this list all the pairs which correspond to the same surface sub-patch. Finally if there are any bounding-box pairs which belong to different surface sub-patches left in the list, then the surface is self-intersecting.

Once we find a surface to be self-intersecting, we perform triangle-triangle intersection of the triangles contained within the intersecting bounding-box pairs. Similar to the surface-surface intersection algorithm, we find points on the self-intersection curve and then fit a polyline through this self intersection curve. However, the main limitation of the algorithm is that a self intersection smaller than the tolerance will be rejected. This can occur as a local self-intersection due to high curvature in an offset surface. In this case detecting the intersection curve will be difficult since the tolerance needs to be infinitesimally small. Figure (5.12) shows two examples where we detect and evaluate self-intersection curves in NURBS surfaces. The example shown on the right took 0.42 seconds to compute the self-intersection curves to a tolerance value of 2×10^{-3}, while the more complicated example shown on the left took 0.97 seconds.

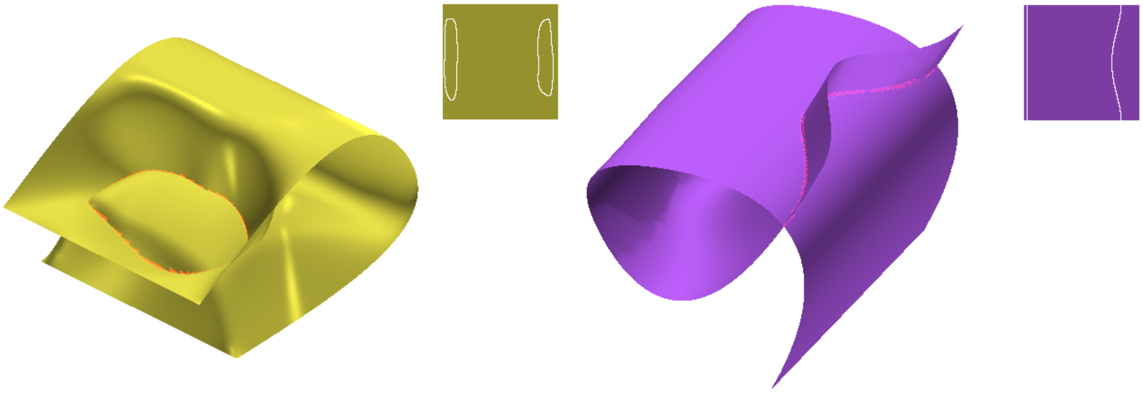


Figure 5.12: *Detection and evaluation of self-intersections in NURBS surfaces.*

5.6.3 Intersection Timing

We timed our GPU-accelerated algorithm for evaluating the intersection curves on a 3GHz CPU with 2GB of RAM equipped with a NVIDIA Quadro FX4500 GPU with 512MB graphics memory running Windows XP. We performed a surface-surface intersection of the two NURBS surfaces shown in Figure (5.13). The surfaces were bi-cubic NURBS with 403×199 and 298×313 control points respectively. We used Algorithm (5.2) to fit the polylines during the timing. We compare our timings to evaluate the intersection curves to the required user-defined tolerance with those of the commercial solid modeling kernel ACIS (v20).

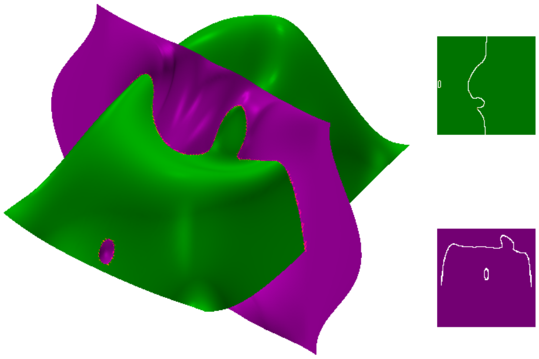


Figure 5.13: *NURBS surfaces used for timing the evaluation of intersection curves.*

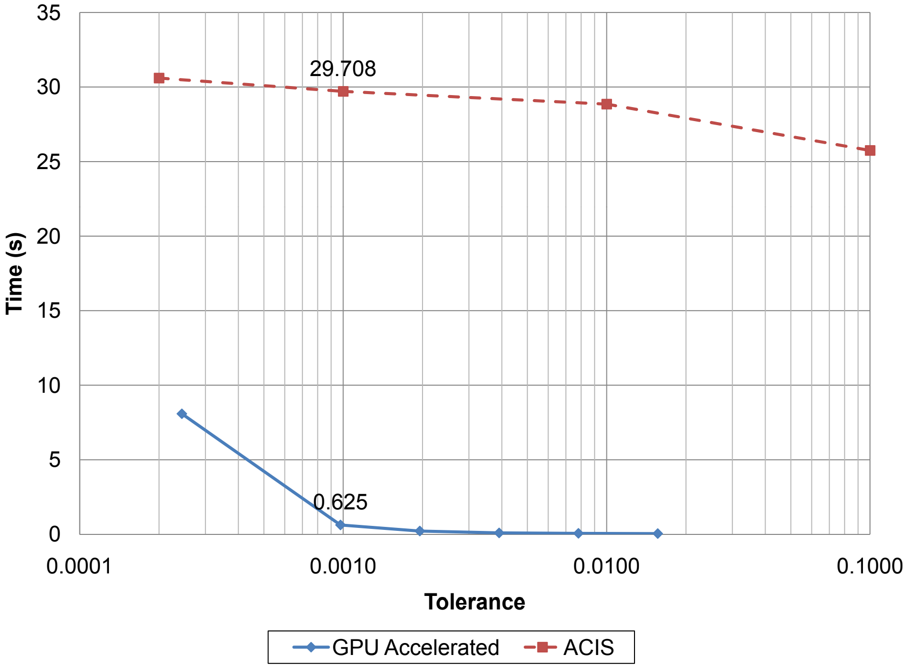


Figure 5.14: *Time taken for evaluating the intersection curves of the two NURBS surfaces shown in Figure (5.13) with different resolutions. Note that we are evaluating many more points on the intersection curve for a given resolution (Figure (5.15)).*

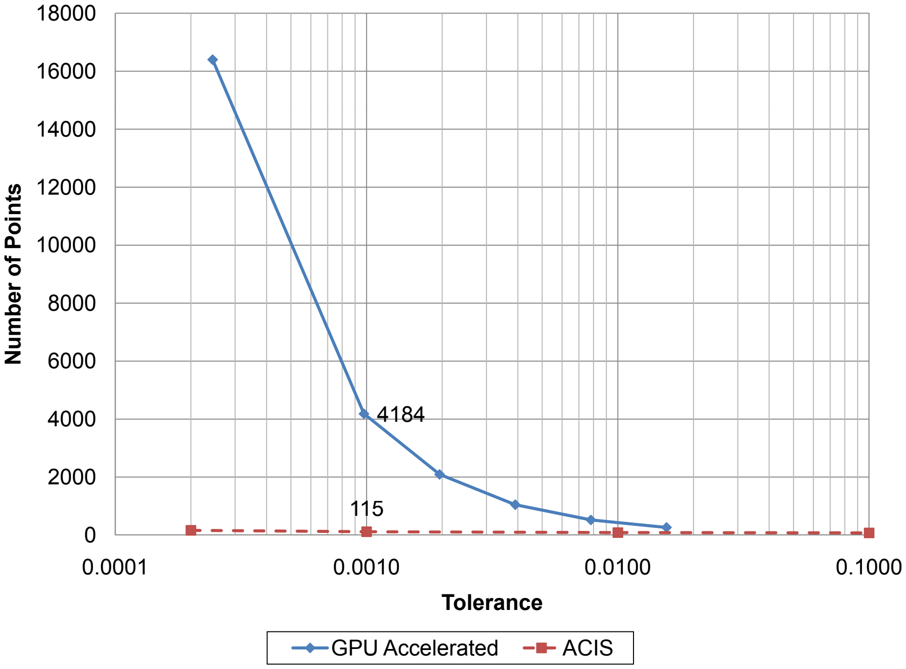


Figure 5.15: *Number of points evaluated on the intersection curve for different resolutions.*

Figure (5.14) compares the time for evaluating the intersection curves by varying the tolerance values. Our GPU-accelerated evaluation is more than 40 times faster than ACIS in computing the intersection curves to the standard tolerance of 10^{-3} used in ACIS. The output from ACIS is an interpolated polyline where the points on the polyline are within the user-defined tolerance value from the exact intersection curve. ACIS does not guarantee any tolerance on the piecewise linear line segments that make up the polyline [Corney and Lim, 2001]. On the other hand, we evaluate dense intersection points with their spacing adjusted based on the tolerance to achieve a guaranteed tolerance on the piecewise linear segments of the polyline as well. We compute almost 40 times as many points on the intersection curve as ACIS does for the standard ACIS tolerance value of 10^{-3} (Figure (5.15)).

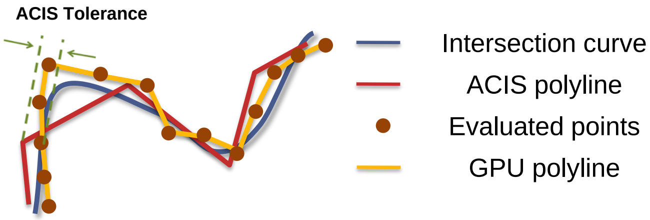


Figure 5.16: *Example of the tolerance definition used by ACIS. ACIS does not guarantee the tolerances for the polyline segments that connect the evaluated points on the intersection curve.*

Operation	**Time(s)**
Evaluate NURBS surfaces	0.27
Perform intersection tests	0.05
Calculate dense intersection points	0.02
Fitting the polyline	0.14
Total	0.48

Table 5.1: *Breakdown of the timing to perform different operations of our intersection algorithm. The values are for the example shown in Figure (5.9) for a tolerance value of* 10^{-3}.

Table (5.1) gives the breakdown of the timing of our intersection algorithm for evaluating the intersection curves shown in Figure (5.9) for a tolerance value of 10^{-3}. The evaluation of the NURBS surfaces is a large fraction of the total time. Note that we do not require such high tolerance values for giving visual feedback; hence, it can be performed at interactive rates.

5.7 Conclusions

The modeling algorithms we have developed do not require the latest graphics cards and are backward compatible with any graphics card that has basic programming capabilities. This is essential for the actual adoption of our algorithms in commercial CAD systems. We expect the performance of our algorithms to only improve with the advent of new and faster graphics cards.

Both our GPU algorithm to sketch on NURBS surfaces as well as our GPU-accelerated algorithm to calculate intersection curves give real-time feedback to the designer about the shape of the curves in the parametric space. This gives a direct handle for the designer to check for inconsistency if models fail during rebuilds in a CAD system. Our interactive trimming tool helps the designer to easily interact with and edit the NURBS models.

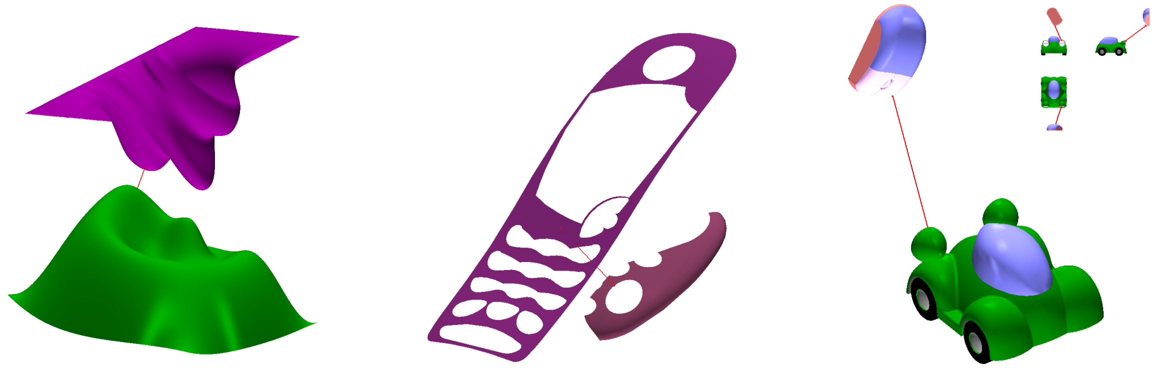
Chapter 6

Separation Distance Queries

6.1 Introduction

In this chapter we present GPU algorithms for accelerating distance queries on models made of trimmed NURBS surfaces. By supplementing surface data with a surface bounding-box hierarchy on the GPU, we answer distance queries such as finding the closest point on a curved NURBS surface given any point in space and evaluating the clearance between two solid models constructed using multiple NURBS surfaces. We simultaneously output the parameter values corresponding to the solution of these queries along with the model space values. Though our algorithms make use of the programmable fragment processor, the accuracy is based on the model space precision, unlike earlier graphics algorithms that were based only on image space precision. In addition, we provide theoretical bounds for both the computed minimum distance values as well as the location of the closest point. Our algorithms are at least an order of magnitude faster and about two orders of magnitude more accurate than the commercial solid modeling kernel ACIS.

Distance queries such as finding the minimum distance to a surface play an important role in many computer aided design and analysis applications, including tolerancing, clearance analysis, and accessibility analysis. Minimum distance queries are especially useful while designing complex assemblies to allow for sufficient clearance between different mechanical components. Such queries are easily answered if the objects or models are made of planar faces and have boxy shapes. However, modern designs make use of curved freeform NURBS surfaces. Minimum distance queries on such freeform surfaces are currently being solved by commercial solid modeling software by first evaluating and tessellating the surface and then finding the minimum distance to the tessellation vertices [Spatial Corporation, 2009a]. This approach, in addition to being extremely slow and computationally intensive, is dependent on the tessellation resolution for the accuracy of the solution; the surface has to be very finely tessellated to get the required accuracy.



(a) *NURBS Clearance* **(b)** *Trimmed NURBS Clearance* **(c)** *Object Clearance*

Figure 6.1: *Minimum distance/closest point computations between NURBS surfaces and complex CAD models accelerated using the GPU.*

A technique to accelerate such slow geometric queries is to use programmable GPUs. Previous GPU-based algorithms that render to the screen to perform these computations have restricted accuracy corresponding to the dimensions of the pixel or window. Our framework allows for the GPU algorithms to operate in the model space; therefore, the results of these geometric queries are accurate to any arbitrary user-defined tolerance.

Solid modeling kernels support certain distance queries such as the minimum distance from a point to a surface and the minimum distance between two surfaces. Applications of such distance queries include: finding the closest surface point on a surface to provide haptic feedback; dimensioning and tolerancing of CAD models; and constructing distance fields. In this chapter, we present an algorithm that uses our hybrid CPU/GPU framework consisting of surface bounding-boxes to accelerate these queries. We focus on performing distance queries on objects made of trimmed NURBS surfaces. However, our algorithms are applicable for any surface that can be supplemented with a surface bounding-box structure. We provide theoretical bounds on the accuracy of both the computed minimum distance as well as the location of the closest point on the surface, which allow for arbitrary user-defined tolerance values. This is especially important in CAD systems since these distances might be used by the designer to define subsequent features; the model might fail to regenerate if there is an error in the computed distance.

To perform geometric computations on NURBS surfaces or assemblies, we make use of a surface bounding-box structure to map the computations to the GPU. We make use of Axis-Aligned Bounding-Boxes (AABBs) constructed from an evaluated mesh of points on the NURBS surface to accelerate the computations (Section (2.5)). The main advantage of AABBs over Oriented Bounding-Boxes (OBBs) is that several geometric computations such as finding intersections and distances are simpler in the case of AABBs. This is especially important because the efficiency of GPU programs can be reduced dramatically with increases in the complexity of the parallel kernels that are used. The individual computational kernels for OBBs are more complex and contain many branching conditions; the GPU has to wait until the most computationally intensive branch of the kernel in a particular pass is completed before proceeding to the next pass. In addition, since OBB kernels make use of more temporary registers, the number of computations that can be active simultaneously on the GPU (called fragments in flight) is reduced; it is difficult to hide the memory access latency in this case. Thus, we found that the advantage provided by tight OBBs is offset by the increase in complexity of the algorithms that use them. We achieve better results by using AABBs even if we must decompose the model to a finer resolution with AABBs than OBBs in order to maintain the same tolerance bounds.

6.2 Related Work

Minimum distance computations are used by many algorithms that generate geometrical constructs such as Voronoi diagrams and medial axis transforms. They are also used in path planning and robot motion planning [Gilbert *et al.*, 1988] and for projecting points onto a patch of a CAD model [Henshaw, 2002]. Minimum distance computations on curved NURBS surface are very time-consuming; hence, the commercial solid modeling system ACIS makes use of the tessellation of the surface to find the closest vertex or pair of vertices while performing tolerance analysis [Spatial Corporation, 2009a]. Johnson et al. [Johnson and Cohen, 1998] gave a unified framework for minimum distance computations, which was later extended to find the closest point for haptics applications by Nelson et al. [Nelson *et al.*, 2005]. We use a similar method that uses AABBs to find the regions of the model that are likely to contain the closest points. However, the methods they describe were better suited for a serial CPU implementation, since they make use of the convex hull of the freeform surface to iteratively refine the search. In our algorithm, the distance computations and search operations are done in parallel, which is better suited for a GPU implementation. In addition, we also provide theoretical guarantees for the solutions we compute.

Edelsbrunner [Edelsbrunner, 1985] proved that the minimum distance between two convex polygons can theoretically be computed in $O(\log n)$. However, the algorithm used in the proof is theoretical and has large time-constants in practice. Quinlan [Quinlan, 1994] extended the minimum distance computations to non-convex objects by first performing a convex decomposition and then using bounding spheres for the convex pieces to create a hierarchy. However, this method is not practical for dynamic geometries since the convex decomposition might be expensive. Chen et al. [Chen *et al.*, 2008] compute the minimum distance between a point and a NURBS curve by subdividing the curve into portions that might contain the closest point. Many minimum distance algorithms use Bounding Volume Hierarchies (BVHs) to accelerate the computations. CPU algorithms usually make use of BVHs that are more complex than AABBs. Gottschalk et al. [Gottschalk *et al.*, 1996] make use of OBBs to perform distance computations. Larsen et al. [Larsen *et al.*, 2000] perform proximity queries using a construct called a sphere swept volume, which consists of a sphere swept over a point, line or a plane, as primitives of a BVH.

Collision detection and distance field computation are two problems that are closely related to minimum distance computations. There have been methods to effectively accelerate both using the GPU. Occlusion queries on graphics hardware were used by Govindaraju et al. [Govindaraju *et al.*, 2003] to detect collisions of polygonal meshes in large environments. Greß et al. [Greß *et al.*, 2006] solve the collision detection problem by generating a bounding-box hierarchy for deformable parameterized surfaces and then detect collisions by checking overlap between the bounding-boxes using the GPU. Sud et al. [Sud *et al.*, 2004] use the GPU to generate 3D distance fields by first slicing the model into 2D slices and by using culling and spatial coherence to reduce the number of distance computations in each slice. Lauterbach et al. [Lauterbach *et al.*, 2009] use the GPU to construct BVHs that can then be used to accelerate collision detection.

There has been only limited use of GPUs to perform geometric operations because they are restricted to image-space resolution if the computations are to be performed by rendering on the screen. Agarwal et al. [Agarwal *et al.*, 2003] make use of the GPU to perform geometric computations on a stream of points by using point-line duality. They compute geometric properties such as diameter and width of a set of points. However, these algorithms are not stable for points that are very close and are limited to image-space resolution. Hoff et al. [Hoff *et al.*, 2001] use the GPU to perform fast proximity queries on 2D shapes using a pixel grid to perform distance computations, but their technique does not extend to 3D shapes. To overcome the image-space resolution for spline intersections, researchers at SINTEF adapted the serial subdivision algorithm to use the GPU. They accelerate the computations by using the GPU to test for intersections and iteratively subdivide the spline patches until a prescribed accuracy is attained [Briseid *et al.*, 2006; Dokken *et al.*, 2005].

Prior algorithms for proximity queries on spline models used higher-order bounding volumes such as OBBs or swept spheres. Krishnan et al. calculate contact between spline models using a combination of bounding volumes that include spherical shells and OBBs [Krishnan *et al.*, 1998]. Even though these higher-order bounding volumes have low memory requirements, the individual overlap computations are more complex. We make use of an AABB hierarchy in which the bounding boxes are not as tight as higher-order bounding volumes, but reduce the complexity of the computations for each bounding-box pair.

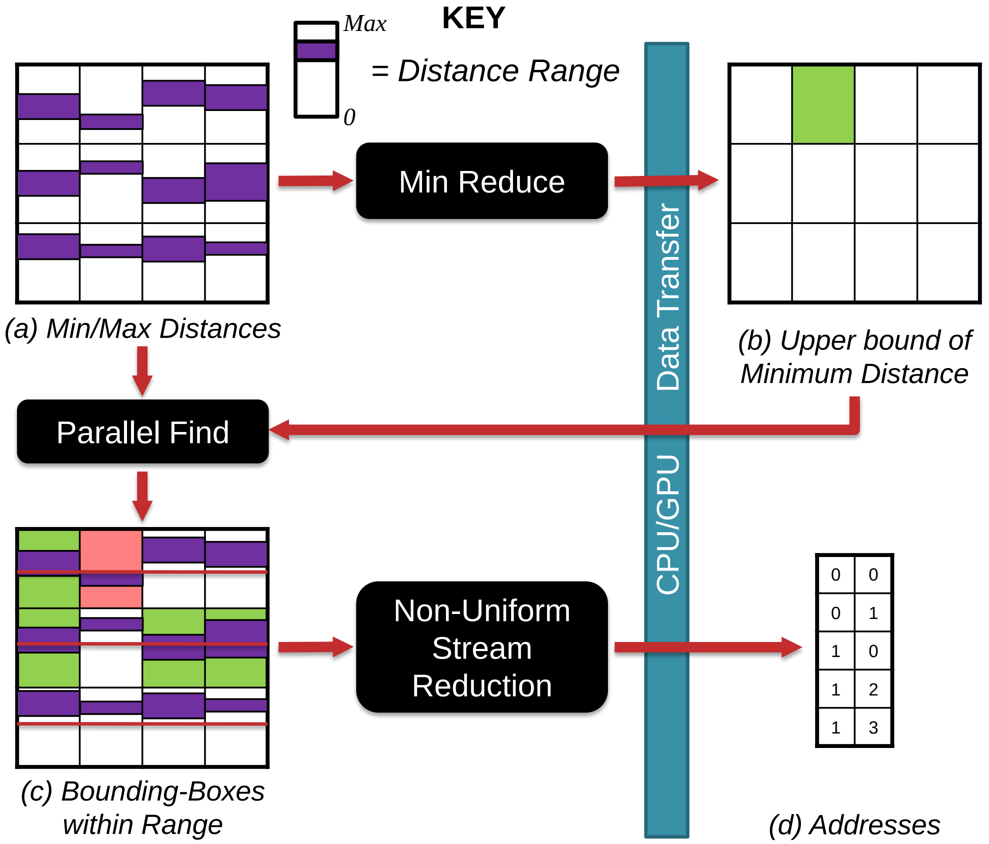
Our algorithm to compute the minimum distance between objects is an hybrid CPU/GPU algorithm that uses the CPU for certain computations that are inherently serial. Lauterbach et al. have recently developed a GPU algorithm where the hierarchy traversal and primitive queries are also performed on the GPU [Lauterbach *et al.*, 2010]. Even though such an exclusive GPU algorithm overcomes the CPU/GPU bottleneck, it requires newer hardware to perform certain operations such as atomic memory writes. However, these operations are not supported by all GPU hardware vendors; as a result, such algorithms may not be readily adopted by the CAD industry. The portions of our algorithm that are performed on the CPU could also be easily ported to GPUs that support atomic operations.

6.3 Distance Queries on NURBS Surfaces

We first present distance queries that are performed on individual NURBS surfaces and later in Section (6.5) extend them to complex objects made up of multiple curved surfaces.

6.3.1 Minimum Distance to a NURBS Surface

The first distance query we accelerate using the GPU is computing the minimum distance and the closest point on a NURBS surface given any point in space. As a first step, we evaluate the NURBS surface as a grid of points at a very fine resolution based on the user-defined tolerances using our NURBS evaluator and construct surface AABBs enclosing four neighboring points. Using these bounding-boxes and the input point, we calculate the range of distances to each bounding box as explained in Section (6.3.2).



0	0
0	1
1	0
1	2
1	3

Figure 6.2: *Schematic of our closest point algorithm showing the inter-communication between the CPU and GPU. The vertical bars represent the range of minimum and maximum distances from the point to each bounding box.*

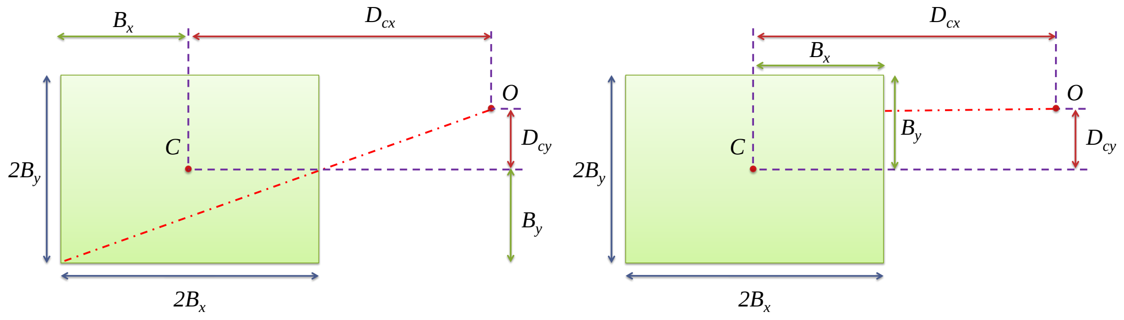
Figure (6.2) shows how our GPU closest point algorithm fits into our hybrid framework. We first use the GPU to compute the minimum and maximum distance to each AABB efficiently in parallel (Figure (6.2a)). These distances are stored using the red and green channels (the choice of channels being arbitrary) in a min/max texture on the GPU. We then perform a parallel reduction in $\log n$ passes on the GPU to find the bounding-box with the minimum lower value for the distance range (Figure (6.2b)). We read back the range of this particular bounding-box. In the next pass, we use the upper bound of this particular bounding-box as a distance cutoff to search for potentially close bounding-boxes. We use the GPU to perform a parallel search on the same min/max texture we computed in the first step to find all the bounding-boxes whose range overlap with the upper bound (Figure (6.2c)). This prunes the list of bounding-boxes to search for the closest point; we read back this smaller list by performing non-uniform stream reduction on the results of the search (Figure (6.2d)).

Once we read back the potentially close bounding-boxes, we approximate the surface patch inside each of the bounding boxes with two triangles formed from the evaluated surface points. We then find the distance to each of these triangles and finally choose the one with the minimum distance. We also find the point lying on the triangle that has the minimum distance as the closest point on the surface. We prove theoretical error bounds for the evaluated minimum distance and the calculated closest point in Section (6.4).

6.3.2 Minimum and Maximum Distance to an AABB

The first step of our minimum distance algorithm requires the computation of the minimum and maximum distance between a point and an AABB. Since we want to perform these computations in parallel for each AABB, the computations have to be efficient and optimized for the GPU. The maximum distance can be computed in a straightforward manner by finding the vertex of the bounding-box that is farthest from the given point. However, to compute the minimum distance, we not only need to find the minimum distance to the vertices of the AABB but also to the faces. The number of computations becomes prohibitively many if we have to check all the possibilities.

We make use of the fact that the bounding-boxes are axis-aligned to efficiently compute the minimum and maximum distance. This makes the calculations simpler and unified for computing both the minimum and maximum distance simultaneously (Figure (6.3)). For computing the maximum distance from a point O to an AABB, we compute the maximum distance along each axis separately and finally take the L_2 norm of the individual maximum distances to find the maximum distance (Equations (6.1) – (6.4)). However, if we extend the same method to compute the minimum distance, we have to make sure that the individual distance components are non-zero; if we directly subtract the half bounding-box widths, we will end up with negative distances. To overcome this, we take the minimum distance along a particular direction as zero if it is negative (Equations (6.5) – (6.8)).



(a) *Maximum Distance* **(b)** *Minimum Distance*

Figure 6.3: *Efficiently computing the maximum and minimum distance between a query point O and an AABB. The example shown here is for the 2D case, but the method can be extended to 3D. See Equations (6.1) – (6.8).*

$$x_{max} = D_{cx} + B_x \tag{6.1}$$

$$y_{max} = D_{cy} + B_y \tag{6.2}$$

$$z_{max} = D_{cz} + B_z \tag{6.3}$$

$$D_{max} = \sqrt{(x_{max}^2 + y_{max}^2 + z_{max}^2)} \tag{6.4}$$

$$x_{min} = \max(D_{cx} - B_x, 0) \tag{6.5}$$

$$y_{min} = \max(D_{cy} - B_y, 0) \tag{6.6}$$

$$z_{min} = \max(D_{cz} - B_z, 0) \tag{6.7}$$

$$D_{min} = \sqrt{(x_{min}^2 + y_{min}^2 + z_{min}^2)} \tag{6.8}$$

This formulation is efficient for GPU implementation, since it has only one branch for each max while computing the minimum distances. We implement these equations using a single fragment program and output the minimum and maximum distance to a texture using the red and green channels. Thus, the minimum and maximum distances are computed simultaneously for all AABBs in parallel. We then use these min/max distances as the input texture for finding the minimum distance to a NURBS surface (Figure (6.2)) as explained in Section (6.3.1).

6.4 Theoretical Bounds

In this section, we give theoretical bounds for both the computed minimum distance and the location of the closest point on the curved surface given any point in space.

Theorem 1. (Minimum Distance Bound) *The computed minimum distance does not deviate from the theoretical minimum distance to the actual surface by more than the surface deviation value K.*

Proof. Let O be the point from which we want to find the minimum distance to a curved surface patch S showed in green in Figure (6.4). Let A_1, A_2, A_3 be three points (of the four points used to construct the bounding-box) evaluated on the surface. The surface can be approximated linearly by triangle $A_1A_2A_3$; the maximum deviation of the linear approximation from the curved surface is K (Equation (2.18)). Let Q be the actual point closest to O on the curved surface and P' be the computed closest point on the triangle. Let P be the closest point to P' on the surface. Since Q is the closest point on the surface from O, $OQ < OP$. From triangle OPP', by applying the triangle inequality to the sides, we get $OP < OP' + PP'$. Since the maximum deviation of the surface from the triangle is K, distance $PP' < K$. Combining these inequalities, we get $OQ < OP' + K$ or $OQ - OP' < K$. This shows that the distance OQ, the theoretical minimum distance, cannot be larger than the computed distance OP' by more than K.

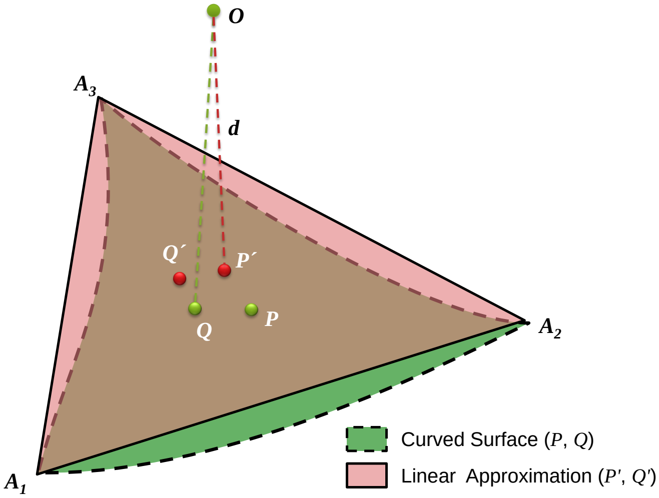
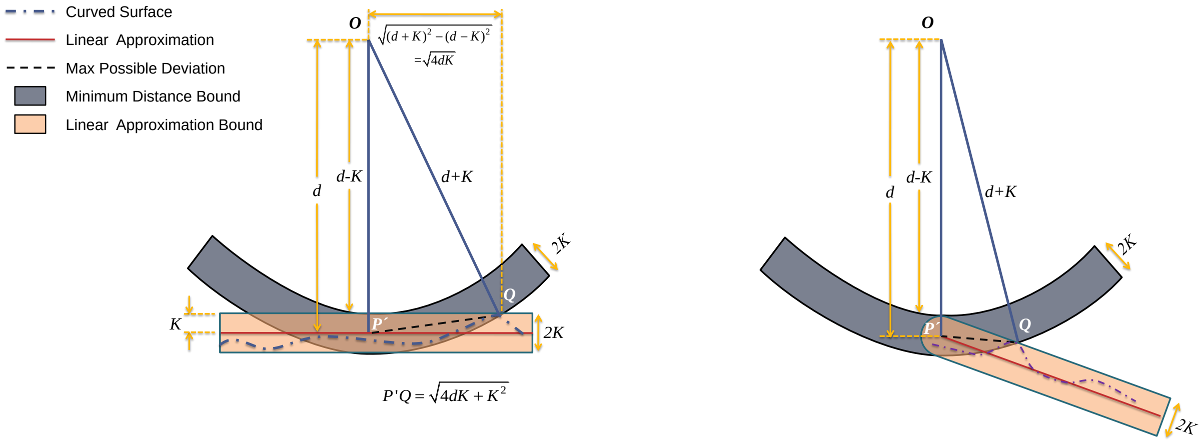


Figure 6.4: *Illustration to prove the bound for the minimum computed distance. The actual curved surface is shown in green while the linearized approximation is shown in orange.*

Now, consider the point on the triangle that is closest to Q, call it Q'. In this case $OP' < OQ'$ since P' is the closest point on the triangle from O. Again from triangle OQQ', we get $OQ' < OQ + QQ'$ and $QQ' < K$ since Q' is the closest point on the triangle from Q. Combining these three inequalities, we get $OP' < OQ + K$ or $OP' - OQ < K$. This shows that the theoretical minimum distance cannot be smaller than the computed distance by K. Combining the minimum and maximum bound on the distance, we get $|OP' - OQ| < K$. □

Thus, from Theorem 1, we know that the theoretical minimum distance is bounded to lie within the range $(d - K, d + K)$, where d is the computed minimum distance. We now show how we use this bound to prove that the location of the closest point we compute is also bounded.



(a) *Case 1*

(b) *Case 2*

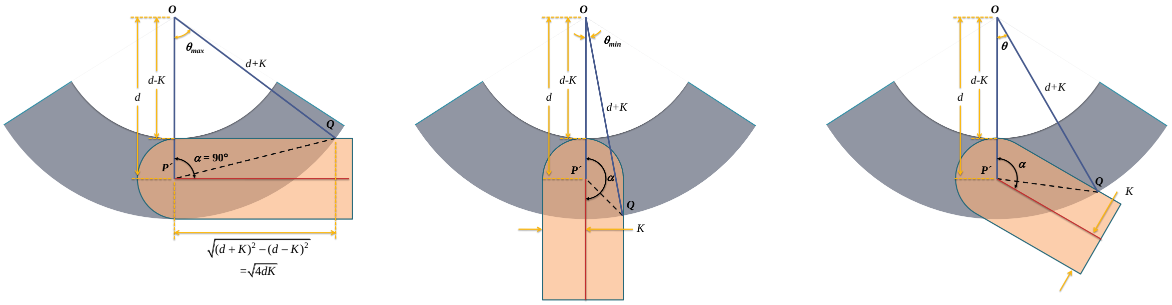
Figure 6.5: *Illustration to evaluate the bound (Theorem 2) for the computed closest point location when the closest point on the plane lies either (a) inside or (b) on the edge of the triangle approximating the surface.*

Theorem 2. (Closest Point Location Bound) *The maximum possible distance between the computed closest point and the theoretical one is $\sqrt{4Kd + K^2}$ where d is the computed minimum distance to the surface.*

Proof. From Theorem 1, the theoretical minimum distance cannot deviate from d by more than K, i.e. $OQ \in [d - K, d + K]$. We have two possible cases: the closest point P' computed on the plane lies inside the triangle used to approximate the surface or it lies on one of the edges of the triangle (see Figure (6.5(a)) and Figure (6.5(b)), which show a 2D cross-section). In the first case (Figure (6.5(a))), the minimum distance bound restricts the theoretical closest point Q to lie in an annular region between spheres with center O and radii $d + K$ and $d - K$ (marked in blue). From our tessellation bound K, we know that the actual surface lies within a region of width $2K$ centered around the approximating triangle (marked in red). Thus the point Q lies in the intersection of these overlapping regions. The maximum possible distance $P'Q$ in this intersecting region is $\sqrt{4Kd+K^2}$. In the second case (Figure (6.5(b))), the approximating triangle is oriented at an obtuse angle with respect to OP'. In this case, the maximum distance in the overlapping region occurs only when OP' is perpendicular to the triangle; for all other angles of rotation of OP', it is always less than $\sqrt{4Kd+K^2}$.

From Theorem 2, we know that there are two possible cases. In the first case, the closest point P' lies inside the triangle and the bound can be computed directly to be $\sqrt{4Kd+K^2}$. However, in the second case, to find the maximum possible value of $P'Q$, we have to consider all possible orientations of the triangle with respect to OP'. Let α denote the angle the triangle makes with OP'; α can vary from 90° to 180° (the two extremes and a general case are shown in Figure (6.6)). Angle α cannot be less than 90° because then P' will no longer be the closest point on the triangle. The angle subtended by $P'Q$ at the center of the sphere, denoted by θ, monotonically decreases from θ_{max} to θ_{min}, as α increases from 90° to 180°. The values of θ_{max} and θ_{min} can be computed to be $\sin^{-1}\left(\frac{\sqrt{4dK}}{d+K}\right)$ and $\sin^{-1}\left(\frac{K}{d+K}\right)$ from Figure (6.6(a)) and Figure (6.6(b)) respectively.

When $90 < \alpha < 180$, $P'Q$ can be computed to be $\sqrt{(d+K)^2+d^2-2d(d+K)\cos\theta}$ from the cosine rule on triangle $OP'Q$ (Figure (6.6(c))). $P'Q$ will be maximized when the term $2d(d+K)\cos\theta$ is minimized, since all the other terms in the expression are positive. $2d(d+K)\cos\theta$ is minimized when θ is the maximum possible value in the range $[\theta_{min}, \theta_{max}]$. Thus $P'Q$ is maximized when $\theta = \theta_{max}$; the extreme case is shown in Figure (6.6(a)) with maximum value of $P'Q$ again being $\sqrt{4Kd+K^2}$ as shown in Figure (6.5(a)). Hence, the maximum possible distance between the computed closest point and the theoretical one is always $\sqrt{4Kd+K^2}$. □



(a) *Upper Limit Case* **(b)** *Lower Limit Case* **(c)** *General Case*

Figure 6.6: *The three different cases that can arise when the closest point computed is on the edge of the triangle.*

Thus, both the computed minimum distance and the location of the closest point are bounded. We show in the Results section that these theoretical bounds translate to realistic values that are useful in practice. Next, we extend our minimum distance computations to compute minimum distance between two NURBS surfaces or two complex CAD objects represented as B-reps.

6.5 Clearance Analysis

6.5.1 Minimum Distance Between Two NURBS Surfaces

We use a method similar to finding the minimum distance from a point to a surface to find the minimum distance between two surfaces. However, it is impractical to use this method directly because the number of distance comparisons increase as $O(n^2)$, where n is the number of AABBs of each surface. Therefore, we make use of a method that uses bounding-box hierarchies to successively refine the number of potentially-close bounding-box pairs. We show that this approach, which is similar to a breadth-first search, can also be fit into our hybrid framework. We perform the search for potentially-close bounding-box pairs in parallel at each level using the GPU.

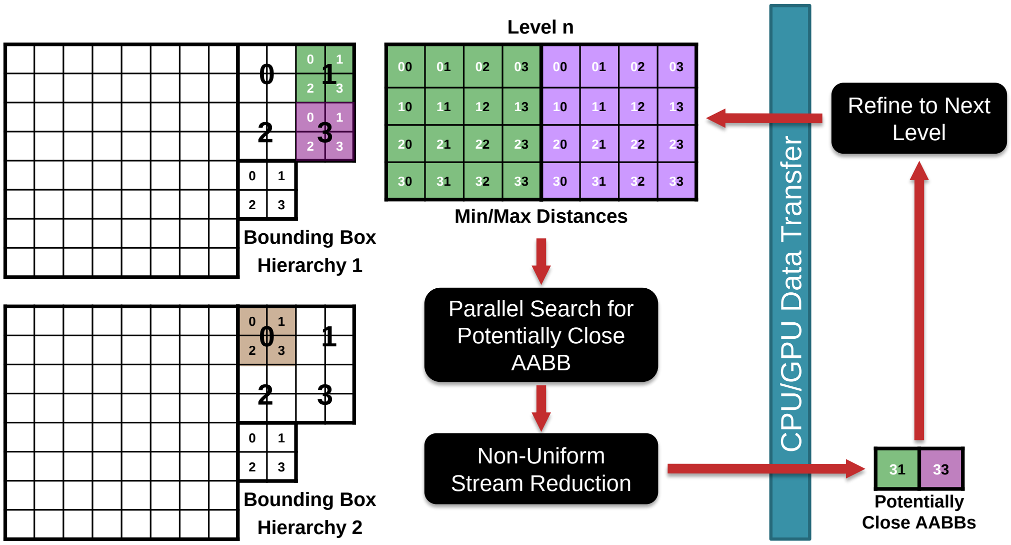


Figure 6.7: *We perform minimum distance computation between two NURBS surfaces with the help of AABB hierarchies for both the surfaces. We compute a list of potentially close bounding-boxes at each level using the GPU and then refine on the CPU until we reach a set of potentially close bounding-boxes at the lowest level.*

We first construct surface AABBs as shown in Figure (2.4); denote these as original AABBs. We then generate a bounding-box hierarchy by recursively combining four AABBs in a level to get a bigger AABB of the next higher level. Thus, we construct an AABB hierarchy starting with the original AABBs and finally reaching a single, level-0 bounding box. This operation can be effectively performed in $O(\log n)$ passes using the GPU. We store the bounding-boxes in a manner that optimizes GPU storage space (Figure (6.7)) similar to mip-map layouts. When the model is transformed (translated or rotated), we fit new AABBs that contain the transformed original AABBs and rebuild the hierarchy. However, we still store and use the original AABBs for fitting after every transformation, since if we keep only the newly fitted AABBs, the bounding-boxes will keep growing in size.

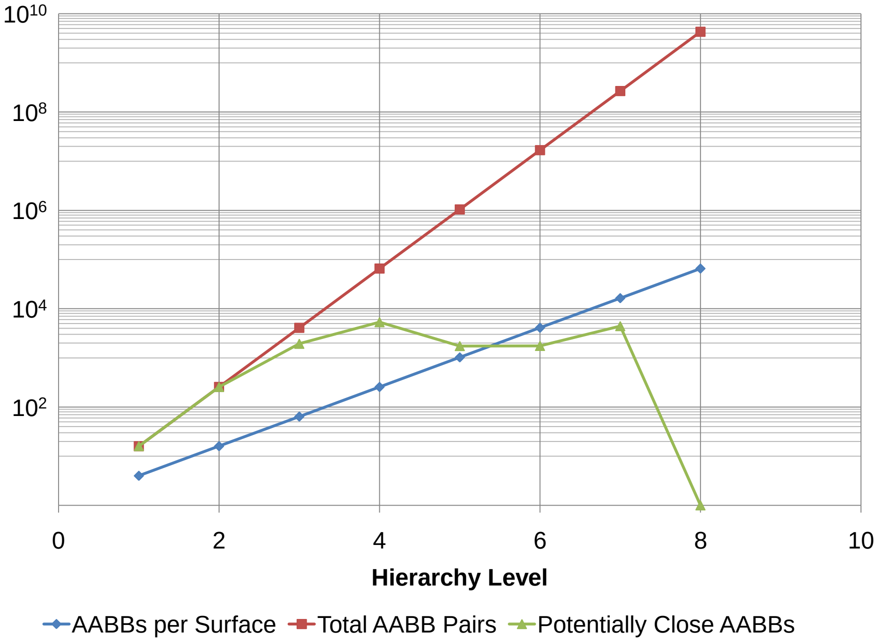


Figure 6.8: *Plot showing the actual number of AABB pairs compared during a typical minimum distance computation. The number of pairs being tested in parallel remains almost constant after level* 3 *of the hierarchy. Note logarithmic scale used for the y-axis.*

We compute the minimum distance between the surfaces by recursively going down the hierarchy and finding potentially-close bounding-boxes at the finest level of the hierarchy. We start at level 1 of the hierarchy where we compute the minimum and maximum distance between four AABBs from surface 1 with each of the four AABBs of level 1 from surface 2, a total of 16 minimum and maximum distance pairs. The method used for finding the minimum and maximum distance between two AABBs is explained in Section (6.5.2). Once we compute the set of minimum and maximum distances, we prune those AABB pairs that are outside the min/max distance range of the closest AABB pair, similar to our method described in Section (6.3.1). We get a list of potentially-close AABB pairs for this level of the hierarchy at the end of the search. We then use the GPU to map the next finer level of the hierarchy, in sets of 4×4 AABB pairs, and repeat finding the potentially-close AABB pairs in the next finer level on the GPU (Figure (6.7)).

Finally at the end of the recursion, we get a list of potentially-closest AABB pairs in the finest or highest level of the hierarchy of both the surfaces. Using a hierarchy to prune AABBs outside the range keeps the number of potentially-close AABB pairs almost constant. Figure (6.8) shows that the number of pairs to be tested increases at first and after level 3 remains almost constant at a few thousand potentially-close pairs. These computations can be done efficiently by the GPU in parallel at each level, as seen in the Results section (Section (6.6)).

Finally, once we obtain all the potentially-closest AABB pairs at the finest level, we compute the closest distance between the surface patches enclosed by these AABBs on the CPU. We perform this operation on the CPU since the list of pairs is usually small; the overhead of setting up the GPU computations would be much higher than the potential performance gains achievable due to GPU parallelism. We approximate each surface patch with two triangles and then compute the minimum distance between the triangles. Similarly, we also compute the pair of closest points.

6.5.2 Minimum and Maximum Distance Between AABBs

We extend our computations described in Section (6.3.2) to compute the minimum and maximum distance between two AABBs (Figure (6.9)). Similar to the point case, we compute the minimum and maximum distance along each dimension and then calculate the overall minimum and maximum distances (Equations (6.9) – (6.16)). As before, if any component is negative while computing the minimum distance, we take that component as zero.

$$x_{max} = D_{cx} + B_{1x} + B_{2x} \tag{6.9}$$

$$y_{max} = D_{cy} + B_{1y} + B_{2y} \tag{6.10}$$

$$z_{max} = D_{cz} + B_{1z} + B_{2z} \tag{6.11}$$

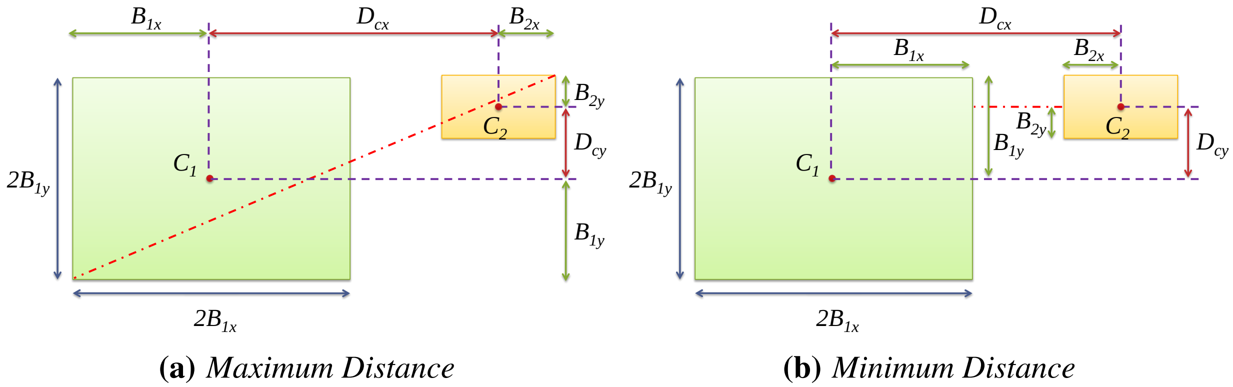
$$D_{max} = \sqrt{(x_{max}^2 + y_{max}^2 + z_{max}^2)} \tag{6.12}$$

$$x_{min} = \max(D_{cx} - B_{1x} - B_{2x}, 0) \tag{6.13}$$

$$y_{min} = \max(D_{cy} - B_{1y} - B_{2y}, 0) \tag{6.14}$$

$$z_{min} = \max(D_{cz} - B_{1z} - B_{2z}, 0) \tag{6.15}$$

$$D_{min} = \sqrt{(x_{min}^2 + y_{min}^2 + z_{min}^2)} \tag{6.16}$$



(a) *Maximum Distance* **(b)** *Minimum Distance*

Figure 6.9: *Computing the maximum and minimum distance between two AABBs. The equations are similar to the point-AABB distance case. See Equations (6.9) – (6.16)*

These equations are implemented using a fragment program on the GPU; we output the values to the red and green channels of a texture. The distances are computed for all potentially-close AABB pairs at a particular level in parallel and are then used for finding the potentially-close AABB pairs in the next level as explained in Section (6.5.1).

6.5.3 Minimum Distance Between Two Trimmed NURBS Surfaces

We extend our NURBS minimum distance computations to find the minimum distance between two trimmed NURBS surfaces. In order to address trimmed NURBS surfaces, we have to cull the bounding-boxes that lie in the trimmed regions of the surface. We generate bounding-boxes for a set of four points evaluated on the surface only if any of the four points lie outside the trimmed region. Since we store the bounding-box data using two four-channel floating-point textures, we can indicate whether the bounding-box is valid by using the fourth alpha channel in the texture. If all the four points lie inside the trimmed region of the NURBS surface, we cull the bounding-box by setting the alpha channel of the corresponding bounding-box texels to zero.

To perform this culling operation, we first generate a trim-texture (Section (4.6)) of the same size as the evaluated points. This gives a one-to-one correspondence for checking whether an evaluated point lies inside the trimmed region. While generating the bounding-boxes for the surface patches on the GPU, an extra test is performed to check if every set of four points lie inside the trimmed region in the parametric space. If so, the bounding-box is culled by setting the alpha channel to be zero; otherwise, it is set to one. Figure (6.10) shows surface bounding-boxes for a trimmed NURBS surface that are not culled. Once we generate the bounding-boxes, we construct the bounding-box hierarchy similar to the method explained in Section (6.5.1). However, we combine only the bounding-boxes that are not culled to generate the hierarchy.

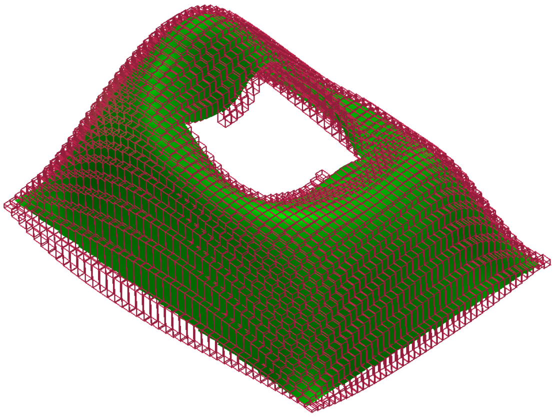


Figure 6.10: *Example of non-culled surface bounding-boxes for a trimmed NURBS surface. The bounding-boxes that lie in the trimmed region are culled.*

After we have generated the bounding-box hierarchy, we use the same algorithm given in Section (6.5.1) to find a list of potentially-close bounding-box pairs. While performing the search operation, we make sure that we do not include the bounding-boxes that are culled in the calculations. Finally, once we obtain all the potentially-closest AABB pairs at the highest level, we approximate each surface patch with two triangles and then compute the minimum distance between the triangles. However, we have to do an additional check to make sure that the computed closest point lies outside the trimmed region of the surface. If the point lies inside the trimmed region, we discard the point and continue the search.

A main implication of the presence of trims is that we cannot always guarantee as tight a tolerance for the minimum distance as in untrimmed surfaces. There are two cases where the tolerances may be looser. The first case happens when the closest point lies on the edge of a trim-curve. In this case, since we do not explicitly find the intersection of the trim-curves and the surface patches that lie inside the closest bounding-boxes, the tolerances calculated for the untrimmed surface cannot be used. However, the tolerance values are still guaranteed to be less than the size of the bounding-box that contains the surface patch. The second case is the degenerate case when the closest point is on an untrimmed feature that is smaller than the parametric tolerance used for creating the trim-texture and the corresponding bounding-box is culled as a result. In this case, since we use the same trim-texture for display, the small feature will also display as trimmed away. When this happens, the user will get visual feedback that the model has not regenerated correctly and can adjust the parametric tolerance accordingly.

6.5.4 Minimum Distance Between Two Complex Objects

Finally, we extend our minimum distance computations between NURBS surfaces to complex objects made up of many NURBS surfaces. CAD systems have support for this query to give feedback about the clearance between the models in an assembly while the user is manipulating them. However, existing systems are not interactive due to long computation times for performing this query. We perform this query in two stages; in the first stage we find a list of potentially close surface pairs and in the second stage we find the minimum distance between the surfaces.

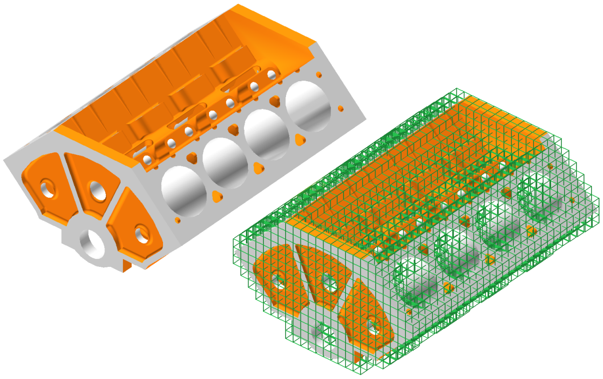


Figure 6.11: *A complex model and its voxel representation. We store the surfaces that intersect with a particular voxel to accelerate the minimum distance computation.*

Voxel-Based First Stage

In the voxel-based approach for the first stage, we construct a grid of voxels in the region occupied by the object (Figure (6.11)). We then consider these voxels as individual AABBs to perform the minimum distance computation. We create the voxel representation of the model as a preprocessing step. We first overlay a regular voxel grid that covers the object completely. We then use the coarse tessellation of the object that is used for display to populate the voxel grid. For each triangle in the tessellation, we find the voxels that the triangle intersects and then add a reference in the voxel to the surface to which the triangle belongs. Thus, each voxel has information about its minimum and maximum point extents that define the AABB and a list of surfaces that intersect it. Since this is done only once per object when the object is loaded for display, we perform this operation on the CPU. In addition, since this is a linear $O(n)$ operation, where n is the number of triangles in the tessellation, it is fast.

As a first step in finding the closest points, we find a set of potentially-close voxel pairs by performing a single pass of minimum distance computation. To perform this operation on the GPU, we map the voxels from the first object to the rows and the voxels from the second object to

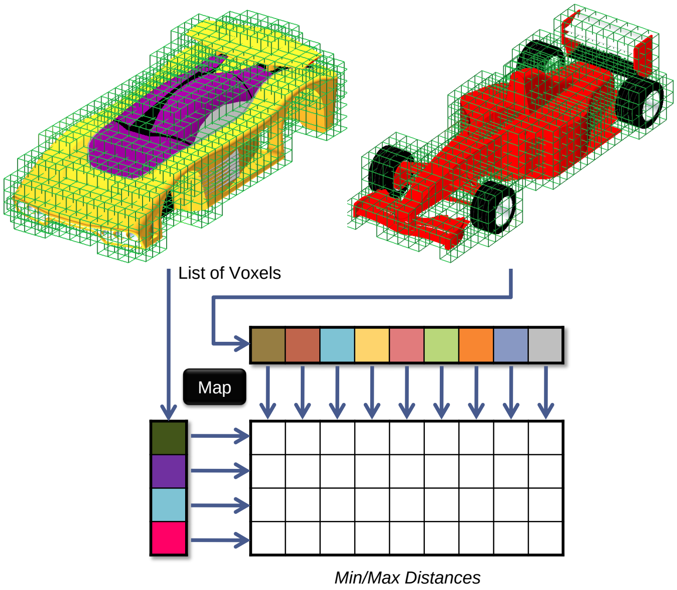


Figure 6.12: *We map the list of voxels of one object to the rows and the other object to the columns of a 2D texture to compute the minimum and maximum distances between the voxels.*

the columns of a 2D texture (Figure (6.12)). We compute the minimum and maximum distances for each voxel pair of the two objects and output these distances to the texture. This texture is then used to find the list of potentially-close voxel pairs that lie within the range of the closest voxel pair (as in Figure (6.2)). We perform non-uniform stream reduction to transfer address information of the potentially-close voxel pairs to the CPU. Since each voxel has information about the surfaces that pass through it, we can create a list of potentially-close surface pairs from these potentially-close voxel pairs. We also make sure that there are no duplicated entries in the surface pairs list, since the same surface can pass through many voxels in the potentially-close voxel pair list.

Surface-Based Second Stage

In the second stage, we compute the minimum distance for each surface pair in the potentially-close surface list using our algorithm explained in Section (6.5.3) or Section (6.5.1), depending on whether the surface is trimmed or not, respectively. We can then output the minimum distance or clearance between the two objects as the minimum distance computed from all the surface pairs. We also output the points on each surface as the closest points on the two objects. Even though we use the coarse tessellation for constructing the voxel grid, we do not use it for the minimum distance computations. Our computations are performed using the NURBS surfaces directly and lie within the computed bounds. Hence, they are more accurate than only using the tessellation for the computations. In the next section, we show that our algorithm performs orders of magnitude faster than a commercial CPU-based kernel, while simultaneously achieving similar or higher accuracy.

6.6 Results

We timed our GPU-accelerated distance queries on a 2.66GHz (quad-core) CPU running Windows Vista with 4GB of RAM and an NVIDIA Quadro FX5800 with 4GB graphics memory. We compare our timings to perform the geometric queries with those of the commercial solid modeling kernel ACIS (v20).

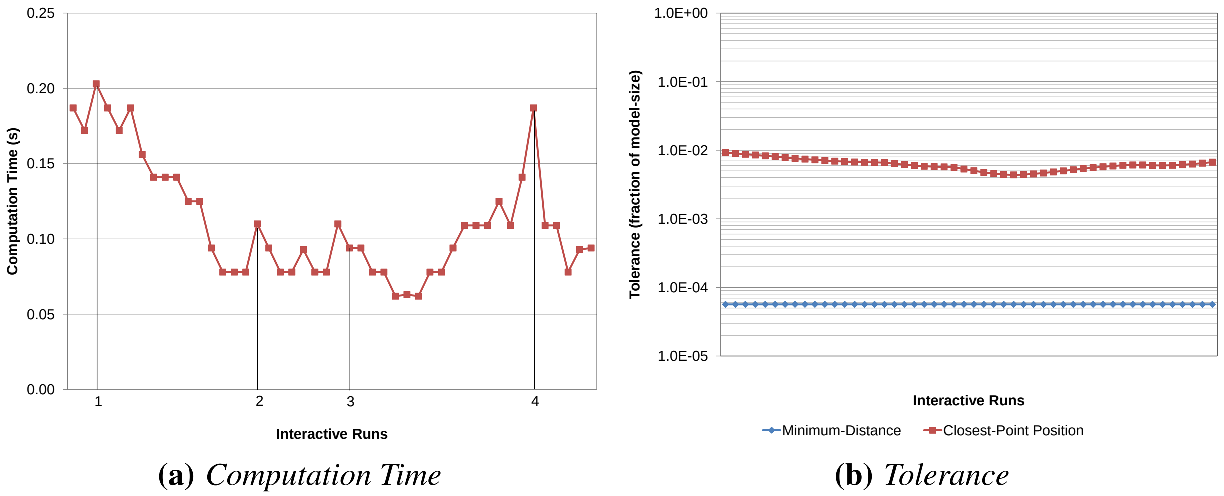
NURBS Minimum Distance Timings

We timed our minimum distance computations between two curved NURBS surfaces by interactively translating as well as rotating one surface made of 199×33 control points relative to the another surface made of 100×105 control points (Figure (6.1(a))). Figure (6.13(a)) shows the interactive computation times recorded during the interaction; the computation times were less than 0.15 seconds for most positions, a near-interactive average frame rate of 9.07 fps. Figure (6.13(b)) shows the distance and position tolerances computed corresponding to the runs in Figure (6.13(a)). Since these tolerance values are dependent on the model size, we report them as a fraction of the model size in order to make them consistent with tolerance definitions used by ACIS [Corney and Lim, 2001]; a value of 0.01 corresponds to 1% of the model size. The model size is the length of the diagonal of the smallest AABB that will enclose the model.

Position	ACIS Time (s)	GPU Time (s)	Speed-up
1	64.4	0.218	296x
2	65.4	0.109	600x
3	66.9	0.093	720x
4	66.2	0.171	387x

Table 6.1: *Time for performing minimum distance computations between two NURBS surfaces.*

We recorded the time taken by ACIS to compute the minimum distance at some arbitrarily chosen positions of the NURBS surfaces relative to one another by using the API command *api_check_face_clearance*. We set the tolerance value for ACIS to be 4×10^{-2}, well looser than our closest-point position tolerances reported in Figure (6.13(b)). We chose this tolerance because we can guarantee it in our algorithm even if the closest point lies on a trim-curve edge; the tolerances can be guaranteed to be much tighter if the closest point does not lie on a trim-curve edge. Table (6.1) summarizes the results of our NURBS minimum distance computations for these positions, including the positions where our algorithm was slowest (note that there is little variation in the ACIS timings for different positions). The GPU accelerated algorithm is at least two orders of magnitude faster than ACIS. This can be explained by the fact that ACIS first tessellates the object to get a dense mesh of points on the surface and then performs the minimum distance computation on these points. We on the other hand use our fast NURBS evaluator to evaluate the surface and



(a) *Computation Time* **(b)** *Tolerance*

Figure 6.13: *Interactive times for evaluating the minimum distance between two NURBS surfaces and the corresponding distance and position tolerances scaled with respect to the maximum model size. The four positions used for ACIS timings in Table (6.1) are marked.*

construct surface bounding-boxes in real time. In addition, we not only achieve better performance but also a higher accuracy; our results have theoretical bounds that are practical for use in a CAD system.

Operation	**Time (s)**
NURBS evaluation Normal evaluation Bounding-box construction Bounding-box hierarchy construction	0.036 0.038 0.019 0.008
Total evaluation time	0.101
AABB distance evaluation Finding potentially close AABBs Finding closest triangles on CPU	0.027 0.045 0.048
Total hierarchy traversal time	0.120
Total computation time	0.221

Table 6.2: *Break-down of the timings for performing different operations of the minimum distance computation algorithm.*

Table (6.2) lists the break-down of the timings for performing different operations while computing the minimum distance between two NURBS surfaces shown in Figure (6.1(a)). It can be seen that evaluation and hierarchy traversal operations have similar run times.

Trimmed NURBS Minimum Distance Timings

Table (6.3) summarizes the results for computing the minimum distance between two trimmed NURBS surfaces, where the bottom surface from Figure (6.1(a)) is replaced by a trimmed version of the same surface (Figure (6.10)). The surfaces were timed by positioning them at the same positions as in Table (6.1) and using the same tolerance of 4×10^{-2} for ACIS. As expected, the GPU timings for finding the minimum distance between trimmed NURBS surfaces are slightly higher than the timings for un-trimmed surfaces. This is because of the extra tests that are performed at each stage to exclude the trimmed regions from the computations. However, the timings are still at least two orders of magnitude faster than ACIS.

Position	ACIS Time (s)	GPU Time (s)	Speed-up
1	55.9	0.234	239x
2	57.7	0.125	461x
3	59.1	0.109	542x
4	58.5	0.187	313x

Table 6.3: *Time for performing minimum distance computations between two trimmed NURBS surfaces.*

Object Clearance Timings

We performed object clearance computations using the CAD models listed in Table (6.4); the models are made of trimmed NURBS surfaces and are of approximately the same complexity as standard CAD models used in a mechanical assembly. We used a voxel grid of 40 × 40 × 40 to perform the first-stage of the minimum distance computations. The objects were also tessellated to a coarse level that is sufficient for display; the number of triangles in this tessellation is given in Table (6.4).

Minimum distance queries were performed between the object pairs shown in Table (6.5); the objects were randomly positioned with respect to each other to perform the queries using both ACIS and our GPU accelerated algorithm. We use parametric tolerances that are at least as tight as those specified in the models are. This guarantees that we can accurately calculate the tolerances. We used the api function call *api_check_solid_clearance* to compute the minimum distances in ACIS. It can be seen that the GPU accelerated algorithm is again at least an order of magnitude faster than ACIS.

Object	Surfaces	Triangles
Car Body	80	7134
Toy Car	127	17170
Scooby	157	72094
Plane	215	68696
Space Ship	631	37914

Table 6.4: *Complexity of the objects used for the minimum distance computations. The number of triangles shown is the default coarse level of tessellation used for display.*

Object1	Object2	ACIS		GPU		Improvement	
		Time (s)	Tolerance	Time (s)	Tolerance	Time	Tolerance
Toy Car	Car Body	9.73	10^{-2}	0.672	2.5×10^{-5}	14x	408x
Car Body	Car Body	17.34	10^{-2}	0.439	2.2×10^{-5}	39x	463x
Scooby	Scooby	70.62	10^{-1}	0.382	13.7×10^{-5}	185x	728x
Plane	Plane	156.03	10^{-1}	1.501	5.2×10^{-5}	104x	1938x
Plane	Space Ship	263.01	10^{-1}	0.794	2.9×10^{-5}	331x	3453x

Table 6.5: *Time for performing minimum distance computations between different complex objects.*

6.7 Conclusions

We have developed a hybrid framework that uses GPUs to accelerate distance computations. The results of our algorithms have theoretical bounds that are tighter than the tolerances required for CAD. They make use of actual surface data and not just the tessellation, which make them independent of tessellation errors. We can also compute minimum distances between trimmed NURBS surfaces, which make the implementation of our algorithms in an existing CAD system simpler. We also show tremendous performance improvements over existing commercial CPU-based systems.

We find that having alternating serial and parallel stages and using the map-reduce motif for parallelism to be ideally suited for developing geometric algorithms that use the GPU. In addition, the parallel stages can be easily modified and executed on a multi-core CPU in the absence of a powerful GPU. Our framework provides for maximum flexibility and optimized performance in developing fast geometric algorithms.

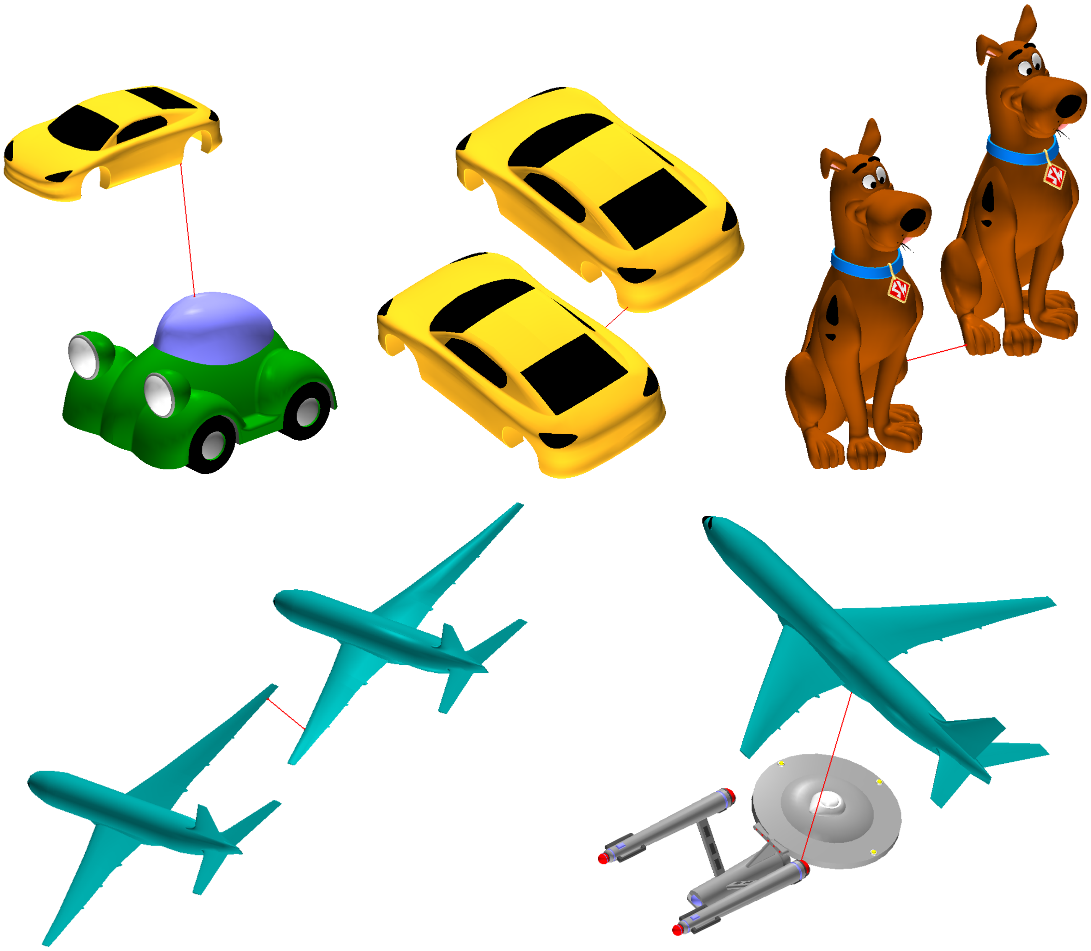


Figure 6.14: *The different pairs of objects that were timed for the minimum distance computations.*

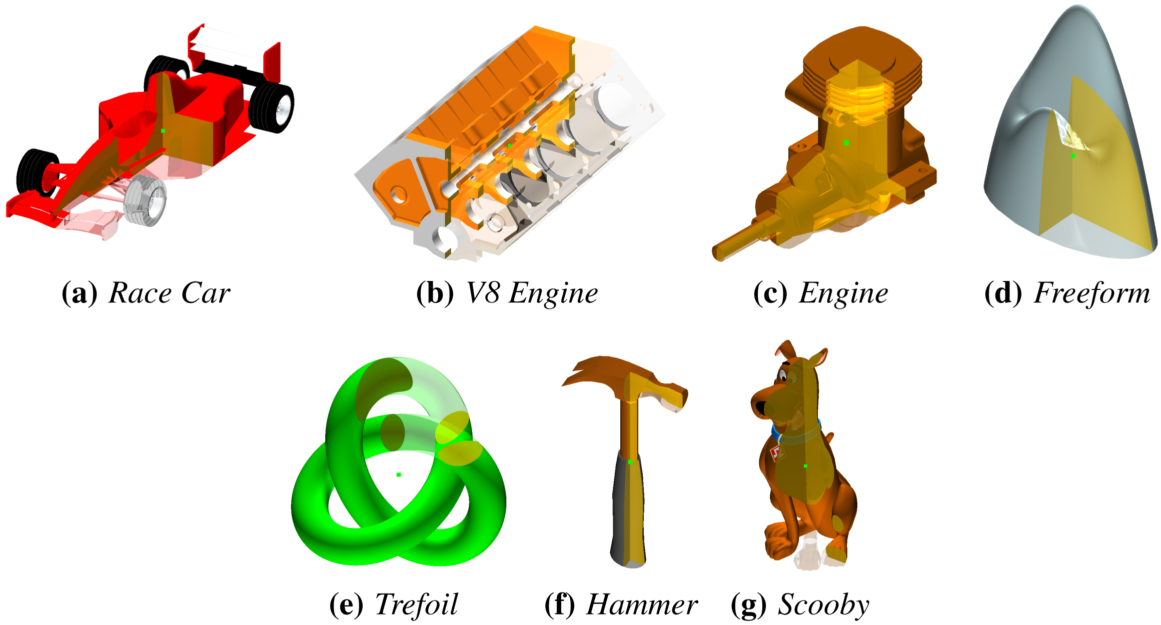
Chapter 7

Moment Computation

7.1 Introduction

In this chapter we present algorithms for computing accurate geometric moments of solid models that are represented using multiple trimmed-NURBS surfaces. We have developed a framework that makes use of NURBS surface coordinates and normals to evaluate surface integrals of trimmed NURBS surfaces in real time. With our framework, we can compute volume and moments of solid models with theoretical guarantees. The framework also supports local geometry changes, which is useful for providing interactive feedback to the designer while the solid model is being designed. We can compute the center of mass and check for stability of the solid model interactively. Applications of such real-time moment computation include deformation modeling, animation, and physically based simulations.

Geometric moments of solid bodies are intrinsic properties of their underlying shape that can provide important design cues to aid in Computer-aided Design (CAD). Computing moments is also essential for physically based simulations that help in improving realism in animations. In addition, many Computer-Aided Manufacturing (CAM) analyses depend on computing accurate volume, center of mass, and moments of inertia. For example, a part that is fixtured close to its center of mass will reduce unbalanced loads on the fixturing system. The moments of inertia can be used to compute the loads that might be transferred to the fixturing system. Although designers have an intuitive sense of the location of the center of mass and principle moments of inertia, accurate feedback about these properties can be a valuable asset while designing. The zeroth-order moment measures the volume enclosed by the solid body. Computing the volume of an object helps the designer estimate the amount of raw material that might be required to manufacture the particular part. This is essential especially in the case of molding, where the volume computation is essential to maintain quality while manufacturing. If a mold is filled incompletely, it leads to voids in the final part; conversely, filling a mold with excess material leads to flash that needs to be trimmed later.



(a) *Race Car* **(b)** *V8 Engine* **(c)** *Engine* **(d)** *Freeform*

(e) *Trefoil* **(f)** *Hammer* **(g)** *Scooby*

Figure 7.1: *Computing the volume and center of mass of complex objects made of multiple trimmed-NURBS surfaces. The center of mass is marked with a green dot; the objects are rendered partially transparent.*

Computing accurate moments of CAD models interactively is not straightforward due to the presence of curved freeform surfaces. Moments of objects made of such freeform surfaces are currently being computed by commercial solid modeling software by first evaluating and tessellating the surfaces and then computing the moments of the tessellated object by projecting them to a common plane that passes through the middle of the object [Spatial Corporation, 2009b]. This approach, in addition to being slow and computationally intensive, is dependent on the tessellation resolution for the accuracy of the solution; the surface has to be very finely tessellated to get the required accuracy. Recent advances in programmable GPUs provide an alternative to accelerate the computations. We have developed a GPU-accelerated algorithm to compute moments of solid objects made of trimmed NURBS surfaces interactively. These algorithms exploit the parallelism of the GPU to provide immediate visual feedback to the designer. The algorithm updates the moment values interactively while the designer changes the design of the part.

In our method, we calculate moments by first converting the volume integrals for moments to surface integrals using the divergence theorem. We then use the GPU to compute these surface integrals. Thus, our algorithm is not restricted to computing moments but can be used to compute general case surface integrals of NURBS surfaces. Computing surface integrals of freeform surfaces form an important part of several physical simulation algorithms, including finite element analysis. In such cases, our algorithm provides a framework to compute surface integrals of NURBS surfaces rapidly and accurately. We have implemented different numerical integration techniques that can be used to evaluate surface integrals of NURBS surfaces on the GPU.

7.2 Related Work

Volume and moment computations have been fundamental to many geometric applications and the literature covering them is vast. We provide a brief summary of some key papers that are close to our work.

Properties such as volume, moment of inertia, etc. can be classified as integral properties that are defined by triple (volumetric) integrals over subsets of three-dimensional Euclidean space. Initial work on computing volume of curved objects by approximating them with polyhedral facets were performed by [Messner and Taylor, 1980]. However, the errors introduced due to the approximation were not analyzed. Lee and Requicha published a two-part paper that discussed the methods that were known up to that time for computing the integral properties of solids. In their first paper ([Lee and Requicha, 1982a]), they summarize the methods for computing volume and moments of solid bodies represented using Boundary Representation (B-rep), primitive instancing etc. In their subsequent paper ([Lee and Requicha, 1982b], they focus on Constructive Solid Geometry (CSG) and propose a new algorithm based on cellular approximation using octrees. They also predict the accuracy of their algorithm in approximating the integral properties.

Boundary Representations (B-reps) have become the de facto standard for representation of solid models in current CAD systems. Lee and Requicha classified the different algorithms for computing moments of B-reps as either direct integration methods or divergence theorem methods. For polyhedral models, direct integration methods can be easily applied since the integration is performed over planes. [Cattani and Paoluzzi, 1990] developed a finite integration method for the computation of various order monomial integrals over polyhedral solid and surfaces. This method can be used for exact evaluation of inertial properties of homogeneous polyhedral objects. Mirtich [1996] developed a fast algorithm to compute moments of polyhedral objects by successively applying the divergence theorem. [Timmer and Stern, 1980] developed a computational scheme that operates directly on parametric bi-cubic spline patches to compute the inertial properties of solids. Their method relied on computing the intersection curves of adjacent surfaces of the solid accurately to compute the integral properties. For interactive applications, [Gonzalez-Ochoa *et al.*, 1998] developed a method to compute moments of piecewise polynomial surfaces that is based on the divergence theorem. They outline several applications in animation where the model shape can be changed to match the moments. However, their method relies on modeling the objects using a particular type of cubic spline surfaces, C^1-surface splines, to maintain interactivity. For graphical animation applications that use subdivision surfaces, [Peters and Nasri, 1997] developed a method to compute the volume of a solid enclosed by subdivision surfaces by estimating the volume of the local convex hull near extraordinary points. [Soldea *et al.*, 2002] developed an analytical method for computing moments of free-form objects that use either parametric surfaces or a constant set of trivariate functions as boundary. In their method, they compute integrals of b-spline blending functions to compute the moments of free-form objects.

GPU geometric algorithms are currently gaining popularity due to the performance gains achievable by using GPUs. However, algorithms to compute geometric moments were limited to volume computations since many of them calculated the volume by rendering to the screen. [Kim *et al.*, 2006] use the depth buffer while rendering to the screen to compute the volume of general shapes and use it for buoyancy simulation. [Khardekar and McMains, 2006; Khardekar, 2008] have developed a GPU algorithm to compute the undercut volumes in molds that can then be used to choose an optimal parting direction while using multi-part molds. GPUs were also used for solving standard geometric problems such as surface-surface intersections [Briseid *et al.*, 2006] and collision detection [Govindaraju *et al.*, 2003; Kipfer *et al.*, 2004; Kolb *et al.*, 2004; Greß *et al.*, 2006]. However, there has been limited use of GPUs to compute integral properties for 3-dimensional solid bodies since algorithms that render to the screen are limited to image-space precision.

One of the advantages of developing a GPU algorithm that uses the divergence theorem is that the integration method can also be used for the evaluation of surface integrals of polynomial forms. These forms commonly arise in physical simulations that use Finite Element Analysis (FEA). Scalar valued functionals, such as energy functionals, are usually evaluated as surface integrals; they can be used to measure the quality of surfaces [Kobbelt, 1997]. Recently, researchers at the University of Wisconsin have developed a method of analyzing 3-D beams by performing surface integration over the boundary of the beam [Samad and Suresh, Accepted 2010; Jorabchi *et al.*, 2009].

On analyzing the related work in the field, we find that even though numerous methods for computing moments and surface integrals exist, there has been only limited analysis on the accuracy of the computations. In addition, there has been limited work on parallelizing the computations to improve their performance. We will try to address both these issues in this work.

7.3 NURBS Surface Bounds

Our moment computation algorithms build on our previous work on GPU NURBS evaluation and modeling. In our NURBS modeling work, in order to bound the errors, we find the maximum possible deviation K of a curved surface from the linearized approximation. These bounds were explained in detail in Section (2.5). We use this constant K in computing the bounds for our algorithms.

Given any three nearby points evaluated on the surface using the uniform grid of size $n \times m$, we can approximate the surface linearly using the triangle formed using these points (Figure (7.2)). We can also bound the coordinates of the curved surface with two parallel triangles that are at a distance K from the linear approximation, since the maximum possible deviation of the surface is K.

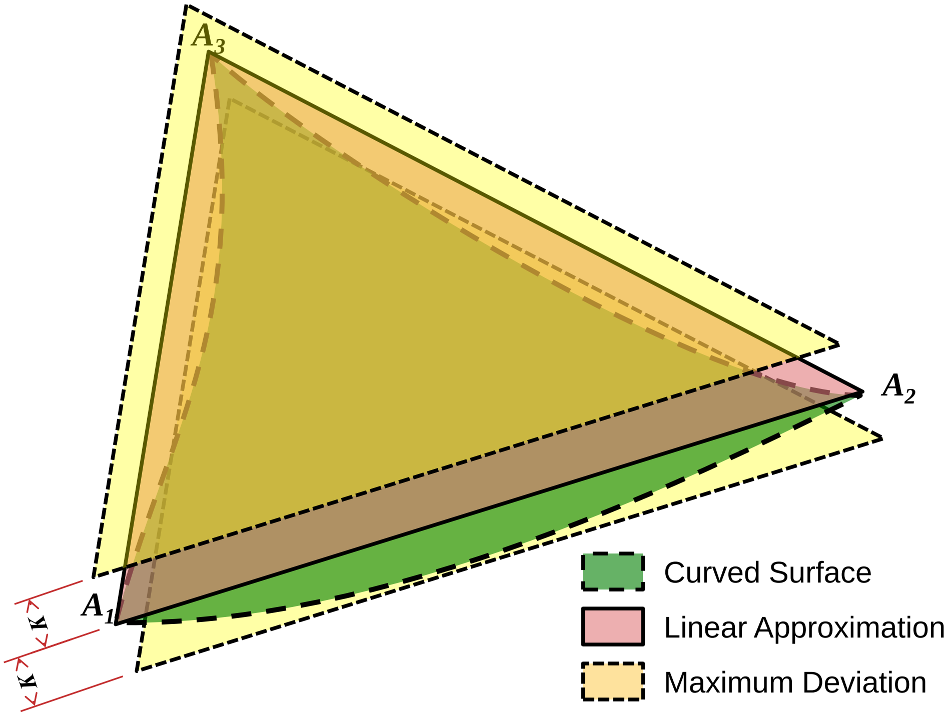


Figure 7.2: *The maximum possible deviation of the actual surface from a linear approximation is K. In this case, the surface is bounded by two parallel triangles that are at a distance K from the linear approximation.*

7.4 Mathematical Formulation

In this section, we briefly explain the mathematical formulations required for moment computations; for a more detailed explanation please refer to [Rudin, 1976]. We make use of Gauss's divergence theorem to convert volume integrals to an integral over the boundary surface of the volume. The divergence theorem is a special case of the generalized n-dimensional Stokes' theorem that is restricted to three dimensions.

Theorem 3. (Divergence Theorem) *Given a vector field* $\mathbf{f}$ *defined over a closed bounded region,* $V \subset \Re^3$*, whose boundary is a piecewise smooth orientable surface S, the volume integral of the divergence of* $\mathbf{f}$ *over V equals the surface integral of the normal component of* $\mathbf{f}$ *over S.*

$$\nabla \cdot \mathbf{f} = \sum \frac{\partial f_i}{\partial x_i} \tag{7.1}$$

$$\int_V \nabla \cdot \mathbf{f} dV = \int_S \mathbf{f} \cdot \hat{\mathbf{n}} dS \tag{7.2}$$

Equations (7.2) formalizes the statement of the divergence theorem. However, as noted by the theorem statement, the theorem is applicable only if both the vector field **f** and the region V satisfy some basic conditions. The vector field must be continuous and have continuous first partial derivatives in the region containing V. In addition, V itself should be closed and its boundary surfaces must be orientable and piecewise continuous. The piecewise continuous surface condition expands the applicability of the theorem to many practical 3-dimensional objects, since it overcomes the limitation of having undefined normals at sharp edges of objects. The divergence theorem is applicable to any 3-dimensional object as long as it has only 2-manifold surfaces. Using the divergence theorem, we can convert volume integrals, which are difficult to evaluate for complex solid objects made of multiple trimmed NURBS surfaces, to surface integrals that can be easily evaluated over the NURBS surfaces.

The evaluations of surface integrals require the calculation of surface normals. We will briefly present the equations for evaluating NURBS derivatives and normals in Section (7.4.1); please refer to [Piegl and Tiller, 1997] for more details.

7.4.1 Evaluation of NURBS Normals

Recall that the parameterized NURBS surface can be represented as a 3-component vector (Equation (7.3)) which is evaluated as the 4-component vector shown in Equations (7.4). The N_i^ps and N_j^qs are the B-spline basis functions of degree p and q respectively; the $(x_{ij}, y_{ij}, z_{ij}, w_{ij})$ coordinates are the NURBS control points defined as a quadrilateral mesh.

$$\mathbf{S}(u,v) = \frac{\mathbf{X}}{w}, \mathbf{X} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{7.3}$$

$$\begin{pmatrix} x \\ \\ y \\ \\ z \\ \\ w \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u)N_j^q(v)x_{ij} \\ \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u)N_j^q(v)y_{ij} \\ \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u)N_j^q(v)z_{ij} \\ \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u)N_j^q(v)w_{ij} \end{pmatrix} \tag{7.4}$$

Given a NURBS surface, we can evaluate the partial derivatives and normals; Equation (7.5) gives the partial u-derivative for the NURBS surface that is evaluated from the derivatives of the basis function of degree p with respect to u, represented as $N_{i,u}^p(u)$. The partial derivative of the surface with respect to v can also be evaluated in a similar manner. In this work, we assume all the weights (w) are positive and hence no poles can occur in S or its partial derivatives. Finally, the normal to the surface is evaluated as the cross-product of the u and v partial dervatives (Equation (7.7)).

$$\mathbf{S}_{,u}(u,v) = \frac{\mathbf{X}_{,u}w - \mathbf{X}w_{,u}}{w^2} \tag{7.5}$$

$$\begin{pmatrix} x_{,u} \\ y_{,u} \\ z_{,u} \\ w_{,u} \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^{p}(u)N_{j}^{q}(v)x_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^{p}(u)N_{j}^{q}(v)y_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^{p}(u)N_{j}^{q}(v)z_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^{p}(u)N_{j}^{q}(v)w_{ij} \end{pmatrix} \tag{7.6}$$

$$\mathbf{n}(u,v) = \mathbf{S}_{,u}(u,v) \times \mathbf{S}_{,v}(u,v) \tag{7.7}$$

7.4.2 Surface Integrals of Parametric Surfaces

Surface integrals of parametric surfaces are straightforward to compute due to the presence of an underlying 2-dimensional parameterization. We can convert the surface integrals to integrals over the parametric domain by changing the variables. In Equation (7.8), P represents the parametric (u,v) domain and J is the Jacobian for the transformation. It can be shown that the Jacobian can be computed to be numerically equal to the length of the normal of the parametric surface (Equation (7.9)).

$$\int_S dS = \int_P |J|\, dP \tag{7.8}$$

$$|J| = |\mathbf{n}| \tag{7.9}$$

The volume integrals simplify with the application of the divergence theorem as given by Equation (7.10). In particular, for NURBS surfaces, the parametric domain is a square domain with the (u,v) range $[0,1] \times [0,1]$ and dP can be replaced with the product $du\,dv$.

$$\begin{aligned}\int_V \nabla \cdot \mathbf{f} dV &= \int_S \mathbf{f} \cdot \hat{\mathbf{n}} dS \\ &= \int_P \mathbf{f} \cdot \hat{\mathbf{n}} \left|\mathbf{n}\right| dP \\ &= \int_P \mathbf{f} \cdot \mathbf{n} dP \\ &= \int_P \mathbf{f} \cdot \mathbf{n} du dv\end{aligned} \tag{7.10}$$

7.4.3 Moments of Solid Bodies

By choosing appropriate vector functions for **f**, we can compute the moments of solid bodies with uniform densities. By applying the divergence theorem to the solid body, we can compute the moments by computing the contribution of each surface and sum the results. In Equation (7.11), P_i represents the parametric surfaces that make up the solid body.

$$\int_V \nabla \cdot \mathbf{f} dV = \sum_i \int_{P_i} \mathbf{f} \cdot \mathbf{n} du dv \tag{7.11}$$

By setting $\nabla \cdot \mathbf{f} = 1$, we get the zeroth-order moment, M_0, the volume of solid body. However, there are many different choices for **f** that satisfies this condition. One option is to choose **f** as shown in Equation (7.12).

$$M_0 = \sum_i \int_{P_i} \begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix} \cdot \mathbf{n} du dv = \sum_i \int_{P_i} z n_z du dv \tag{7.12}$$

The first-order moments, defined by Equations (7.13)–(7.16), can also be computed by carefully choosing **f**. For example, in order to compute the first-order moment M_x, we need to set $\nabla \cdot \mathbf{f} = x$. Similar to the volume computation case, there are several choices for **f** that satisfy this requirement; we choose the x and y components to be both 0, and the function xz for the z component. The other first-order moments can be computed in a similar manner. The center of mass C_M of the object is computed by dividing the first-order moments by the volume of the object (Equation (7.17)).

$$\mathbf{M_1} = \begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix} = \begin{pmatrix} \int_V x dV \\ \\ \int_V y dV \\ \\ \int_V z dV \end{pmatrix} \tag{7.13}$$

$$M_x = \sum_i \int_{P_i} x z n_z \, du \, dv \tag{7.14}$$

$$M_y = \sum_i \int_{P_i} y z n_z \, du \, dv \tag{7.15}$$

$$M_z = \sum_i \int_{P_i} \left(\frac{z^2}{2} \right) n_z \, du \, dv \tag{7.16}$$

$$\mathbf{C_m} = \begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix} = \begin{pmatrix} M_x / M_0 \\ M_y / M_0 \\ M_z / M_0 \end{pmatrix} \tag{7.17}$$

The second-order moments form the components of the inertia tensor, I, given by Equation (7.18). The components of the inertia tensor can be computed using Equations (7.19)–(7.24).

$$\begin{aligned} I &= \begin{pmatrix} M_{yy} + M_{zz} & -M_{xy} & -M_{xz} \\ -M_{xy} & M_{xx} + M_{zz} & -M_{yz} \\ -M_{xz} & -M_{yz} & M_{xx} + M_{yy} \end{pmatrix} \\ &= \begin{pmatrix} \int_V (y^2 + z^2) \, dV & -\int_V xy \, dV & -\int_V xz \, dV \\ -\int_V xy \, dV & \int_V (x^2 + z^2) \, dV & -\int_V yz \, dV \\ -\int_V xz \, dV & -\int_V yz \, dV & \int_V (x^2 + y^2) \, dV \end{pmatrix} \end{aligned} \tag{7.18}$$

$$M_{xx} = \sum_i \int_{P_i} x^2 z n_z \, du \, dv \tag{7.19}$$

$$M_{yy} = \sum_i \int_{P_i} y^2 z n_z \, du \, dv \tag{7.20}$$

$$M_{zz} = \sum_i \int_{P_i} \left(\frac{z^3}{3} \right) n_z \, du \, dv \tag{7.21}$$

$$M_{xy} = \sum_i \int_{P_i} xyzn_z du dv \tag{7.22}$$

$$M_{yz} = \sum_i \int_{P_i} y \left(\frac{z^2}{2}\right) n_z du dv \tag{7.23}$$

$$M_{xz} = \sum_i \int_{P_i} x \left(\frac{z^2}{2}\right) n_z du dv \tag{7.24}$$

7.5 Moment Computation Algorithm Overview

We first give a broad overview of our algorithm that makes use of the theoretical formulation explained in Section (7.4). Given a solid object that is made of multiple trimmed NURBS surfaces, we compute the total moment by summing the moment contribution from each surface (Figure (7.3)). If the surface is a flat plane, we directly compute its moment contribution by using the triangulation of the plane. If it is a NURBS or a trimmed NURBS surface, we compute its moment contribution by performing surface integration using our GPU algorithm. It must be noted that the surfaces of the object must be 2-manifold for the computed moments to be valid. However, the main advantage of computing the moments from surface integrals is that the algorithm is robust in handling the small gaps between trimmed surfaces that exist in a "tolerant" solid model, since we do not evaluate points on the edges. Tolerant solid modeling provides a framework that seamlessly covers these small gaps; any two points that lie within a user-defined tolerance is taken to represent the same point. This is required to interpret the model as being watertight when it has many trimmed-NURBS surfaces that vary in parameterization along their common edges.

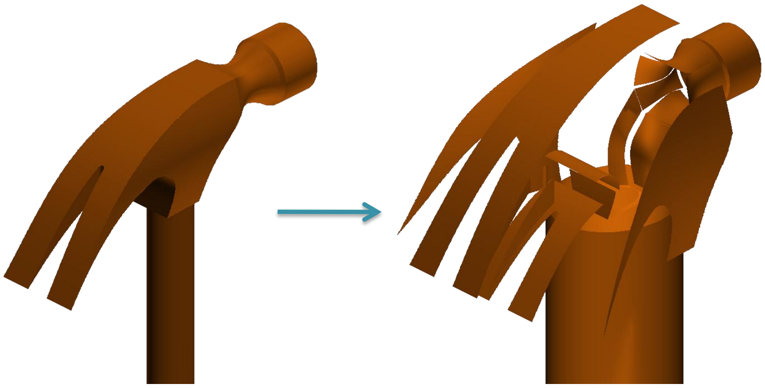


Figure 7.3: *We decompose the object into individual boundary surfaces and then compute the moment contribution from each surface.*

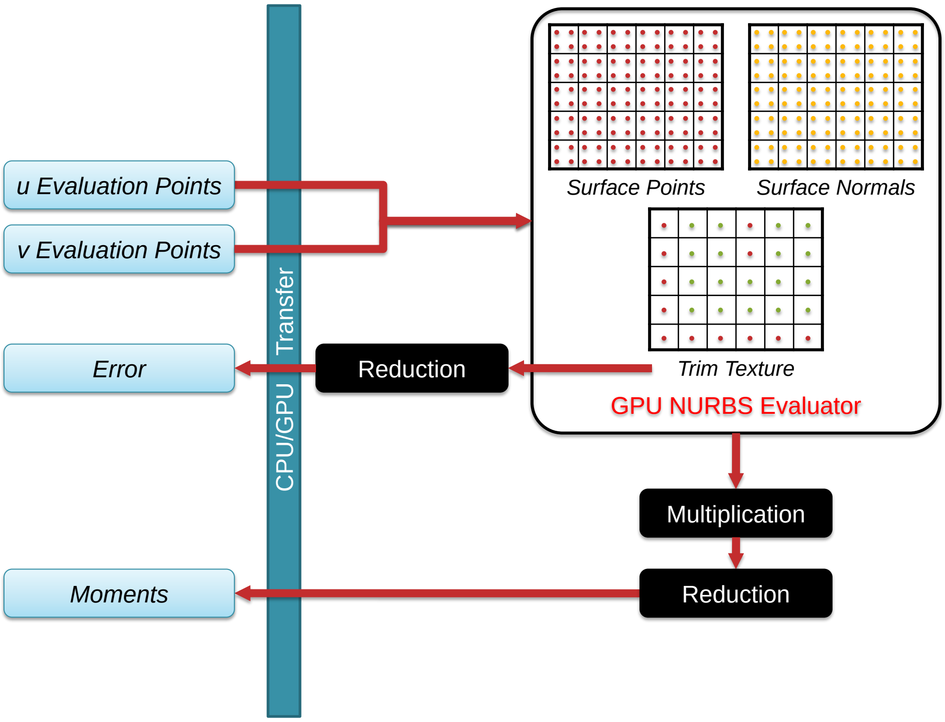


Figure 7.4: *Overview of our surface integration algorithm. The u and v evaluation parameters are chosen based on the number of sub-patches the NURBS surface is divided into and the type of integration scheme.*

Figure (7.4) gives an overview of our GPU surface integration algorithm. We divide each NURBS surface into sub-patches equal to the number of knot-intervals in each parametric direction; we call this the base number of sub-patches. Based on the number of sub-patches and the type of integration scheme used, we create a vector of u and v parametric positions where we want to evaluate the surface and normals. If it is a trimmed NURBS surface, we also generate a trim-texture (Section (4.6)) based on the number of sub-patches. We then compute the moment contribution from each patch by multiplying the corresponding functions for moments with the integration weights. Finally, we compute the sum of all the sub-patch moment contributions to get the moment contribution of the surface.

7.5.1 GPU Implementation

Once we have evaluated the surface coordinates and normals at the integration points using our GPU NURBS evaluator, we compute the moment contribution of each surface sub-patch in parallel using the GPU. We compute four moment values simultaneously, since GPUs are optimized for simultaneously computing values using four RGBA channels. For example, we compute the volume and the three first moments simultaneously in a single pass. We compute the moment contribution for each sub-patch by multiplying the moment functions with the integration weights. The moment functions are polynomial functions of the surface coordinates that correspond to the moment being computed; they are given by the corresponding equations in Section (7.4.3). The integration weights are based on the type of integration used (Section (7.6)). Once we have computed the individual contributions of each the sub-patches, we sum the values to get the surface moments by using GPU reduction.

We make use of Cg shader programs to implement the GPU operations to perform the moment computations and reductions. This is because we found in our preliminary testing that our shader programs provide better performance for NURBS evaluations compared to our optimized CUDA implementation of the same algorithm(Section (4.8.1)). In addition, making use of shader programs to perform these operations makes our implementation cross-platform and backwards-compatible; our implementation can run on both NVIDIA and AMD GPUs.

7.6 Numerical Surface Integration of NURBS

We compute the surface integrals of NURBS surfaces numerically using the Gaussian quadrature rule. In numerical analysis, a quadrature rule approximates the definite integral of a function using a weighted sum of function values at specified points within the domain of integration. An n-point Gaussian quadrature rule yields an exact result for the integration of polynomials up to degree $2n-1$ with suitable choice of points t_i and weights w_i (Equation (7.25)). The domain of integration for such a rule is conventionally taken as $[-1,1]$; however, the domain can be easily changed to any value by using a dummy variable for integration (Equation (7.26)).

$$\int_{-1}^{1} f(t)\,dt \approx \sum_{i=1}^{n} w_i f(t_i) \tag{7.25}$$

$$\int_{a}^{b} f(t)\,dt \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}t_i + \frac{b+a}{2}\right) \tag{7.26}$$

We can extend the quadrature rules to compute 2-dimensional integrals by having two weights; one for each direction. The integration rule can then be modified as given by Equation (7.27). As in the 1-dimensional case, the domain of integration is $[-1,1]\times[-1,1]$; this can be converted to any rectangular domain by changing the integration variables.

$$\int_{-1}^{1} f(t)\,dt \approx \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j f(t_{ij}) \tag{7.27}$$

In the following sections, we provide details of the weights and evaluation points for performing 1-point, 2-point, and 3-point Gaussian quadrature integration of NURBS surfaces.

7.6.1 1-point Gaussian Quadrature Scheme

The 1-point quadrature (mid-point scheme) is the simplest of the numerical integration schemes. It approximates the integral value with a single point that is evaluated at the center of the integration domain. The standard weight used for the point is 4.0 for the 2-dimensional case (Equation (7.28)). This being the simplest rule for integration, it can only integrate accurately up to degree 1 or linear polynomials.

$$\int_{-1}^{1} f(t)\,dt \approx 4\,f(0,0) \tag{7.28}$$

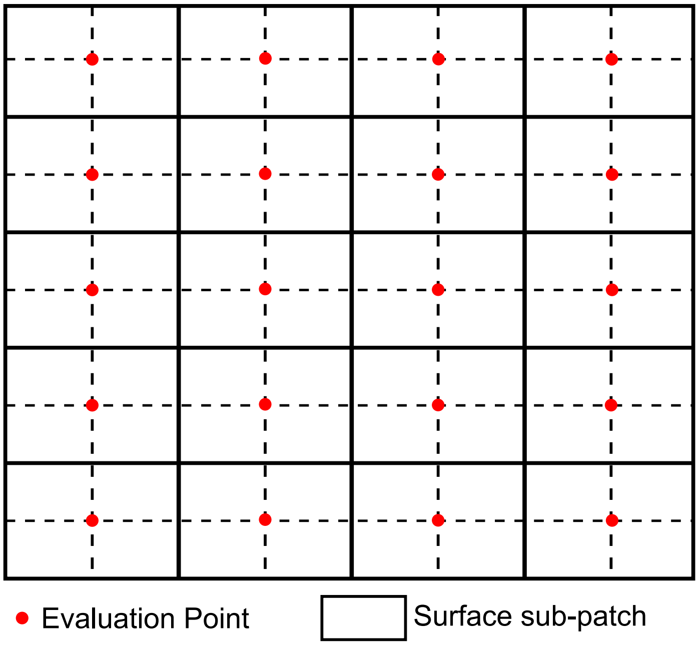


Figure 7.5: *Location of evaluation points in the parametric domain for computing the surface integrals using the mid-point integration scheme.*

Figure (7.5) shows how we can use the mid-point scheme to integrate over the surface of a NURBS patch. For integrating over the surface of a NURBS patch, we divide the parametric domain into sub-patches along the u and v parametric directions. For each sub-patch, we evaluate the function at its midpoint; we perform the required change of variable implicitly. The mid-point scheme is not accurate enough in computing the surface integrals in many practical cases; however, it can be used to get a rough approximation for the solution since it is easy to evaluate. Another important advantage is that the mid-point scheme can be implemented on the GPU using uniform grid spacing; the evaluation points are spaced uniformly along both the u and v directions in the parametric domain.

7.6.2 2-point Gaussian Quadrature Scheme

Solid models that are created using popular CAD systems such as SolidWorks are usually composed NURBS surfaces that are bi-cubic. In order to compute the volume accurately for such solid models, we need to use the 2-point quadrature scheme. Implementing the 2-point quadrature scheme is slightly more involved than the mid-point scheme since the spacing between the evaluation points is not uniform. However, the weights used are constant and they are equal to 1.0; the sum of the weights of the four evaluation points is 4.0. The evaluation points in the normalized $[-1,1]$ domain are $-1/\sqrt{3}$ and $1/\sqrt{3}$. Equation (7.29) expands the integration terms for the 2-dimensional case.

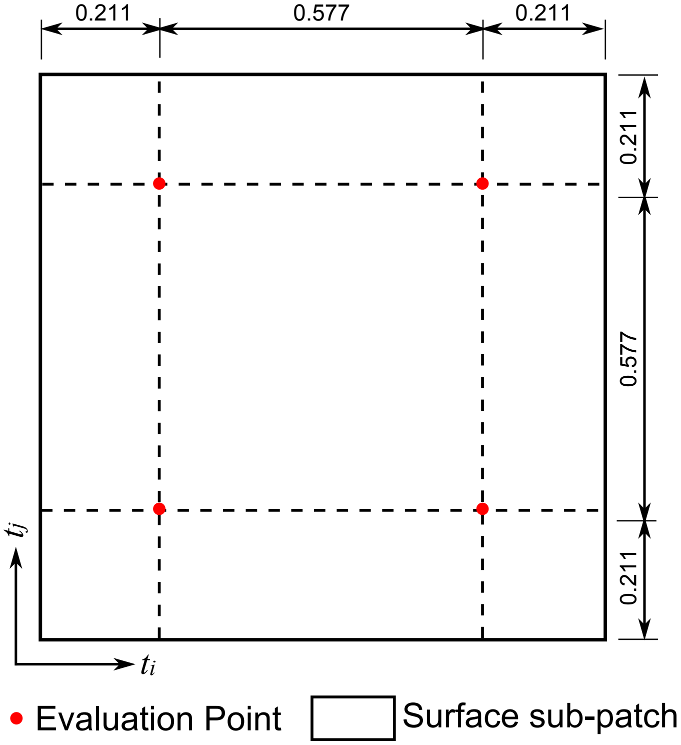


Figure 7.6: *Distribution of evaluation points in the 2-point Gaussian quadrature integration scheme. All the four integration points have a uniform weight of* 1.0.

$$\begin{aligned}\int_{-1}^{1} f(t)\,dt \approx\; & f\left(\frac{-1}{\sqrt{3}},\frac{-1}{\sqrt{3}}\right)+f\left(\frac{-1}{\sqrt{3}},\frac{1}{\sqrt{3}}\right)\\ & +f\left(\frac{1}{\sqrt{3}},\frac{-1}{\sqrt{3}}\right)+f\left(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}}\right)\end{aligned} \tag{7.29}$$

Figure (7.6) gives the fraction of the intervals between the evaluation points in a single sub-patch of the NURBS surface. The evaluation points are positioned symmetrically with respect to both the u and v directions inside the sub-patch. Figure (7.7) gives an example of the position of the evaluation points in a complete NURBS surface. In this example, the NURBS surface is divided into 4×5 sub-patches in the u and v directions respectively. The marked points' parametric positions are sent as a vector to the GPU evaluator and are then used to evaluate the surface coordinates and normals.

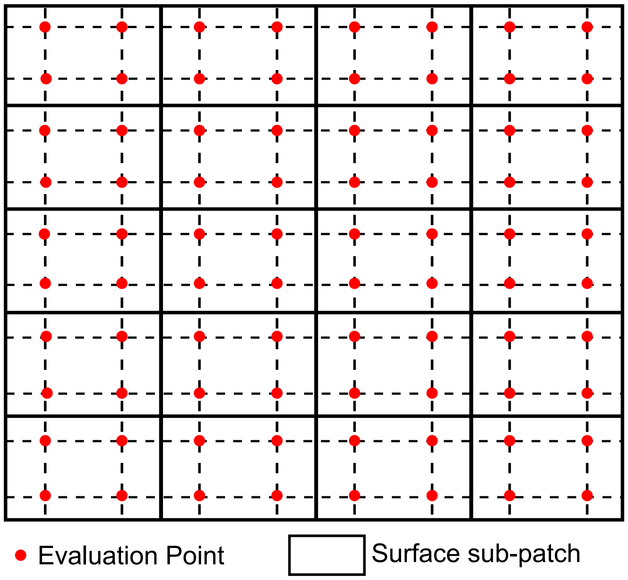


Figure 7.7: *Example of the evaluation points' location in the parametric domain for computing the surface integrals using the 2-point Gaussian quadrature integration scheme.*

7.6.3 3-point Gaussian Quadrature Scheme

Computing the higher order moments accurately in a solid model consisting of bi-cubic NURBS patches requires the 3-point quadrature scheme. Computing using the 3-point scheme requires the calculation of different weights for the different evaluation points. The parametric positions, t_is, of the evaluation points and their corresponding weights, w_is, are given by Equation (7.31). The definite integral in the canonical domain is approximated by the double-sum of the product of the weights and the integrand evaluated at the corresponding points. It can be noted that there are nine evaluation points in the 2-dimensional case; the sum of the product of the weights ($\sum w_i w_j$) is 4.0.

$$\int_{-1}^{1} f(t)\,dt \approx \sum_{i=1}^{3} \sum_{j=1}^{3} w_i w_j f(t_i, t_j) \tag{7.30}$$

$$
\begin{aligned}
t_1 &= -\sqrt{\left(\frac{3}{5}\right)},\ w_1 = \frac{5}{9}\\
t_2 &= 0,\ w_2 = \frac{8}{9}\\
t_3 &= \sqrt{\left(\frac{3}{5}\right)},\ w_3 = \frac{5}{9}
\end{aligned}
\tag{7.31}
$$

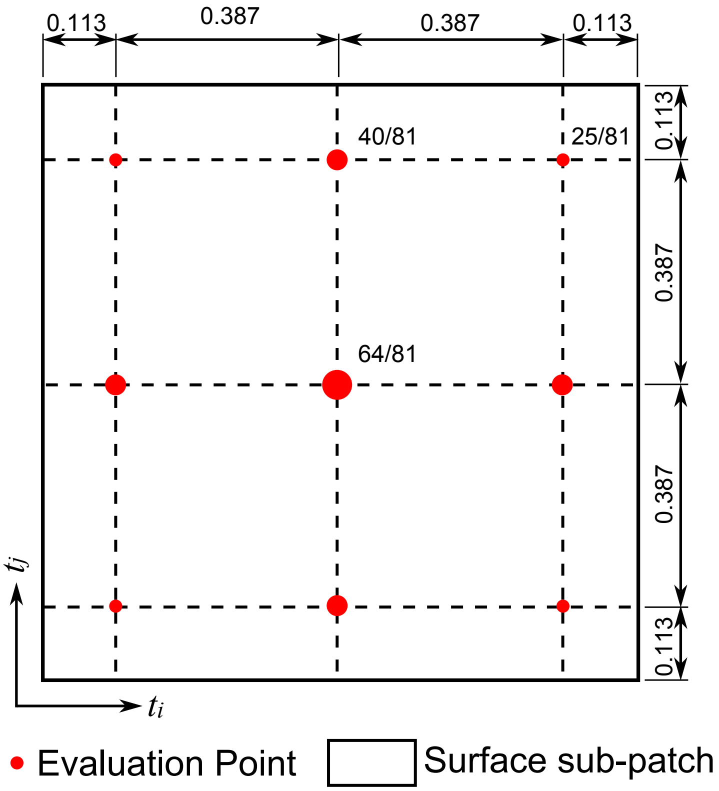


Figure 7.8: *Distribution of evaluation points in the 3-point Gaussian quadrature integration scheme. The size of the integration point indicates its relative weight; the weights are also indicated next to the points. The sum of all the weights equals* 4.0.

Figure (7.8) gives the location of the 9 evaluation points as a fraction of the sub-patch size. The size of the evaluation point indicates its relative weight. The center point has the maximum weight followed by the 4 points near the edge midpoints; the 4 corner points have the least weight.

7.6.4 Surface Integrals of Trimmed NURBS

We compute the surface integrals of trimmed NURBS in two stages. In the first stage, we treat them as un-trimmed surfaces and compute the moment contribution of each sub-patch using the methods explained in Section (7.6). We classify the sub-patches into three different cases: inside, outside or on the boundary of the trim curves. If the sub-patch lies inside a trimmed region, its contribution to the total surface integral is taken to be 0. Similarly, if the sub-patch lies outside the trimmed region, its contribution to the surface integral is taken as 1. The contribution for all the sub-patches that lie on the boundary of the trim-curves is weighted by the fraction of the sub-patch that lies outside the trimmed region.

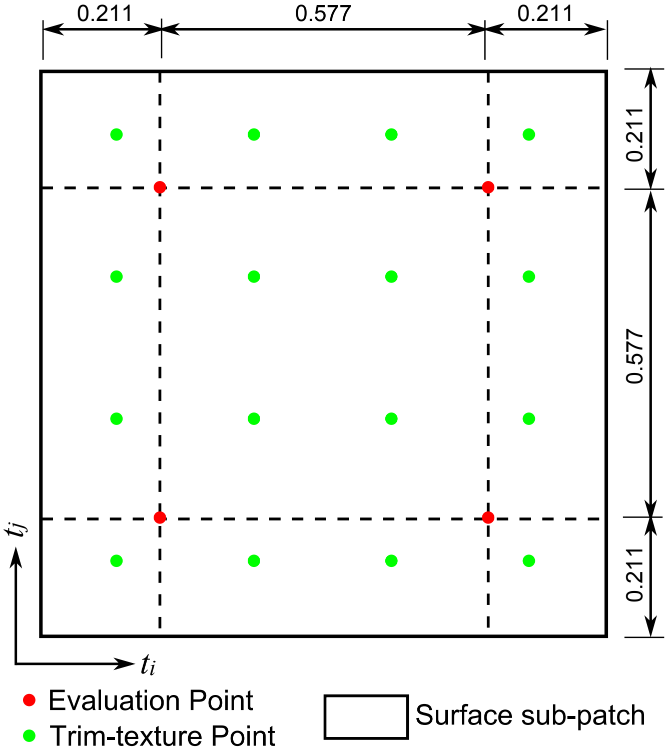


Figure 7.9: *Example of the distribution of the trim-texture points for a case with 4 points in each parametric direction. The 2-point Gaussian quadrature integration scheme evaluation points are also shown for comparison.*

In order to perform the weighting operation, we generate a trim-texture on the GPU by using the method we described in Section (4.6). Recall that this binary texture has the value 0 in the trimmed regions (and only the trimmed regions) of the surface in the parametric domain. In order to obtain an accurate value for the moments, we set the height and width of the trim texture to be three or four times the number of sub-patches that will be evaluated in that dimension. We find the fraction of the number of points that lie outside the trimmed region in a given sub-patch and multiply the moment contribution of the sub-patch with this fraction. This is similar to oversampling used in computer graphics for obtaining sub-pixel accuracy while rendering. Figure (7.9) shows an example where the trim texture is evaluated 4 times more densely than the number of sub-patches in each of the u and v parametric directions. This leads to a total of 16 positions inside the sub-patch where the trim is evaluated. The moment contribution of the patch is multiplied with the fraction of the number of points that lie outside the trimmed region.

7.7 Error Analysis

In this section, we derive estimates for the error in computing the volume of a solid body; this analysis can be extended in a similar manner to compute the error in higher-order moments. Computing the errors in a Gaussian integration scheme directly is difficult since it involves computing the $2n$-order derivative for an n-point integration scheme. A known simpler method is to compute the error as the difference between two different Gaussian integration schemes. It can be shown that this is a good estimate for the error since the integration value converges to the theoretical value as the order of Gaussian integration is increased. However, the surface integrals have to be evaluated twice using different orders of integration in this method. In our case, we estimate the error in the integration by computing the bounds for the coordinates used for the integration; the bounds are compued from the maximum possible deviation of the surface from a linear approximation. Using this method, we compute the error in the volume of several practical CAD models in Section (7.8).

In our method, for each surface sub-patch, we find the second partial derivatives of the surface as explained in Section (2.4). We then compute the maximum deviation K for each surface sub-patch based on the maximum second partial derivative. This gives the maximum deviation in the coordinates used for the surface integration. By neglecting higher order error terms, we can show that the error in volume computation can be calculated approximately using Equation (7.32). Intuitively, this error estimate measures the maximum possible deviation of the volume from the polyhedral approximation of the NURBS surface. It can be shown from Figure (7.2) that this error estimate measures the volume between the two parallel planes that are located at a distance K from the linear approximation.

$$\begin{aligned} \Delta M_0 &\approx \sum_i \int_{P_i} |\Delta z n_z| \, du\, dv \\ &\leq \sum_i \int_{P_i} |2 K n_z| \, du\, dv \end{aligned} \tag{7.32}$$

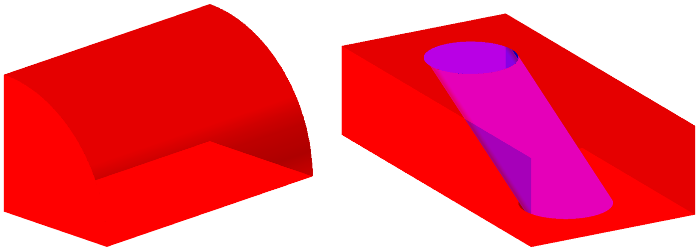
We calculate this error simultaneously while computing the moment values. For each sub-patch in the surface, we compute the value of K and store it in a separate texture. While computing the moments, we compute the error terms for each sub-patch. We then perform the multiplication and reduction operations just as for the moment computations explained in Section (7.5.1). Finally, after summing all the error contributions from each sub-patch, we get the total error for the surface. We perform the same operations on all the surfaces to get the total error in the volume of the solid object. In case of trimmed-NURBS surfaces, we consider only the error contributions from those sub-patches that lie outside the trimmed region.

7.8 Results

We timed our GPU-accelerated queries on a 2.40GHz CPU (dual core) running Windows XP with 3GB of RAM and an NVIDIA GeForce 9600M GT GPU with 256MB graphics memory. First, we discuss the accuracy of our algorithm by using it to compute moments of trimmed-NURBS solid models whose volume can be calculated theoretically. Next, we vary our integration schemes and the number of sub-patches used for the integration and check for the convergence of the solution. Finally, we compare the actual values of volume and center of mass of complex CAD models with the moment values obtained using ACIS; we also compare the time for the computations.

7.8.1 Accuracy of the Integration

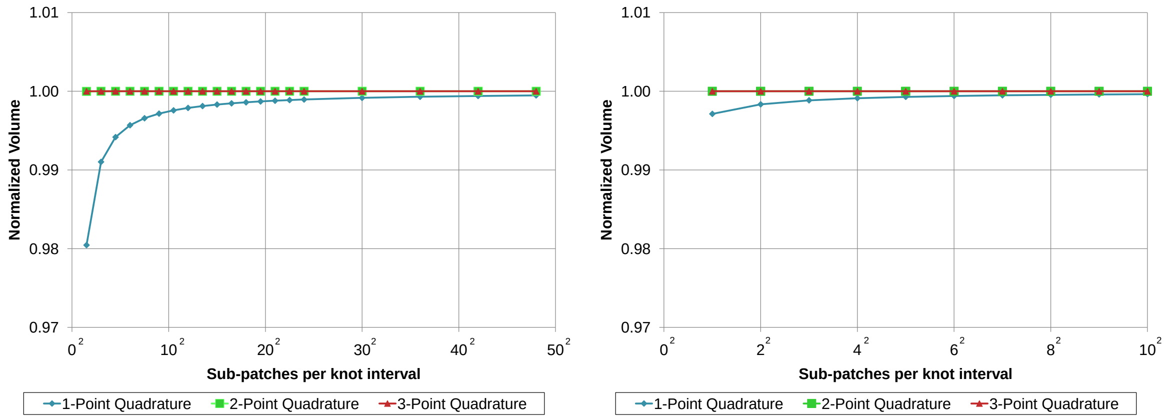
Since the exact value for the moments are very difficult to obtain for complex models, we tested the accuracy of our integration on a single NURBS patch of a quarter cylinder. We chose a cylinder since we can accurately compute its theoretical volume and center of mass for comparison, yet it is non-trivial for our GPU algorithm to evaluate since it is a rational surface. In addition, the cylinder was placed horizontally in the coordinate system to make sure the moment contributions from the cylindrical surfaces are not zero (Figure (7.10(a))). As expected, the higher order quadrature schemes provide a more accurate answer, particularly with smaller numbers of sub-patches (Figure (7.11(a))).



(a) *Quarter Cylinder* **(b)** *Slanted Cylinder Block*

Figure 7.10: *Solid models used for measuring the accuracy of our moment computation algorithm. The model on the right consists of a slanted cylinder cut out from a rectangular parallelepiped. The models are rendered using transparency to show inside surfaces.*

In order to assess the accuracy of our algorithm with the presence of trimmed NURBS surfaces, we constructed a solid object consisting of a block with a slanted cylindrical section cut from it (Figure (7.10(b))). We can theoretically calculate the volume and center of mass of this object since the slanted cylinder is still a prism with a circular base. Figure (7.11(b)) shows the accuracy as a function of the number of patches per knot interval used for evaluating each NURBS surface. We set the number of sub-patches equal to the number of knot intervals in the respective u and v parametric direction of the NURBS surface as a first level of subdivision; the total base number of sub-patches for this model being 128.



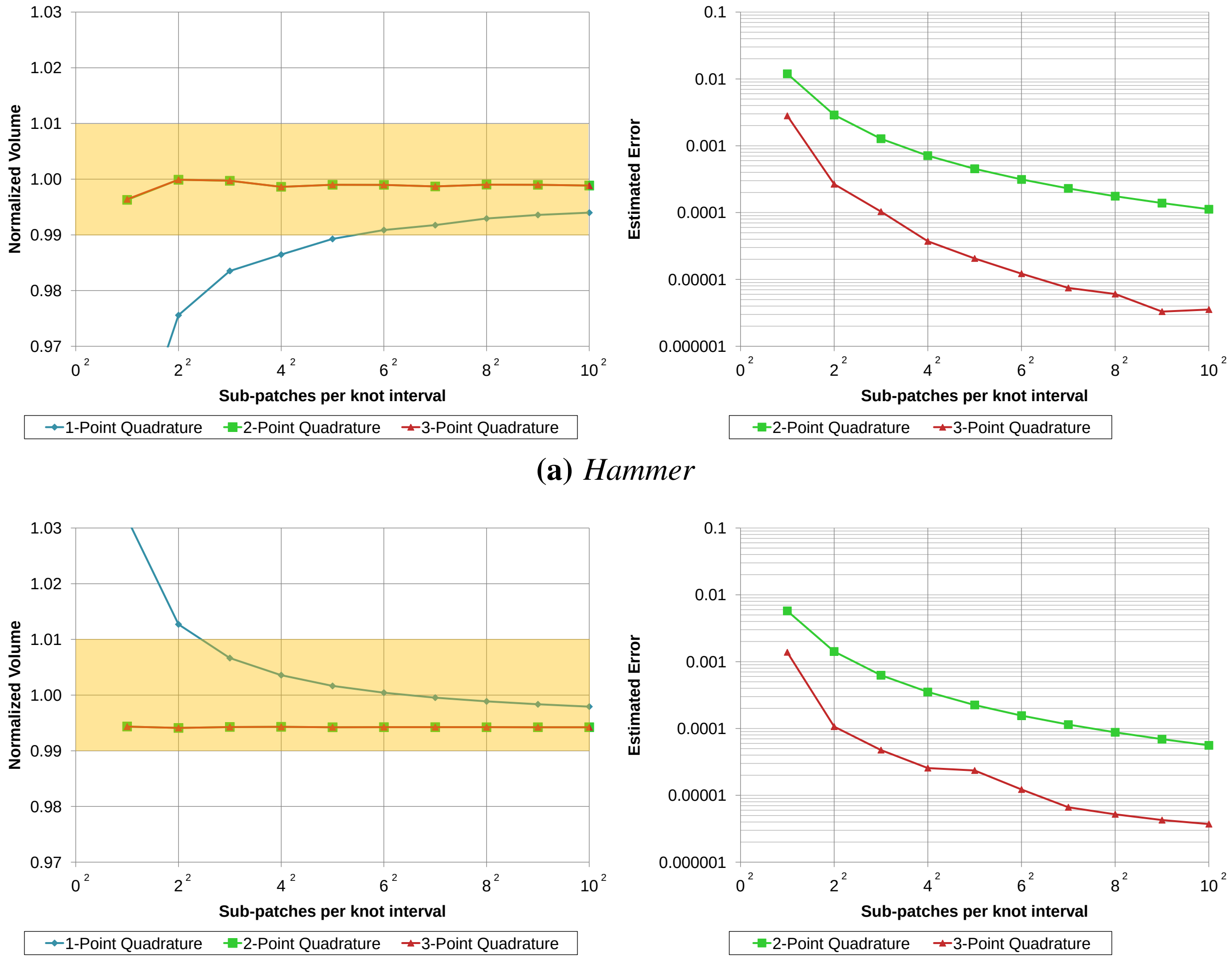
(a) *Quarter Cylinder*

(b) *Slanted Cylinder Block*

Figure 7.11: *Graphs showing the accuracy of our volume computation algorithm for two cases whose volume can be theoretically computed. The volume is normalized with respect to the theoretical volume. The x-axis for the graphs shows the number of sub-patches per knot interval used for integration of each NURBS surface. It can be noted that the 2 and 3-point quadrature schemes give exact results with very few sub-patches.*

7.8.2 Volume and Error Analysis of CAD Objects

In order to test the applicability of our algorithm to realistic solid models, we computed the volume of several CAD models with multiple trimmed NURBS surfaces. For comparison, we calculated the volume of the models using ACIS with accuracy 0.01 measured as a fraction of the computed moment. The ACIS algorithm does not have a bound on the precision of the mass properties because of hard-coded convergence criteria in their functions [Spatial Corporation, 2009b]. If the accuracy does not meet the requested value, they tighten the convergence criteria and repeat the calculation. However, if the mass properties remain unchanged, they are as close as can be achieved by their algorithm. In our algorithm, we estimate the error in the volume while using 2-point or 3-point quadrature integration as explained in Section (7.7).



(a) *Hammer*

(b) *Freeform*

Figure 7.12: *Graphs showing the convergence of our volume computation algorithm for the "Hammer" and "Freeform" models. The volume is shown as a fraction of the volume computed by ACIS. The x-axis shows the number of sub-patches per knot interval used for the integration, the base number of sub-patches being* 3859 *and* 4096 *for the "Hammer" and "Freeform" models respectively. The orange band indicates the ACIS error bounds. Note logarithmic scale in the y-axis of the error graphs.*

We computed the volume of these objects using 1-point, 2-point, and 3-point quadrature by varying the number of sub-patches that are used for the integration of each NURBS surface. As before, we set the base number of sub-patches equal to the number of knot intervals in the respective u and v parametric direction of the NURBS surface. Figure (7.12) shows the convergence in the volume computation for the respective solid models shown in Figure (7.1).

7.8.3 Moment Computation Results

We compared the results for evaluating the volume and center of mass of different CAD objects that were made of multiple trimmed-NURBS surfaces using both our GPU algorithm and ACIS. Table (7.1) gives the complexity of the objects based on the number of surfaces that make up each object. Please note that the "Scooby" model has the highest percentage of NURBS surfaces while the "Race Car" has the least percentage of NURBS surfaces. It can be seen that our algorithm is most effective when the object is made up of a large number of NURBS surfaces.

Object	**Total Surfaces**	**NURBS**	**Trimmed**
Scooby	157	116	89
Trefoil	4	4	0
Freeform	2	1	1
V8 Engine	931	87	36
Engine	580	16	12
Race Car	1076	49	42

Table 7.1: *Complexity of the different models showing the distribution of trimmed NURBS surfaces.*

Object	**GPU**			**ACIS**	
	Volume	**Estimated Volume Error Fraction**	**Time (s)**	**Volume**	**Time(s)**
Scooby	1.99516×10^{8}	6.248×10^{-2}	1.516	1.99601×10^{8}	15.547
Trefoil	8.57171×10^{6}	4.658×10^{-3}	0.063	8.57174×10^{6}	0.175
Freeform	9.79816×10^{5}	1.251×10^{-3}	0.015	9.82948×10^{5}	0.718
V8 Engine	9.97957×10^{6}	1.933×10^{-6}	1.188	9.99160×10^{6}	5.078
Engine	1.29932×10^{5}	1.000×10^{-5}	0.235	1.30284×10^{5}	0.922
Race Car	1.83031×10^{9}	5.534×10^{-5}	0.703	1.83799×10^{9}	4.140

Table 7.2: *Volumes computed by our GPU algorithm and ACIS for different CAD models. Our values were computed using the 2-point quadrature scheme with* 2^2 *sub-patches per knot interval for each surface. The errors were estimated using our method explained in Section (7.7). The ACIS error bound was set to be* 0.01

Table (7.2) summarizes the results of the volume computations. However, it should be noted that neither the ACIS mass properties function nor our GPU algorithm have been optimized for performance. In addition, the time taken by our GPU algorithm includes the time for evaluating all the NURBS surfaces, which takes the largest percentage of the total time ($\approx 90\%$). The values were computed using the 2-point quadrature scheme with two sub-patches per knot interval in both parametric directions. It can be noted that our GPU algorithm computes accurate moments with low estimated errors.

Table (7.3) gives the magnitude of the difference between the center of mass values computed by our GPU algorithm and ACIS as a fraction of the model size. The model size is computed as the largest dimension of the axis-aligned bounding-box that encloses the model.

Object	**Object Size**	**Difference Fraction**		
		x	y	z
Scooby	2037.02	5.32×10^{-5}	2.47×10^{-4}	7.40×10^{-5}
Trefoil	792.46	3.71×10^{-7}	3.87×10^{-7}	6.01×10^{-7}
Freeform	494.18	4.86×10^{-6}	1.36×10^{-4}	1.40×10^{-5}
V8 Engine	753.57	5.11×10^{-4}	5.29×10^{-4}	1.13×10^{-3}
Engine	158.14	4.23×10^{-5}	6.91×10^{-4}	5.44×10^{-4}
Race Car	5085.59	2.02×10^{-4}	1.77×10^{-5}	2.93×10^{-4}

Table 7.3: *Difference between the center of mass values computed by our GPU algorithm and ACIS for different CAD models expressed as a fraction of the model size. Our values were computed using the 2-point quadrature scheme with* 2^2 *sub-patches per knot interval for each surface.*

An advantage of our GPU algorithm is that the updates to the volume and center of mass due to changes in a single surface can be performed interactively. For example, the GPU algorithm computes the volume and the center of mass for the "Freeform" model in less than 0.02s. This means that even though the initial moment computation for complex models takes more than a second, the moment values can be updated up to 50 times per second while interactively editing the solid model.

7.9 Conclusions

We have developed a framework that uses GPUs to accelerate moment computations. Our moment computations can be performed interactively, while the model is being edited. Our algorithms have error estimates and they are based on object-space resolution instead of just image-space resolution. They make use of actual surface data and not just the tessellation, which make them independent of tessellation errors. We also show significant performance and accuracy improvements over an existing commercial CPU-based system.

Our GPU-based surface integration algorithms can be extended for use in analysis tools such as FEA. Accurate and fast surface integration will aid in interactive analysis of complex objects that will provide functional feedback to the designer. Such functional feedback will reduce the design lead time of a component, ultimately resulting in significant cost savings.

Chapter 8

Conclusions and Future Work

General purpose GPU computing has grown tremendously in the last few years. GPUs are currently being used to accelerate computations in a variety of fields ranging from advanced simulations to financial analysis. We have contributed to this nascent field by developing new GPU algorithms for mechanical CAD. Our algorithms improve the performance of fundamental CAD operations, thereby enabling interactive feedback to the designer.

Many of the challenges that we have addressed in our algorithms for CAD are frequently encountered in GPU programming. Some of our algorithms can be directly extended to similar problems in design and analysis such as collision detection and simulations.

In Section (8.1), we outline some direct extensions of our work. Our main contributions are summarized in Section (8.2).

8.1 Future Research Directions

8.1.1 Collision Detection

Our surface-surface intersection algorithm makes use of hierarchical bounding-box structure on the GPU. This data-structure can be used for detecting collisions efficiently. Our algorithm can be easily extended to develop a fast static collision detection system. Our implementation employs an efficient method to map the hierarchical structure to the GPU memory architecture that provides optimum performance. This mapping can be extended to perform collision detection at each level of the hierarchy, where the collision tests could be performed efficiently using GPU kernels.

Another application of our minimum distance computations is continuous collision detection. We can use the minimum distance and its direction vector to compute the maximum distance an object can be moved safely without collisions in an interactive system. The speed of our computations will enable such a system to be interactive even with complex CAD objects. Such continuous collision detection systems are invaluable to designers while simulating the product assembly process.

8.1.2 Optimization

Another related field where our algorithms could be directly used is in the field of optimization. Optimization methods usually make use of higher dimensional surfaces that correspond to the constraints of the optimization. These constraint surfaces are often approximated with polynomial spline surfaces. Our NURBS evaluator could be used to evaluate such arbitrary degree spline surfaces. In addition, our evaluation algorithm could be easily extended to higher dimensions by using multiple textures and render targets. Using our evaluator provides a method to sample large parts of the constraint surface in a fast manner.

Another related problem in optimization is finding the closest point on the constraint surface given any point that is in the feasible region of the configuration space. Our minimum distance computations could be extended to find the closest point to a surface in higher dimensions. This is especially useful in algorithms that need to find the closest point on a constraint surface in higher dimensions.

8.1.3 Design Analysis

Our algorithms could be used as building blocks of many design analysis methods. One such method that could benefit tremendously from GPU acceleration is Finite Element Analysis (FEA), which is used to analyze a component and provide functional feedback to the designer about its behavior when subject to forces. Such functional feedback will reduce the design lead-time of a component, ultimately resulting in significant cost savings.

NURBS are recently being used as interpolation functions for FEA. This will not only unify modeling and analysis with NURBS as a standard, but also reduce the approximation errors due to discretization of the analysis domain. In addition, since CAD systems represent solid models using NURBS surfaces by default, it will also be simpler to integrate the FEA directly with the CAD system. Such methods, called isogeometric analysis, need a fast NURBS evaluator to interpolate higher dimensional element basis functions. Our NURBS evaluator could be directly used as a fast interpolator for such higher dimensional surfaces.

Our GPU-based surface integration algorithms could also be extended for use in FEA. Volume integrals in FEA are usually converted to surface integrals using divergence theorem. These surface integrals then need to be evaluated, taking into account the different boundary conditions. Our surface integration algorithms could be adapted to calculate these integrals accurately.

8.2 Contributions

We have developed a GPU method for evaluating arbitrary degree NURBS surfaces with an arbitrary number of control points and knots with the same unified fragment program. Our method uses the GPU to evaluate a grid of points on the NURBS surface that can be directly used for rendering as well as for further modeling operations. Our algorithms are backward compatible, which make use of standard OpenGL extensions or features that are available even on older cards, while still taking advantage of the improved performance on newer cards. We have also developed a rendering method to display trimmed-NURBS surfaces by interpreting the points already evaluated as vertices. The rendering algorithm is capable of dynamic continuous LOD based on the size and location of the surface with respect to the viewpoint. We found our method to be capable of interactively evaluating and rendering up to 300 NURBS surfaces. For interactive display of a large number of trimmed NURBS surfaces, we have demonstrated that GPU-based evaluation of the exact surfaces is a viable option.

Extending our evaluator, we have developed GPU algorithms to perform modeling operations such as inverse evaluation, ray intersection, and surface-surface intersection. We have developed an efficient algorithm to perform inverse evaluation of NURBS surfaces on the GPU. This algorithm finds the parametric (u,v) coordinate given any (x,y,z) coordinate on the NURBS surface within an arbitrary user-defined tolerance. Combining this with our trimmed NURBS rendering, we have developed a novel method to interactively trim and sketch on a NURBS surface in real time. This is possible because our fast inverse evaluation algorithm enables us to sketch in the model space, not just in the parametric space, with the correspondence tracked simultaneously. Our NURBS surface-surface intersection algorithm is fast and robust in finding the intersection curve within user-specified tolerances. The intersection curve, like the sketch curve above, is simultaneously output in the model space as well as in the parametric spaces of the two NURBS surfaces. We have also extended the surface-surface intersection algorithm to evaluate self-intersections in NURBS surfaces. This algorithm can be used to detect self-intersections and output the intersection curves if the surface is self-intersecting.

We have also developed an hybrid CPU/GPU framework to accelerate minimum distance computations that can be used to find the minimum distance to a surface given any point in space. We have extended it to a fast algorithm that computes the minimum distance between two surfaces or between two solid models represented by B-reps, using bounding-box hierarchies on the GPU. Our algorithm is orders of magnitude faster and more accurate than the commercial solid modeling kernel ACIS in calculating these distances. These algorithms make use of our unified framework that uses the GPU as a co-processor to improve the performance of algorithms used for solving geometric queries. This framework can be extended to accelerate several related queries that are based on properties such as normals or curvature of the underlying shapes . We also provide theoretical guarantees for all of our geometric computations. They allow for user-defined tolerance values that are essential for integrating our algorithms in a CAD system.

Finally, we have developed GPU algorithms to accelerate moment computations. We can calculate the volume, center of mass, moment of inertia, etc. of solid models that are represented using multiple trimmed NURBS surfaces. Our computations are independent of the tessellation of the solid models. They allow for interactive updates of moments while the model is being edited. Our moment computation algorithms make use of our GPU implementation of numerical surface integration on trimmed NURBS surfaces. We have implemented midpoint, 2-point Gaussian quadrature, and 3-point Gaussian quadrature integration schemes on the GPU. We compare the accuracy of our algorithm to theoretical results. Our volume computations have error estimates that allow for user-defined tolerance values. Our volume computation algorithms are both faster and more accurate than ACIS for many complex CAD models.

8.3 Concluding Remarks

We have developed different GPU algorithms and tools that can be used to improve the performance of CAD systems. We envision a CAD system that integrates design and analysis tools and can take advantage of parallel hardware such as GPUs to accelerate computations in an interactive framework. The resulting performance improvements and the ability to get real-time feedback can completely transform the design experience, enabling the designer to explore the design space more effectively. However, development of such systems requires a paradigm shift in thinking about CAD systems; traditional algorithms have to be rebuilt with interactivity as a fundamental requirement. We hope that our tools will form an integral part of the future research in this field.

Bibliography

[Abi-Ezzi and Wozny, 1990] S. S. Abi-Ezzi and M. J. Wozny. Factoring a homogeneous transformation for a more efficient graphics pipeline. *Computer Graphics Forum*, 9(3):245–255, 1990.

[Agarwal *et al.*, 2003] Pankaj Agarwal, Shankar Krishnan, Nabil Mustafa, and Suresh Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *11th European Symposium on Algorithms*, 2003.

[Barnhill and Kersey, 1990] R. E. Barnhill and S. N. Kersey. A marching method for parametric surface surface intersection. *Computer Aided Geometric Design*, 7(1-4):257–280, 1990.

[Blelloch, 1990] Guy E. Blelloch, editor. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Bolz and Schröder, 2002] Jeffrey Bolz and Peter Schröder. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Web3D 2002*, pages 11–17, 2002.

[Briseid *et al.*, 2006] Sverre Briseid, Tor Dokken, Trond Runar Hagen, and Jens Olav Nygaard. *Computational Science - Lecture Notes in Computer Science*, volume 3994/2006, chapter Spline Surface Intersections Optimized for GPUs, pages 204–211. Springer, 2006.

[Carr *et al.*, 2006] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, pages 203–209, 2006.

[Cattani and Paoluzzi, 1990] C. Cattani and A. Paoluzzi. Boundary integration over linear polyhedra. *Computer Aided Design*, 22(2):130–135, 1990.

[Chen *et al.*, 2008] Xiao-Diao Chen, Jun-Hai Yong, Guozhao Wang, Jean-Claude Paul, and Gang Xu. Computing the minimum distance between a point and a NURBS curve. *Computer-Aided Design*, 40(10-11):1051 – 1054, 2008.

[Corney and Lim, 2001] Jonathan Corney and Theodore Lim. *3D Modeling with ACIS*. Saxe-Coburg, 2001.

[Dokken *et al.*, 2005] Tor Dokken, Vibeke Skytt, Trond Runar Hagen, and Jens Olav Nygaard. Us patent 20080259078 - apparatus and method for determining intersections. US Patent Application: 20080259078, 2005.

[Edelsbrunner, 1985] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *Journal of Algorithms*, 6(2):213–224, 1985.

[Elber and Kim, 2001] Gershon Elber and Myung-Soo Kim. Geometric constraint solver using multivariate rational spline functions. In *SMA 2001: Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, pages 1–10. ACM, 2001.

[Fernando and Kilgard, 2003] Randimo Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, Boston, 2003.

[Filip *et al.*, 1987] Daniel Filip, Robert Magedson, and Robert Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1987.

[Gilbert *et al.*, 1988] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[Gonzalez-Ochoa *et al.*, 1998] Carlos Gonzalez-Ochoa, Scott McCammon, and Jörg Peters. Computing moments of objects enclosed by piecewise polynomial surfaces. *ACM Transactions on Graphics*, 17(3):143–157, 1998.

[Gottschalk *et al.*, 1996] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *ACM SIGGRAPH*, pages 171–180. ACM, 1996.

[Govindaraju *et al.*, 2003] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 25–32. Eurographics Association, 2003.

[Greß *et al.*, 2006] A. Greß, M. Guthe, and R. Klein. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*, 25(3):497–506, 2006.

[Guthe *et al.*, 2005] Michael Guthe, Ákos Balázs, and Reinhard Klein. GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005.

[Guthe *et al.*, 2006] Michael Guthe, Ákos Balázs, and Reinhard Klein. GPU-based appearance preserving trimmed NURBS rendering. *Journal of WSCG*, 14, 2006.

[Haller, 2006] Kirk Haller. Personal communication, 2006.

[Henshaw, 2002] W. D. Henshaw. An algorithm for projecting points onto a patched CAD model. *Engineering with Computers*, 18(3):265–273, 2002.

[Hoff *et al.*, 2001] Kenneth E. Hoff, Andrew Zaferakis, Ming Lin, and Dinesh Manocha. Fast and simple 2D geometric proximity queries using graphics hardware. In *I3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 145–148. ACM, 2001.

[Hoffmann, 1989] Christoph M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann Publishers Inc., 1989.

[Horn, 2005] Daniel Horn. *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Stream Reduction Operations for GPGPU Applications. Addison-Wesley, 2005.

[Johnson and Cohen, 1998] D.E. Johnson and Elaine Cohen. A framework for efficient minimum distance computations. *IEEE International Conference on Robotics and Automation*, 4:3678–3684, 1998.

[Jorabchi *et al.*, 2009] Kavous Jorabchi, Joshua Danczyk, and Krishnan Suresh. Efficient and automated analysis of potentially slender structures. *Journal of Computing and Information Science in Engineering*, 9(4), 2009.

[Kahlesz *et al.*, 2002] Ferenc Kahlesz, Ákos Balázs, and Reinhard Klein. Multiresolution rendering by sewing trimmed NURBS surfaces. In *SMA '02: ACM Symposium on Solid Modeling and Applications*, pages 281–288, 2002.

[Kanai, 2007] Takashi Kanai. Fragment-based evaluation of Non-Uniform B-spline surfaces on GPUs. *Computer-Aided Design and Applications*, 4(3):287–294, 2007.

[Khardekar and McMains, 2006] Rahul Khardekar and Sara McMains. Fast layered manufacturing support volume computation on GPUs. In *Proceedings of the ASME Design Engineering Technical Conferences*. ASME, 2006.

[Khardekar, 2008] Rahul Khardekar. *Real-time manufacturability feedback*. PhD thesis, University of California, Berkeley, Mechanical Engineering Department, 2008.

[Kilgariff and Fernando, 2005] Emmett Kilgariff and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter The GeForce 6 Series GPU Architecture, pages 471–491. Addison-Wesley, 2005.

[Kim *et al.*, 2006] Jinwook Kim, Soojae Kim, Heedong Ko, and Demetri Terzopoulos. Fast GPU computation of the mass properties of a general shape and its application to buoyancy simulation. *Visual Computer*, 22(9):856–864, 2006.

[Kipfer *et al.*, 2004] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: a GPU-based particle engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 115–122. ACM, 2004.

[Kobbelt, 1997] Leif Kobbelt. Robust and efficient evaluation of functionals on parametric surfaces. In *SCG '97: Proceedings of the thirteenth annual symposium on computational geometry*, pages 376–378. ACM, 1997.

[Kolb *et al.*, 2004] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 123–131. ACM, 2004.

[Kriezis *et al.*, 1990] G. A. Kriezis, P. V. Prakash, and N. M. Patrikalakis. A method for intersecting algebraic surfaces with rational polynomial patches. *Computer Aided Design*, 22(10):645–654, 1990.

[Krishnamurthy *et al.*, 2007] Adarsh Krishnamurthy, Rahul Khardekar, and Sara McMains. Direct evaluation of NURBS curves and surfaces on the GPU. In *ACM Symposium on Solid and Physical Modeling*, pages 329–334. ACM, 2007.

[Krishnan and Manocha, 1997] Shankar Krishnan and Dinesh Manocha. An efficient surface intersection algorithm based on lower-dimensional formulation. *ACM Transactions on Graphics*, 16(1):74–106, 1997.

[Krishnan *et al.*, 1998] S. Krishnan, M. Gopi, M. Lin, D. Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using shelltrees. *Computer Graphics Forum*, 17(3):315–326, 1998.

[Kumar and Manocha, 1995] Subodh Kumar and Dinesh Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design*, 27(7):509–521, 1995.

[Kumar *et al.*, 1996] Subodh Kumar, Dinesh Manocha, and Anselmo Lastra. Interactive display of large NURBS models. *IEEE Transactions on Visualization and Computer Graphics*, 2(4):323–336, 1996.

[Larsen *et al.*, 2000] E. Larsen, S. Gottschalk, Ming Lin, and Dinesh Manocha. Fast distance queries with rectangular swept sphere volumes. *Proceedings of ICRA '00: IEEE International Conference on Robotics and Automation*, 4:3719–3726, 2000.

[Lauterbach *et al.*, 2009] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. In *Proceedings of Eurographics 2009*. Eurographics Association, 2009.

[Lauterbach *et al.*, 2010] Christian Lauterbach, Qi Mo, and Dinesh Manocha. gProximity: Hierarchical GPU-based operations for collision and distance queries. In *Proceedings of Eurographics 2010*, page To Appear, 2010.

[Lee and Requicha, 1982a] Yong Tsui Lee and Aristides A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. I. Known methods and open issues. *Communications of the ACM*, 25(9):635–641, 1982.

[Lee and Requicha, 1982b] Yong Tsui Lee and Aristides A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. II. A family of algorithms based on representation conversion and cellular approximation. *Communications of the ACM*, 25(9):642–650, 1982.

[Loop and Blinn, 2005] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005*, pages 1000–1009. ACM, 2005.

[Loop and Blinn, 2006] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics*, 25(3):664–670, 2006.

[Mark *et al.*, 2003] W. R Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[Martin *et al.*, 2000] William Martin, Elaine Cohen, Russell Fish, and Peter Shirley. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools*, 5(1):27–52, 2000.

[Mattson *et al.*, 2004] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[Messner and Taylor, 1980] A. M. Messner and G. Q. Taylor. Algorithm 550: Solid polyhedron measures [z]. *ACM Transactions on Mathematical Software*, 6(1):121–130, 1980.

[Mirtich, 1996] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2):31–50, 1996.

[Nelson *et al.*, 2005] Donald D. Nelson, David E. Johnson, and Elaine Cohen. Haptic rendering of surface-to-surface sculpted model interaction. In *SIGGRAPH '05: ACM SIGGRAPH Courses*. ACM, 2005. Course: Recent advances in haptic rendering and applications.

[Nishita *et al.*, 1990] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. In *ACM SIGGRAPH 90*, pages 337–345, 1990.

[Owens *et al.*, 2007] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[Pabst *et al.*, 2006] H.F. Pabst, J.P. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed NURBS surfaces on the GPU. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 151–160, 2006.

[Patrikalakis, 1993] Nicholas M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95, 1993.

[Peters and Nasri, 1997] Jöorg Peters and Ahmad Nasri. Computing volumes of solids enclosed by recursive subdivision surfaces. *Computer Graphics Forum*, 16:89–94, 1997.

[Pharr, 2005] Matt Pharr, editor. *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[Piegl and Tiller, 1997] Les A. Piegl and Wayne Tiller. *The NURBS Book*. Springer, second edition, 1997.

[Piegl, 1991] Les Piegl. On NURBS: a survey. *IEEE Computer Graphics Applications*, 11(1):55–71, 1991.

[Purcell *et al.*, 2002] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[Purcell *et al.*, 2003] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[Quinlan, 1994] Sean Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 3324–3329. IEEE, 1994.

[Requicha and Rossignac, 1992] Aristides A. G. Requicha and Jarek R. Rossignac. Solid modeling and beyond. *IEEE Computer Graphics Applications*, 12(5):31–44, 1992.

[Rockwood *et al.*, 1989] Alyn Rockwood, Kurt Heaton, and Tom Davis. Real-time rendering of trimmed surfaces. In *ACM SIGGRAPH 89*, pages 107–116, 1989.

[Rudin, 1976] Walter Rudin. *Principles of Mathematical Analysis*, chapter Integration of Differential Forms, pages 253–275. McGraw-Hill, 3 edition, 1976.

[Samad and Suresh, Accepted 2010] Wael Abdel Samad and Krishnan Suresh. CAD-integrated analysis of 3-D beams: A surface-integration approach. *Engineering with Computers*, Accepted 2010.

[Sederberg *et al.*, 1998] Thomas W. Sederberg, Jianmin Zheng, David Sewell, and Malcolm Sabin. Non-uniform recursive subdivision surfaces. In *Computer Graphics Proceedings, Annual Conference Series*, pages 387–94. ACM SIGGRAPH 98, July 1998.

[Sederberg *et al.*, 2003] Thomas W. Sederberg, Jianmin Zheng, Almaz Bakenov, and Ahmad Nasri. T-Splines and T-NURCCs. *ACM Transactions on Graphics*, 22(3):477–484, 2003.

[Sengupta *et al.*, 2007] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Symposium on Graphics Hardware*, pages 97–106. ACM, Eurographics Association, 2007.

[Shiue *et al.*, 2005] Le-Jeng Shiue, Ian Jones, and Jörg Peters. A real-time GPU subdivision kernel. *ACM Transactions on Graphics*, 24(3):1010–1015, 2005.

[Soldea *et al.*, 2002] Octavian Soldea, Gershon Elber, and Ehud Rivlin. Exact and efficient computation of moments of free-form surface and trivariate based geometry. *Computer-Aided Design*, 34(7):529–539, 2002.

[Spatial Corporation, 2009a] Spatial Corporation. *ACIS Geometric Modeler: User Guide*, 2009. api_check_face_clearance, Version 20.0.

[Spatial Corporation, 2009b] Spatial Corporation. *ACIS Geometric Modeler: User Guide*, 2009. api_body_mass_props, Version 20.0.

[Sud *et al.*, 2004] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum*, 23(10):557–566, 2004.

[Thompson and Cohen, 1999] T. Thompson and E. Cohen. Direct haptic rendering of complex trimmed NURBS models. In *8th Annual Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 1999.

[Timmer and Stern, 1980] H.G. Timmer and J.M. Stern. Computation of global geometric properties of solid objects. *Computer-Aided Design*, 12(6):301–304, 1980.

[Toth, 1985] Daniel L. Toth. On ray tracing parametric surfaces. In *ACM SIGGRAPH 85*, pages 171–179, 1985.

[Woo *et al.*, 2004] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL(R) Programming Guide, Version 1.4*, chapter Drawing Filled Concave Polygons Using the Stencil Buffer, pages 600–601. Addison-Wesley, fourth edition, 2004.