ETER: Elastic Tessellation for Real-Time Pixel-Accurate Rendering of Large-Scale NURBS Models

RUICHENG XIONG*, University of Science and Technology of China, China
YANG LU* and CONG CHEN, Sheyun Technology, China
JIAMING ZHU, YAJUN ZENG, and LIGANG LIU†, University of Science and Technology of China, China

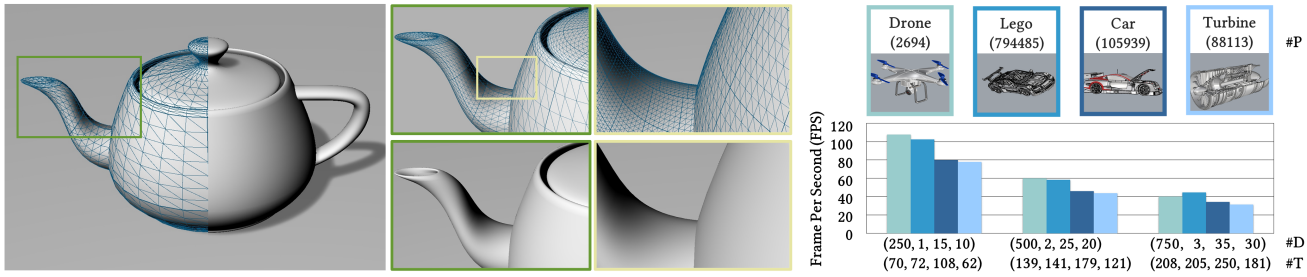


Fig. 1. Our elastic tessellation framework, ETER, enables the rendering of NURBS models with pixel accuracy while maintaining real-time frame rates. By utilizing on-the-fly computation of pixel-accurate tessellation factors and our novel crack-filling algorithm, we are able to achieve sufficient mesh density and high rendering quality without cracks at arbitrary zoom levels, as shown in the green and yellow boxes. The performance of our ETER is demonstrated in the histogram on the right, which shows the FPS of rendering according to various scales of four NURBS models (upper right, the numbers #P below the models indicate the numbers of the Bézier patches respectively). The horizontal axis represents the duplication count (#D) and the triangle count (#T, in millions) of the models respectively. The results show that ETER is able to render millions of Bézier patches with 0.25 billion triangles in real-time on an RTX 3090 Ti GPU.

We present ETER, an elastic tessellation framework for rendering large-scale NURBS models with pixel-accurate and crack-free quality at real-time frame rates. We propose a highly parallel adaptive tessellation algorithm to achieve pixel accuracy, measured by the screen space error between the exact surface and its triangulation. To resolve a bottleneck in NURBS rendering, we present a novel evaluation method based on uniform sampling grids and accelerated by GPU Tensor Cores. Compared to evaluation based on hardware tessellation, our method has achieved a significant speedup of 2.9 to 16.2 times depending on the degrees of the patches. We develop an efficient crack-filling algorithm based on conservative rasterization and visibility buffer to fill the tessellation-induced cracks while greatly reducing the jagged effect introduced by conservative rasterization. We integrate all our novel algorithms, implemented in CUDA, into a GPU NURBS rendering pipeline based on Mesh Shaders and hybrid software/hardware rasterization. Our performance data on a commodity GPU show that the rendering pipeline based on ETER is capable of rendering up to 3.7 million patches (0.25 billion tessellated triangles) in real-time (30FPS). With its advantages in performance, scalability, and visual quality in rendering large-scale NURBS models, a real-time tessellation solution based on ETER can be a powerful alternative or even a potential replacement for the existing pre-tessellation solution in CAD systems.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; *Rendering*.

Additional Key Words and Phrases: NURBS, adaptive tessellation, GPU-based algorithms, real-time rendering

ACM Reference Format:
Ruicheng Xiong, Yang Lu, Cong Chen, Jiaming Zhu, Yajun Zeng, and Ligang Liu. 2023. ETER: Elastic Tessellation for Real-Time Pixel-Accurate Rendering of Large-Scale NURBS Models. *ACM Trans. Graph.* 1, 1 (April 2023), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Non-uniform rational B-splines curves and surfaces (NURBS) are the industry standard for modeling primarily because they precisely represent both freeform shapes and analytical curves and surfaces [Rogers 2001]. Efficient rendering of large-scale NURBS Models without visual artifacts is highly desirable and regarded as a hard problem [Blomgren and Kasik 2002; Kumar et al. 1995].

Existing commercial CAD systems deal with this problem by pre-tessellating NURBS models and discretizing them into LOD (level-of-detail) triangle meshes in an offline manner. However, the pre-tessellation nature of those solutions leads to three defects. First, the resolution of the tessellated triangle meshes is fixed once generated, for which unpleasing faceted shading effects and rough polygonal silhouette might be perceived at certain viewing angles. Second, cracks between adjacent patches/LOD levels are difficult to deal with [Karis et al. 2021] and most likely will be revealed at

*Joint first authors
†The corresponding author

Authors' addresses: Ruicheng Xiong, University of Science and Technology of China, China, xiongruicheng@mail.ustc.edu.cn; Yang Lu, luyang@lyteflow.cn; Cong Chen, Sheyun Technology, China, chencong@lyteflow.cn; Jiaming Zhu, sanlayn@mail.ustc.edu.cn; Yajun Zeng, yajun_zeng@mail.ustc.edu.cn; Ligang Liu, University of Science and Technology of China, China, lgliu@ustc.edu.cn.

0730-0301/2023/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

close-up views if not handled well [Schollmeyer and Froehlich 2018]. Third, CAD users are obliged to configure a smaller tolerance value to limit object-space sampling error ([McNeel 2020; Systèmes 2023]) for favorable rendering quality, resulting in substantially increasing file size and memory usage, putting heavy burden on I/O bandwidth and affecting rendering performance eventually.

Instead of tessellating in a pre-processing step, we turn to pursue an on the fly adaptive tessellation framework for real-time rendering of large-scale NURBS models. However, this task is non-trivial. The challenges are three folds. First, it is challenging to develop a metric that accurately measures the sampling error and to utilize it in NURBS. Extensive studies have been conducted over the last decades, computing tessellation factor on-the-fly based on viewpoint and/or curvature of the surface [Buchenau and Guthe 2021; Guthe et al. 2005; Krishnamurthy et al. 2009; Schollmeyer and Froehlich 2018; Schwarz and Stamminger 2009]. Yeo et al. [2012] introduced pixel accuracy, a precise metric to measure screen-space error between the exact surface and its triangulation, making precise the criterion stated in Reyes [Cook et al. 1987]. Regretfully, their proposed implementation is not applicable to NURBS (for details see supplementary material). Still, we decided to adopt their pixel accuracy definition to explore our own framework, considering it can keep tessellation being conducted under pixel resolution at arbitrary zoom levels. Second, tessellation-induced crack is an issue that must be addressed, either by preventing cracks that might set restrictions on the framework or by designing crack detection and elimination algorithms if their occurrence is allowed. Third, performance is key to real-time rendering. Most recent works are based on hardware tessellation, which is unable to access some key GPU resources such as shared memory, making non-optimal performance an inevitable consequence.

Up to the present, no NURBS rendering framework has achieved pixel accuracy, crack-free, and real-time frame rate at the same time when handling large-scale NURBS models. The most recent and comprehensive study in this area is [Schollmeyer and Froehlich 2018], who implemented anti-aliased rendering of large trimmed NURBS models (around 150,000 Bézier patches) based on hardware tessellation. Unfortunately, their solution forwent pixel accuracy for performance reasons and could only achieve an interactive frame rate at a modest resolution. Earlier works either rarely provided test cases for NURBS models comparable to [Schollmeyer and Froehlich 2018] in size, or relied upon heuristic-based adaptive tessellation algorithm without guarantee of pixel accuracy [Krishnamurthy et al. 2009; Schwarz and Stamminger 2009].

1.1 Contribution

We present a novel adaptive tessellation framework, named ETER, for achieving pixel-accurate, crack-free, and real-time rendering at the same time for large-scale NURBS models. We develop a highly parallel adaptive tessellation algorithm, a CUDA-based and Tensor Cores accelerated NURBS evaluation algorithm, and an efficient crack-filling algorithm. To evaluate ETER's capability, we implement a prototype rendering system that integrates ETER with hybrid rasterization and generates the output into the visibility buffer [Burns and Hunt 2013].

To aim for maximal parallelism while achieving pixel accuracy, we introduce a uniform grid based adaptive tessellation algorithm that allows the tessellation factor of each patch to be computed independently without the information of adjacent patches. Because the generation of a uniform sampling grid does not require a hardware tessellator, this allows us to leverage the recently introduced Mesh Shaders based pipeline [Kubisch 2018] to its full potential, and to achieve better programmability. In addition, with uniform sampling grids, we are able to formulate the main step of the evaluation process of a Bézier patch as matrix multiplications. This opens a new door for us to leverage Tensor Cores [Appleyard and Yokim 2017] introduced with Nvidia Volta GPUs to achieve unprecedented evaluation efficiency, solving a major performance bottleneck in NURBS real-time rendering. Under pixel-accurate tessellation many (sub)pixel scale triangles may be generated, which are terrible for the hardware rasterizer [Karis et al. 2021]. In order to handle these small triangles efficiently, we implement hybrid software/hardware rasterization. Furthermore, thanks to the pixel-accurate nature of our tessellation algorithm, we develop a crack-filling algorithm based on visibility buffer and conservative rasterization to ensure sufficient rendering quality at a minimal performance cost. It is also worth noting that our crack-filling algorithm is perfectly compatible with software rasterization. Tests on an RTX 3090 Ti GPU show that our proposed ETER-based pipeline is able to render 0.25 billion triangles generated from 3.7 million Bézier patches tessellated on-the-fly at 30 FPS (frame-per-second).

The novel aspects of our work include:

- An elastic framework for tessellating large-scale NURBS models, with smooth integration into the state-of-the-art GPU-driven real-time rendering pipeline.
- A highly parallel adaptive tessellation algorithm and an efficient crack-filling algorithm to achieve pixel-accurate and high-quality rendering.
- Leverage Tensor Cores in Nvidia GPU to accelerate CUDA-based NURBS evaluation, and introduce Mesh Shaders and hybrid software/hardware rasterization for NURBS model rendering.

To the best of our knowledge, this work is the first in the literature to achieve pixel-accurate, crack-free, and real-time rendering at the same time for large-scale NURBS models. We are also the first to utilize Tensor Cores, Mesh Shaders, and hybrid software/hardware rasterization in NURBS model rendering. We are convinced ETER is highly elastic for the following reasons. First, our framework scales well to the degrees of surfaces as well as the sizes of models. Second, the adaptive tessellation algorithm in our framework adapts well to the surface curvature and viewpoint while guaranteeing pixel accuracy. Third, ETER runs entirely on GPU with the unique capability to harness the ever-growing neural computing power and can achieve great scalability on GPU due to the high parallelism of our algorithms. Finally, this framework can be easily integrated into the most advanced GPU-driven real-time rendering pipeline [Haar and Aaltonen 2015; Karis et al. 2021] due to the use of the visibility buffer. With its elastic nature, we expect the performance of this framework to only improve, if not significantly, with the advent of newer and faster GPUs and continuously upgraded graphic

pipelines. We believe our framework holds huge potential to become the favored NURBS rendering solution in future CAD systems.

2 RELATED WORK

2.1 Adaptive Tessellation

Approximation Error Control. Researchers have proposed various approaches for adaptive tessellation of NURBS surfaces to improve rendering efficiency and accuracy. Some of them use heuristic methods based on viewpoint-surface distance and/or screen bound, such as [Krishnamurthy et al. 2009; Schollmeyer and Froehlich 2018; Schwarz and Stamminger 2009]. Others like [Buchenau and Guthe 2021; Guthe et al. 2005] compute the tessellation factor by controlling the error between the exact surface and its triangular approximation using the method proposed by [Filip et al. 1986]. But such error lacked precise geometric interpretation, until Yeo et al. [2012] introduced the concept of pixel accuracy, which requires that the tessellation of a surface has the correct depth ordering and differs from the exact surface by at most half a pixel in screen space. Yeo et al. [2012] used upper and lower piece-wise linear bounds called slefes to achieve pixel accuracy, but this method cannot handle rational surfaces. [Hjelmervik 2014] achieved pixel-accurate rendering based on [Filip et al. 1986], which can be applied to any C^2 continuous surface. However, this method, along with other methods based on [Filip et al. 1986], requires the bounds on derivatives of surfaces, which are expensive to estimate in rational cases. To address this issue, Zheng and Sederberg [2000] proposed a method using homogeneous coordinates for rational surfaces that makes derivatives simple, and bound the maximum deviation based on the associated derivatives in homogeneous space.

Tessellation Induced Cracks. Adaptive tessellation with uniform sampling grids could produce cracks along patch boundaries due to inconsistent sampling rates of adjacent patches. Guthe et al. [2005] proposed to fill cracks by rendering an additional line strip around the boundary, but doing so requires an additional pass involving changes of the pipeline and might significantly impact the performance. Claux et al. [2014] presented an image-space crack detection and filling method, but this method may include non-crack pixels due to the lack of geometric information. Another approach to avoid cracks is to use semi-uniform sampling grids, which allows for the assignment of a common tessellation factor at the shared patch boundary that does not necessarily agree with the interior factors [Nießner et al. 2016; Schwarz and Stamminger 2009]. However, this often requires the use of hardware tessellation which has its own limitations - the tessellator has a limit on the maximum allowed tessellation level, if exceeded will require extra pre-tessellation [Schollmeyer and Froehlich 2018], and the evaluation performance of the domain shader is poor due to its inflexibility. An alternative to the hardware tessellation unit is a software-implemented tessallator [Dyken et al. 2009; Schwarz and Stamminger 2009], but inevitably the performance will be compromised.

Non-Uniform Sampling Grid. Because the curvature of a surface can vary significantly, using a uniform or semi-uniform sampling grid may result in unnecessary over-tessellation. [Eisenacher et al. 2009] presented a method using recursive subdivisions to achieve non-uniform sampling, but this method is not suitable for GPU parallelization. [Buchenau and Guthe 2021] proposed a re-parameterized method for achieving non-uniform sampling that is more parallelized and can utilize the tessellation units. However, this method is based on the fact that bi-cubic surfaces have linear second-order derivatives, making it unable to apply to rational surfaces or higher-order surfaces. In addition, this method does not outperform the uniform sampling method even with the reduced number of primitives.

2.2 Evaluation

NURBS surfaces can be directly evaluated in the exact NURBS form [Concheiro et al. 2014; Gatilov 2016; Krishnamurthy et al. 2009]. Other approaches firstly convert NURBS patches to Bézier patches because the Bézier evaluation is less computationally intense and has a higher degree of precision for certain optimization techniques such as back-patch culling. During tessellation, the sampling rate for each Bézier patch can be computed individually, thus mitigating the problem of unnecessary excess primitives due to strongly varying curvature across spans when using uniform sampling grids. Concheiro et al. [2014] mentioned that using a knot insertion algorithm to convert NURBS to Bézier may introduce artifacts at the boundaries of the surface due to floating-point errors. However, this conversion is typically performed on the CPU using double precision, while the rendering is performed on the GPU using single precision, so the floating-point errors introduced by the conversion do not exist during rendering at the boundaries.

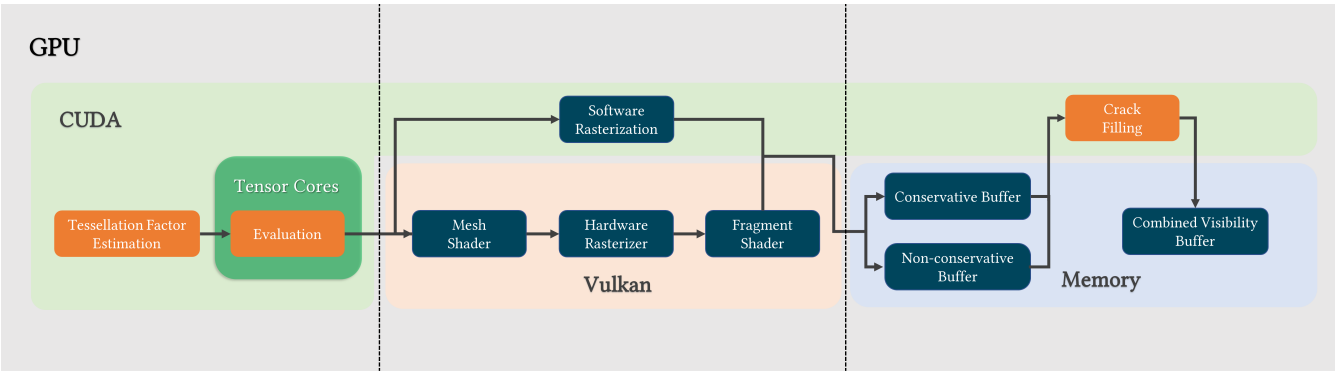
2.3 Rendering

It is common for CAD models to contain many complex shaped surfaces and small parts, resulting in a large number of triangles after tessellation. Sometimes the triangle counts could reach tens to hundreds of millions, in which case the traditional pipeline can easily reach bottlenecks. To address this challenge, a new programmable geometric shading pipeline built on Mesh Shaders was introduced in Nvidia's Turing architecture to increase the flexibility and efficiency of the geometry pipeline [Kubisch 2018].

For the real-time rendering of large-scale triangle meshes, one of the latest GPU-driven implementations is Nanite [Karis et al. 2021]. Nanite is designed for real-time gaming to render pixel-level details and high object counts, which is also common in CAD rendering scenarios. Some of the engineering practices in Nanite are inspiring for CAD rendering, such as soft rasterization and visibility buffer [Burns and Hunt 2013]. However, Nanite is still based on pre-tessellated meshes, and there is currently no solution similar to Nanite for real-time rendering of NURBS.

3 OVERVIEW

Input. We first convert the NURBS model into a set of rational Bézier surfaces using knot insertion [Piegl and Tiller 1997]. Then the bounds of the second derivatives of each Bézier surface in homogeneous space are estimated to determine tessellation factors. In addition, we compute the normal cones of Bézier surfaces defined in [Shirmun and Abi-Ezzi 1993] for efficient back patch culling. These steps comprise the pre-process.



(a) Adaptive Tessellation (b) Hybrid Rasterization (c) Crack Filling

Fig. 2. Overview of our rendering pipeline. Boxes with background in orange represent our proposed algorithms.

Here, we define a rational Bézier surface $\mathbf{S}(u,v)$ of order $m \times n$ in homogeneous space as:

$$\begin{aligned}\overline{\mathbf{S}}(u,v) &= (\mathbf{R}(u,v), w(u,v)) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{C}_{ij} B_i^m(u) B_j^n(v) \\ &= \begin{pmatrix} B_0^m(u) \\ B_1^m(u) \\ \vdots \\ B_m^m(u) \end{pmatrix}^T \begin{pmatrix} \mathbf{C}_{00} & \mathbf{C}_{01} & \cdots & \mathbf{C}_{0n} \\ \mathbf{C}_{10} & \mathbf{C}_{11} & \cdots & \mathbf{C}_{1n} \\ \vdots & \vdots & & \vdots \\ \mathbf{C}_{m0} & \mathbf{C}_{m1} & \cdots & \mathbf{C}_{mn} \end{pmatrix} \begin{pmatrix} B_0^n(v) \\ B_1^n(v) \\ \vdots \\ B_n^n(v) \end{pmatrix} \\ &= \mathbf{b}_m^T(u) \mathbf{C} \mathbf{b}_n(v), \ (u,v) \in [0,1]^2\end{aligned}$$

where $\mathbf{C}_{ij} = (x_{ij}w_{ij}, y_{ij}w_{ij}, z_{ij}w_{ij}, w_{ij}) \in \mathbb{R}^4$ are the control vertices, $B_i^m(u) = \binom{m}{i} u^i (1-u)^{m-i}$ and $B_j^n(v) = \binom{n}{j} v^j (1-v)^{n-j}$ are the Bernstein basis functions, $i = 0, \cdots, m$, $j = 0, \cdots, n$. In Euclidean space, $\mathbf{S}$ can be represented as: $\mathbf{S}(u,v) = \frac{\mathbf{R}(u,v)}{w(u,v)}$.

3.1 Preliminaries

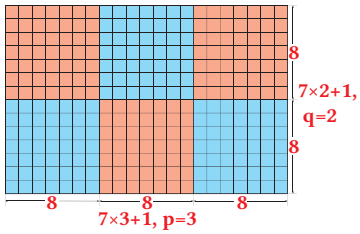
Uniform Sampling Grid. We use a uniform sampling grid to tessellate each Bézier patch. On a uniform sampling grid $\{u_0, ..., u_k\} \times \{v_0, ..., v_l\} \subset [0,1]^2$, we can formulate the evaluation of all the vertices as matrix multiplications:

$$\mathbf{S} = \mathbf{B}_m^T \mathbf{C} \mathbf{B}_n, \tag{1}$$

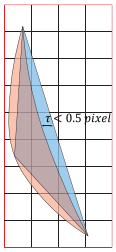
where $\mathbf{B}_m = (\mathbf{b}_m(u_0), \cdots, \mathbf{b}_m(u_k))$, $\mathbf{B}_n = (\mathbf{b}_n(v_0), \cdots, \mathbf{b}_n(v_l))$. We then use Tensor Cores [Appleyard and Yokim 2017] to perform the matrix multiplications to accelerate the evaluation process.

Meshlets. Using Mesh Shaders for rendering requires the meshes to be decomposed into meshlets (similar to triangle clusters), to optimize vertex reuse within a single meshlet, reduce the overall memory accesses and reduce fixed-function impact in primitive processing [Kubisch 2018]. According to Nvidia, it is recommended to use a meshlet with 64 vertices, which in our case corresponds to 8×8 sampling vertices.

Sampling Grid Division. We scale up the sampling rate to $(7p+1) \times (7q+1)$, so that we can divide the sampling grid into $p \times q$ blocks, each with 8×8 vertices to generate a meshlet. The inset figure on the right shows the case where $p = 3, q = 2$. Besides, Tensor Cores only support matrices of certain sizes. By scaling up the sampling rate, for surfaces with degrees less than or equal to 7, the evaluation process can be converted into $2 \times p \times q$ multiplications of 8×8 matrices, which are suitable for Tensor Cores.



Pixel Accuracy. We adopt the error metric called pixel accuracy proposed by [Yeo et al. 2012] for our tessellation algorithm, which is defined as covering accuracy plus parametric accuracy. Covering accuracy requires that each pixel's color be controlled by all unoccluded patches, which cover it sufficiently for correct depth ordering. Parametric accuracy requires that the tessellated triangle meshes differ from the exact surface by at most 0.5 pixels in screen space. The inset figure on the right shows the parametric accuracy, where the red part is the curved surface and the blue part is the linear approximation. However, Yeo's method [Yeo et al. 2012] to achieve pixel accuracy cannot be applied to rational surfaces. Therefore, we propose our own tessellation algorithm to compute the tessellation factor based on [Zheng and Sederberg 2000], and implement it with CUDA.



3.2 Pipeline Overview

Figure 2 provides an overview of our ETER-based Mesh Shader pipeline. The pipeline mainly consists of three stages: adaptive tessellation, hybrid rasterization, and crack filling.

3.2.1 *Adaptive Tessellation.*

Tessellation Factor Estimation. We implement our tessellation factor estimation algorithm with CUDA. During this stage, the computation of tessellation factors of each patch does not require any information of adjacent patches, which allows our algorithm to achieve maximal parallelism. We derive an approximation error in object space which ensures pixel accuracy, and then apply Zheng's method [Zheng and Sederberg 2000] to get the tessellation factors (Section 4). The choice of rasterization is also made during this stage (Section 6.2).

Surface Evaluation. Our evaluation algorithm is also implemented with CUDA. The formulation (1) allows us to leverage Tensor Cores to accelerate matrix multiply-accumulate (MMA) for evaluation [Appleyard and Yokim 2017], thus removing a major performance bottleneck in NURBS real-time rendering. As the final step of evaluation, we use the method proposed by [Ootomo and Yokota 2022] to recover single precision accuracy from Tensor Cores with a very minimal performance cost (Section 5.2). The tessellated triangle meshes are split and stored into two buffers based on the rasterization decision made during the previous stage.

3.2.2 Hybrid Rasterization. The triangle meshes generated during the tessellation stage are decomposed into meshlets. Meshlets consisting of small triangles are processed by a software rasterizer implemented with CUDA, and other meshlets are processed by a Mesh Shader pipeline (Task Shaders are not used). Both rasterization schemes use conservative rules, and write the results into two visibility buffers, depending on if the fragments are generated in accord with conservative rules (Section 6.2).

3.2.3 Crack Filling. Adaptive tessellation with uniform sampling grids may produce cracks. Thanks to pixel-accurate tessellation, we develop an efficient crack-filling algorithm based on conservative rasterization and visibility buffer (see Section 6.1). After crack filling, a visibility buffer without tessellation-induced cracks is generated.

4 PIXEL-ACCURATE TESSELLATION

4.1 Tessellation Factor Estimation

To achieve pixel accuracy, the distance in the model object space between the exact surface and its tessellation must be controlled. As described above, we choose the method by [Zheng and Sederberg 2000] since it only requires bounds on derivatives of rational surfaces in homogeneous space. But before we can apply the method by [Zheng and Sederberg 2000], we need to first derive an approximation error in object space from screen-space error.

Approximation Error. Considering perspective projection, the projection matrix transforms a vertex $V = (x, y, z, 1)$ in camera space to $V' = (A\frac{x}{-z}, B\frac{y}{-z}, \hat{z}, 1)$ in clip space, where $\hat{z}$ is the depth value of the vertex. Let R be the rectangular area in clip space centered at V' with width τ_x and height τ_y, and we want to compute the size of the largest cube centered at V in the preimage (i.e., the inverse of the projection transformation) of R (see Figure 3). Because any point in such a cube will be projected in R, if we set the size of R to be the size of a pixel, we can then use the size of the cube as the desired object space error to guarantee parametric accuracy. We can accomplish it by computing the sizes of this cube in x and y directions separately and then taking the minimum of them.

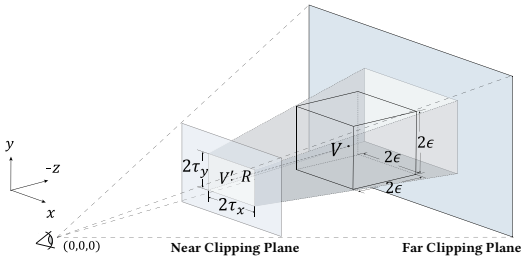


Fig. 3. The gray area is the preimage of the rectangular region on the near clipping plane with respect to the projection transformation. By identifying the largest cube that can fit within this gray area, we can determine the approximation error that ensures parametric accuracy.

Take x direction as an example, and we expand the cube centered at V until it touches the left or the right side of the preimage of R, and the size of the cube is $2\epsilon_1$ or $2\epsilon_2$, which should satisfy the following equations respectively:

$$A\frac{x+\epsilon_1}{|z|-\epsilon_1} = A\frac{x}{|z|} + \tau_x,\ A\frac{x-\epsilon_2}{|z|-\epsilon_2} = A\frac{x}{|z|} - \tau_x.$$

Solve them and we obtain the desired size:

$$\epsilon_x = \min\{\epsilon_1, \epsilon_2\} = \frac{|z|\tau_x}{A + A|\frac{x}{z}| + \tau_x}.$$

For the region with $x > \tau_x$ and the region with $x < -\tau_x$, we can obtain the same result. Analogously, for the y direction, the size of the cube is: $\epsilon_y = \frac{|z|\tau_y}{B+B|\frac{y}{z}|+\tau_y}$, so we have:

$$\epsilon = \min\left\{\frac{|z|\tau_x}{A(1+|\frac{x}{z}|)+\tau_x}, \frac{|z|\tau_y}{B(1+|\frac{y}{z}|)+\tau_y}\right\}.$$

We set τ_x, τ_y to be the size of half a pixel in the clip space, namely:

$$\tau_x = \frac{1}{W},\ \tau_y = \frac{1}{H},$$

where W, H are the width and height of the screen. For a patch, we can use its bounding box in the camera space to easily find the minimum value of z and the maximum value of $\left|\frac{x}{z}\right|$ and $\left|\frac{y}{z}\right|$ to compute ϵ.

Tessellation Factor. After acquiring the error bound, we directly apply it to the inequality proposed by [Zheng and Sederberg 2000] to get the distance between sampling vertices: $\Delta u, \Delta v$. Then we get the tessellation factors $t_u = \lceil\frac{1}{\Delta u}\rceil$, $t_v = \lceil\frac{1}{\Delta v}\rceil$, which guarantees the parametric accuracy.

Covering Accuracy. To ensure covering accuracy (Section 3.1, Pixel Accuracy) in addition to parametric accuracy, the tessellation factor estimation stage requires an additional error bound ϵ_z in z component. The error bound used in Zheng's method [Zheng and Sederberg 2000] should be determined by taking the minimum value between ϵ and ϵ_z. For each patch, ϵ_z can be the minimum distance to any non-adjacent patches.

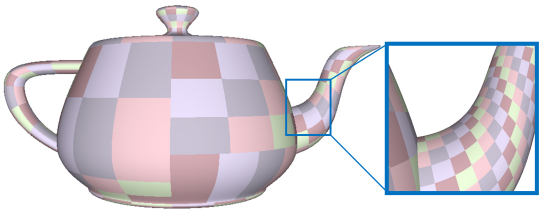


Fig. 4. Triangle meshes are decomposed into meshlets. Each color block represents a single meshlet, which corresponds to an 8x8 sampling grid. When zooming in, pixel accuracy is ensured through an increase in the tessellation factor, resulting in the generation of additional meshlets.

4.2 Tessellation Factor Adaption

After computing the tessellation factors, we then scale them up (Section 3.1, Sampling Grid Division), so that the tessellated meshes can be intuitively divided into $p \times q$ meshlets with 64 vertices (see Figure 4), without additional processing in the case of insufficient vertices in the meshes for a meshlet. Due to the fixed topology of an 8×8 uniform sampling grid, the vertex indices in a meshlet can be computed based on the triangle index without the need for the index buffer.

Note that even though the adaption of tessellation factors will increase the number of sampling vertices, it will not impact the performance due to our unique evaluation approach, which will be further explained in Section 5.2.

5 GPU ACCELERATED NURBS EVALUATION

Since we formulate the evaluation of Bézier surfaces on uniform grids as Equation (1), during evaluation, each thread only needs to compute two Bernstein basis functions, one for matrix $\mathbf{B}_m$ and one for matrix $\mathbf{B}_n$. Then other Bernstein basis functions can be fetched from other threads by interthread data exchange. The remaining work is to linearly combine these basis functions with control vertices, i.e., the matrix multiplications in Equation (1). CUDA supports efficient interthread data-sharing mechanisms such as shared memory, which is not available for domain shaders. Evaluation using domain shaders requires computing $m+n$ Bernstein basis functions in a single thread, resulting in much worse evaluation performance than using CUDA. Another thing worth noting is that if a semi-uniform sampling grid is used, boundary and interior vertices typically do not have common uv coordinates, so they cannot share the Bernstein functions. In addition, the distribution of boundary vertices is often irregular, which can lead to divergence if boundary and interior vertices are evaluated within the same warp. Hence we need to evaluate boundary vertices separately, which also leads to poorer evaluation performance.

5.1 Tensor Cores Acceleration

Evaluation Bottlenecks. Computing a single Bernstein basis function requires $n+1$ multiplications, since the binomial coefficient $\binom{n}{i}$ can be precomputed. The matrix multiplication in equation (1) is $O(n^2)$. Memory operations, like interthread data exchange, are more expensive than arithmetic operations, making matrix multiplication the most time-consuming step of the evaluation process.

Evaluation with Tensor Cores. Tensor Cores in NVIDIA GPU can deliver remarkable performance with mixed-precision matrix multiply [Appleyard and Yokim 2017], which can be utilized in our implementation to achieve significant performance speedups. For surfaces with degrees less than or equal to 7×7, after dividing the sampling grid into several blocks with 8×8 vertices, the orders of all matrices during evaluation are less than 8×8. Tensor Cores provide hardware acceleration for matrix multiplication and addition of certain sizes, which happens to be very suitable for our problem. Tensor Cores support the matrix size of m16n8k8, a 16×8 matrix multiplied by an 8×8 matrix, which can be directly introduced into our algorithm without much overhead. Therefore, our evaluation process can be divided into two steps. The first step is to compute the Bernstein basis function, and the second step is to use Tensor Cores to perform matrix multiplication.

Since we use 16×8 matrices, and only 8×8 slots in them are used, we can put the derivatives of the Bernstein basis functions in the other 8×8 slots, allowing both the coordinates and the normals of the vertices to be computed at the same time.

After computing $\mathbf{B}_m, \mathbf{B}'_m, \mathbf{B}_n, \mathbf{B}'_n$ in the first step, use Tensor Cores to compute:

$$\begin{pmatrix} \mathbf{B}_m^T \\ \mathbf{B}_m^{T\prime} \end{pmatrix}_{16\times 8} \mathbf{C}_{8\times 8} = \begin{pmatrix} \mathbf{B}_m^T \mathbf{C} \\ \mathbf{B}_m^{T\prime} \mathbf{C} \end{pmatrix}_{16\times 8},$$

then multiply the result with matrix $\mathbf{B}_u$ to get:

$$\begin{pmatrix} \mathbf{B}_m^T \mathbf{C} \\ \mathbf{B}_m^{T\prime} \mathbf{C} \end{pmatrix}_{16\times 8} (\mathbf{B}_n)_{8\times 8} = \begin{pmatrix} \overline{\mathbf{S}} \\ \overline{\mathbf{S}}_v \end{pmatrix}_{16\times 8},$$

and do the last matrix multiplication:

$$\begin{pmatrix} \mathbf{B}_m^T \mathbf{C} \\ 0 \end{pmatrix}_{16\times 8} (\mathbf{B}'_n)_{8\times 8} = \begin{pmatrix} \overline{\mathbf{S}}_u \\ 0 \end{pmatrix}_{16\times 8}.$$

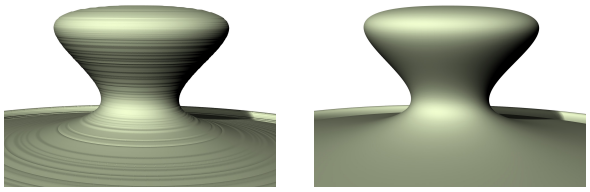
And finally, $\frac{\mathbf{R}_u w - \mathbf{R} w_u}{w^2} \times \frac{\mathbf{R}_v w - \mathbf{R} w_v}{w^2}$ is computed to complete the evaluation of coordinates and normals simultaneously.

5.2 Discussion and Remark

Single Precision Recovery. Using Tensor Cores can result in a loss of accuracy due to the adoption of half-precision floating point numbers, which may cause noticeable visual artifacts, as shown in Figure 5. To address this, we use the method proposed by [Ootomo and Yokota 2022] to recover single precision accuracy from Tensor Cores with very minimal performance cost (less than 10% as shown in Table 1).

Analysis. As shown in Table 1, a 40% to 302% speedup is achieved by Tensor Cores acceleration with single precision recovery, with shared memory optimized evaluation kernel as the baseline. Besides shared memory, the baseline version has also adopted various other optimization techniques including coalescing load and store, bank conflict elimination, and persistent threading. The speedup is more significant as the surface degree increases. The remarkable speedup can be explained from a GPU SM (streaming multiprocessor) perspective. To be specific, a considerable proportion of memory accesses and floating point calculations have been migrated to Tensor Cores.

When Tensor Cores are used to accelerate surface evaluation, irrespective of surface orders the matrices are always of order 16×8



(a) Evaluation with half-precision.

(b) Evaluation with single precision recovery.

Fig. 5. The result of using Tensor Cores to accelerate surface evaluation. (a) Insufficient numerical accuracy of half-precision results in wavy artifacts. (b) Recovering single precision accuracy from Tensor Cores can greatly eliminate the wavy artifacts.

Surface Degree	3	4	5	6	7
w/o Tensor Cores	4.373	5.456	6.711	8.197	9.885
w/ Tensor Cores	2.904	2.949	2.966	2.947	2.976
w/ Tensor Cores recover	3.125	3.158	3.267	3.261	3.266

Table 1. Time (in ms) spent on evaluating vertex positions and normals in 3 CUDA kernels (shared memory optimized, Tensor Cores accelerated, Tensor Cores accelerated with single precision recovery). 81,920,000 sampled vertices are evaluated on 20,000 patches, each with a tessellation factor of 64x64.

or 8×8. As a result, the performance difference of evaluation of different surfaces is mainly, if not only attributed to the computation of the Bernstein basis function, which has a very small computational cost. Therefore, the evaluation time of the surface is almost irrelevant to the surface orders. This also explains why higher order surface gains greater acceleration with Tensor Cores, because the increasing surface order will significantly slow the evaluation performance when Tensor Cores are not used.

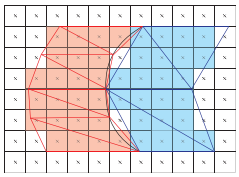
Since we need to scale up the sampling rate to $(7p+1) \times (7q+1)$, this would potentially increase the number of samples. Nevertheless, with our approach, the time required to compute 8×8 points is the same as any number of points less than 8×8. This is again because the size of the matrix computed by Tensor Cores is always fixed.

6 REAL-TIME RENDERING

In order to support complex lighting and material systems, we implement a similar rendering pipeline to [Karis et al. 2021] based on the visibility buffer introduced by [Burns and Hunt 2013]. Each sample in the visibility buffer uses 8 bytes, of which 4 bytes is depth, and the other 4 bytes is meshlet index and triangle index. This allows us to perform efficient atomic depth test [Liu et al. 2010]. After the visibility buffer is created, we obtain the vertex information of the triangle through the meshlet index and the triangle index, compute the barycentric coordinates of the pixel, and interpolate the attributes of the three vertices to complete the rendering process.

6.1 Crack Filling

Tessellation Induced Cracks. Because uniform sampling grids are used and the tessellation factors are determined independently for



each patch, the triangular approximations of adjacent patches may produce gaps at the boundaries if their tessellation factors are different. During the typical rasterization stage, if the center of a pixel falls within this gap interval, the pixel will not be considered covered thereby resulting in cracks, as shown in the inset figure on the left.

Since our tessellation factors ensure parametric accuracy, the distance between the Bézier patch and its triangular approximation at the boundary should be less than 0.5 pixel in screen space. Because adjacent patches share one identical boundary curve, an intuitive conclusion is that the distance between the triangular approximations of adjacent patches at the boundary is less than 1 pixel in screen space. Thus, at each pixel where the crack occurs, there is always at least one triangle edge lying in it, and under the rules of conservative rasterization, this pixel would be considered covered. Therefore, when conservative rasterization is used, tessellation-induced cracks will not appear, and we can complete crack filling without the need for an additional pass.

Combined Crack Filling. The biggest advantage of using conservative rasterization is that it provides all the candidate fragments that could be used for crack filling, although we cannot apply them directly for rendering. Due to the generation of redundant fragments, in some cases such as when patches are assigned with different colors (Figure 6, b), if we apply conservative rasterization directly, it will produce severe jagged effects at the edges of patches with different colors. Thanks to our visibility buffer and pixel-accurate tessellation based approach, we develop an efficient crack-filling algorithm using fragments generated by conservative rasterization rules, while keeping the run-time cost at minimal.

First, depending on whether the pixel center is inside or outside the triangles, the fragments are written into two different buffers respectively: non-conservative buffer and conservative buffer. Then during rendering stage, we compare the elements in these two buffers to determine whether the pixel contains a crack. It is a trivial case if an element in either buffer is the background. However, if both elements are non-background, we first compare their depths. If the depth of the element in the non-conservative buffer is smaller, we render that buffer. Otherwise, we need to compare the NURBS indices of the two elements. If they are identical, we continue to render the non-conservative buffer. If the NURBS indices are different, we examine eight surrounding pixels in the non-conservative buffer. Because the size of a crack does not exceed 1 pixel, if the pixel has a crack, its corresponding NURBS index will appear at most 3 times in 9 pixels including itself. So when the NURBS index of the pixel has repeat times less than or equal to 3 times among the 9 pixels, the pixel is likely to have a crack, and we should render the conservative buffer (see supplementary material for pseudo code).

Discussion and Remark. When processing pixels whose center is not inside the triangle, conservative rasterization will perform extrapolation of triangle vertices, which can result in problems such as incorrect depth ordering and incorrect trimming results. Therefore, we need to truncate the depth value and *uv* coordinates within the range of the triangle. In addition, when rendering with

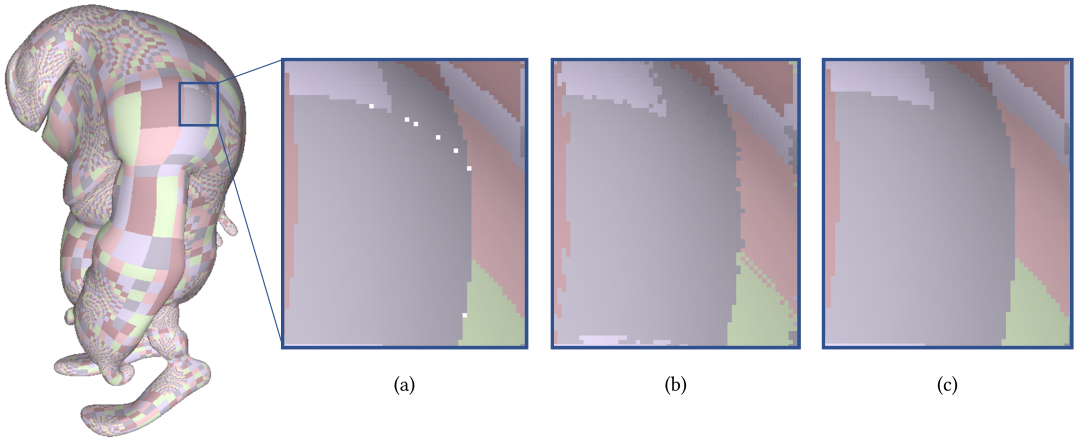


Fig. 6. (a) Using independent tessellation factor estimation and uniform sampling grid result in cracks, as shown in white pixels. (b) Directly applying conservative rasterization fills the cracks, but also results in severe jagged effects. (c) Our algorithm not only fills the cracks, but also avoids conservative rasterization induced jagged effects.

visibility buffer, the obtained barycentric coordinates also need to be truncated within the range from 0 to 1 to avoid extrapolation.

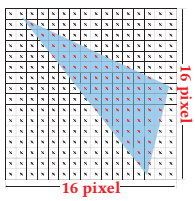
Compared to directly applying conservative rasterization, our algorithm can greatly reduce the jagged effect, as shown in Figure 6. The remaining effect can be further smoothed out by applying a common image-space anti-aliasing algorithm like [Jimenez et al. 2012], to a level that might be barely noticeable.

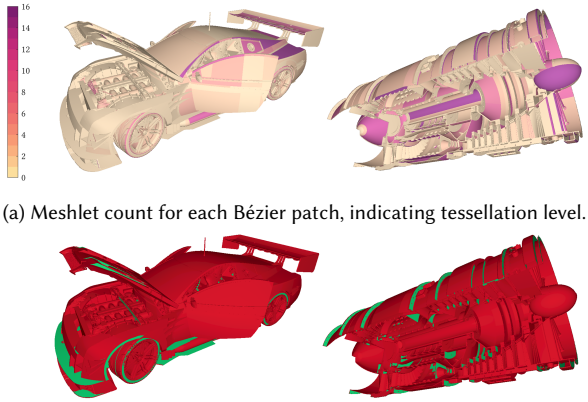
6.2 Hybrid Rasterizer

The hardware rasterizers and distribution of fragment shaders of modern GPUs are designed and optimized to handle big triangles that cover multiple pixels. In our case, using a pixel-accurate sampling rate may generate many pixel-level or even sub-pixel-level triangles, having significant impacts on the performance of fragment generation and computation (Figure 7).

According to [Karis et al. 2021], using a software rasterizer can achieve higher speed than hardware rasterizers in the cases above. This is also proved to be true according to our software rasterizer implementation, see Figure 8.

Before applying software rasterization, we need to first determine the maximum size of a small triangle. Similar to hardware rasterization, we need to project the vertices of the triangle on the screen, and then snap the vertices to the grid point of the subpixel. Usually in such grid, each has 256*256 subpixels. For triangles whose bounding rectangles are smaller than 16×16 pixels in screen space, their vertices after snapping can be represented by 12-bit fixed-point numbers, with a 4-bit integer part and an 8-bit fractional part. The result of multiplication can be accurately represented by 24-bit fixed-point numbers with an 8-bit integer part and a 16-bit fractional part. This means we can achieve a level of precision similar to hardware rasterizers through the use of floating-point numbers. Therefore, in our definition, the bounding rectangle of a small triangle cannot exceed 16×16 pixels.





(a) Meshlet count for each Bézier patch, indicating tessellation level.

(b) Triangles smaller than 16 pixels are marked as red, and the other triangles are marked as green.

Fig. 7. (a) The meshlet count for each Bézier patch, which is equivalent to tessellation factors. For most patches, the meshlet count will not exceed 16. (b) Distribution of triangles smaller than 16 pixels under the tessellation factors in (a). Most of the triangles are small, results in poor performance when using hardware rasterization.

Additionally, if we can determine a sampling rate such that all tessellated triangles are smaller than 16×16 pixels, we can complete the entire geometry pass of tessellation factors estimation and evaluation with our pure GPU software rendering pipeline. Besides, such sampling rate can also be used to determine the rasterization mode for a patch when using hybrid rasterization. If the pixel-accurate sampling rate is larger than the aforementioned rate, it indicates that the edge lengths of the tessellated triangles will be less than 16×16 pixels in screen space and should be processed by a software rasterizer.

We can control the edge length of the triangle with the first derivative:

$$\|S(u_1, v_1) - S(u_2, v_2)\| \leq \|S_u\||u_1 - u_2| + \|S_v\||v_1 - v_2|.$$

Similar to using the second-order derivative to control the sampling error, the first-order derivative of the rational surface is also difficult to calculate. So we draw a similar conclusion according to the derivation process of [Zheng and Sederberg 2000]. For rational surface $\mathbf{S}(u, v) = \frac{\mathbf{R}(u,v)}{w(u,v)}$, the sampling rate Δu, Δv should satisfy

$$\frac{\Delta u D_u + \Delta v D_v}{\min w} < \epsilon, \tag{2}$$

where

$$D_u = \begin{cases} \sup(\|\mathbf{R}'_u(u, v)\| + (r - \epsilon)|w'_u(u, v)|), & \epsilon < r \\ \sup \|\mathbf{R}'_u(u, v)\|, & r \leq \epsilon < 2r \\ 0, & \epsilon > 2r \end{cases},$$

and D_v is analogous to D_u. Let $\frac{\Delta u}{\Delta v} = \frac{D_v}{D_u}$, and the solution to Inequality (2) is:

$$\Delta u < \frac{\epsilon \min w}{2D_u}, \quad \Delta v < \frac{\epsilon \min w}{2D_v}.$$

Similar to Section 4.1, we use the screen-space error to determine ϵ, only that now the error in the clip space corresponds to 16 pixels, that is, $\tau_x = \frac{32}{W}$, $\tau_y = \frac{32}{H}$.

7 IMPLEMENTATION AND RESULTS

	NURBS Surfaces	Bézier Surfaces	Trimming Loops	Storage (Geom+Trim)
Lego	309,245	794,485	175,324	202+130 MB
Car	61,245	105,939	49,176	22.5+33.6 MB
Turbine	46,466	88,113	42,765	22.4+36.6 MB
Drone	1,030	2,694	767	0.68+0.48 MB

Table 2. The number of surfaces and trimming loops in the NURBS models. Storage shows the memory usage during run-time, including geometry files for control vertices, and trimming loops encoded in KD-tree for fast trimming tests.

We implement ETER in CUDA, and integrate it into a Mesh Shaders, hybrid rasterization, and visibility-buffer-based rendering pipeline in Vulkan. We use the Blinn-Phong reflection model for visibility buffer display. Table 2 lists the geometric information and storage requirements of the models for testing. All tests are performed on an Intel core i7 13700kf PC with 64GB RAM equipped with a single NVidia GeForce RTX 3090 Ti GPU with 24GB VRAM.

7.1 Implementation Detail

Before evaluation, we can adopt several culling methods to reduce the overall workload. During the tessellation factor estimation stage, we compute the minimum z, the maximum $\left|\frac{x}{z}\right|$ and $\left|\frac{y}{z}\right|$ of a patch in camera space. In fact, we can additionally compute maximum z, minimum $\left|\frac{x}{z}\right|$ and $\left|\frac{y}{z}\right|$ for view frustum culling, where two z values can be used for far and near planes culling, minimum $\left|\frac{x}{z}\right|$ and $\left|\frac{y}{z}\right|$ can be used to compare with the field of view for side culling.

During the evaluation stage, we use 32 threads (i.e. a warp) for each 8×8 sampling grid to perform the evaluation and output a meshlet. There are two main reasons for taking this approach. One is that the minimum operating unit of Nvidia's GPU is a warp, and the threads in a warp always execute the same instruction simultaneously which improves parallelism by minimizing synchronizations between warps. The second reason is in RTX20 and 30 series graphics cards, Tensor Cores only support warp level matrix multiply and add. In our implementation, each thread in a warp is mapped to 4 Bernstein basis functions, 2 in the u direction and 2 in the v direction. This allows us to compute all the necessary Bernstein basis functions required for evaluating 8×8 sampling vertices of Bézier patches with degrees up to 7. Then we can read the patch control vertices and directly perform matrix multiplication with the basis functions previously computed.

Tensor Cores support a variety of matrix sizes, among which m16n8k8 and m16n8k16 seem to be most suitable for our problem. The last size m16n8k16 also provides two different kinds of instructions - dense and sparse. In order to use Tensor Cores, the data in a warp must meet a specific layout. It is quite straightforward to convert the data into the layout required by Tensor Cores in advance. However, our evaluation requires two-step matrix multiplication, meaning the result of the first step will be used as the left matrix in the second step, so the best case is that the layout of the result of the matrix multiplication is consistent with the input layout, where the m16n8k8 instruction turns out to be a perfect match for our requirement. For the m16n8k16 instruction, we can empty half of the 0s while filling in the required data. This allows us to use the sparse instruction, whose theoretical speed can reach twice that of the dense instruction, and the result of this instruction can be used as the left matrix of the second multiplication without additional processing, making it also a good choice. However, programs with sparse instructions are more complex and difficult to implement. According to [Sun et al. 2022], sparse instruction generally cannot reach its theoretical speed. In our own implementation, we find that there is not much speed difference between using sparse instructions and dense instructions for evaluation (Table 3).

We implement three different rendering schemes based on different rasterization methods: pure hardware rasterization, hybrid rasterization, and pure software rasterization. The first two schemes use the simplest Mesh Shaders which consume the meshlets generated by our CUDA-implemented tessellation framework (see Sections 4 and 5).

For real-time tessellation and rendering of NURBS surfaces, the practical trimming method is usually to use the uv coordinates of the fragment to determine whether it is to be discarded in fragment shaders. Guthe et al. [2005] used trim textures, but such textures need to be constructed for each frame depending on viewpoint to ensure certain view-dependent accuracy. Schollmeyer and Fröhlich [2009] built a hierarchical acceleration structure by splitting the trim curves into monotonic segments with respect to the two parameter dimensions of the surface patches, and used a bisection method to classify a parametric point. Later, they improved their acceleration structure using kd-tree [Schollmeyer and Froehlich

Surface Degree	3	4	5	6	7
Hardware Tess	15.624	27.033	44.09	65.374	91.345
CUDA	6.786	7.795	9.048	10.447	12.078
w/ MMA sparse	5.334	5.361	5.386	5.431	5.424
w/ MMA dense	5.17	5.242	5.263	5.326	5.3
w/ MMA dense and recover	5.431	5.505	5.55	5.587	5.632

Table 3. Time (in ms) spent on evaluating vertex positions and normals in the Mesh Shader pipeline and hardware tessellation pipeline respectively. 81,920,000 sampled vertices are evaluated on 20,000 patches, each with a tessellation factor of 64x64. Rasterizer discard is enabled to discard tessellated triangles before the rasterization stage. A method introduced by [Mann and DeRose 1995] is implemented for hardware tessellation.

2018], which has better performance. Since the source code is not available, we implement our own modified version of [Schollmeyer and Froehlich 2018]. Note that our trimming implementation may still have some defects that could result in artifacts, such as popping up of untrimmed boundaries while moving the camera. However, efficient NURBS trimming is non-trivial and outside the scope of this work, and is regarded as an interesting direction for future research. See Table 4 for the performances comparison of different rasterization schemes when trimming is enabled.

		Min	Max	Avg
1280×720	Lego	6.565 (4.663)	9.713 (6.549)	8.054 (5.215)
	Car	1.757 (1.151)	3.178 (1.979)	2.118 (1.27)
	Turbine	1.825 (1.063)	4.145 (1.992)	2.319 (1.251)
	Drone	0.665 (0.549)	1.175 (0.914)	0.757 (0.598)
1920×1080	Lego	7.654 (5.04)	12.764 (7.855)	9.632 (5.808)
	Car	2.129 (1.297)	3.972 (2.301)	2.598 (1.489)
	Turbine	2.339 (1.253)	5.745 (2.965)	3.159 (1.573)
	Drone	0.796 (0.594)	1.356 (1.01)	0.898 (0.656)

Table 4. Frame times (in ms) of different models in 720p and 1080p. The numbers in brackets are frame times without trimming test. We carefully select 3 camera distances, ranging from close to far, and methodically rotate the camera around the model at each distance. Note that all tests are performed in a view similar to isometric view.

7.2 Result Analysis

Table 3 demonstrates the superior performance of ETER's evaluation algorithm in comparison to hardware tessellation. The proposed Tensor Cores accelerated evaluation algorithm with single precision accuracy recovery is approximately 2.9-16.2x faster, with an increasingly significant speedup as the surface degree increases. As the table also shows, a shared memory optimized CUDA kernel without Tensor Cores acceleration can achieve a 2.4-7.5x speedup, further validating the benefits of using a Mesh Shader pipeline with greater flexibility.

Table 4 compares the frame times when the trimming test is on or off. It shows that when the trimming test is on, the overall frame time increases by 20%-100%. This highlights that the trimming test is one of the bottlenecks in the NURBS rendering pipeline. The variation in the extent of the slowdown can be attributed to the diverse geometric complexities of the models used.

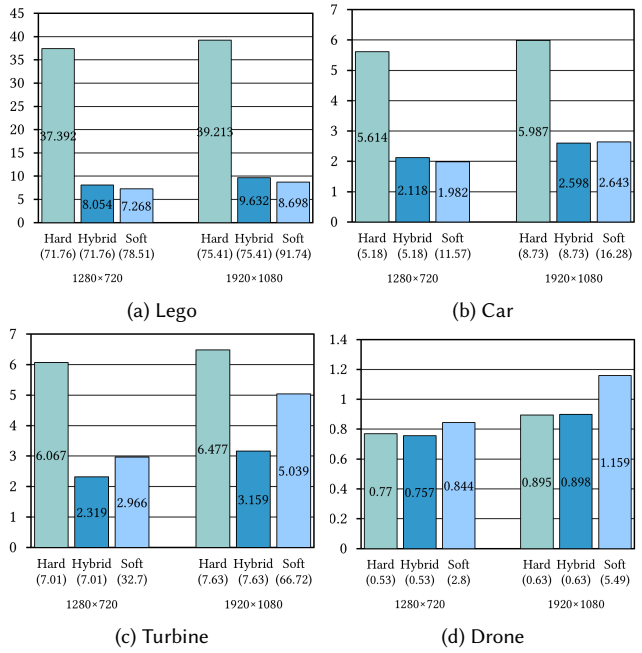
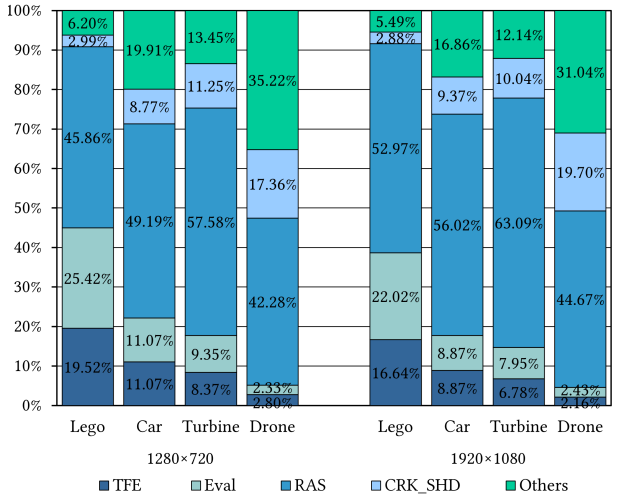


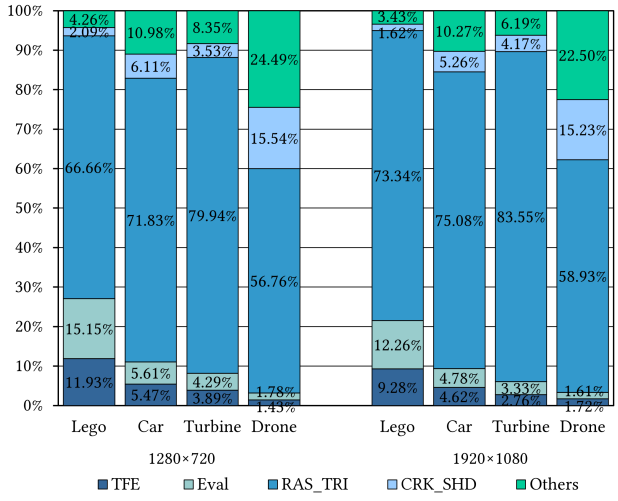
Fig. 8. Comparison of frame time (in ms) with different rasterization schemes. The triangle count (in millions) is listed below the rasterization method. When using pure hardware or hybrid rasterization, since the same pixel-accurate tessellation factors are applied, they will result in the same triangle counts. For pure software rasterization, the tessellation factors outlined in Section 6.2 are utilized, which in some cases can lead to a much higher triangle count thus worse performance. In general, for large models, implementing hybrid rasterization significantly improves rendering speed.

Figure 8 illustrates the frame times under different rasterization schemes. Hybrid rasterization achieves 2-5x speedup compared to pure hardware rasterization, which aligns with the previously reported 3x speedup by [Karis et al. 2021]. It is worth noting that pure software rasterization performs best in several cases. However, to apply pure software rasterization all tessellated triangles must be small enough, which inevitably leads to a significant increase in triangle numbers and ultimately results in a performance decline for most models. Therefore, hybrid rasterization is preferred for the pixel-accurate rendering of large-scale NURBS models.

Figure 9 illustrates the breakdown of the frame into four main stages: tessellation factor estimation (TFE), evaluation (EVAL), rasterization and trimming test (RAS_TRI), and crack filling and shading (CRK_SHD). RAS_TRI accounts for the majority of the frame time, with values ranging from 67% to 84% when the trimming test is on, and 46% to 63% when it is off. This is further supported by Table 6, which shows that RAS_TRI takes up 60% to 78% of the frame time for arrays of duplicated models. Therefore, RAS_TRI can be considered the main bottleneck. Despite TFE and EVAL taking up a



(a) Without trimming test



(b) With trimming test

Fig. 9. The breakdown of the frame time for each stage of the models. It shows the percentage of the total frame time allocated to each stage, including tessellation factor estimation (TPE), evaluation (Eval), rasterization (RAS), trimming test (TRI), crack filling (CRK), shading (SHD), and others.

similar amount of time, their combination never exceeds 65% of that of RAS_TRI. However, they may become bottlenecks in a hardware tessellation pipeline given the worse performance in comparison to ETER, as pointed out in the first paragraph of this section.

To further demonstrate the scalability and elasticity of ETER, we conducted experiments at 4K resolution with complex lighting integrated, as shown in Table 5. The total frame time at 4K (no shadow) increased by 56% to 79% compared to 1080p, with RAS_TRI remaining the major bottleneck. The variation in frame time increase is due to the proportion of flat surfaces within a model. We demonstrated the elasticity of our system by implementing PCSS [Fernando 2005] and HBAO+ [Bavoil et al. 2008]. To obtain a shadow map without

		TFE&Eval	RAS_TRI	CRK_SHD	Total
no shadow	Lego×1	3.156	22.138	0.322	26.623
	Car×15	5.062	29.137	0.273	36.215
	Turbine× 10	2.971	24.577	0.259	28.901
	Drone×250	3.092	20.272	0.334	24.653
shadow+AO	Lego×1	3.482	35.849	4.283	44.578
	Car×15	5.853	48.037	3.878	58.627
	Turbine× 10	3.581	44.532	4.457	53.548
	Drone×250	3.602	37.609	3.469	45.63

Table 5. Frame times (in ms) at 4K resolution with Shadow+AO enabled and disabled. When Shadow+AO is enabled, RAS_TRI includes shadow map computation, and CRK_SHD includes PCSS and HBAO+. The test scene contains a single directional light for shadow computation.

tessellation-induced cracks, we compute a pixel-accurate tessellation factor corresponding to the light source. Computation of the shadow map requires additional rasterization with respect to the light source, and patch-level culling must be disabled to ensure the accuracy of the shadow map. Therefore, the performance significantly declines for frames involving shadow map computation, as shown in Table 5, with RAS_TRI time increasing by 61% to 86%. However, in CAD applications, models and light sources remain static for most of the time, and shadow map computation only occurs when the scene changes. Notably, PCSS and HBAO+ (CRK_SHD of shadow+AO) have a minimal impact on the total frame time compared to RAS_TRI (no shadow).

To sum up, the ETER-based visibility buffer rendering pipeline has the capability to generate and rasterize over 0.2 billion triangles at a real-time frame rate (30FPS) on an RTX 3090 Ti GPU. Thanks to ETER, we are now able to handle depth complexity greater than 100 triangles per pixel, which is more than sufficient for practical CAD scenarios. As a result, a significant portion of GPU computing power can be spared and allocated for other rendering tasks such as material and lighting.

8 DISCUSSION AND FUTURE WORK

As our results show, after integrating the ETER framework into the rendering pipeline, NURBS surface evaluation is no longer a major performance bottleneck in the real-time rendering process, even though pixel-accurate rendering requires high-density tessellations. However, there are still two bottlenecks remaining in the rendering process after tessellation. The first is rasterization. Although software rasterization will provide better performance, it requires relatively high-density tessellation to generate small triangles. This may result in over-tessellation in some cases. For instance, a highly enlarged view of even flat surfaces will have unnecessarily high tessellation density. In the future, we would consider an approach that subdivides large triangles into small triangles, thus allowing us to use software rasterization under pixel-accurate tessellation factor. Additionally, our implementation of software rasterization uses 64-bit atomic operations in GPU memory for depth testing, and the performance could potentially be improved through the use of more

	1280×720					1920×1080				
	TFE & Eval	RAS_TRI	CRK_SHD	Total	Triangles	TFE & Eval	RAS_TRI	CRK_SHD	Total	Triangles
Lego×1	1.189, 1.568	6.527	0.157	9.765	71.88	1.165, 1.69	8.99	0.202	12.4	75.83
Lego×2	2.455, 3.044	11.072	0.204	17.15	140.63	2.821, 3.138	14.978	0.187	21.528	147.65
Lego×3	3.685, 4.417	13.745	0.169	22.419	205.23	4.132, 4.638	18.192	0.228	27.558	218.23
Car×15	2.569, 2.271	7.221	0.15	12.532	107.87	2.519, 2.297	12.151	0.176	17.5	111.04
Car×25	4.602, 3.821	12.773	0.175	21.718	178.73	4.877, 3.781	17.897	0.191	27.119	182.06
Car×35	6.838, 5.223	16.671	0.172	29.246	249.82	6.218, 5.189	23.581	0.194	35.567	251.75
Turbine×10	1.368, 1.293	9.631	0.169	12.857	61.64	1.421, 1.372	12.292	0.219	15.755	66.47
Turbine×20	3.223, 2.665	16.404	0.175	22.845	121.4	3.773, 2.729	24.146	0.204	31.25	152.49
Turbine×30	4.169, 3.836	23.215	0.169	31.796	181.27	4.277, 3.978	31.973	0.212	40.896	188.71
Drone×250	1.052, 1.703	6.015	0.196	9.285	69.95	1.067, 1.826	9.047	0.242	12.58	80.06
Drone×500	2.149, 3.349	10.613	0.239	16.679	138.57	2.891, 3.45	15.598	0.282	22.64	151.88
Drone×750	3.941, 4.751	15.621	0.268	24.958	207.78	3.862, 5.144	23.9065	0.382	33.725	228.91

Table 6. Frame time breakdown (in ms) and tessellated triangle count(in million) of multiple duplicated models when trimming test is enabled. By duplicating the models, we aim to determine the maximum number of Bézier patches and triangles that can be rendered in real-time on an RTX 3090 Ti GPU. The results of this study show that it is possible to achieve real-time frame rates while rendering millions of Bézier patches with triangle count up to 0.25 billion, despite the impact of trimming loop count and complexity. Note that all tests are performed in a view similar to isometric view.

efficient software rasterization techniques such as tiled-based rasterization [Laine and Karras 2011], due to better cache performance and the use of shared memory atomic operations. Another major performance bottleneck is trimming. Currently, we implement a modified version of the algorithm of [Schollmeyer and Froehlich 2018]. Based on our observations, a significant proportion of the surface area may be trimmed for an individual patch. Therefore, in the future, we may consider first identifying trimmed meshlets to bypass evaluation and the subsequent pipeline stages.

In CAD applications, a large proportion of surfaces have degrees equal to or less than 3. Therefore, when utilizing Tensor Cores to accelerate evaluation, using only 8×8 matrices will result in a waste of computation resources. Tensor Cores provide m8n8k4 instruction mode to accelerate the operations of 8×4 matrices, which can be used for surfaces of lower degrees. In addition, we upscale the tessellation factor of surfaces to at least 8×8, which is too high for surfaces that are far away from the viewpoint. To address this issue, in the future, we can consider 4×4 as the minimum sampling grid, or perform rasterization of pixel scale patches individually. CAD models may contain surfaces with degrees greater than 7. As suggested by [Strathaus 2008], in automotive design multi-span NURBS and surfaces with degrees more than 7 should be avoided to achieve Class-A quality, which is often considered one of the highest surface quality levels in engineering practice. Therefore, this work does not optimize Tensor Cores accelerated algorithm for surfaces with degrees greater than 7, but uses our optimized CUDA-only version for their evaluation instead. Considering the integrity of this work and to further tune the performance, Tensor Cores acceleration optimization for surfaces with degrees greater than 7 is worth further implementation.

Currently, we have not studied the issue of insufficient GPU memory in the case of extremely large models. To address this issue, a natural approach would be using memory swapping, a common memory management scheme in operating systems. Partial updates related to NURBS model editing during run-time can be achieved efficiently since our tessellation factor estimation algorithm is independent of adjacent patches. We have implemented frustum culling and back patch culling to improve efficiency. For complex assemblies or scenes, in the future, we plan to further integrate HZB (hierarchical z-buffer) occlusion culling and space-partitioning data structure (such as BVH or KD-tree) based culling techniques.

Limitations. ETER utilizes adaptive tessellation with uniform sampling grids, which may produce cracks. These cracks can be eliminated by the proposed crack-filling algorithm, but they do still impose two limitations. First, our current crack-filling method does not support order-independent transparency (OIT) algorithms that require the processing of multiple fragments in a single pixel. To our knowledge, there is no existing literature that discusses this issue. We intend to address the OIT support in future work. Second, our tessellated meshes are not water-tight, making them unsuitable for use in ray tracing scenarios. It is worth noting that trimmed NURBS models have intrinsic trimming-induced cracks that need to be addressed alongside tessellation-induced cracks to ensure water-tightness. We think that resolving these cracks in real-time tessellation without converting the models into approximated representations poses a challenging research direction for the future.

9 CONCLUSION

In this paper we present ETER, an elastic framework that enables real-time rendering of large-scale NURBS models with pixel-accurate and crack-free quality. Our contributions to designing ETER are prominent, that is, proposing three novel algorithms (i.e. a highly parallel pixel-accurate adaptive tessellation algorithm, a CUDA-based and Tensor Cores accelerated NURBS evaluation algorithm, an efficient visibility buffer and conservative rasterization-based crack-filling algorithm) and evaluating the performance of our framework by integrating into the rendering pipeline based on Mesh Shaders and hybrid software/hardware rasterization. In ETER, we observe an impressive speed-up of 2.9 to 16.2 times in Tensor Cores accelerated

evaluation compared with hardware tessellation and an unprecedented capability to render up to 3.7 million Bézier patches (0.25 billion tessellated triangles) in real-time (30FPS). Namely, with our ETER framework, adaptive tessellation is no longer the bottleneck. We disclose the significant impact of the software rasterization and the trimming test on the overall rendering performance. Moreover, it is more than reasonable to assume that our proposed evaluation algorithm whose huge performance superiority has been demonstrated, will continue to benefit from the ever-growing neural computing power. We are also convinced that ETER is of great value from a cost-effective perspective. At the time of writing, the latest midrange GPU RTX 4070 Ti is already equipped with greater computing power than RTX 3090 Ti. A foreseeable fact is that owing to ETER, real-time rendering of large-scale NURBS models with advanced visualization effects can be achieved without expensive high-end GPUs. Overall, we believe that ETER, with its unprecedented performance, inherent scalability and capability to achieve the highest visual quality, has the potential to become a fundamental building block in future CAD systems.

ACKNOWLEDGMENTS

We gratefully acknowledge Bay Raitt's "Big Guy", Martin Newell's "Utah Teapot", and GrabCAD community for the trimmed NURBS models. We would like to thank the anonymous reviewers for their constructive suggestions and comments. This work is partially supported by the National Key R&D Program of China (2022YFB3303400) and the National Natural Science Foundation of China (62025207). This research was done when Ruicheng Xiong was working as a part-time intern at Sheyun Technology.

REFERENCES

Jeremy Appleyard and Scott Yokim. 2017. *Programming Tensor Cores in CUDA 9.* Retrieved October 17, 2017 from https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. 2008. Image-Space Horizon-Based Ambient Occlusion. In *ACM SIGGRAPH 2008 Talks* (Los Angeles, California) *(SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article 22, 1 pages. https://doi.org/10.1145/1401032.1401061

Robert M. Blomgren and David J. Kasik. 2002. Early Investigation, Formulation and Use of NURBS at Boeing. *SIGGRAPH Comput. Graph.* 36, 3 (aug 2002), 27–32.

Christoph Buchenau and Michael Guthe. 2021. Real-Time Curvature-aware Re-Parametrization and Tessellation of Bézier Surfaces. In *Vision, Modeling, and Visualization*. The Eurographics Association.

Christopher A Burns and Warren A Hunt. 2013. The visibility buffer: a cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (2013), 55–69.

Frédéric Claux, Loïc Barthe, David Vanderhaeghe, Jean-Pierre Jessel, and Mathias Paulin. 2014. Crack-free rendering of dynamically tesselated B-rep models. *Computer Graphics Forum* 33, 2 (2014), 263–272.

Raquel Concheiro, Margarita Amor, Emilio J. Padrón, and Michael Doggett. 2014. Interactive rendering of NURBS surfaces. *Computer-Aided Design* 56 (2014), 34–44.

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*. Association for Computing Machinery, New York, NY, USA, 95–102.

Christopher Dyken, Martin Reimers, and Johan Seland. 2009. Semi-Uniform Adaptive Patch Tessellation. *Computer Graphics Forum* 28, 8 (2009), 2255–2263.

Christian Eisenacher, Quirin Meyer, and Charles Loop. 2009. Real-Time View-Dependent Rendering of Parametric Surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. Association for Computing Machinery, New York, NY, USA, 137–143.

Randima Fernando. 2005. Percentage-Closer Soft Shadows. In *ACM SIGGRAPH 2005 Sketches* (Los Angeles, California) *(SIGGRAPH '05)*. Association for Computing Machinery, New York, NY, USA, 35–es. https://doi.org/10.1145/1187112.1187153

Daniel Filip, Robert Magedson, and Robert Markot. 1986. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design* 3, 4 (1986), 295–311.

Stepan Yu. Gatilov. 2016. Vectorizing NURBS surface evaluation with basis functions in power basis. *Computer-Aided Design* 73 (2016), 26–35.

Michael Guthe, Aákos Balázs, and Reinhard Klein. 2005. GPU-Based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Trans. Graph.* 24, 3 (jul 2005), 1016–1023.

Ulrich Haar and Sebastian Aaltonen. 2015. Gpu-driven rendering pipelines. *Lecturer on SIGGRAPH* (2015).

Jon Hjelmervik. 2014. Direct pixel-accurate rendering of smooth surfaces. In *Mathematical Methods for Curves and Surfaces*. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–247.

Jorge Jimenez, Jose I. Echevarria, Tiago Sousa, and Diego Gutierrez. 2012. SMAA: Enhanced Subpixel Morphological Antialiasing. *Comput. Graph. Forum* 31, 2pt1 (may 2012), 355–364.

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. In *ACM SIGGRAPH*, Vol. 21.

Adarsh Krishnamurthy, Rahul Khardekar, and Sara McMains. 2009. Optimized GPU evaluation of arbitrary degree NURBS curves and surfaces. *Computer-Aided Design* 41, 12 (2009), 971–980.

Christoph Kubisch. 2018. *Introduction to Turing Mesh Shaders.* Retrieved September 17, 2018 from https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/

Subodh Kumar, Dinesh Manocha, and Anselmo Lastra. 1995. Interactive Display of Large-Scale NURBS Models *(I3D '95)*. Association for Computing Machinery, New York, NY, USA, 51–ff.

Samuli Laine and Tero Karras. 2011. High-Performance Software Rasterization on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (Vancouver, British Columbia, Canada) *(HPG '11)*. Association for Computing Machinery, New York, NY, USA, 79–88.

Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. 2010. FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (Washington, D.C.) *(I3D '10)*. Association for Computing Machinery, New York, NY, USA, 75–82. https://doi.org/10.1145/1730804.1730817

Stephen Mann and Tony DeRose. 1995. Computing values and derivatives of Bézier and B-spline tensor products. *Computer Aided Geometric Design* 12, 1 (1995), 107–110.

McNeel. 2020. *Rhino Mesh Settings - Detailed Info.* Retrieved August 14, 2020 from https://wiki.mcneel.com/rhino/meshfaqdetails

M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer. 2016. Real-Time Rendering Techniques with Hardware Tessellation. *Computer Graphics Forum* 35, 1 (2016), 113–137.

Hiroyuki Ootomo and Rio Yokota. 2022. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications* 36, 4 (2022), 475–491.

Les Piegl and Wayne Tiller. 1997. *The NURBS Book (2nd Ed.).* Springer-Verlag, Berlin, Heidelberg.

David F Rogers. 2001. *An introduction to NURBS: with historical perspective*. Morgan Kaufmann.

Andre Schollmeyer and Bernd Froehlich. 2018. Efficient and anti-aliased trimming for rendering large NURBS models. *IEEE Transactions on Visualization and Computer Graphics* 25, 3 (2018), 1489–1498.

Andre Schollmeyer and Bernd Fröhlich. 2009. Direct Trimming of NURBS Surfaces on the GPU. *ACM Trans. Graph.* 28, 3 (jul 2009).

Michael Schwarz and Marc Stamminger. 2009. Fast GPU-based Adaptive Tessellation with CUDA. *Computer Graphics Forum* 28, 2 (2009), 365–374.

Leon A. Shirmun and Salim S. Abi-Ezzi. 1993. The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum* 12, 3 (1993), 261–272.

Werner Strathaus. 2008. *Building a Sportscar Exterior to Class-A Surfacing Standards*. Autodesk, Inc., San Rafael, CA 94903, USA.

Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2022. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numerical Behaviors. *arXiv preprint arXiv:2206.02874* (2022).

Dassault Systèmes. 2023. *SOLIDWORKS Help.* https://help.solidworks.com/2023/English/SolidWorks/sldworks/HIDD_OPTIONS_IMAGE_QUALITY_display.htm?verRedirect=1

Young In Yeo, Lihan Bin, and Jörg Peters. 2012. Efficient Pixel-Accurate Rendering of Curved Surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Association for Computing Machinery, New York, NY, USA, 165–174.

Jianmin Zheng and Thomas W. Sederberg. 2000. Estimating Tessellation Parameter Intervals for Rational Curves and Surfaces. *ACM Trans. Graph.* 19, 1 (jan 2000), 56–77.