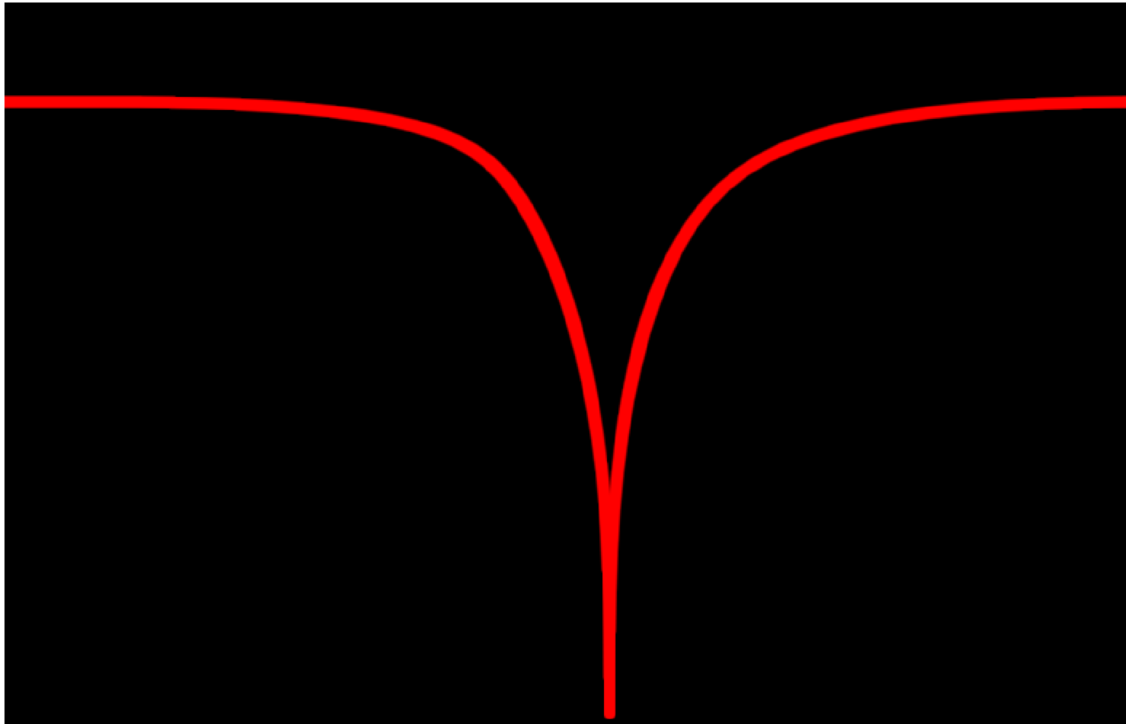
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Efficient Path Rendering For Low-End GPUs

Master's thesis in Computer science and engineering

Linus Wallman

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Efficient Path Rendering For Low-End GPUs

Linus Wallman

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Efficient Path Rendering For Low-End GPUs
Linus Wallman

© Linus Wallman, 2025.

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisor: Daniel Gunnarsson, Kollmorgen
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Efficient Path Rendering For Low-End GPUs
Linus Wallman
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern discrete graphics cards are powerful, and succeed in massively rendering paths for varying domains, one being path-based layouts for automated guided vehicles (AGVs). However, integrated GPUs (iGPUs) are not as powerful as discrete graphics cards, and therefore there is a need for algorithms to accelerate the process of massively rendering paths on iGPUs. This thesis presents a method to massively render paths, more specifically B-spline curves, efficiently and scalably on iGPUs. Instead of rasterizing the polylines, we determine which pixels should be colored based on their intersections with line segments sampled from B-spline curves. We show that our method can massively render paths efficiently and scalably, but only for static layouts. To extend our method to dynamic scenarios, we suggest using a linear quad tree or AABB tree instead of a uniform grid.

Keywords: Computer science, computer graphics, path rendering, integrated GPU.

Acknowledgements

Firstly, I would like to give a thank you to my supervisor, Erik Sintorn, for his invaluable guidance in shaping this thesis and for his support throughout the project.

Secondly, I am grateful for Daniel Gunnarsson and Henrik Persson at Kollmorgen for making this thesis possible. Furthermore, I would like to give an additional thank you to my company advisor, Daniel Gunnarsson, for his guidance throughout this project.

Lastly, I would also like to thank my colleagues at Kollmorgen, including Ali Majidi, Mohammadamin Mohammadi, and the rest of Purple Team for their stimulating discussions, support, encouragement, and for making the work environment enjoyable.

Linus Wallman, Gothenburg, 2025-06-10

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Limitations . . . 3
  - 1.3.1 Paths . . . 3
  - 1.3.2 Dynamic Layouts . . . 3

2 Previous Work 5
- 2.1 CPU and GPU Path Rendering . . . 5
- 2.2 Path Rendering in Vector Graphics . . . 5
- 2.3 Rendering Paths Using Hardware Tessellation . . . 6
- 2.4 Contribution . . . 6

3 Theory 9
- 3.1 Graphics Pipeline . . . 9
- 3.2 B-Spline . . . 10
  - 3.2.1 B-Spline Basis Function . . . 10
  - 3.2.2 Cox-De Boor Recurrence Basis Function Construction . . . 11
  - 3.2.3 Varying B-Spline Behavior via Knot Vector . . . 12
  - 3.2.4 Cubic B-Spline Curve . . . 12
- 3.3 Hardware Tessellation . . . 13
  - 3.3.1 Tessellation Shaders . . . 13
- 3.4 Acceleration Structures . . . 14

4 Method 15
- 4.1 Overview . . . 15
- 4.2 Tessellation Shader Implementation . . . 16
  - 4.2.1 Dynamic Adaptive Sampling . . . 16
  - 4.2.2 Sampling in Tessellation Evaluation Shader . . . 17
  - 4.2.3 View Frustum Culling . . . 17
  - 4.2.4 Pre-Pass Compute Shader Implementation . . . 18
  - 4.2.5 B-Spline Bounding Box . . . 19

4.3 Per-Pixel Intersection Implementation 20
4.3.1 Uniform Grid 21
4.3.2 Sampling 21
4.3.3 Counting Segments in Grid Cells 21
4.3.4 Computing Prefix Sum 22
4.3.5 Binning 22
4.3.6 Per-Pixel Intersection Testing 22

5 Results **25**
5.1 Environment and Hardware Setup 25
5.2 Datasets 26
5.3 Performance 27
5.3.1 Tessellation Shader Implementation Performance 29
5.3.2 Per-Pixel Intersection Testing Implementation Performance 29
5.4 Path Appearance 31

6 Conclusion **33**
6.1 Discussion 33
6.1.1 Dynamic Layouts 33
6.1.2 Anti-Aliasing 34
6.2 Ethics and Risk Analysis 34
6.2.1 Intellectual Property 34
6.2.2 Conflict of Interest 34
6.2.3 Social and Environmental Responsibility 34
6.3 Future Work 34
6.4 Conclusion 35

Bibliography **37**

List of Figures

1.1 Example of a B-spline curve with 8 control points representing a path. 1

3.1 Overview of OpenGL's rendering pipeline. Dashed lines indicate optional stages. Green stages are programmable, and yellow are non-programmable. 9
3.2 B-Spline parametric equation. 10
3.3 Four B-Spline basis functions of third degree with a domain defined in the normalized knot vector span $t \in [0, 1]$. 10
3.4 Expanded recursion tree for the basis function $N_{0,4}$. 11
3.5 **Left**: Open uniform B-Spline curve. **Middle**: Clamped uniform B-Spline curve. **Right**: Clamped closed uniform B-Spline curve. 12
3.6 Cubic B-Spline curve with 4 control points. 12
3.7 Tessellation stages in the rendering pipeline. 13

4.1 Tessellation Evaluation Shader B-spline segment sampling implementation. 17
4.2 View frustum culling compute shader pseudocode. 18
4.3 Overview of compute shader dispatch flow. A four phase single-dispatch is executed in the order depicted in the figure. Then, using the result from the binning step, pixel's resulting color are set. 20
4.4 Pseudocode to convert world positions to grid cell indices. 21
4.5 Pseudocode to increment the number of segments per grid cell. 22
4.6 Naive parallel prefix sum pseudocode. 22
4.7 Screen-space to world space computation. 23

5.1 Dataset $\mathcal{D}_1$, rendered using our proposed per-pixel intersection testing method. 26
5.2 Dataset $\mathcal{D}_2$, rendered using our proposed per-pixel intersection testing method. 26
5.3 Plot with four different zoom-levels showcasing how performance is increased when non-visible paths are culled. The four samples showcased in the figure are from the dataset, $\mathcal{D}_1$. 28
5.4 View frustum culling plotted alongside tessellation and rendering times as a function of input size. 29
5.5 Four phase compute initialization comparison for datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. 30

5.6 Paths rendered using per-pixel method. **Left**: smooth lines by calculating coverage and fading depending on distance to pixel. **Right:** non-smooth lines. 31
5.7 Clear visual aliasing and jaggedness rendered using per-pixel intersection testing method.**Left**: Jaggedness and aliasing in non-smoothing/fading solution. **Right**: Less visual aliasing by smoothing and fading edges. 31

List of Tables

5.1 Maximum number of B-spline curves rendered at 60 FPS across two datasets, $\mathcal{D}_1$ and $\mathcal{D}_2$, using the two proposed methods. Per-pixel intersection enables rendering more paths at the expense of higher memory usage. 27
5.2 Per-pixel intersection test performance under varying grid occupancy using dataset $\mathcal{D}_1$. 30
5.3 Per-pixel intersection test performance under varying grid occupancy using dataset $\mathcal{D}_2$. 30

1

Introduction

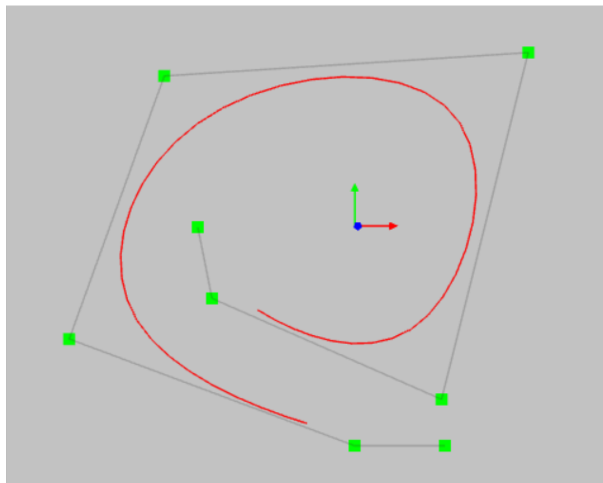


Figure 1.1: Example of a B-spline curve with 8 control points representing a path.

In AGV (Automated Guided Vehicle) applications, application engineers specialize in designing path-based layouts that define vehicle routes within logistics environments such as warehouses. These layouts rely on B-spline curves, as shown in Figure 1.1, to model smooth and precise paths. Since layout visualization often involves interactive panning and zooming during design, real-time performance is essential. The system must remain efficient and scalable to prevent bottlenecks that could hinder usability.

While rendering a few high-quality paths with a small number of control points can be performed efficiently on the CPU, the computational demand increases significantly as the number of paths and control points grows. In such cases, leveraging the GPU becomes essential to achieve real-time performance.

In this thesis, we investigate how to massively render paths, more precisely, cubic basis-splines (B-Splines), efficiently and scalably. We present a new method that can massively render smooth curves efficiently using the iGPU. As a comparison, a naïve method of rendering paths is implemented using tessellation shaders. Furthermore, throughout this report, the terms path and B-spline curve are used interchangeably.

1.1 Motivation

There is currently no hardware support for rendering paths on the GPU. However, there exist several acceleration techniques using the GPU to render paths efficiently (see Chapter 2). Unfortunately, application engineers designing layouts do not typically have access to discrete graphics cards that these techniques assume. They do, however, have iGPUs that we could target, but the acceleration techniques presented in Chapter 2 are not suitable for iGPUs due to limited memory bandwidth, compute power, and available memory. This motivates the investigation into how to maximize iGPU utilization in massively rendering paths.

1.2 Problem Statement

This thesis aims to answer the following question: How are paths rendered efficiently and scalably on iGPUs? To answer that question, the following sub-questions will be explored:

- What are the current limitations of iGPUs in efficiently rendering paths?
- How can we maximize GPU utilization in efficiently rendering paths, and what techniques can be adapted, combined, or improved to enhance path rendering on iGPUs?

1.3 Limitations

1.3.1 Paths

We only consider constructing paths using cubic B-splines in this thesis. Typically, cubic degree NURBS, Bézier or B-splines are sufficient to describe paths given their continuity properties, simplicity, affine transformation invariance, and more [1]. Arbitrary degree NURBS offer more generality, however, such as being able to accurately model conic objects [2] which is crucial in CAD tools.

1.3.2 Dynamic Layouts

In this thesis, we only consider visualizing paths, and not the scenario where new paths are created or existing paths changed. However, the use case of how to efficiently visualize paths is still interesting, and therefore propose how to extend or improve our current method towards a full solution allowing dynamic layouts. This is further discussed in the discussion section, and methods of addressing this issue are proposed.

2

Previous Work

Path rendering is an active research area, often with a focus on vector graphics and CAD-related tools. In this chapter we motivate and describe how our contribution fits into the research of path rendering.

2.1 CPU and GPU Path Rendering

Gatilov [3] shows how to vectorize NURBS curves on the CPU using Horner's rule to reduce additions and multiplications when evaluating polynomials. Additionally, Gatilov states that pre-computing blending between control points and basis functions significantly increases performance. While vectorization on the CPU itself falls outside our scope, identifying other sources of speed-up is relevant to our purpose; as Gatilov suggests, pre-computing blending between control points and basis functions is beneficial as we seek to improve performance in the evaluation of B-spline curves.

Path rendering is not limited to the CPU. Krishnamurthy et al. present a method for optimizing the evaluation of NURBS curves on the GPU [4]. In their paper, they show how to efficiently evaluate arbitrary degree NURBS curves by packing the knot values and basis functions into textures. By storing the knot values in the four channels (RGBA) and basis functions in the height direction based on the degree of the curve, they can efficiently and uniformly calculate any degree with one unified fragment shader. However, this method requires building these textures on the CPU in a pre-processing step, which is computationally expensive and memory-intensive.

2.2 Path Rendering in Vector Graphics

Loop and Blinn [5] propose a method to render resolution independent curves on the GPU, which is later adapted and used by the same authors to render vector art [6]. Typically, one either tessellates the path representation into triangles or takes sufficient samples to render paths as polylines. Loop and Blinn propose to instead exploit that the Bézier curve lies inside of the convex hull region created by its control points. The control points are then used as vertices of a triangle and a set of texture coordinates are mapped to its vertices. In the fragment shader they evaluate $f(u, v) = u^2 - v < 0$ and consider a pixel to be inside the curve if the

equation is true. To achieve anti-aliased curves, they approximate a signed distance function in the fragment shader. They then compute the pixel's alpha value by either looking up in a precomputed texture with alpha values based on the distance or calculate a blending function using the distance on the fly.

Santina [7] uses a similar approach to Loop and Blinn [5] to render paths, but for NURBS curves. Additionally, Santina converts the curved parts of the path to triplets of vertices, where two of the vertices are on the curve, and the third is not. The remaining parts of the paths are triangulated using a modified Delaunay triangulation. As the number of paths grow, this method will suffer from potential overdraw with expensive per-fragment operations. Furthermore, the method fails to scale as the number of paths increase — for every segment, they will generate one triangle, and paths typically consist of more than one segment. Thus, rendering 1 million paths requires generating at least 1 million triangles.

Kilgard and Bolz [8] suggest an alternative method called *Stencil, then Cover* to render filled or stroked paths. Contrary to other methods, they do not use any intersection tests to determine coverage; instead these are implicit via stencil buffer usage.

Batra et al. [9] build on the work of Kilgard and Bolz [8] where they present a method using hardware tessellation to shade paths and a novel method of repurposing RGBA buffers to render CMYK colors.

Scanline rasterization is another method of rendering paths introduced by Li et al. [10]. They propose a GPU scanline approach that first generates boundary fragments (pixels on Bézier or segment boundaries); first for all monotonic segments, and then for all curved segments. These are then sorted by path index, merged per pixel that is covered by the same path, and lastly rendered with a coverage mask.

2.3 Rendering Paths Using Hardware Tessellation

Kumar and Sud presents a method using tessellation shaders to render strokes [11]. In their proposed method they instrument the tessellation shader to output polylines representing cubic Bézier curves, and then approximate these as quadratic Bézier curves in the geometry stage by pairing together successive points from the polyline. Then to render the strokes they use the same method as proposed by Nehab and Hoppe [12].

2.4 Contribution

We aim to render a very large number of paths (on the order of hundreds of thousands) at real-time frame rates. Although modern desktop CPUs often have 8 cores, 16 threads, and support SIMD (Single Instruction Multiple Data) vectorization, even a modest iGPU typically exposes 32 or more execution units (EUs) with wide vector pipelines [13]. As a result, when rendering hundreds of thousands of paths or more, a CPU-only approach becomes infeasible: it simply cannot match the amount of

parallel threads that an iGPU can launch. In contrast, for scenes with only a small number of paths, the CPU may still be faster due to higher clock frequencies.

Pre-computing the blending between control points and basis functions, as concluded by Gatilov [3], yields negligible performance gain for NURBS curves when sampling the curves on the GPU, which Gatilov also concluded for the CPU.

In this thesis, we focus on cubic B-splines, which is one reason why the method proposed is not suitable [4]. We want to reduce bandwidth and achieve that by exploiting that the knot values are uniform and repeated per B-spline in our case, and the basis polynomials are constant since we only consider cubic B-splines.

Paths in CAD-related tools are rarely changed extensively and could therefore be considered mostly static. Therefore, our contribution proposes to use a technique where we, instead of using any of the techniques mentioned above, decide each pixel's color directly by testing if a pixel is on any of all sampled line segments. Consequently, we are able to render one million paths in 60 frames per second.

3

Theory

3.1 Graphics Pipeline

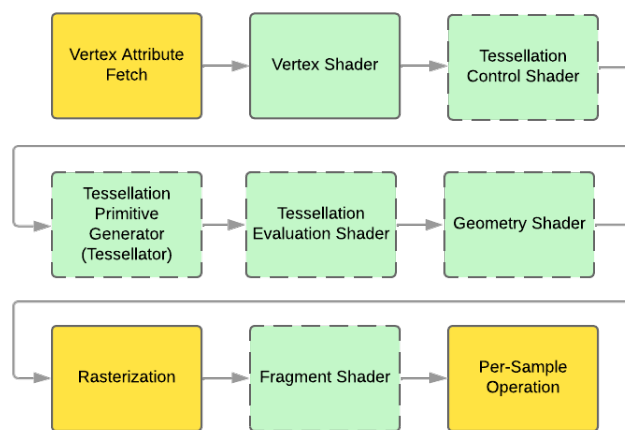


Figure 3.1: Overview of OpenGL's rendering pipeline. Dashed lines indicate optional stages. Green stages are programmable, and yellow are non-programmable.

The graphics pipeline consists of a set of stages that primitives, often triangles, pass through when requested to be rendered using a graphics API such as OpenGL [14] to generate an image displayed on the screen. Every stage plays a core role in the pipeline with its own responsibilities, initiated by a draw call on the CPU side, and subsequently processed in parallel in the order depicted in Figure 3.1. These stages consist of both programmable and non-programmable stages, where the programmable ones are called shaders, which are programs that are executed on the GPU [15].

The core idea with having a graphics pipeline is to enable high-performance, but also to provide a programming model that enables programmers to configure the pipeline to suit their needs, instead of only providing fixed-function hardware [16]. There are stages that are fixed-function hardware and non-programmable, as shown in Figure 3.1. One example is the rasterization stage where inputted geometry is sent to the rasterizer which computes which fragments are covered by the inputted geometry [15].

3.2 B-Spline

$$\mathbf{C}(t) = \sum_{i=0}^{n} N_{i,p}(t)\mathbf{P}_i$$

Figure 3.2: B-Spline parametric equation.

A basis-spline (B-Spline) is a spline, which is a way to mathematically model a smooth curve that is determined by a set of polynomials partitioned over the sub-domains of the spline domain [1]. The partitioning is defined by a set of values, called the knot vector, a sequence of knots with uniform spacing which subdivide the domain. Formally, the knot vector is defined as:

$$T = \{t_0, t_1, \ldots, t_m\}, \quad t_0 \leq t_1 \leq \cdots \leq t_m.$$

$$m = n + d + 1.$$

where **d** is the degree of the B-Spline, and **n** is the number of control points.

3.2.1 B-Spline Basis Function

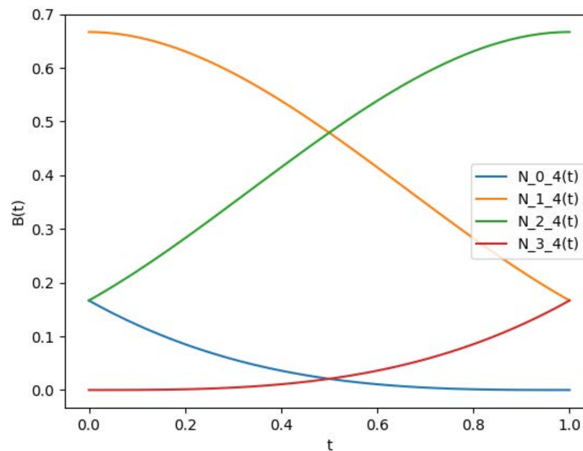


Figure 3.3: Four B-Spline basis functions of third degree with a domain defined in the normalized knot vector span $t \in [0, 1]$.

In Figure 3.2 the parametric equation for B-Splines is depicted, where $N_{i,p}(t)$ is a basis function, and $\mathbf{P}_i$ is a control point. This is expressed as a linear combination, where the basis function acts as a weight that is blended with every control point [1]. Basis functions are polynomials with degree $p - 1$ (p is the order of the B-Spline) that are non-zero in the knot span $[t_i, t_{i+d+1})$. For this reason, moving a control point affects only a local part of the B-Spline.

B-Splines have the following properties: (1) local support, (2) linear independence, (3) C^{d-1} continuity, and (4) partition of unity [1]. Cubic B-Splines are typically used

because they have C^2 continuity, and thus are smooth at every joint between the polynomial curves. Partition of unity means that:

$$\mathbf{C}(t) = \sum_{i=0}^{n} N_{i,p}(t) = 1$$

on the knot span $[t_i, t_{i+1})$. Consequently, every B-Spline is contained by the convex hull created by its control polygon [2].

Rendering a B-Spline is done by first decomposing it into polylines (line segments). Decomposing the B-Spline into polylines is typically done by sampling the parametric space with sufficiently high sampling rate to draw a good looking B-Spline [2].

3.2.2 Cox-De Boor Recurrence Basis Function Construction

$$N_{i,0}(t) = \begin{cases} 1 & \text{if } t_i \leq t < t_i + 1 \\ 0 & otherwise \end{cases}$$
$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t) \tag{3.1}$$

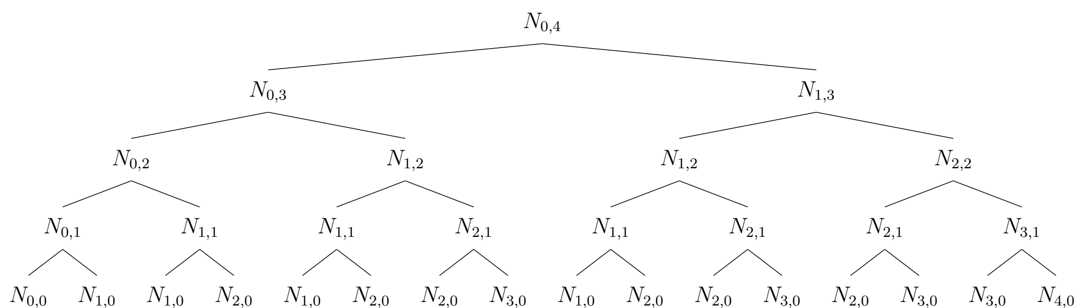


Figure 3.4: Expanded recursion tree for the basis function $N_{0,4}$.

Basis functions can be constructed for arbitrary degrees using Cox-De Boor recursion formula 3.1 in an iterative top-down fashion by considering the recursion tree [2], [17] shown in Figure 3.4. For example, to evaluate $N_{0,3}$ we first have to evaluate $N_{0,2}$ and $N_{1,2}$ recursively until we reach the the leaf nodes.

As shown in Figure 3.3, the base case, $N_{i,0}(t)$, is a step function defined over the knot span. In the general case, $N_{i,p}(t)$, is a recursive linear interpolation of interpolations [2].

3.2.3 Varying B-Spline Behavior via Knot Vector

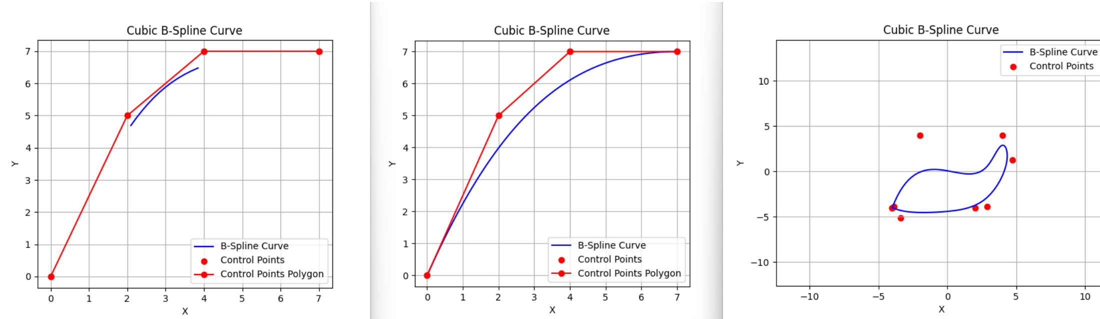


Figure 3.5: **Left**: Open uniform B-Spline curve. **Middle**: Clamped uniform B-Spline curve. **Right**: Clamped closed uniform B-Spline curve.

There are three variants of B-Splines that can be constructed using different knot vectors: open, clamped and closed form, shown in fig 3.5. An open uniform B-Spline curve can be constructed by distributing knot values with uniform spacing over the parametric domain, and the clamped uniform B-Spline curve is constructed by repeating knot values in the knot vector such that:

$$t_0 = t_1 = \cdots = t_{k-1}, \qquad t_{n-k+1} = t_{n-k+2} = \cdots = t_n.$$

holds. Clamped closed uniform B-Spline curves are loops, meaning the start and end points are joined. This is typically achieved by repeating start control points at the end of the B-Spline [2].

3.2.4 Cubic B-Spline Curve

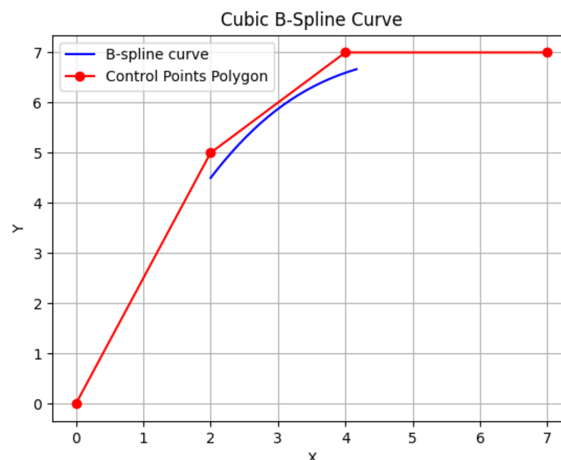


Figure 3.6: Cubic B-Spline curve with 4 control points.

Cubic B-Spline curves are typically sufficient for most computer aided design related modeling [1]. By normalizing the parameter $t \in [0, 1]$ and analytically computing the basis matrix, we then arrive at the resulting equation shown in Equation 3.2.

$$C(t) = \frac{1}{6}[\,t^3\,t^2\,t\,1\,]\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 0 & 3 & 0\\ 1 & 4 & 1 & 0\end{bmatrix}\begin{bmatrix}\mathbf{P_0}\\ \mathbf{P_1}\\ \mathbf{P_2}\\ \mathbf{P_3}\end{bmatrix} \tag{3.2}$$

Furthermore, to model a B-Spline curve consisting of more than one B-Spline segment, we exploit that B-Splines can be modeled as a linear combination of open uniform B-Splines because of its linear independency property [1]. Thus, we model every segment as eq. 3.3 where every B_j corresponds to the j:th row in the basis matrix as shown in Figure 3.2.

$$S_i(t) \;=\; \sum_{j=0}^{3} B_j(t)\,\mathbf{P}_{i-2+j}\,, \qquad t \in [0,1] \tag{3.3}$$

3.3 Hardware Tessellation

Tessellation in GPUs is the process of taking simple geometry and dynamically deciding level of detail using fixed function hardware on the GPU. Furthermore, we can for example, input a B-Spline curve description to the tessellation stages, sample the parametric equation 3.2 and thereby avoid storing the sampled points in memory. Instead, these are passed on through the pipeline as they are processed [15].

Hardware tessellation effectively reduces memory bus transactions between CPU and GPU in the case where e.g., a curve should have more detail when more of it is visible, while it should have less detail when less of it is visible. This is because for every frame, the CPU would have to compute level of detail given for example distance, and transfer it to the GPU. Using hardware tessellation, or tessellation shaders as described in detail in Section 3.3.1, the programmer can compute the distance and dynamically change level of detail [16].

3.3.1 Tessellation Shaders

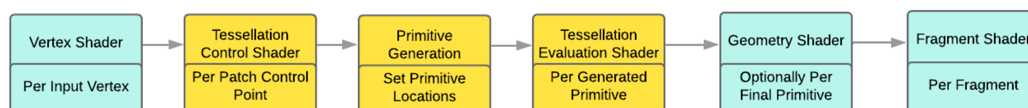


Figure 3.7: Tessellation stages in the rendering pipeline.

Existing graphics APIs expose access to the tessellation stages of GPUs in the same way, but with different names [15]. The names for every stage depicted in Figure 3.7 are the names used in OpenGL's API [14].

The tessellation stages operate on abstract patches, configured by a specified number of vertices. Every vertex in every patch is processed in the vertex shader, and inputted into the Tessellation Control Shader (TCS) where per patch operations

are done. It is in this stage that the programmer can define logic to dynamically determine level of detail by modifying built in variables. The outputted variables are then used by the fixed-function hardware tessellator to generate primitives that the Tessellation Evaluation Shader (TES) operates on [16].

3.4 Acceleration Structures

Performing intersection testing can be computationally expensive when dealing with detailed or complex geometry. Naively performing intersection testing means comparing every object in a scene against every other object. This translates to a time complexity of $O(n^2)$, and for large n, this becomes infeasible for even the cheapest intersection testing such as an AABB vs AABB test [15]. However, there are ways to reduce the number of needed intersection tests. Spatially structuring data reduces the number of necessary tests because we only have to compare the objects that are spatially grouped together. For example, in a tree data structure this would be equivalent to only checking a node against the sub-tree that it is in. This effectively reduces the necessary number of tests to a time complexity of $O(log\, n)$ [18].

There exist many algorithms and data structures to spatially organize data, one being trees and grid. If we divide the whole scene into a 2D grid and then map every object in the scene onto a grid cell, then we only have to check every object against all others objects in the same cell [18].

4 Method

In this chapter, we present two methods used to address the research questions outlined in Section 1.2.

4.1 Overview

We first describe an implementation of rendering B-spline curves, using tessellation shaders in OpenGL. We identify two primary issues with this implementation: first, the GPU's hardware-imposed tessellation factor prevents us from simplifying to arbitrary numbers of generated isolines, something that is necessary to achieve high performance while keeping sufficient visual quality. More critically, we notice that the throughput is the main bottleneck. For example, 250k cubic B-spline curves consist of at least one million control points, which translates to 250k abstract patches. To overcome these limitations, we propose a second method which we call per-pixel intersection algorithm, in which each pixel's final color is determined by testing pixel intersections against polylines sampled from B-spline curves. More importantly, the spline segments are only sampled once in a pre-render loop phase, which reduces the problem into figuring out which pixels are on the resulting polylines. To further reduce the number of intersection tests, we introduced a uniform grid acceleration structure, which significantly reduces the number of intersection tests.

4.2 Tessellation Shader Implementation

We render B-spline curves as one pixel wide polylines on the GPU by using a modern OpenGL technique called indirect rendering together with OpenGLs tessellation stages. Indirect rendering is a concept in OpenGL and other graphics APIs where instead of creating draw calls for every mesh or object, we instead batch draw commands. As a benefit, the driver is not throttled by draw commands and can instead determine how to schedule them internally.

The tessellation shader implementation approach has two key benefits:

1. Reduced memory usage and memory bandwidth, and
2. dynamic control of subdivision on the GPU, effectively reducing CPU-GPU memory transactions.

We group control points into the abstract path primitives expected by the Tessellation control Shader (TCS). Since it is not known how many control points there are per B-spline curve, we define every patch as 4 vertices (4 control points). This means that we effectively process one B-spline segment per Tessellation Evaluation Shader (TES) invocation.

4.2.1 Dynamic Adaptive Sampling

When rendering the paths, we want to render at high quality while keeping maintaining high performance. If we take too few samples with respect to the path's coverage in screen-space, then the rendered path will have visible edges. On the other hand, if we take too many samples, we lose performance. Therefore, we adaptively determine a tessellation factor, analogous to determining how many samples to take per B-spline segment, depending on its coverage in screen-space.

To calculate the coverage, we considered three alternatives: use a rough estimate by either considering the area of the convex hull created by the control points, calculating the total distance between every consecutive control point, or by taking a sufficient number of samples and computing the distance similar as to the second alternative.

We tested, and realized that sampling a B-spline is fast, and also provides a more accurate representation of the polyline considering the pixel coverage than the other two alternatives. We therefore use the third alternative: take 5 uniformly-spaced samples per segment ($t \in 0.0, 0.25, 0.5, 0.75, 1.0$), project them to clip space, and accumulate the distance between consecutive samples. Finally, we clamp that accumulated value to the range [1, 64]. All this is done once in the TCS per patch primitive.

4.2.2 Sampling in Tessellation Evaluation Shader

The TES is straightforward in the sense that we only need to implement a simple algorithm that samples Equation 3.2. The parameter t in the Equation. 3.2 is passed to the TES from the TCS signifying where along the B-spline segment we should sample.

Figure 4.1: Tessellation Evaluation Shader B-spline segment sampling implementation.

```
1  coefficient = 1.0 / 6.0
2  cubic_basis_polynomials = 1.0 / 6.0 * mat4(
3      -1, 3, -3, 1,
4      3, -6, 3, 0,
5      -3, 0, 3, 0,
6      1, 4, 1, 0
7  );
8
9  sample_bspline(control_points, t)
10     // Compute the coefficients
11     ts = [t*t*t, t*t, t, 1.0]
12     basis = basis_polynomials * ts
13     // Multiply basis function with control points vector
14     x = dot(basis, points)
15     y = dot(basis, points)
16     return x, y
```

4.2.3 View Frustum Culling

We only want to render visible curves. To do so, we employed a view frustum culling pre-pass compute shader where we build a buffer of indirect draw commands based on visible paths. View frustum culling is a culling technique, meaning non-visible objects are not processed during the rendering step for every frame. Since we use orthographic projection, the view frustum will actually be a box instead of a cone shape.

To cull paths we also needed a container to do intersection tests against. For this, we exploited that B-splines are contained by the convex hull created by its control polygon. We create these in a first step when all control points are loaded into the program. We also saved metadata for every B-spline curve. This metadata contains the number of control points, an index into the control point buffer, and lastly a bounding box containing the B-spline curve.

4.2.4 Pre-Pass Compute Shader Implementation

Figure 4.2: View frustum culling compute shader pseudocode.

```
1
2  // Assume SSBO definitions of atomic_index, spline_metadata_buffer,
   ↪  and indirect_command_buffer.
3
4  // Fetch index pointing into spline curve metadata buffer
5  curve_id = gl_GlobalInvocationID.x
6
7  // Ensure curve id is within the buffer range
8  if curve_id >= len(spline_metadata_buffer)
9      // if outside, exit.
10     return
11
12 //  Load B-spline curve metadata
13 curve = spline_metadata_buffer[curve_id]
14
15 // Check if the B-spline curve associated AABB is within the
   ↪  viewport
16 if visible(curve.abbb)
17     // If so, use an atomic counter to find
18     // where in the indirect command buffer
19     // to write the indirect command.
20     index = atomicInc(atomic_index)
21     // Build an indirect command from curve metadata
22     cmd = create_indirect_command(curve)
23     // .. and write it to the indirect command buffer
24     indirect_command_buffer[index] = cmd
```

We implemented a pre-pass frustum culling in a compute shader shown in Figure 4.2 that is executed every frame before rendering. We assign a GPU thread per B-spline curve to render, and check if the curve is visible by testing if its AABB is contained within the viewport by AABB vs AABB intersection testing [18]. Since all GPU threads write to the same buffer, and possibly same indices if a counter is shared, we use an atomic counter to ensure that there are no race conditions. A different approach could be to use shared memory, and allocate a range of indices assigned for the thread block sharing memory.

4.2.5 B-Spline Bounding Box

We save memory by computing a single bounding box per B-spline curve, and not per B-spline segment. A B-spline curve can consist of a variable amount of B-spline segments. For our data set, it averages to 8 B-spline segments per B-spline curve. We could compute the bounding boxes on the fly and thereby reduce memory usage, but chose instead to compute and store only a single bounding box. Unfortunately, this leads to some extra B-spline segments being rendered even if not visible. However, this is not the main bottleneck with this technique, because it is a common scenario that all curves are visible at once.

We populated both the control point buffer and the index buffer by reading control points from a file exported from an application where B-spline curves are used to model paths. The indirect command buffer was populated in a pre-pass frustum culling compute shader where only visible B-spline curves were written to the indirect command buffer. We could not figure out a different way to implement this compute shader without having an additional buffer, a metadata buffer, containing an index to where the first control point is, the number of control points, and lastly a bounding box. We construct the bounding box by exploiting that every B-spline curve is contained within the convex hull of its control points, as described in Section 3.2. This does not create a perfectly tight fit, but we realized empirically that it is sufficient for our purpose.

4.3 Per-Pixel Intersection Implementation

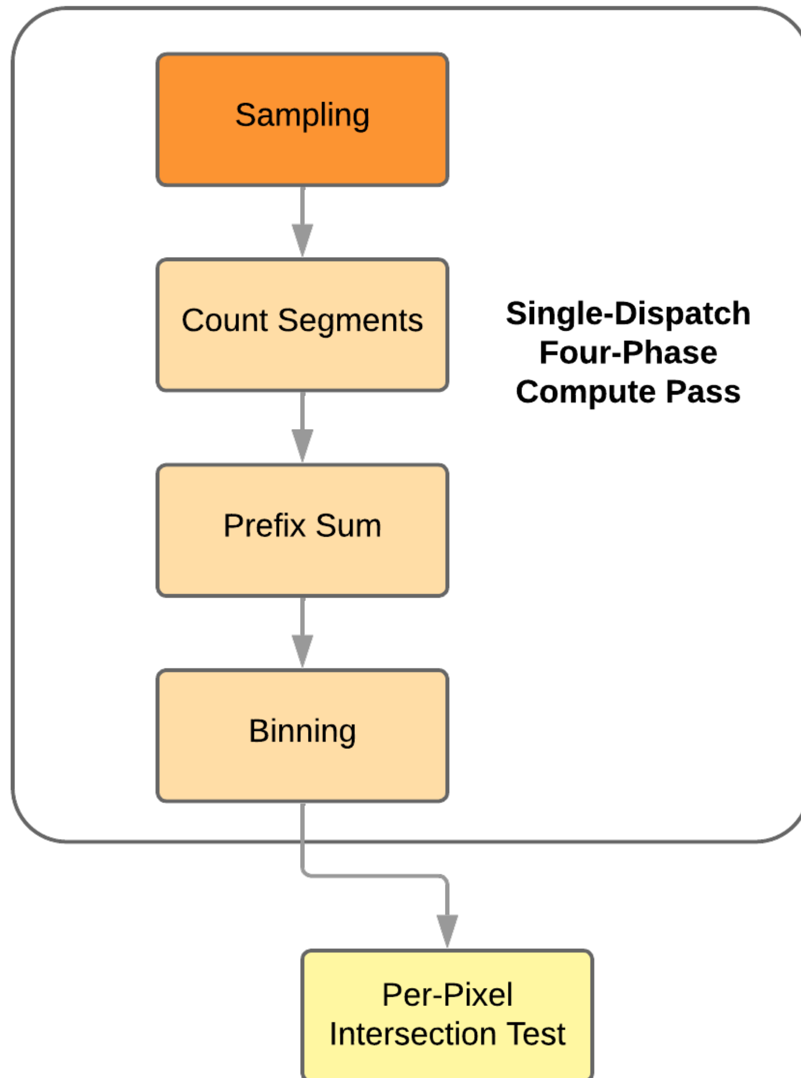


Figure 4.3: Overview of compute shader dispatch flow. A four phase single-dispatch is executed in the order depicted in the figure. Then, using the result from the binning step, pixel's resulting color are set.

The tessellation shader method, as previously presented, is a rasterization problem. Instead of rasterizing polylines, we propose a method that we call per-pixel intersection testing that computes which color every pixel should have.

In this section we describe the steps in implementing our per-pixel intersection testing method. The algorithm uses a pre-compute step with four dispatch phases, with the purpose of assigning line segments, sampled from B-spline curves, in a grid. After the initialization phase, the grid is used to intersection test every pixel's corresponding world space coordinate against all line segments that fall inside of the same cell as the pixel.

4.3.1 Uniform Grid

One key performance gain in this method is from using a uniform grid. As described in Section 3.4, we use this to spatially assign our primitives, line segments, and efficiently access them. Therefore, the goal with the initial four phase, as depicted in 4.3, is to build a grid that we can use to accelerate our per-pixel intersection tests.

For our test cases, we used a 128x128 grid size where every cell has a uniform size determined by a boundary where all B-spline curves are in world space.

4.3.2 Sampling

Sampling is done using a similar method to how we did it in Section. 4.2.2. Instead of using tessellation shaders, we dispatch a compute shader. Additionally, the samples are uniformly spaced depending on the sample count. For example, if we set sample count per B-spline segment to 10 would mean sampling such that $t \in [\, 0.0, 0.11, 0.22, 0.33, 0.44, 0.55, 0.66, 0.77, 0.88, 1.0\,]$. Lastly, we aggregate the samples into line segments.

One issue with this method of sampling is that we are writing to a buffer per GPU thread (thread block of size 32), meaning there are data races if we do not use any synchronization. To ensure there are no data races, we use an atomic counter to get a unique index into the output line segment buffer. There are other ways to efficiently implement this by allocating a range of indices that is shared within a thread block group using shared memory, but we reasoned that this is executed only once, and decided therefore to not use this method.

4.3.3 Counting Segments in Grid Cells

For our per-pixel intersection testing to work, we need an index to know where in a global buffer that every line segment associated with a cell can be accessed. In this step of the four-phase initialization, we keep a counter for every cell in a global buffer corresponding to the number of line segments that intersect that cell.

We find every intersecting line segment with the corresponding grid cells by first computing the bounding box of the line segments on the fly, by finding the minimum and maximum points of the line segment. This bounding box is then mapped onto the corresponding grid cells by computing the displacement to our grid origin and then divide by cell size as shown in Figure 4.4. The grid origin is defined as the bottom left point of the global grid bounding box.

Figure 4.4: Pseudocode to convert world positions to grid cell indices.

```
1 min_cell_box = floor((min_bbox - grid_origin) / cell_size)
2 max_cell_box = floor((max_bbox - grid_origin) / cell_size)
3
4 min_cell_box = clamp(min_cell_box, 0, 128 - 1)
5 max_cell_box = clamp(max_cell_box, 0, 128 - 1)
```

Once the bounding box is mapped onto corresponding grid cells, we can iterate over the grid cell indices to increment the counter for every cell that intersects with the line segment as shown in Figure 4.5.

Figure 4.5: Pseudocode to increment the number of segments per grid cell.

```
For every min_cell_bbox.y <= max_cell_bbox.y
    For every min_cell_bbox.x <= max_cell_bbox.x
        cell_id = y * grid_size + x
        grid_cells[cell_id] += 1
```

4.3.4 Computing Prefix Sum

Figure 4.6: Naive parallel prefix sum pseudocode.

```
For every i in parallel
    For j = 0 to cell_index
        prefix_sum_buffer[cell_index] += grid_cell_segment_count[j]
```

Prefix sum is an algorithm that given a set of input values, calculates the i-th value given all preceding values by adding them and itself up in e.g an array of integers [19]. More formally, we define the prefix sum for the i-th element as: $y_i = y_{i-1} + x_i$.

We implemented a $O(n^2)$ naive parallel prefix sum in a compute shader to calculate the starting position in the line segment buffer for every cell. The shader uses the buffer from the previous phase, *Count Segments*, as input. Then, we assign one GPU thread per element in a new output buffer where every thread is responsible to compute its prefix sum using the input buffer, by executing the algorithm specified in Figure 4.6.

4.3.5 Binning

The final phase of the initialization is the binning step where we assign line segments to a buffer. We use the same algorithm as previously mentioned in Section. 4.3.3, but in addition use two additional buffers: the prefix sum buffer and an atomic buffer with same size as the grid dimensions. The atomic buffer ensures that writes to the same grid cell are done atomically.

4.3.6 Per-Pixel Intersection Testing

The last phase of this algorithm is to determine which color every pixel should have. To do so, we use the uniform grid from the previous initialization phase to access line segments depending on which cell a pixel in world space resides in.

Figure 4.7: Screen-space to world space computation.

```
1 screen_to_world(pixel_pos)
2     pixel_center = (pixel_pos + 0.5) / screen_size
3     to_ndc(pixel_center)
4     return inverse_projection_matrix * pixel_center
```

To get the world space coordinates for every pixel, we compute the pixels' center point, transform it to normalized device coordinates (NDC), and then apply the inverse projection matrix as shown in Figure 4.7. The inverse projection matrix is only computed once and passed as a uniform. Other uniforms passed to the shader are the size of the grid, the size of every cell, a grid origin, pixel size, and line radius. The pixel size is computed by taking the differential with respect to x and y. Our line radius uniform is used to determine the width of the final rendered line, and is empirically determined to $1.5 * pixel_{size}$ and clamped to $[2.0, 150.0]$ to adjust its visual appearance at different zoom levels.

After computing a pixel's world space coordinate, we have an early exit test to avoid unnecessary point vs line tests. This early exit exploits the bounding box of the uniform grid, and simply tests if the pixel is inside and otherwise early exits. If a pixel is inside of the boundaries of the grid, then we find the corresponding cell that it resides in, similarly as described in Section. 4.3.3 with the difference that we are mapping a point and not a bounding box into grid cell indices.

When the affected grid cell is known, we use the two buffers constructed from *Prefix Sum* and *Count Segments* steps to retrieve the start index of the line segments for the cell and the number of line segments. We then loop and access every segment and perform point vs line intersection test, and calculate a max coverage that we later use to render smooth lines.

$$proj_{\mathbf{v}}(\mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{\mathbf{v} \cdot \mathbf{v}} \mathbf{v} \tag{4.1}$$

The point vs line intersection test uses the projection formula described in Equation. 4.1. We use this because it consists of few inexpensive operations and we can implement it using a threshold factor to compute a width. We consider the width as a empirically decided softness factor, which determines how fast the surrounding pixels fade out. Lastly, we use a glsl function called smoothstep, which implements cubic Hermite interpolation to smoothly fade the edges of the final lines. The smoothstep function is defined on an interval [0, 1] where the tangents are zero at $x \leq 0$ and $1 \leq x$ [15].

5

Results

In this chapter, we present the results from our proposed method to render paths on iGPUs. We present the performance using the methods presented in Chapter 4 and present visual figures of smooth and non-smooth lines.

To evaluate the performance of the two methods presented in Chapter 4, we used two different datasets with varying densities; the first dataset is taken from a practical real-world example where paths are uncommonly overlapped but still dense, while the second dataset is generated to stress-test our methods by being distributed such that paths are overlapping (see Figure 5.1 and Figure 5.2). Every time unit is an average that is computed by either sampling the FPS or compute time (ms) over a minute, and then computing the average.

5.1 Environment and Hardware Setup

All experiments were performed on a modern laptop with 32 GB RAM, an Intel Core Ultra 7 155H processor, and integrated Intel Arc Graphics.

We used OpenGL 4.6 to enable modern programmable graphics features, including tessellation and compute shaders, which are essential for the implementation and comparison of both rendering methods. All benchmarks were performed at a resolution of 1920x1080 unless otherwise stated.

All figures in this chapter are from an OpenGL application which implements the methods as described in chapter 4. Every B-spline curve is rendered in 2D, and can be visualized by scrolling and panning. Since the zoom-level affects the number of visual curves, we only compare the test cases where all B-spline curves are visible and at the same zoom-level. This is particularly important for our per-pixel intersection method results, as it performs worse when all pixels on screen are in the grid.

5.2 Datasets

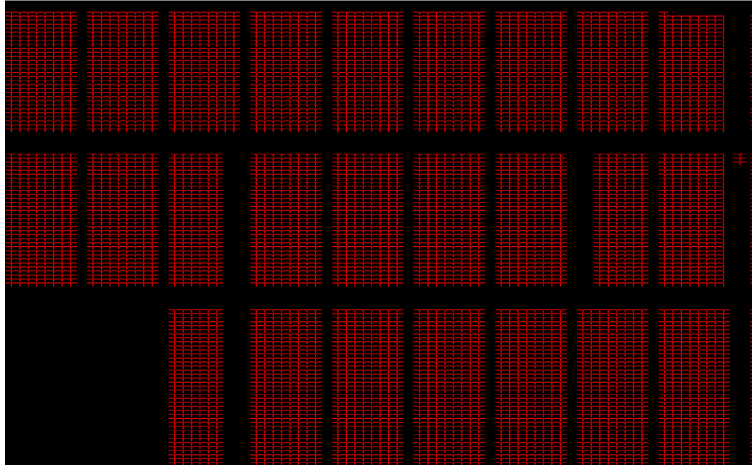


Figure 5.1: Dataset $\mathcal{D}_1$, rendered using our proposed per-pixel intersection testing method.

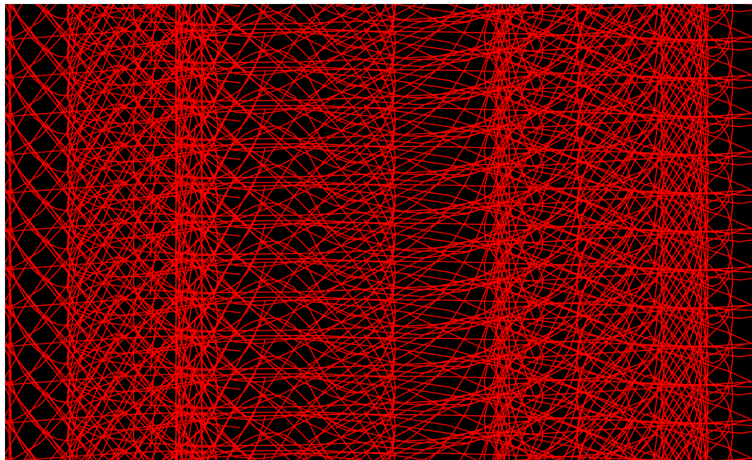


Figure 5.2: Dataset $\mathcal{D}_2$, rendered using our proposed per-pixel intersection testing method.

We used two different datasets to evaluate the rendering performance under practical and stress-test conditions. One dataset, $\mathcal{D}_1$, is used from a real-world application involving B-spline curves used to define paths for Automated Guided Vehicles (AGVs). The second dataset, $\mathcal{D}_2$, is synthetically constructed by combining and duplicating B-spline curves to create high-density overlapping regions, designed to stress-test the GPU utilization on an iGPU. These two datasets are visualized in Figure 5.1 and Figure 5.2.

5.3 Performance

Table 5.1: Maximum number of B-spline curves rendered at 60 FPS across two datasets, $\mathcal{D}_1$ and $\mathcal{D}_2$, using the two proposed methods. Per-pixel intersection enables rendering more paths at the expense of higher memory usage.

Dataset	Method	Max Paths @60 FPS	Memory Usage (MB)
$\mathcal{D}_1$	Tessellation Shader	225,000	128
	Per-Pixel Intersection	1,020,350	1090
$\mathcal{D}_2$	Tessellation Shader	222,000	126
	Per-Pixel Intersection	756,000	893

As shown in Table 5.1 we noticed an expected memory usage increase when testing our method with rendering more paths. This follows our intuition, since the advantage with using tessellation shaders is that the geometry is tessellated on hardware without us explicitly using a buffer in memory to store the resulting polylines for future rendering. Furthermore, we note that the per-pixel intersection method struggles in scenarios when all pixels are covered by multiple paths, which is the case in dataset $\mathcal{D}_2$. However, we stress that this is not a typical example of a real-world example, but exists to stress-test our methods.

A straight comparison as shown in Table 5.1 is not fair. The tessellation shader method and per-pixel method excel at different things: when the tessellation shader is fully zoomed in, we effectively cull the majority of non-visible paths making it a suitable method. In the opposite case, when all paths are visible the tessellation shader performs worse than the per-pixel intersection method. We still present this data in this way since they are two methods that we used to render paths, to put our per-pixel intersection testing method into perspective with a naive implementation to understand how much better or worse it is.

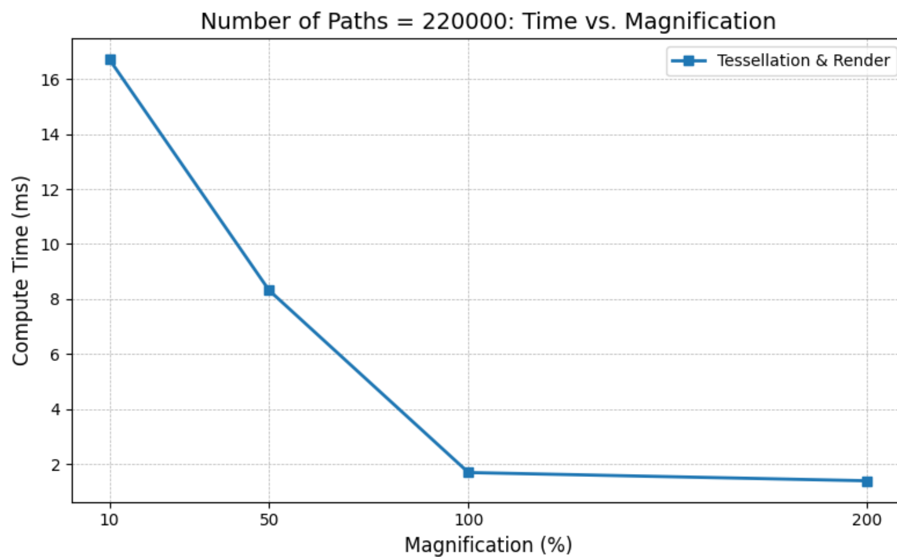


Figure 5.3: Plot with four different zoom-levels showcasing how performance is increased when non-visible paths are culled. The four samples showcased in the figure are from the dataset, $\mathcal{D}_1$.

As shown in Figure 5.3, there is a clear correlation between compute time and magnification. As zoom-levels change, or if the viewport is panned, non-visible paths are culled. When the magnification is 10%, most of the dataset is visible, leading to higher compute time. This pattern tapers off at 100% magnification, and there is very little performance to gain which can be seen by comparing the data point at 200% magnification with 100%.

5.3.1 Tessellation Shader Implementation Performance

The tessellation shader implementation performs similarly in both datasets, as expected. Geometry throughput is the limiting factor — as the number of control points grows, so does the number of patches that need to be processed, and we lose performance.

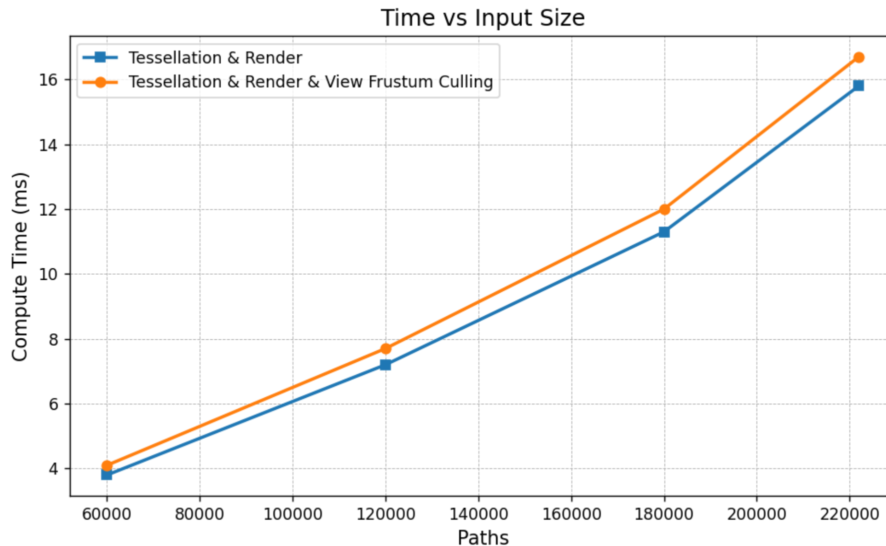


Figure 5.4: View frustum culling plotted alongside tessellation and rendering times as a function of input size.

View frustum culling has a small impact on the total compute time when the number of paths grow, as shown in Figure 5.4. In the worst-case, the view frustum culling compute time is around 1 ms. The Tessellation & Render plot shown in Figure 5.4 depicts the total time for the pre-pass view frustum culling compute shader, tessellation and render time.

We also measured the rasterization times for the same number of paths, but found that rasterization had a marginal effect on the total compute time. We think the reason for this is that since we are rendering isolines (OpenGL terminology for polylines), the rasterizer will not create fragments that have large coverage. We further verified that the rasterizer was not a bottleneck by using a profiling tool called RenderDoc. RenderDoc allowed us to capture a frame, analyze pipeline statistics, and more importantly, rasterization invocations and times.

5.3.2 Per-Pixel Intersection Testing Implementation Performance

To test the performance of our proposed per-pixel intersection testing method, we used the two previously presented datasets, $\mathcal{D}_1$ and $\mathcal{D}_2$. For both datasets, we duplicated the number of paths such that they contained the same numbers, 90k, 180k, 360k, 540k and 720k paths. Then, we re-ran the initialization phase 10 times per dataset and averaged the result shown in Figure 5.6

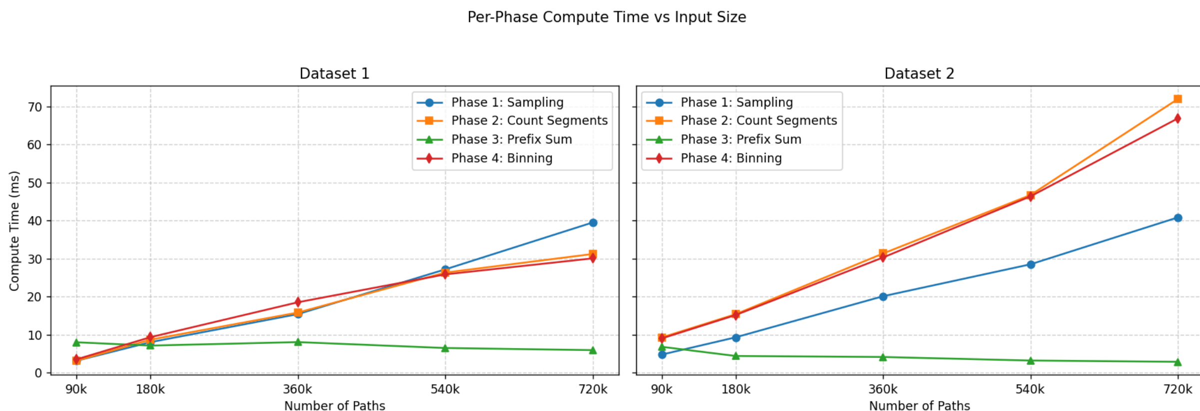


Figure 5.5: Four phase compute initialization comparison for datasets $\mathcal{D}_1$ and $\mathcal{D}_2$.

We used OpenGL timestamp queries to get as close to accurate measurements for every phase, meaning for every compute shader dispatch. Sampling scales linearly with the number of paths, and is similar in both datasets as shown in Figure 5.6. The most interesting part is that phase 2 and phase 4 takes much longer, which is especially noticeable in $\mathcal{D}_2$. In our implementation we use atomics for writing to cells. This seems to be the main reason to why phase 2 and phase 4 are slower in $\mathcal{D}_2$, since if there are more paths intersecting a cell in the grid, then every thread will compete against writing to the same index.

Scenario	% Inside Grid	Total Time (ms)
Best-case	0%	1.42
Average	∼50%	4.90
Worst-case	∼100%	7.50

Table 5.2: Per-pixel intersection test performance under varying grid occupancy using dataset $\mathcal{D}_1$.

Scenario	% Inside Grid	Total Time (ms)
Best-case	0%	1.44
Average	∼50%	10.82
Worst-case	∼100%	16.92

Table 5.3: Per-pixel intersection test performance under varying grid occupancy using dataset $\mathcal{D}_2$.

Our per-pixel intersection testing method performance depends on the percentage of pixels on the grid. Therefore, we present rough estimates in percentage of pixel coverage on the grid, and how it varies depending on the coverage as shown in Table 5.2 and Table 5.3. We conclude that our per-pixel intersection testing method

performs worse in high grid occupancy cases, since every pixel has to perform more intersection tests.

5.4 Path Appearance

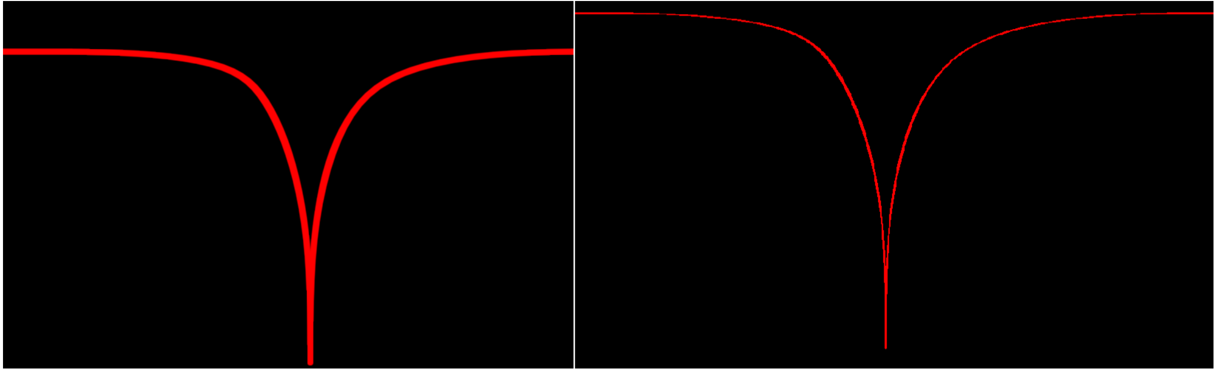


Figure 5.6: Paths rendered using per-pixel method. **Left**: smooth lines by calculating coverage and fading depending on distance to pixel. **Right:** non-smooth lines.

Figure 5.6

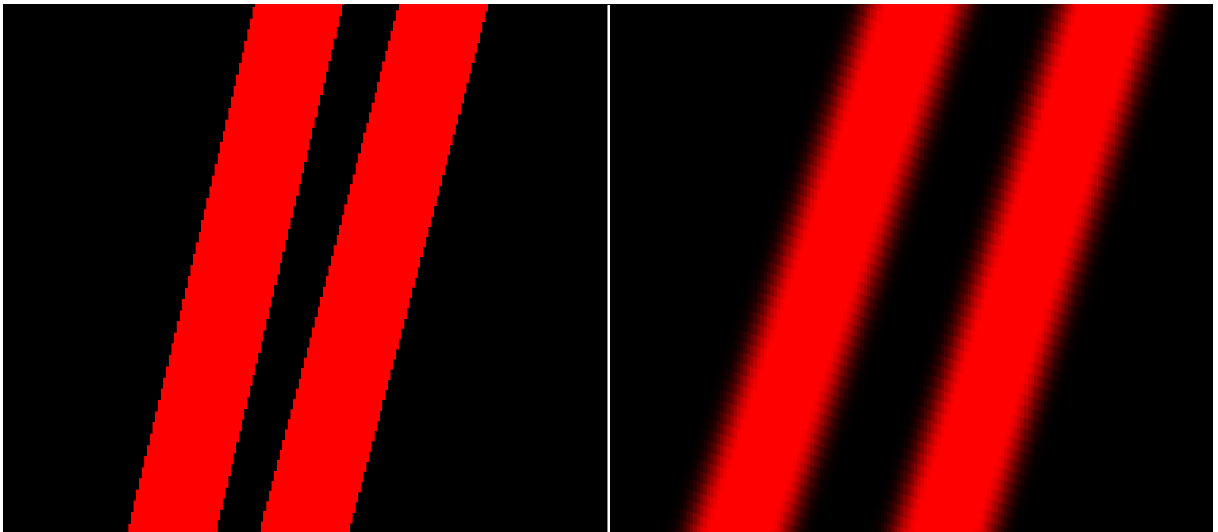


Figure 5.7: Clear visual aliasing and jaggedness rendered using per-pixel intersection testing method.**Left**: Jaggedness and aliasing in non-smoothing/fading solution. **Right**: Less visual aliasing by smoothing and fading edges.

The right image in Figure 5.6 illustrates the original, and unsmoothed path. When viewing the same image at higher magnification the jaggedness is more evident along the edges, as shown to the left in Figure 5.7. The other images in Figure 5.6 and Figure 5.7 we can see that the edges are blurry in the latter figure to the right, while not visible in the path in Figure 5.6.

6
Conclusion

6.1 Discussion

In this section, we discuss possible alternatives, suggested improvements, and issues with the proposed per-pixel intersection method.

6.1.1 Dynamic Layouts

Visualizing paths is not the only use case that users typically have. Paths are typically edited, transformed, or deleted at large scale. This is not a feature that our per-pixel intersection method supports, since the initialization phase where the grid is built is too slow to run in real-time; see Figure 5.6 for exact numbers. We cannot avoid sampling the paths with our per-pixel intersection testing method. However, sampling is only required once at large scale, and afterwards it is considered cheap. For example, if a user transforms a large number of paths at once, we only have to resample the transformed paths to render them. Unfortunately, this method still suffers from sampling and rebuilding compute times as the number of paths moved grows making this method infeasible for dynamic layouts. There exist other grid building methods that we could extend or adapt that for example focuses on memory efficiency [20].

A uniform grid does not lend itself well to dynamic layouts. Instead, we propose exploring alternative acceleration structurese.g., an AABB tree or a linear quadtreeto mitigate overlap issues and adapt cell size at runtime. In particular, a linear quadtree can be built efficiently on the GPU and automatically adjusts both grid resolution and extent.

Ultimately, we believe that a uniform grid is not suitable for dynamic layouts. Instead, we propose exploring alternative acceleration structures such as an AABB (Axis-Aligned Bounding Box) tree [21] or a linear quadtree [22] to mitigate overlap issues and adapt cell size at runtime. In particular, a linear quadtree can be built efficiently on the GPU and automatically adjusts grid size. AABB trees are also suitable because they inherently solve the issues as previously mentioned as with linear quadtrees, but can instead suffer from becoming unbalanced because of varying path density distributions.

6.1.2 Anti-Aliasing

Instead of using smoothstep in our per-pixel intersection testing compute shader to fade out the edges and calculate maximum coverage, we could rely on calculating sub-pixel coverage by storing a sufficient number of line segments that intersect each pixel. Then, we can test the known intersecting line segments against sub-pixels to calculate a sub-pixel coverage in a similar manner to super-sampling anti-aliasing (SSAA) [23].

6.2 Ethics and Risk Analysis

This thesis is conducted in close cooperation with a company and therefore an ethics and risk analysis is necessary.

6.2.1 Intellectual Property

All code will be implemented independently of proprietary data or applications, allowing results to be shared freely.

6.2.2 Conflict of Interest

There might be a conflict of interest in terms of the academic objective and the company's interest. To address the risks, the thesis will have a strong foundation in objectivity while developing results that align with the company's interests.

6.2.3 Social and Environmental Responsibility

This thesis addresses maximizing the utilization of the iGPU in path rendering, resulting in increased power consumption. However, an iGPU is a System on Chip (SoC) solution that is designed for low-power consumption [13], meaning, if we can find a solution that allows application engineers to fully utilize their iGPU for rendering paths, this might decrease the need for discrete GPUs which have a much higher power consumption.

6.3 Future Work

As previously discussed in the discussion section, we think the next step is to replace the uniform grid with a more fitting acceleration structure, e.g a linear quadtree to support dynamic layouts. Linear quadtrees, as mentioned in Section 6.1.1, are fast to build but slower to query [22]. We would also like to investigate how to best implement an anti-aliasing technique, as mentioned in Section 6.1.2 to replace or extend the technique of using the smoothstep function to fade out the edges.

The prefix sum phase, as shown in Figure 5.6, is the fastest out of all four phases. There is performance to be gained; for example, by using a parallel prefix-sum algorithm such as the one proposed by Harris et al. [24].

6.4 Conclusion

To summarize, we have answered our research questions by introducing a method called per-pixel intersection testing. This technique launches one GPU thread per pixel and computes the final color by performing intersection tests against line segments, effectively enabling the rendering of massive numbers of paths on iGPUs. While currently limited to visualizing static scenes, we propose several improvements to reduce or eliminate these limitations.

Bibliography

[1] A. Chaudhuri, *B-splines*, 2021. arXiv: 2108.06617 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2108.06617.

[2] L. Piegl and W. Tiller, *The NURBS book*. Springer Science & Business Media, 2012.

[3] S. Y. Gatilov, "Vectorizing nurbs surface evaluation with basis functions in power basis," *Computer-Aided Design*, vol. 73, pp. 26–35, 2016, ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2015.10.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010448515001591.

[4] A. Krishnamurthy, R. Khardekar, and S. McMains, "Optimized gpu evaluation of arbitrary degree nurbs curves and surfaces," *Comput. Aided Des.*, vol. 41, no. 12, pp. 971–980, Dec. 2009, ISSN: 0010-4485. DOI: 10.1016/j.cad.2009.06.015. [Online]. Available: https://doi.org/10.1016/j.cad.2009.06.015.

[5] C. Loop and J. Blinn, "Resolution independent curve rendering using programmable graphics hardware," in *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2005*, [Accessed: Jan. 30, 2025], 2005. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/p1000-loop.pdf.

[6] C. Loop and J. Blinn, "Rendering vector art on the gpu," in *GPU Gems 3*, H. Nguyen, Ed., Addison-Wesley Professional, 2007, ch. 25. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-25-rendering-vector-art-gpu.

[7] R. Santina, "Resolution independent nurbs curves rendering using programmable graphics pipeline," Sep. 2011.

[8] M. J. Kilgard and J. Bolz, "Gpu-accelerated path rendering," *ACM Trans. Graph.*, vol. 31, no. 6, Nov. 2012, ISSN: 0730-0301. DOI: 10.1145/2366145.2366191. [Online]. Available: https://doi.org/10.1145/2366145.2366191.

[9] V. Batra, M. J. Kilgard, H. Kumar, and T. Lorach, "Accelerating vector graphics rendering using the graphics hardware pipeline," *ACM Trans. Graph.*, vol. 34, no. 4, Jul. 2015, ISSN: 0730-0301. DOI: 10.1145/2766968. [Online]. Available: https://doi.org/10.1145/2766968.

[10] R. Li, Q. Hou, and K. Zhou, "Efficient gpu path rendering using scanline rasterization," *ACM Trans. Graph.*, vol. 35, no. 6, Dec. 2016, ISSN: 0730-0301. DOI: 10.1145/2980179.2982434. [Online]. Available: https://doi.org/10.1145/2980179.2982434.

[11] H. Kumar and A. Sud, "Fast, memory efficient and resolution independent rendering of cubic bezier curves using tessellation shaders," in *SIGGRAPH Asia 2019 Posters*, ser. SA '19, Brisbane, QLD, Australia: Association for Computing Machinery, 2019, ISBN: 9781450369435. DOI: 10.1145/3355056.3364548. [Online]. Available: https://doi.org/10.1145/3355056.3364548.

[12] D. Nehab and H. Hoppe, "Random-access rendering of general vector graphics," *ACM Trans. Graph.*, vol. 27, no. 5, Dec. 2008, ISSN: 0730-0301. DOI: 10.1145/1409060.1409088. [Online]. Available: https://doi.org/10.1145/1409060.1409088.

[13] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, "Performance characterisation and simulation of intel's integrated gpu architecture," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 139–148. DOI: 10.1109/ISPASS.2018.00027.

[14] J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.

[15] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/crc Press, 2019.

[16] M. NieSSner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer, "Real-time rendering techniques with hardware tessellation," *Computer Graphics Forum*, vol. 35, no. 1, pp. 113–137, 2016. DOI: https://doi.org/10.1111/cgf.12714. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12714. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12714.

[17] C. De Boor, "On calculating with b-splines," *Journal of Approximation theory*, vol. 6, no. 1, pp. 50–62, 1972.

[18] C. Ericson, *Real-Time Collision Detection*. CRC Press, 2004, pp. 79–80.

[19] G. E. Blelloch, "Prefix sums and their applications," 1990.

[20] E. M. T. II and S. N. Pattanaik, "A memory efficient uniform grid build process for gpus," *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, no. 3, pp. 50–67, Sep. 2016, ISSN: 2331-7418. [Online]. Available: http://jcgt.org/published/0005/03/04/.

[21] G. van den Bergen and, "Efficient collision detection of complex deformable models using aabb trees," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 1–13, 1997. DOI: 10.1080/10867651.1997.10487480. eprint: https://doi.org/10.1080/10867651.1997.10487480. [Online]. Available: https://doi.org/10.1080/10867651.1997.10487480.

[22] J. Dupuy, J.-C. Iehl, and P. Poulin, "Quadtrees on the gpu," in *GPU PRO 360 Guide to GPGPU*, AK Peters/CRC Press, 2018, pp. 211–222.

[23] M. Holländer, T. Boubekeur, and E. Eisemann, "Adaptive supersampling for deferred anti-aliasing," *Journal of Computer Graphics Techniques*, vol. 2, no. 1, pp. 1–14, 2013.

[24] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.