CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Efficient and Scaleable Web-Based Path Rendering

Unlocking the Potential of Compute Shaders on the Web

Master's thesis in Computer science and engineering

Oliver Karmetun

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Efficient and Scaleable Web-Based Path Rendering

Unlocking the Potential of Compute Shaders on the Web

Oliver Karmetun

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Efficient and Scaleable Web-Based Path Rendering
Unlocking the Potential of Compute Shaders on the Web
Oliver Karmetun

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisor: Marcus Nilsson, Toyota Material Handling Logistics Solutions AB
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2024

Efficient and Scaleable Web-Based Path Rendering
Unlocking the Potential of Compute Shaders on the Web
Oliver Karmetun
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With the rise of simulation and digital twin technology within the industry there has been an increased interest in taking advantage of web-based rendering, because of its user accessibility and device compatibility. However, this introduces challenges such as limitations on data transfer, data storage and limited API capabilities compared to classic rendering solutions. This is especially problematic for massive simulations such as those used within the autonomous vehicle industry. This thesis aim to demonstrate how technologies recently introduced to a web context, such as compute shaders, can be used to accelerate the common problem of path rendering or stroking, which is the rendering of visible curves on the screen. Our method is general and allows for rendering dynamic paths with dynamic levels of detail. Performance is comparable to other high performance solutions.

Keywords: Computer science, computer graphics, vector graphics, stroking, web, rendering.

Acknowledgements

I want to thank my academic supervisor Erik Sintorn for the help he has offered throughout the project. I also want to thank my advisors at TMHLS and T-Hive, Marcus Nilsson and Tanut Treratanakulwong for the great help they offered.

Oliver Karmetun, Gothenburg, 2024-06-25

Contents

List of Figures

List of Tables

1

Introduction

The use of web-based rendering to enable 3D applications in the browser has many advantages compared other methods. Firstly is its simplicity for the user in that it requires no installation of additional plugins or programs, thus requiring less effort to use by the user. Furthermore, WebGL, which is one of the web based graphics api, supports many types of devices, both desktop and mobile [1] and is thus a popular choice in the industry.

However, it does not come without its own challenges and drawbacks. Firstly, in the case of WebGL, it is more limited than OpenGL, preventing certain common computer graphics technologies to be used, such as geometry and tessellation shaders [2].

Furthermore, users on the web expect a high degree of responsiveness and short load times[3]. For large scenes with many objects to be rendered and information that is continuously updated, such as digital twins and simulations used in the context of autonomous vehicles, this can be challenging. One reason is that WebGL apps often depend on browser storage, which is limited in capacity [4]. Furthermore, the data might be hosted on servers and transferred over network connection and thus load speeds may be affected by fluctuations in connectivity and the type of network technology used[5].

One potential solution to these constraints on memory and data transfer for certain applications is to move away from complex 3D polygon/mesh representations of geometry and rasterized images towards simpler vector based representation of geometry where complexity is generated at the time and point where viewing happens. Vector representations of some types of geometry, such as cubic bezier curves, are less expensive memory wise compared to finalized mesh representations. Use of vector format could potentially lead to faster load speeds, as long as the load time and computation time of the conversion and rendering of the vector format at the time of viewing does not surpass that of using a finalized mesh.

The type of application used as an example in this thesis (autonomous vehicle digital twins) tend to use paths or maps that might already have vector representations [6] and therefore may benefit from keeping data in said format as much as possible.

There are however major challenges to rendering vector graphics. Firstly, modern gpu:s are built and adapted for regular 3D polygon/triangle rendering [7]. If we for example want to draw a curved line with a certain width, a solution is to convert

that line into a polygon representation, using enough triangles for the curve to look smooth. However, a challenge with this is that it is costly in terms of the computations and number of triangles needed and it may result in long and thin triangles that are not optimal [8] and cause aliasing issues. Also, since we preferably want the paths to be modifiable at runtime, we have to redo the computations every frame if the curve definitions change continuously, such as with smooth animations.

In this thesis we will present a method of constructing, tessellating and rendering paths in real time on the web, using GPU compute shader capabilities recently introduced to the web through the development of WebGPU [9]. Our implementation allows for curves that are dynamic, animated and resolution independent. It allows the rendering paths from their vector formats, cutting down on data transfer, from servers all the way to the GPU. It could be further extended to support other types of vector graphic elements such as those defined through svg format.

2

Theory

2.1 The Rendering Pipeline

The Rendering Pipeline refers to a sequence of stages that take initial data about a scene such as positions of vertices in it and outputs a final 2D image to be displayed on the screen [7]. Usually it refers to polygon rasterization, where vertices are processed into primitives (usually triangles) which are projected onto the screen, converted to fragments (pixels) which get a final colour through shading and depth testing (ensuring fragments obscured by other geometry are not drawn).

While polygon rasterization has become the commonly used mode of rendering, with hardware offering acceleration to support, it struggles with certain use cases [10]. Specifically it might struggle with vector graphics, which are represented using equations and points rather than polygons. Because the set of primitives that the gpu can draw is limited [11], complex shapes like curves and paths might need to be converted to a triangle representation, which can both be computationally expensive and lose fine detail and resolution independence that vector graphics allow. Furthermore, long and thin triangles or sub-pixel sized triangles often do not render well and might result in aliasing [8].

2.2 Vector Graphics

Vector graphics is the representations of images using math formulas, such as paths using line equations or bezier curves defined by control points and an interpolation equation[12]. Examples of file formats that use vector graphics are pdf and svg[13]. Common use cases for vector graphics are text and maps [6]. Vector graphics have many advantages, among them being smaller file sizes and resolution independence. A simple example is the cubic Bezier curve which can be represented using only 4 control coordinates/points in a vector format, with the curve being interpolated from them. If we convert a Bezier curve to polygons it might use 3 points per triangle and need 100s of triangles to appear smooth, depending on shape, size and resolution.

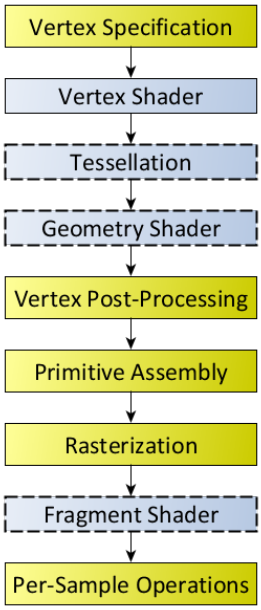

Figure 2.1: Overview of the rendering pipeline

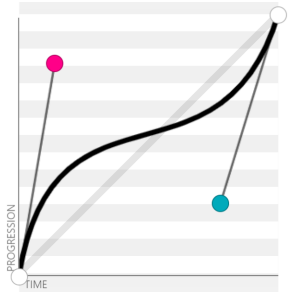

Figure 2.2: An example of a cubic bezier curve, defined by 4 control points

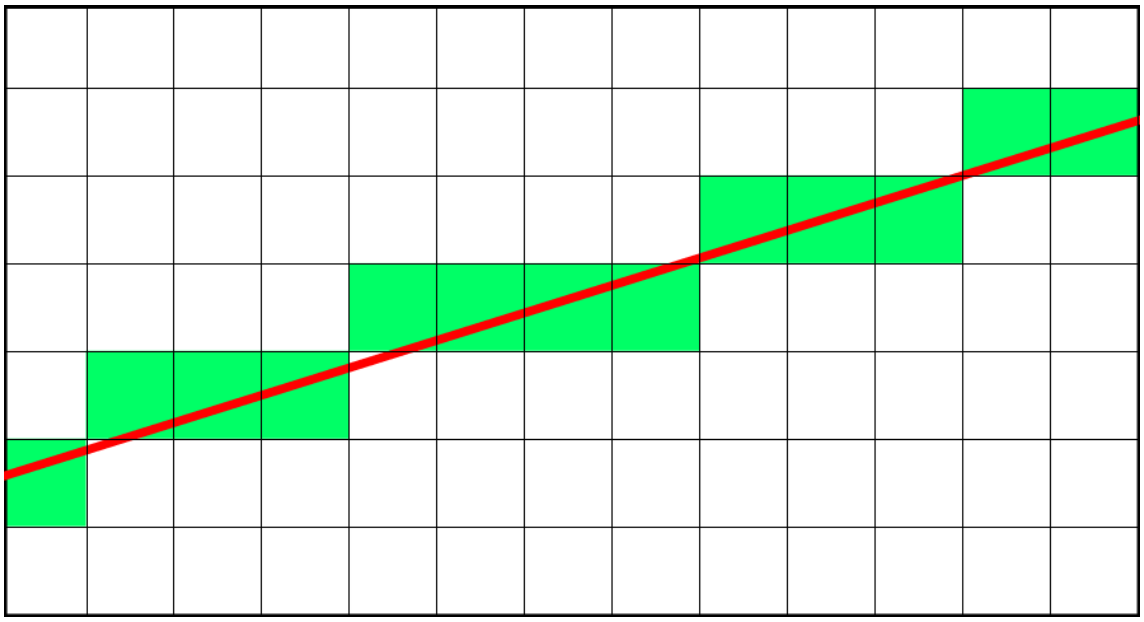

Figure 2.3: An example of jaggies. The red diagonal line that we wish to display has to be aproxximatedd with the pixels, shown as green.

2.3 Shaders

Shaders refer to programs that are run on the GPU[14]. They usually relate to a certain stage in the rendering pipeline, such as vertex or fragment shaders, where they for example determine vertex positions or pixel colour outputs.

Compute shaders are a more general form of shader that can handle generic computations not tied to a specific stage of the rendering pipeline. Compute shaders work on user-defined work-groups unlike the previously mentioned shaders which work per vertex or pixel. Threads are divided among work-groups. Each work-group can have many invocations of the shader code, depending on how its size is organized. All invocations within a work-group has access to shared data and communication that different work-groups don't share between each other [15].

2.4 Anti-Aliasing

Anti-Aliasing refers to to techniques used to combat aliasing, a visual artifact caused by having a lower sampling frequency than the max frequency of the signal. In the context of screen based anti-aliasing, the issue appears because screens have a discrete number of samples (pixels). Rendering curves or diagonal lines therefore results in jaggedness (See Figure 2.3). Furthermore, repeated patterns may result in moiré patterns, as can be seen in figure 2.4.

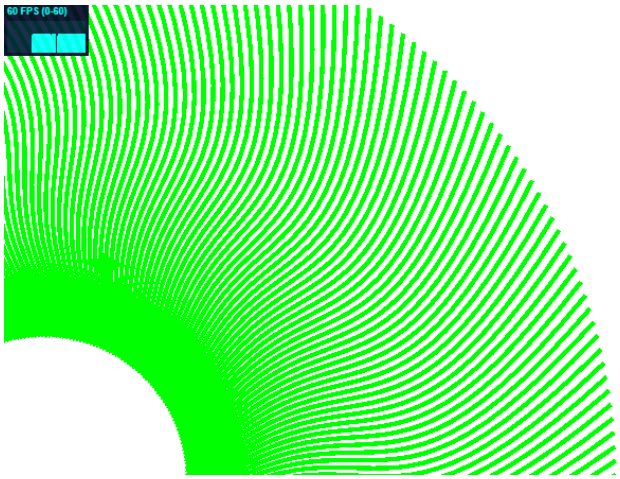

Figure 2.4: An example Moire patterns

2.5 WebGL

WebGL[16] is one of the leading standards of web Rendering, developed by the Khronos Group, who also developed the most used 3D graphics API, OpenGL[17]. Unlike other standards such as Adobe Flash and Unity Web Player, WebGL works in all compatible browsers without the use of a plugin. WebGL uses JavaScript for control code and OpenGL ES Shading Language (GLSL ES) for shader code, and outputs a 2D image to an HTML5 Canvas [16]. WebGL is based on OpenGL ES 2.0, which is a version of OpenGL tailored for embedded systems and less powerful hardware, containing only a subset of its features. WebGL lacks features such as compute shaders [2] and workarounds are needed to accomplish more generic computations.

2.6 WebGPU

WebGPU is a new JavaScript API developed by the World Wide Web Consortium (W3C) *GPU for the Web Community Group* with the stated mission to "provide an interface between the Web Platform and modern 3D graphics" [18]. Like WebGL, it allows for 2D and 3D rendering in compatible browser without the use of plugins. In addition to this they aim to bring general purpose computing through the form of compute shaders, something that has been lacking in previous standards. Furthermore, WebGPU has shown increased performance when compared to WebGL in certain cases [19].

2.7 Bezier curves

Bezier curves are curves interpolated from a number of control points P, using a parametric function of a step variable t between 0 and 1, as seen in equation 2.1 [20]. In order to approximate a bezier curve using linear line segments we can evaluate the function for s number of steps of t between 0 and 1. A t value of 0 fully will

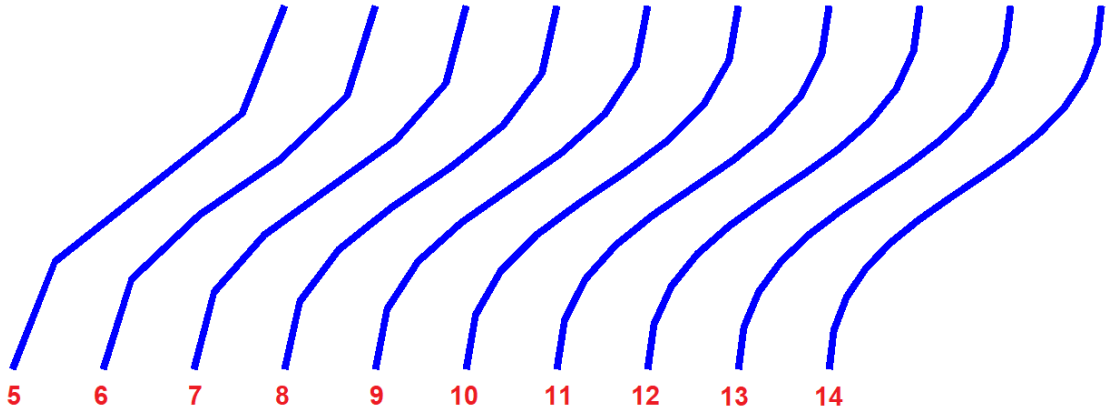

Figure 2.5: The effects on different values for the subdivision-count on the final linear approxiation of a cubic bezier curve.

evaluate to the first control points, 1 will evaluate to the last control point, and values in between will be interpolated between all control points using the equation below.

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} \mathbf{P}_i \tag{2.1}$$

The number of steps s is called the sub-division count. The effect of different subdivision counts can be seen in figure 2.5.

2.8 Tessellation

Tessellation describes the process of turning polygons into smaller polygons through sub-division, often for the purpose of increasing detail and smoothness dynamically [21]. For example we could subdivide a surface or mesh consisting of squares in a grid, by converting every triangle to two similar triangles (see figure 2.6). This could be used for a height-map where every newly generated vertex can get its height by sampling a displacement map.

Another use case that we make use of in this paper is path rendering, where a discrete simplification of a curve made out of line segments can get closer to its true shape by generating more segments and interpolating each one according to the mathematical function that describes the curve (see section 2.7).

Tessellation can be done using different methods. One way is using the CPU. However, this has a couple of issues. Firstly, the limited amount of cores of a CPU does not fit well with tessellation, which is often is the same simple program ran for a lot of separate data. If the data is too much, it might leave only pre processed, static tessellation as a feasible strategy for a large amount of objects. Furthermore, before rendering, the tessellated data needs to be moved to the GPU, making bandwidth a limiting factor [22].

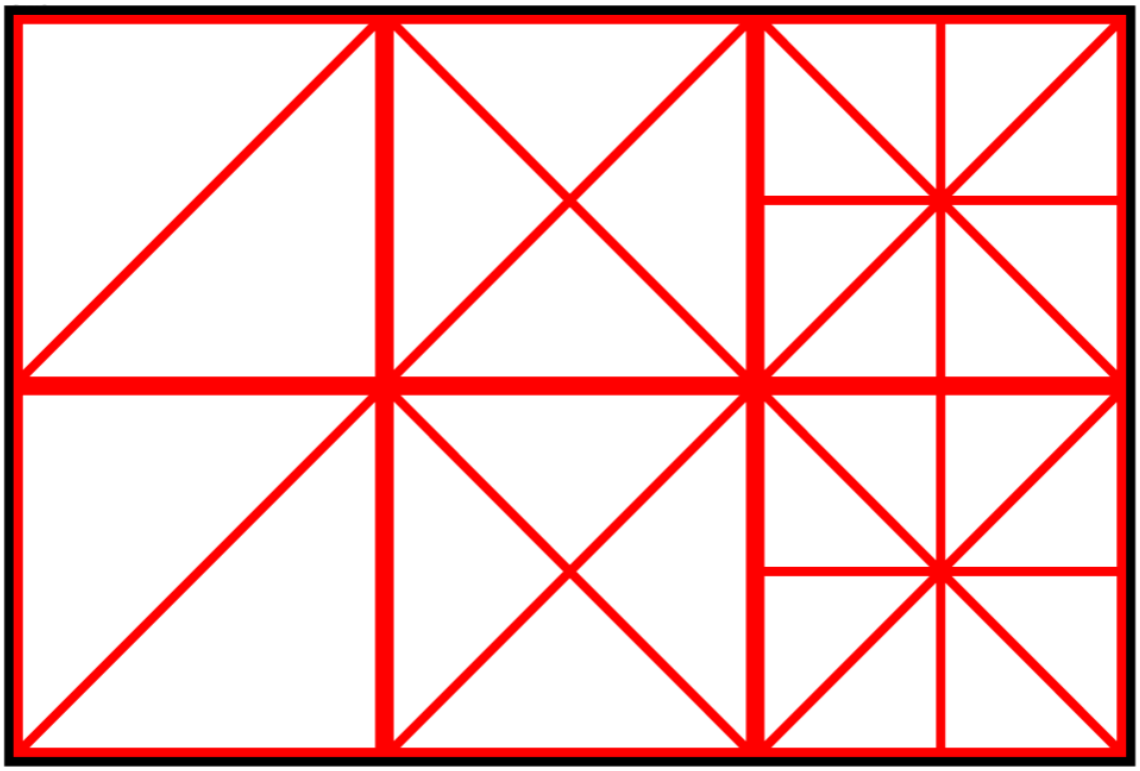

Figure 2.6: An example of a grid mesh being subdivided into smaller triangles.

Some tessellation can also be done using shaders on the GPU. The geometry shader is capable of some tessellation [23]. With the introduction of OpenGL 4.0 a dedicated tessellation shader was developed [24], but it has been criticized for slowing the pipeline, being limited in its tessellation factor [25] and requiring careful design by the artist to avoid artifacts [26]. Finally, compute shaders have been shown to be able to efficiently perform tessellation [25], while having the advantage of being flexible in usage and implementation.

2.9 Related Work and Contribution

2.9.1 Path Rendering

In this paper, path rendering refers to the rendering or drawing of curves, lines and paths. This process has also been called path stroking [27], referring to the stroke of a pencil or brush. This is contrasted to path filling, which refers to filling the region inside a path.

There are different ways to do path rendering. The most straight forward way is to first do tessellation to convert the vector format to triangles and then render them.

2.9.1.1 Stencil buffer methods

Loop & Blinn presented a method for resolution independent rendering of paths and regions defined by a path boundary [28]. The method takes a vector object of line

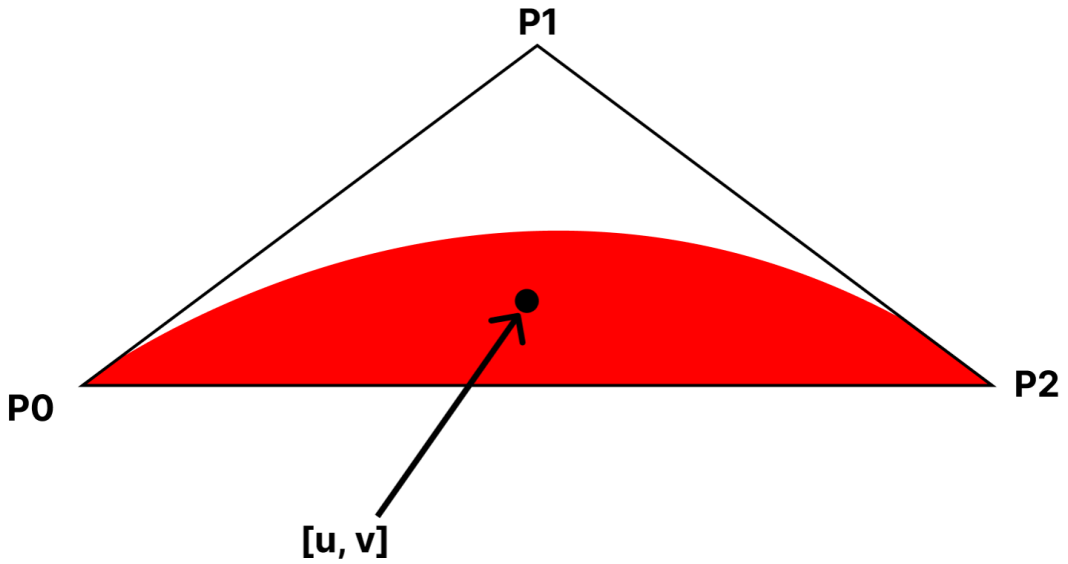

Figure 2.7: Uv coordinates are interpolated from the triangle vertices by the gpu, and used to determine if a point is inside or outside of the curve using the curve formula in the pixel shader.

segments and Bézier curves defined by control points. From these points polygons are generated and triangulated in a way that avoids curved edges, giving a region that can simply be filled and drawn to a colour buffer. Next another set of triangles are created by connecting control points of the Bézier curves after which triangle subdivision is done in order to avoid overlapping triangles. These can be drawn to a colour buffer, using a pixel shader to decide if a pixel is inside or outside of the curve (see figure 2.7). A problem with this method was that the initial steps before drawing, such as triangle subdivision, were expensive and needed to be done through pre-processing on the cpu, meaning that dynamic paths could not be done at a massive scale.

Kokojima et al. used a similar approach, but skipped the triangulation steps for regions without curved edges, instead using the stencil buffer to figure out what regions should be filled allowing more processing to happen on the gpu and for dynamic modifications to the paths [29]. A problem with this method is that the stencil buffer must be reset between each shape, making large amounts of small paths problematic.

Kilgard & Bolz also use a stencil buffer approach, but detaches the stencil part and the cover or shading part, as well as offering specific algorithms for stroking paths, dashing and mixing 3D and path rendering [30].

Another method published by Nehab & Hoppe is to use a vector texture approach [31]. It parallelizes over samples rather than shapes, meaning that samples are processed in parallel and shapes are then processed sequentially for each sample, whereas the previously mentioned methods work on the shapes sequentially and

the samples for a shape are processed in parallel. The method has a sequential pre-processing stage which Ganacim et al. later parallellized as well [32].

2.9.1.2 Scanline method

Li et al. presented a new method based on scanline rasterizers, that separates boundary fragments (Edges of paths and shapes) and computes them in parallell on the GPU while non- boundary processed in bulk similarily to cpu scanline rasterizers [33].

2.9.2 Hardware tessellation

2.9.2.1 Compute Shader Tessellation

Khoury presented a method of tessellating triangles and quad polygons into smaller triangles using compute shaders on the gpu [26]. Their implementation utilized a quadtree structure to efficiently subdivide triangles either as they got closer to the viewpoint or as they got larger on the screen.

Khourys method was preceeded by a number of other hardware tessellation methods, as described by NieBner et al. in their survey on the topic of Hardware tessellation[22].

2.9.2.2 Nanite Meshes

An example of tessellation used in production is Nanite [34]. With the release of Unreal Engine 5, Epic Games introduced a new geometry technology called Nanite. When Nanite is enabled on a mesh it clusters its triangles into a tree like structure, where nodes further up in the tree represent less detailed but larger areas of the geometry. This happens as a pre-processing step. During rendering the level in the tree will be chosen based on the camera view, with neighbouring clusters connecting without artifacts. This allows for much more detailed meshes than what is usually feasible to use.

2.9.3 Our Contribution

Our primary contribution is proving that tessellation for path rendering can be done efficiently using compute shaders in a web context using web api:s, with the example of bezier curves. While the algorithm used for subdivision and tessellation and the use of compute shaders is not novel, to our knowledge the combination has not been implemented in a web context. This finding is promising since it might open up the use of other types of tessellation algorithms with the benefits of web-based rendering.

3

Methods

Our initial design works in two phases. One compute phase, where the control points of the curves are used to construct a polygon based representation. The compute phase has an optional sub-phase, where subdivision levels are calculated and memory is more efficiently allocated for each curve. This sub-phase will be used to determine whether the improved memory management is worth the additional computation time. The compute phase is followed by a render phase, which is quite arbitrary in that it renders the generated polygons in the traditional way.

3.0.1 Buffers

The implementation uses 5 types of buffers. Firstly, a uniform buffer that holds uniforms common for every invocation of the shader for a single frame. For rendering, we need a Model View Projection matrix according to traditional 3D rendering. For computing, we need the camera position for the purpose of scaling subdivision count according to the distance from the curve to the camera.

Secondly, We have the control point buffer. It holds the control points that define the curves and is used as input in the compute phase to interpolate the vertices resulting from subdivision and tessellation.

Thirdly we have the curve specs buffer, which holds additional, per curve information such as width and colour. This is used as input in both in the compute and render phases.

Then we have a vertex buffer which holds the vertices for the curves constructed in the compute phase. It is an output from the compute shader in the compute phase and is used as an input for the vertex shader in the rendering phase. The vertex buffer holds the vertex data for multiple curves. The reason why is to avoid doing a seperate draw call for each curve. Because of this the vertex buffer is sized in relation to the max buffer size.

Because of the shared buffer and the fact that the compute shader runs many threads in paralell, we need to make sure that there are no conflicts. Because the vertex buffer is only written to in the compute phase and reading only happens in the render phase at which point the compute phase is done, writing a specific curve to a unique allocated space in the buffer is safe. We have two ways to allocate space considering the dynamic subdivision count of each curve.

Firstly, we can simply for each curve allocate the maximum amount of space a curve will use when it gets the maximum subdivison count. However, this will lead to a underutilized buffer, and thus, unnecessary data being transferred, slowing down the rendering. But it has the benefit of requiring no additional computation.

Another option is to do what we called a sub-phase of the compute phase, in which we calculate the subdivision count for each curve and determine the specific place and amount in the buffer that each curve should occupy. This require some additional buffers. Firstly, a subdivision count buffer, to know the subdivision of each curve. Furthermore, we need a buffer to, for each curve, store the prefix sum (accumulated sum of all previous curves subdivision count) as well as some intermediary buffers to help compute the prefix sum. But we explain more about that in section 3.0.2.

Finally we have an index buffer. It holds the indices that defines the triangles that make up the finished curves. Indices are generated with their corresponding curve in the compute shader and outputted to their specific allocation in the index buffer in a similar manner to the vertices for the vertex buffer.

3.0.2 Compute phase

The compute phase is responsible for using the curve definitions and control points to output a polygon representation of the curve.

Depending on the amount of curves to be drawn and the maximum buffer size of the device, a certain number of buffers of each type are made and multiple compute passes are issued.

3.0.2.1 Subdivision Determination

As mentioned in section 3.0.1, we have two alternatives for how and where to output the vertices to in the buffer. Firstly, the simplest way is to just allow for each curve to take up the space needed if the curve used the maximum possible amount of vertices, meaning that the maximum subdivision count is used. But this would likely be unnecessary in most cases, and a lot of unused, empty memory would be transferred and more buffers might be needed to accommodate many curves.

Instead, we use a separate compute phase with its own compute shaders in order to maximise the usage of each buffer. The method is as following. Firstly we use a separate compute shader in order to determine the subdivision count. This determination is made using a linear interpolation based on the distance from the camera and the curve, clamped between a subdivision count of at least 5 or at most 40. These values are chosen quite arbitrarily in order to show the effect of varying subdivisions, but 40 will generally look very smooth, while 5 will look quite jagged. These values are, for each curve, put in a buffer that we call the subdivision buffer.

Pseudo code for subdivision determination is shown below.

```
Define a structure "Uniforms" with the following field:
  - CameraPosition
```

```
Define a buffer "ControlPoints":

Define a buffer "SubdivisionOutput"

Define a compute function "main":
  – Get the control points at positions for index.
  – Calculate the centre point.
  – Calculate curve to camera distance.
  – Interpolate the subdivision level.
  – Store the subdivision level to subdivisionOutput.
```

3.0.2.2 Cumulative Index Computation

After determining the subdivision count for each curve, we need to determine where the vertices and indices of each curve should be placed in the shared buffers when they are computed. But this depends on the specific subdivision and necessary memory for each curve preceding a curve. Therefor we need to performs a cumulative sum calculation.

There are different algorithms developed for this purpose, but we chose to use a straightforward sequential algorithm that is parallelized using WebGPU compute shaders.

There are 2 seperate shaders and compute passes for this method. Firstly, in stage 1, a shader takes as input the subdivison buffer computed described in section 3.0.2.1. The input data is divided into groups of 256 each. For each index, or curve to be, we sum up all preceding subdivision counts in that group and store it at that index in the cumulative index buffer. Furthermore, for the final index of a group, the cumulative sum for the whole group is stored in a separate auxiliary buffer.

In stage 2, for each group, the auxiliary buffer contents of preceding groups are added to each index in the cumulative index buffer.

3.0.2.3 Curve construction

The subdivison algorithm itself is based on De Casteljau's Algorithm for bezier curve evaluation. In short we do a cubic bezier interpolation between the control points for a discrete amount of steps between 0 and 1, generating 2 vertices per step that are pushed perpendicularly to the tangent of the curve to opposite sides in accordance with the width value. The steps can be seen below:

- Determine the step length: 1 divided by the subdivison value. Initialize step value to 0.
- For each step.
  - Perform the cubic bezier interpolation for the current step value t_i and the next step value t_{i+1} in order to get the position p_i, p_{i+1} and the tangent.

  - Make 2 new vertices at pi, offset by the width of the curve, perpendicularly to the tangent.
  - Store the vertices in the vertex buffer at the correct location.
- Generate the indices, starting and ending with max value to indicate a new curve.

3.0.3 Baseline methods

In order to measure the performance of our gpu curve tessellation method, we use a number of baseline methods.

3.0.3.1 Dynamic cpu curve construction

Firstly, we use a simple cpu implementation of a similar algorithm to our gpu implementation. It takes a float array of control points and outputs two other float array that instead contains vertices and indices for the tessellated curves. These are then written to WebGPU buffers followed by the same render phase as our gpu method uses. This is repeated every frame in order to allow for changes to curve definitions at runtime (dynamic curves).

3.0.3.2 Preconstructed curve rendering

In addition to the cpu and gpu dynamic construction, we use a pre processing solution in order to determine how many curves can be rendered if there is no need to reconstruct the curves each frame. This gives us an idea of how much of a performance impact the curve construction has compared to the rendering of the resulting triangles. The curves are constructed using the cpu algorithm and vertices and indices are placed in buffers. This is only done once and the rest of the frames simply renders the curves using the same buffer data.

3.0.4 Test Case

3.0.4.1 Simple Curves Case

The Simple Curves Case is intended to measure the performance of the subdivison algorithm on dynamic bezier curves. It is composed of rows and circles of cubic bezier curves, defined by 4 control points each. Each full row is composed of 1024 curves, and each circle is composed of 256 curves. All control points are put in a float arrays which our method as well as the baseline methods can use as input. The control points are generated once as a pre-process step and the generation does not affect the measured results after that.

The particular layout of the test case should not have any effect on the results, since the same shader program will be ran regardless. The exception is if dynamic subdivision counts are used, since this may affect the number of computations for curves far away and a sparse layout could be expected to perform better than a dense, zoomed in layout.

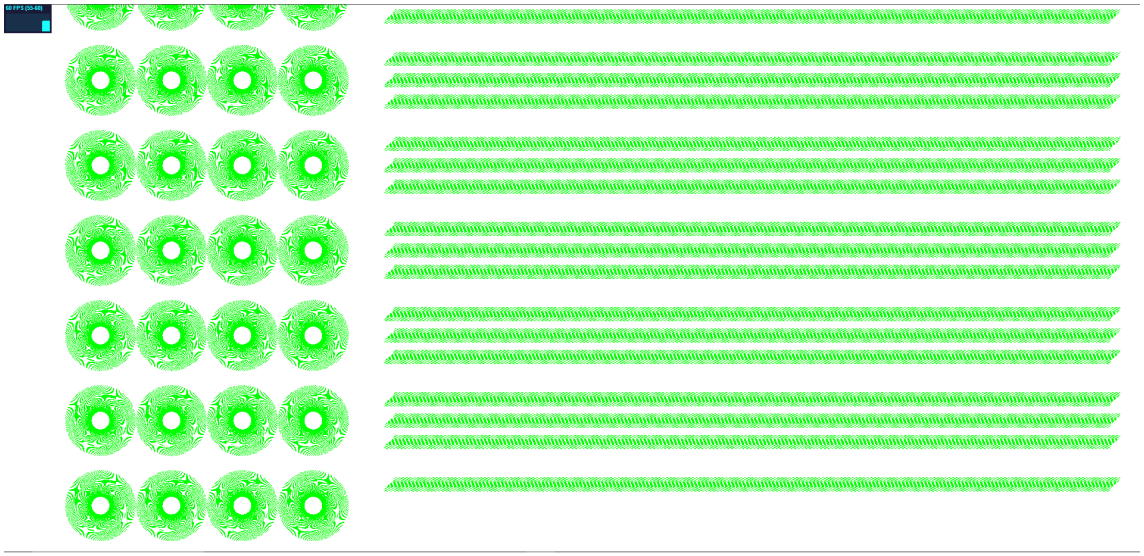

Figure 3.1: The simple curves case

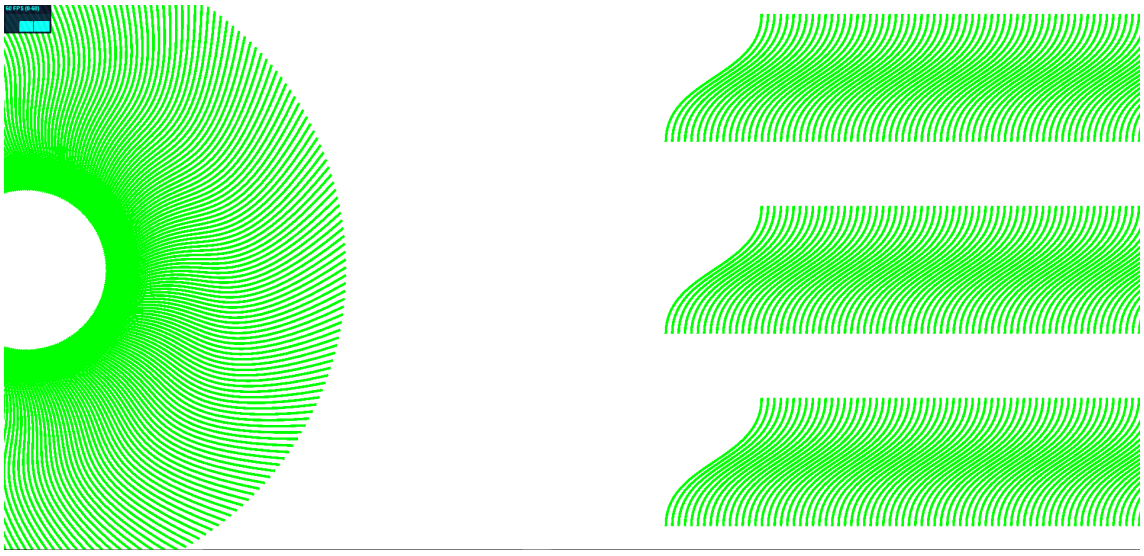

Figure 3.2: The simple curves case, zoomed in

3.0.5 Testing method

For testing the performance of the methods different techniques are used.

Firstly, for each method, the number of curves that breaks a stable 60 fps are recorded. This give a general idea of how well all steps of the method in combination works.

Secondly, for the cpu construction of curves, the time taken to construct the curves from control points is recorded using the javascricpt performance.now() function. Times are taken before and after the construction methods and an average is taken over multiple frames.

Finally, for the render phase as well as the compute shaders, there is WebGPU functionality to measure the exact time for the shaders to run. We record both render and compute times.

4

Results

The measurement that is taken is the amount of curves that can be drawn while reaching 55 fps. The test case used is the Simple Curves Case. The graphics card used is a Geforce RTX 3080 and the cpu is a AMD Ryzen 3900X as well as 32 GB of RAM. The methods tested are CPU curve construction, compute shader construction with and without dynamic subdivision counts (LOD) and with and without the memory management solution mentioned 3.0.2.2 as well as pre-constructed curve rendering.

Table 4.1 shows the number of curves that can be rendered at 60 fps for each method.

For each method the time of running the computation of curves using compute shaders as well as the time for the render phase to finish is recorded. The results can be seen in figures 4.1, 4.2 and 4.3.

Furthermore in figure 4.4 the exact execution times in microseconds for each input and method can be seen. The cpu construction only has times recorded up to 5000 curves, while the pre-processed solution additionally has data recorded for 1500000 curves and 2000000 curves.

Method	Number of curves at 60 fps
CPU construction	1000
Compute shader (basic)	679000
Compute shader (LOD)	874000
Compute shader (LOD/Memory)	836500
Pre-Constructed curve rendering	1650000

Table 4.1: Results for the Simple Curves Case

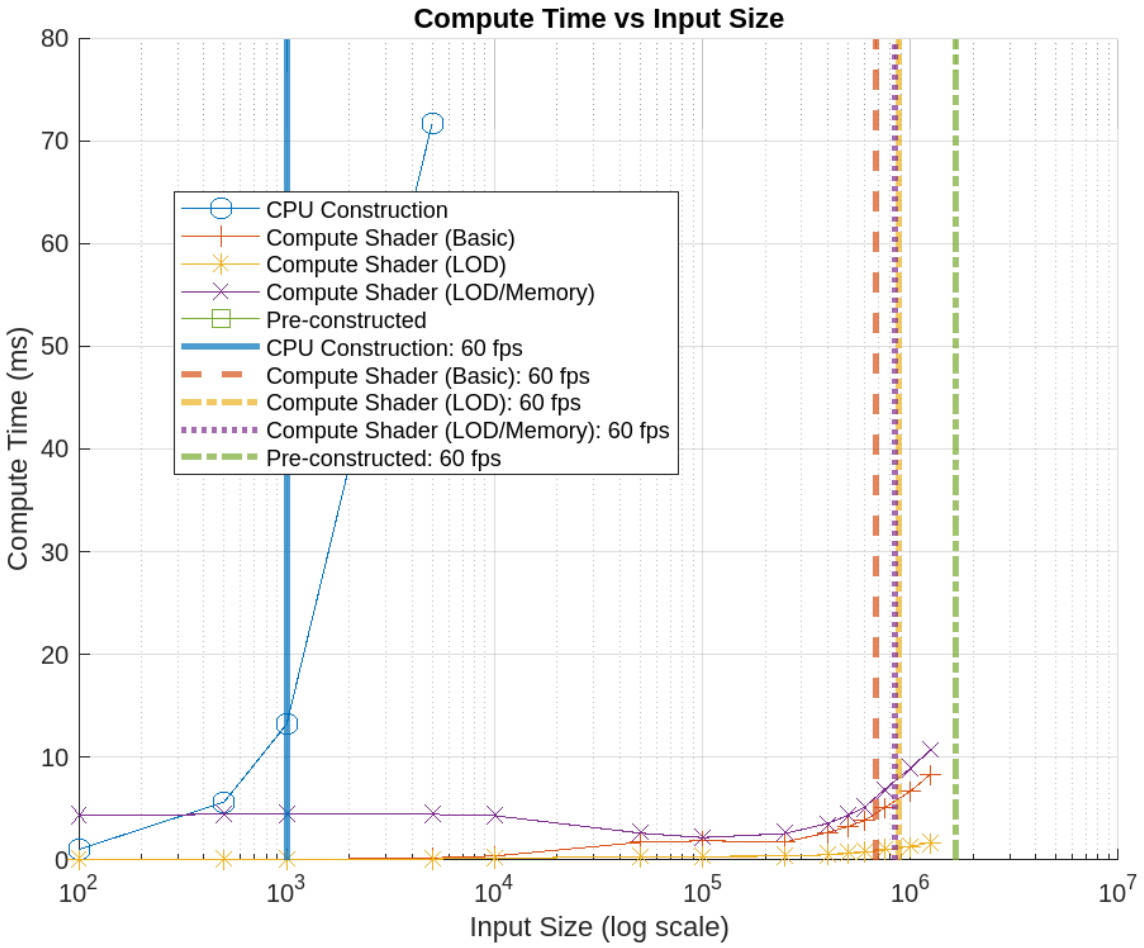

Figure 4.1: The time in milli seconds for the compute phase to finish for each of the methods.

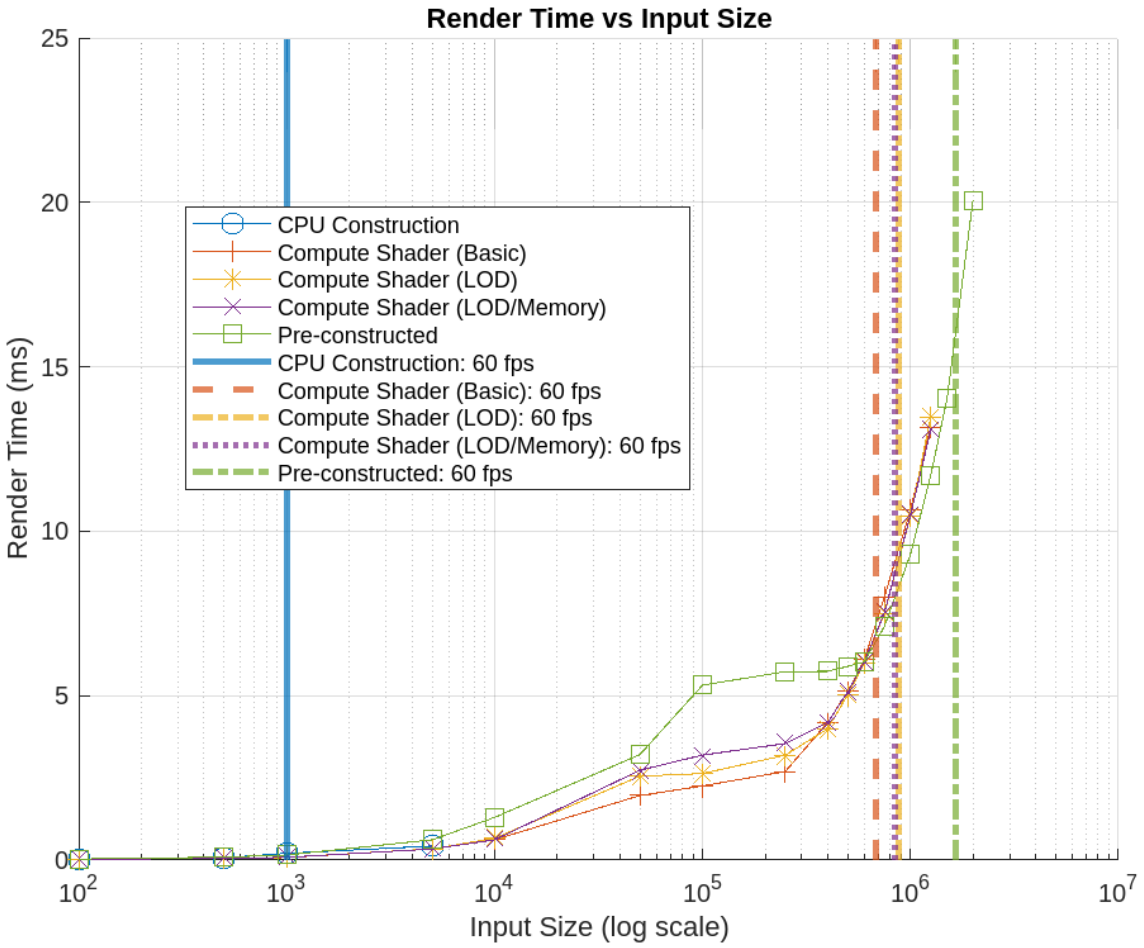

Figure 4.2: The time in milli seconds for the render phase to finish for each of the methods.

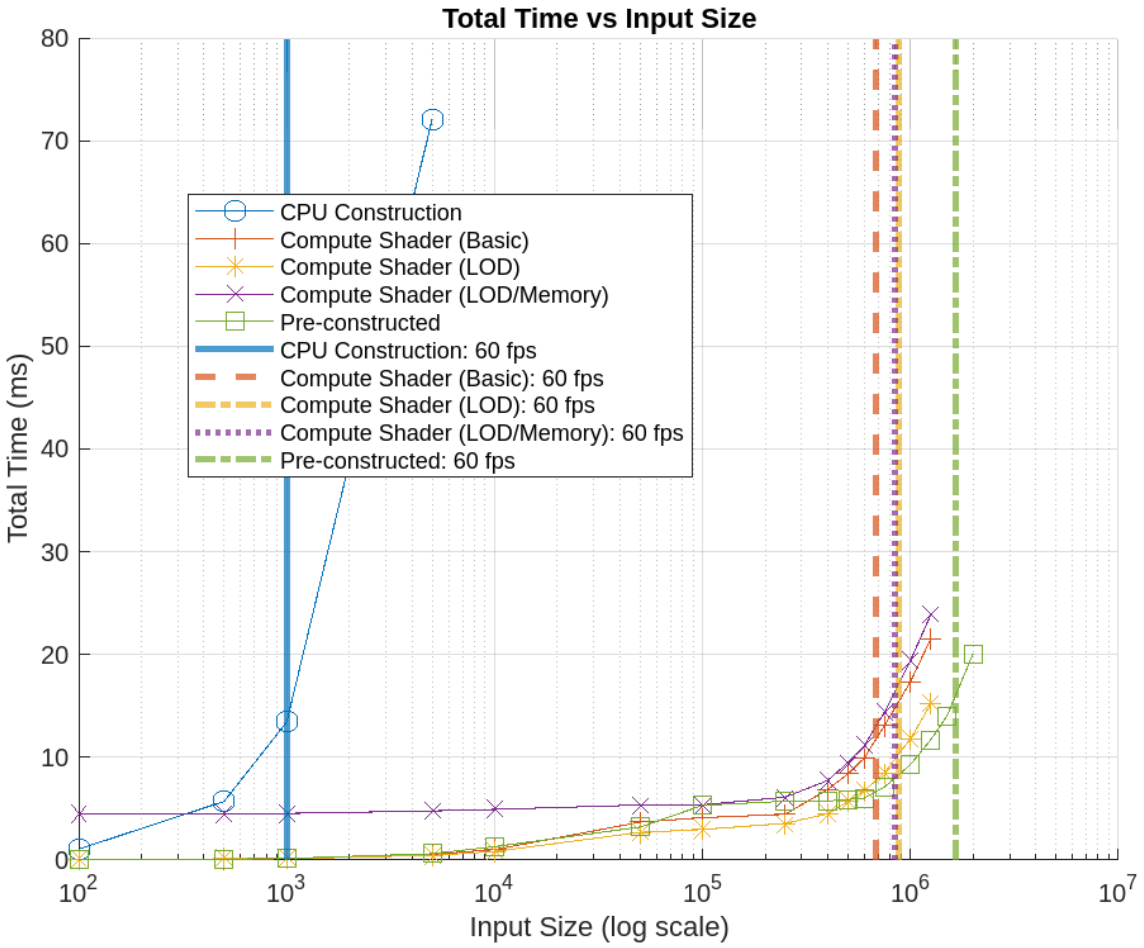

Figure 4.3: The time in milli seconds for the total compute + render shaders or methods to finish for each of the methods.

		Execution time (ms) (100 curves)	Execution time (500 curves)	Execution time (1000 curves)	Execution time (5000 curves)	Execution time (10000 curves)	Execution time (50000 curves)
CPU Construction	Compute:	1032	5643	13233	71642		
	Render:	30	68	204	424		
	Total:	1061	5711	13438	72066		
Compute shader (basic)	Compute:	43	44	44	203	392	1725
	Render:	26	46	78	338	617	1961
	Total:	69	89	123	541	1009	3686
Compute shader (LOD)	Compute:	62	61	61	107	169	330
	Render:	26	49	80	337	660	2550
	Total:	88	110	141	444	830	2686
Compute shader (LOD/Memory)	Compute:	4404	4438	4453	4445	4313	2624
	Render:	25	49	81	334	615	2730
	Total:	4429	4487	4533	4779	4928	5353
Pre-constructed curve rendering:	Total:	31	88	152	614	1296	3207

		Execution time (100000 curves)	Execution time (250000 curves)	Execution time (400000 curves)	Execution time (500000 curves)	Execution time (600000 curves)	Execution time (750000 curves)
Compute shader (basic)	Compute:	1868	1763	2669	3256	3831	5082
	Render:	2254	2693	4183	5124	6087	8007
	Total:	4122	4456	6853	8380	9919	13089
Compute shader (LOD)	Compute:	347	372	509	640	770	977
	Render:	2631	3188	3991	5034	5978	7503
	Total:	2978	3561	4500	5674	6748	8480
Compute shader (LOD/Memory)	Compute:	2185	2577	3559	4352	5115	6854
	Render:	3192	3543	4181	5118	6018	7526
	Total:	5377	6120	7740	9470	11133	14380
Pre-constructed curve rendering:	Total:	5317	5731	5740	5890	6022	7109

		Execution time (1000000 curves)	Execution time (1250000 curves)	Execution time (1500000 curves)	Execution time (2000000 curves)		
Compute shader (basic)	Compute:	6701	8351				
	Render:	10642	13155				
	Total:	17343	21506				
Compute shader (LOD)	Compute:	1322	1702				
	Render:	10453	13453				
	Total:	11775	15155				
Compute shader (LOD/Memory)	Compute:	8931	10754				
	Render:	10485	13063				
	Total:	19416	23817				
Pre-constructed curve rendering:	Total:	9293	11674	14026	20069		

Figure 4.4: Exact execution times in micro-seconds for all methods and inputs

5

Conclusion

5.1 Discussion

5.1.1 Performance

As can be seen in section 4 and table 4.1, our basic method outperforms the CPU construction baseline by a factor of almost 700. Furthermore, we can see that by enabling dynamic subdivision counts we can get an even better performance, being able to construct and render 874 times more curves than the cpu baseline.

The pre-processed solution is able to render the most curves, at slightly more than double the amount of curves that our solution can render. The pre-processed solution mainly gives an idea of how much of the execution time is spent on rendering and other overhead. So the results indicates the limit to what can be rendered in real time with static curves. And that our solution manages to render at least half of that with dynamic curves.

5.1.2 Level of Detail

As can be seen in figure 4.4, the LOD solution has a higher compute execution time than the basic solution until a cutoff point of where it is strictly better. We explain the worse performance in the beginning with the fact that the first curves are close to the initial position of the camera and thus will be rendered with a high subdivision, making the distance check and dynamic subdivision count determination less effectful. However, after this the LOD solution is clearly better in terms of compute execution time, with the final check of 1250000 curves giving the LOD solution a 4.9 times improvement. However, we notice that the render time is similar for both solutions, and it is the bigger part of total execution time in later stages.

5.1.3 Memory management solution

Our improved memory management solution did not show as promising results as the plain LOD solution. As can be see in table 4.1, the solution could not reach the performance of the LOD solution, although the difference was only 4.3 percent.

Looking at the detailed information in figure 4.4, we can see that the compute time of this solution is always worse than the other compute shader solutions, with the

render phase time being about as good as for the other solutions. However, we highlight that it in reality performed better than the basic solution. We think the reason why it still outperforms is that some time spent on moving data on or to the GPU does not show up in our measurements, and that time is shortened using this improved memory management solution.

It is conceivable that the solution would perform better with a more efficient algorithm for prefix sum.

5.1.4 Limitations

5.1.4.1 Underutilized buffer space

As mentioned in 3.0.1, the naive implementation manages buffers in an inefficient way. If we enable dynamic subdivision count, we can lower the amount of computations done, but we will still use the same size of buffers. If we zoom out fully, so that all curves use the smallest subdivision count of 5, we will still allocate space that could accommodate a subdivision count of 40, leading to a buffer memory utilization of approximately 12,5%.

The solution we attempted and described in 3.0.2.2, to use a cumulative sum algorithm on the subdivision counts to determine the index for each curve, had an inverse effect on the total computation time. We think that it can have different reasons. Firstly, the algorithm we use might not be efficient enough, where another algorithm would work better. Another reason might be that the extra amount of buffers used for preparatory steps causes more data transfer to and from the gpu.

5.1.4.2 Aliasing

The current implementation uses no form of anti-aliasing, which leads to visual artifacts when the camera zooms out and look at the curves from a distance, especially if many curves are close to each other. This can be seen in figure 3.1. Even zoomed in, because curves will often be long and thin, they can appear as "blocky" on the edges.

Other path-rendering solutions described in 2.9.1 might have natural anti-aliasing solutions, such as using the pixel shader to interpolate the alpha value of the colour filling at edges. Our solution ends up still suffering from the issues of long, thin, sub pixel sized triangles that do not render well.

5.1.4.3 Measurements

We take 2 types of measurements. Firstly, we determine the cutoff point where 60 fps can no longer be sustained. This measurement is not exact in that it is influenced by sporadic changes in the machines free resources. But it does give a rough idea of the general performance of each method.

As for the exact measurements of the execution time of the shaders, while they are exact, fail to capture the time taken to perform writes and reads of gpu buffer

data. We are yet to find a way to capture the execution time of these asynchronous operations. But the two types of measurements in combination still gives a good idea of the performance of each method.

5.2 Future work

One area of future work is to find an efficient scan or prefix sum algorithm that works within the context of WebGPU. This would be valuable not only for our particular implementation, but many other applications that depend on such operations.

Another area is to examine what anti-aliasing solutions would work well within the context of WebGPU and the particular case of long and thin objects. There are anti-aliasing solutions such as temporal anti-aliasing that could work well visually, but their feasibility in WebGPU should be examined.

For our particular implementation, future work is to better measure every stage of the solution to get a clearer idea of the bottlenecks and where focus should be put for maximum gain.

5.3 Ethical Considerations

There are a couple of ethical considerations for this thesis. Firstly, care had to be taken to protect sensitive and private information from TMHLS, such as identifiable data from customers. The work laid out in this thesis was done in a new, seperate codebase and all data presented was collected and fabricated for this thesis alone, by the authors.

The energy consumption of data movement is nowadays conisdered higher than computations [35], which make our solution beneficial in that data movement all the way to the GPU is cut down in favor for computations at runtime.

5.4 Summary

Even with the limitations to our current implementation, we see a big increase in performance compared to our baseline. While we have not been able to compare using test cases from other papers, it appears that our solution is comparable to other solutions in performance, but lacking in visual quality in regards to the aliasing.

Bibliography

[1] "Web gl 2.0 availability." (), [Online]. Available: https://caniuse.com/webgl2 (visited on 04/15/2024).

[2] T. K. Group. "Webgl 2.0 specification." (), [Online]. Available: https://registry.khronos.org/webgl/specs/latest/2.0/ (visited on 04/15/2024).

[3] F. Nah, "A study on tolerable waiting time: How long are web users willing to wait?," vol. 23, Jan. 2003, p. 285. DOI: 10.1080/01449290410001669914.

[4] 2. U. Technologies. "Memory in unity webgl." (), [Online]. Available: https://docs.unity3d.com/Manual/webgl-memory.html (visited on 04/16/2024).

[5] T. F. Transport and C. Agency. "Factors affecting the speed and quality of internet connection." (), [Online]. Available: https://www.traficom.fi/en/communications/broadband-and-telephone/factors-affecting-speed-and-quality-internet-connection (visited on 04/16/2024).

[6] D. Rushton. "Raster vs vector maps: What's the difference which are best?" (), [Online]. Available: https://carto.com/blog/raster-vs-vector-whats-the-difference-which-is-best (visited on 04/15/2024).

[7] N. Developer. "Life of a triangle - nvidia's logical pipeline." (), [Online]. Available: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline (visited on 04/16/2024).

[8] N. Developer. "Creating optimal meshes for ray tracing." (), [Online]. Available: https://developer.nvidia.com/blog/creating-optimal-meshes-for-ray-tracing/ (visited on 04/16/2024).

[9] W3C. "Webgpu." (), [Online]. Available: https://www.w3.org/TR/webgpu/ (visited on 04/16/2024).

[10] S. Imbleau, "Understanding hardware-accelerated 2d vector graphics," Ph.D. dissertation, May 2022. DOI: 10.13140/RG.2.2.25593.54887.

[11] T. K. Group. "Primitive." (), [Online]. Available: https://www.khronos.org/opengl/wiki/Primitive (visited on 04/17/2024).

[12] P. D. T. Weinkauf. "Bezier curves, splines and surfaces." (), [Online]. Available: https://www.kth.se/social/files/55492cacf276542be2fc547a/BezierCurvesAndSurfaces.pdf (visited on 04/18/2024).

[13] Adobe. "Vector files." (), [Online]. Available: https://www.adobe.com/creativecloud/file-types/image/vector.html (visited on 04/18/2024).

[14] T. K. Group. "Shader." (), [Online]. Available: https://www.khronos.org/opengl/wiki/Shader (visited on 01/18/2024).

[15] T. K. Group. "Compute shaders." (), [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader (visited on 05/21/2024).

[16] T. K. Group. "Low-level 3d graphics api based on opengl es." (), [Online]. Available: https://www.khronos.org/webgl/ (visited on 01/16/2024).

[17] T. K. Group. "Opengl overviews." (), [Online]. Available: https://www.khronos.org/opengl/#:~:text=OpenGL%C2%AE%20is%20the%20most,wide%20variety%20of%20computer%20platforms. (visited on 01/18/2024).

[18] "Gpu for the web community group." (), [Online]. Available: https://www.w3.org/community/gpu/ (visited on 01/15/2024).

[19] E. Fransson and J. Hermansson, *Performance comparison of webgpu and webgl in the godot game engine*, 2023.

[20] B. Barnhart. "The birth of bézier curves and how it shaped graphic design." (), [Online]. Available: https://www.linearity.io/blog/bezier-curves/ (visited on 05/22/2024).

[21] T. Akenine-Moller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. "Real-time rendering fourth edition." (2018).

[22] M. NieBner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer, "Real-time rendering techniques with hardware tessellation," *Comput. Graph. Forum*, vol. 35, no. 1, pp. 113–137, Feb. 2016, ISSN: 0167-7055. DOI: 10.1111/cgf.12714. [Online]. Available: https://doi.org/10.1111/cgf.12714.

[23] M. Löwgren and N. Olin, *Pn-triangle tessellation using geometry shaders : The effect on rendering speed compared to the fixed function tessellator*, 2010.

[24] I. Cantlay, "Directx 11 terrain tessellation," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:15040902.

[25] J.-N. Khoury, J. Dupuy, and C. Riccio, "Adaptive gpu tessellation with compute shaders," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:198116975.

[26] J.-N. Khoury, "Gpu tessellation with compute shaders," 2018.

[27] M. J. Kilgard, "Polar stroking: New theory and methods for stroking paths," *CoRR*, vol. abs/2007.00308, 2020. arXiv: 2007.00308. [Online]. Available: https://arxiv.org/abs/2007.00308.

[28] C. Loop and J. Blinn, "Resolution independent curve rendering using programmable graphics hardware," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1000–1009, Jul. 2005, ISSN: 0730-0301. DOI: 10.1145/1073204.1073303. [Online]. Available: https://doi.org/10.1145/1073204.1073303.

[29] Y. Kokojima, K. Sugita, T. Saito, and T. Takemoto, "Resolution independent rendering of deformable vector objects using graphics hardware," in *ACM SIGGRAPH 2006 Sketches*, ser. SIGGRAPH '06, Boston, Massachusetts: Association for Computing Machinery, 2006, 118–es, ISBN: 1595933646. DOI: 10.1145/1179849.1179997. [Online]. Available: https://doi.org/10.1145/1179849.1179997.

[30] M. Kilgard and J. Bolz, "Gpu-accelerated path rendering," *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)*, vol. 31, no. 6, to appear, Nov. 2012.

[31] 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.

[32] F. Ganacim, R. S. Lima, L. H. de Figueiredo, and D. Nehab, "Massively-parallel vector graphics," *ACM Trans. Graph.*, vol. 33, no. 6, Nov. 2014, ISSN: 0730-0301. DOI: 10.1145/2661229.2661274. [Online]. Available: https://doi.org/10.1145/2661229.2661274.

[33] 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.

[34] I. Epic Games. "Nanite virtualized geometry." (), [Online]. Available: https://dev.epicgames.com/documentation/es-mx/unreal-engine/nanite-virtualized-geometry-in-unreal-engine (visited on 01/20/2024).

[35] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal"' for computer architecture," *Computing in Science Engineering*, vol. 15, pp. 16–26, Nov. 2013. DOI: 10.1109/MCSE.2013.95.

A

Appendix 1