UiT The Arctic University of Norway

Department of Computer Science and Computational Engineering

Meshless Animation Framework

Gustav Adolf Johansen

Master’s thesis in Applied Computer Science, SHO6264, June 2020

Abstract

This report details the implementation of a meshless animation framework for blending surfaces. The framework is meshless in the sense that only the control points are handled on the CPU, and the surface evaluation is delegated to the GPU using the tessellation shader steps. The framework handles regular grids and some forms of irregular grids.

Different ways of handling the evaluation of the local surfaces are investigated. Directly evaluating them on the GPU or pre-evaluating them and only sampling the data on the GPU. Four different methods for pre-evaluation are presented, and the surface accuracy of each one is tested.

The framework contains two methods for adaptively setting the level of detail on the GPU depending on position of the camera, using a view-based metric and a pixel-accurate rendering method. For both methods the pixel-accuracy and triangle size is tested and compared with static tessellation.

Benchmarking results from the framework are presented. With and without animation, with different local surface types, and different resolution on the pre-evaluated data.

Acknowledgements

I would like to thank my supervisors Jostein Bratlie and Rune Dalmo, for helping me understand the problem, and providing valuable feedback while writing the report.

Contents

List of Tables v

List of Figures v

1 Introduction 1
1.1 Problem Description . . . 1
1.2 Tasks . . . 2
1.3 Limitations . . . 2
1.4 Related Work . . . 3
1.4.1 Hardware Tessellation . . . 3
1.4.2 Blending Splines . . . 3
1.5 Pixel-Accurate Rendering . . . 4
1.6 Patch Culling . . . 4

2 Methods & Technology 5
2.1 Hardware Tessellation . . . 5
2.2 Bézier Patchwork . . . 6
2.3 Blending Surfaces . . . 7
2.4 Lattice . . . 9
2.5 OpenMesh . . . 11
2.6 GLM . . . 12
2.7 Vulkan . . . 12
2.8 Dear ImGui . . . 13
2.9 Adaptive Level of Detail . . . 13

3 Implementation 15
3.1 Source Code . . . 15
3.2 Base Vulkan Framework . . . 16
3.3 Bézier Patchwork . . . 17
3.4 Lattice . . . 18
3.4.1 OpenMesh . . . 18
3.4.2 Adding Patches . . . 19
3.4.3 Induce Lattice . . . 20
3.4.4 Local Surfaces, Loci and Patches . . . 21
3.4.5 T-Loci . . . 23
3.4.6 Data . . . 24
3.5 VulkanLattice . . . 25
3.5.1 Vulkan . . . 25
3.5.2 Shaders . . . 27
3.5.3 Direct Evaluation . . . 29
3.5.4 Pre-Evaluation Using Images . . . 30
3.5.5 Pre-Evaluation Using Batched Images . . . 30

3.5.6 Pre-Evaluation Using Buffers . . . 32
3.5.7 Normals . . . 32
3.5.8 Surface Accuracy Display . . . 32
3.5.9 Pixel-Accuracy Display . . . 32
3.5.10 Triangle Size Display . . . 33
3.6 LatticeExample . . . 33
3.7 Adaptive Level of Detail . . . 33
3.8 Animation . . . 34

4 Testing Setup **35**
4.1 Hardware . . . 35
4.2 Benchmarking . . . 35
4.3 Direct Evaluation . . . 36
4.4 Surface Accuracy . . . 37
4.5 Pixel-Accuracy . . . 38

5 Results **39**
5.1 Bézier Patchwork . . . 39
5.2 Lattice Rendering . . . 41
5.3 Lattice Benchmarks . . . 47
5.4 Animation . . . 57
5.5 GPU Memory Usage . . . 58
5.6 Surface Accuracy . . . 60
5.7 Pixel-Accuracy . . . 64

6 Discussion **68**
6.1 Bézier Patchwork . . . 68
6.2 Lattice Framework . . . 68
6.3 Evaluation Methods . . . 69
6.4 Adaptive Level of Detail . . . 70
6.5 Future Work . . . 70
6.5.1 Lattice . . . 70
6.5.2 Vulkan . . . 71
6.5.3 Patch Culling . . . 71

References **73**

Appendix A GUI **76**

Appendix B Setup Guide **79**

Appendix C Problem Description **82**

List of Tables

1 Computer Specifications . . . 35
2 Bézier Patchwork Rendering Statistics . . . 40
3 Bézier Patchworks Benchmarks . . . 40
4 Bézier Patchworks Animation Benchmarks . . . 41
5 Benchmarks of Direct Evaluation . . . 47
6 Benchmarks of Direct Evaluation . . . 47
7 Benchmarks of Direct Evaluation . . . 48
8 Pre-Evaluation Using Images Benchmarks . . . 50
9 Pre-Evaluation Using Images Benchmarks . . . 50
10 Pre-Evaluation Using Batched Images Benchmarks . . . 52
11 Pre-Evaluation Using Buffer Benchmarks . . . 54
12 Pre-Evaluation Using Buffers with No Interpolation Benchmarks . . . 54
13 Animation Benchmarks . . . 57
14 Animation Benchmarks . . . 57
15 Statistics From the Different Level of Detail Methods . . . 67

List of Figures

1 Traditional and Tessellation Pipeline Comparison . . . 1
2 GPU Shader Stages . . . 5
3 Quad Patch Tessellation . . . 6
4 Bi-cubic Bézier Surface Tessellation Pipeline . . . 7
5 B-Functions Plot . . . 8
6 Lattice Grid Points . . . 9
7 Halfedge Data Structure . . . 11
8 Edge Surrounding Sphere for Setting Tessellation Levels . . . 13
9 Problems With Not Using the Surrounding Sphere of the Edge . . . 14
10 UML Class Diagram . . . 15
11 Vulkan API Overview . . . 16
12 Local Surface Control Point Creation Methods . . . 22
13 Handling T-loci in lattice grid . . . 24
14 Texture Containing Pre-Evaluated Local Surface Data . . . 30
15 Texture Containing Pre-Evaluated Patch Data . . . 31
16 Benchmarking Setup . . . 36
17 Surface Accuracy Test Setup . . . 37
18 Pixel-Accuracy and Triangle Size Display Test Setup . . . 38
19 Bézier Patchwork Rendering . . . 39
20 Bézier Patchwork Rendering With 100 Models . . . 39
21 Bézier Patchwork Wireframe Rendering . . . 40
22 Lattice Grid Rendering . . . 41
23 Local Surface Rendering . . . 42

24 Lattice Rendering . . . 43
25 Lattice Rendering With Grid . . . 44
26 Lattice Rendering With Local Surfaces . . . 45
27 Lattice Normals Rendering . . . 45
28 Irregular Grid Lattice Rendering . . . 46
29 Direct Evaluation Benchmarks . . . 49
30 Pre-Evaluation Using Images Benchmarks . . . 51
31 Pre-Evaluation Using Batched Images Benchmarks . . . 53
32 Pre-Evaluation Using Buffer Benchmarks . . . 55
33 Evaluation Methods Comparison . . . 56
34 Animation Benchmarks . . . 58
35 Evaluation Method Memory Comparison Diagram . . . 60
36 Pre-Evaluated Image Surface Accuracy . . . 60
37 Pre-Evaluated Batched Image Surface Accuracy . . . 61
38 Pre-Evaluated Buffer Surface Accuracy . . . 61
39 Pre-Evaluated Batch & Buffer Perfect Surface Accuracy . . . 62
40 Pre-Evaluated Buffer No Interpolation Surface Accuracy . . . 63
41 Pixel-Accuracy and Triangle Size Display of Static Tessellation Levels . . . 64
42 Pixel-Accuracy and Triangle Size Display of Dynamic Tessellation . . . 65
43 Pixel-Accuracy and Triangle Size Display of Pixel-Accurate Rendering . . . 66
44 Flat Surface Rendered in Wireframe With Different Level of Detail Methods 67
45 VulkanLattice Menu Options . . . 76
46 LatticeExample Menu Options . . . 77
47 CMake Shaderc Options . . . 80

1 Introduction

1.1 Problem Description

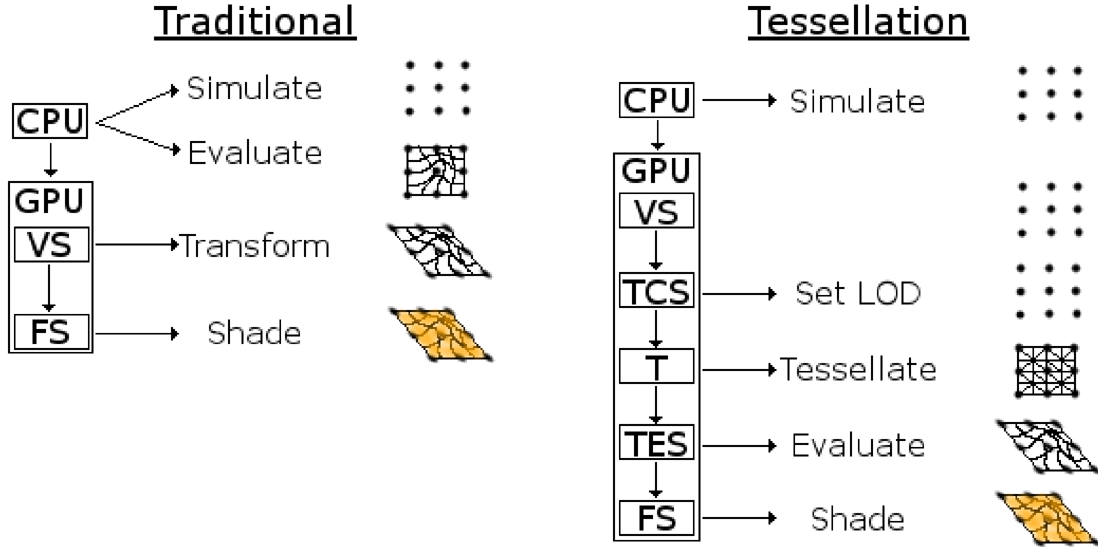


Figure 1: A comparison of the different tasks that would go into rendering a surface on the traditional and tessellation pipelines when rendering a surface. Given a set of control points, the surface is evaluated and shaded, an idea of how the geometry could look at each stage is also shown. VS, TCS, TES and FS are the vertex, tessellation control, tessellation evaluation and fragment shaders. T is the fixed function tessellator.

The traditional way of animating surfaces involves changing the coefficients and re-evaluating the surface on the CPU. Every frame these control points then needs to be re-uploaded to GPU memory. This approach contains two possible bottlenecks. The surface evaluation can be a costly operation depending on how densely sampled the surface is, and the sampled points have to be re-uploaded to the GPU every frame.

With the introduction of OpenGL 4.0 and DirectX 11 tessellation shader steps were added to the graphics pipeline. The tessellation shaders are perfect for surface rendering. Given only the control points as input, a densely tessellated domain can be created. And then the surface is evaluated using the highly parallelized architecture of the GPU. Figure 1 shows an overview of what tasks would go into both methods, and which parts of the CPU/GPU would handle each task. An overview of how the geometry could look at each stage is also shown.

A surface construction that can really benefit from this is the blending type surface construction. This type of surface allows blending not only points, but also functions, e.g. other surfaces. However, this means that not only the global surface have to be evaluated, but also the control surfaces. With this type of surface construction it is possible to change the surface by not only translating the control surfaces, but also

rotating and scaling them.

With this approach, the evaluation (rendering) and the simulation (animation) can reside in separate systems, and the evaluation system is constant/stable with respect to changing coefficients from the simulation system. A composed framework will be meshless, meaning that the meshing is delegated to the graphics API through the tessellation shaders.

With the surface evaluation being done in the shaders, the local surfaces will still have to be evaluated there, and different evaluation kernels will have to be made to allow different local surfaces to be used. A more general approach could pre-evaluate the local surfaces and then only sample them inside the tessellation shaders, making the approach agnostic with respect to the local surface type. Moreover, it is possible that this could also lead to increased performance on the surface evaluation.

1.2 Tasks

Considering the tensor product Bézier implementation examples from Nvidia[1], and the provided prototype[2]. The tasks include:

- Compose patchwork from Bézier patches on the GPU (tessellation shader steps).
- Tensor product blending splines (on the GPU):
  - Send coefficients to the shader.
  - Perform tessellation and pixel-accurate rendering.
- Simulation (e.g. animations) by changing the coefficients.
- Regular and irregular grids.
- Pre-evaluation of the local surface.

1.3 Limitations

In this work, only the quad tessellation primitive and tensor-product surfaces are considered. Some work has been done for handling irregular grids. This is limited to T-points, star-points are not considered. When it comes to animating the surface, only affine transformations on the local surfaces' model matrix is performed. The local surface control points are kept constant during the rendering.

Vulkan is used as the rendering framework, but the program does not seek to be the most optimal Vulkan implementation. A base framework is used to take care of all the setup and provide some helper classes. No multi-threading or custom memory allocators are used.

1.4 Related Work

1.4.1 Hardware Tessellation

Since its introduction the tessellation shader steps have found several applications in academia, movie production and video games. One area where the tessellation shaders have become popular is with techniques that use quad patches that are then offset by some simulation and/or displacements maps, like terrain[1][3][4] or water[5] rendering.

A survey of the research done with hardware tessellation is done in [6]. This state of the art article covers topics such as smooth surface rendering, adaptive level-of-detail (LOD), displacement mapping and culling techniques. Much of the work done had been to implement techniques that were previously only available to offline rendering. One of the most commonly used surface representations used for movie production is subdivision surfaces. In [7] a method for rendering Catmull-Clark subdivision surfaces was presented. The method is exact and implements the full RenderMan specification of Catmull-Clark surfaces. The method was called Feature Adaptive Subdivision, or FAS for short.

An example of FAS put into practice can be found in Pixar's OpenSubdiv[8]. OpenSubdiv is a set of open source libraries that implement high performance subdivision surface evaluation on massively parallel CPU and GPU architectures. Subdivision surfaces are commonly used for offline rendering tasks like animation films. Before this, the artists would have to work with a polygonal hull, instead of a smooth accurate limit surface. Using the GPU to render the smooth, accurate limit surface is a big improvement.

In another article [9], another approach for rendering subdivision surfaces was presented. This method outperforms FAS and was implemented into a production game engine.

Hardware tessellation has become the de-facto standard nowadays when it comes to terrain rendering[3]. Quads are sent to the GPU where they are tessellated and displaced depending on some given displacements maps. Using this approach also makes it very easy to change the LOD of a given patch when it is far from the camera, or increase it when it is close.

In [5] a method for sea surface simulation using the tessellation shaders is presented. The tessellation shaders are used to dynamically set the LOD, increasing the fineness of the sea surface grid, thereby enhancing the sea surface rendering.

1.4.2 Blending Splines

In [10] an approach for constructing blending surfaces on irregular grids was presented. The surfaces based on irregular grids can be regarded as a collection of surfaces on regular grids that are connected at the edges and the corners in a smooth, but irregular way. This involves T-junctions and star-junctions. To make this work, Lakså expressed generic blending functions, including GERBS, in terms of classical B-splines.

The work in [10] was one of the works that lead to [2] where a method for evaluating smooth blending surfaces on the GPU using the tessellation shader steps were presented. These articles[10][2] are primarily what this project is based on.

1.5 Pixel-Accurate Rendering

In [11] a single-pass method guaranteeing pixel-accuracy in interactive rendering applications is presented. The screen space distance between the tessellated surface and the corresponding surface point is used as an error metric. The algorithm can be used for any C^2-continuous surface with bounds on the second order derivatives. In [12] Hjelmervik et al. present an approach to have pixel-accurate rendering at interactive frames for LR- and T-Splines.

Two other articles [13] and [14] uses an approach with calculating the error estimate using slefe boxes, without actually computing them. This approach uses a compute shader pre-pass that needs to run every time the animation of the surface or the view changes. The approach requires, depending on the model, between 1% and 5% extra work. However, by avoiding over-tessellation, pixel-accurate rendering is often faster than rendering based on heuristics.

1.6 Patch Culling

In [15] an effective technique for back-patch culling for hardware tessellation is presented. They present a novel approach using the Bézier convex hull of the parametric tangent plane. The approach is both more effective and more efficient that the popular cone-of-normals approach. Another article[16] looks at several different methods of culling geometry on the GPU, this one specifically for NURBS surfaces. The article looks at many different applications/methods including higher-order surface rendering, adaptive tessellation, displacement mapping and patch culling.

2 Methods & Technology

2.1 Hardware Tessellation

With OpenGL 4.0 and DirectX 11 three new shader steps were added to the graphics pipeline. Using OpenGL/Vulkan terminology these three steps are the programmable tessellation control shader (TCS), the fixed-function tessellation primitive generator, and the programmable tessellation evaluation shader (TES). Figure 2 shows an overview of the current graphics pipeline shader stages with the tessellation steps marked in green and purple.

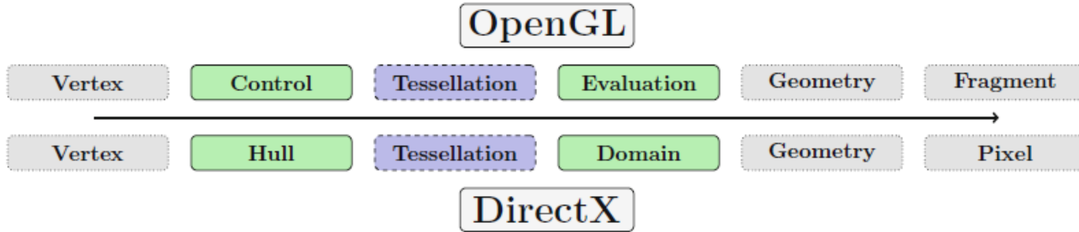


Figure 2: The GPU shader stages as of OpenGL 4.0 and DirectX 11. The shaders are ordered from left to right. The programmable tessellation shaders are marked in green, and the fixed-function tessellation stage is marked in purple. Source: [2].

The fixed-function tessellator consumes each input patch (after vertex shading) and produces a new set of independent primitives (points, lines or triangles)[17]. The amount of tessellation is determined by the tessellation levels set in the TCS. Some execution modes must also be set in either the TCS or TES. One execution mode specifies the type of subdivision and topology, the three possible values being triangles, quads and isolines. Only the quad mode is considered in this work, a visualization of the output of a quad type primitive is shown in Figure 3. Each vertex produced by the tessellator has an associated (u,v) position in normalized parameter space, with parameter values in the range [0,1]. For the quad, all four outer and two inner tessellation levels are relevant and can be set individually.

Another execution mode controls the spacing of the segments along the edges of the patch. The three modes are spacing_equal, fractional_even_spacing and fractional_odd_spacing. Equal spacing clamps the tessellation level to an integer in the range [1, max], where max is implementation dependent, but must be at least 64[18]. For the other two modes, the level is clamped to the range [2, max] and [1, max - 1] and rounded up to the nearest even/odd integer for even or odd spacing respectively. The edge is then subdivided into n - 2 segments of equal length, and two equal but smaller segments. The inner rectangle is subdivided uniformly. The even/odd spacing create a smoother transition between tessellation levels, but does not work for creating a watertight tessellation between patches of different edge sizes. The vertex winding order can also be specified.

Based on the tessellation levels set in the TCS, the fixed-function tessellator will

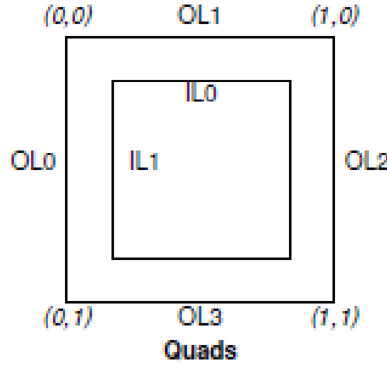


Figure 3: A quad patch with a top left origin, showing where the relevant inner(IL) and outer(OL) tessellation levels are. Source: [17].

then create a tessellated domain based on those levels. Then for each generated vertex, the tessellator will send the normalized (u,v) coordinates to the TES, together with the output from the TCS. The TES uses the normalized (u,v) coordinates from the tessellator to transform the position of the created vertices. If any of the relevant outer tessellation levels are set to 0, no vertices will be created in the tessellator, effectively culling the patch. The triangles created by the tessellator are generated with a topology similar to triangle lists[17].

2.2 Bézier Patchwork

A tensor-product Bézier surface of degree d is given by:

$$P(u,v) = \sum_{i=0}^{d} \sum_{j=0}^{d} c_{i,j} B_j^d(v) B_i^d(u), \tag{1}$$

where $c_{i,j}$ are the $(d+1)^2$ control points of the surface, and B_k^d are the Bernstein polynomials of degree d given by:

$$B_k^d(t) = \binom{d}{k} (1-t)^{d-k} t^k \tag{2}$$

P(u,v) is a mapping from the domain $[0,1]^2$ in $\mathbb{R}^2$ to $\mathbb{R}^3$. This construction maps perfectly to the TES, as the vertices of the quad patches are in the same domain. In order to maintain a C^0-continuous surface the $(d+1)$ control points of an edge along two neighbouring patches must be the same. However, the cross-boundary derivatives may not be constructed equally, which can produce different normal vectors along the shared edges[6]. This problem can be solved by instead using a B-spline construction.

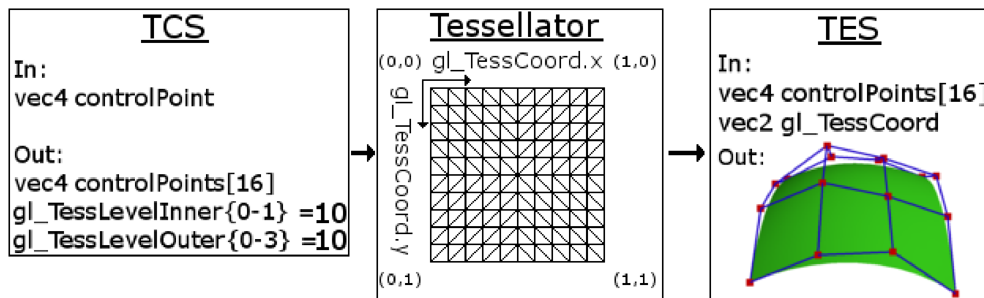


Figure 4: An example of what goes into the tessellation shader steps for a bi-cubic Bézier surface. The control points are input to the TCS one at a time until it reaches 16. Then the Tessellator uses the gl_TessLevelInner/Outer to create a triangulated domain. The vertices are transformed by the TES using the parameter values passed from the tessellator as gl_TessCoord together with any other output from the TCS. Here the TES would evaluate the surface using the 16 control points. A patch is displayed in green, with its control points in red.

Bézier surfaces of degree 2 and 3 are commonly used for rendering because of their simple construction. Figure 4 shows how the tessellation shader steps for rendering a bi-cubic Bézier surface could look.

2.3 Blending Surfaces

Instead of blending points, like in the case of Bézier and B-splines. The blending surface construction blends scalars, points or functions (like other surfaces) using B-functions. A B-function is a C^k-smooth function that has the following properties:
1. $B\colon I \to I \quad (I = [0,1] \subset \mathbb{R})$,
2. $B(0) = 0$,
3. $B(1) = 1$,
4. $B'(t) >= 0$, $t \in I$,
5. $B(t) + B(1 - t) = 1$, $t \in I$.

The simplest B-function is just $B(t) = t$, which gives a linear blending. Figure 5 shows the plots of three different blending functions and their derivatives. Here B_k means that the blending function will produce a C^k-smooth blending surface everywhere except the knot vectors. At the knot vectors the blending surface will fully interpolate the local surface and all its derivatives. Notice how the logistic expo-rational blending function bends much quicker than the others, and is therefore better suitable for local approximation. One negative is that it is not defined at the endpoints of the interval, so any implementation has to check for the endpoints and return those values manually, and branching is not good inside shaders.

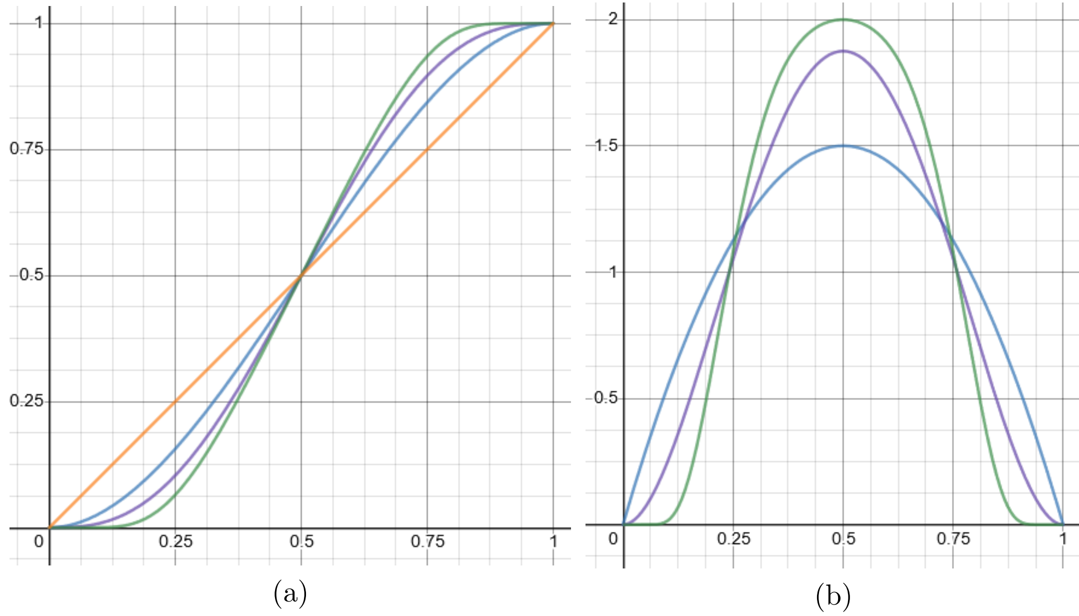


Figure 5: Some B-functions (a) and their derivatives (b) plotted on the interval [0,1]. The B-functions are $B_0(t) = t$ in orange, the polynomial functions $B_1(t) = 3t^2 - 2t^3$ in blue and $B_2(t) = 6t^5 - 15t^4 + 10t^3$ in purple and the logistic expo-rational B-function $B_\infty(t) = \frac{1}{1+e^{(\frac{1}{t} - \frac{1}{1-t})}}$ in green.

In [10], Laksâ expressed generic blending functions, including GERBS (Generalized Expo-Rational B-Spline), in terms of classic B-splines:

$$\mathbb{B}_{d,k}(t) = B \circ \omega_{d,k}(t) b_{d-1,k}(t) + (1 - B \circ \omega_{d,k+1}(t)) b_{d-1,k+1}(t) \tag{3}$$

where $\omega_{d,i}(t) = \frac{t-t_i}{t_{i+d}-t_i}$, $b_{0,i}(t) = \begin{cases} 1; & \text{if } t_i \leq t < t_{i+1}, \\ 0; & \text{otherwise,} \end{cases}$, and B is a blending function.

A tensor-product blending surface is a 1st degree tensor-product B-spline surface adjusted with a B-function, given by:

$$S(u,v) = \sum_{j=1}^{n} \sum_{i=1}^{m} \ell_{i,j}(u,v) \mathbb{B}_{1,i}(u) \mathbb{B}_{1,j}(v) \tag{4}$$

The blending surface basis functions has minimum support over two knot intervals. The knots around the boundary have a multiplicity of 2, therefore the local surfaces in the corners and along the edges cover only 1 and 2 patches, respectively. The inner local surfaces cover all its adjacent patches.

In addition to moving the control points of the local surfaces to change the overall shape of the blending surface. This construction makes it possible to perform affine transformations on the local surfaces to animate the surface[19]. Thus, when modifying the local geometry to animate the surface, it allows not only translation, but also rotation and scaling.

2.4 Lattice

Based on the new definitions presented in [10], [2] presents a way of evaluating blending surfaces on the GPU using the tessellation shader steps. The following definitions were given:
-*Render lattice* to describe a grid structure arising from the net of spline knots.
-*Render locus* to describe loci in the render lattice, closely related to spline knots and regular-, T- and star-points.
-*Render patch*[1] to describe each line or face in a render lattice, i.e. the subset of the lattice that will be handled by a patch-type primitive.
Here regular, T- and star-points are defined as they were in [10]. Figure 6 shows an illustration of each type of point. The points are defined as:
-A T-point is defined as a grid (parameter) line ending in an orthogonal grid line.
-A *Star*-point is defined as a point where several grid lines meets in a non-orthogonal way.

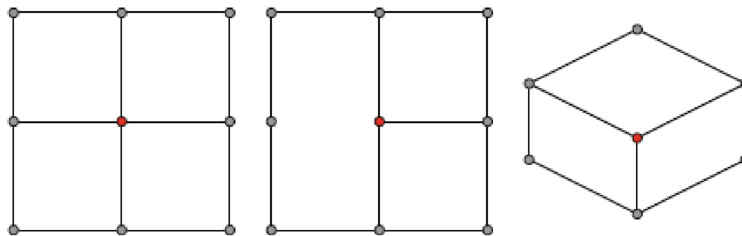


Figure 6: From left to right, a regular point, T-point and star-point marked in red on a grid. Source: [2].

In this work the star-points are not considered. Based on the locus definition above, the following locus definitions are used in this work:
-*Corner Locus* for a locus in the corner of the lattice.
-*Boundary Locus* for a locus along the boundary of the lattice.
-*Inner Locus* for a locus inside the lattice.
-*T-locus*, defined as the T-point above.
-*Terminal Locus* for the loci connected to the *T-locus* along the same parameter line.

In the regular tensor-product blending spline formula in (3) and (4), the knot vectors are used to map from the global domain over to the local domain in [0,1] for the current knot interval. When using the TES the domain is already $[0,1]^2$ and we do not need to use the knot vectors. Any patch will be affected by the 4 local surfaces of the patch's adjacent loci. Here we do need some kind of function to map from the domain of the patch, such that only the parts of the local surface that falls within the patch will be

[1] Originally referred to as *Render block* in [2], but later changed to *Render Patch*.

evaluated. Given this, the formula from (4) can be simplified to:

$$S(u,v) = \sum_{j=1}^{2}\sum_{i=1}^{2} \ell_{i,j} \circ \omega_{i,j}(u,v)B(u)B(v) \tag{5}$$

where $u, v \in [0,1]$ are the patch parameters, $B(t)$ is a B-function, $\ell_{i,j}$ are the local surfaces, and $\omega_{i,j}(u,v)$ are "map-to-local" functions mapping the domain of the patch to the domain of the local surface for the current patch given by:

$$\omega_{i,j}(t) = (1-t)s_{i,j} + te_{i,j} \tag{6}$$

where $s, e \in [0,1]$ are the start and end of the domain of the local surface for that patch.

For a given patch, let $p_{00}(u,v) = \ell_{00} \circ \omega(u,v)$, $p_{10}(u,v) = \ell_{10} \circ \omega(u,v)$, $p_{01}(u,v) = \ell_{01} \circ \omega(u,v)$ and $p_{11}(u,v) = \ell_{11} \circ \omega(u,v)$ be the local surfaces evaluated with their transformed local coordinates. Then the blending surface and the first order partial derivatives are given by

$$\begin{aligned}
c_0(u,v) &= p_{10}(u,v) + (p_{00}(u,v) - p_{10}(u,v)) \cdot (1 - B(u)),\\
c_1(u,v) &= p_{11}(u,v) + (p_{01}(u,v) - p_{11}(u,v)) \cdot (1 - B(u)),\\
c_{0u}(u,v) &= p_{10u}(u,v) + (p_{00u}(u,v) - p_{10u}(u,v)) \cdot (1 - B(u)) + (p_{00}(u,v) - p_{10}(u,v)) \cdot -B'(u),\\
c_{1u}(u,v) &= p_{11u}(u,v) + (p_{01u}(u,v) - p_{11u}(u,v)) \cdot (1 - B(u)) + (p_{01}(u,v) - p_{11}(u,v)) \cdot -B'(u),\\
c_{0v}(u,v) &= p_{10v}(u,v) + (p_{00v}(u,v) - p_{10v}(u,v)) \cdot (1 - B(u)),\\
c_{1v}(u,v) &= p_{11v}(u,v) + (p_{01v}(u,v) - p_{11v}(u,v)) \cdot (1 - B(u)),
\end{aligned}$$

and

$$\begin{aligned}
S(u,v) &= c_1(u,v) + (c_0(u,v) - c_1(u,v)) \cdot (1 - B(v)),\\
S_u(u,v) &= c_{1u}(u,v) + (c_{0u}(u,v) - c_{1u}(u,v)) \cdot (1 - B(v)),\\
S_v(u,v) &= c_{1v}(u,v) + (c_{0v}(u,v) - c_{1v}(u,v)) \cdot (1 - B(v)) + (c_0(u,v) - c_1(u,v)) \cdot -B'(v).
\end{aligned}$$

In [2] two different strategies for evaluating the local surfaces are proposed. The first one is using a general evaluation scheme of pre-sampled data. And the second is directly evaluating the local surfaces in the TES, writing specialized shaders for each local surface type.

For CPU evaluation to allow editing or animation of the surface, pre-evaluation and optimal organization of the recalculations has to be used to achieve interactive rendering times[20]. For the GPU evaluation the spline representation is kept all the way to the GPU and must be re-evaluated in the TES every frame. This means the rendering time is constant and predictable, even when changing the coefficients.

2.5 OpenMesh

OpenMesh is a generic and efficient library for representing and manipulating polygonal meshes[21][22]. It is possible to choose between a triangular or polygonal mesh data structure for holding the geometry. A tri-mesh only allows faces with a valence of 3, but poly-mesh can hold faces of arbitrary valence.

The halfedge data structure is used for storing the relationships between vertices, edges and faces in OpenMesh. Figure 7 shows the idea behind the halfedge data structure. The edges are split in half into two directed edges pointing at the vertices at opposite ends. Each halfedge also references its opposite halfedge, the vertex it points at, the next halfedge in the face and the face it is connected to. Optionally the halfedge can also hold a reference to its previous halfedge. If this property is not present, the edge has to iterate over all the next halfedges until it reaches the previous one. The vertices references one outgoing halfedge, and the faces reference one halfedge. With this structure iterating over the neighbourhood around a vertex/face becomes very easy, and OpenMesh contains a lot of iterators/circulators to perform these tasks.

1. Vertex ↦ one outgoing halfedge,
2. Face ↦ one halfedge,
3. Halfedge ↦ target vertex,
4. Halfedge ↦ its face,
5. Halfedge ↦ next halfedge,
6. Halfedge ↦ opposite halfedge (implicit),
7. Halfedge ↦ previous halfedge (optional).

Figure 7: The halfedge data structure. Shows the relationship between the vertices, edges and faces. Source: [21].

OpenMesh supports a great deal of customization. The base OpenMesh class takes in a template parameter containing the traits of the class, these traits specifies which attributes are stored for each geometric part. As default the class only stores the position of the types, but the attributes can be used to also store color, normals, etc. Attributes are also used to store the optional previous halfedge reference.

The user can also add additional user-defined properties to each type by using the property handles. It is possible to provide custom types for the vertices, edges and faces, or you can use the predefined structures of OpenMesh. OpenMesh does not contain a matrix type.

2.6 GLM

OpenGL Mathematics(GLM) is a header only C++ mathematics library for graphics software[23]. GLM provides 3x3 and 4x4 matrix types, and geometrical functions to perform affine transformations on the matrices. For the matrix transformations such as translation, rotation and scaling, 3-component GLM vectors must be used. GLM provides hashing functions for their 3- and 4-component vectors, making it possible to use them for keys to hash-maps.

2.7 Vulkan

Vulkan is a new generation graphics and compute Application Programming Interface (API) from Khronos that provides high-efficiency, cross-platform access to modern GPUs used in a wide variety of devices from PCs and consoles to mobile phones and embedded platforms[24]. The group behind Vulkan is the same group that maintains the OpenGL specification.

Vulkan is a lower level and more verbose API than OpenGL, giving the user more control at the cost of increased complexity. Work is done by submitting pre-recorded command buffers to a queue. Most of the state, e.g. graphics pipelines, must be set before the rendering begins. Memory must be allocated and copied by the user explicitly. Synchronization becomes the users responsibility.

The design of Vulkan is layered, meaning that different layers can be added on top of the base API to extend functionality. Layers are added in between the API call and the execution and can add functionality like validation and debug information. These layers can then be removed for release builds, adding no extra cost.

In [25] a study on the performance of Vulkan is looked at, as well as some comparisons between Vulkan and OpenGL. Vulkan offers less CPU overhead by eliminating expensive driver operations, and more direct control over the GPU than OpenGL.

One of the recurring complaints about OpenGL is the inconsistent compiling of GLSL shaders between implementations[25]. Vulkan uses an intermediate language for defining shaders called the Standard, Portable Intermediate Representation - V (SPIR-V)[26]. A number of different compilers have been made to compile other shader languages like GLSL and HLSL into SPIR-V, giving the user the freedom to choose which shader language to use.

Glslang[27] can be used to compile GLSL into SPIR-V through the command line. Shaderc[28] wraps around the core functionality in glslang and provides a library for compiling shaders into SPIR-V inside the project. They both provide language validation that is helpful for avoiding errors.

Vulkan provides the possibility to perform queries during the rendering pipeline. These queries include things like how many vertices and primitives were input to the pipeline, how many invocations of a given shader, how many primitives were input to the clipping stage, and how many passed. It is also possible to capture timepoints at a given stage in the pipeline. You could get the timepoint at the beginning and end of the pipeline and use those to calculate how much time the GPU spent on rendering.

2.8 Dear ImGui

Dear ImGui is a graphical user interface library for C++[29]. ImGui creates vertex buffers that can be rendered in a 3D-pipeline, and provides example files for Vulkan so that it can easily be implemented in a Vulkan rendering application. The library provides a lot of controls that can be used for controlling the application and display statistics.

2.9 Adaptive Level of Detail

In an interactive application where the camera is free to move around it is not possible to achieve good results using static tessellation factors. Setting them too low will make the surface appear non-smooth when getting too close, but setting it too high will decrease performance. To get the best mix of visual quality and performance the tessellation factors should be set inside the TCS depending on the camera's position and the geometry of the surface. Also, to avoid cracks in the surface, two neighbouring patches must share the same edge tessellation factor.

Using equal_spacing for the tessellation gives a more uniform subdivision, but when adaptively changing the tessellation levels, the fractional spacing options will give a smoother transition between different levels. For adaptive level-of-detail the tessellation levels of the shared edge between two neighbouring patches must be the same, the implementation guarantees that this will happen for edges of the same length. However, for an irregular grid, there is no guarantee that the segments will be layed out equally for edges of different lengths, and the equal spacing must be used to avoid cracks.

In [1] a method for setting the tessellation factors depending on some desired number of triangles per edge is presented. The method uses the diameter of the patch edge's bounding sphere projected to screen-space as a metric to set the tessellation level of that edge (see Figure 8). Using the boundary sphere makes the method rotation invariant. Using just the edge length can lead to under tessellation when the edge is planar towards the camera (Figure 9), as the patch vertices might be displaced out of the plane.

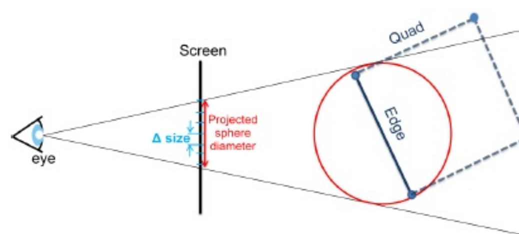


Figure 8: Setting the tessellation levels dynamically based on the camera. The diameter of the edge's surrounding sphere is used to set the tessellation factor based on a given target pixels per edge. Source: [1].

Figure 9: Using the edge's surrounding sphere for setting the tessellation levels makes it rotation invariant. Using just the projected length of the edge can create problems like the one seen here. The edge's length is in screen space is small and therefore the tessellation levels are set low. However, the actual geometry of the patch is displaced out of the patch, actually requiring higher tessellation levels to reach the target pixels per edge. Source: [1].

A more accurate approach to adaptive tessellation is pixel-accurate rendering. The concept of pixel-accurate rendering is used to determine a tessellation factor that guarantees a tessellation of a surface differs from the true surface, when measured in screen-space pixel coordinates, by at most half a pixel[6]. The resulting surface will be displayed on a pixel screen, meaning that any error between the true and tessellated surface less than half a pixel will not be noticeable. If the distance between this point and the equivalent point of the true surface projected to screen space is less than half a pixel, then that point will be pixel-accurate.

In [11] a method for pixel-accurate rendering using the bounds of the second order derivatives is presented. The formula used for computing the tessellation levels such that the surface is rendered pixel accurate is:

$$\frac{8\epsilon^x}{A} \geq \Delta_u^2 \left(\frac{\lceil z_{uu} \rceil \lceil x \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil x_{uu} \rceil}{\lfloor z \rfloor} \right) + \Delta_v^2 \left(\frac{\lceil z_{vv} \rceil \lceil x \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil x_{vv} \rceil}{\lfloor z \rfloor} \right) + 2\Delta_u \Delta_v \left(\frac{\lceil z_{uv} \rceil \lceil x \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil x_{uv} \rceil}{\lfloor z \rfloor} \right),$$

where ϵ^x is the error tolerance, setting this to be equal to 0.5/window_width will guarantee that the error will be less than half a pixel. A is the first element of the projection matrix. Δ_u and Δ_v are the sampling distances. x and z are the coordinates/derivatives in eye space, and $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ denotes the maximal and minimal absolute value respectively. The article proposes to start with the most dense boundary tessellation in each parameter direction, i.e., 1/64, and iteratively refine the parameter direction that reduces the approximation error the most.

The max tessellation factor on a modern GPU is usually 64, therefore a surface will not be rendered correctly if the factor needs to be higher than that. A workaround for this is to subdivide the patch until it's factors falls into the possible range.

3 Implementation

3.1 Source Code

The source code for the project can be found at [30] and [31]. For a guide on how to set up and run the project, see Appendix B.

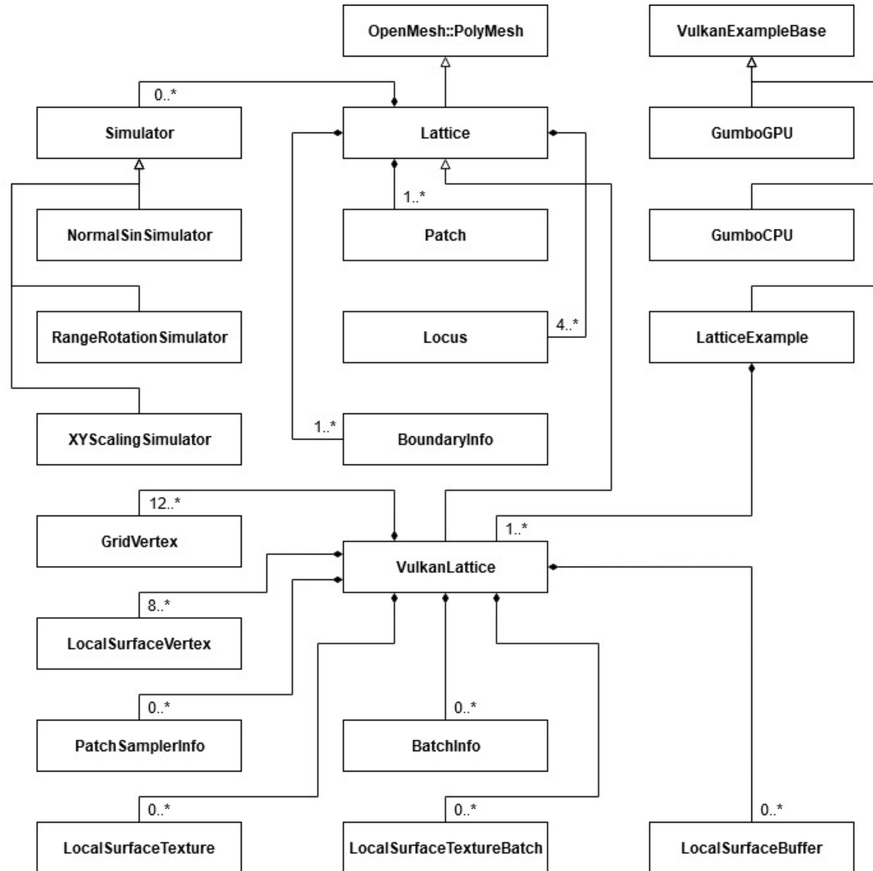


Figure 10: A diagram showing the relationship between the user defined classes and structs and what they inherit from in the implementation.

Figure 10 shows an overview of the user defined classes and structs in the implementation, and also which non user-defined classes they inherit from. `Lattice` provides the basic geometry of the lattice grid and setting up the data like local surfaces, loci and patches. `Lattice` uses OpenMesh to store and handle the vertices, edges and faces. `VulkanLattice` provides an implementation of `Lattice` that can be rendered with a Vulkan renderer. `LatticeExample` inherits from `VulkanExampleBase` to provide a basic Vulkan renderer that renders the `VulkanLattice`s. This section will look closer at the implementation and results of the individual parts of this diagram.

3.2 Base Vulkan Framework

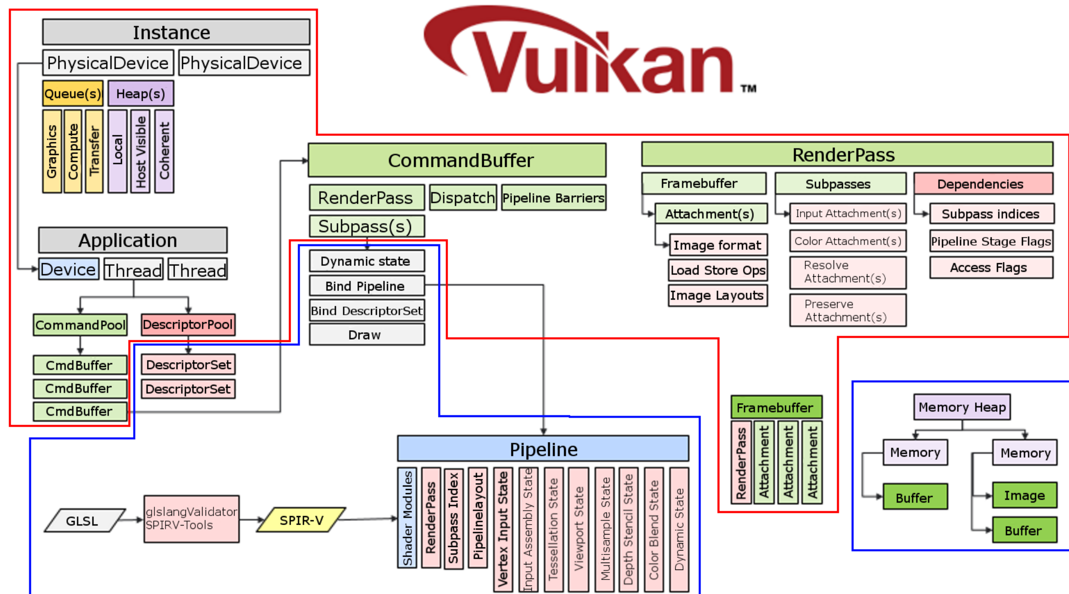


Figure 11: Overview of the different steps that has to be done in order to render with Vulkan. The parts marked with red are covered by `VulkanExampleBase` and the parts marked with blue by `VulkanLattice`. Source: [32].

To set up a Vulkan rendering application a lot of code is needed to set up the Vulkan instance/device and prepare for the actual rendering code. To ease the development and be able to focus on the actual problem, a base framework (hereafter referred to as the example framework) that takes care of all the Vulkan setup was used. The example framework used is the one used as a base for the examples found in [33].

Figure 11 shows an overview of what has to be done in order to render with Vulkan. The parts marked in red are the ones that are taken care of by the example framework and the parts in blue are taken care of by the `VulkanLattice` implementation. Additionally the example framework also takes care of window creation and sets up an instance of ImGui that is used to display information about the lattice and control parts of the rendering.

To use the example framework the class have to inherit from `VulkanExampleBase` and override a few functions. Most importantly the `prepare` function, which is where the resources for the example are set up. The `render` function that should submit a command buffer to the graphics queue to be executed. The `getEnabledFeatures` function to set up the device features needed to run the example. And lastly `OnUpdateUIOverlay` which gets called to add controls to the GUI. The three notable examples created with the framework are the `GumboGPU` and `GumboCPU` examples that render the Bézier patchwork, and the `LatticeExample` that renders lattices. The example framework also contains a lot of helpful classes/functions that helps with creating buffers, images, etc.

Two modifications for better performance have been made to the example framework. The example framework was using `vkWaitDeviceIdle` for synchronization in the rendering loop and was therefore only rendering one frame at a time on the GPU. It was changed to use semaphores and fences (Based on code from[34]) to allow more frames to be rendered at the same time. The other modification is preventing ImGui from updating every frame. The process of updating the GUI requires some work, so limiting it to updating only once per 50 ms gives a good performance gain, while still keeping the GUI responsive.

3.3 Bézier Patchwork

During the preliminary study, an example was created to test out Vulkan and the TES. The example used the code from [35] and modified it to work with the example framework from Section 3.2. To be able to get an idea of the differences between evaluating the surface using the TES and evaluating it on the CPU two different examples were implemented. The `GumboGPU` example uses the TES to evaluate the tensor-product bi-cubic Bézier patches while the `GumboCPU` example does the evaluation on the CPU.

The code from [35] is implemented in OpenGL, and therefore a couple of changes had to be made to make it work in Vulkan. First of all the shaders had to be compiled to SPIR-V, to get it to compile some additional changes had to be made to the shaders. For example, uniform buffers in Vulkan have to be in their own buffer, and is not in a global scope like in OpenGL. The tasks that the examples have to perform in order to render are first to set up the vertex and uniform buffers and store them on the GPU. The next task is to create a rendering pipeline and load in the shaders. And the last is to create a command buffer that can be submitted in the render loop to draw the model.

For the GPU example, the control points are read and stored in the vertex buffer. The vertex buffer is then used to render using 16 vertices per TCS invocation. All the work is then done in the TES to evaluate the patch. For the CPU example, the patches are evaluated before the rendering begins and then uploaded to a vertex buffer. The CPU example also allows multi-threading in the evaluation by using OpenMP and the `#pragma omp parallel for` directive. The CPU example is implemented using triangle strips, to try and be as similar to the GPU example as possible. The TES uses a topology that is similar to triangle strips[17].

Both examples allows the user to change the tessellation factor. For the GPU example this is done by setting the tessellation factors inside the TCS. For the CPU example it is done by changing the number of sample points. Since the number of sample points corresponds to the number of vertices, while the tessellation factor corresponds to the number of patches, the number of sample points has to be set to 1 more than the tessellation factor to get the same result.

Additionally, two pipeline queries were set up to query pipeline statistics and timings. The examples also allows additional models, from the menu up to 100 models can be rendered, using the same buffers, but individual draw calls. It is also possible to animate the models by moving the 4 middle control points of every patch. During animation all the control points are sent to the GPU, not just the once that have changed. This shows

the animation workload for the worst case.

3.4 Lattice

The lattice implementation is split in two classes. The first class `Lattice` contains the basic geometric functionality, with functions for adding patches and setting up the loci/patches of the lattice. The second class `VulkanLattice` inherits from `Lattice` and contains the functionality necessary to render the lattice in a Vulkan framework. Finally, the `LatticeExample` class inherits from `VulkanExampleBase`, providing a basic framework where `VulkanLattice` instances can be created and rendered. This section looks at the implementation details of `Lattice`, the other classes will be looked at in subsequent sections.

3.4.1 OpenMesh

The `Lattice` class inherits from `OpenMesh::PolyMesh_ArrayKernelT<LatticeTraits>` to provide a container for the underlying geometry input by the user. The `PolyMesh` class provides a container for meshes with polygonal faces that works well with quad faces. The data is stored in an array, which works well for static meshes. But is not the best suited when deleting geometry. When resolving T-loci, geometry is deleted and added. The array kernel still works, with a workaround on the face indices, to still use the array kernel.

`LatticeTraits` is a structure containing additional details on the types used for the class, the structure can be seen in Listing 1. The vertices are colored by their valence, and the edges are colored based on their boundary status. Therefore the vertices and edges are given the color attribute to store it. The status attribute is used to make deleting geometry possible. The previous halfedge reference is stored to make querying it faster.

```
1  // Custom traits passed to the OpenMesh::PolyMesh class.
2  struct LatticeTraits {
3      typedef Vec3f Point; // Use a vector of 3 floats as the Point type
4
5      VertexAttributes(OpenMesh::Attributes::Color | OpenMesh::Attributes::
       Status);
6      EdgeAttributes(OpenMesh::Attributes::Color | OpenMesh::Attributes::
       Status);
7      HalfedgeAttributes(OpenMesh::Attributes::Status | OpenMesh::
       Attributes::PrevHalfedge);
8      FaceAttributes(OpenMesh::Attributes::Status);
9  };
10
11 // Custom properties for vertices and faces
12 namespace LatticeProperties {
13     static OpenMesh::VPropHandleT<size_t> VertexValence;
14     static OpenMesh::VPropHandleT<uint32_t> LocusIndex;
15     static OpenMesh::VPropHandleT<LocusType> Type;
16     static OpenMesh::FPropHandleT<uint32_t> FaceIndex;
```

```
17 };
```

Listing 1: Definition of the LatticeTraits structure used to pass additional type information to the PolyMesh class. And the additional properties used by the Lattice class.

OpenMesh also allows adding additional custom properties to the data types. The properties used are defined in the namespace `LatticeProperties` shown in Listing 1. To get the valence of a vertex, it has to iterate over all the neighbouring vertices. The valence is used in a couple of places, so the valence is stored on each vertex to save some computations. The vertex also stores an index into the loci vector for the locus placed on it, and the type of that locus. When deleting geometry while using an array kernel the indices will no longer be correct for all the items. Since the `Patch` stores an index to the face, all the faces are given a custom index that won't change when geometry is deleted.

3.4.2 Adding Patches

```
1  // Functions for creating patches
2  void addPatch(Vec3f topLeft, Vec3f topRight,
3                Vec3f bottomLeft, Vec3f bottomRight);
4  void addPatch(Vec2f topLeft, Vec2f topRight,
5                Vec2f bottomLeft, Vec2f bottomRight);
6  void addPatch(Vec2f topLeft, float width, float height);
7  void addPatch(Vec3f topLeft, float width, float height, float depth);
8  // Helper functions to add more involved geometry
9  void addGrid(Vec2f topLeft, float width, float height,
10              int rows, int cols);
11 void addGridRandom(Vec2f topLeft, float width, float height,
12                    int rows, int cols);
13 void addCylinder(Vec3f center, float radius, float height,
14                  int rows, int cols);
15 void addSphere(Vec3f center, float radius, int segments, int slices);
16 void addTorus(Vec3f center, float radius, float wheelRadius,
17               int segments, int slices);
```

Listing 2: The different functions defined to add patches to the lattice.

Listing 2 shows the functions implemented in `Lattice` to allow the user to input patches. The first `addPatch` function is the function that actually adds geometry into the OpenMesh container. The parameters have to be given in the specified order; top left, top right, bottom left and then bottom right. Using a consistent ordering on the patches ensures that iterating over the geometry with OpenMesh will always be consistent. The other 3 overloaded `addPatch` functions are used to make it easier to use, the functions taking 2-component vectors adds patches in the xy-plane.

The class also provides some helper functions to add more complex lattices easier. The `addGrid` function adds a grid in the xy-plane from the provided arguments. The `addGridRandom` function is very similar, except that the patches are added to the lattice in random order, instead of from top left corner to the bottom right corner. This function is used for testing purposes to make sure the functions are not working just for

that particular case. `addCylinder` adds a cylinder built up by quads, and `addSphere` adds a sphere built up by quads. However, as it is not possible to create a sphere from quads using only regular vertices, the top and bottom of the sphere are not closed. So when animating the sphere, visible cracks will be displayed at the caps. `addTorus` adds a torus built up from quad faces.

To add a face using OpenMesh the vertices have to be added first, and then you can add the face by giving it the handles of those vertices. If a vertex that already exists in the mesh is added, the handle of that vertex is instead returned. However, since vectors of floats are used, it is possible that errors can happen when vertices that should be the same or not actually the same. This is mostly a problem when generating the points using floating point arithmetic operations, like in the helper functions for adding patches.

A first naive approach iterated over all the already added vertices and checked if they were the same by checking if the difference between them was smaller than some small epsilon. The current way of doing it is storing each added vertex in a hash map, and checking if the point is in there. The hash map uses the point as a key and stores the vertex handle as its value. Thereby making look ups much faster. The hash map is deleted after the lattice is induced.

3.4.3 Induce Lattice

After the user has finished adding patches, the `induceLattice` function has to be called to finalize the lattice creation. The function performs several tasks. First the valence, locus type and color of the vertices are set up. Then, T-loci are identified and dealt with. After that the color attribute of the edges are set up. Lastly the local surface geometry, loci, patches and boundary information is created. The Locus, Patch and BoundaryInfo structs are defined as shown in Listing 3.

```
struct Locus {
    uint32_t controlPointIndex;
    uint32_t controlPointCount;
    uint32_t matrixIndex;
    Vec3f normal;
    glm::vec3 color;
    std::unordered_map<uint32_t, uint32_t> boundaryIndices;
};

struct Patch {
    glm::vec3 color;
    uint32_t faceIdx;
    std::array<size_t, 4> lociIndices;
};

// Contains the start and end values for u and v
struct BoundaryInfo {
    float us, ue, vs, ve;
}
```

Listing 3: Definition of the Locus, Patch and BoundaryInfo structs.

The first step iterates over all the vertices added to the mesh. The valence of each vertex is set up by iterating over all its neighbouring vertices. Based on the valence of the vertex the color and type is set (see Figure 22 for an example showing all the different types). The locus type is set such that a valence of 2 is a corner locus, a valence of 3 is a boundary locus and a valence of 4 is an inner locus.

After this the vertices are iterated through again, this time looking for terminal loci. The details of this is covered in Section 3.4.5. When the terminal- and T-loci have been resolved the boundary status of all the edges will be correct, and the edge color can be set up based on this information.

The last step in the `induceLattice` function is setting up the local surfaces, loci and patches. Each locus contains information on the local surface placed on it, as well as the boundary info for each adjacent patch. The patches contains an array of the indices of its four loci. This info will then be used in `VulkanLattice` to set up the vertex buffers used for rendering.

3.4.4 Local Surfaces, Loci and Patches

Each local surface will affect the adjacent patches of the locus it is located at. Each local surface is therefore created to cover all the patches it will influence. In the framework there are currently two methods for creating these local surfaces. The first method works when the lattice is not planar, e.g a torus. However, it does not work with irregular grids where the faces can have a valence that is not 4. The second method works for irregular grids.

Figure 12 shows the idea behind the two local surface control point creation methods. The old method works by iterating over a vertex's connected edges and finding the vectors marked in red. Depending on which of the face's vertices the method was called for, the control points have to be ordered and created differently to make sure the ordering of the control points is correct. Moreover, the origin of the surface is located at the locus.

The new control point creation method tries to be more general. It takes a vertex and two vectors indicating the left and down directions of the surrounding faces as input. Then it finds and orders the adjacent faces' vertices such that the ordering is top left to right and then bottom left to right, and it removes all other vertices, e.g. T-points. Then using the relevant points that can be seen in red it sets up the control points. This method works for planar grids where the grid lines are parallel.

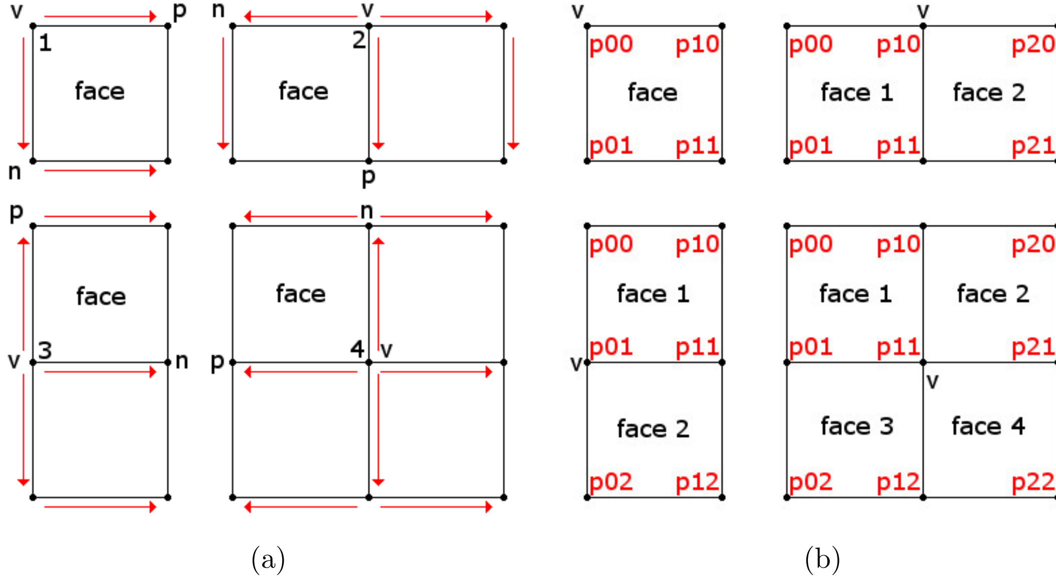


Figure 12: The old and new local surface control point creation methods visualized. (a) shows the old method. For a given face, the method is called, taking in the vertex(v), the next vertex(n), the previous vertex(p) and the vertex's number on the face. Then the red vectors shown are found by calculating the vectors along the edges. (b) shows the new method. Given a vertex(v), the method finds the adjacent faces, and orders them as seen in the illustration. The corner points of the faces are then ordered in the same way and used to set up the control points. Not in the illustration, intermediate points are created along the short edges in the boundary cases so the total number of points becomes 9.

Both methods create 4 points in the corners and 9 for the boundary and inner loci. These points are then used to create the actual control points depending on the type of local surface. The 3 different available local surfaces; the plane, Bézier degree 2 and Bézier degree 3 all need different amounts of control points. For the Plane, the excess points are discarded, for the Bézier surfaces, when there are less points than control points the missing points are generated using the other points. These control points are then added to the vector of control points, and the index of the first one is used for the locus creation.

In the `Lattice` header file there are three different pre-processor directives that can be used to control the control point creation.

```
#define USE_OLD_LOCAL_SURFACE_METHOD
#define TRANSLATE_MIDDLE_POINTS_OF_LOCAL_SURFACE_RANDOM
#define TRANSLATE_MIDDLE_POINTS_OF_CUBIC_BEZ_PRE_DEFINED
```

The first one makes sure the old local surface creation method is used, this should only be commented out if using irregular grids. The other two can be used to offset middle control points of the local surfaces. The last one translates all the control points by given amounts, this is used to get consistent lattices when testing different approaches.

All the local surfaces are created with their origin at their respective locus. The

position of the locus is used to set up their transformation matrices. These matrices are then added to a common vector and the index of the matrix is used to set up the locus. The normal member is created by crossing two edge vectors of the patch, this normal is used for translating the middle control points and by some simulators. The color member is set up at random, making it easier to distinguish between different local surfaces.

Each patch needs to know the start and end of the parametric domain for evaluating the parts of the local surface that cover that patch. This is what the BoundaryInfo struct is for. Using the points, the start and end values of the u and v parameter directions are computed. The BoundaryInfo structs can be reused by several local surfaces. For any regular uniform grid only 9 BoundaryInfos are needed. Each locus also contains a map containing the indices of the adjacent faces as keys to the associated BoundaryInfo.

Finally, when all the vertices adjacent to the face are set up as loci, the Patch can be created. The color of the patch is either a user specified color, or a random color if the random color option is used. The faceIdx member contains the index of the face and is used to index the boundaryIndices map.

3.4.5 T-Loci

Because the faces are added as quads with only 4 vertices, the faces around the T-point are not recognized as being adjacent, because they do not share an edge. Figure 13 (a) shows this, the edges on the neighbouring edges are rendered in black, meaning on the boundary. With this in mind a terminal locus can be defined as having either a valence of 5, or being on the boundary and having a valence of 4. Also, the two terminal points will be connected with an edge. So, after finding one, the opposite one can be found by circulating the vertex and finding another one with the same condition. The T-locus can then be found by finding the vertex that is connected to both of the terminal points. To fix the topology, the edge between the terminal points is deleted (Figure 13 (b)), and a new face is inserted with the 5 surrounding points (Figure 13 (c)). After fixing the topology, the affected vertices have their locus type and color fixed.

To guarantee a smooth blending over a T-loci, the local surface on the terminal- and T-locus must be sub-surfaces of a common surface[10]. To guarantee this, the affected loci are created pointing to the same set of control points and transformation matrix, but with different boundary information in their face mappings. The local surface control points are created similarly to the methods described above.

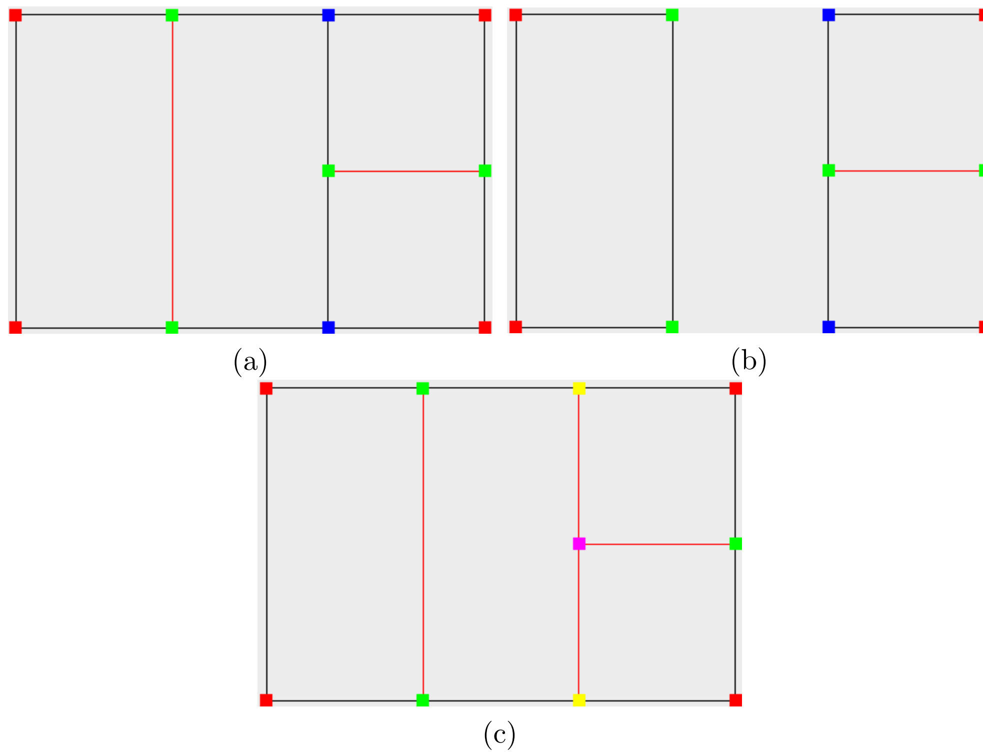


Figure 13: Handling T-loci in the underlying geometry. Initially the faces are not recognized as being adjacent (a), and the edges are flagged as being on the boundary. To fix this, the edge between the two terminal vertices is deleted, effectively deleting the face too (b). Then a new face is added with the 5 surrounding vertices (c), and the color of terminal and t-loci are fixed.

3.4.6 Data

The `Lattice` class stores all of the data created inside protected vector member variables as shown in Listing 4. This data can be accessed by other classes inheriting from it for setting up rendering resources.

```
class Lattice : public OpenMesh::PolyMesh_ArrayKernelT<LatticeTraits> {
    ...
protected:
    std::vector<Locus> m_loci;
    std::vector<Patch> m_patches;
    std::vector<glm::vec4> m_controlPoints;
    std::vector<BoundaryInfo> m_boundaries;
    std::vector<glm::mat4> m_matrices;
    ...
};
```

Listing 4: How the data is stored inside the Lattice class.

3.5 VulkanLattice

The `VulkanLattice` class is an implementation of `Lattice` to render lattices in a Vulkan framework. The class is loosely coupled with the example framework as it uses some helper functions from the `VulkanDevice`, `VulkanBuffer` and `VulkanUIOverlay` classes. With some extra work it could remove any dependencies on the example framework and could work with any Vulkan framework. The details of the Vulkan implementation will be explained in Section 3.5.1. An overview of the GUI implementation can be found in Appendix A.

The implementation includes 5 different methods for evaluating the local surfaces. Directly evaluating them in the TES explained in Section 3.5.3. And the other four methods evaluates the local surfaces before the rendering begins, and then uses different methods to sample them in the TES. The first creates an image for every local surface (Section 3.5.4), the next evaluates several local surfaces in the same image (Section 3.5.5) and the last two methods stores the data inside a buffer (Section 3.5.6), the two methods differ in the way they sample those buffers.

3.5.1 Vulkan

`VulkanLattice` does the work marked in blue in Figure 11 to make the rendering work. Listing 5 shows the functions that need to be called after the lattice is created in order to be able to render the lattice. The listing also shows the structs used as input to the rendering pipelines.

```
1  void initVulkan(
2      VkDevice* device, vks::VulkanDevice vulkanDevice,
3      VkQueue* queue, VkCommandPool* commandPool,
4      VkDescriptorPool* descriptorPool, VkRenderPass* renderPass,
5      VkAllocationCallbacks* allocator);
6  void addToCommandBufferPreRenderpass(VkCommandBuffer& commandBuffer);
7  void addToCommandBuffer(VkCommandBuffer& commandBuffer);
8  static void CheckAndSetupRequiredPhysicalDeviceFeatures(
       VkPhysicalDeviceFeatures& deviceFeatures, VkPhysicalDeviceFeatures&
       enabledFeatures);
9
10 struct LocalSurfaceVertex {
11     uint32_t controlPointIndex;
12     uint32_t controlPointCount;
13     uint32_t matrixIndex;
14     uint32_t boundaryIndex;
15     glm::vec3 color;
16     uint32_t s;
17     uint32_t t;
18     uint32_t numSamples;
19     uint32_t dataIndex;
20 };
21
22 struct GridVertex {
23     Vec3f pos;
24     Col3 col;
```

```
25 };
```

Listing 5: Some functions that needs to be called in order to render the lattice

`VulkanLattice` takes in a number of parameters. For most of the Vulkan functions the first parameter is always a `VkDevice` because it needs to know which device the function is called for. `VulkanDevice` is a class from the example framework and is used as an encapsulation for a `VkPhysicalDevice`. The class is needed to be able to allocate memory on the GPU. A `VkQueue` is needed to perform the copy commands used for moving data to device local buffers, and the `VkCommandPool` is used to allocate these copy commands from. The `VkDescriptorPool` is used to allocate descriptor sets from. The `VkRenderPass` used in the command buffer needs to be passed to the graphics pipelines when they are built. Lastly, the `VkAllocationCallbacks` is a pointer to a custom allocator used for memory allocations. For the current implementation this is just a nullptr, i.e. no custom allocations.

The first thing done in `initVulkan` is creating the vertex data used for rendering and setting up the descriptor sets. If the lattice uses a pre-evaluation method the local surfaces will also be evaluated at this stage. The grid uses the GridVertex structure to hold the data used for rendering. Every vertex of the OpenMesh container is iterated over and one GridVertex is made for each vertex, and the color is set based on the color attribute. Then every edge is iterated over to create the GridVertexes used for rendering edges.

To render local surfaces and patches the LocalSurfaceVertex struct is used. Using the vectors of loci and patches from `Lattice` these vertices are set up with indexes into the buffers containing control points, matrices and BoundaryInfos. When rendering local surfaces only one vertex is used per patch. For patches four vertices are used per patch. The LocalSurfaceVertex struct contains additional parameters which are only used by certain pre-evaluation methods.

The buffers used in the shaders need to be set up as descriptor sets to be usable. There are four buffers created regardless of the evaluation method, and all these buffers are in the same descriptor set but have different bindings. The first one is a uniform buffer used to hold the model-view and projection matrices and some parameters used for controlling the rendering. These values are often changed and the uniform buffer is allocated to be host visible and host coherent, this means that the CPU can change the contents much easier without having to use a command buffer.

Then there are the three shader storage buffers used to hold the control points for the local surfaces, the matrices of the local surfaces and the BoundaryInfos. These buffers are moved to device local buffer using staging buffers and copy commands. The reason these three are in their own buffer and not together is because of the way the shaders were made initially. At first the size of the buffers were given by specialization constants that were set in the pipeline creation stage before being compiled. Inside a buffer, only the last element can contain arrays that are dynamically sized with specialization constants.

In addition to the 4 buffers used for rendering, two more buffers are created on the GPU to hold the results from the pipeline queries and timings. These two buffers are

also created in host visible and host coherent memory so the CPU can read those values more easily.

The `initVulkan` function also handles creating the `VkPipelines`. A `VkPipeline` contains all the state that goes into the rendering pipeline. First the pipeline needs to know how the vertices are input, and the binding and attributes descriptions of the vertices have to be set. The binding and attribute descriptions are retrieved from static functions inside the vertex structs. The input assembly state also has to be specified. For the grid-points it is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, for grid-lines it is `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` and for the rest it is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`. The rasterization state determines how the polygons are filled and if face culling is used. For rendering `VK_POLYGON_MODE_LINE` is used to render surfaces in wireframe, otherwise `VK_POLYGON_MODE_FILL` is used.

The shader modules used in the pipeline have to be set up in the pipeline. The `VulkanLattice` class contains several different pipelines that use different shaders depending on the local surface type and the evaluation method. The grid-points and grid-lines pipelines are the same for all different configurations.

To render a surface, the command buffer submitted for rendering has to call the draw functions that will render that surface. The `addToCommandbuffer` functions selects the correct pipelines and calls the `vkCmdDraw*` functions depending on the current rendering parameters. Pipeline queries have to be reset before the rendering pass has started to be usable. The class therefore also has the function that adds those commands to the command buffer before the render pass has begun.

To be able to use some functionality it has to first be enabled on the device. For `VulkanLattice` these features include `.tessellationShader` and `.geometryShader` to be able to use those shader stages. `.fillModeNonSolid` is used to be able to render the surfaces in wireframe. `.pipelineStatisticsQuery` must be enabled to allow querying statistics and timings on the device. `.vertexPipelineStoresAndAtomics` and `.fragmentStoresAndAtomics` have to be set to enable shader storage buffers to be used.

The static function `CheckAndSetupRequiredPhysicalDeviceFeatures` takes two VkPhysicalDeviceFeatures structs. It checks the struct containing the GPUs features, and sets them to true in the struct containing the required features, signalling that they should be enabled. If any of the features are missing, the program will fail to run.

3.5.2 Shaders

At first, the shaders were written outside of the project and compiled from GLSL to SPIR-V using the command line compiler from glslang. But as there are a number of very similar shaders a more modular approach was later selected. Now the shaders are created inside the project by combining several pre-defined and dynamic strings to create the source for one shader. The shaders are then compiled with C++ using shaderc. Shaders are then stored inside an unordered_map so they don't need to be compiled more than once.

The vertex shader is the same for all patch-based rendering pipelines. It takes in two 4-component vectors that contain the 8 possible member values from the

LocalSurfaceVertex struct. This data is then used to set up a LocalSurfaceInfo struct that is just the same as the LocalSurfaceVertex struct, but defined inside the shader. This struct and the given color is then passed to the TCS. For rendering the grid components, the vertex shader simply performs the matrix transformations to clip space and passes the color to the fragment shader. For grid points, vertices with a color of full white will be discarded by the fragment shader. Loci are colored white if they have a valence less than two.

The TCS is responsible for setting up the four outer, and two inner tessellation levels of the patch. This can either be done by using the values inside the uniform buffer for the inner and outer levels. Or, the levels can be set dynamically, either using a view-based metric or a pixel-accurate rendering method.

The TES is different for most of the pipelines. The main task of the TES is evaluating the surface positions and normals. Some pipelines also sets the color of the surface based on calculations inside the TES. The normal display and triangle size pipelines also contain geometry shaders.

The fragment shader of most pipelines uses the normal vector to apply some simple shading so the surface will look nicer. The pixel accuracy display pipeline uses the fragment shader to perform the calculations to set the color of the fragment.

Listing 6 shows the function used to get tessellation evaluation shaders. All the functions for getting shaders return the same type - `NameSpirvPair`. The pair contains the name used to identify the shader and a reference to a vector containing the binary SPIR-V code. The `loadShader` function inside `VulkanLattice` will then use this to compile this binary code again to create `VkShaderModules`.

```
1  using NameSpirvPair = std::pair<std::string, std::vector<uint32_t>&>;
2
3  static NameSpirvPair GetTeseShader(LocalSurfaceType lsType,
4      TeseShaderType teseType, EvaluationMethod evalMethod,
5      ShaderOptions& options);
6
7  struct ShaderOptions {
8      uint32_t numControl;
9      uint32_t numLocal;
10     uint32_t numPatches;
11     uint32_t numBoundaries;
12     uint32_t numSamplesU;
13     uint32_t numSamplesV;
14     float maxError;
15     float normalLength;
16 };
17
18 enum class TeseShaderType {
19     Lattice = 0,
20     Local,
21     Pixel_Accuracy,
22     Surf_Accuracy,
23     Normals
```

```
24 };
```

Listing 6: The function used to get TES.

How the source string for the TES is created depends on several parameters. The local surface evaluation function is added depending on which local surface type is specified. If the shader uses direct evaluation it will additionally choose the evaluator function based on the local surface type. The type of TES (TeseShaderType) will also slightly alter the resulting string. Lastly a struct of options is used to set some parameters inside the shader. All the buffers that contain arrays need to have a fixed size before the shader is compiled, so that the GPU can allocate memory correctly for it.

3.5.3 Direct Evaluation

Direct evaluation directly evaluates the local surfaces inside the TES. Using the `BoundaryInfo` struct specified by the boundaryIndex the u,v coordinates are transformed so only the relevant parts of the local surface is evaluated. Then these new local u,v-coordinates are used to evaluate the surface based on the control points from the `controlPointBuffer` specified by the controlPointIndex value. When the surface position and its first order partial derivatives are evaluated they are transformed using the matrix specified with matrixIndex into the `matrixBuffer`.

The main function inside the TES is mostly the same for all the evaluation methods and local surface types. The 4 local surfaces are sampled using the evaluateLocal function, which will be different depending on the local surface type. Then those four samples are blended using the B-function to create the final position and the first order partial derivatives of the surface. The formulas for this is shown in Section 2.4.

The B-function used for blending is the B_2 polynomial function displayed in Figure 5. The surface needs to be at least C^2-continuous to be able to do the pixel-accurate rendering method.

The rendering uses one draw call to render all the patches. All the data used for evaluating the surfaces are stored in common shader storage buffers and the TES gets the indices into the arrays stored in these buffers via the vertex attributes.

3.5.4 Pre-Evaluation Using Images

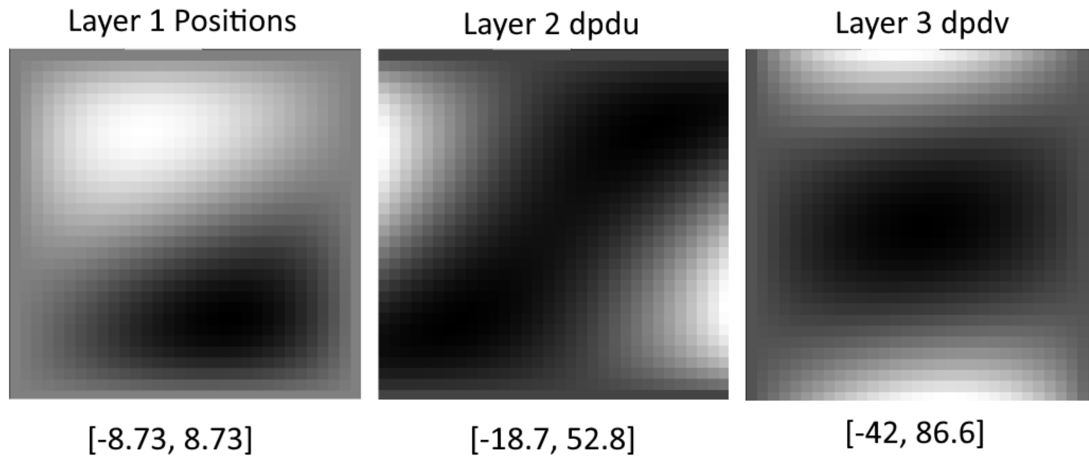


Figure 14: Z-component of a local surface texture viewed inside RenderDoc. The values are scaled by the range shown at the bottom such that the min value is black and the max value is white. The image has 32x32 samples.

The pre-evaluation method using images evaluates all the local surface and stores them inside `VkImage` objects before the rendering begins. The local surfaces are stored inside layered images where the position is stored in layer 1, the partial first order derivatives for u and v are stored in layers 2 and 3, respectively.

An additional descriptor set is used by the pipeline. This set contains 4 bindings where each one is a combined array image sampler. The evaluation of the image and the creation of the `VkImage`, `VkImageView` and the `VkSampler` is done inside the `LocalSurfaceTexture` class. The class uses the `VulkanTexture` class from the example framework to do this work. A `PatchSamplerInfo` struct holds a pointer to each of the 4 `LocalSurfaceTexture`s needed to evaluate the patch. For rendering local surfaces only one `LocalSurfaceTexture` is used.

One descriptor set is created for each patch, and the patches must therefore be rendered one at a time, switching descriptor set between each draw call. To evaluate the local surfaces inside the shader the evaluate function simply samples the texture with the local u,v-coordinates.

3.5.5 Pre-Evaluation Using Batched Images

The image pre-evaluation method runs into problems of running out of available memory objects when big lattices are rendered. The limitation on number of memory objects is 4096 for the GPU used. Therefore another method that batches several local surfaces together inside the same texture was implemented. This method also uses a different approach to store the local surfaces inside the image.

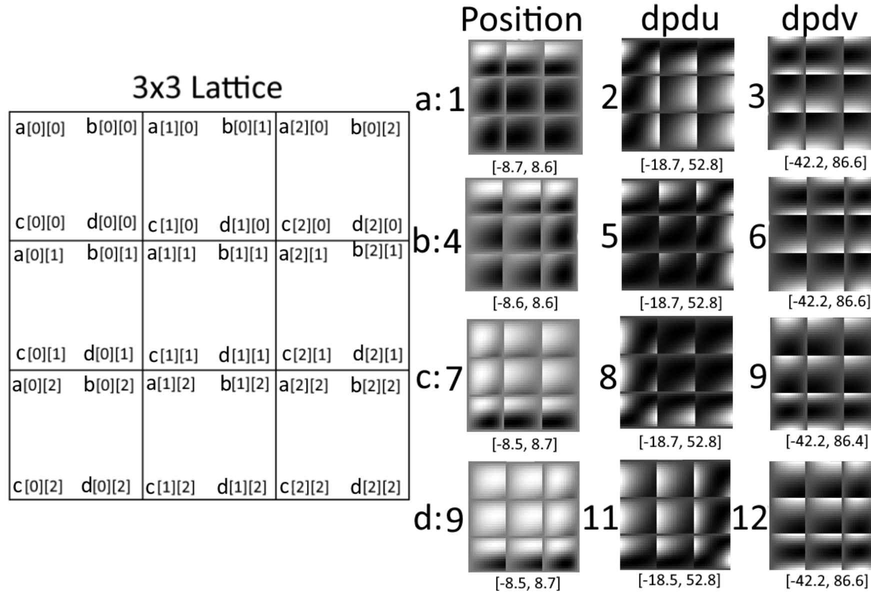


Figure 15: An illustration of how the pre-evaluated data for a 3x3 grid lattice could be stored inside the 12 layer texture. Here, a refers to the first 3 layers, b to layers 4, 5 and 6, and so on. Textures are viewed from RenderDoc. In the image only the z-component of the surface is displayed, and the value is scaled to fit the ranges shown under the textures.

The images used for this method contains 12 layers. The position and partial derivatives of the patches' top left local surface is stored in layers 1-3, top right is stored in layers 4-6, bottom left is stored in layers 7-9 and bottom right is stored in layers 10-12. Figure 15 shows an illustration of this. Using this method the local surfaces are not evaluated in their entirety, but evaluated with respect to the boundary info. How many surfaces are stored in each image can be specified, and the number of draw calls will be equal to the number of batches. The number of batches to be used has to be set by the user inside `VulkanLattice`, it is not done automatically. To render local surfaces with this method it uses the control points and direct evaluation, not the image.

The class `LocalSurfaceTextureBatch` is similar to `LocalSurfaceTexture` in how it handles creating the Vulkan resources. New local surfaces can be evaluated and added to the image before it is full or the image is created. The pipeline uses an extra descriptor set containing only one sampler per batch. In addition to allowing more local patches to be rendered, this approach also ensures that all the local surfaces of a patch will have a uniform sampling. For the former approach, the same sampling rate is used on all local surfaces. So a local surface in a corner would have a sampling size 4 times greater than a local surface on the inside of the lattice for that patch. Also because of this, a sampling size of x would be equivalent to a sample size of 2x for inner local surfaces of the other pre-evaluation methods.

This method uses some of the additional properties inside `LocalSurfaceVertex`. The s and t members are set up to contain the index of the top left corner of the given

patch, and the numSamples member contains the number of samples for both directions. This method only works when the same sampling size is used for both directions. When evaluating the local surfaces in the TES, the u and v coordinates are transformed from the [0,1] range to the range of the image where the data for that patch is stored, and these coordinates are used to sample the layers of the image. The different local surfaces and their derivatives are all stored in the same place, so the same coordinates are used for all the layers.

3.5.6 Pre-Evaluation Using Buffers

When using pre-evaluation using buffers the local surfaces are evaluated and stored inside a buffer containing one array of 4-component vectors. This buffer is inside its own descriptor set. The data is layed out starting from the top left corner of the local surface and layed out row by row. For each local surface the position and the first partial derivatives in u- and v-directions are stored after each other in that order. The dataIndex field of the LocalSurfaceVertex struct contains the index of the first point of the surface. The evaluator uses a function to sample the buffer that linearly interpolates the values of the 4 surrounding sample points of any given u,v-coordinates. The `LocalSurfaceBuffer` class is used to evaluate and store the data.

An additional pre-evaluation using buffers method is implemented. This one does not use any interpolation on the buffer sampling. This makes the method faster, but limits the tessellation levels to be set such that the number of vertices inside the patch is equal to the sampling size, or smaller such that all the generated vertices overlap with some pre-evaluated surface point.

3.5.7 Normals

The normal rendering pipeline uses the same shaders as the normal surface rendering pipeline, with an added geometry shader. In the geometry shader each triangle is read in, and from the positions of the vertices a line is created from the point and along the normal.

3.5.8 Surface Accuracy Display

When evaluating the surface on pre-sampled data the blended surface is not guaranteed to produce an accurate result. To test how far off the resulting surface is from the actual surface the color of the calculated points are set depending on how big the difference is. To calculate the difference the position is calculated twice in the TES. First using the sampling method, and then using the direct evaluation method on the control points.

3.5.9 Pixel-Accuracy Display

To be rendered pixel-accurate the tessellated surface has to be different from the true surface by no more than half a pixel when projected to screen-space. To test for this, a pipeline that colors the surface based on the difference between the tessellated and

true surface in screen-space is used. From the TES the u,v coordinates are passed to the fragment shader, where they will be interpolated for the current fragment. To find the point on the true surface for that fragment the position is calculated similarly as in the TES but on the interpolated u,v coordinates.

By projecting this newly calculated point to screen-space and comparing it with the position stored in `gl_FragCoord` the difference in pixels is found, and can be used to set the color of the surface.

3.5.10 Triangle Size Display

The color surface by triangle size pipeline sets the color of each triangle depending on the longest edge of each triangle in screen-space. The pipeline uses the same shaders as the regular surface rendering pipeline, but adds in a geometry shader. The geometry shader reads in triangles and outputs them the same as they came in. The color output is generated by transforming the triangle to screen-space and finding the longest edge.

3.6 LatticeExample

The `LatticeExample` class inherits from `VulkanExampleBase` and is used to render one or more instances of `VulkanLattice`. `LatticeExample` is also extendable, where each implementation only deals with setting up the geometry. These implementations makes it easier to set up different examples by only specifying the parameters of the geometry. Some notable examples are `GridLatticeExample`, `TorusLatticeExample`, `AllMethodsGridExample`, `PixelAccTest` and `BenchmarkGrid`.

`LatticeExample` keeps all the lattices inside a vector. When building the command buffers it will then first call the function that resets the query results in `VulkanLattice` before the render pass is started. After beginning the render pass and setting the dynamic state, it will call `addToCommandBuffer` for each lattice. It will then draw the GUI and finish the creation of the command buffers.

3.7 Adaptive Level of Detail

The first adaptive LOD method for setting the tessellation levels is the view-based method presented in [1]. In the TCS, the corners of the patch are evaluated. Using these points a surrounding sphere and the center point of each edge is found. This sphere is then projected and used to set the tessellation level based on some target pixel per edge variable. The inner tessellation factors are set to be the max tessellation factor of the edges in their respective directions. Because the patches does the calculations on the same edges on the shared boundaries, the tessellation levels will be the same, resulting in a crack-free surface rendering.

For the pixel-accuracy rendering pipeline the formula shown in Section 2.9 is used. The formula requires the bounds on the second order partial derivatives and surface positions to be found. To do this, the TCS evaluates the surface at uniformly spaced points. Then for each evaluation, it will test if the absolute value of the x or z components

of the second order partial derivatives or the surface position are bigger/smaller than the already found values, and replace them if needed.

The error is calculated starting with the most dense sample sizes 1/64 in both directions. Then, while the error is smaller than the target error, the tessellation levels are refined until either both levels are 1, or the error exceeds the target error. After this, the tessellation levels are set up so that inner and outer tessellation levels are using the tessellation levels that were found for their respective directions.

Currently there is no communication between the shaders, so the boundary tessellation levels will most likely be different from each other, thereby resulting in a cracked tessellation.

3.8 Animation

Animating the `Lattice` is done by performing affine transformations on the local surfaces' matrices. This work is done by a class inheriting from the `Simulator` class and implementing the two virtual functions:

```
1 virtual void simulate(double dt, glm::mat4& matrix) = 0;
2 virtual void undoTransformation(glm::mat4& matrix) = 0;
```

Any transformations can be done on the matrices passed in in the `simulate` function, but it has to be able to undo everything done when `undoTransformation` is called.

Three different simulators are implemented. The first one `NormalSinSimulator` translates the local surface along their normal using the sin function. The speed and amplitude of the translation is random, but can be controlled by setting the min and max values for the ranges used to set up the random values. If the min and max values are the same that value will be used.

The second simulator, `RangeRotationSimulator`, rotates the local surface around a given axis, by a given number of degrees in both directions. For the implemented version in `VulkanLattice` the axis will always be the normal of the patch. The speed of rotation and the range of rotation is set up randomly, but can be controlled similarly as for the `NormalSinSimulator`.

The last simulator, `XYScalingSimulator`, scales the local surface in the xy-plane. The speed and how much scaling is done is random, but can be controlled similarly to the other simulators.

The simulators can be used by themselves or together. In the `Lattice::update` function the `simulate` function of all added simulators are called. `undoTransformation` is called when the simulator is removed, and the effects of that simulator is gone.

4 Testing Setup

4.1 Hardware

The computer specifications used for development and testing are listed in Table 1. When a number is mentioned referencing limitations of the GPU it is the limit for the GPU listed in Table 1. A detailed list of the Vulkan capabilities of a big selection of GPUs can be found at [36].

CPU	Intel Core i5-4690K CPU @ 3.50GHz
GPU	Nvidia GeForce GTX 970
GPU Driver Version	432.00
RAM	16.0 GB
OS	Windows 10
Resolution	1920 x 1080

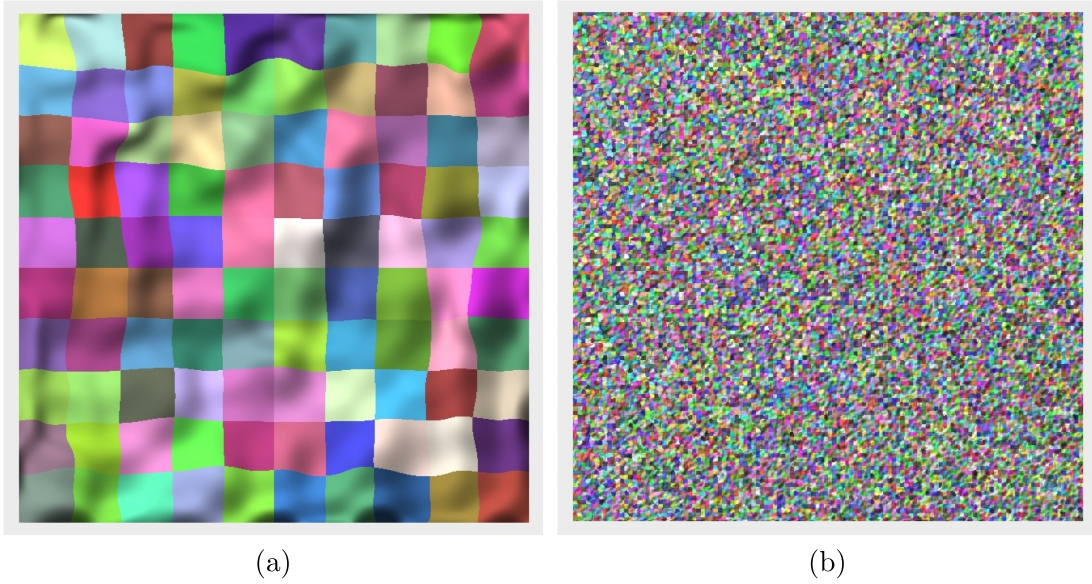
Table 1: The computer specifications used for development and testing

4.2 Benchmarking

All the benchmarking results are done using the built-in benchmarking functionality from the example framework (Section 3.2). When benchmarking is active only the rendering function is called, no GUI updates/rendering will be done. The render function is first called for a given amount of warm up time. After that the render function is called again for a given duration and the frame time is recorded for each call. All the recorded frame times are stored in a vector, and when it finishes the min, max, average and mean frame times are calculated and saved to a file. The average frame times are presented in Section 5, the files containing all the data can be found in [30].

When doing benchmarks all Vulkan validation layers are turned off. No pipeline queries or timepoints are done. The application is run in fullscreen. The application is launched from Visual Studio and runs as a 64-bit release build. A warm up of 2 seconds and a benchmarking duration of 5 seconds is used.

Figure 16 displays two of the test cases used for benchmarking. The benchmarking is done on grid lattices of sizes ranging from 10x10 to 120x120 in increments of 10. The lattices are 500x500 units in size and is viewed from the top such that the whole surface is visible.



(a) (b)

Figure 16: The surface setup used to benchmark the different surface evaluation methods. Grid lattices from 10x10 (a) and incremented by 10 all the way up to 120x120 (b). The local surfaces in the images are Bézier surfaces of degree 3, with the middle control points randomly translated. The patches are colored at random.

4.3 Direct Evaluation

Two extra benchmarks on direct evaluation are done to get some understanding of how the rendering would be different if instead the rendering uses multiple draw calls and passes the data to the shaders in smaller buffers specific for each patch. The first test uses the same vertex buffer but renders the patches in separate draw calls one at a time. To simulate the GPU having to switch descriptor sets, the descriptor set is rebound before each draw call. The second test looks at the impact of having more data stored in the buffers used by the shaders by adding extra data to the end of the control point buffer.

4.4 Surface Accuracy

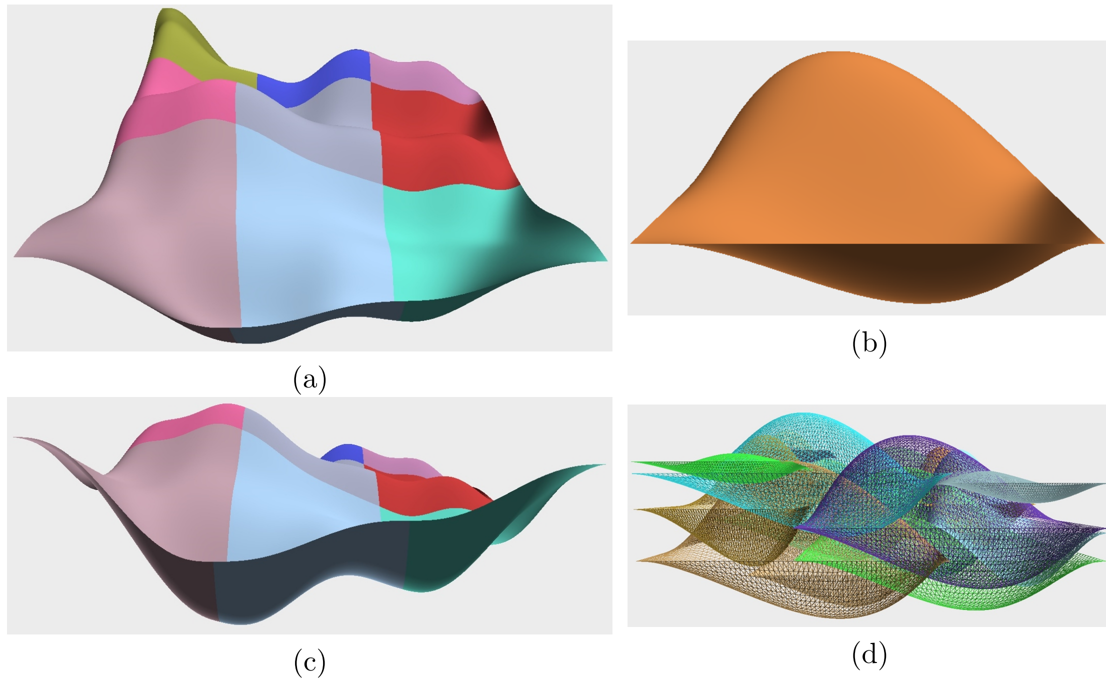
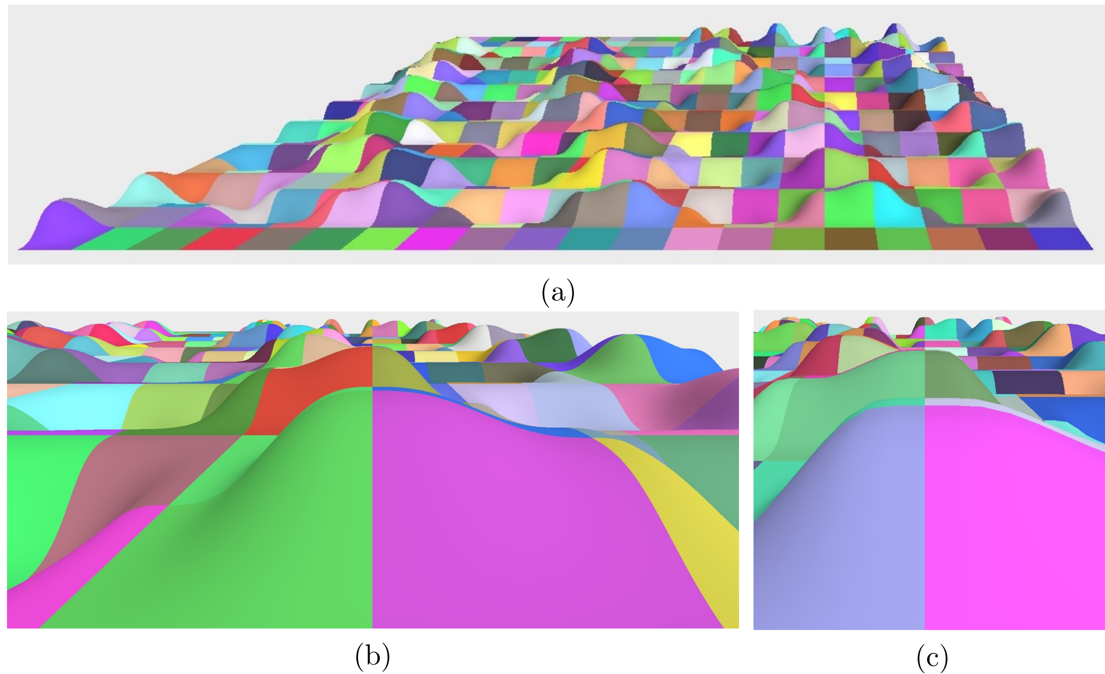


Figure 17: The setup used for testing surface accuracy. The lattice is a 3 by 3 grid lattice with a width and height of 200. The local surfaces are translated along the z-axis by at most +/- 50. (a) and (c) shows the final surface, (b) shows one of the local surfaces and (c) shows all the local surfaces. The local surfaces are Bézier surfaces of degree 3. All the local surfaces are using the same offsets on the control points.

Figure 17 shows the test setup used to test the surface accuracy of the pre-evaluation methods. The setup is a 3x3 grid lattice with a width and height of 200 units. The local surfaces are Bézier surfaces of degree 3. The 4 middle control points of the local surfaces are translated 225, 150, -75 and -200 units along the patch's normal vector.

The color of the surface when using surface-accuracy display is dependent on some variable maxError that can be specified by the user. The color will then be grey if the difference is less than maxError*0.1, green if less than maxError*0.2, blue if less than maxError*0.5, yellow if less than maxError and red if it is bigger than maxError. For the tests maxError is set to 1.

4.5 Pixel-Accuracy



(a)

(b) (c)

Figure 18: The setup used for testing the pixel-accuracy of the different tessellation setting methods. (a) shows the full surface. (b) shows the part that the camera sees during the tests. The results then get cropped to what is displayed in (c).

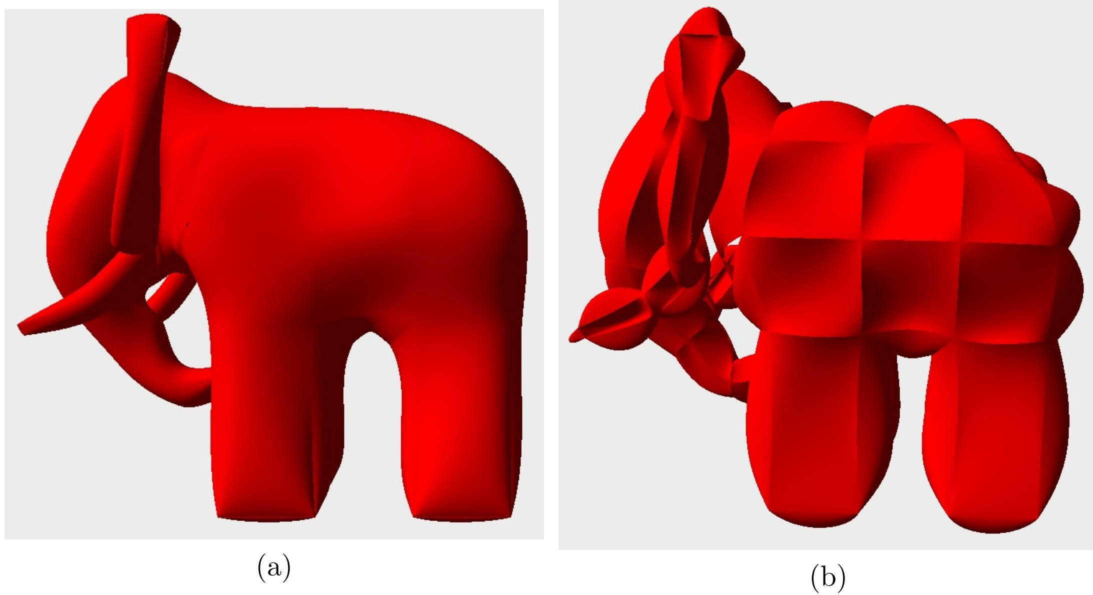
Figure 18 shows the test setup used for testing pixel accuracy and triangle sizes of the different tessellation setting methods. The test is run with the `PixAccTest` example. The example is using direct evaluation with flat bi-cubic Bézier local surfaces. Local surfaces are translated by some pre determined amount. The lattice has patches close to the camera that needs high tessellation levels to be rendered accurately, and patches far away that don't need that high tessellation levels.

The color of the surface when using the pixel-accuracy display is set depending on the difference between the tessellated surface and the true surface in screen space. If the difference is smaller than 0.1 pixels it is colored grey. Otherwise if is less than 0.5 it is green, less than 2 is blue, less than 5 is yellow and otherwise it will be red. If a fragment is colored grey or green, then that part of the surface will be considered pixel-accurate.

For the triangle size display the surface is colored by the longest edge of the triangle in pixels. The triangle is colored grey if the length is less than 1 pixel, else if the length is less than 5 pixels it is green, blue if less than 10, yellow if less than 20 and otherwise it will be colored red.

5 Results

5.1 Bézier Patchwork



(a) (b)

Figure 19: Bézier patchwork rendered with inner and outer tessellation levels set to 32. The model is made up of 128 bi-cubic Bézier patches. (a) shows the model and (b) shows the model animated by moving the middle 4 control points along the patch normal. Both images are done using the GPU implementation.

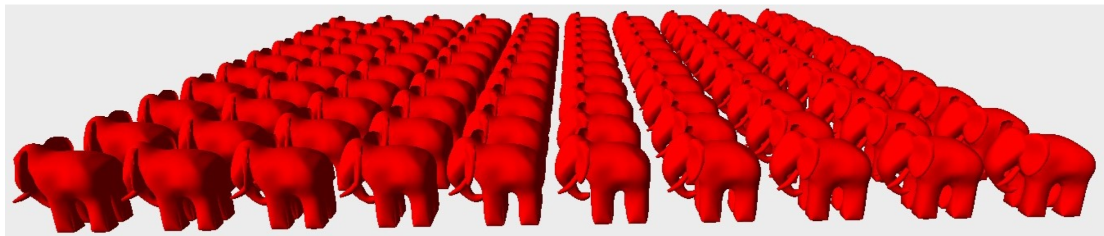


Figure 20: 100 models rendered using inner and outer tessellation levels of 32. The models use the same vertex buffers but individual draw calls.

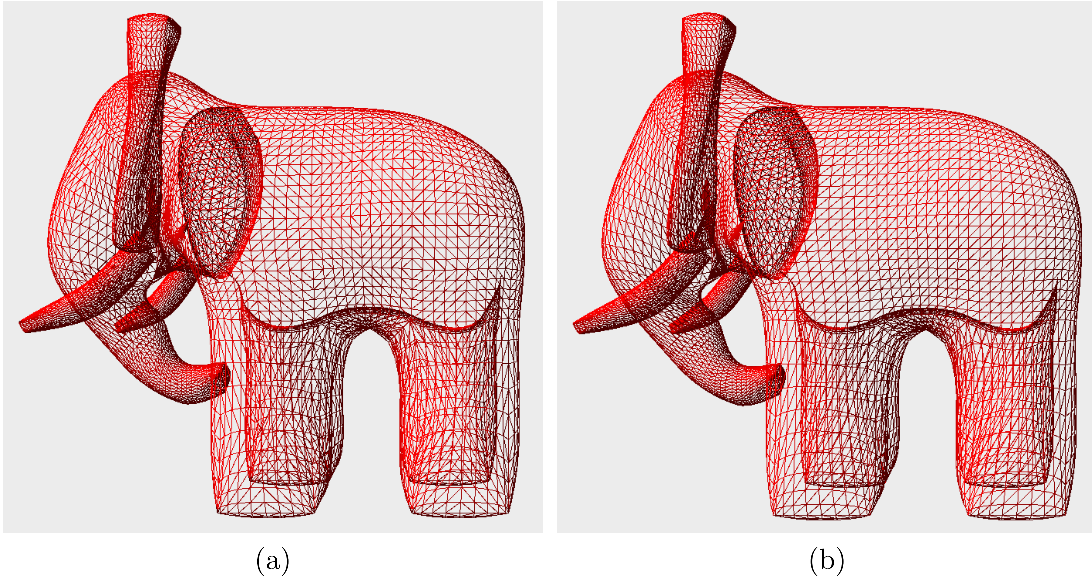


Figure 21: (a) shows the GPU implementation rendering a model in wireframe with tessellation levels set to 10. (b) shows the CPU implementation rendering a model in wireframe with sample sizes set to 11.

	CPU 1 model	GPU 1 model	CPU 100 models
Input Vertices	1064960	2048	204800
Input Primitives	10485776	128	12800
VS Invocations	1135967	2048	204800
TCS patches	0	2048	204800
TES invocations	0	823168	82316800
Clipping Invocations	1048576	1048576	104857600

Table 2: Pipeline statistics from the Bézier patchwork implementations.

Tess/Samples	Models	GPU evaluation	CPU evaluation
10/11	1	0.2124 ms	0.2127 ms
	50	0.5668 ms	0.5405 ms
	100	0.9116 ms	0.9130 ms
32/33	1	0.2602 ms	0.2689 ms
	50	2.6352 ms	3.4553 ms
	100	5.0172 ms	7.3869 ms
64/65	1	0.4415 ms	0.4928 ms
	50	9.4722 ms	14.1128 ms
	100	18.5384 ms	30.4151 ms

Table 3: Benchmarking results from the Bézier patchworks implementations

Tess/Samples	GPU eval	CPU eval	CPU eval w/ OpenMP
10/11	0.7699 ms	2.4144 ms	1.6839 ms
20/21	0.7797 ms	5.0748 ms	2.8792 ms
32/33	0.8065 ms	11.9187 ms	6.0120 ms
48/49	0.8642 ms	23.4966 ms	11.3249 ms
64/65	0.9354 ms	38.8120 ms	18.1432 ms

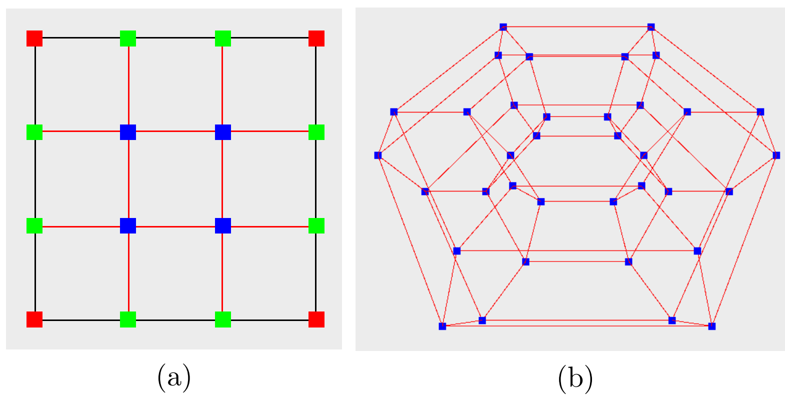
Table 4: Benchmarking results from the Bézier patchworks implementations with animation. The benchmarks are done on 1 model.

Figure 19 shows the rendering of the Bézier patchwork example, with and without simulation. The elephant model is built up from 128 bi-cubic Bézier surface patches. The number of models to be rendered can vary between 1 and 100, Figure 20 shows 100 models rendered. When rendering more than one model they all use the same vertex buffer, but are rendered in separate draw calls. Figure 21 is rendered in wireframe mode and displays the similarities between the GPU and CPU examples.

Table 2 shows the pipeline statistics from running the GPU and CPU examples with the maximum tessellation levels, and equivalent sample sizes. The number of vertex shader and tessellation shader invocations are not equal, but the resulting primitives sent to the clipping stage is the same. The figure also displays the statistics for the 100 model rendering, here the statistics are just 100 times bigger than those for the GPU example.

Table 3 shows the results gathered from benchmarking the examples at different tessellation levels/sample sizes and model counts. Table 4 shows the benchmarking results from the implementations when animating.

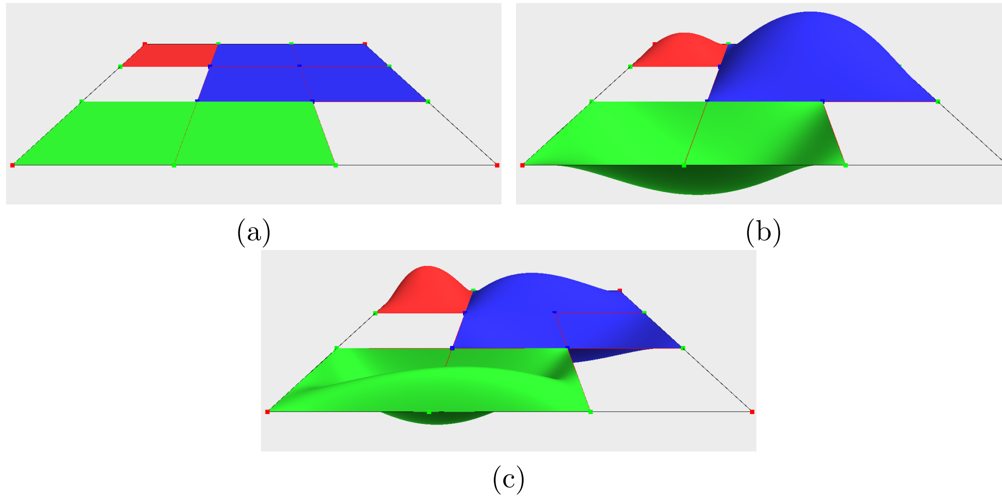
5.2 Lattice Rendering



(a) (b)

Figure 22: The grid of two lattices rendered. (a) shows the grid for a lattice with 3x3 patches. (b) shows the grid of a torus lattice. Corner loci are red, boundary loci green and inner loci blue. Boundary edges are black and inner edges are red.

Figure 22 shows a rendering of two different lattice grids. When rendering the grid the boundary edges are colored black and the inner edges are colored red. Corner loci are colored red, boundary loci green, inner loci blue, T-loci purple, and terminal loci yellow.



(a) (b) (c)

Figure 23: Some local surfaces of a lattice with 3x3 patches rendered. The three available local surface types Plane (a) and tensor product Bézier surface of degree 2 (b) and 3 (c). Only a corner, boundary and inner local surface is displayed, the rest are moved out of the way.

Figure 23 shows rendering of the 3 different local surface types in a 3x3 grid lattice. Only 3 local surfaces are shown, the rest are moved out of the way. The red surface is created for a corner loci, the green for a boundary loci and the blue for an inner loci. the red one covers only 1 patch, the green covers 2 patches and the blue covers 4 patches.

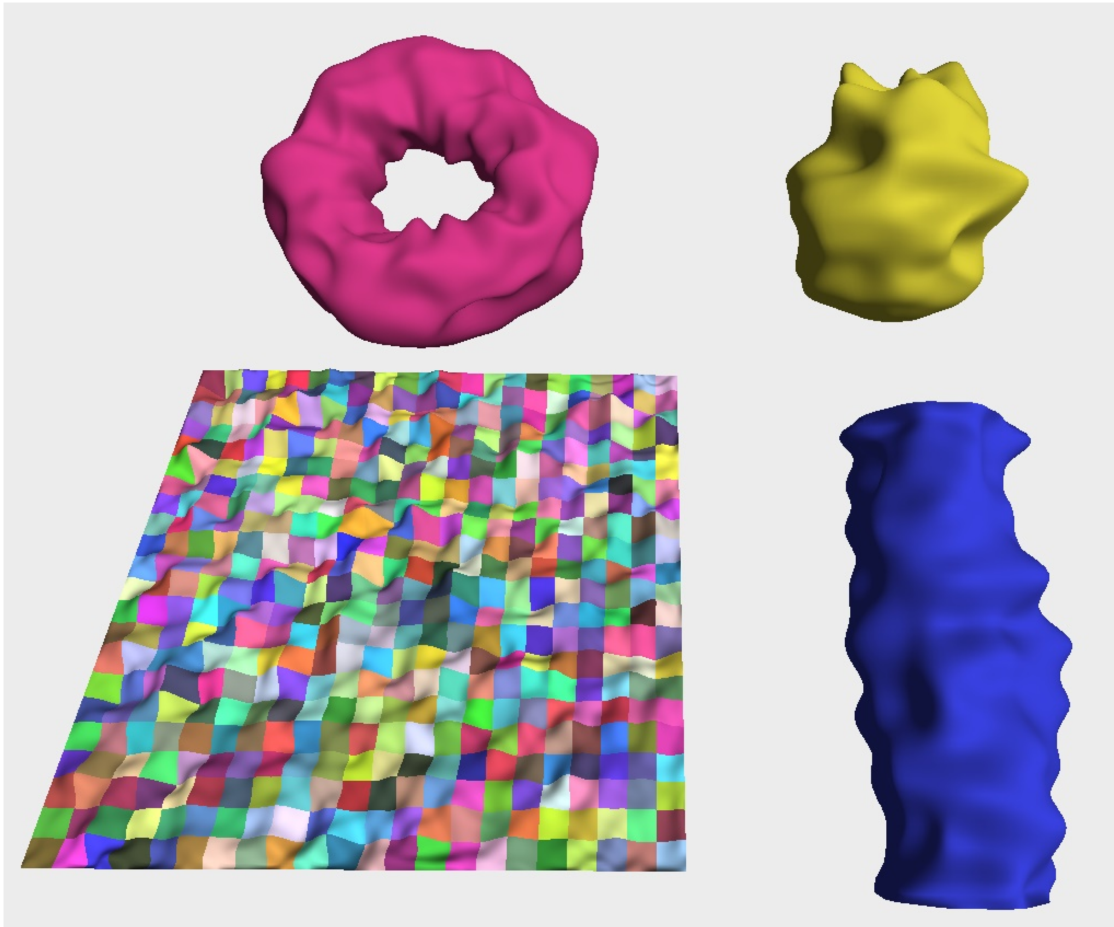


Figure 24: Displays the rendering of four different lattices. A torus, sphere, grid and cylinder. The grid is rendered with random colors for each patch.

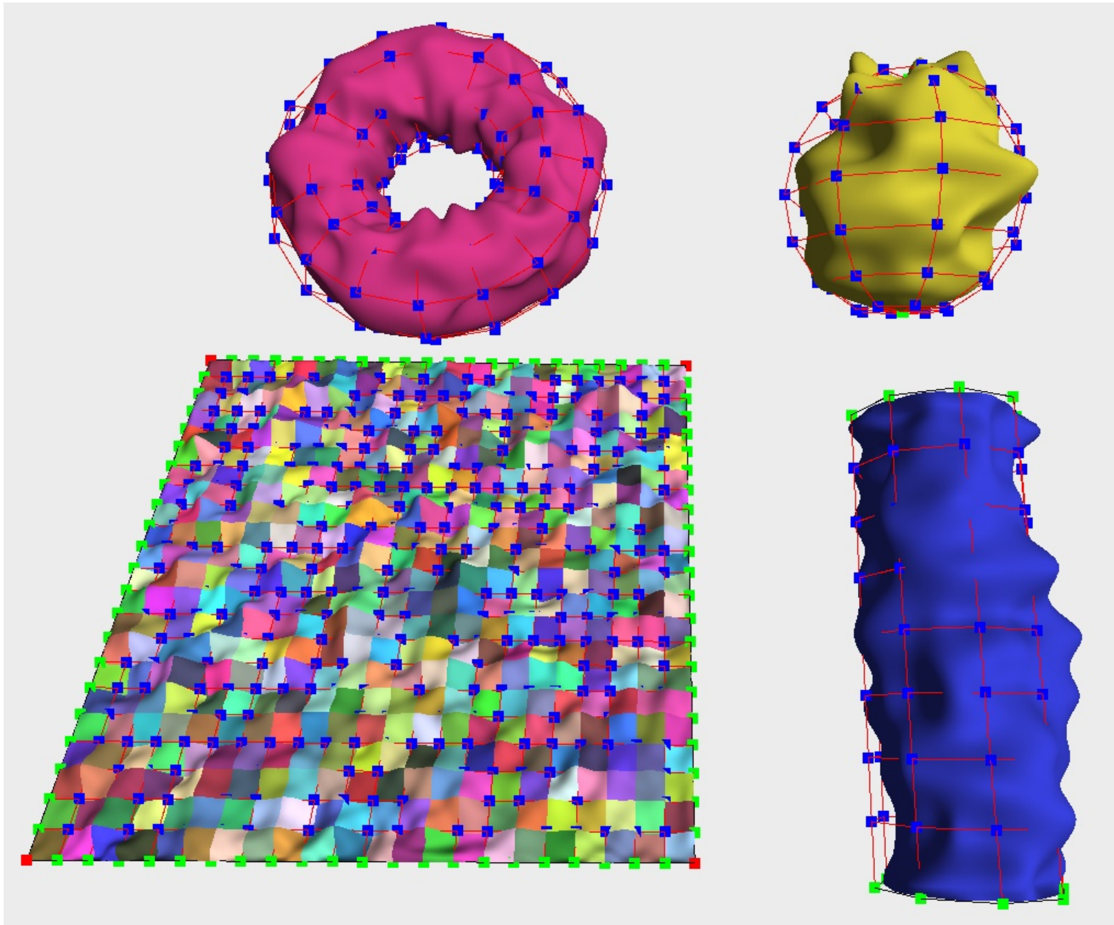


Figure 25: Displays the rendering of four different lattices and their grids. A torus, sphere, grid and cylinder. The grid is rendered with random colors for each patch.

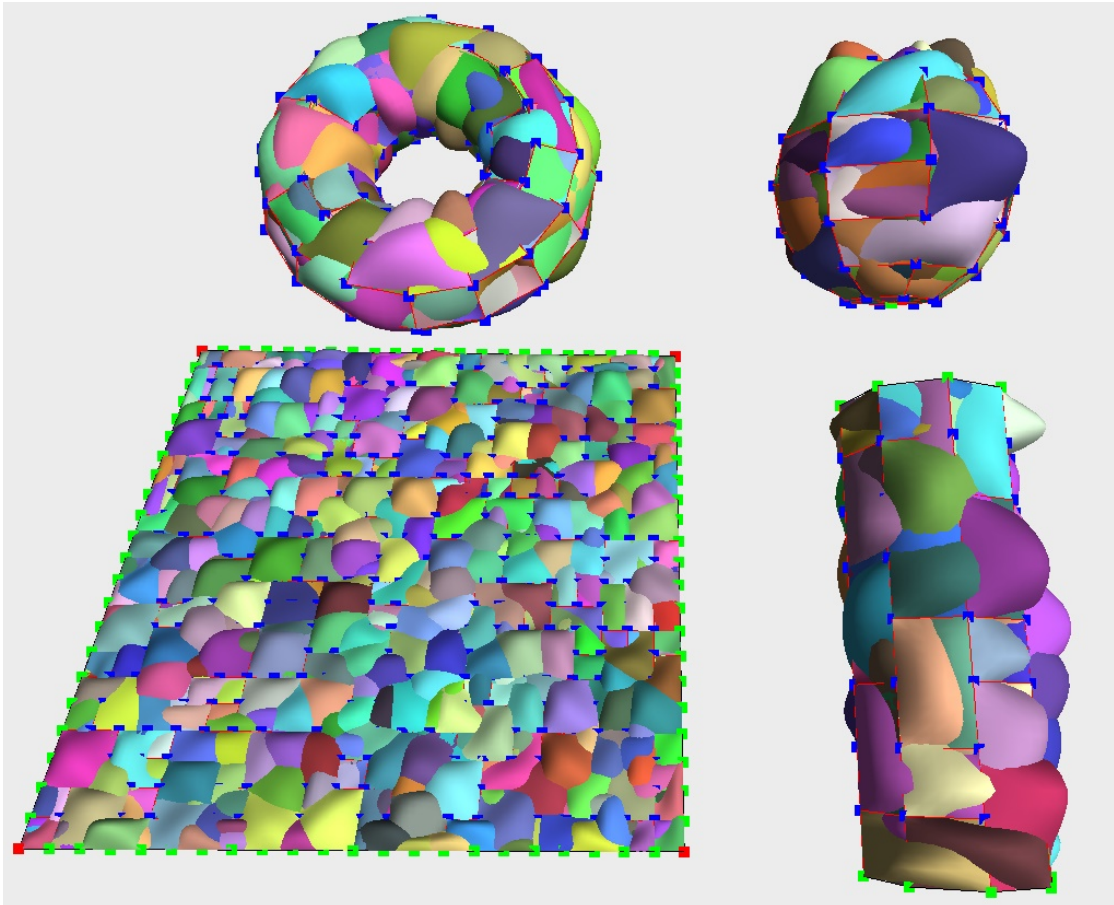


Figure 26: Displays the rendering of the local surfaces of four different lattices and their grids. The torus, sphere, grid and cylinder. The grid is rendered with random colors for each patch.

Figure 24 shows the rendering of four different lattices. A torus, a sphere, a grid and a cylinder. The grid is rendered using random color for each patch. Figure 25 shows the same rendering but with the lattice grids displayed. The local surfaces for each lattice is shown rendered in Figure 26.

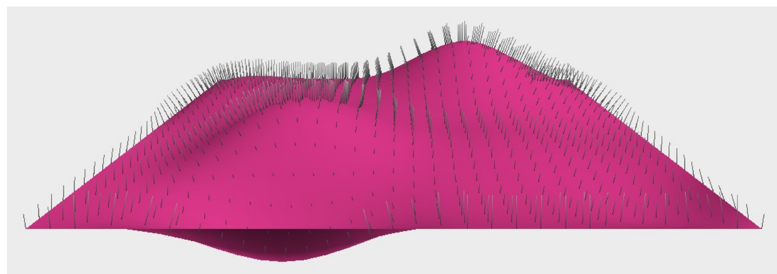


Figure 27: The normals of a lattice rendered.

Figure 27 shows the normals of a 3x3 grid lattice rendered. The normals are displayed at each vertex, so changing the tessellation levels will change the number of normals displayed.

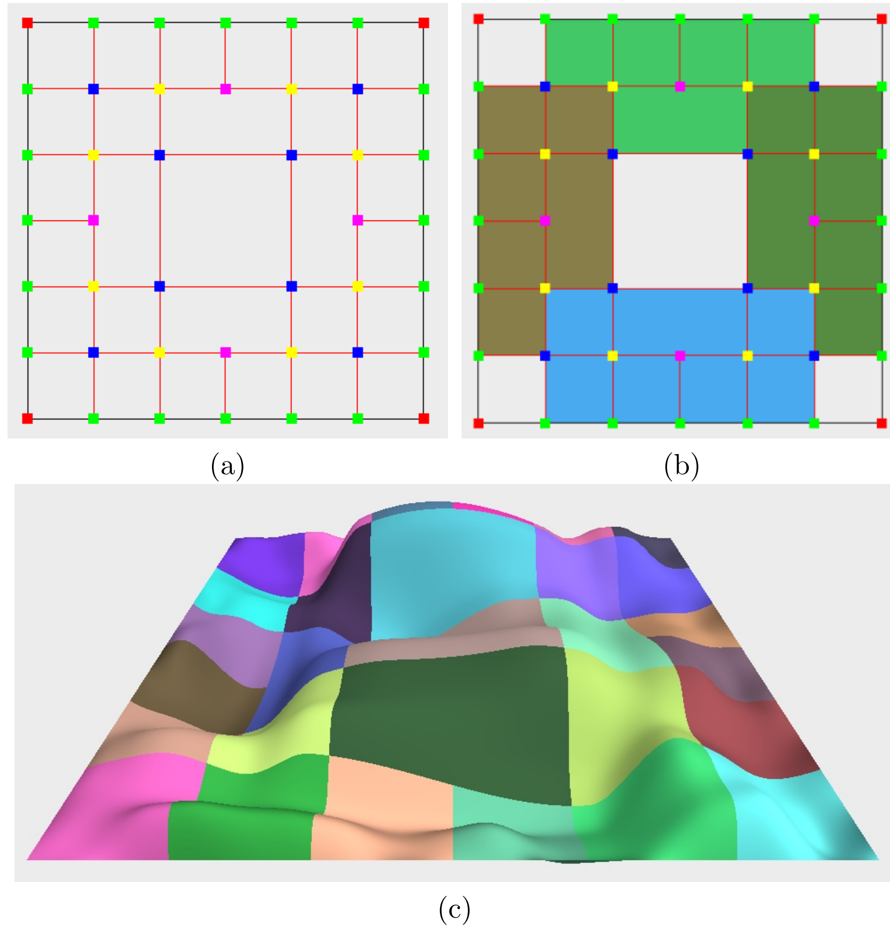


Figure 28: Rendering of an irregular grid lattice. The grid is rendered in (a), here T-points are colored purple and terminal points are colored yellow. (b) shows the rendering of the four local surfaces covering the neighbourhood of the irregular grid loci. (c) shows the final rendering of the surface, note that (b) and (c) are not renderings of the same lattice.

Figure 28 shows the rendering of an irregular grid lattice. The lattice contains 4 T-loci on different edges. The local surfaces for the T- and terminal loci are covering the entire neighbourhood of those loci. No work has been done to set up the tessellation factors for different sized patches. Each patch in the image has the same tessellation factor, and visible artifacts can be expected.

5.3 Lattice Benchmarks

Grid	Direct Eval (Plane)			Direct Eval (Bézier deg 2)		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2387	0.3006	0.6307	0.2528	0.3387	0.8087
20x20	0.3054	0.6387	1.7932	0.3512	0.8385	2.5974
30x30	0.4536	1.1349	3.7106	0.5876	1.6493	5.5422
40x40	0.6555	1.8406	6.3757	0.9014	2.6898	9.5711
50x50	0.9340	2.7266	9.8035	1.3157	4.0395	14.8089
60x60	1.2583	3.8112	14.0079	1.7814	5.6946	21.3311
70x70	1.6001	5.0811	18.9615	2.3211	7.6543	28.9461
80x80	1.9980	6.5451	24.6569	2.9421	9.9029	37.7024
90x90	2.4539	8.2163	31.2023	3.6506	12.4626	47.7563
100x100	2.9710	10.1083	38.5578	4.4305	15.2904	58.7990
110x110	3.5377	12.1988	46.6766	5.3110	18.4535	71.1756
120x120	4.1269	14.4632	55.4540	6.2613	21.9264	84.6075

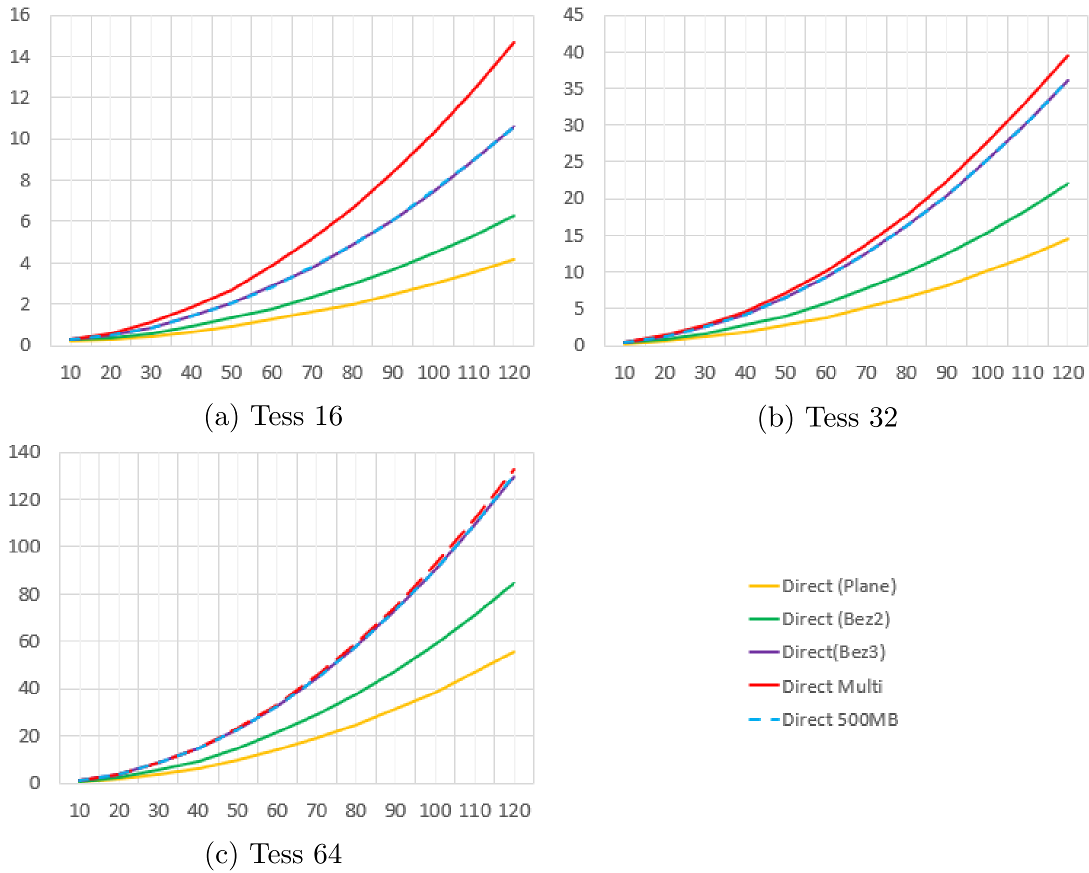
Table 5: Benchmarking results for direct evaluation with plane and Bézier degree 2 local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds.

Grid	Direct eval (Bézier deg 3)			Direct eval multi (Bézier deg 3)		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2549	0.4451	1.1637	0.3125	0.4851	1.1943
20x20	0.4846	1.2693	3.8797	0.5914	1.3672	3.9735
30x30	0.8707	2.5239	8.4333	1.1009	2.7458	8.5595
40x40	1.4149	4.2565	14.6960	1.8124	4.6173	15.0092
50x50	2.0489	6.5553	22.8054	2.7042	7.0967	23.2990
60x60	2.8583	9.2570	32.7458	3.8687	10.0888	33.4410
70x70	3.7692	12.4837	44.4311	5.1828	13.6307	45.4029
80x80	4.8477	16.2033	57.8689	6.6647	17.6936	59.1736
90x90	6.0810	20.4544	73.1884	8.3700	22.3358	74.8848
100x100	7.4361	25.1615	90.2479	10.2595	27.4880	92.3607
110x110	8.9474	30.4120	109.1668	12.3726	33.2801	111.6588
120x120	10.6013	36.1131	129.8453	14.6770	39.5456	132.8862

Table 6: Benchmarking results for direct evaluation using 1 and multiple draw calls with Bézier degree 3 local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds.

Grid	Direct eval 500MB (Bézier deg 3)		
	Tess 16	Tess 32	Tess 64
10x10	0.2528	0.4408	1.1596
20x20	0.4820	1.2711	3.8693
30x30	0.8695	2.5187	8.4221
40x40	1.4144	4.2422	14.6903
50x50	2.0638	6.5073	22.7915
60x60	2.8550	9.2468	32.7135
70x70	3.7810	12.4916	44.4185
80x80	4.8341	16.1969	57.9445
90x90	6.0777	20.4355	73.1656
100x100	7.4407	25.1937	90.2616
110x110	8.9495	30.4282	109.1534
120x120	10.5962	36.1674	129.8879

Table 7: Benchmarking results for direct evaluation with 500MB extra data in the control point buffer, with Bézier degree 3 local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds.



(a) Tess 16

(b) Tess 32

(c) Tess 64

Figure 29: Plots of the benchmarking results for direct evaluation using different local surface types. The x-axis is the grid size, the y-axis is the frame time in milliseconds. The tessellation levels for the patches are 16 in (a), 32 in (b) and 64 in (c). The red lines shows the benchmarks from the direct evaluation using multiple draw calls test. And the blue line has 500MB extra data inside the control point buffer. Both these methods use Bézier local surfaces of degree 3.

The results from benchmarking direct evaluation is shown in Table 5, 6 and 7. The results show the average frame time of the tests in milliseconds, for varying grid sizes and tessellation levels. The results include the three different local surfaces, and also the multiple draw calls and 500MB data buffer tests, which both use Bézier local surfaces of degree 3. In Figure 29 the benchmarks are visualized as line diagrams. The x-axis shows the grid size with the frame time in milliseconds on the y-axis.

Grid	Pre-Eval Img (Plane) (16s)			Pre-Eval Img (Bézier deg 3) (16s)		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2907	0.3361	0.6965	0.2949	0.3368	0.6997
20x20	0.4214	0.7691	2.2342	0.4206	0.7724	2.2412
30x30	0.7226	1.5309	4.6891	0.7282	1.5322	4.6965
40x40	1.1126	2.5059	8.0787	1.1149	2.5116	8.0969
50x50	1.6675	3.7612	12.4882	1.6752	3.7539	12.4829
60x60	2.3772	5.3007	17.8708	2.3703	5.2960	17.8728

Table 8: Benchmarking results for pre-evaluation using images with plane and Bézier degree 3 local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. Both using 16x16 sample sizes for the local surfaces.

Grid	Pre-Eval Img (Samples 32x32)			Pre-Eval Img (Samples 64x64)		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.3097	0.3440	0.7031	0.4744	0.4664	0.7215
20x20	0.4888	0.7856	2.2570	0.9821	0.9934	2.2859
30x30	0.8238	1.5719	4.7366	1.7513	1.9497	4.7792
40x40	1.3400	2.5878	8.1881	2.7982	3.1796	8.2420
50x50	2.0309	3.8679	12.6144	4.1501	4.7373	12.6882
60x60	2.8944	5.4482	18.0654	5.8133	6.6352	18.1658

Table 9: Benchmarking results for pre-evaluation using images with plane local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. Using 32x32 and 64x64 sample sizes for the local surfaces.

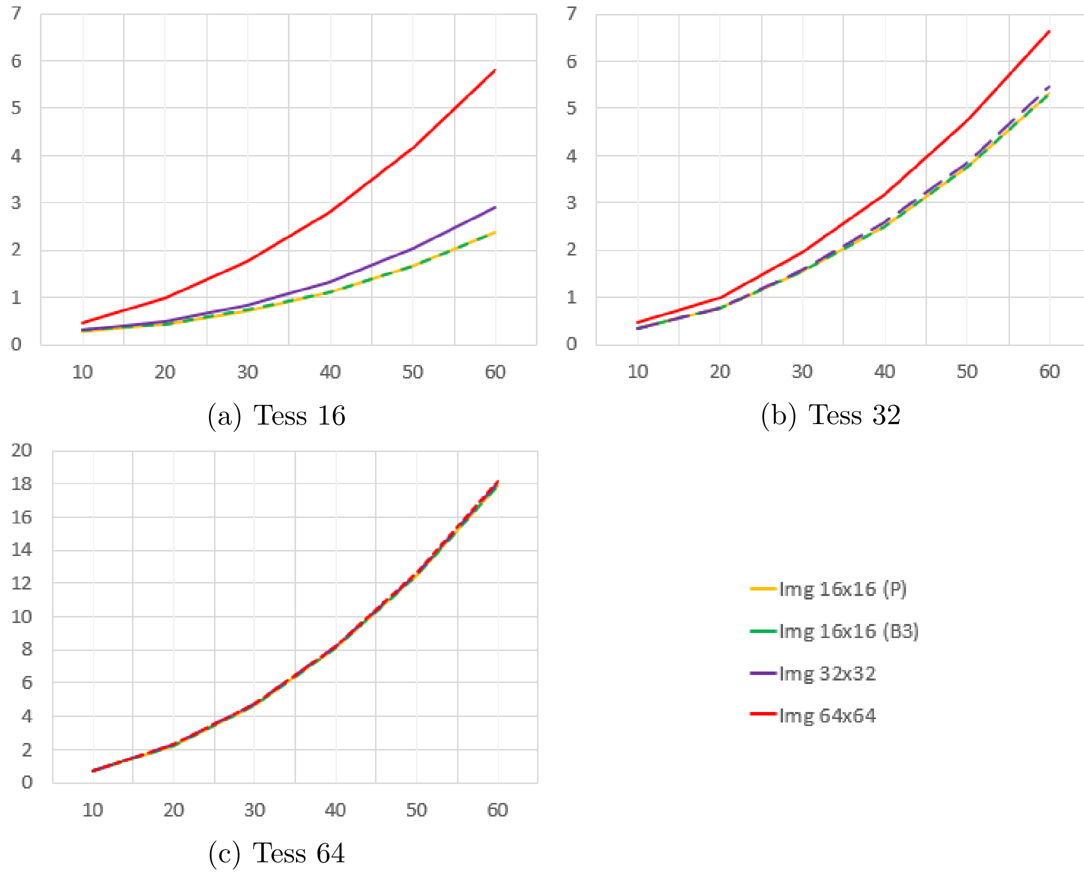


Figure 30: Plots of the benchmarking results for pre-evaluation using images. The x-axis is the grid size, the y-axis is the frame time in milliseconds. The tessellation levels for the patches are 16 in (a), 32 in (b) and 64 in (c). The tests are using different sample sizes for the images storing the pre-evaluated data. The green line is using Bézier local surfaces of degree 3, the other ones are using Plane local surfaces.

Table 8 and 9 shows the results from the benchmarking with the pre-evaluation using images method. The benchmarks include different local surface types and sample sizes, tested on grids of varying sizes and tessellation levels. Because of the limitations on the amount of memory objects with the GPU used, the benchmarks does not go any higher than a 60x60 grid. Figure 30 shows the benchmarking results plotted inside separate diagrams for each tessellation level. The x-axis shows the grid size with the frame time in milliseconds on the y-axis.

Grid	Pre-Eval Batch (Samples 8x8)			Pre-Eval Batch (Samples 16x16)		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2494	0.3049	0.6650	0.2876	0.3221	0.6716
20x20	0.3204	0.6828	2.0745	0.3962	0.6940	2.0811
30x30	0.4896	1.3195	4.3116	0.5856	1.3235	4.3340
40x40	0.7182	2.1332	7.4430	0.8723	2.1471	7.5042
50x50	1.0258	3.1770	11.4637	1.2407	3.1960	11.5640
60x60	1.3953	4.4555	16.4033	1.6583	4.4743	16.5417
70x70	1.8015	5.9577	22.1759	2.0954	5.9881	22.4034
80x80	2.2642	7.7005	28.9526	2.6080	7.7376	29.1706
90x90	2.7936	9.6658	36.6427	3.1918	9.7147	36.8753
100x100	3.3724	11.8621	45.1610	3.8395	11.9217	45.4563
110x110	4.0312	14.3071	54.6238	4.5593	14.3727	54.9455
120x120	4.7363	16.9667	65.0031	5.3514	17.0366	65.3862

Table 10: Benchmarking results for pre-evaluation using batched images with plane local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. Using 8x8 and 16x16 sample sizes for the local surfaces.

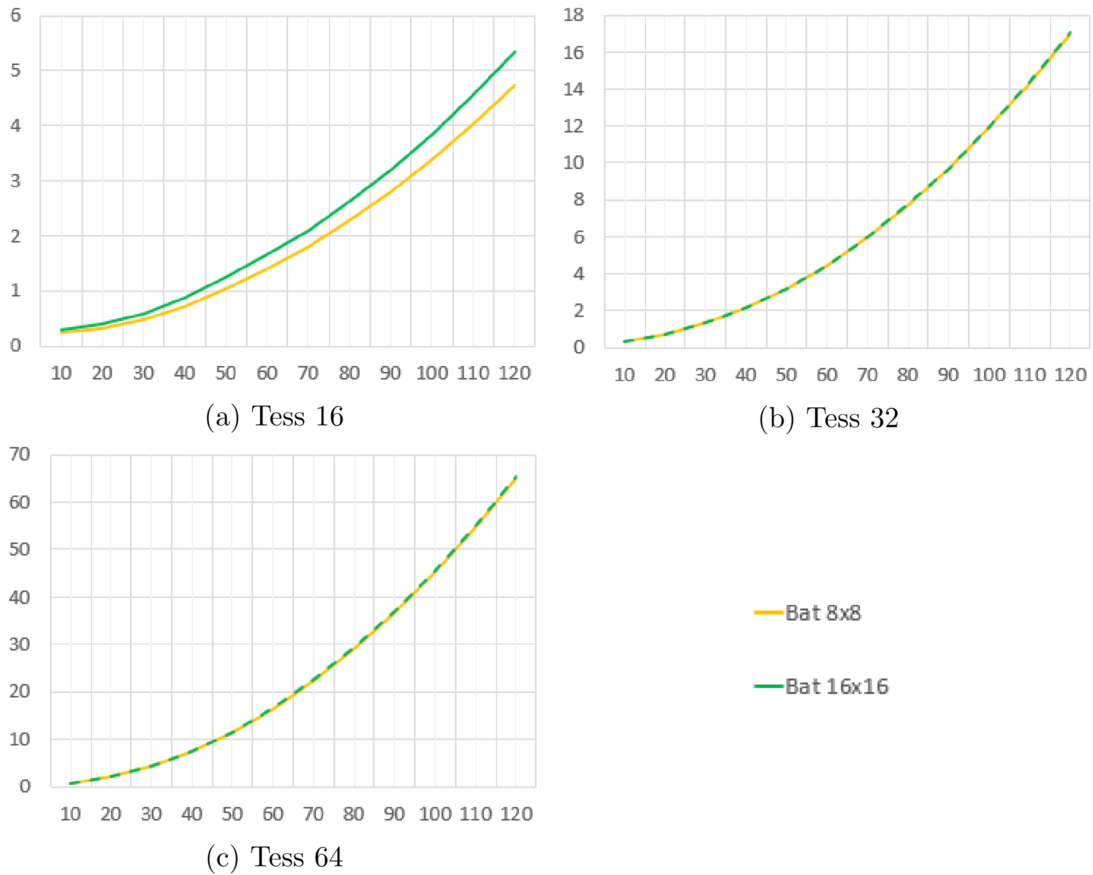


Figure 31: Plots of the benchmarking results for pre-evaluation using batched images. The x-axis is the grid size, the y-axis is the frame time in milliseconds. The tessellation levels for the patches are 16 in (a), 32 in (b) and 64 in (c). The tests are using different sample sizes for the images storing the pre-evaluated data. All tests are using Plane local surfaces.

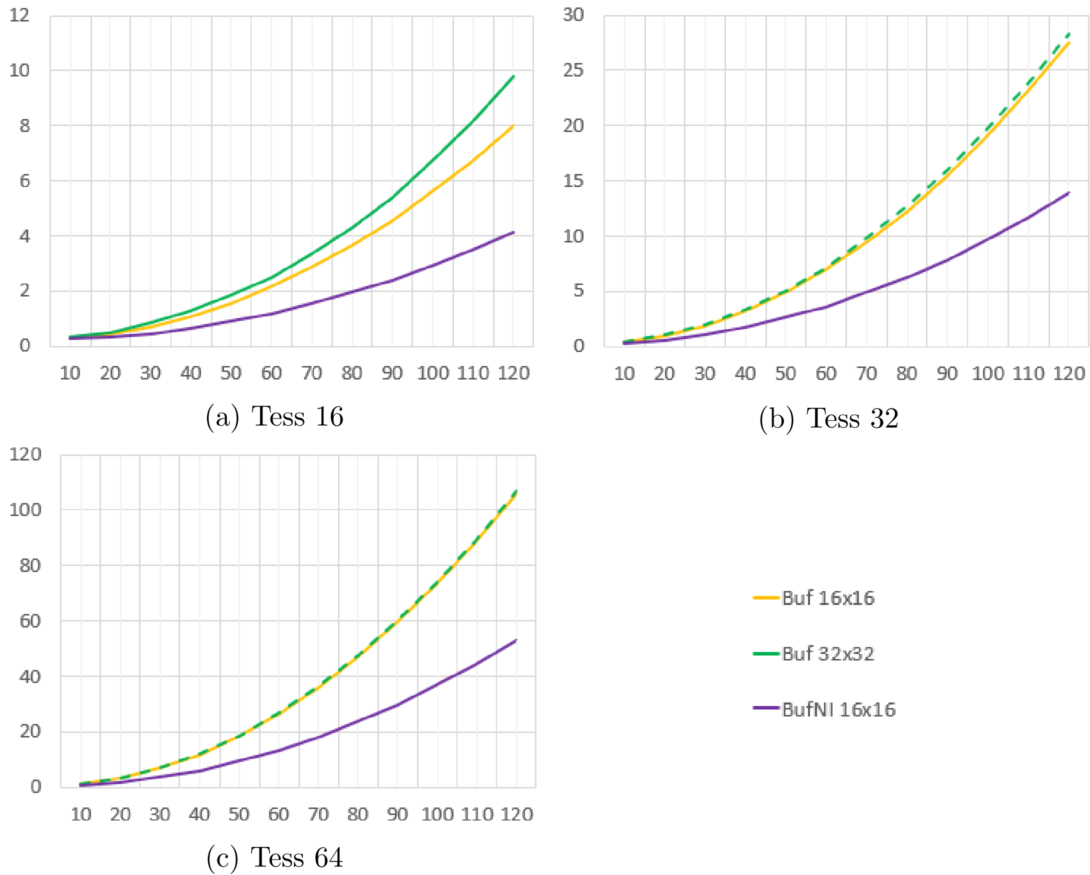
Table 10 shows the benchmarking results from rendering with the pre-evaluation using batched images method. Two different sample sizes are tested with varying grid sizes and tessellation levels. Figure 31 displays the results as line diagrams where the x-axis shows the grid size and the y-axis shows the frame time in milliseconds.

Grid	Pre-Eval Buffer (16x16 Samples)			Pre-Eval Buffer (32x32 Samples)		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2559	0.3672	0.9312	0.3190	0.3828	0.9458
20x20	0.3980	0.9760	3.1340	0.4935	1.0037	3.1670
30x30	0.6889	1.9096	6.8073	0.8281	1.9789	6.8833
40x40	1.0846	3.2120	11.9098	1.2868	3.3268	12.0521
50x50	1.5585	4.8956	18.4789	1.8417	5.0636	18.7072
60x60	2.1523	6.9579	26.5331	2.4917	7.2047	26.8617
70x70	2.8496	9.3566	36.0446	3.3267	9.7280	36.4573
80x80	3.6531	12.2062	47.0240	4.2941	12.6511	47.5509
90x90	4.5698	15.4809	59.5514	5.3903	15.9851	60.0759
100x100	5.5873	19.0759	73.5466	6.6917	19.6532	74.0948
110x110	6.7279	23.0843	88.8678	8.1627	23.7866	89.3733
120x120	7.9834	27.5083	105.7965	9.7674	28.3337	106.8378

Table 11: Benchmarking results for pre-evaluation using buffer with plane local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. Using 16x16 and 32x32 sample sizes for the local surfaces.

Grid	Pre-Eval Buffer No Interpolation		
	Tess 16	Tess 32	Tess 64
10x10	0.2484	0.2706	0.5500
20x20	0.2980	0.5735	1.7143
30x30	0.4397	1.0819	3.5414
40x40	0.6359	1.7629	6.0853
50x50	0.8909	2.6024	9.3789
60x60	1.1818	3.6305	13.3925
70x70	1.5516	4.8416	18.1403
80x80	1.9462	6.2463	23.6210
90x90	2.3998	7.8574	29.8535
100x100	2.9082	9.6453	36.7786
110x110	3.4821	11.6300	44.4902
120x120	4.1058	13.8323	52.9616

Table 12: Benchmarking results for pre-evaluation using buffer without interpolation, with plane local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. Using 16x16 sample sizes for the local surfaces.



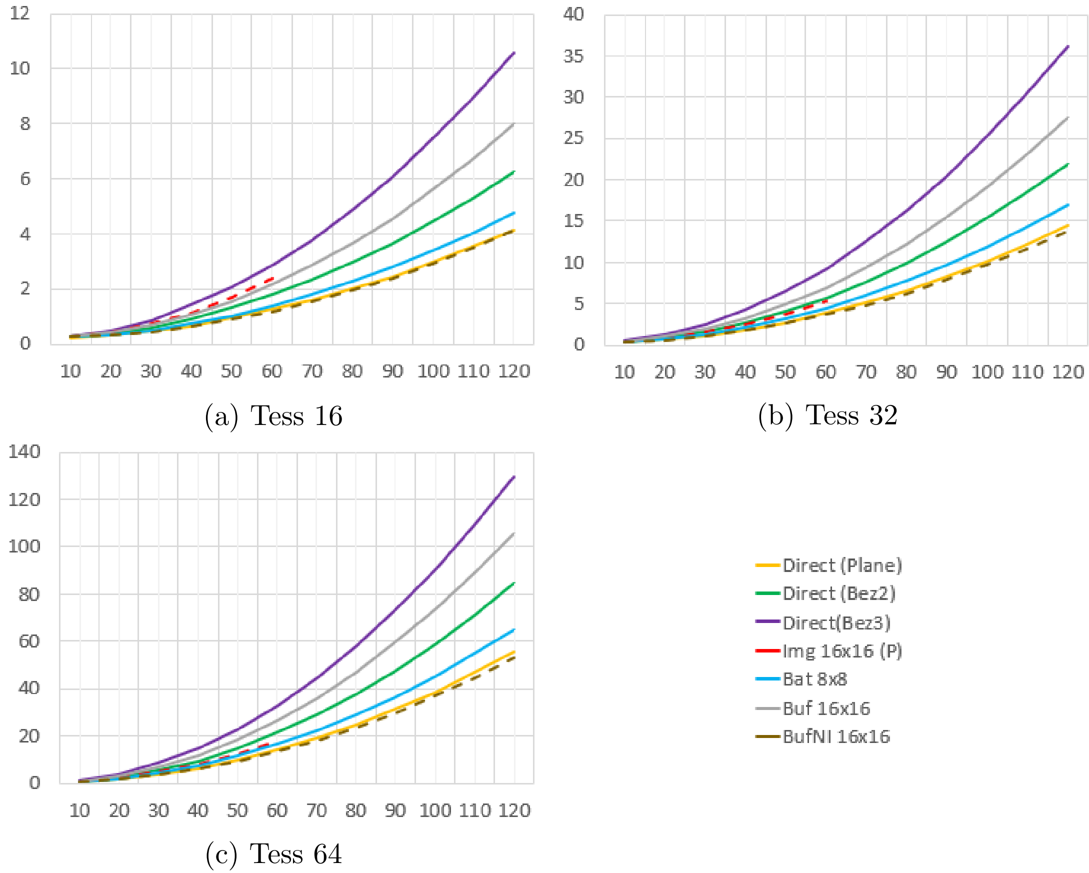
(a) Tess 16

(b) Tess 32

(c) Tess 64

Figure 32: Plots of the benchmarking results for pre-evaluation using buffer. The x-axis is the grid size, the y-axis is the frame time in milliseconds. The tessellation levels for the patches are 16 in (a), 32 in (b) and 64 in (c). The tests are using different sample sizes for the data stored inside the buffer. The purple line is not using any interpolation on the data it samples. All tests are using Plane local surfaces.

Table 11 and 12 displays the benchmarking results from the pre-evaluation using buffer method. The benchmarks are done using two different sample sizes, the benchmark on the method without interpolation during buffer sampling uses a sample size of 16x16. The results are plotted in Figure 32, where the x-axis is the grid size and the y-axis shows the frame time in milliseconds.



(a) Tess 16

(b) Tess 32

(c) Tess 64

Figure 33: A comparison of the different evaluation methods. The x-axis is the grid size, the y-axis is the frame time in milliseconds. The tessellation levels for the patches are 16 in (a), 32 in (b) and 64 in (c). All pre-evaluation methods are using Plane local surfaces, and their sample sizes are displayed after their names. The brown line is not using any interpolation when sampling the local surface data.

Figure 33 shows the benchmarking results from all the different local surfaces with direct evaluation and the pre-evaluation methods. The pre-evaluation with image method is only shown up to a grid size of 60.

5.4 Animation

A video showing animation with different lattices and simulators can be found in the supplementary material.

Grid	Animation Translation			Animation Rotation		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2611	0.4563	1.1835	0.2638	0.4589	1.1945
20x20	0.5060	1.3061	3.9313	0.5289	1.3321	3.9524
30x30	0.8900	2.5846	8.4679	0.9383	2.6499	8.5602
40x40	1.4958	4.3513	14.8041	1.5548	4.4180	14.8708
50x50	2.1928	6.6536	22.9509	2.2950	6.7528	23.0851
60x60	3.0396	9.4417	32.9214	3.2045	9.5982	33.1393
70x70	4.0174	12.7423	44.7968	4.2413	12.9858	45.0169
80x80	5.1642	16.5894	58.3770	5.4715	16.9906	58.7263
90x90	6.4909	20.9489	73.8293	6.8838	21.4094	74.3146
100x100	7.9497	25.8345	91.0878	8.4613	26.4065	92.0064
110x110	9.6632	31.3377	110.2238	10.4301	31.9690	110.9433
120x120	11.5397	37.2721	131.1168	12.3101	38.1401	131.9621

Table 13: Benchmarking results for direct evaluation Bézier degree 3 local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. The surfaces are animated by translating or rotating the local surfaces.

Grid	Animation Scale			Animation All		
	Tess 16	Tess 32	Tess 64	Tess 16	Tess 32	Tess 64
10x10	0.2606	0.4551	1.1841	0.2708	0.4657	1.1924
20x20	0.5084	1.3121	3.9281	0.5531	1.3544	3.9879
30x30	0.8956	2.5959	8.5257	0.9889	2.6880	8.5317
40x40	1.5024	4.3613	14.8124	1.6594	4.5265	14.9980
50x50	2.2024	6.6668	22.9526	2.4740	6.9665	23.3458
60x60	3.0406	9.4415	32.9620	3.4849	9.9813	33.5844
70x70	4.0298	12.7612	44.7515	4.6950	13.7537	45.7178
80x80	5.1740	16.5751	58.3613	6.2314	17.6522	59.4964
90x90	6.5092	20.9689	73.8348	7.9324	22.4856	75.4233
100x100	7.9560	25.8744	91.1190	9.8018	27.7987	93.0221
110x110	9.6479	31.2664	110.1088	12.1735	33.9257	112.7658
120x120	11.5581	37.2079	131.0677	14.5867	40.3910	134.2994

Table 14: Benchmarking results for direct evaluation Bézier degree 3 local surfaces for grid lattices of different sizes and tessellation levels. All results are displayed in milliseconds. The surfaces are animated by scaling or translating, rotating and scaling the local surfaces.

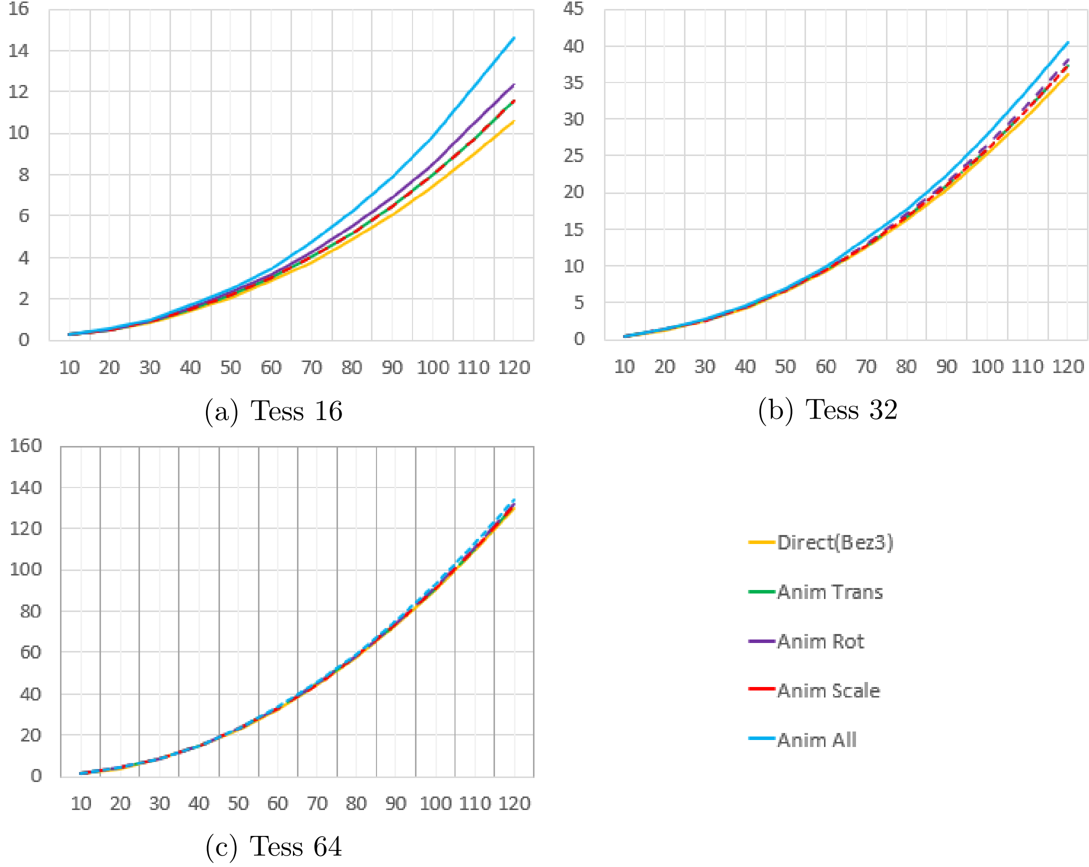


Figure 34: Benchmarks from the different simulators. All tests are using direct evaluation with Bézier local surfaces of degree 3. The orange line is not using any simulators. The x-axis is the grid size, the y-axis is the frame time in milliseconds. The tessellation levels for the patches are 16 in (a), 32 in (b) and 64 in (c).

Table 13 and 14 displays benchmarking results from using the different simulators to animate the surface. The surfaces are using direct evaluation and Bézier local surfaces of degree 3. The different simulators used are the `NormalSinSimulation` for translation, `RangeRotationSimulator` for rotation, `XYScalingSimulator` for scaling, and combining all of them for the last results. In Figure 34 the four results are plotted together with the equivalent surface without any simulation. The x-axis shows the grid size and the y-axis shows the frame time in milliseconds.

5.5 GPU Memory Usage

The amount of data in bytes that is stored on the GPU when using direct evaluation is given by:

$$\text{GPU_mem}_{\text{Dir}} = 64s + 16(l + 2e) + 44(s + 4p) + 16sc + 16b + 336$$

where s is the number of local surfaces, l is the number of loci, e is the number of edges, p is the number of patches, c is the number of control points per local surface, and b is the number of BoundaryInfos. The size of the uniform buffer and the query and timing buffers make up 336 bytes, regardless of the number of patches.

The memory usage for a regular uniform x-by-x grid like the ones used in the benchmarks can be simplified to:

$$\text{GPU_mem}_{\text{Dir}} = \hat{g}(16c + 124) + 176g + 64\hat{x}x + 480$$

where x is the number of rows/columns, $\hat{x} = x + 1$, $g = x^2$, $\hat{g} = \hat{x}^2$ and c is the number of control points per local surface. For a regular uniform grid with $x > 1$ the number of BoundaryInfos will always be 9, so that is an additional 144 bytes of fixed memory.

For the pre-evaluation methods using images and buffers the memory usage is the same. Both methods evaluates the local surfaces by a given sample size and store the data inside 4-component float vectors. Additionally, all pre-evaluation methods store the same data as the direct evaluation method. The memory usage in bytes of the image and buffer pre-evaluation methods are given by:

$$\text{GPU_mem}_{\text{Img/Buf}} = \text{GPU_mem}_{\text{Dir}} + 48\gamma s$$

where γ is the total number of samples used per local surface, and s are the number of local surfaces.

The formula for calculating the GPU memory usage of the pre-evaluation using batched images method is given by:

$$\text{GPU_mem}_{\text{Bat}} = \text{GPU_mem}_{\text{Dir}} + 192\gamma b \left\lceil \frac{p}{b} \right\rceil$$

where b is the number of patches per batch, γ is the total number of samples used for each local surface per patch, p is the number of patches and $\lceil \cdot \rceil$ denotes the number rounded up to the nearest integer. If the number of patches per batch is not set properly, the method will use more memory than is needed.

Note that these formula only account for the data explicitly stored on the GPU by the program using `vkAllocateMemory`. Any difference in the actual memory allocations, or memory used by the Vulkan application is not accounted for.

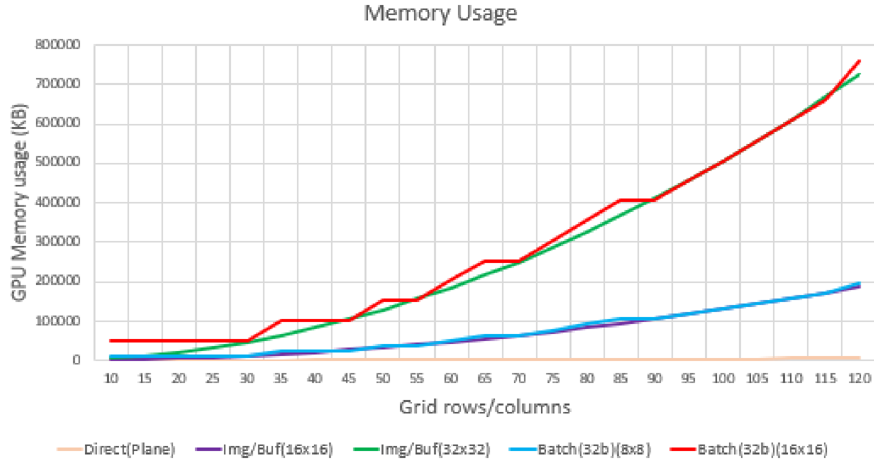
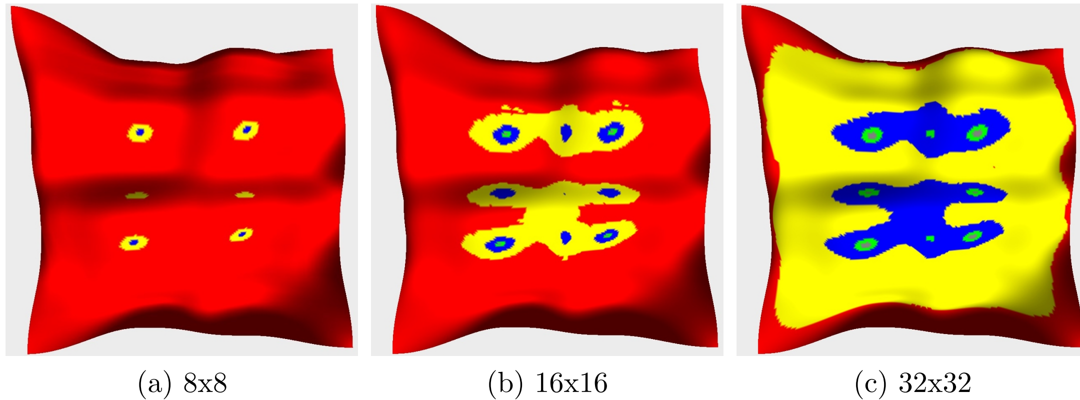


Figure 35: Plots of the formulas presented for calculating the GPU memory usage of the different local surface evaluation methods. For the pre-evaluation methods, the number inside the parenthesis is the number of samples used. The 32b after Batch means that 32x32 patches are stored per image.

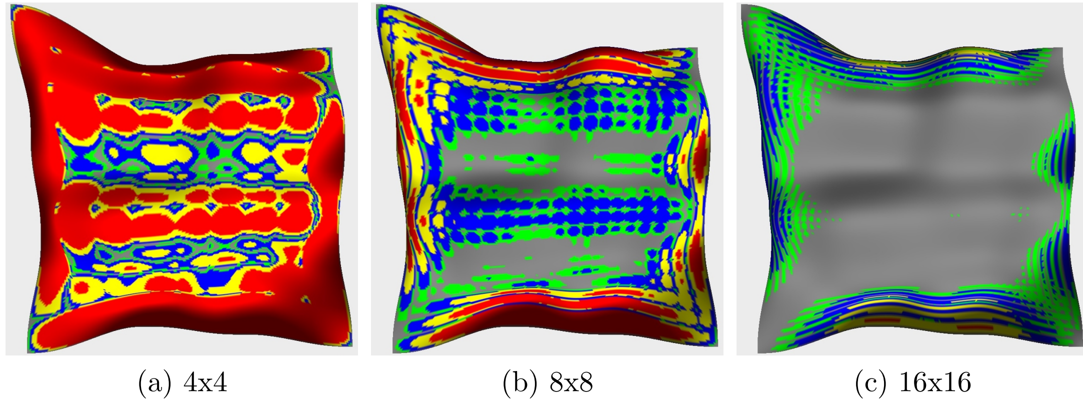
Figure 35 shows the memory usage of the different evaluation methods on a regular uniform grid of varying sizes. All methods are using plane local surfaces. The batched image pre-evaluation method is using a batch size of 32x32 patches per image.

5.6 Surface Accuracy



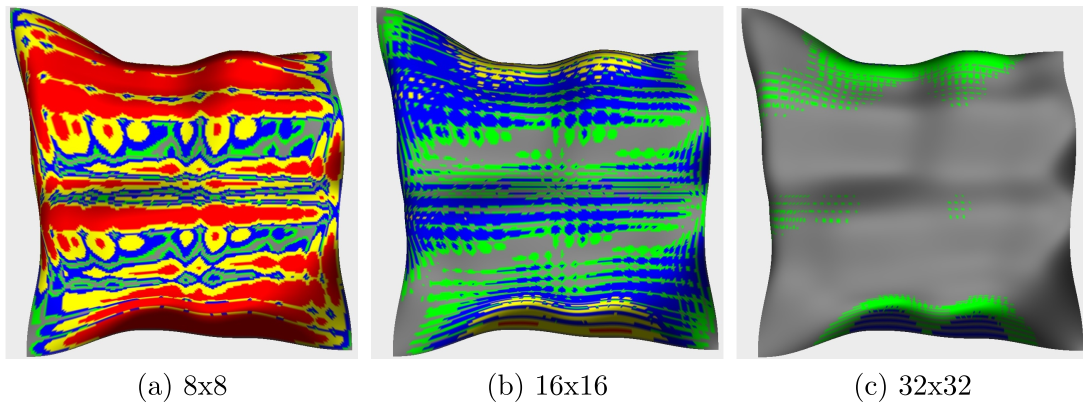
(a) 8x8 (b) 16x16 (c) 32x32

Figure 36: Surface rendered using the pre-evaluation using images evaluation method. The surface is colored based on the distance between the surface evaluated with the pre-evaluated local surfaces and the true surface in model-space. The coloring is such that grey <= 0.1, green <= 0.2, blue <= 0.5, yellow <= 1 and red > 1. The three images shows different sampling sizes of the local surfaces. (a) is 8x8, (b) is 16x16, and (c) is 32x32.



(a) 4x4 (b) 8x8 (c) 16x16

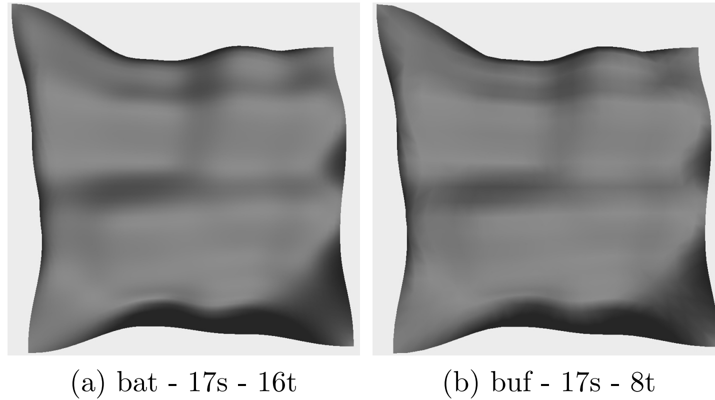
Figure 37: Surface rendered using the pre-evaluation using batched images evaluation method. The surface is colored based on the distance between the surface evaluated with the pre-evaluated local surfaces and the true surface in model-space. The coloring is such that grey <= 0.1, green <= 0.2, blue <= 0.5, yellow <= 1 and red > 1. The three images shows different sampling sizes of the local surfaces. (a) is 4x4, (b) is 8x8, and (c) is 16x16.



(a) 8x8 (b) 16x16 (c) 32x32

Figure 38: Surface rendered using the pre-evaluation using buffer evaluation method. The surface is colored based on the distance between the surface evaluated with the pre-evaluated local surfaces and the true surface in model-space. The coloring is such that grey <= 0.1, green <= 0.2, blue <= 0.5, yellow <= 1 and red > 1. The three images shows different sampling sizes of the local surfaces. (a) is 8x8, (b) is 16x16, and (c) is 32x32.

The surface accuracy of the pre-evaluation method using images is shown in Figure 36, the pre-evaluation method using batched images in Figure 37, and the pre-evaluation method using buffer in Figure 38. In the figures the results are shown for different sampling sizes. Because of the differences in the implementations, a sampling size of x for the batched images method is equivalent to a sampling size of $2x$ for the others.

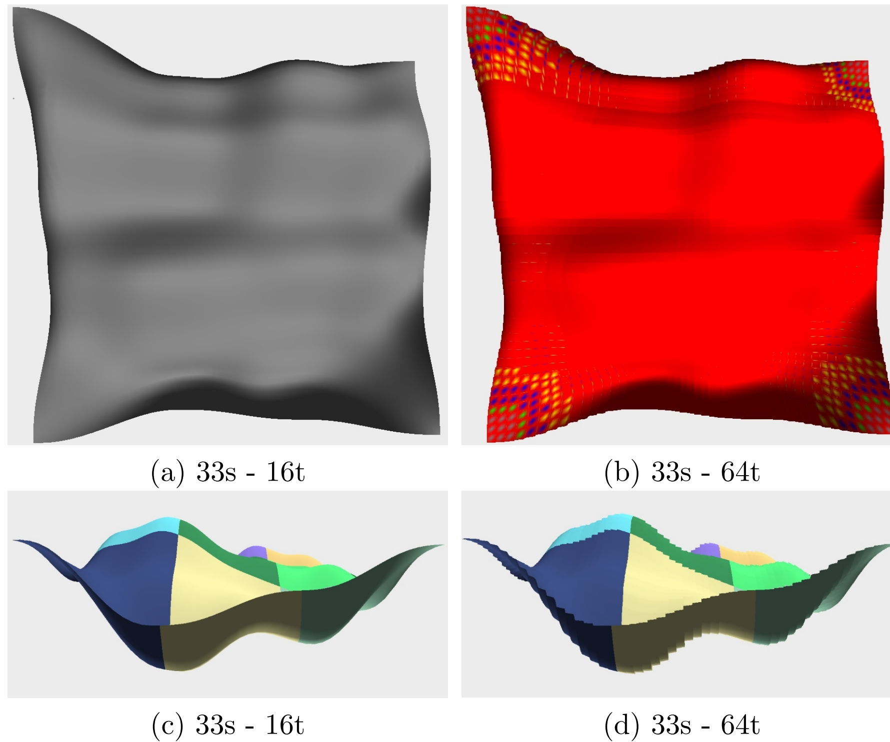


(a) bat - 17s - 16t (b) buf - 17s - 8t

Figure 39: Perfect surface accuracy displayed for the batched image and buffer pre-evaluation methods when the sampling size and tessellation levels are set such that the values in the TES correspond directly with the pre-sampled data. s denotes the samplings size and t denotes the tessellation levels.

For the batched image and buffer methods it is possible to get perfect accuracy on the surface by making sure that all the coordinates created in the tessellator correspond to the actual pre-sampled data. Because the batched image method samples only the parts of a local surface inside the patch, it is enough that the tessellation factor is 1 less than the sample size, or a multiple of the sample size minus 1 that would make all the sampling points line up with the pre-sampled data. For the buffer method the tessellation factor needs to be no bigger than the sampling size minus 1 divided by 2, because the buffer method's local surfaces might be spanning 2 patches. The tessellation level equals the number of patches, not vertices, while the sample size is equal to the number of vertices. Two examples of this can be seen in Figure 39, for other tessellation levels the accuracy is similar to that of a surface with sample size 16.

For the points to line up with the pre-sampled data it is best to use `equal_spacing` in the tessellation shaders. The fractional spacings may produce non-uniform tessellations, which will not line up with the uniform distribution of the sampled points.



(a) 33s - 16t (b) 33s - 64t

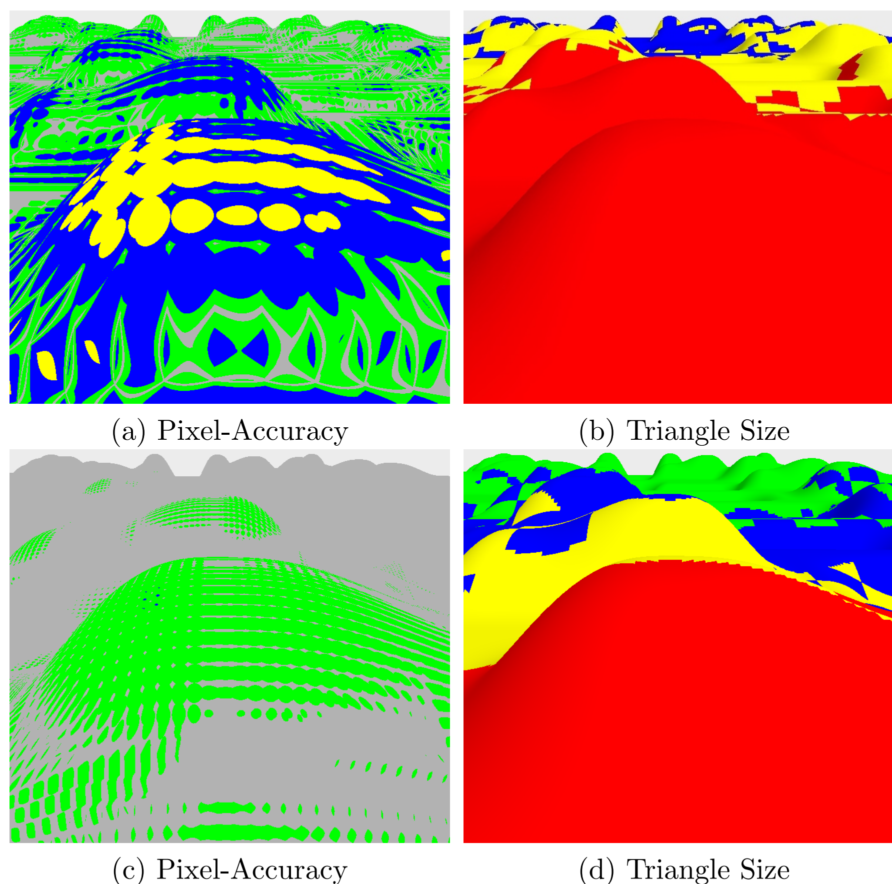
(c) 33s - 16t (d) 33s - 64t

Figure 40: The surface accuracy and renderings using the pre-evaluation with buffer method without any interpolation on the sampled data. In (a) and (b) a gray surface means that the difference between the surface evaluated on the actual control points of the local surfaces and the surface evaluated on the pre-sampled data is less than 0.1 in model space. Red means that the difference is bigger than 1. The local surfaces are sampled with 33x33 points, so with a tessellation level of 16 the surface looks smooth. In (d) you can see the surface becoming less smooth as the points in between sample values are clamped to one of the sampled values. s denotes the samplings size and t denotes the tessellation levels.

The surface accuracy and smoothness when using the pre-evaluation method with buffer without interpolation on the sampled data is shown in Figure 40. Because the method is not using any interpolation on the input data the TES must sample the local surfaces at the same points as they have been pre-evaluated. If not the surface will become jagged like in Figure 40(d), the points that get sampled in between pre-evaluated data is instead given the value of the closest data.

For a sampling size of 33x33, the surface will be equivalent to that of the direct evaluation when the tessellation levels are set to 16, 8, 4 or 2, otherwise it will be mostly bad.

5.7 Pixel-Accuracy



(a) Pixel-Accuracy (b) Triangle Size

(c) Pixel-Accuracy (d) Triangle Size

Figure 41: Pixel-accuracy and triangle size display of a surface rendered with direct evaluation. The tessellation levels of the patches are set to 10 in (a) and (b), and 30 in (c) and (d). The surfaces in (a) and (c) are colored by how different the tessellated surface is from the true surface in screen space. The color is such that if the difference is less than 0.1 pixels then it is grey, less than 0.5 is green, less than 2 is blue, less than 5 is yellow and red is greater than 5. The surfaces in (b) and (d) are colored by the longest edge of the triangles in screen space. Where grey is less than 1 pixel, green is less than 5, blue is less than 10, yellow is less than 20, and red is greater than 20.

Figure 41 shows the pixel-accuracy display and triangle size display rendering of a surface with tessellation levels fixed at 10 (a)/(b) and 30 (c)/(d). With fixed tessellation levels the small patches in the back will be much finer than the bigger patches close to the camera. For the pixel-accuracy display if the surface is colored grey or green that fragment is considered to be rendered pixel-accurate.

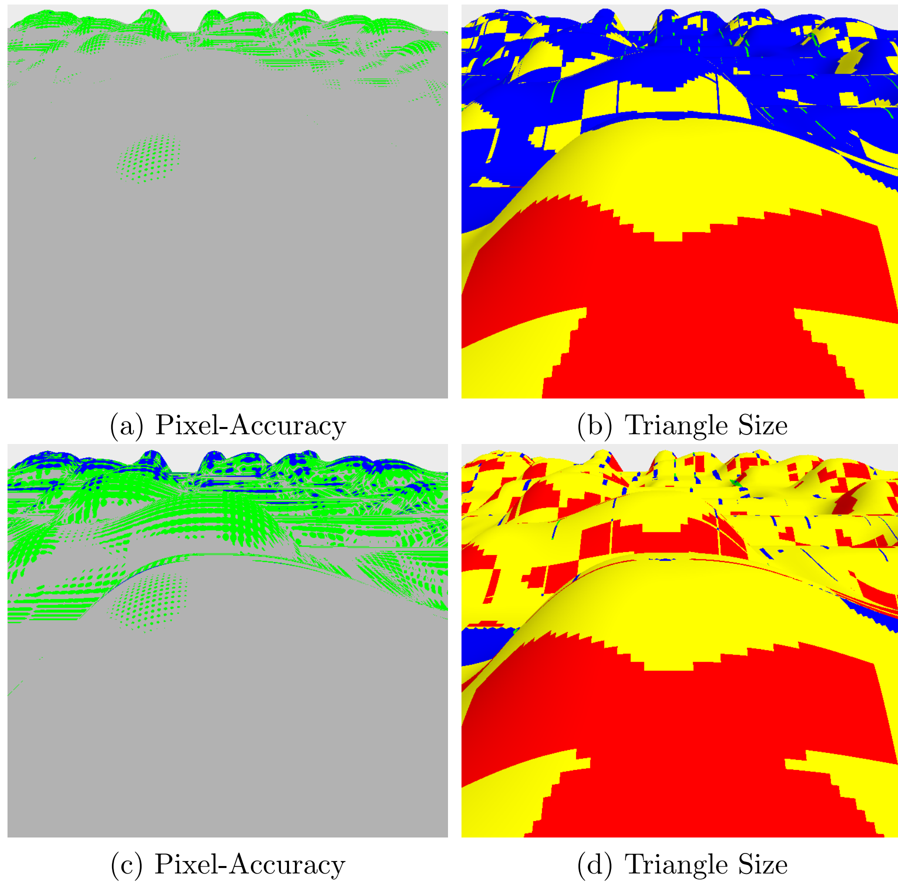
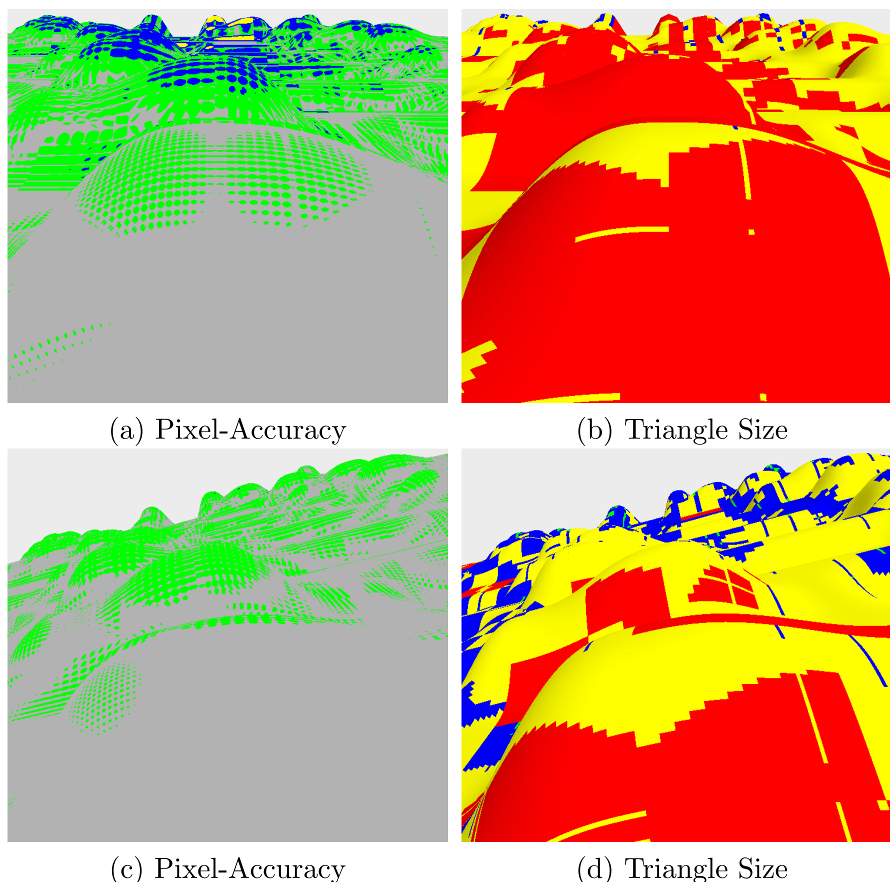


Figure 42: Pixel-accuracy and triangle size display of a surface rendered with direct evaluation. The tessellation levels of the patches are set by a dynamic view based method. (a) and (b) are using a target of 15 pixels per edge, and (c) and (d) are using a target of 30 pixels per edge. The surfaces in (a) and (c) are colored by how different the tessellated surface is from the true surface in screen space. The color is such that if the difference is less than 0.1 pixels then it is grey, less than 0.5 is green, less than 2 is blue, less than 5 is yellow and red is greater than 5. The surfaces in (b) and (d) are colored by the longest edge of the triangles in screen space. Where grey is less than 1 pixel, green is less than 5, blue is less than 10, yellow is less than 20, and red is greater than 20.

Figure 41 shows the pixel-accuracy display and triangle size display rendering of a surface with dynamic tessellation levels set by a view-based method. The view-based method takes a target pixel size for the edges and sets the tessellation levels to try and match that. The surfaces in (a)/(b) are using a target of 15 pixels per edge, and (c)/(d) a target of 30. For the pixel-accuracy display if the surface is colored grey or green that fragment is considered to be rendered pixel-accurate. Note that the target pixels per edge is used on the patch edge, not the tessellated and displaced triangles, so the colors will likely not match up with the target. The target in (a)/(b) is chosen as the biggest value where the surface is mostly pixel-accurate, and (c)/(d) is a bigger value

for comparison.



(a) Pixel-Accuracy (b) Triangle Size

(c) Pixel-Accuracy (d) Triangle Size

Figure 43: Pixel-accuracy and triangle size display of a surface rendered with direct evaluation. The tessellation levels of the patches are using a pixel-accurate rendering method, using 5 sample points per patch. The surfaces in (a) and (c) are colored by how different the tessellated surface is from the true surface in screen space. The color is such that if the difference is less than 0.1 pixels then it is grey, less than 0.5 is green, less than 2 is blue, less than 5 is yellow and red is greater than 5. The surfaces in (b) and (d) are colored by the longest edge of the triangles in screen space. Where grey is less than 1 pixel, green is less than 5, blue is less than 10, yellow is less than 20, and red is greater than 20.

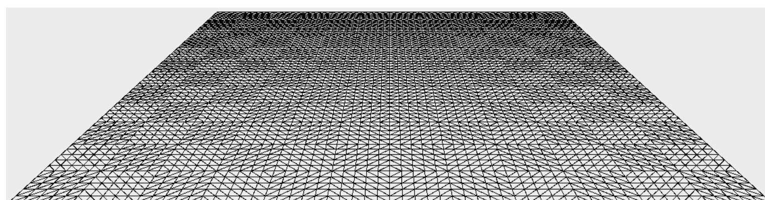
Figure 43 shows the rendering of the pixel-accuracy display and triangle size display pipelines when using the pixel-accuracy rendering pipeline. Each patch is sampled in 25 uniformly distributed points to try and find the bounds on the second order partial derivatives to set the tessellation levels to get a pixel-accurate surface. The method works better when the camera gets rotated slightly, like in (c)/(d).

Table 15 displays the number of TES invocations and the rendering times for the pixel-accuracy results shown in this section. Note that the full surface is bigger than the what is shown in the figures, and a lot of the surface is outside the view-frustum.

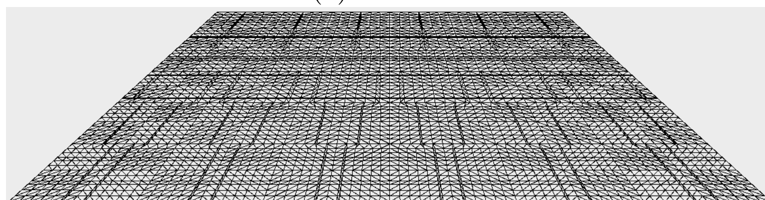
It is uncertain what happens to the patches that falls outside the view-frustum for the pixel-accurate and dynamic methods, as there is no patch frustum culling implemented.

	Stat 10	Stat 30	Dyn 15	Dyn 30	PAR 5	PAR 5 Rot
TES Invocations	73600	585600	862027	538586	686991	704066
CPU Time	0.44 ms	1.25 ms	1.61 ms	1.11 ms	1.64 ms	1.68 ms
GPU Time	0.19 ms	0.94 ms	1.28 ms	0.82 ms	1.31 ms	1.35 ms

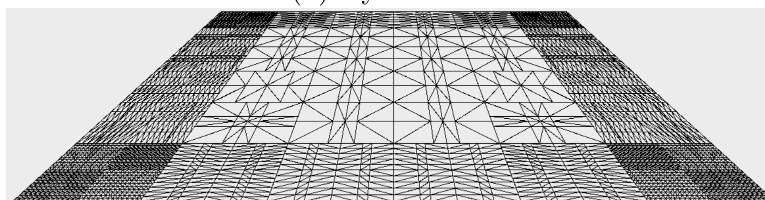
Table 15: Statistics comparing the different methods for setting the tessellation levels for the patches. Stat is using static tessellation levels of 10 and 30 for every patch. Dyn is using a view-based metric to set the tessellation levels with a target pixels per edge of 15 and 30. PAR is the pixel-accurate rendering method using 5 sample points per patch, where one of them has the camera rotated slightly to increase the accuracy.



(a) Static 10



(b) Dynamic 30



(c) Pixel-Accurate 5

Figure 44: A 6x6 grid lattice rendered in wireframe using different level of detail methods. (a) is using static tessellation levels of 10. (b) is using a view-based metric with a target of 30 pixels per edge. (c) is using a pixel-accurate rendering method.

Figure 44 shows a flat 6x6 grid lattice rendered in wireframe with static tessellation levels of 10, dynamic tessellation levels targeting 30 pixels per edge and pixel-accurate rendering with 5x5 samples.

6 Discussion

6.1 Bézier Patchwork

A bit surprisingly, the GPU example outperforms the CPU example when the number of vertices becomes larger. Even though the GPU rendering pipeline is doing a lot more work in the shaders than the CPU pipeline. It seems creating the geometry in the tessellation shaders is faster, even though the Bézier surface of degree 3 is also evaluated for every patch, every frame. Some of the difference could be explained by the CPU example having more vertex shader invocations than the GPU example have TES invocations. But the difference in rendering times is larger than the difference in invocations.

The GPU example far outperforms the CPU example when simulation is added, as was expected. The GPU example already evaluates the surface every frame, so the only added overhead is simulating the control points and uploading them to the GPU. Meanwhile the CPU example has to simulate the control points, and then re-evaluate the surface on the CPU before re-uploading all the vertices to the GPU. Adding in some multi-threading helps on the performance.

It should be noted that the CPU example does not try to be the most efficient it could possibly be, but simply tries to mimic the GPU example as much as possible by performing the same tasks, but on the CPU. However, the GPU example is not very efficient when it comes to simulation either. When the control points are changed, the vertex buffer is freed and re-allocated on the GPU.

6.2 Lattice Framework

The framework is capable of rendering tensor-product blending surfaces on the GPU using the tessellation shader steps. The framework works by setting up patches for rendering based on the geometry added by the user, and then tessellating and evaluating them on the GPU. In addition to rendering the global surface, the framework can render the local surfaces, the lattice grid with the loci colored by their type, the normals of the global surface, and the global and local surfaces in wireframe mode. Lattices can be created using helper functions for creating a grid, a cylinder, a sphere or a torus (Displayed in Figure 24). Or the patches can be input individually by the user, like in the irregular grid example in Figure 28.

The framework supports some irregular grids with T-loci. As can be seen in Figure 28 the framework correctly handles T-loci on any edge of a face. It also works when one or both of the terminal points are on the boundary. It can not handle more than one T-locus per face, or when T-loci share terminal points.

It is possible to animate the lattice by adding simulators that perform affine transformations on the local surfaces' model matrices. Three different simulators are implemented and can be used alone or together with other simulators.

6.3 Evaluation Methods

Five different evaluation methods are possible to use for the rendering. Direct evaluation inside the TES, or pre-evaluating the local surfaces and storing them inside individual images, batched images or a shared buffer. Three different local surface types are implemented and can be used for either evaluation method. The local surface types are planes and Bézier surfaces of degree 2 or 3.

For direct evaluation the complexity of the local surface evaluation has a big impact on performance, especially as the number of patches and the tessellation levels grows. This is to be expected as the function for evaluating the local surfaces is called 4 times for every patch. For the pre-evaluation methods the type of local surface has no effect on the evaluation time. However, sampling the pre-evaluated data and interpolating the values if the sampling values are between the data adds some overhead. The buffer method that does not do any interpolation when sampling the data is about 2 times faster than the buffer method that uses interpolation.

Using multiple draw calls definitely adds some overhead to the rendering, this is most notable when a lot of patches with low tessellation levels are rendered. The extra data added to the control point buffer did not seem to have any effect on the rendering. However, the pre-evaluation methods with bigger sample sizes did add some extra rendering time. The extra data is never accessed by the shaders, that could be the reason for no overhead. The single draw call method would most likely still outperform a multi draw call method with small buffers, but extra testing would be needed to tell for certain.

The surface accuracy of the batched image and buffer methods are the best ones. And if the tessellation levels are static it is possible to set the sample sizes to guarantee a fully accurate surface. The buffer method is better than the batched image one around the edges. One explanation for this is that the way that the batched method works will make it so every local surface for every patch will have the same number of samples. The buffer method however will have local surfaces along the boundary and in the corners that will have double the amount of sample size in one or both parameter directions, compared to the center and batch local surfaces. So it could be the extra samples that make the surface more accurate along the edges.

It is unexpected that the image method is worse than the batched image method, considering they use the same parameters to create the image and samplers. The only explanation is that the coordinates used to sample the image are not correct. When mapping the patch u,v-coordinates into local coordinates for a given patch inside a batched image the translation adds 0.5 units to move the values into the center of the texel. This is not done for the image, so that could be the problem.

It is difficult to choose the best evaluation method without knowing which purpose it will be used for. If surface accuracy is very important then direct evaluation is the best option, unless only static tessellation levels are going to be used. If more complex local surfaces like Bézier degree 3 or many different types of local surfaces are going to be used then going with a more general approach like the batched image method could be good. This would also make it possible to have different local surface in the same lattice without any extra modification to the shaders. It is important to also consider

the memory usage, as the pre-evaluation methods can use up a lot of memory if big sample sizes and a lot of patches are used.

Another thing to consider is if the control points of the local surfaces will be changed during the rendering. As it is now the pre-evaluated data would then need to be re-evaluated on the CPU, which will is a costly operation.

6.4 Adaptive Level of Detail

If the camera is free to move around in the scene it is definitely a good idea to use some form of adaptive level of detail. Just setting the tessellation levels for all patches to some value can make the visual quality lacking if patches close to the camera get under-tessellated. And the performance might suffer if several patches get over-tessellated, from the results it is evident how quickly the performance decreases when the tessellation levels increases.

The dynamic LOD method works decently. The target pixels per edge can be adjusted to get something that works well depending on how the patch surfaces are different from the edges. The method struggles a bit when there is variation in how much the patches are displaced from their edges. Here the more displaced a surface is the more likely it is to be under-tessellated, while the more flat patches will likely be over-tessellated.

The pixel-accurate rendering method does somewhat work. When viewing the lattice straight on it has some problems, but gets better when rotating the camera slightly. There are problems with using a method that requires finding the bounds on the second order partial derivatives when using a surface construction where that is not possible. An attempt to get around this by evaluating the surface in the TCS and using the best values found somewhat works, but in addition to not being the true bounds, it also adds some extra work for the GPU. The pixel-accurate method is by far the best for the flat surface test, even if it does struggle with the edges.

It is a bit unexpected that the static pixel-accurate rendering has the least TES invocations. From the images it would be expected that the rotated pixel-accurate rendering method would have less TES invocations. However, the images only shows a part of the full surface.

6.5 Future Work

6.5.1 Lattice

The framework supports animation by performing affine transformations on the local surfaces, a next step could include adding the possibility of also moving the local surfaces' control points.

If further work is to be done with this framework it would be a good idea to decide on just one evaluation method, and then do some refactoring/cleaning up all the places where extra steps are taken to account for all the different methods. If a pre-evaluation method is chosen it would be a good idea to move the evaluation of the local surfaces

into some compute pipeline or compute shader. This pipeline should also be able to only evaluate the specified local surfaces, opening up the possibility to save evaluation time if only one local surface needs re-evaluation. Depending on the sample size and amount of local surface, evaluating them on the CPU before starting up the program can take a lot of time.

With only one evaluation method it could be beneficial to go back to compiling the shaders outside of the codebase, especially if a pre-evaluation method is chosen. Compiling the shaders inside the code adds a lot of overhead during the startup of the program, which can be a problem when starting the program often. Another possibility is to save the binary SPIR-V files after compilation, and then only recompile them if the shaders are changed. It could also be that the use of shaderc is not optimal right now. Compiling the shaders before did not take a lot of time at all, maybe glslang is faster, or there is some optimization that can be done with shaderc.

The two methods for creating local surface control points should be fixed so only one is necessary for all cases. The user could also be given more control over this process, either by being able to set up the control points of the local surfaces, or by being able to control how much the control points are offset from the positions generated by the framework.

The `Lattice` class uses both glm::vec3 and OpenMesh::Vec3f for storing vectors. And some places these also clash and one type has to be converted to the other. OpenMesh does allow the user to provide their own types for storing the points. Trying to do this and removing the conflicts would be a good idea.

6.5.2 Vulkan

Better use of Vulkan would be a good next step. The synchronization of the base Vulkan framework used has been improved, but there is still a lot of upgrade potential there. One issue is that when buffers are changed they are just destroyed and re-allocated. So when animating the matrix buffer has to be destroyed, then re-created and copied to local GPU storage. It would probably be a better idea to just perform a copy command before the rendering starts to move the new matrices. This would also be beneficial if only a small amount of the matrices needed updating.

Another option could be to just start from scratch and build a better base Vulkan framework, or find a better one to use. Khronos' example framework is more up to date and could be a better option. The Vulkan resources seems to be cleaned up properly right now (i.e. no warnings/errors from the validation layers) when only a single lattice is used. However, if more than one lattice is present when the program is closed the validation layers will complain. This should be investigated.

6.5.3 Patch Culling

In the graphics pipeline fragments are culled if they are outside the view frustum. However, this operation is done after the tessellation shader steps. Even if a patch ends up getting culled, all the tessellation shader invocations and the work done in them will

still be done. It would therefore be advantageous to cull the patches as early as possible, before any work is done on them. One way to achieve this is frustum culling, i.e., culling any patches that fall outside the view frustum. One simple way of achieving this can be found in Nvidias tessellation sample[1]. A patch where any relevant outer tessellation levels are set to 0 will not move past the TCS, effectively culling it. As an example, terrain rendering with a big lattice grid would involve lots of patches that fall outside the frustum as the camera is moved around.

It is also possible to cull the back-facing primitives in the graphics pipeline, but similarly to above, all the TES invocations would still have to be performed. Another way of reducing tessellation shader invocations include back-patch culling[15][16], culling any patches where all the faces would be pointing away from the camera. And occlusion culling, i.e. culling patches that would not be seen because they are behind other opaque geometry. A sphere for example would only show about half its patches at any time during rendering.

References

[1] I. Cantlay, "DirectX 11 Terrain Tessellation," 2011. Nvidia Whitepaper.

[2] J. Bratlie, R. Dalmo, and B. Bang, "Evaluation of Smooth Spline Blending Surfaces Using GPU," in *Curves and Surfaces* (J.-D. Boissonnat, A. Cohen, O. Gibaru, C. Gout, T. Lyche, M.-L. Mazure, and L. L. Schumaker, eds.), (Cham), pp. 60–69, Springer International Publishing, 2015.

[3] C. González, M. Pérez, and J. M. Orduña, "A Hybrid GPU Technique for Real-Time Terrain Visualization," 2016.

[4] M. O. Özek and E. Demir, "Pixel-Based Level of Detail on Hardware Tessellated Terrain Rendering," in *ACM SIGGRAPH 2017 Posters*, SIGGRAPH '17, (New York, NY, USA), pp. 1–2, Association for Computing Machinery, 2017.

[5] Q.-y. Mao, Y.-j. Chen, and X. Yuan, "Dynamic Grid Sea Surface Simulation Using Tessellation," *DEStech Transactions on Engineering and Technology Research*, pp. 515–520, 2018.

[6] M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer, "Real-Time Rendering Techniques with Hardware Tessellation," *Comput. Graph. Forum*, vol. 35, p. 113–137, Feb. 2016.

[7] M. Nießner, C. Loop, M. Meyer, and T. Derose, "Feature-Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces," *ACM Trans. Graph.*, vol. 31, Feb. 2012.

[8] Pixar, "OpenSubdiv Documentation." `http://graphics.pixar.com/opensubdiv/docs/intro.html`. Last Accessed: 05.06.20.

[9] M. Nießner, C. T. Loop, and G. Greiner, "Efficient Evaluation of Semi-Smooth Creases in Catmull-Clark Subdivision Surfaces," in *Eurographics (Short Papers)*, pp. 41–44, 2012.

[10] A. Lakså and B. Bang, "Surface Constructions on Irregular Grids," in *Large-Scale Scientific Computing* (I. Lirkov, S. D. Margenov, and J. Waśniewski, eds.), (Cham), pp. 385–393, Springer International Publishing, 2015.

[11] J. Hjelmervik, "Direct Pixel-Accurate Rendering of Smooth Surfaces," in *Mathematical Methods for Curves and Surfaces* (M. Floater, T. Lyche, M.-L. Mazure, K. Mørken, and L. L. Schumaker, eds.), (Berlin, Heidelberg), pp. 238–247, Springer Berlin Heidelberg, 2014.

[12] J. M. Hjelmervik and F. G. Fuchs, "Interactive Pixel-Accurate Rendering of LR-Splines and T-Splines," in *EG 2015 - Short Papers* (B. Bickel and T. Ritschel, eds.), pp. 65–68, The Eurographics Association, 2015.

[13] Y. I. Yeo, S. Bhandare, and J. Peters, "Efficient Pixel-accurate Rendering of Animated Curved Surfaces," in *Mathematical Methods for Curves and Surfaces* (M. Floater, T. Lyche, M.-L. Mazure, K. Mørken, and L. L. Schumaker, eds.), (Berlin, Heidelberg), pp. 491–509, Springer Berlin Heidelberg, 2014.

[14] Y. I. Yeo, L. Bin, and J. Peters, "Efficient Pixel-Accurate Rendering of Curved Surfaces," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, (New York, NY, USA), p. 165–174, Association for Computing Machinery, 2012.

[15] C. Loop, M. Nießner, and C. Eisenacher, "Effective Back-Patch Culling for Hardware Tessellation," in *VMV 2011 - Vision, Modeling and Visualization*, pp. 263–268, 2011.

[16] R. Concheiro, M. Amor, E. Padrón, and M. Doggett, "Efficient culling techniques for interactive deformable NURBS surfaces on GPU," in *VISIGRAPP 2016 - Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, pp. 17–27, SciTePress, 2016.

[17] The Khronos Vulkan Working Group, "Vulkan 1.2.133 - A Specification (with KHR extensions)," 2020.

[18] Khronos, "OpenGL Wiki - Tessellation." `https://www.khronos.org/opengl/wiki/Tessellation`. Last Accessed: 05.06.20.

[19] L. T. Dechevsky, B. Bang, and A. Lakså, "Generalized expo-rational B-splines," *International Journal of Pure and Applied Mathematics*, vol. 57, pp. 833–872, 2009.

[20] A. Lakså, "Pre-evaluation and interactive editing of B-spline and GERBS curves and surfaces," *AIP Conference Proceedings*, vol. 1910, no. 1, p. 060005, 2017.

[21] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, "OpenMesh - a generic and efficient polygon mesh data structure," 2002.

[22] L. Kobbelt, "OpenMesh 8.0 Documentation." `https://www.graphics.rwth-aachen.de/media/openmesh_static/Documentations/OpenMesh-8.0-Documentation/index.html`. Last Accessed: 05.06.20.

[23] "OpenGL Mathematics." `https://glm.g-truc.net/0.9.9/index.html`. Last Accessed: 05.06.20.

[24] Khronos, "Vulkan Website." `https://www.khronos.org/vulkan/`. Last Accessed: 05.06.20.

[25] J. A. Shiraef, "An exploratory study of high performance graphics application programming interfaces," 2016. Masters Theses and Doctoral Dissertations. `https://scholar.utc.edu/theses/446`.

[26] Khronos, "OpenGL Wiki - SPIR-V." https://www.khronos.org/opengl/wiki/SPIR-V. Last Accessed: 05.06.20.

[27] Khronos, "Glslang." https://github.com/KhronosGroup/glslang. Last Accessed: 05.06.20.

[28] Google, "Shaderc." https://github.com/google/shaderc. Last Accessed: 05.06.20.

[29] O. Cornut, "Dear ImGui." https://github.com/ocornut/imgui. Last Accessed: 05.06.20.

[30] G. A. Johansen, "Meshless Animation Framework." https://github.com/gustavaj/OMMeshlessAnimationFramework. Last Accessed: 05.06.20.

[31] G. A. Johansen, "Gumbo Tessellation Vulkan." https://github.com/gustavaj/GumboTessellationVulkan. Last Accessed: 05.06.20.

[32] AMD, "V-EZ." https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ. Last Accessed: 05.06.20.

[33] S. Willems, "Vulkan C++ examples and demos (commit fcb0a2a)." https://github.com/SaschaWillems/Vulkan. Last Accessed: 05.06.20.

[34] A. Overvoorde, "Vulkan Tutorial: Synchronization." https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Rendering_and_presentation#page_Synchronization. Last Accessed: 05.06.20.

[35] P. Rideout, "Quad Tessellation with OpenGL 4.0." https://prideout.net/blog/old/blog/index.html@tag=tessellation.html. Last Accessed: 05.06.20.

[36] S. Willems, "GPU Info." https://vulkan.gpuinfo.org/. Last Accessed: 05.06.20.

Appendix A GUI

VulkanLattice

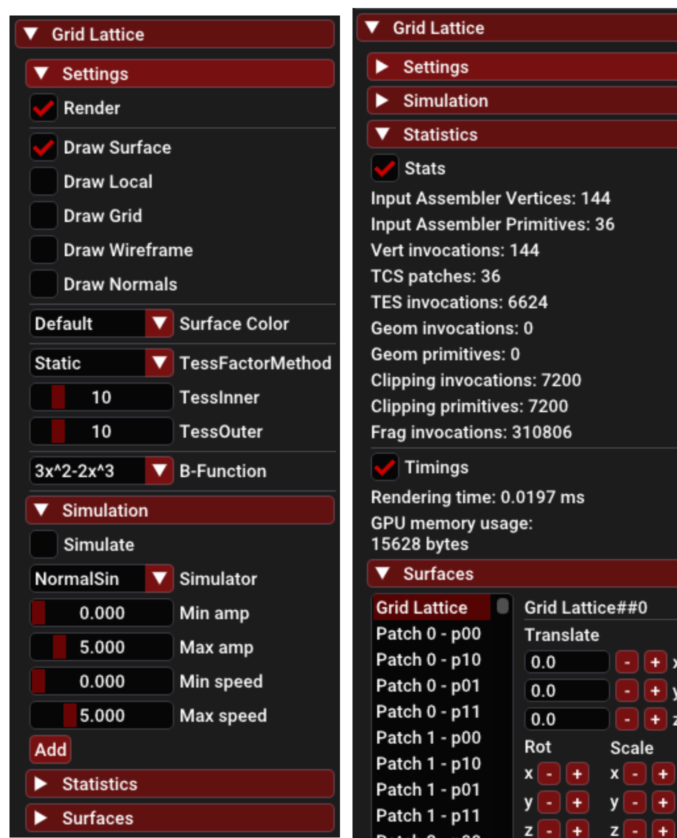


Figure 45: The controls and statistics available in the `VulkanLattice` menu. Settings controls different parameters for the rendering. Simulation is used for adding/removing and modifying different simulators. Statistics displays pipeline statistics and timings and GPU memory usage. Surfaces allows editing the transformation matrix of the lattice and its local surfaces.

Figure 45 shows the controls and information available in the `VulkanLattice` menu. The first set of checkboxes decides which pipelines to use while rendering, and changing any of them will require the command buffers to be rebuilt. Changing how the surface is colored will also result in a rebuild of the command buffers. The rest of the controls in the settings tab changes values inside the uniform buffer, and any changes to them will lead to the uniform buffer being copied to GPU memory.

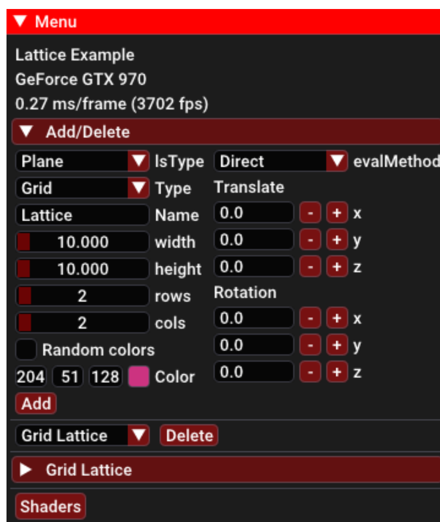
The simulation tab controls everything regarding simulation. If the simulate checkbox is checked then the local surface matrices will be updated in the render loop. With the drop down menu different simulators can be chosen. The sliders are for setting the range of possible values for the simulators. After a simulators has been added it can either be updated or removed. Several simulators can be added at the same time.

Adding/updating/removing simulators calls the equivalent simulator functions in the `Lattice` class.

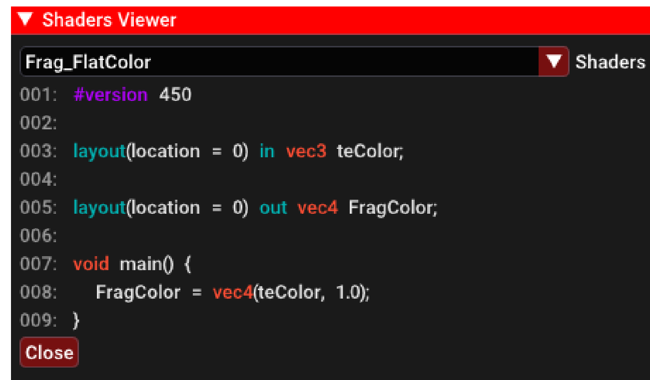
The statistics tab shows different pipeline statistics that shows how many invocations are executed by each shader stage, the number of vertices and primitives input in the draw call, the amount of TCS patches, and how many primitives pass the clipping stage. The total time it took for the lattice's draw calls is shown in rendering time. GPU memory usage shows the size of all the data that has been sent to GPU memory, not the actual size of the memory allocations. The queries will add a little bit of extra work on the GPU, so unchecking the boxes will lead to those queries not being done.

The last tab allows editing of the `Lattice's` and the local surfaces' transformation matrices. The translation can be manually input, and will show the actual values of the translation at all times. For rotation, the buttons will rotate around the chosen axis by +/- 10 degrees. The scale buttons will scale by 0.9/1.1 along the chosen axis. Any editing of the local surfaces will lead to all the matrices having to be uploaded to GPU memory again. If the matrix of the lattice is changed then the uniform buffer will be uploaded again, since the lattice's matrix is stored there.

LatticeExample



(a)



(b)

Figure 46: (a) Menu for the lattice examples. Shows the frame-time and fps. Allows the user to add and delete lattices. Contains the menus for the individual lattices and a button to open the shader viewer. (b) A window for viewing shaders inside the program. Some keywords are colored to make the shaders more readable.

Figure 46 (a) shows the menu implemented in `LatticeExample`. From the "Add/Delete" tab it is possible to add new lattices based on the specified parameters. The menu allows the user to set up every parameter that could be set in code. For the type it is possible

to chose Grid, Cylinder, Sphere or Patch, which lets the user create a list of patches to use for the creation. Lattices can also be selected from a drop-down list and deleted.

In many cases the shaders are created by combining a number of predefined or dynamically created strings. This makes it hard to know what the final shader is going to look like. The shader viewer window allows the user to see all the shaders that have been loaded into the program. The shaders have some keywords and user defined types colored by different colors to make it more readable. All the keywords are added to an unordered_map where the word is they key, and the value is an index into a vector of different colors. This map is then used to color each word when displaying a shader. An example of a simple fragment shader is shown in Figure 46 (b).

Appendix B Setup Guide

This setup guide will explain how to build and run the project on Windows using Visual Studio 2019. To follow this guide the following software is required:

- Visual Studio 2019 (`https://visualstudio.microsoft.com/vs/`)
- Git Bash (`https://gitforwindows.org/`)
- CMake GUI (`https://cmake.org/download/`)
- Python 3 (`https://www.python.org/downloads/`)

MeshlessAnimation Framework

The source code from the project can be found in the supplementary material, or cloned using git. First open up Git Bash and change directory to where the project will be located. Then clone the project with:

- git clone git@github.com:gustavaj/OMMeshlessAnimationFramework.git OMMeshlessAnimationFramework

The vulkan header and lib file comes with the source code. If there are any problems with it, the Vulkan SDK can be downloaded from `https://www.lunarg.com/vulkan-sdk/`, where those files can be found.

GLM

Next, move into the "OMMeshlessAnimationFramework/external" folder and clone GLM:

- git clone git@github.com:g-truc/glm.git glm

OpenMesh 8.0

To download OpenMesh 8.0 go to `https://www.graphics.rwth-aachen.de/software/openmesh/download/`. Scroll down to Old Versions, OpenMesh 8.0 and download the precompiled binaries "64-bit without apps, 8.0 (static) for VS2017". Run the downloaded .exe file to install OpenMesh. When the installer asks for the installation path, put it inside the external folder:

- "path-to-project\OMMeshlessAnimationFramework\external\OpenMesh 8.0"

Shaderc v2019.1

To get Shaderc, navigate to the external folder with Git Bash and clone it:

- git clone git@github.com:google/shaderc.git shaderc

After cloning it, it is very important to change to the correct commit. The newest version compiles shaders inside the project about 10 times slower. So navigate inside the Shaderc folder and checkout the correct commit with:

- git checkout f76bb2f

Then, to get the third party libraries that Shaderc need, open file explorer and navigate to the utils folder inside shaderc. Inside the folder, rename "git-sync-deps" to "git-sync-deps.py" and double click it.

Now open up CMake GUI. Set the source code and binaries directories to point into the shaderc folder. Click "Configure" and set "Visual Studio 16 2019" as the generator. Set up the options as shown in Figure 47, and click "Generate". The most important option is changing `LLVM_USE_CRT_DEBUG` and `LLVM_USE_CRT_RELEASE` to MDd and MD, respectively.

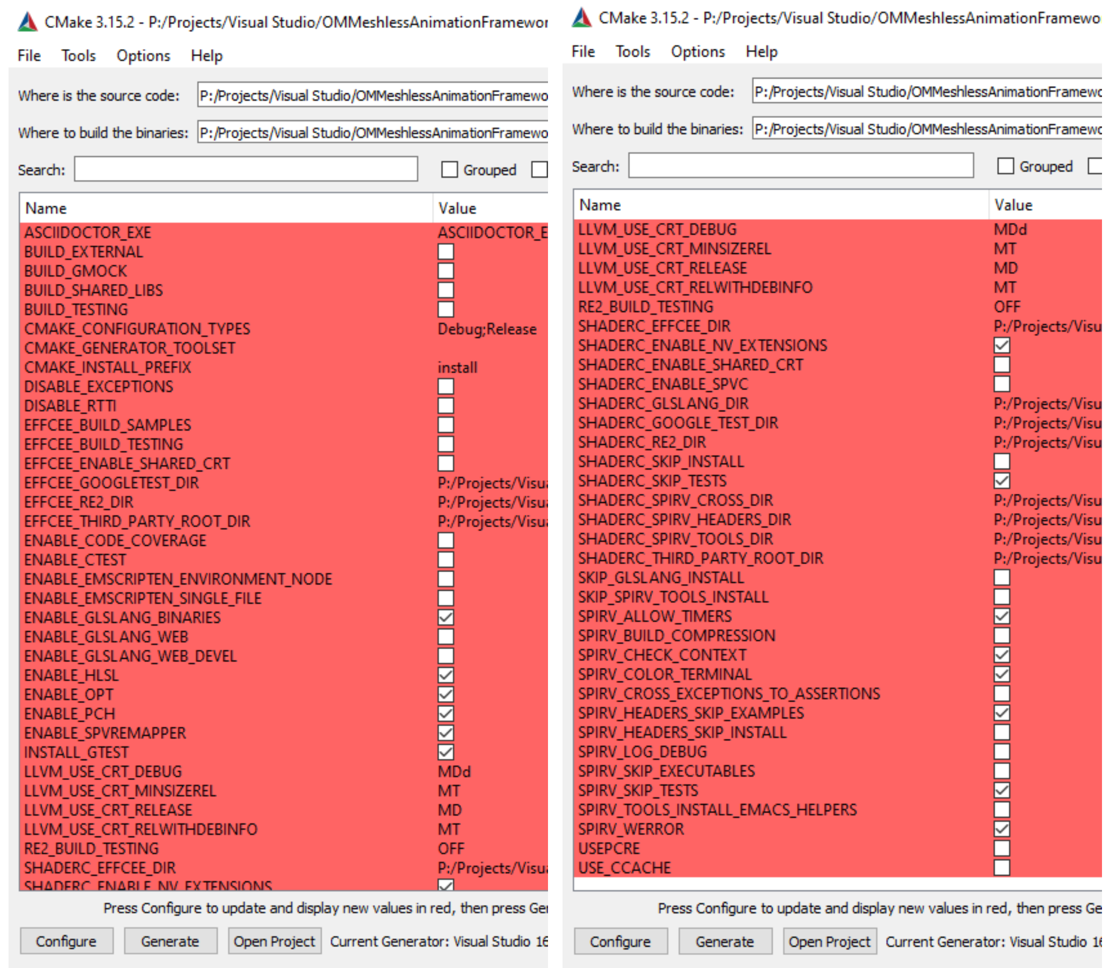


Figure 47: CMake Shaderc Options

After running CMake you should have a file called "shaderc.sln" inside the shaderc folder, open it in Visual Studio. Now, right click on "shaderc_combined_genfile" inside the Solution Explorer and click build. Do this in both debug and release build. If everything is successful you are now ready to run the project.

Running the Project

First open “OMMeshlessAnimationFramework.sln” in Visual Studio. If everything was set up correctly inside the correct folders you should be able to just build and run the project.

Possible Errors

If Visual Studio complains about not finding the one of the libraries, you might have put them in the wrong place. You can open up the “Project -> Properties” window and check the paths. Inside the “C/C++ -> General” tab you can see the include directories by clicking on the “Additional Include Directories” row and clicking the arrow on the right side and <Edit...>. If everything looks fine, check the “Linker -> General” tab. Similarly to the include directories, open up the “Additional Library Directories”. You should make sure that the lib files listed in “Linker -> Input -> Additional Dependencies” can be found inside the “Additional Library Directories”.

If Visual Studio gives an error about mismatch detected for RuntimeLibrary for “shaderc_combined.lib”, then you need to set the options correctly when generating the solution files for Shaderc in CMake.

Gumbo Tessellation Vulkan

The setup for the Gumbo Tessellation Vulkan project is very similar to the previous project. The source code can be found in the supplementary source code or cloned with git using:

- git clone git@github.com:gustavaj/GumboTessellationVulkan.git GumboTessellationVulkan

Gumbo only depends on GLM, so you can skip the OpenMesh and Shaderc steps.

Appendix C Problem Description

Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering
UiT - The Arctic University of Norway

Meshless Animation Framework

Gustav Adolf Johansen

Thesis for Master of Science in Technology / Sivilingeniør

Problem description

Given a capable hardware architecture (which supports tessellation shader control) facilitating an explicit geometry construction that can be evaluated directly on the client side (GPU). Then the evaluation (rendering) and the simulation (animation) can reside in separate systems, and the evaluation system is constant/stable with respect to chaning coefficeints from the simulation system.

A composed framework should be meshless, meaning that the meshing is delegated to the graphics API through tessellation shaders, where the geometry (spline) data is communicated "one-way" to the graphics API. The simulation system is independent of the rendering system, but utilizes the same spline basis and underlying coefficients. One example application is animations / skinning.

Objectives:

Assuming a hierarchical surface representation in terms of blending splines, consisting of local surface patches with coefficients, it is expected that a load balancing between pre-evaluation and direct evaluation can be investigated.
Composition of GPU spline evaluation kernels can be approached in different ways, one such approach can be to design a domain spesific language (DSL) compiler which produces host and client code.

Consider the tensor product Bezier implementation examples from nVidia, and the provided prototype.

1. Compose patchwork from Bezier patches on the GPU (tessellation shader steps).

2. Tensor product blending splines (on the GPU):
  i. Send coefficients to the shader
  ii. Perform tessellation and pixel-accurate rendering

3. Simulation (e.g. animations) by changing the coefficients.

4. Extend to tensor-product blending splines.

Implement a prototype to verify the hypothesis and show the benefits it will bring to rendering applications.
Possible technologies include, but are not limited to, Vulkan and OpenCL.

Dates

Date of distributing the task:	<06.01.2020>
Date for submission (deadline):	<08.06.2020>

Contact information

Candidate	Gustav Adolf Johansen gjo067@post.uit.no
Supervisor at UiT-IVT	Jostein Bratlie jostein.bratlie@uit.no
Supervisor at UiT-IVT	Rune Dalmo rune.dalmo@uit.no

General information

This master thesis should include:

- ❄ Preliminary work/literature study related to actual topic
  - A state-of-the-art investigation
  - An analysis of requirement specifications, definitions, design requirements, given standards or norms, guidelines and practical experience etc.
  - Description concerning limitations and size of the task/project
  - Estimated time schedule for the project/ thesis
- ❄ Selection & investigation of actual materials
- ❄ Development (creating a model or model concept)
- ❄ Experimental work (planned in the preliminary work/literature study part)
- ❄ Suggestion for future work/development

Preliminary work/literature study

After the task description has been distributed to the candidate a preliminary study should be completed within 3 weeks. It should include bullet points 1 and 2 in "The work shall include", and a plan of the progress. The preliminary study may be submitted as a separate report or "natural" incorporated in the main thesis report. A plan of progress and a deviation report (gap report) can be added as an appendix to the thesis.

In any case the preliminary study report/part must be accepted by the supervisor before the student can continue with the rest of the master thesis. In the evaluation of this thesis, emphasis will be placed on the thorough documentation of the work performed.

Reporting requirements

The thesis should be submitted as a research report and could include the following parts; Abstract, Introduction, Material & Methods, Results & Discussion, Conclusions, Acknowledgements, Bibliography, References and Appendices. Choices should be well documented with evidence, references, or logical arguments.

The candidate should in this thesis strive to make the report survey-able, testable, accessible, well written, and documented.

Materials which are developed during the project (thesis) such as software / source code or physical equipment are considered to be a part of this paper (thesis). Documentation for correct use of such information should be added, as far as possible, to this paper (thesis).

The text for this task should be added as an appendix to the report (thesis).

General project requirements

If the tasks or the problems are performed in close cooperation with an external company, the candidate should follow the guidelines or other directives given by the management of the company.

The candidate does not have the authority to enter or access external companies' information system, production equipment or likewise. If such should be necessary for solving the task in a satisfactory way a detailed permission should be given by the management in the company before any action are made.

Any travel cost, printing and phone cost must be covered by the candidate themselves, if and only if, this is not covered by an agreement between the candidate and the management in the enterprises.

If the candidate enters some unexpected problems or challenges during the work with the tasks and these will cause changes to the work plan, it should be addressed to the supervisor at the UiT or the person which is responsible, without any delay in time.

Submission requirements

This thesis should result in a final report with an electronic copy of the report including appendices and necessary software, source code, simulations and calculations. The final report with its appendices will be the basis for the evaluation and grading of the thesis. The report with all materials should be delivered according to the current faculty regulation. If there is an external company that needs a copy of the thesis, the candidate must arrange this. A standard front page, which can be found on the UiT internet site, should be used. Otherwise, refer to the "General guidelines for thesis" and the subject description for master thesis.

The supervisor(s) should receive a copy of the the thesis prior to submission of the final report. The final report with its appendices should be submitted no later than the decided final date.