TU WIEN Informatics

Verbesserte Dreieckscodierung für gecachte adaptive Tessellation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Linus Emanuel Horváth
Matrikelnummer 01525536

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Wien, 7. Februar 2022

Linus Emanuel Horváth	Michael Wimmer

TU WIEN Informatics

Improved Triangle Encoding for Cached Adaptive Tessellation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Linus Emanuel Horváth
Registration Number 01525536

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Vienna, 7th February, 2022

Linus Emanuel Horváth	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Linus Emanuel Horváth

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Februar 2022

Linus Emanuel Horváth

Danksagung

Ich möchte speziell Bernhard Kerbl dafür danken, dass er mich auf dem deutlich länger als erwarteten Weg beim Schreiben dieser Bachelorarbeit unterstützt hat. Ohne ihn hätte ich es nicht geschafft, zusätzlich ein Poster mit dem Hauptteil dieser Arbeit bei der High-Performance Graphics Konferenz 2020 einzureichen, alle Überprüfungen zu bestehen und die Gelegenheit zu bekommen, unsere Arbeit im Rahmen der Konferenz zu präsentieren.

Danken möchte ich auch meiner Freundin und meiner Familie, die mich in stressigen Zeiten unterstützt und ermutigt haben, dafür gesorgt haben, dass ich das Begonnene zu Ende bringe und es mir ermöglicht haben, mir die Zeit zu nehmen die notwendig war.

Acknowledgements

I want to thank Bernhard Kerbl very much for supporting me through the way longer than expected journey of writing this bachelor thesis. Without him pushing me, I would not have been able to also submit a poster containing the principal part of this thesis to the High-Performance Graphics conference 2020, have it pass all checks and get the opportunity to present our work as part of the conference.

I also want to thank my girlfriend and my family for supporting and encouraging me through stressful times, making sure that I finish what I started and enabling me to take the time I needed.

Kurzfassung

Das Ändern der Vertex-Anzahl einer gegebenen Geometrie durch Hardware-Tessellation auf der GPU ist durch heutige Standards begrenzt. Die begrenzten Edge-Splits (64 Unterteilungen pro Kante) sowie die schlechter werdende Performance mit tieferen Unterteilungsebenen lassen Raum für Verbesserungen. Heutzutage bieten experimentelle softwarebasierte Lösungen, welche GPU-Shader verwenden, viel mehr Flexibilität und Funktionen. Eine mögliche Lösung, auf die wir uns konzentrieren werden, wurde 2018 von Jad Khoury [KDR18] vorgestellt, die einen Tessellation-Cache auf der GPU implementiert. Dies ermöglicht es während des Tessellationsschrittes, nicht nur die Daten des vorherigen Frames wiederzuverwenden, sondern ihn auch adaptiv zu machen, indem ausschließlich die Änderungen seit dem letzten Frame berechnet werden müssen. Der adaptive Cache verbessert die Performance auf Kosten des Speichers auf der GPU, aber die spezielle Funktionsweise verlangsamt den Ablauf in tieferen Unterteilungsebenen aufgrund der rekursiven Natur des Algorithmus. Unsere Arbeit ersetzt den rekursiven Algorithmus durch eine zeitkonstante Lösung, indem wir die Gitterstruktur der tessellierten Geometrie ausnutzen.

Abstract

Changing the vertex count of a given geometry through hardware tessellation on the GPU is limited by today's standards. The capped edge splits (64 splits per edge) as well as increasingly worse performance with deeper levels certainly leaves room for improvement. So in the meantime, software-based solutions using GPU shaders provide much more flexibility as well as features. One possible solution, which we will be focusing on, was presented by Jad Khoury [KDR18] in 2018 which implements a tessellation cache on the GPU. This enables the tessellation step to not only reuse the data of the previous frame but also makes it adaptive by only having to calculate the changes since the last frame. The adaptive cache improves tessellation performance at the cost of memory on the GPU but their particular implementation still slows down on deeper tessellation levels because of the recursive nature of their algorithm. Our work replaces their recursive algorithm with a constant-time solution by exploiting the grid structure of their tessellated geometry.

Contents

CHAPTER 1

Introduction

Triangles play an important role in computer graphics as they are widely used to visualize objects digitally. By connecting many triangles to form some shape, a so called mesh (in this case a triangle mesh) is built, often also referred to as topology. The more triangles a mesh contains, the more fine grained the control over its shape resulting in more detail and thus higher quality. Having too many triangles to render (to display) slows down performance which often results in choppy visuals or long wait times for still images. Avoiding these problems by reducing the number of triangles beforehand is a common solution but if low-poly meshes (objects with only a few triangles) aren't acceptable it becomes necessary to artificially increase the triangle count again later on. That generally describes a technique known as tessellation. Many tessellation algorithms exist which can be loosely grouped into ones that use hardware-based tessellation, a specific program (called shader) provided by all modern GPUs (graphics cards), and software-based tessellation which uses manually implemented general purpose shaders.

In this paper we provide a drop-in replacement for a paper by Jad Khoury et al. [KDR18] to increase their algorithm's performance and correct the false claim that their solution's performance doesn't slow down with deeper subdivision levels. We start with an explanation of what tessellation is and provide an outline on the state of tessellation in recent years with a focus on terrain rendering.

1.1 Motivation

The gaming industry and others push the upper limits of what is considered high quality each year and a key factor for that, next to cutting-edge GPUs, is handling huge amounts of data in a smart way to improve performance. A large part of that is handling geometry using tessellation as it can be used to increase the level of detail, reduce video memory usage and speed up mesh data transfers to the GPU thanks to manipulating geometry on-the-fly inside shaders.

Tessellation has become a hot topic in computer graphics over the years with new research pushing the limits every year. Fast progress in recent years and the resulting versatility of software-based solutions is putting the widespread use of hardware tessellation into question. Currently, hardware-based tessellation, meaning subdividing geometry on the GPU using dedicated shader pipeline stages, is limiting. Due to that the alternative, software-based tessellation, meaning subdividing geometry on the GPU using shader pipeline stages for general purpose computations, has received a lot of attention. With the introduction of the task/mesh shader pipeline, a replacement for the restrictive tessellation shader pipeline, developers gained a lot more needed flexibility. This new shader pipeline brings even more possibilities to software-based tessellation techniques.

With recent developments to overcome the limitations of hardware-based tessellation, much higher levels of detail can be accomplished using software-based tessellation. Figure 1.1 shows a terrain that uses only a square (composed of two right triangles) as its base geometry and is then tessellated to very high subdivision levels, resulting in 8,625,195 triangles, using a software-based tessellation technique.

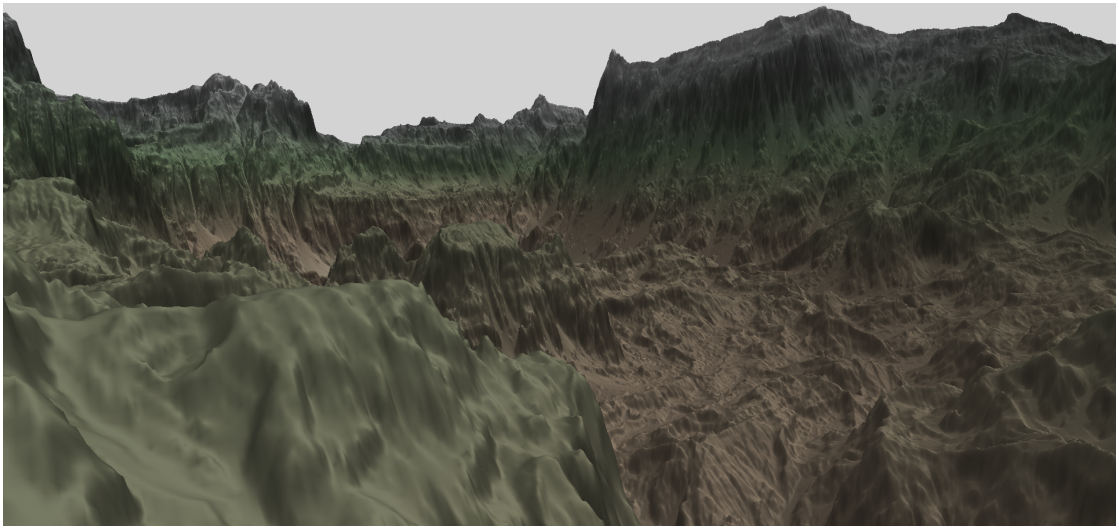

Figure 1.1: The input geometry for this mountain range consists only of two right triangles that build a flat square which is then tessellated into 8,625,195 sub-triangles.

1.2 Tessellation

Tessellation in computer graphics describes the process of automatically subdividing the triangles that comprise the geometry of an object, usually into smaller triangles, to increase its detail. This is desirable since in most cases the input geometry is simplified to reduce storage and bandwidth requirements. Tessellation is then used during run-time each frame, for each mesh that requires it, to increase the number of triangles again which are then used to restore the original quality as best as possible. The data required

to restore the original quality, by manipulating the generated sub-triangles, comes from other, cheaper resources like height maps or displacement maps, often in the form of a texture. This can be seen in Figure 1.2, the left side showing the simplified geometry and the generated high detail geometry, manipulated by a height map, on the right. It is also possible to define criteria that indicate how detailed the tessellation should be, for example the distance between geometry and camera. The result is a scene that uses little geometric detail in the far distance as it would not be visible anyway and feature very high detail on objects close to the camera.

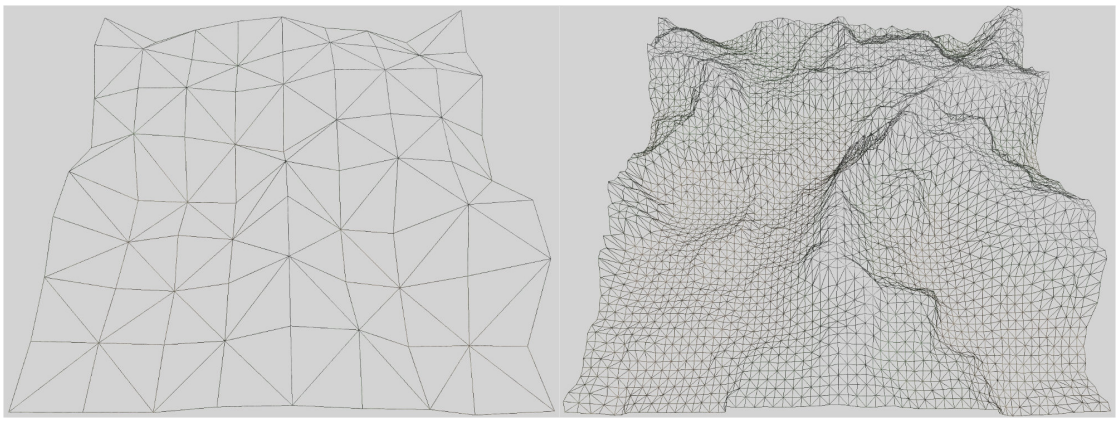

Figure 1.2: Tessellation increases the number of triangles from the geometry on the left to improve the level of detail as can be seen on the right.

1.3 Use cases

Tessellation is useful in many scenarios that contain geometry for example to improve visual quality through dynamic fine-granular levels of detail that are part of the rendering pipeline. These LODs (levels of detail) are generated directly on the GPU which saves otherwise required CPU-GPU transfer bandwidth resulting in better performance. Another benefit of on-chip LOD generation is that less geometry has to be stored on the GPU reducing the required GPU memory. Basically, the more geometry is present in a scene, the more an application can benefit from using tessellation. Use cases range from artistic workflows to high quality offline rendering to real-time applications like games but also any other interactive tool. An artistic workflow could be greatly improved by being able to work on low-poly geometry and not having to wait long to see changes in the high-quality tessellated object which makes it easier to stay in the flow. Additionally, some tessellation techniques allow changing an input mesh according to parameters that influence the style of the output. One major use case we want to focus on is terrain rendering, concerning any application that tries to display some kind of landscape, usually large ones like mountain ranges or even entire planets. While we put focus on terrain rendering in our work, the techniques presented in Chapter 3 are applicable to any triangle-based geometry.

1.4 Hardware tessellation

Hardware tessellation is supported by all major graphics card vendors by default nowadays. To utilise it, two dedicated stages are available in the shader pipeline that can be programmed to some extent. The benefit of having hardware support for subdividing triangles can be summed up to being reasonably fast as well as easy to use. By today's standards, hardware tessellation on its own is limited. For one, there is a hard limit on the maximum subdivision depth of 64 splits per edge. This may seem a lot at first glance though it may be limiting when large geometry is required, for example in application like Google Maps or open-world games with vast landscapes. This may be circumvented by dividing the large geometry into separate pieces in advance and applying tessellation separately on each piece but that opens the door for problems like alignment issues where the pieces touch each other. A common approach to circumvent the maximum subdivision depth is to additionally subdivide the separate pieces before passing them to the tessellation stage as shown by Yusov et al. [YS11]. Another drawback is the increased time it takes to compute subdivided triangles at higher tessellation levels.

1.5 Software tessellation

Besides hardware tessellation it is of course possible to implement tessellation only in software using more generalized graphics cards features like compute shaders. Alternatively, the geometry shader as well as the task/mesh shader pipeline are also suitable to implement software-based tessellation manually. The benefit over not using the tessellation shader (the hardware way) is increased flexibility at the cost of more implementation complexity and depends drastically on the available graphics card. This allows for techniques that far exceed 64 edge splits and also enables the use of caching techniques that prevent the need to do the whole tessellation procedure from scratch each frame.

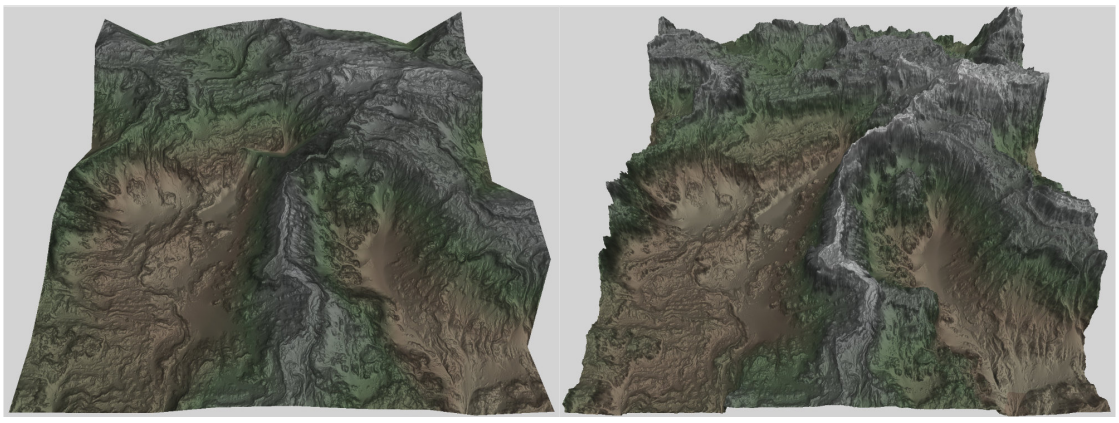

Figure 1.3: Comparison of a low-poly (left) and high-poly (right) tessellated landscape.

An example of this can be seen in Figure 1.3 where the landscape on the left, set to

subdivision level 6 (that's 64 edge splits), is much less detailed than the one on the right which is set to level 20 (that's 1,048,576 edge splits). The landscape on the left is at the limit of what hardware tessellation on its own can do. In both figures the landscape uses only a flat square consisting of two right triangles as input geometry. A wire-frame version can be seen in Figure 1.2.

CHAPTER 2

Related Work

Real-time terrain rendering has been a hot topic for many years and lots of its optimizations can be applied outside the field of terrain meshes. In its basic form, it's about taking a small part out of a larger mesh and blowing up its triangle count at the right pipeline step. Or in other words, it's about processing as little initial data as possible while maintaining high visual quality. Many topics play a part in this, for instance intelligently picking which portion of a mesh needs to be considered, or reducing storage and bandwidth requirements by bringing down the vertex count beforehand. Further examples encompass dynamically increasing the vertex count for selected steps as well as storing previously computed data where its needed later. We will take a look at papers about these fields released in the last few years and discuss their applicability in real-time terrain rendering focusing on tessellation techniques as well as subdivision.

2.1 Subdivision & Simplification

Subdivision is the process of splitting mesh faces like triangles into smaller mesh faces to provide more vertices to represent or manipulate the shape of a mesh. How and where to place these new vertices, and maybe re-position or delete the existing ones, is one of the key elements of a subdivision algorithm. Two popular and established approaches to do this are the Catmull-Clark subdivision method [Sta98] originally conceived in 1978 and the Loop subdivision method [Loo87] from 1987. Those two became the basis of many more advanced methods over the years.

A more recent approach uses standard linear methods adapted to Möbius transformations to subdivide meshes [VMW18]. In principle, the algorithm looks at each vertex separately, applies a linear subdivision method on it and it's surrounding connected vertices and then blends the result onto the original mesh using Möbius transformations. This approach makes it possible to use any linear subdivision method which in turn changes the subdivision pattern. As mentioned in their paper, turning every mesh more and

more into a sphere is not always desirable, like in the case of terrain tessellation where mountains would lose all their detail and be smoothed out into hills.

The shape subdivided faces take is often influenced by simple criteria like splitting edges into a fixed number of parts. But these shapes are at best loosely correlated to what the mesh is trying to mimic. That is favorable when modelling artificial objects but not for most natural objects. In cases like these, voronoi cells can be used to generate natural looking patterns as has been done many times when dealing only with 2-dimensional spaces like textures. Zayer et al. [ZMSS18] managed to apply the concept to 3D meshes. By regarding the cell separating lines as areas instead and allowing them to have some thickness even in the underlying calculations the complexity can be reduced significantly and parallelization on the GPU becomes efficient. Their optimizations result in computation times within a few milliseconds making the whole process useful for real-time applications. While this paper is not specifically on subdivision, their method produces realistic looking cell structures that could be applied on top of a coarse terrain mesh, for instance to render a dry desert floor.

As the above example might suggest, having more control over the shape and alignment of the subdivided faces of a mesh is favorable in some cases, mostly artistic ones. So having more knobs and dials, like being able to adjust the covariance mesh of a model [PBW19], to tune the shape of the resulting mesh opens up many possibilities as can be seen in Figure 2.1. The downside of this solution is having to compute the covariance mesh of a base mesh which takes much time and requires a lot of memory depending on the number of vertices in the base mesh. Of course, this covariance mesh has to be rebuilt each time the base mesh changes which, considering the processing time requires multiple seconds, may interrupt the creative process of the artist.

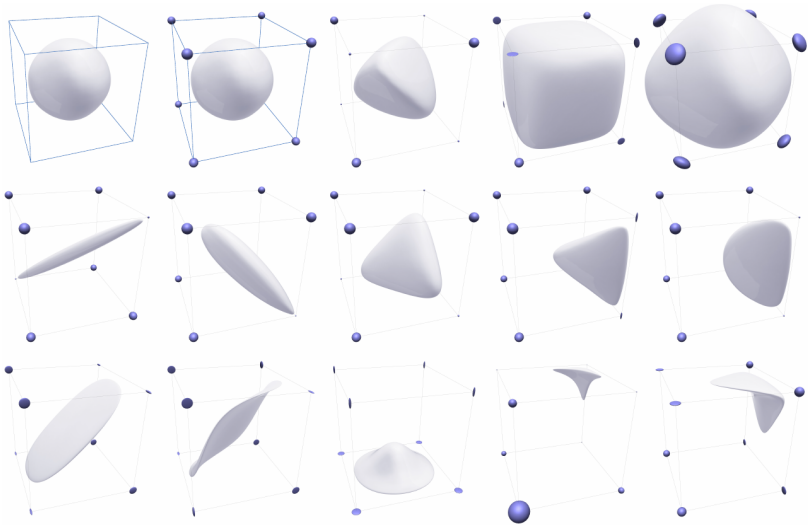

Figure 2.1: Different shapes generated from the same base mesh using only different parameters for the covariance mesh [PBW19]

Traditionally, subdivision operations are carried out on the CPU but utilizing the much faster GPU for such tasks yields more performance and/or higher quality results as a new formulation of the Catmull-Clark subdivision in linear algebra shows [MWS+20]. Basically, the whole mesh is represented as various matrices that contain all of the vertices, edges and faces including their adjacency information which are then processed using linear algebra kernels and can therefore be run in parallel on the GPU. The main drawback is the huge memory consumption which increases further the more detailed a mesh is.

Neural networks are everywhere nowadays which means they are also used in subdivision [LKC+20]. The specific task the neural network takes on in this algorithm is deciding where to place the new vertices, everything else is basically handled by a normal loop subdivision. Depending on the mesh(es) the network is trained on the style of the resulting subdivided mesh is influenced as the network learns and then reproduces it. For example, to emphasize or keep sharp edges it is best to train the neural network using meshes with lots of sharp edges. This could be particularly interesting for artists to quickly iterate over various styles for inspiration. Important to know is the fact that the network can be trained unsupervised which means no manual interaction is required so using it is less complicated.

The inverse operation of subdivision is called simplification and is designed to reduce the number of vertices of a mesh in a way that still preserves the general shape. Simplification is relevant for tessellation because it is usually applied to relatively coarse geometry that is later subdivided to a desired level of detail. Being able to automatically generate a decent coarse mesh from its detailed base is key to a fast and convenient workflow that doesn't inhibit artists. One such simplification algorithm is a combination of two conventional operations that are reversed [SAX+17]. On one hand, the method requires multiple seconds for objects with a lot of detail and is therefore not usable for real-time application but on the other hand that is usually not necessary as simplification is purely used as a pre-processing step.

Simplification can pursue various goals and while we focus mostly on rendering related topics we also include an example that prioritizes not visual quality but property retainement. The technique [LLT+20] combines an edge-collapse algorithm with a cost metric that minimizes the difference of the Laplacian of the original and the simplified mesh. This iterative process utilizes a priority queue with the cost metric as it's weight and stops when the mesh reaches the intended resolution.

2.2 Smoothing

Improving the quality of an original mesh and making sure that a simplified mesh has the best possible quality is important as the resulting tessellated mesh's quality depends on the quality of all earlier stages. In general, the higher the quality of the base mesh and the better the simplified mesh represents the base mesh, the higher the quality of the tessellated mesh.

One method to improve the quality of a mesh is smoothing it, or in other words, making it appear less pointy. A way to do this is to re-position all the vertices in a way that minimizes the aspect ratio of each vertex [HX17]. The aspect ratio of a vertex can be calculated by dividing the length of the longest edge connected to the vertex through the length of the shortest edge. The result is a clean-looking mesh that closely resembles the original shape and still has the same number of vertices, as can be seen in their paper. Even though this method only works on local criteria and is fast to compute, it is not considerably useful in real-time applications as it is best used as a pre-processing step where required time is usually not a limiting factor.

The following method uses the inner and outer angles between the connecting edges of a mesh to reduce the amount of small angles [HCGL20]. This results in a smoother mesh that still retains corners where they should be but most importantly, the topological mesh looks much more clean. Particularly interesting is the fact that the algorithm works on each element separately even though that is usually avoided because it introduces visual artifacts in most cases, though not here. Figure 2.2 shows a single triangle (left) being transformed into a more regular shape (right) with reduced small angles. Since it processes each element separately, the whole smoothing process could possibly be executed in parallel. As with most smoothing algorithms, this one is also intended as a pre-processing step so time measurements are not provided. That said, the nature of the algorithm working on each element separately infers its linear time complexity.

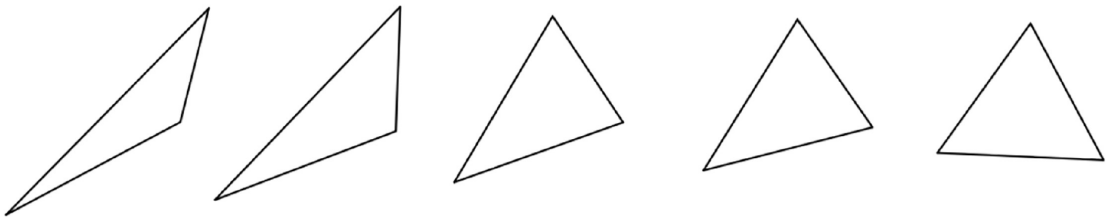

Figure 2.2: A triangle in different stages of transformation, reducing small angles each stage going from left to right [HCGL20]

Reducing the irregularity of a mesh's topology is important for making it look smoother and improve its quality. To do this, one could look at each vertex and its connected triangles and improve the quality of each connected triangle separately. As a consequence, there are now multiple possible positions for each vertex that need to be merged to result in a coherent mesh. Weighting the possible positions by the size and quality of the improved connected triangles and also taking the original into account increases the average mesh quality while also being very adaptable as this paper shows [HGCH21]. The result is a cleaner and smoother-looking topology that still has the same number of vertices and edges as can be seen in Figure 2.3. It should also be quite fast as it runs in linear time $\mathcal{O}(n)$ since it processes each vertex once though the paper does not provide time measurements.

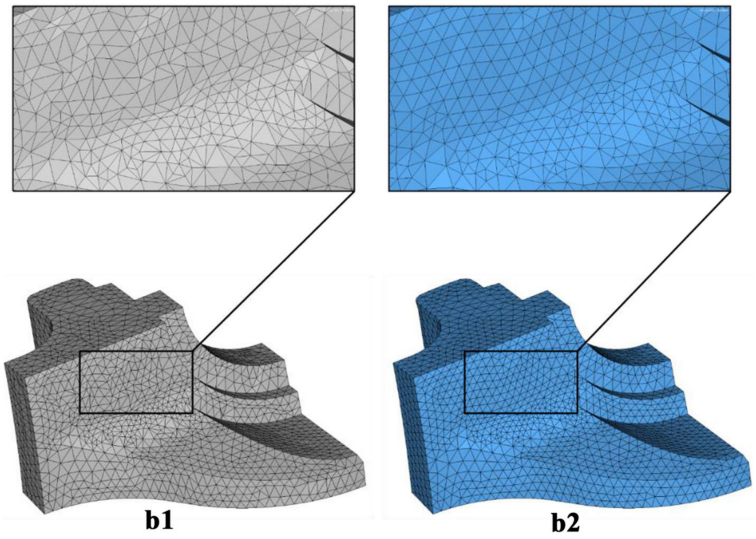

Figure 2.3: A mesh before (b1) and after (b2) smoothing it by moving only the existing vertices [HGCH21]

2.3 Tessellation techniques

Tessellation is a culmination of the previous techniques and usually also includes a dynamic component that decides the level of detail for each mesh that should be tessellated. The smoothing and simplification can be used to reduce the high quality objects that artists create into coarse, meaning low-quality, objects. These require a lot less space to store and will later be processed using subdivision techniques to restore them to their original quality (or better) as close as possible.

One method that builds upon Catmull-Clark subdivision is this work from Nießner et al. [NLMD12] about adaptively subdividing a mesh specifically near its features. This technique not only works with triangle meshes but also quadrilateral meshes. A feature can be any part of a mesh that significantly contributes to its defining looks, often called creases. They can be imagined as intentionally sharp edges, for example pointy animal ears or the structural support beams of a bridge. Basically anything that should not be smoothed out during subdivision. The general idea of this technique is to only subdivide the parts of a mesh that contain a significant amount of detail as can be seen in Figure 2.4. This results in fast, real-time capable tessellation.

Using Catmull-Clark subdivision in combination with hardware tessellation and displacement mapping has been proven to be efficient thanks to another paper by Nießner et al. [NL13]. A problem with displacement maps, which enhance the surface detail of meshes during rendering, is that the displaced geometry has no normals on its own. A common solution is to use normal maps that provide the missing normal information but these usually need to be pregenerated and introduce issues of their own. Nießner et al. use an analytic displacement function based on bi-quadratic B-splines to directly infer the

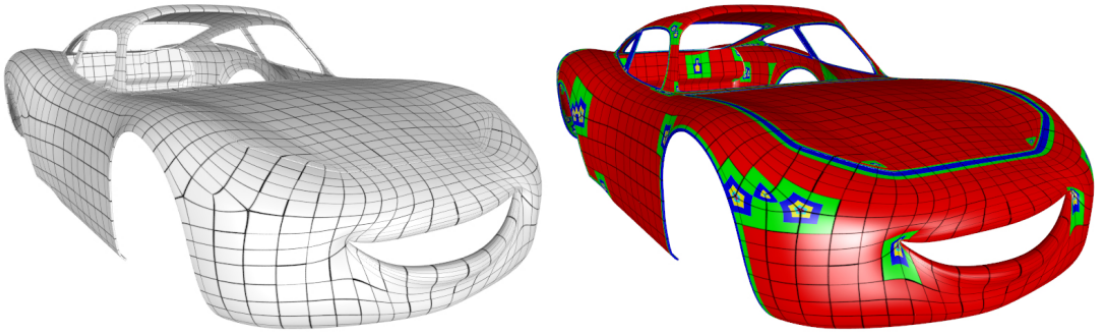

Figure 2.4: Fast subdivision using feature adaptive rendering. The left side shows the input mesh while the right side shows the subdivided areas with increasing subdivision levels near sharp edges (green and blue) [NLMD12]

normal information without the need for normal maps. Hardware tessellation comes into play to generate renderable primitives out of the bi-quadratic B-splines. Additionally, dynamic level of detail is possible thanks to per vertex tessellation factors calculated based on classic criteria like camera distance or screen space edge length. Their solution also enables artists to directly change the displacement data, which means being able to work on the visual end-result, instead of working on the base mesh and having to wait for subdivision to see the actual results.

When handling terrain meshes, which are usually gigantic, it is vital to use some kind of tessellation. One older approach is to prepare levels of detail for the height map [YS11]. Terrain rendering usually makes use of a height map to store the vertical component while the x and y coordinates can be computed as needed. Exploiting this fact, the method prepares LODs for the height map to reduce the needed GPU memory and bandwidth. Adaptive rendering is then supported by storing the currently used LOD blocks, which are further broken down into patches, and storing them as a quad tree in GPU memory. For every block an error is computed that represents the divergence of its simplified points from the height map's values. Each frame, the cached patches errors are checked to see if a different LOD needs to be used for any part of the displayed terrain. When rendering a patch, the triangulation resolution can be further decreased by choosing triangulation patterns with larger triangles that consider less height values accordingly. To not introduce visual artifacts, the relevant patches' errors are compared to the targeted edge length during rendering. Targeted edge length refers to how many pixels any single edge is allowed to occupy on the finished image. The edges neighbouring each patch are also considered to calculate the correct tessellation factors. Tessellation itself is then done via the standard GPU-based tessellation pipeline.

Another approach is shown in this paper from 2015 [KJCH15] which uses an approach that does some pre-processing on the CPU and then passes only the relevant data to the GPU for refinement. On the CPU the large terrain mesh is processed into a quadtree which is then used to pass only the required quadtree tiles with varying tessellation

factors to the GPU. This means that each tile can have a different level of detail to save performance. While the classic approach for terrain data uses a height map to store the vertical displacement information here geometry images are used instead to improve the triangle-count distribution further.

Varying the level of detail of a mesh according to common metrics like camera distance reduces the processed vertex count considerably. Metrics like these only take parameters of the digital world into account so Tiwary et al. [TRK20] proposed to take the way human perception works into account. In their method the process of tessellating a mesh is simplified by exploiting the sporadic movement when the point focused on changes and the small area where we perceive the most detail. Concentrating the highest level of detail on that small area and only updating it in the interval of the sporadic eye movements while reducing the LOD the further away a vertex is from the target area results in a great reduction of rendered triangles and therefore yields better performance. This method might be useful in more cases than stated in their paper, for example not only when the gaze location is known but in all cases where the camera is controlled by a user. Specifically, by applying the mentioned tessellation criteria during camera movement that exceeds a certain threshold and choosing the center of the screen as the target area. Being able to reduce the area where high level of detail is required through a scene-independent criteria is perfect for terrain rendering where wide landscapes span further than the horizon. Since this is basically only a LOD selection criteria it can be implemented anywhere regardless of whether hardware or software tessellation is used.

Incorporating information gained during the simplification step into the subdivision procedure allows for even more control during the tessellation phase. This was achieved by Yuan et al. in 2016 [YWH+16]. The technique establishes a two-step procedure to first generate a coarse mesh from the original using an inverse subdivision approach, send the coarse mesh to the GPU and then have the tessellation stage of the pipeline generate a detailed mesh with only little differences to the original. The interesting part is the generation of the coarse mesh where they focus on only simplifying triangles with low impact on shape and detail. In addition, the vertices of the simplified mesh are then re-positioned in a way that supports better re-construction by the tessellation stage by simulating it during the error calculation of the simplification step. The subdivision step is then implemented via the standard hardware tessellation stage.

Another method that incorporates both the simplification and subdivision steps in one technique was presented by Thibaud Lambert et al. [LBG18]. To increase the accuracy of the reconstructed mesh, they modified the edge collapse step to consider the deviations from the original mesh of both the vertex positions and texture coordinates for their approximate error metric. This metric is used to simplify a complex mesh and dynamically select the most fitting level of detail for each patch depending on the camera view. Their approximate error metric expresses the highest difference of all vertex positions on the simplified mesh to the original mesh for a given view distance and direction. During the tessellation control shader stage on the GPU the LOD with the smallest error for a patch is dynamically chosen for the given view direction.

Although the last two methods are working as intended for most meshes, terrain meshes usually don't start off as high-detail 3d models except maybe some real-life height maps or scans. On one hand, having no high quality base mesh to calculate the error metric renders the solutions impractical for most terrain rendering applications. On the other hand, this would make it possible to pre-generate terrain meshes with extreme detail to use as the base for the pre-computation of the coarse meshes.

2.4 Caching & more

When tessellating geometry on the GPU, the whole process is started from scratch each frame which wastes a lot of performance. Caching can be implemented to make use of the tessellated geometry calculated in the previous frames, trading performance for memory consumption. The memory required to temporarily store the tessellated geometry can be very high since tessellation turns a few triangles into orders of magnitude more. This makes it necessary to use some kind of compression or packing algorithm to make the whole trade-off actually worth it. Additionally, a data structure that is fast to read and write is required to manipulate the compressed data efficiently.

In the context of terrain data it is possible to get rid of some of the mesh information when storing it and add it later during rendering because it is inherent to the nature of height maps. One approach that makes use of this fact is to pack all the data to render a triangle into a single vertex and then storing these new vertices in a quad-tree as shown by Lee et al. [LS19]. The compressed vertex only contains 6 float values (x, y, z, r, g, b) so the uv coordinates and height value have to be calculated for each decompressed vertex accordingly. The quad-tree is re-used each frame to refine it to the desired level of detail, subdividing or simplifying one step up or down as required by the LOD metric. As is the nature of a quad-tree, subdividing once turns one triangle into 4 triangles and simplifying turns 4 triangles into one triangle. During rendering, the quad-tree is read in the geometry shader to decompress the data into a triangle strip. If the level of detail does not match the required LOD, the compressed vertices are split/merged once and then decompressed. The quad-tree is implemented as a double buffer to minimize popping and artifacts but during complete view changes the buffer is rebuilt from scratch. To prevent cracks, they recursively split the adjacent triangle of a neighbouring patch that has a lower level of detail into smaller triangles selectively.

While not specifically the topic of this paper, we want to quickly mention this work from Matthias Englert about using mesh shaders for terrain rendering [Eng20]. A lot of performance can be gained by using mesh shaders to handle processing heavy tasks on the GPU instead of the CPU. This is accomplished by generating render batches only for parts of the terrain that are relevant for the current camera position and view-angle. These render batches each contain a part of the relevant terrain data in a compressed format and are then processed on the GPU in parallel. The GPU is then able to triangulate the compressed terrain data, also in parallel, which is often done on the CPU. The difference is that a lot less data has to be transferred to the GPU if the GPU itself handles the

triangulation. Additionally, the generated triangles are further tessellated using a mesh shader implementation of a simplistic subdivision scheme.

2.5 Reference approach paper

Continuing the topic of caching we are going to take a detailed look at *Adaptive GPU Tessellation with Compute Shaders* by Khoury et al. [KDR18] which our work is based on. Their goal is to improve upon the drawbacks of the standardized hardware tessellation by allowing more levels of subdivision and removing the increasing performance cost of deeper subdivision levels. The concept makes use of a very simple subdivision scheme that halves a triangle into two equally sized sub-triangles. Each triangle in any subdivision level is then uniquely represented by a single integer they call a key. These keys are stored in a buffer on the GPU and form the tessellation cache that gets to be re-used each frame. As the keyword "adaptive" suggests, this cache allows to continue the tessellation progress where it was left last frame and subdivide or simplify the topology from there. Their implementation actually only carries out one subdivision or simplification per frame, depending on whether the desired level of detail is reached or not which implies slight visual artifacts when the buffer is built from scratch.

2.5.1 Basic concept

The backbone of the concept is the cache which has to allow fast read and writes in a parallel fashion. To allow reading and writing the cache at the same time without interference it is implemented as a double buffer, reading from the first, writing into the second and swapping the two afterwards. This does double memory consumption but since the buffer only stores integers it is quite compact anyway. The buffer itself is implemented as a simple array as reads and writes are fast and the whole array has to be iterated over each time to reconstruct the actual triangles. The fact that the array is implemented as a double buffer including that every key has to be processed allows for an implicit clean-up mechanic, by just emptying the second (write) buffer before writing into it, so there are no unused or duplicate keys cached. Reconstructing the actual triangle data from the keys is possible because they encode barycentric coordinates. This also means that the last step to actually generate a triangle that can be rendered is a barycentric interpolation that uses the decoded barycentric coordinates and the real coordinates of the coarse base triangle.

2.5.2 Barycentric transformation matrix

The transformation matrix used by Khoury et al. [KDR18] is defined as

$$M = \begin{pmatrix} s & -0.5 & 0 \\ -0.5 & -s & 0 \\ +0.5 & +0.5 & s \end{pmatrix} \tag{2.1}$$

It describes how to move the vertices of any triangle to create a new triangle that is exactly half the size of the original one. It works for any triangle because it transforms the barycentric coordinates instead of actual position data. There is exactly one input parameter `s` to tell the transformation matrix to create one or the other half-sized triangle as can be seen in Figure 2.5. This transformation can be applied multiple times to subdivide the triangle into smaller and smaller sub-triangles, or in other words to increase the level of detail. This effectively implements a subdivision technique known as longest edge bisection (LEB). LEB, as the name implies, splits a triangle exactly in half by injecting a new edge between the midpoint of its longest edge and the opposite corner.

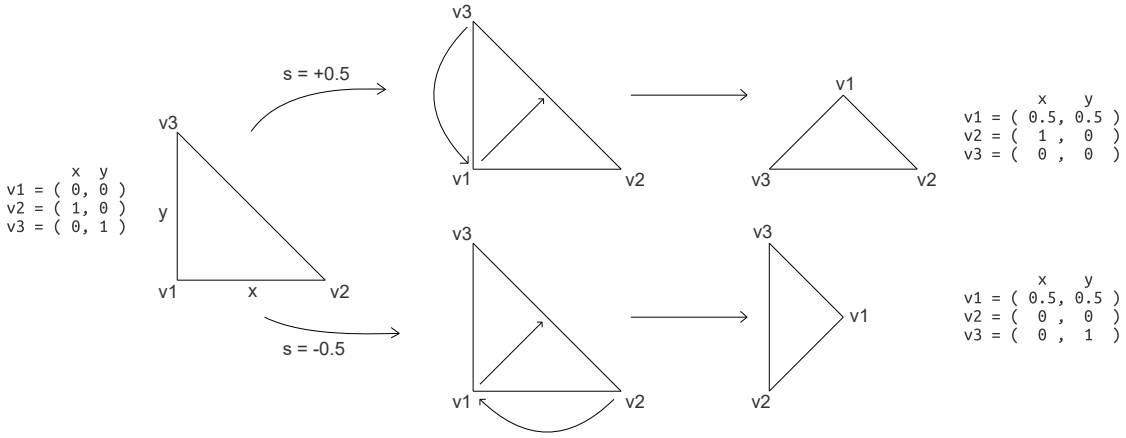

Figure 2.5: The barycentric transformation matrix `M` shown in Section 2.5.2 moves the vertices to create either one of the subdivided triangles. The parameter **s** determines which half is created.

2.5.3 Cache key encoding and decoding

Cache keys are represented by 32-bit integers that, when viewed in binary form, consist of a series of zeroes and ones. These digits represent the decision to subdivide into one or the other triangle-half and when used in series as input parameters to the barycentric transformation matrix result in the exact barycentric coordinates that create the desired sub-triangle when resolved. Encoding a triangle is just a matter of remembering the exact series of subdivisions to get to the desired sub-triangle as is shown in Figure 2.6. A simple bit-shift operation during each subdivision does the trick as the process initially always begins at the coarse base triangle. To keep track of the level of detail of an encoded triangle, the most significant bit is set to one and is automatically moved due to the bit-shift operation. Considering the implementation shown in Listing 1.2 of their paper, it is obvious that longer keys, in other words deeper subdivision levels, require more matrix multiplications when they are being decoded and thus slightly more time to compute than shorter keys due to the algorithms recursive nature.

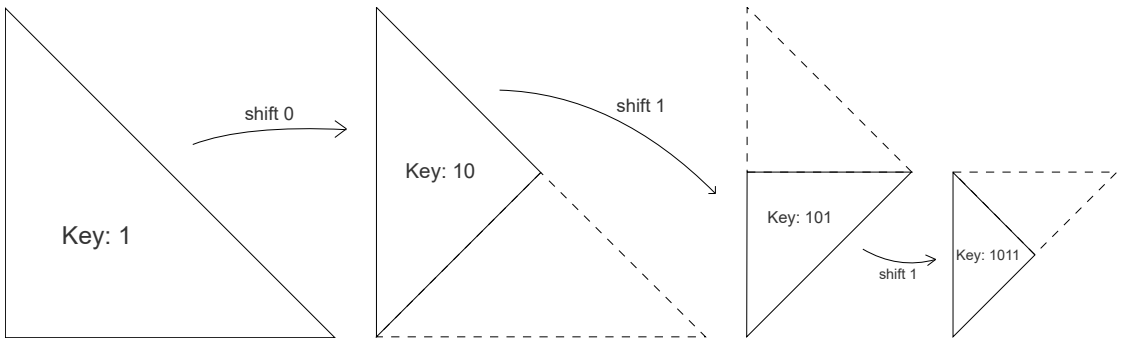

Figure 2.6: Leftmost is the coarse base triangle with its key consisting only of the most significant bit. Then either one or zero is shifted into the key to go down the subdivision levels.

2.5.4 Finding the parent and child of a triangle

Going up and down the subdivision levels is a simple process considering how the triangles are encoded. There are some noteworthy precautions shown that prevent duplicates and other unwanted side effects that we are going to cover. To go down a subdivision level, one triangle is split into two sub-triangles that replace the parent triangle. The keys for both sub-triangles can easily be generated by shifting a zero to the parents key to get one half and a one for the other half. Both keys are then stored in the cache while the parents key is removed, or in case of the double buffer implementation it is just not stored again. See Figures 2.5 and 2.6 to get a better understanding of the process. Going up a subdivision level is a little trickier as to not produce multiple identical parent triangles. To circumvent this problem, only one of the child triangles actually generates the parent, in their implementation the child with the zero digit. Getting the actual key is as simple as unshifting the last bit. Due to the double buffer implementation only the new parent key (triangle) is stored which automatically removes the child keys (triangles) from the cache.

2.5.5 Claims and reality

According to their paper, this tessellation strategy solves two major limitations of standard hardware tessellation. For one, having only 6 subdivision levels available and for another requiring increasingly more time with deeper subdivision levels. The first limitation is easily lifted by their encoding scheme which is in principle capable of unlimited subdivision levels and only limited by implementation details and hardware support for integers with more bits. They also mention this in Section 1.2.2 of their paper. The second limitation is where their algorithm is flawed. That's because the simplicity of manipulating a key solely through single bit-shifts is also a major disadvantage to the algorithm: The subdivision depth dictates the number of barycentric transformation matrix multiplications which shows the recursive nature and therefore linear time complexity of the decoding step.

CHAPTER 3

Improved triangle encoding

The shortcoming of Khoury et al.'s solution with regards to the recursive nature of the algorithm somewhat dulls its potential. We propose a solution that improves upon their algorithm by replacing the scheme used to encode and decode the keys with one that runs in constant time. In theory, this should increase performance for scenes with higher levels of detail when compared to the original implementation. In other words, decoding a key should be much faster considering the context of terrain rendering where many different levels of detail come into play. The likely drawback of a constant-time implementation is that it requires more time to generate a sub- or parent-triangle than it did in the original. However, this downside can be safely ignored if reading (decoding a key) happens more often than writing (encoding a triangle) in most scenarios. That is because every key in the cache has to be decoded once every frame but encoding only happens if the target level of detail for the triangle changed, otherwise the key is just stored as is again. So in scenes where the tessellated object doesn't move, which is most likely all of them in the context of terrain rendering, the only other option that influences the criteria is the camera. There are two relevant states for a camera in this context, moving and not moving. A camera that doesn't move doesn't trigger a change in the target level of detail so there is no problem in that case. A camera that does move will cause the target level of detail to change so triangles will be encoded but the amount depends on the speed of the camera's movement.

3.1 The grid

Due to the longest edge bisection used by the reference approach paper, the subdivision gradually generates a repeating grid pattern as can easily be seen in Figure 3.1. The grid's cell size decreases with every second subdivision level as the odd levels only add diagonal lines. We are working with the triangles in barycentric space, which is 3-dimensional, but the grid represents a 2-dimensional space. So in order to get to

2-dimensional space we drop the 3rd coordinate of barycentric space since we don't need it. Having a coordinate system at hand means that we are able to represent each point inside it with a two-dimensional location. This allows us to ditch the whole recursive transformation matrix approach that relied on the preceding transformations to find the correct coordinates. If we have the `x` and `y` coordinates of a triangle available, it should be possible to directly calculate (without preceding transformations) the barycentric coordinates of the sub-triangle.

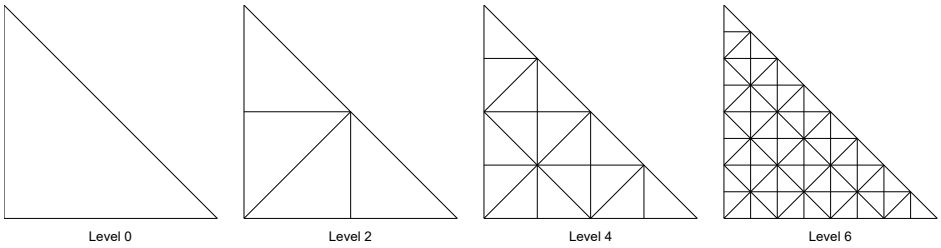

Figure 3.1: Subdivision levels viewed in barycentric space. Notice the emerging grid pattern when going down the subdivision levels. The grid's size decreases with every second subdivision level which is why the odd ones aren't shown.

There is an issue that comes with the differing levels of detail though and it has to do with the various grid resolutions. Even if we can describe a location inside the grid using `x` and `y`, we wouldn't know which grid size these coordinates are related to. This means, we need another coordinate `z` that specifies the subdivision level and in effect the related grid size. The grid divides the barycentric space, which is a range from `0` to `1`, into equally sized squares. While `z` increases by one every subdivision level, the grid's cell size only decreases with every second subdivision level, increasing it's resolution by an order of 2 each time because of the subdivision rule. This makes the biggest possible coordinate value of each grid size always a member of order 2. Refer to Figure 3.2 for visual aid on this.

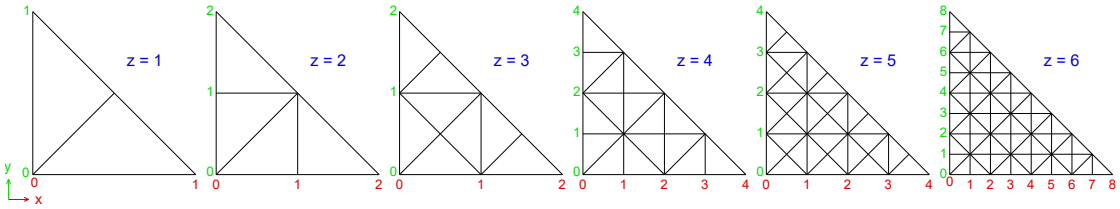

Figure 3.2: `z` denotes the subdivision level. The grid resolution is doubled every second subdivision level which is shown by the axes labels.

Now that we have all the necessary coordinates to properly navigate the grid system and point to any grid cell uniquely, another problem stands out. As each cell is a square, there is never only a single triangle inside it. So in order to know which triangle we are supposed to generate from a key we need another value, let's call it `i`, to identify

it uniquely. Having the grid change resolution every second subdivision level results in the triangle count per square to alternate between two and four each level. Additionally, there are two different possibilities how the triangles are orientated if there are only two inside a grid cell. Their orientation conveniently follows a simple rule as can be seen in Figure 3.2. Every second column, and shifted by one cell every second row, has the diagonal going from the bottom left to the upper right while every other column, also shifted by one cell every other row, has the diagonal going from the upper left to the bottom right. Take a look at Figure 3.3 for all possible triangle constellations.

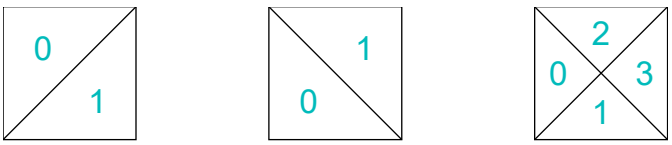

Figure 3.3: All the possible triangle orientations inside a grid's cell. The light blue numbers denote each triangle's identifier `i`.

The problem now becomes how to store the coordinates `x`, `y` and `z` and the identifier `i` inside a single integer efficiently and preferably without requiring more space than the original solution.

3.2 Key, encoding, decoding

Packing multiple values inside a single integer necessitates clever use of their binary representations, the number of required bits as well as which information we are able to imply into the schema. For example, in the original algorithm, the first bit of any key in binary representation is set to `1`, generally called the most significant bit, and signifies the subdivision level through it's index. We keep this behaviour to encode `z` (the subdivision level) inside a single bit. While every shift/unshift operation in the original schema results in going one subdivision level up or down, conveniently moving the most significant bit exactly one place, we have to make sure our key's length changes by only one by going up or down one subdivision level too or we would be wasting precious bits.

Next there are the various possible triangle constellations inside a grid cell represented by `i`. `i` has to be able to discriminate between eight different triangle orientations which would require 3 bits in binary. We can do better though because the grid behaviour explained above implies that subdivision levels with even numbers always have two triangles in a cell and levels with odd numbers always have four triangles in a cell (refer to Figure 3.2). So, in keys for even levels `i` only needs one bit to represent two states, and two bits for odd levels to represent four states since we know if the current subdivision level is even or odd through `z`.

Lastly, we have the `x` and `y` coordinates left. There are two mechanics at play here that help us minimize the necessary number of bits. First, both require the same number of bits on a given subdivision level thanks to the square grid. Second, because the grid's

cell size is quartered every second subdivision level, the highest possible value `x` and `y` have to be able to represent doubles also only every second level. So at levels two and three `x` and `y` require one bit to represent `0, 1`, at levels four and five they require two bits to represent `0, 1, 2, 3`, and so on. This means both `x` and `y`, when looked at in binary representation, change in length only every second subdivision level.

We put our key together in the same order that we mentioned the variables in: `z i x y`. It is important to note, that we manage to keep the behaviour of the most significant bit as indicator for the subdivision level thanks to the specific way our variables change in length. At level two, both `x` and `y` need one bit each and `i` also needs one bit, that's a total of three bits. At level three, `x` and `y` still need only one bit each but `i` now needs two bits to correctly describe the four possible triangle orientations in a cell, that's four in total. At level four, the grid's resolution increases so `x` and `y` now need two bits each but `i` only has to describe two possible triangle orientations now so it only needs one bit, that's a total of five bits. As one can see, our key requires one additional bit for every additional subdivision level, just as the original. Take a look at Figure 3.4 to verify this, the keys shown are each a child of the one below. They also match the grid in Figure 3.2 so you can trace them manually.

z	i	x	y		z	i	x	y
1	0	101	010		6	0	5	2
1	01	10	01		5	1	2	1
1	0	10	01		4	0	2	1

Figure 3.4: The structure of our keys in binary (left) and decimal (right). Both `x` and `y` always have the same number of bits and no upper limit. `i` is always either one or two bits long. `z` forms the most significant bit, it's index in the binary representation of the integer encodes the subdivision level.

Encoding our key is then just a matter of putting each variable at it's correct position in the binary representation of an integer. Doing so requires a series of bit-shifts, bitwise OR operations and additions, 9 in total. Our GLSL implementation of that is shown in Listing 3.1. Notice that `z` is not passed as a parameter because its value is determined by its index as the most significant bit automatically.

Decoding our key is also mostly a series of bitwise operations as can be seen in Listing 3.2. As mentioned before, we have to calculate whether `i` is one or two bits long according to the subdivision level `z` (Line 6). We also calculate, based on `z`, how many bits `x` and `y` are taking up to extract their correct values (Line 7). Notice again that `z` is not returned since the relevant information has already been extracted into `numBitsI` and `numBitsXY` which are returned instead. It is important that decoding the key can be done fast since most of the other functions have to work on the separate variables instead of the whole key.

Listing 3.1: Encoding our key requires only a series of bit operations.

```
Input: i is the triangle orientation. x, y are the barycentric grid coordinates.
       numBitsI, numBitsXY specify how many bits i, x, y actually occupy.
Output: The finished key as an unsigned integer.
uint encodeKey(
    in uint i, in uint numBitsI,
    in uint x, in uint y, in uint numBitsXY
) {
    uint numBitsXY2 = numBitsXY + numBitsXY;
    return (1u << (numBitsI + numBitsXY2)) |
           (i << numBitsXY2) |
           (x << numBitsXY) |
            y;
}
```

Listing 3.2: Decoding our key is also just a matter of bit operations.

```
Input: key is an unsigned integer using our key encoding scheme.
Output: i is the triangle orientation. x, y are the barycentric grid coordinates.
        numBitsI, numBitsXY specify how many bits i, x, y actually occupy.
void decodeKey(in uint key,
    out uint i, out uint numBitsI,
    out uint x, out uint y, out uint numBitsXY
) {
    uint z = findMSB(key);
    numBitsI = ((z + 1u) & 1u) + 1u;
    numBitsXY = (z - 1u) >> 1u;
    uint numBitsXY2 = numBitsXY + numBitsXY;
    uint iMask = ((1u << numBitsI) - 1u) << numBitsXY2;
    uint yMask = (1u << numBitsXY) - 1u;
    uint xMask = yMask << numBitsXY;
    i = (key & iMask) >> numBitsXY2;
    x = (key & xMask) >> numBitsXY;
    y = key & yMask;
}
```

3.3 Finding the children of a triangle

Encoding and decoding the key is just one part of the whole procedure, the other parts being how to calculate keys for child triangles and the parent triangle of a key. Additionally, we have to re-implement some helper functions that are required to correctly go up and down the subdivision levels. We will start with the process of descending a

subdivision level from a given key as it is a lot simpler than ascending. Take a look at Listing 3.3 for implementation details. As mentioned in Section 2.5.4, going down a level always splits the current triangle into two sub-triangles, generating two new keys in the process. To work with our keys it is always necessary to decode them first in order to have access to the separate variables.

Listing 3.3: Function to calculate the children keys of a given key.

```
Input: The parent key for which to generate children keys.
Output: An array containing the two finished children keys.
1  void childrenKeys(in uint key, out uint children[2])
2  {
3      uint i, numBitsI, x, y, numBitsXY;
4      decodeKey(key, i, numBitsI, x, y, numBitsXY);
5      uint newX = x * numBitsI;
6      uint newY = y * numBitsI;
7      numBitsI = numBitsI - 1u;
8      uint numBitsIFlipped = numBitsI ^ 1u;
9      uint k1X = newX + (numBitsI * childTable[0][i]);
10     uint k1Y = newY + (numBitsI * childTable[1][i]);
11     uint k2X = newX + (numBitsI * childTable[2][i]);
12     uint k2Y = newY + (numBitsI * childTable[3][i]);
13     uint iTableIndex = (numBitsI << 3u) | (i << 1u) |
14                        ((x + y) & 1u);
15     uint k1I = childrenKeysTableI[iTableIndex][0];
16     uint k2I = childrenKeysTableI[iTableIndex][1];
17     numBitsXY = numBitsXY + numBitsI;
18     numBitsI = numBitsIFlipped + 1u;
19     children[0] = encodeKey(k1I,numBitsI, k1X,k1Y,numBitsXY);
20     children[1] = encodeKey(k2I,numBitsI, k2X,k2Y,numBitsXY);
21 }
```

Starting with `i` there are two rules that we have to keep in mind. First, `i` alternates between one and two bits each level. And second, every other subdivision level the grid's resolution is doubled. Both rules are explained in detail in Section 3.1. Calculating the new number of bits for `i` is therefore simple. We know its value is always either `1` or `2` so considering the current value we subtract `1` to bring it into the `0,1` range (Line 7), flip the bit using a bitwise xor operation (Line 8) and finally add `1` again (Line 18) so `1` becomes `2` and `2` becomes `1`. The new actual value of `i` is trickier as it entirely depends on the current values of `i`, `x` and `y`. Conveniently, there is a repeating pattern for generating the values of `i` thanks to the grid. We are able to take advantage of that by setting up a lookup table (`childrenKeysTableI`) to avoid code branches. As a result, we need to create unique identifiers out of the current values of `i`, `x` and `y` to properly navigate the lookup table. This is in addition to mapping out all the possible results.

Let's first consider going from an even subdivision level to an odd one or in other words, from one bit to two bits for `i`. Refer to Figures 3.2 and 3.3 for visual support on this. There are two possible triangle orientations for one bit `i` values. The first one (with `0` at the upper left) appears in all grid cells that satisfy the condition `(x + y) % 2 == 0`. In these cases, `0` becomes `0` or `2` while `1` becomes `1` or `3`. This takes care of half the cells, the other half (with `0` at the lower left) satisfies the condition `(x + y) % 2 == 1` and in those cases, `0` becomes `0` or `1` while `1` becomes `2` or `3`. Going from an odd subdivision level to an even one, so from two bits to one bit, is simpler as there is just one possible triangle orientation for all cells. We are looking exclusively at the new `i` values at this point so positional cell changes are of no concern for now. So `0` becomes `0` and `0` (upper left and lower left cells), `1` becomes `1` and `0` (lower left and lower right cells), `2` becomes `1` and `0` (upper left and upper right cells) and `3` becomes `1` and `1` (upper right and lower right cells). Take a look at the full lookup table in the appendix in Listing 6.1. Due to the way the lookup table index is created (Lines 13 and 14), the values for new one bit `i` values have to occur twice.

Calculating the new `x` and `y` values is much simpler in comparison. Going from an even subdivison level to an odd one doesn't change the original values so we just keep them. Going from an odd level to an even one always doubles the grid resolution so we multiply the original `x` and `y` values by two (Lines 5 and 6). Then we need to push either `x` or `y` or both one cell further to take the previously mentioned positional cell change into account (Lines 9, 10, 11 and 12). For this we implemented another lookup table (`childTable`) that you can also find in Listing 6.1. No specially created index is required for this lookup table, only the original `i` value is used to cover all cases on even subdivision levels. On odd levels, `numBitsI` is zero at that point so the influence of `childTable` is negated. Finally, it is necessary to adjust the number of bits (`numBitsXY`) for the `x` and `y` values (Line 17). Just as with the new `x` and `y` values, no change is required when going from an even subdivision level to an odd one. Going from an odd level to an even one doubles the grid resolution so we have to add `1` to the number of bits for `x` and `y` since one additional bit doubles the representable numbers.

As we are working with a finite number of bits of an integer in the actual implementation, we need a mechanism to decide when to stop subdividing. In the original code from Khoury et al. exists a method for exactly that purpose called `isLeafKey`, shown in Listing 3.4, which we have to adjust slightly. The deepest possible subdivision level for 32bit integers using our concept is 31, one level less than the original concept allowed. More on this in Section 5.

Listing 3.4: Helper method to know when to stop subdividing.

Input: The key for which to check if it can be subdivided.
Output: True if the input key can be subdivided, false otherwise.

```
1 bool isLeafKey(in uint key)
2 {
3     return findMSB(key) == 31u;
4 }
```

3.4 Finding the parent of a triangle

Next we will explain in detail how to go up the subdivision levels, or in other words, how to merge children keys into their parent key. Just like in Section 3.3 it is important to keep in mind that the number of bits for the values change as well as that the grid's resolution changes every second subdivision level but this time it is halved instead of doubled. The complete function is shown in Listing 3.5.

Starting with the number of bits for `i` again, we use the same procedure as in Section 3.3 since it alternates on every level change (Lines 5, 6 and 10). So `1` becomes `2` and `2` becomes `1` as required. The actual value for `i` is again resolved using a lookup table (Line 9) called `parentKeyTableI` which is shown in Listing 6.2. Even and odd subdivision levels also require a different approach when calculating the parent key so we will explain the cases in detail. Refer to Figures 3.2 and 3.3 for visual support on this.

Listing 3.5: Calculating the parent key of a given key.

```
Input: A child key for which to generate a parent key.
Output: The finished parent key.
1  uint parentKey(in uint key)
2  {
3      uint i, numBitsI, x, y, numBitsXY;
4      decodeKey(key, i, numBitsI, x, y, numBitsXY);
5      numBitsI = numBitsI - 1u;
6      uint numBitsIFlipped = numBitsI ^ 1u;
7      uint iTableIndex = (numBitsI << 4u) | (i << 2u) |
8                         ((x & 1u) << 1u) | (y & 1u);
9      i = parentKeyTableI[iTableIndex];
10     numBitsI = numBitsIFlipped + 1u;
11     x = x >> numBitsIFlipped;
12     y = y >> numBitsIFlipped;
13     numBitsXY = numBitsXY - numBitsIFlipped;
14     return encodeKey(i, numBitsI, x, y, numBitsXY);
15 }
```

Going from an even subdivision level to an odd one, we can ignore the various triangle orientations since odd levels use the two bit `i` values. For a cell on even `x` and `y` coordinates (like `(0,0)`) an `i` value of `0` becomes `0` and `1` becomes `1`. A cell on odd `x` and even `y` coordinates (like `(1,0)`) results in `0` becoming `1` and `1` becoming `3`. A cell on even `x` and odd `y` coordinates (like `(0,1)`) results in `0` becoming `0` and `1` becoming `2`. And a cell on odd `x` and `y` coordinates (like `(1,1)`) results in `0` becoming `2` and `1` becoming `3`. Going from an odd subdivision level to an even one, requires taking into account the position of the cell to produce the correct triangle orientation. So a cell where `x + y` is even (like `(0,0)` or `(1,1)`) results in `0` and `2` becoming `0` and `1` and `3` becoming `1`. And a cell where `x + y` is odd (like `(0,1)` or `(1,0)`) results in `0` and `1` becoming `0` and `2` and `3` becoming `1`.

Since the number of bits for `x` and `y` only changes every second subdivision level (when going from an even level to an odd one), we just have to subtract `1` when the original `numBitsI` equals `1` (Line 13). For the new actual values of `x` and `y` we just have to do the opposite of what happens during children key generation so we have to half their original values when the grid's resolution changes (Lines 11 and 12). Remember that we are working with integers so when dividing odd values fractions are just discarded which is the reason why just halving their values actually works (e.g. `5 / 2` equals `2`). Thanks to integers and always dividing by `2` we replace the division with a bitwise right shift of either `0` or `1` depending on the current subdivision level.

When trying to generate a key for the parent of a triangle, it is important to know which of the two child triangles to use so that only a single parent key is created as mentioned in Section 2.5.4. In the original implementation by Khoury et al. this was a simple check that only had to consider the first bit of a key: `(key & 1u) == 0u`.

Listing 3.6: Helper method to know which child triangle to use for generating a parent key.

Input: The key for which to check if it is allowed to generate a parent key.
Output: True if the input key is allowed to generate a parent, false otherwise.

```
1  bool isChildZeroKey(in uint key)
2  {
3      uint i, numBitsI, x, y, numBitsXY;
4      decodeKey(key, i, numBitsI, x, y, numBitsXY);
5      uint xyEven = (x + y) & 1u;
6      if (numBitsI == 1u) {
7          return xyEven == 0u;
8      } else {
9          return ((i >> (xyEven ^ 1u)) & 1u) == 0u;
10     }
11 }
```

For our new procedure, shown in Listing 3.6, we have to consider the different triangle orientations to decide correctly. In principle, it does not matter which child triangle produces the parent as long as it is always exactly one. First we decode the key and check if the current subdivision level is even or odd based on the number of bits for `i` (Line 6). If it is even (Line 7), we know that there are `8` triangles to consider because on the parent level the grid's resolution will be halved and each cell will have `4` different triangle orientations. For simplicity's sake, we only look at the `x` and `y` coordinates leaving out the current `i` value. So as long as the key is located in a cell where the diagonal is going from the bottom left to the top right we accept it as the "zero key" that should produce a parent. If it is an odd subdivision level (Line 9), the grid's resolution stays the same but we have to take into account the alternating triangle orientations on the parent level. For cells where the parent has the diagonal going from the bottom left to the top right, we let the triangles on the left and bottom (in the context of a cell with 4 triangles inside)

produce a parent key. For the other cells with the diagonal going from the top left to the bottom right, we let the left and top triangle produce a parent.

Additionally, it is important to know when to stop merging triangles. In the original implementation by Khoury et al. exists another helper function for checking exactly that called `isRootKey`. The original checked if `key == 1u` while our concept requires a small change as can be seen in Listing 3.7. The reason being that a key with value `3` (`11` in binary) contains only `z` and `i` with `x` and `y` having zero bits available and thus having a value of `0` each. Since there cannot be a key with zero `i` bits, both `2` (`10`) and `3` (`11`), with the respective `i` values `0` and `1`, are the root keys.

Listing 3.7: Helper method to know when to stop looking for a parent triangle.

Input: The key for which to check if it can be merged into a parent key.
Output: True if the input key can be merged, false otherwise.

```
1 bool isRootKey(in uint key)
2 {
3     return key <= 3u;
4 }
```

The whole process of creating a parent key could possibly be further improved by only covering the triangles that return `true` for `isChildZeroKey` since those that return `false` won't produce a parent key anyway.

3.5 Key to triangle

Now that we know how the new keys are structured and how to generate children and parent keys, the last piece of the puzzle is how to calculate the barycentric coordinates for a triangle encoded in our keys. In the original solution by Khoury et al. this consists of `n` recursive matrix multiplications where `n` is the subdivision level of a key. As mentioned in Section 2.5.3, this is why performance drops with deeper subdivision levels. Our implementation of this procedure can be seen in Listing 3.8.

Our new procedure requires first decoding the keys in order to have access to the encoded parameters. Thanks to the grid, the triangle coordinates are fixed points and we just have to select the correct ones. This means it is necessary to first calculate the size of the cells `cellSize` at the current subdivision level `z` (Line 6). Remember that `z` is not returned by the `decodeKey()` function because the information is already encoded in `numBitsI` and `numBitsXY`. The cell size is calculated with `1 / numCells` due to barycentric coordinates being expressed in percent. `numCells` is defined as the number of cells in either one of the two dimensions of the grid and is calculated as $2^{numBitsXY}$. Due to working with integers, we can circumvent a `pow()` call by using a bitwise shift `1u « numBitsXY` instead.

Next we calculate all the fixed points that are relevant for the cell a given key's triangle resides in (Lines 8 to 18). Taking all possible triangle orientations over all subdivision

levels into account results in only five points. These are the four corner points of the cell as well as its center point that becomes relevant on odd subdivision levels. All that's left is picking the correct points for the triangle orientation of the given key. Since there are only eight possible triangle orientations in total, we implemented the point picking using another lookup table 6.3 that holds the five fixed points and an index into the lookup table (Lines 19 and 20). All of this results in a constant time procedure to turn a key into the barycentric coordinates of a sub-triangle.

Listing 3.8: Calculating the barycentric coordinates of the given key.

Input: The key for which to calculate the barycentric coordinates.
Output: Array containing the barycentric coordinates of all 3 triangle vertices.

```
1  void calcBarycentricCoordinates(
2      in uint key, out vec2 u_out[3]
3  ) {
4      uint i, numBitsI, x, y, numBitsXY;
5      decodeKey(key, i, numBitsI, x, y, numBitsXY);
6      float cellSize = 1.0f / float(1u << numBitsXY);
7      float cellSizeHalf = cellSize * 0.5f;
8      float lowerX = float(x) * cellSize;
9      float lowerY = float(y) * cellSize;
10     float upperX = lowerX + cellSize;
11     float upperY = lowerY + cellSize;
12     vec2 pointTable[5] = {
13         vec2(lowerX, lowerY),
14         vec2(upperX, lowerY),
15         vec2(lowerX, upperY),
16         vec2(upperX, upperY),
17         vec2(lowerX + cellSizeHalf, lowerY + cellSizeHalf)
18     };
19     uint pointMapIndex = ((numBitsI - 1u) << 3u) |
20                          (i << 1u) | ((x + y) & 1u);
21     u_out[0] = pointTable[pointMap[pointMapIndex][0]];
22     u_out[1] = pointTable[pointMap[pointMapIndex][1]];
23     u_out[2] = pointTable[pointMap[pointMapIndex][2]];
24 }
```

CHAPTER 4

Results

To properly verify the results of our solution, it is necessary to look at them both visually as well as in terms of performance. Starting with the visual comparison, we have to make sure that our new method produces the exact same results for all subdivision levels. In other words, the generated topological grid has to be identical to the one generated by the original. To verify this, we exported frames at different subdivision levels for both methods. Figure 4.1 shows the same portion of a terrain mesh at increasing subdivision levels, starting with level 4.

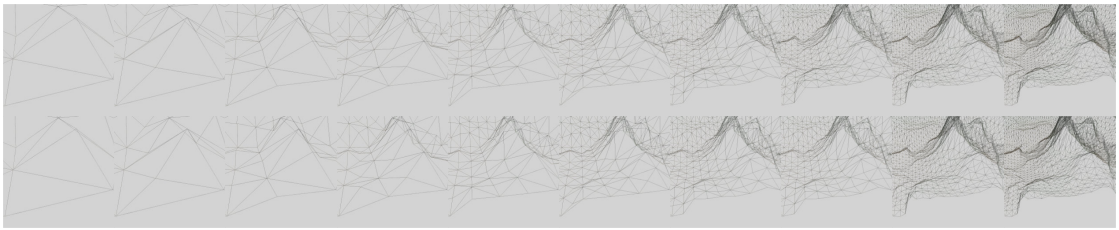

Figure 4.1: Visual comparison of the generated grid between the original (top row) and our new method (bottom row).

4.1 Test scene

We setup a test scene to be able to better compare our new procedure to the original. Conveniently, the original demo by Khoury et al. already contains a terrain example featuring a mountain range landscape. To be able to use this as a test scene more easily, we set it up so that the camera always starts in a position that triggers the deepest subdivision level.

Additionally, we implemented a short animation for the camera to follow along a fixed path including rotation of the camera. This is important because we can only make sure

everything works correctly when going down as well as up the subdivision levels in a natural way. Switching between the original procedure and ours is done through a simple checkbox and dedicated buttons make it easy to stop and reset the animation to the beginning.

The terrain as well as the animation path are shown in Figure 4.2. The animation starts at the left end of the red line with the camera looking towards the center of the terrain. This makes sure, that as much terrain as possible is visible and also a wide range of subdivision levels is covered by a portion of the animation. Later on, the camera passes closely by the rock face a few times to also cover deep subdivision levels.

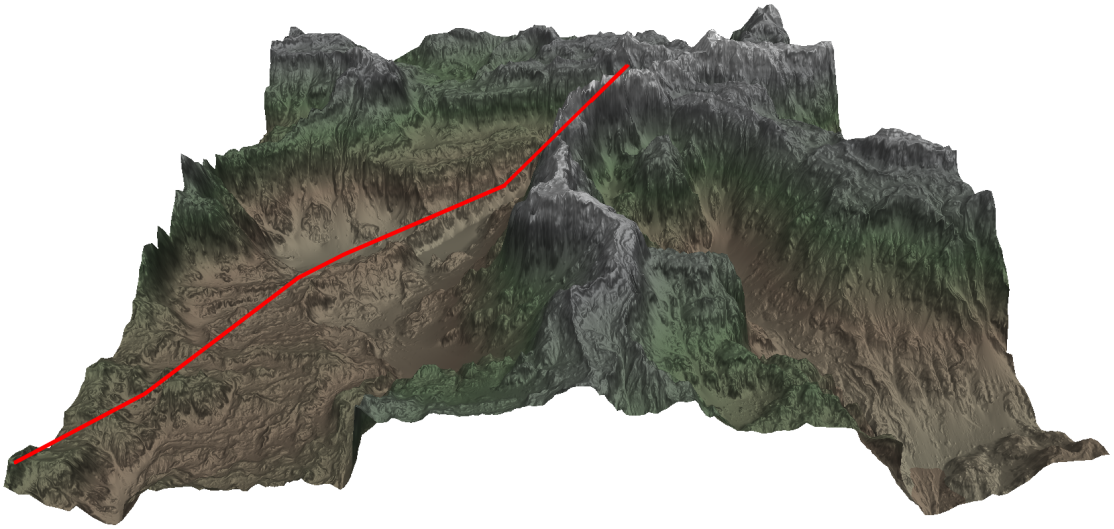

Figure 4.2: The terrain used in the test scene. The red line shows the camera animation path.

4.2 Performance comparison

In theory, our procedure should be faster than the original as it runs in constant time instead of linear. It is also likely, that our new method takes longer for some of the shallowest subdivision levels and overtakes the original only at deeper levels. To compare the two methods effectively, we recorded the frametimes while playing the camera animation for both solutions at a resolution of 1920x1080. We used the demo application by Khoury et al. and replaced their compute shader implementation with our implementation. To thoroughly test our solution we ran it on multiple machines and recorded multiple animation-runs for both solutions. In order to reach very deep subdivision levels we set the targeted edge length to 1 pixel. We left the compute shader's work group size at 32.

As can be seen in Figure 4.3, our solution manages to not only perform better at deep subdivision levels but is also faster at shallow levels. We managed to bring the average frametime of 19 ms down to 11 ms in our test scene. Comparing the average frametimes of both methods reveals an average performance improvement of 40%. Thanks to the nature of constant time algorithms, this difference will increase when using even deeper subdivision levels. Considering the variance of the frametimes, it is already obvious when looking at the graph that we managed to bring it down significantly, an average of 70% to be exact.

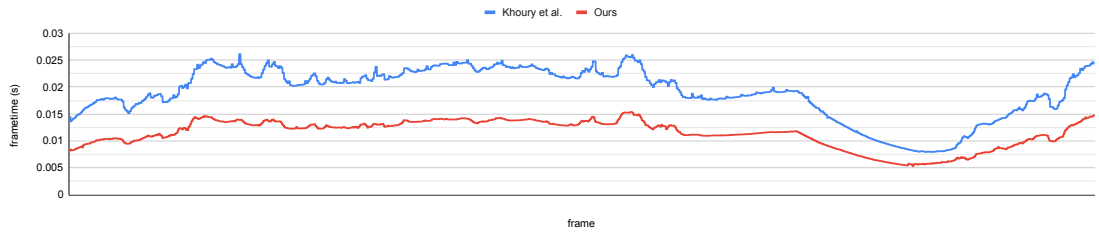

Figure 4.3: Frametime comparison (in seconds) between Khoury et al.'s solution (blue) and ours (red). The timings shown were recorded at a resolution of 1920x1080 and using an NVIDIA RTX 2070 Super GPU.

4.3 Drawbacks

There is currently a minor drawback to our new method that concerns the maximum number of subdivision levels. The base shape that the subdivision is applied to is always a triangle but the underlying representation we chose for our new method is a grid of squares. Since a right triangle is only half of a square, we are actually not using half of all possible keys. This comes down to our keys always requiring the orientation identifier `i` which makes our keys exactly one bit longer than Khoury et al.'s. The results is that our method allows one less maximum subdivision level compared to the original by Khoury et al., meaning our solution can only handle `31` subdivision levels instead of `32` when using 32bit integers. There is a way to fix this and reclaim the lost bit which we briefly discuss in Chapter 5.

CHAPTER 5

Conclusion and future work

The goal of this paper was to give an overview of current tessellation techniques with a focus on terrain rendering as well as to take a solution to a common problem and improve it by developing a drop-in replacement for its key component. As we have shown in Section 2, the topic of tessellation has many facets and is applicable in many areas from artistic creation to offline rendering to real-time applications. The latter has become more and more popular over the last few years and invites change regarding common hardware-based tessellation. With the introduction of the task/mesh shader pipeline, solutions like the one from Khoury et al. became even more relevant as they allow for much needed flexibility.

As mentioned in Section 2.5.5, Khoury et al. claimed that their solution solves two major problems of hardware-based tessellation while only the limit of at most 6 subdivision levels is solved. Their solution doesn't solve the second problem of longer computation times with deeper subdivision levels since their procedure is of linear time complexity. We managed to correct that mistake with our solution which runs in constant time, meaning subdivision levels don't impact the performance of sub-triangle generation. The results shown in Section 4 speak for themselves and show how software-based tessellation easily outperforms hardware-based solutions with it's two major flaws circumvented.

As mentioned before, our solution still has a minor flaw that brings the maximum subdivision level down by one compared to the original solution. We left this flaw for future work since we achieved our goal of finding a constant time solution. Also, if even deeper subdivision levels are required one can just use 64bit keys or even bigger ones through slight adjustments to the way our keys are encoded and decoded. Though this problem is fixable when considering that the number of triangles doubles with every subdivision level. We mentioned before in Section 4.3 that half of our keys are not used due to their represented triangles lying outside the input triangle. So on subdivision level zero there is `1` key unused, level one has `2` unused keys, level two has `4` unused keys and so on. That means that the number of unused keys on any level make up half of the

number of keys on the next level. For example, if we could only have 3 subdivision levels and want to regain the lost bit to build level 4, we can take `1` unused key from level zero, `2` unused keys from level one, `4` unused keys from level two and `8` unused keys from level three. This makes `15` unused keys out of `16` needed. The one missing key can then be inherently defined as the zero key. So it is theoretically possible to take all leftover keys on all subdivision levels to build the 32nd level, in effect regaining the lost bit.

CHAPTER 6

Appendix

Shown here are our implementations of the lookup tables required for calculating children keys, parent keys and barycentric points.

Listing 6.1: Lookup tables for the `x`, `y` and `i` values during children key generation.

```
uniform uint childTable[4][4] = {
    { 0u, 0u, 0u, 1u }, { 1u, 0u, 1u, 1u },
    { 0u, 1u, 1u, 1u }, { 0u, 0u, 1u, 0u }
};

uniform uint childrenKeysTableI[16][2] = {
    { 0u, 2u }, { 0u, 1u }, { 1u, 3u }, { 2u, 3u },
    { 0u, 0u }, { 0u, 0u }, { 0u, 0u }, { 0u, 0u },
    { 0u, 0u }, { 0u, 0u }, { 1u, 0u }, { 1u, 0u },
    { 1u, 0u }, { 1u, 0u }, { 1u, 1u }, { 1u, 1u }
};
```

Listing 6.2: The lookup table for the `i` value during parent key generation.

```
uniform uint parentKeyTableI[32] = {
    0u, 0u, 1u, 2u, 1u, 2u, 3u, 3u,
    0u, 0u, 0u, 0u, 0u, 0u, 0u, 0u,
    0u, 0u, 0u, 0u, 1u, 0u, 0u, 1u,
    0u, 1u, 1u, 0u, 1u, 1u, 1u, 1u
};
```

Listing 6.3: The lookup table for finding the correct points of a barycentric triangle.

```
uniform int pointMap[16][3] = {
    { 2, 3, 0 }, { 0, 2, 1 }, { 1, 0, 3 }, { 3, 1, 2 },
    { 0, 0, 0 }, { 0, 0, 0 }, { 0, 0, 0 }, { 0, 0, 0 },
    { 4, 0, 2 }, { 4, 0, 2 }, { 4, 1, 0 }, { 4, 1, 0 },
    { 4, 2, 3 }, { 4, 2, 3 }, { 4, 3, 1 }, { 4, 3, 1 }
};
```

List of Figures

Listings

Bibliography

[Eng20] Matthias Englert. Using mesh shaders for continuous level-of-detail terrain rendering. In *ACM SIGGRAPH 2020 Talks*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.

[HCGL20] Yongqing Hai, Siyuan Cheng, Yufei Guo, and Shaojing Li. Mesh smoothing algorithm based on exterior angles split. *PLOS ONE*, 15(5):1–24, 05 2020.

[HGCH21] Yongqing Hai, Yufei Guo, Siyuan Cheng, and Yunpeng Hai. Regular position-oriented method for mesh smoothing. *Acta Mechanica Solida Sinica*, 34(3):437–448, Jun 2021.

[HX17] Xuan Huang and Dianna Xu. Aspect-ratio based triangular mesh smoothing. In *ACM SIGGRAPH 2017 Posters*, SIGGRAPH '17, New York, NY, USA, 2017. Association for Computing Machinery.

[KDR18] Jad-Nicolas Khoury, J. Dupuy, and C. Riccio. Adaptive gpu tessellation with compute shaders. 2018.

[KJCH15] HyeongYeop Kang, Hanyoung Jang, Chang-Sik Cho, and JungHyun Han. Multi-resolution terrain rendering with gpu tessellation. *The Visual Computer*, 31(4):455–469, Apr 2015.

[LBG18] Thibaud Lambert, Pierre Bénard, and Gaël Guennebaud. A view-dependent metric for patch-based lod generation & selection. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1), July 2018.

[LKC+20] Hsueh-Ti Derek Liu, Vladimir G. Kim, Siddhartha Chaudhuri, Noam Aigerman, and Alec Jacobson. Neural subdivision. *ACM Trans. Graph.*, 39(4), July 2020.

[LLT+20] Thibault Lescoat, Hsueh-Ti Derek Liu, Jean-Marc Thiery, Alec Jacobson, Tamy Boubekeur, and Maks Ovsjanikov. Spectral mesh simplification. *Computer Graphics Forum*, 39(2):315–324, 2020.

[Loo87] Charles Loop. *Smooth Subdivision Surfaces Based on Triangles.* PhD thesis, January 1987.

[LS19] Eun-Seok Lee and B. Shin. Hardware-based adaptive terrain mesh using temporal coherence for real-time landscape visualization. *Sustainability*, 11:2137, 2019.

[MWS+20] D. Mlakar, M. Winter, P. Stadlbauer, H.-P. Seidel, M. Steinberger, and R. Zayer. Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the gpu. *Computer Graphics Forum*, 39(2):335–349, 2020.

[NL13] Matthias Nießner and Charles Loop. Analytic displacement mapping using hardware tessellation. *ACM Trans. Graph.*, 32(3), jul 2013.

[NLMD12] Matthias Nießner, Charles Loop, Mark Meyer, and Tony Derose. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Trans. Graph.*, 31(1), feb 2012.

[PBW19] Reinhold Preiner, Tamy Boubekeur, and Michael Wimmer. Gaussian-product subdivision surfaces. *ACM Trans. Graph.*, 38(4), July 2019.

[SAX+17] Zhuo Shi, Yalei An, Songhua Xu, Zhongshuai Wang, Ke Yu, and Xiaonan Luo. Mesh simplification method based on reverse interpolation loop subdivision. In *Proceedings of the 8th International Conference on Computer Modeling and Simulation*, ICCMS '17, page 141–145, New York, NY, USA, 2017. Association for Computing Machinery.

[Sta98] Jos Stam. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, page 395–404, New York, NY, USA, 1998. Association for Computing Machinery.

[TRK20] Ankur Tiwary, Muthuganapathy Ramanathan, and Jiri Kosinka. Accelerated Foveated Rendering based on Adaptive Tessellation. In Alexander Wilkie and Francesco Banterle, editors, *Eurographics 2020 - Short Papers*. The Eurographics Association, 2020.

[VMW18] Amir Vaxman, Christian Müller, and Ofir Weber. Canonical möbius subdivision. *ACM Trans. Graph.*, 37(6), December 2018.

[YS11] E. Yusov and M. Shevtsov. High-performance terrain rendering using hardware tessellation. *J. WSCG*, 19:85–92, 2011.

[YWH+16] Yazhen Yuan, Rui Wang, Jin Huang, Yanming Jia, and Hujun Bao. Simplified and tessellated mesh for realtime high quality rendering. *Computers & Graphics*, 54:135–144, 2016. Special Issue on CAD/Graphics 2015.

[ZMSS18] Rhaleb Zayer, Daniel Mlakar, Markus Steinberger, and Hans-Peter Seidel. Layered fields for natural tessellations on surfaces. *ACM Trans. Graph.*, 37(6), December 2018.