GPU Tessellation with Compute Shaders

a dissertation presented
by
Jad Khoury
to
The Department of Computer Sciences

in partial fulfillment of the requirements
for the degree of
Master of Science
in the subject of
Computer Graphics

Thesis Advisor: Mark Pauly
Project Supervisor: Jonathan Dupuy

E.P.F.L.
Lausanne, Switzerland
April 2018

©2018 – Jad Khoury
all rights reserved.

Thesis advisor: Professor Mark Pauly
Project Supervisor: Jonathan Dupuy

Jad Khoury

GPU Tessellation with Compute Shaders

Abstract

In this thesis, we present a new adaptive tessellation method capable of running entirely on the GPU. Our GPU implementation on the (nonrecursive) linear quadtree data-structure, in only storing the leaves of the tree as a concatenation of 2bits-codes describing the path to the root. We show how to exploit this data-structure to tessellate either triangles or quad polygons into smaller triangles by manipulating a quadtree per polygon. In order to produce efficient GPU level-of-detail, we exploit two different strategies to control the depth of the quadtrees that lead to two distinct implementations. First, we study a distance-based metric that consists of subdividing the quadtrees as primitives get closer to the viewpoint. We show that this metric works well for scenes carrying polygons of roughly the same shape and size. Second, we study a screen-space metric that consists of subdividing the quadtrees as primitives get larger in screen-space. We show that this metric is more general, but leads to high subidivision levels in the near projection plane, which is undesirable for primitives that lie outside of the view frustum. Both our implementations are free from complex pre-processing stages, lead to adaptive and crack-free surfaces, and offer up to 31 levels of subdivision, which surpasses the capabilities of the tessellation stage offered by current GPUs. We demonstrate the effectiveness of our implementations on terrain and arbitrary meshes, and conclude on the perspective of using our contributions to bring Catmull-Clark subdivision surfaces to the gaming industry.

Contents

0 Introduction **1**

1 Quadtrees on the GPU **5**
- 1.1 Introduction . . . 5
  - 1.1.1 Surface Representation . . . 5
  - 1.1.2 Quadtrees . . . 6
  - 1.1.3 Notations . . . 7
- 1.2 Previous Work: Linear Quadtree . . . 9
  - 1.2.1 Data-structure description and update . . . 9
  - 1.2.2 Mapping from leaf-space to quadtree-space . . . 10
  - 1.2.3 From quadtree-space to object-space, using quadtrees for meshes . . . 10
- 1.3 Contribution: Adapting Linear Quadtrees to Triangles . . . 11
  - 1.3.1 New tessellation scheme, same data-structure . . . 11
  - 1.3.2 Mapping from leaf-space to quadtree-space . . . 12
  - 1.3.3 Mapping from quadtree-space to object-space . . . 13
  - 1.3.4 Conclusion . . . 14

2 Adaptive Surface Subdivision - LoD and T-Junctions **15**
- 2.1 Introduction . . . 15
- 2.2 Distance Based Approach . . . 16
  - 2.2.1 Previous work: Distance Based LoD . . . 16
  - 2.2.2 Distance Based T-Junction Removal . . . 17
  - 2.2.3 Limitations of the Distance Based Approach . . . 20
- 2.3 Contribution: Screen-Space Approach . . . 20
  - 2.3.1 Screen-Space LoD . . . 20
  - 2.3.2 Screen-Space T-Junction Removal . . . 23
  - 2.3.3 Limitations of Screen-Space LoD . . . 28

3 Implementation Details **29**
- 3.1 Contribution: Pipeline Structure . . . 29
  - 3.1.1 Notations and definitions . . . 29
  - 3.1.2 First Approach: 3 passes, 1 array of command buffer . . . 30
  - 3.1.3 Second Approach: 2 passes & 1 copy pass . . . 35
- 3.2 Key Data-Structure . . . 38
  - 3.2.1 Format Description . . . 38
  - 3.2.2 Contribution: Key Initialization . . . 40
  - 3.2.3 Contribution: Key Decoding . . . 42
- 3.3 Contributions: Neighbour LoD Check, two other approaches . . . 45
  - 3.3.1 Neighbour Key Recovering . . . 45

3.3.2 Pre-Mapping Reflect 48

4 Results **49**
4.1 Distance Based Pipeline 49
4.2 Screen-Scape Pipeline 50

5 Conclusions **54**
5.1 Future Work 54
5.2 Personal Insight 55

References **57**

Acknowledgments

I would first like express my immense gratitude to Jonathan Dupuy, who supervised me during the complete duration of this project, and who always pushed me to pursue more complex and interesting solutions without ever showing any sign of doubt in my ability to find them.

I would also like to thank Christophe Riccio, who devoted countless hours to sharing with me his invaluable knowledge about GPUs, graphical libraries, and helped me shape the pipeline as it is today.

My deepest appreciation goes to the rest of the Grenoble Unity Labs team, that took me in as one of their own, and offered me the opportunity to stay a little bit more in this tiny research paradise that is their lab.

At last but not at least, I would like to thank Mark Pauly, for giving me the taste for Computer Graphics, offering me projects with his lab, and always supporting my love for rendering.

Finally, none of this would have been possible without the help and love from my family and friends.

0
Introduction

Context. In this thesis, we are concerned with the generation of digital images with a computer; Figure 1 illustrates a few such images. More specifically, we aim at developing an adaptive surface representation for objects that relies on Subdivision Surfaces and is suitable for GPU-accelerated rendering.

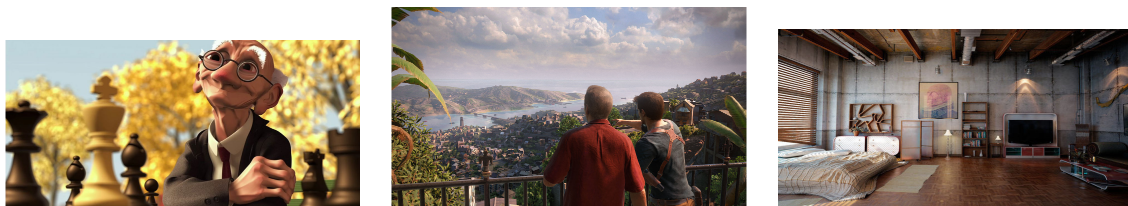


Figure 1: *Exemple of image renderings* (1) An animation movie: Pixar - Geri's Game (1997) (2) A video-game: Naughty Dog - Uncharted 4 (2016) (3) A photo-realistic render: Denis Vema - Industrial Loft 2 (2013)

Subdivision Surfaces. For most rendering purposes, the description of 3D objects is limited to their surfaces via the meshing of a set of surfaces sample points; figure 2 provides an illustration of this representation. A fundamental property of this representation is that the amount of detail of an object is directly proportional to the number of points used to describe it. As the ever-increasing expectations for realism requires more detailed 3D surfaces, subdivision surfaces have been developed to avoid the manipulation of unreasonably complex meshes, and have been used as the standard primitive in the animation industry since it was first used in Pixar's *Geri's Game* short movie in 1997. Intuitively, the idea behind a subdivision surface is to generate a smooth continuous surface from a coarse

set of meshed points. This can be accomplished by first densely tessellating the input mesh, and then displacing the new points according to a smoothing function; Figure 3 illustrates this idea. In this work, we want to generate such surfaces in parallel on the GPU. Our primary motivation is to bring this rendering primitive to the video-game industry, which is still an open problem [Brainerd et al., 2016].

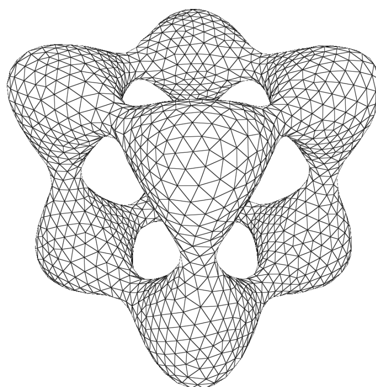


Figure 2: Example of meshing of a set of surfaces sample points

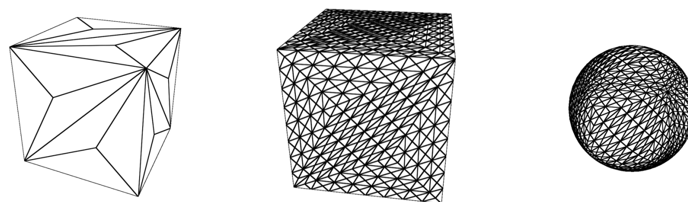


Figure 3: Example of surface subdivision applied on a cube mesh

The GPU. Real-time rendering pipelines leverage the Graphics Processing Unit (GPU). The GPU provides a hardware implementation of the Z-Buffer algorithm, which allows to convert polygon meshes into 2D images [Carpenter, 1984] [Catmull, 1974]. Nowadays, GPUs are parallel processors that can be programmed through *shaders*. An efficient surface representation on the GPU should satisfy two constraints. First, it should be fundamentally parallel in nature. Second, it should guarantee the polygons its handles span onto more than a few pixels, as below this limit, the Z-Buffer starts aliasing and the rasterizer's efficiency decreases drastically [Riccio, 2012].

Tessellation Shaders. Recent GPUs provide a tessellation engine [Cantlay, 2011] [D3D, 2009] that can be configured through tessellation shaders. Tessellation engines have been developed as an attempt to bring Subdivision Surfaces in real-time.

Despite their wide availability, their use has been limited for three main reasons. First, tessellation units significantly slow down GPU rendering pipelines, especially when tessellation factors are high. Second, they are limited to very low tessellation factors, allowing only 6 subdivision steps. Third, they often require intensive pre-processing on artist-designed meshes in order to yield crack-free surfaces [Brainerd et al., 2016].

Adaptive Surfaces. To be usable in real-time and on the GPU, the subdivision algorithms should produce optimal on-screen polygons by simplifying and/or enriching the mesh of the fly. This optimality resides in both producing an uniform level of detail on the generated image, while avoiding the production of micro-polygons. Since this subdivision is not uniform across the mesh, this pipeline should then decide, for each image, where and how much it should refine the surface. Figure 4.5 shows an example of such refinement. Quadtrees provide such a scheme, and have been used extensively since the early days of rendering, especially for terrain rendering [Lindstrom et al., 1996], but they have been ported on the GPU only recently [Dupuy et al., 2014] as their parallelization is not trivial. Even available on the GPU, they remained to be adapted as a part of a complete pipeline that subdivides terrains but also generic 3D meshes, in parallel.

Contributions and Outline. In the remainder of this thesis, we base our approach on the available GPU implementation of Quadtrees from [Dupuy et al., 2014] to propose a complete rendering pipeline. We leverage our pipeline to subdivide terrains but also generic 3D meshes, in parallel on the GPU in a few milliseconds. This contrasts with previous methods that either rely on the CPU [Strugar, 2009], produced cracks [Patney & Owens, 2008] or require sophisticated precprocessing stages [Brainerd et al., 2016] . We hope that our pipeline will accelerate the adoption of subdivision surfaces in the real time rendering industry. Our contributions are:

- A parallel quadtree representation suitable for the GPU
- Support for both triangles and quads
- 31 tessellation levels
- Real-time execution, *i.e.* less than 17 ms.
- Simple implementation, *i.e.* free from any kind of mesh pre-processing

First, in chapter 1, we discuss how we generate geometry using Quadtrees. In chapter 2 we determine where should this generation take place in the scene, and

how we can obtain crack-free surfaces regardless of the input mesh. Following the explanation of our approach, we will present in chapter 3 some of the implementations in greater details, before providing qualitative results of our algorithms in chapter 4.

1

Quadtrees on the GPU

1.1 Introduction

1.1.1 Surface Representation

The standard metric to qualify a rendering pipeline as “Real-Time” is the ability to reach a rate of 60 frames per second (FPS) on consumer-grade hardware, granting real-time algorithms less than 17ms to treat both the geometry and the shading. To satisfy this major performance constraint, real-time graphics solution to surface representation has been to formalize them as piecewise-linear surfaces, defined by a set of simple polygons stitched together. Most of them break down into storing a set of 3D position corresponding to vertices, and a set of subsets of these vertices that define either faces, edges, or oriented half-edges. Storing more vertices and faces can ensure a higher level of detail, like a higher sampling rate allows better reconstruction of a signal. GPUs excel in very fast parallel treatment of many (sometime millions) simple geometric primitives in real-time. But as the meshes become more and more detailed, the primary bottleneck has become the memory bandwidth between the memory storing the heavier surface representation and the processing unit in charge of treating it [Nießner et al., 2016].

In an effort to palliate the memory bandwidth limitation, Subdivision Surfaces have been adapted to the real-time pipeline, mainly with the development of Hardware Tessellation. The insight is still to generate geometry from a coarser mesh, but now on-the-fly and during a configurable stage of the pipeline. This tessellation allows very fast generation of many primitives but offers only a limited number of controls on the tessellation scheme itself, from the tessellation level that is set for the whole primitive (non-recursive LoD) to the tessellation pattern that is

configurable but not programmable. Furthermore, a maximum level of subdivision is enforced by the hardware, and because of the way it tessellates geometry, the use of Hardware Tessellation often requires the meshes to be pre-processed in order to obtain nicely subdivided crack-free surfaces

1.1.2 Quadtrees

Quadtrees are tree data-structures in which each node either has exactly four children, or is a leaf. They are often used for 2D space partitioning, as their construction is very intuitive and easy to code in its naive recursive variant. From a high level algorithmic point of view, constructing a quadtree consists at each step in considering a node, evaluating some division criterion on this node, and if the criterion is met, adding four children to the node and recursively repeat the routine on each of the created child. A visual example is given fig. 1.1.

When applied to surface subdivision, each quadtree node corresponds to a section of finite area in the considered 2D space, and the criterion can be any metric computed on this region (number of object in the region, color variation in the region, ...). The creation of four children corresponds to the partitioning of the space into four regions following a given *Tessellation Rule*. Traditionally, the considered 2D space is the unit square, and the children correspond to square regions in the root unit square. The region covered in object-space by a given node can then be computed using bilinear interpolation.

This algorithm has long been used for terrain representation [Lindstrom et al., 1996], as it perfectly fits a quad grid on which we generate details by creating children to the closest nodes, and then add details using a displacement map. Nevertheless, we want our algorithm to work not only on quad grids, but also on any given 3D mesh.

The first insight on our method is therefore the creation of one quadtree per mesh primitive (and later one quadtree per root triangle). We will then draw one base primitive per quadtree leaf, that we map from leaf-space to quadtree-space, and map once again from quadtree-space to the corresponding control mesh polygon in object-space, as illustrated in figure 1.2. This base primitive can be a quad or a quad grid, a triangle or a triangle grid, but it has to satisfy one condition: it has to be identical to an uniformly subdivided quadtree. In the case of a quad control mesh for example, the quadtree space will be the unit square. The base primitive can then either be a unit square, or a grid that is identical to a quad quadtree that is uniformly subdivided n times, giving a $2^n \times 2^n$ grid. The first few levels of quad grids are pictured in figure 1.3.

As we want our algorithm to be dynamic and run in real-time, in parallel and

on the GPU, the first order of business has been to get rid of the recursion and define an iterative way to create and update a quadtree.

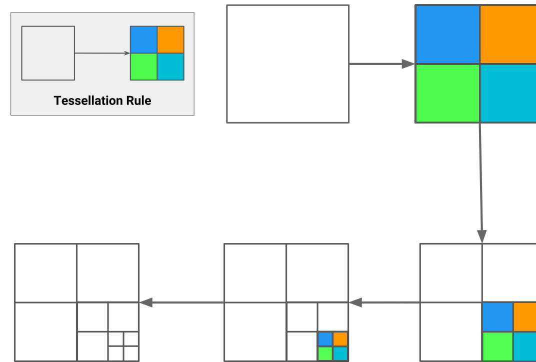


Figure 1.1: Generic run of the quadtree algorithm

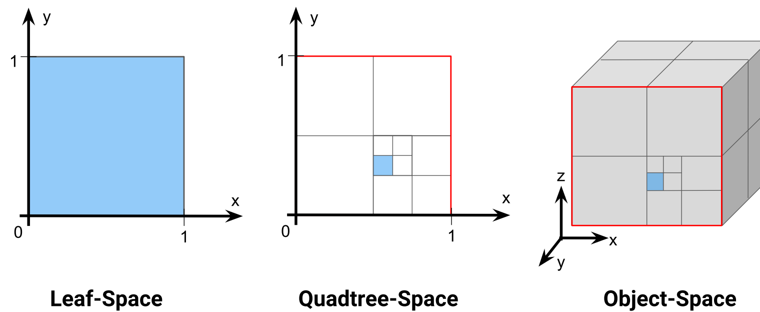


Figure 1.2: Illustration of the space nomenclature. In blue is represented the base primitive

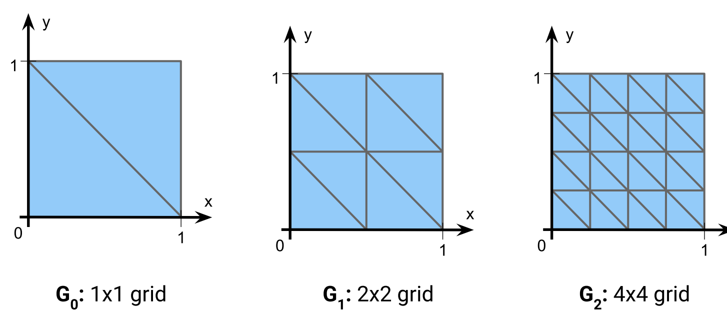


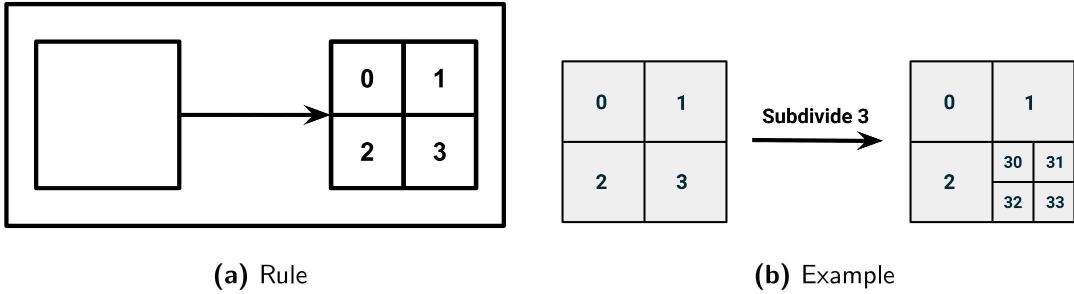
Figure 1.3: Three possible levels of the quad instanced grid

1.1.3 Notations

Before diving in the subject, let us define some notations:

- **Unit square / quad:** Polygon defined by $\{(0,0),(0,1),(1,1),(1,0)\}$

- **Unit (right) triangle:** Polygon defined by $\{(0,0),(0,1),(1,0)\}$
- **Unit Polygon:** designates either the unit quad or the unit triangle
- **Relevant Unit Polygon:** Unit quad if the control mesh is defined by quads, unit triangle if the control mesh is defined by triangles.
- **Unit Space:** 2D unit coordinate system $\{x, y \in \mathbb{R} \mid (x, y) \in [0,1]^2\}$
- **World Space:** 3D coordinates system $\{x, y, z \in \mathbb{R} \mid (x, y, z) \in \mathbb{R}^3\}$
- **Control Mesh:** Original mesh, before any subdivision scheme is applied
- **Transformation:** geometric application from $\mathbb{R}^n$ to $\mathbb{R}^n$. In our application, it's considered to be a set of one rotation, one translation and one scaling, stored in a 4×4 matrix.
- **Mapping:** geometric application from $\mathbb{R}^m$ to $\mathbb{R}^n$, mapping points from one space to another
- **Branching:** ramification of the tree data-structure at a given subdivided node. By extension, designates also the branch identifier defined by the Tessellation Scheme
- **Base primitive:** primitive generated on the CPU and drawn once per quadtree leaf. Depending on the control mesh, it can be a quad or a quad grid, a triangle or a triangle grid. For simplicity, we consider that the base primitive *always* spans on the same relevant unit polygon as the quadtree it's supposed to be used with.
- **Unit grid G_i:** Grid of dimension $2^i \times 2^i$ spanning on the relevant unit polygon, visually equivalent to a quadtree uniformly subdivided i times.
- **Leaf-space:** instance of unit space in which the base primitive spans on the relevant unit polygon
- **Quadtree-space:** instance of unit space in which the quadtree spans on the relevant unit polygon, and where the base primitive is mapped to cover the region designated by the considered node and tessellation scheme
- **Object-space:** Instance of world-space where the coordinate system corresponds to the basis of the control mesh.
- **Relative branching transformation $\mathbf{T_i(p)}$:** Mapping from Leaf Space to Quadtree-Space. Corresponds to the transformation mapping vertices from the base primitive defined on the relevant unit polygon, to the position corresponding to the branching i in the quadtree after *one* subdivision. In other words, it designates the transformation transforming a parent node into its child i, under the assumption that the parent node spans on the relevant unit polygon in unit space, as detailed in the beginning of this section.



(a) Rule

(b) Example

Figure 1.4: Numerated Quad tessellation rule along with an example of key construction

Final note on the used quadtree lexicon: as we always use a quadtree to partition a unit polygon, the relation between a node of the data-structure and the region it designates is bijective, and we can therefore use the quadtree lexicon to label both the data-structure concepts and the corresponding spacial regions interchangeably. Since we create one quadtree per control mesh polygon and that, before being mapped to object space, all computations are done in the same unit space, this non-distinction can be made without loss of generality. In this regard, we can say that the quadtree spans on a unit triangle, even though the quadtree itself is a data-structure and not a geometric entity.

1.2 Previous Work: Linear Quadtree

1.2.1 Data-structure description and update

Linear Quadtrees, proposed by [Gargantini, 1982], approach the creation of the tree in an iterative manner, freeing us from the limitations induced by implementing recursion on the GPU. To do so, the Tessellation Rule is numerated in such a way that each children position relative to its parent is identified by an integer between 0 and 3 (see figure 1.4a), that we call *branching identifier*. By accumulating these relative positions, we can identify each node uniquely by a key containing all the branching from the root to the node, as shown on fig. 1.4b. The key of a given child is then defined as the key of its parent, appended with the branching identifier of the child. Hence, the quadtree can be uniquely defined as a list of all its leaves keys.

To make the algorithm iterative, we first need to create two lists representing alternatively the quadtree at its state t and $t-1$. Then, we update it by evaluating, for each key, the subdivision criterion. Each key then produces in the "opposite" list either itself, its parent, its four children or nothing, as detailed in algorithm 1.

We observe many major advantages in the algorithm 1 : the tree is stored only

Algorithm 1 Linear Quadtree Update (called each frame)

```
1: read_list ← list0
2: write_list ← list1
3: for all key k in read_list do
4:     compute quadtree-space position of node from k
5:     evaluate Subdivision criterion
6:     if criterion requires subdivision then
7:         for i from 0 to 3 do
8:             child_k = append(k, i)
9:             write child_k in write_list
10:    else if criterion requires merging then
11:        if k represents an upper-left child then
12:            parent_k = truncateLastBranching(k)
13:            write parent_k in write_list
14:    else
15:        write k in write_list
16: swap list0 and list1
```

by the leaves, there never are duplicates, the execution is completely agnostic of the order of the key and we never need to sort them, and at last but not at least, each key can be treated independently from the others at a given frame. This allows each key to be treated in complete isolation, and we can therefore implement this algorithm in parallel. In particular, we can have one thread per key to read in the *read_list*, provided that the lists data-structures are accessed in a safe concurrent manner. The upper-left child test is used to avoid writing four times the key of the parent (since with a well constructed criterion, if a child wants to merge, all 3 of its siblings also want to merge).

1.2.2 Mapping from leaf-space to quadtree-space

Recall that in Linear Quadtrees, the surface on which spans the tree is the unit square $[0, 1]^2$. The base primitive that corresponds to a node is also a unit square (or a unit grid if the global subdivision level has to be artificially increased). The transform that maps the base primitive from leaf-space to its position in quadtree-space can then be decomposed in a scaling and a translation. These transforms can be recovered directly using the decoding routines presented in [Dupuy et al., 2014] and are illustrated in figure 1.5.

1.2.3 From quadtree-space to object-space, using quadtrees for meshes

To subdivide a mesh using this approach, one quadtree per mesh primitive has to be created and updated as explained in algo 1. As the quadtree subdivides

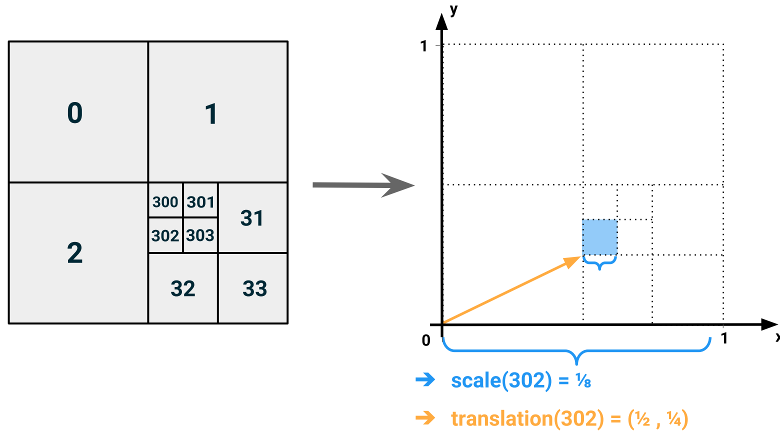


Figure 1.5: Example of transformation recovering from the key using the Linear Quadtree decoding

the surface defined by the unit square $[0,1]^2$, the position of the leaves are easily mapped from 2D quadtree-space to their position on the mesh in 3D object-space using bilinear interpolation: Let $\mathbf{p} = (p_x, p_y) \in \mathbb{R}^2$ be a point in quadtree-space and Q be the quad in object-space defined by $\mathbf{q_0}, \mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3} \in \mathbb{R}^3$. The mapping from quadtree-space to object-space can be computed as

$$\mathbf{q_{01}} = p_x * \mathbf{q_0} + (1 - p_x) * \mathbf{q_1}$$
$$\mathbf{q_{32}} = p_x * \mathbf{q_3} + (1 - p_x) * \mathbf{q_2}$$
$$\mathbf{p_{object-space}} = p_y * \mathbf{q_{01}} + (1 - p_y) * \mathbf{q_{32}}$$

The major limitation of this approach is its restriction to quads subdivision, therefore it cannot be used for triangular meshes that remain the standard surface representation in real-time graphics. Our first contribution has been the adaptation of the Linear Quadtree approach to triangle subdivision.

1.3 Contribution: Adapting Linear Quadtrees to Triangles

1.3.1 New tessellation scheme, same data-structure

To subdivide triangles, we first had to design a new tessellation pattern adapted to that polygon. Let us first define the *unit triangle* as the surface inside the triangle $\{(0,0),(0,1),(1,0)\}$. On this unit right triangle, we base the new tessellation pattern on the Sierpinski space filling curve and obtain the subdivision identification pictured on the fig. 1.6. Note that this tessellation scheme has been used in previous work ([Duchaineu et al., 1999], [Pajarola, 2002]), but our approach differs in that it computes two subdivision stages at once.

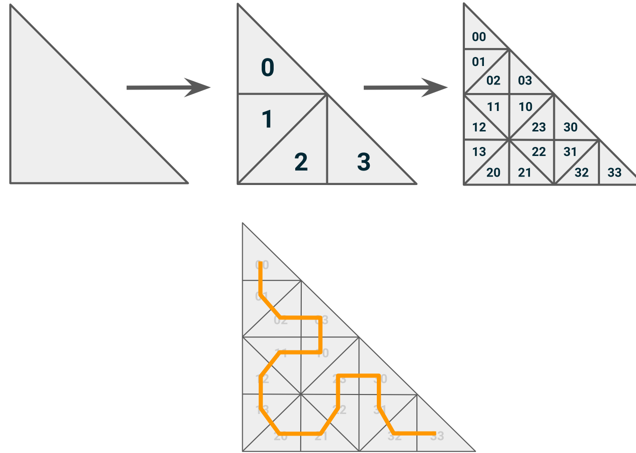


Figure 1.6: Top: tessellation scheme, bottom: Sierpinski space filling curve

As the numbering of the branchings and the tree construction do not change from the Linear Quadtree approach, the implementation of the creation and update of the tree using two arrays of keys can be kept as it is.

1.3.2 Mapping from leaf-space to quadtree-space

Nevertheless, the decoding of the keys, *i.e.* the recovering of the transformation mapping the base primitive to its position in quadtree-space, needed to be redesigned from scratch. Indeed, while quads only required scaling and translation, triangles also need rotations to fit inside the unit triangle. Intuitively, in figure 1.6, if we tried to obtain the child 1 from its parent, we would need to scale it down, translate but also rotate by $3\pi/2$. For this reason, instead of being able to recover the transform directly from the key, we need to decode it branching per branching, from the root to the leaf, and accumulate the relative branching transformations depicted in fig. 1.7.

In order to explain the intuition behind this method, we can picture a scenario where we have to recover the transform corresponding to the node 302. Recall that we define the *relative branching transformation* T_i as the transformation transforming a parent node into its child i, under the assumption that the parent node spans on the unit triangle. First, we recover T_3 which contains a scaling and a translation. Then, we recover T_0 and accumulate the scaling by multiplying it with the one found in the previous step, and add the found translation with the one in T_3. Finally, we compute T_2 and accumulate the translation and scaling in the same manner as before, while also adding the rotation that requires the branching 2.

Special care has to be given when accumulating the translation in cases where

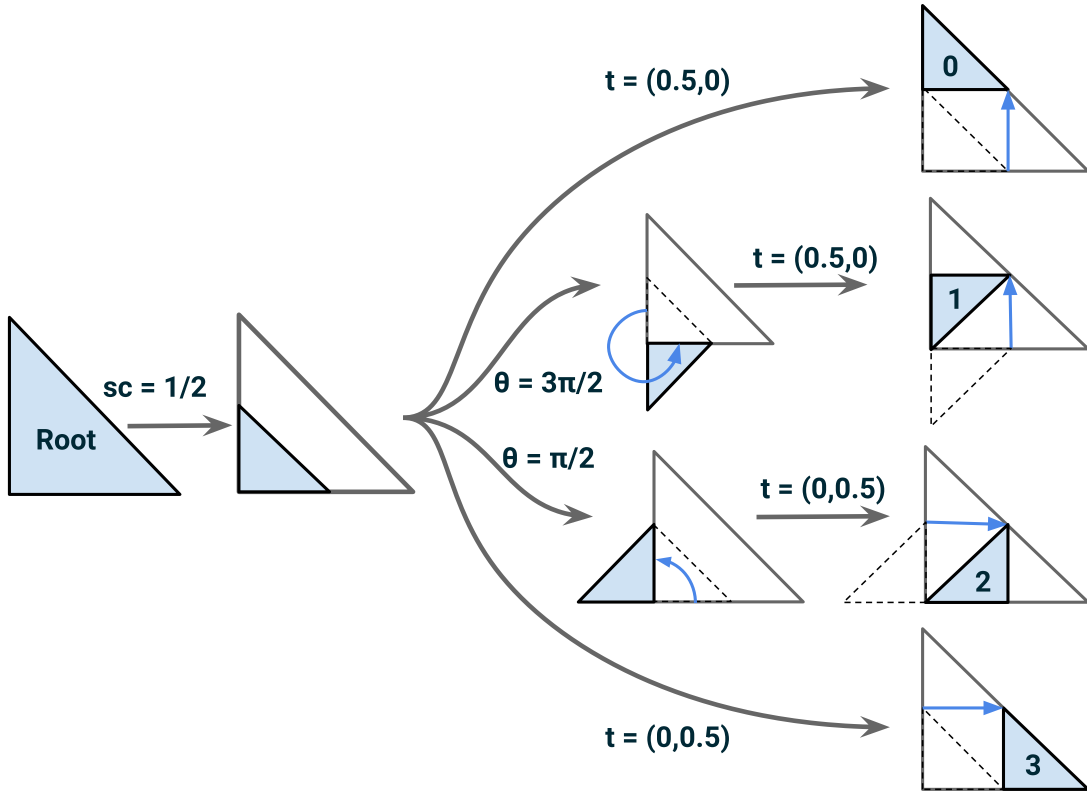


Figure 1.7: The four relative branching transformations associated with our tessellation scheme

previous steps found a rotation, since the direction should be rotated by the accumulated rotation, as explained in section 3.2.3. By accumulating these transforms and storing them in a 4 by 4 matrix, we obtain the complete mapping from leaf-space to quadtree-space corresponding to any given node. The algorithms used for this recovering are detailed in section 3.2.3.

1.3.3 Mapping from quadtree-space to object-space

Recall that for triangle quadtrees, the quadtree-space is defined by the unit triangle $\{(0,0),(0,1),(1,0)\}$. Any given point $\mathbf{p} \in \mathbb{R}^2$ in this space can therefore be expressed as

$$\mathbf{p} \in [0,1]^2$$
$$\mathbf{p} = (p_x, p_y)$$
$$p_x + p_y \leq 1$$

We can set

$$\lambda_0 = p_x$$
$$\lambda_1 = p_y$$
$$\lambda_2 = (1 - p_x - p_y)$$

Expressing p in function of the unit triangle vertices, we get

$$\begin{aligned}\mathbf{p} &= (p_x, p_y) \\ &= (1,0) * p_x + (0,1) * p_y \\ &= (1,0) * \lambda_0 + (0,1) * \lambda_1 + (0,0) * \lambda_2\end{aligned}$$

And since we have $\lambda_0 + \lambda_1 + \lambda_2 = 1$, the coordinates of $\mathbf{p}$ in quadtree-space can be used as barycentric coordinates of the point p inside the unit triangle. Using this, we can trivially map from the position in quadtree-space to the corresponding position on the target mesh triangle in object-space by using the quadtree-space coordinates as barycentric coordinates inside the target object-space triangle $T = \{\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}\}$:

$$\mathbf{p_{object-space}} = \mathbf{v_0} * p_x + \mathbf{v_1} * p_y + \mathbf{v_2} * (1 - p_x - p_y)$$

1.3.4 Conclusion

To summarize the triangle implementation of linear quadtrees, we obtained a complete definition of quadtrees on triangle as we:

1. Kept the same key format and data-structure
2. Defined a new tessellation pattern in the unit triangle
3. Designed the corresponding relative branching transformation
4. Developed a routine that constructs mapping from leaf-space to quadtree-space
5. Used the design of the quadtree-space to map points from quadtree-space to object-space

2

Adaptive Surface Subdivision - LoD and T-Junctions

2.1 Introduction

We presented a way to dynamically create and delete geometry from a control mesh that can be defined by quads or by triangles. To obtain a complete Adaptive Subdivision algorithm, what remains to be determined is where in the scene should more geometry be generated, and how to dynamically remove T-Junctions from the created geometry. The first problem can be broken down into finding an LoD function. This function should take some kind of geometric measure corresponding to the current node as input, and output the level of detail (hence LoD) of this node. The design of this function should aim at satisfying two goals:

1. Reach a constant polygon-per-pixel ratio
2. Obtain a Restricted Tree, *i.e.* a tree in which two adjacent nodes have **at most** one LoD of difference.

The first goal is oriented toward performance optimization. All polygons having the same size on the screen, no matter their position in the scene, ensures that the rendering has an uniform level of detail from the camera viewpoint, while minimizing micro-polygons that decrease drastically the efficiency of the GPU rasterizer [Riccio, 2012].

The second goal is rather a constraint that, when satisfied, allows us to run our T-Junction removal algorithms and ultimately to ensure a **crack-free, water-tight surface**. A T-junction is a degenerated situation where, when generating

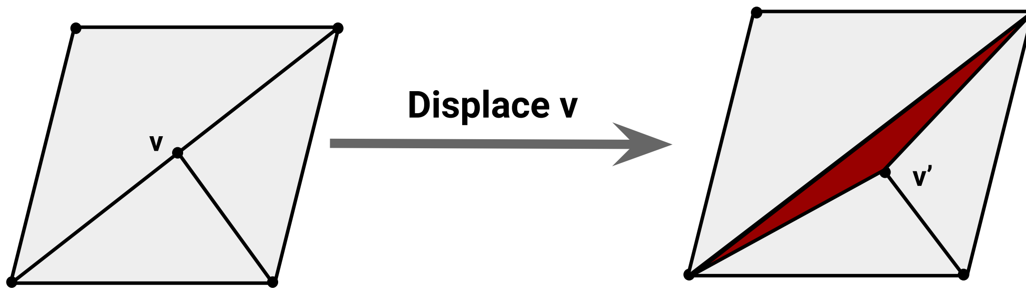


Figure 2.1: Example of a T-Junction. We see that if the vertex is displaced, it creates a crack (in dark red) on the surface.

geometry, a vertex is placed along the edge of an adjacent polygon. If we were to displace the mesh, regardless of the displacement method, we would obtain a crack in the surface, which is something to avoid by all cost. Figure 2.1 shows an example of such scenario.

2.2 Distance Based Approach

2.2.1 Previous work: Distance Based LoD

The final step of real-time rendering often consists in projecting the 3D objects in the scene from their world-space coordinates to their position on the screen using a Perspective Projection Matrix. This intuitively enlarges objects closer to the camera and shrinks the ones further away. An intuitive solution to finding the LoD function would then be to use the distance from the camera to the polygon as input metric, and the shorter it is, the higher is the LoD returned. This method has a long standing tradition in the rendering of terrains, as the control mesh is often a very coarse displaced grid, and close landforms would need a lot of generated detail to be visually satisfying. It implicitly tends towards a polygon-per-pixel ratio, and by using the $\log_2$ of a function of this distance, we are ensured to obtain a restricted tree.

The LoD function we used for Distance-Based tessellation is detailed in eq. 2.1:

$$LoD(dist) = -\log_2\left(\frac{dist * \tan\left(\frac{fovy}{2}\right)}{\sqrt{2} * factor}\right) \tag{2.1}$$

The $fovy$ constant corresponds to the horizontal angle of the field of view of the projection matrix. The $factor$ is a user-set float constant that increase or decrease the global level of detail continuously.

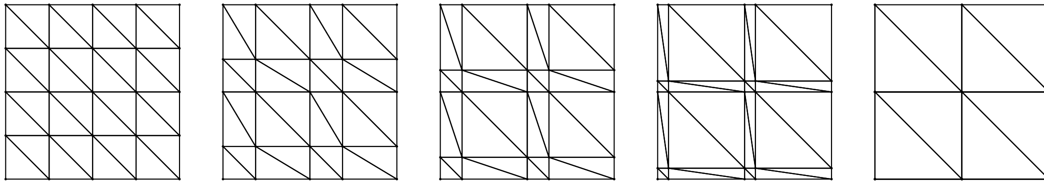


Figure 2.2: CDLOD morph function applied to the G_2 grid, with *morph_ factor* = 0, 0.25, 0.50, 0.75, 1.0 from left to right

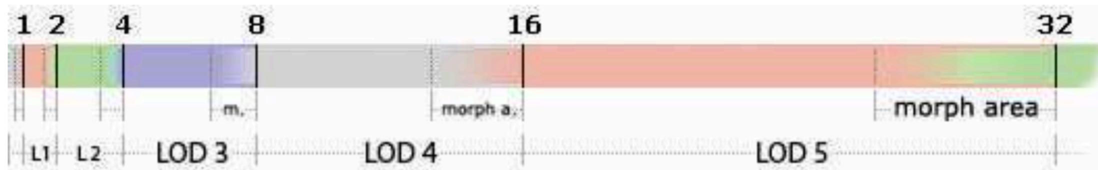


Figure 2.3: Illustration of the morph zones in 1D, image taken from CDLOD paper

For each node, we project from leaf-space to object-space the centroid of the relevant unit polygon ($(0.5, 0.5)$ for quads, $(0.3, 0.3)$ for triangles) from leaf-space to world-space as we would project any vertex of the base primitive, and use the distance from camera position to this projected polygon centroid as metric for this LoD function.

2.2.2 Distance Based T-Junction Removal

We now can associate a LoD to each node from its key, and the tree is ensured to be restricted, meaning that we can assume that no pair of adjacent nodes in the scene have more than one LoD of difference. Together with the instancing of quad or triangle grid with of dimension superior to 2x2, this allows us to implement the T-Junction removal approach proposed in [Strugar, 2009].

Previous Work: CDLOD T-Junction Removal This algorithm consists in defining a morphing zone between two distance threshold, in which the instanced grids vertices will morph along the outer edges to emulate the next lower LoD. That way, when reaching the actual distance threshold, the grid will already have artificially decreased its LoD and there will be no sudden LoD transition. In practice, this is implemented in a Vertex Shader function that takes as input the current vertex position in the leaf-space instanced grid, as well as a morphing factor between 0 and 1 that progressively morphs the vertex from its original position on the quad to its morphed position. The morph zone concept is visible in figure 4.1 (page 49), illustrated in figure 2.3 and the morphing proposed by [Strugar, 2009] is depicted in figure 2.2.

The idea behind this morphing is fairly simple: let us say that our base primitive

is a quad grid G_i of dimensions $2^i \times 2^i$, with $i \geq 2$. Recall that the subdivision of the instance grid and the subdivision of the quadtree are visually identical, meaning that if we subdivide once a quadtree on which we instantiate G_1, it yields the same set of vertices and edges than subdividing twice a quadtree on which we instantiate G_0, but also a non divided quadtree on which we span the grid G_2. Example of these grids are visible on figure 1.3 (page 7). More formally, subdividing a given node once following the tessellation rule pictured in figure 1.4b (page 9), divides it into a 2×2 grid, and on one of each of the four resulting quads, we draw one instance of the base primitive. As an effect, the original region on which spanned a $2^i \times 2^i$ grid is now represented by a $2^{i+1} \times 2^{i+1}$ quad grid. A harsh transition between two LoDs will then be visible as a $2^i \times 2^i$ grid, adjacent to a $2^{i+1} \times 2^{i+1}$ grid, creating many T-Junction. To avoid this sudden change in LoD, the solution is to morph the vertices present in G_{i+1} and not in G_i to progressively slides along the edges of G_{i+1} to eventually overlap the vertices of G_i, in such a way that there is no visual distinction between G_i and G_{i+1} in its completely morphed state.

For better expressive power, the method will be explained alongside an example. Let us say that we instantiate the grid G_2 of dimensions $2^2 \times 2^2 = 4 \times 4$. We denote G_2^* the set of vertices present in G_2 but not in G_1, and note that by construction

$$G_1 \subseteq G_2 \text{ and } G_2 = G_1 \cup G_2^*$$

We would like to slide the vertices in G_2^* of G_2 to finally overlap the vertices of G_1. The first step is to determine the direction of this slide. To do so, we scale G_2 by the number of interval (*i.e.* its 1D dimension, *e.g.* 4 for G_2) such that each interval has length 1, and then scale it down by a factor of 0.5. This has for effect of giving the vertices of G_1 integer values, while the vertices in G_2^* are decimal. Computing the `fract` (*i.e.* the decimal part) of a vertex position after this scaling, we obtain 0 for vertices in G_1, and 0.5 for vertices in G_2^*. By multiplying by 2 and dividing by the number of interval, we finally obtain the vector from a vertex position in its original grid to its position in the morphed grid. Multiplying this vector by a morphing factor and subtracting it from the vertex position in the base grid, we obtain the desired morphing function, which pseudo-code is given in listing 2.1

```
vec2 morphVertex(vec2 vertex, float morphK)
{
   float patchTessFactor = 4.0; // = nb of intervals per side of the primitive
   vec2 fracPart = fract(vertex * patchTessFactor * 0.5);
   fracPart *= 2.0 /patchTessFactor;
   return vertex - fracPart * morphK;
}
```

Listing 2.1: CDLOD morph function

Contribution: Adapting CDLOD morph function to triangle grids
This method only works when instancing quad grids, and we wanted to adapt it to run with our triangle quadtree. The LoD computation and subdivision can be transposed without further modifications, but the morphing of the instanced grid had to be adapted. Indeed, if used as it is, the vertices on the outer-edges of the triangle grid are also displaced as shown in figure 2.4. We then have to modify the morphing function that is applied to the vertices positions such that it becomes symmetric *w.r.t.* the hypotenuse median, as shown on figure 2.13, guaranteeing matching vertices position on two H-Neighbour. First, one can notice that the CDLOD morphing function always displaced the vertices in the negative direction, aiming at overlapping them with the vertices in G_1 that are either in a lower, a left, or a lower-left position. To achieve the symmetrical morphing desired, we should displace the vertices alternatively in two opposite direction depending on the interval it lays in. By taking the floor part of the scaled grid, we obtain the interval of the vertex, easily compute one factor per axis, each one set to 1 if the interval is even and -1 if the interval is odd (and stored in the `vec2 signVec` in listing 2.2)

```
vec2 morphVertex(vec2 vertex, float morphK)
{
   float patchTessFactor = 4.0; // = nb of intervals per side of the primitive
   vec2 fracPart = fract(vertex * patchTessFactor * 0.5);
   fracPart *= 2.0 /patchTessFactor;
   if (prim_type == TRIANGLES) {
      vec2 intPart = floor(vertex * patchTessFactor * 0.5);
      vec2 signVec = mod(intPart, 2.0) * vec2(-2.0) + vec2(1.0);
      return vertex.xy - (signVec * fracPart) * morphK;
   } else if (prim_type == QUADS) {
      return vertex - fracPart * morphK;
   }
}
```

Listing 2.2: Updated morph function for our triangle grid

This LoD and T-Junction Removal altogether can be seen in action in section 4.1

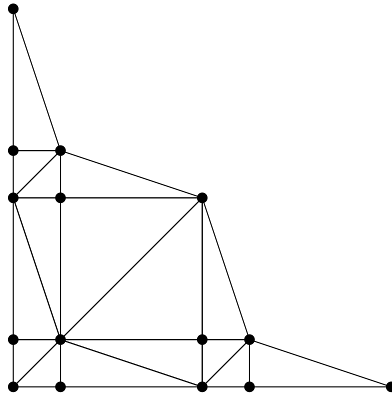


Figure 2.4: Problem when applying the CDLOD morph function to our triangle grid

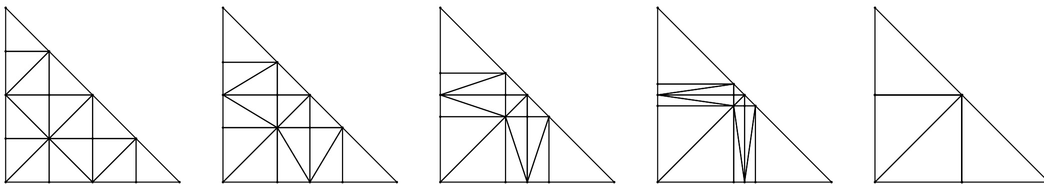


Figure 2.5: Updated morphing for our triangle grid, , with *morph_ factor* = 0, 0.25, 0.50, 0.75, 1.0 from left to right

2.2.3 Limitations of the Distance Based Approach

The problem with the distance based intuition is that we implicitly assume that the control mesh is regular, *i.e.* that the primitives all have approximately the same area. This may be the case for grids used in terrain rendering, but our application should also work on artist-designed meshes, with great variance in primitive areas. figure 2.6 depicts a scenario where using distance based level of detail trivially yields bad results. As we don't want to assume any additional constraints on the input mesh, a Screen-Space approach has been developed to circumvent the mesh regularity requirement.

2.3 Contribution: Screen-Space Approach

2.3.1 Screen-Space LoD

Intuitively, we should choose a metric that is representative of the Screen-Space area on which a given primitive spans. The larger the area, the more the control-mesh face should be subdivided. Using the projected area itself is nevertheless not a great idea: if we were to treat with this approach a very long and thin triangle with near-zero area, the LoD computed would be very low, hence not subdividing the triangle sufficiently and losing potentially many details of a displacement map (for example). The projected edge length could be a more robust alternative to

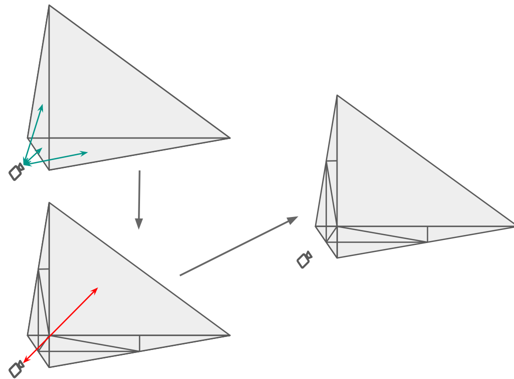
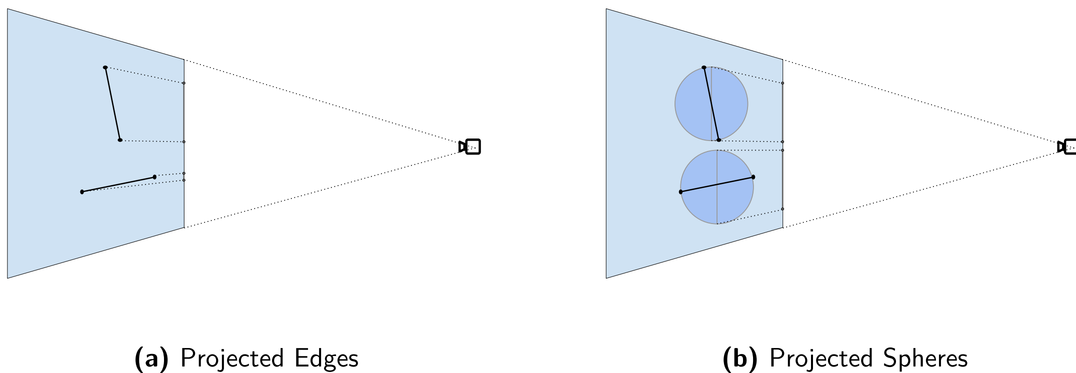


Figure 2.6: Showcase example of a situation where distance-based LoD is unsatisfactory. In green: LoD check requires subdivision. In red: no subdivision

the projected area, and is often used along with Hardware Tessellation for its simplicity and intuitive validity. Nevertheless, this approach could also lead to problems near silhouettes, as illustrated on figure 2.7a. To palliate to both of these problems, the chosen Screen-Space metric is the projected bounding sphere of the control mesh edges, like in Pixar's OpenSubdiv Real-Time API, illustrated in figure 2.7b.



(a) Projected Edges

(b) Projected Spheres

Figure 2.7: Showcase of a scenario problematic with edge-based metrics near silhouettes, but not with projected spheres

Since we chose to use the projected containing sphere diameter as the metric for the LoD evaluation, we should first determine on which edge will the LoD get evaluated. We could evaluate the LoD on the mean projected metric of the three edges of the triangle (or four for a quad), but that would break the restricted tree constraint (recall 2 adjacent leaves cannot have more than one level of difference). We therefore chose to triangulate every mesh primitives into a triangle fan.

Let us add two definitions to our lexicon for spaces and primitives:

- **Root Triangle:** Triangle resulting from the triangulation of the control mesh, detailed in the screen-space approach. One root triangle is subdivided by one quadtree, and one quadtree spans on exactly one root triangle.
- **Polygon-space:** Instance of unit-space, outlined by the relevant unit polygon, where the quadtree spans on the corresponding root triangle. In the chain of mappings, it sits between the quadtree-space and object-space.

A control mesh triangle is divided into 3 root triangles, and a control mesh quad is divided into 4 root triangles. On each one of these root triangle we map a triangular quadtree, and always evaluate the LoD on the hypotenuse of the instanced base primitive, as shown in figure 2.8

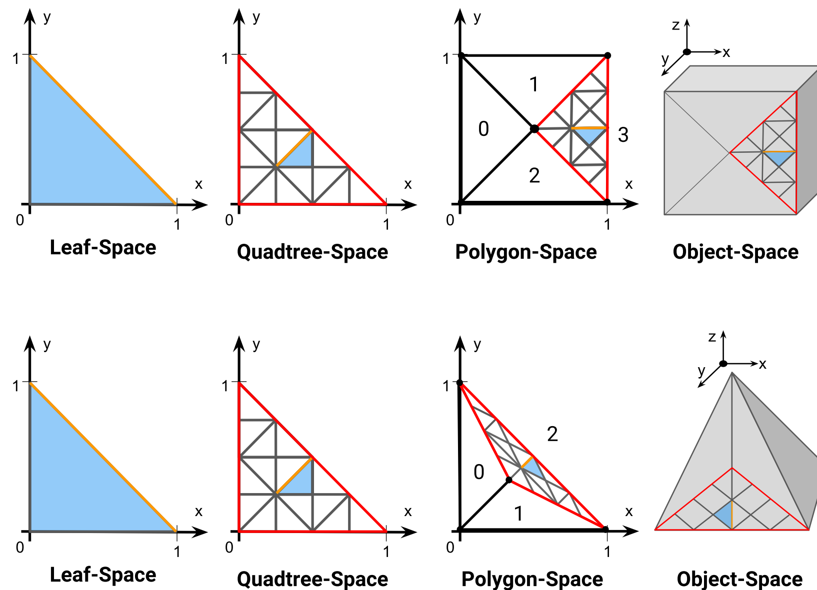


Figure 2.8: Triangulation of control mesh polygons, adding an intermediate space between quadtree and object. Top: Quads, Bottom: Triangles

The desirable effects of this triangulation are threefold:

- We can now assume that **all the quadtrees are triangular**, and most of the code that had to be adapted to both quad and triangle primitive is now simplified. As long as we triangulate, we always obtain triangle quadtrees and only the last step of the mappings from polygon-space to object-space is primitive-dependent.
- Since the base primitive is always a unit right triangle $\{(0,0),(1,0),(0,1)\}$ (or a grid outlined by this triangle), we can **always choose the same edge to be evaluated** for all drawn triangle: the hypotenuse $[(1,0),(0,1)]$

- By the construction of our triangulation, we are ensured that the triangles instanced along the borders of the mesh primitive always have their hypotenuse overlapping said mesh edge. And since the LoD is evaluated on this hypotenuse, we are ensured that **two triangles that are adjacent but belong to two different mesh primitive always have the same LoD**, see figure 2.9. For that reason, all T-Junction that could arise always do inside of the mesh primitive, between two triangles that our algorithm created. The T-Junction removal problem is now much more constrained and easier to solve.

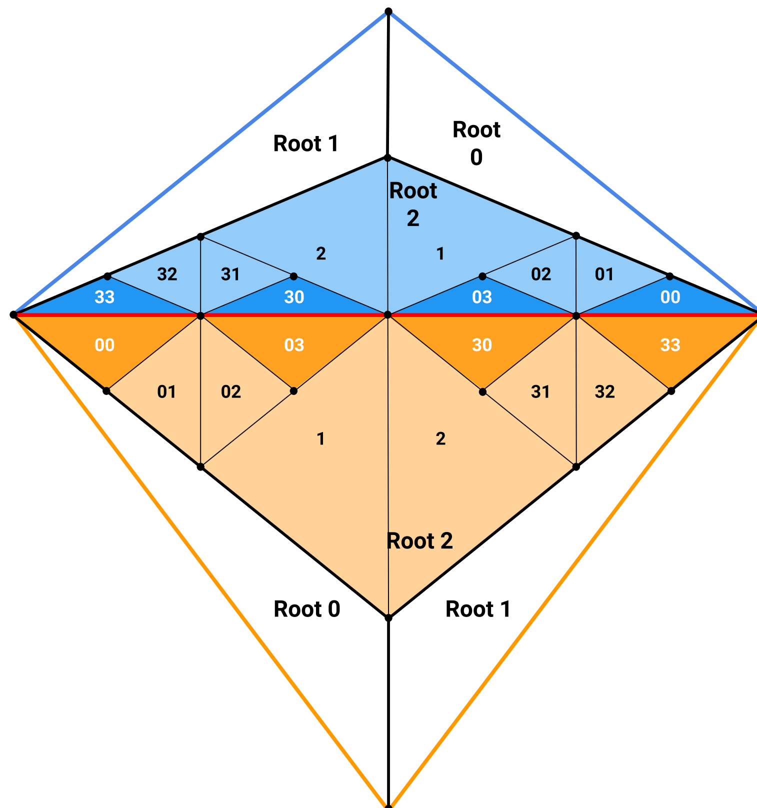


Figure 2.9: After triangulation, control mesh edges always support leaves hypotenuses (in red)

2.3.2 Screen-Space T-Junction Removal

To solve the T-Junction problem, the new triangulation and Screen-Space metric approach required the Compute Pass to first detect and then remove T-Junctions in a local neighbourhood.

T-Junction Detection Detecting T-Junctions around a given triangle is equivalent to recovering the LoD of its adjacent triangles. By definition, a triangle can have only 3 neighbours (we only consider edge-wise neighbourhood), and as

our instanced triangles are unit right triangles, we can denote the H-Neighbour (H for hypotenuse), the X-Neighbour (sharing the $[(0,0)-(1,0)]$ edge), and the Y-neighbour (sharing the $[(0,0)-(0,1)]$ edge) as depicted in figure 2.10. Note that, by construction of our tessellation scheme, with $Kn(T)$ designating the K-Neighbour of T, for two triangles T_0 and T_1 :

$$Hn(T_0) = T_1 \iff Hn(T_1) = T_0$$
$$Xn(T_0) = T_1 \iff Yn(T_1) = T_0$$

Since we always evaluate the LoD on the triangle hypotenuse, two H-Neighbours will always have the same LoD, and so at a local scale, detecting T-Junctions can be summarized in the algorithm 2

Algorithm 2 T-Junction Detection

1: Compute the coordinates in object-space of the X-Neighbour and Y-Neighbour's hypotenuse (in orange in figure 2.10)
2: Evaluate the LoDs for these coordinates
3: Compare the estimated LoDs with the current triangle LoD
4: **if** Computed LoD is inferior **then**
5: T-Junction is detected

Note that we always check for lower LoD to avoid ping-pong situation between two LoDs.

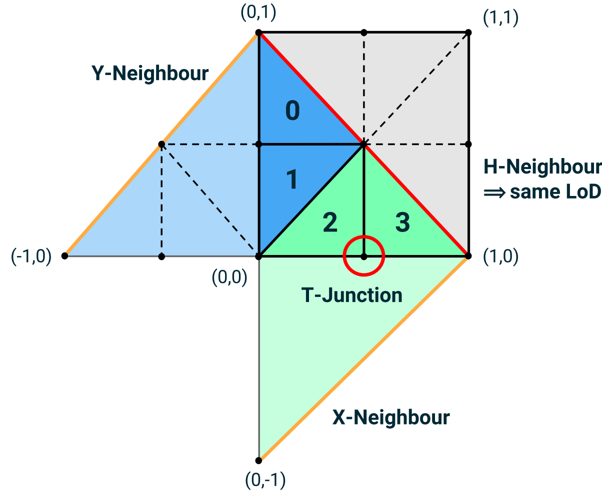


Figure 2.10: Neighbour notation, in leaf-space

To facilitate the second part of T-Junction removal, the removal itself, we do this evaluation on the parent's level. So in the scenario depicted in figure2.10, we place ourselves in the leaf-space of the parent of the current leaf. But since the two upper children (0 and 1) have no effect on the lower T-Junction and vice-versa,

we can restrict the computation to detect T-Junctions along the relevant edge of the parent. The step 1 of the T-Junction Detection algorithm 2 is then updated and detailed in algorithm 3

Algorithm 3 T-Junction Detection - Updated

1: **if** current leaf is on the X border of the parent (child 2 or 3) **then**
2: Compute the X-Neighbour hypotenuse object-space coordinates
3: **else if** current leaf is on Y border of the parent (child 0 or 1) **then**
4: Compute the Y-Neighbour hypotenuse object-space coordinates
5: Evaluate the LoDs for these coordinates
6: Compare the estimated LoDs with the current triangle LoD
7: **if** Computed LoD is inferior **then**
8: T-Junction is detected

The evaluation of the neighbour hypotenuse coordinates in object-space is less trivial that it may seem at first sight. We developed three separate solutions to this problem:

- Neighbour Key Recovering
- Pre-Mapping Reflect
- Post-Mapping Reflect

As we chose the last algorithm in the final implementation for its efficiency, we will only detail this one in the current section, and explain the other two in section 3.3 as they could have further interest (in particular the Neighbour Key Recovering).

The Post-Mapping Reflect method consists in reflecting the parent's hypotenuse *after* we apply the mappings to the hypotenuse. Since we already computed the object-space coordinates of the current node hypotenuse to evaluate its LoD, this approach allows us to save some computation and use these directly to compute the desired object-space coordinates. We define the reflection $Reflect(\mathbf{P}, \mathbf{O})$ as

$$\begin{aligned} Reflect(\mathbf{P}, \mathbf{C}) &= \mathbf{C} + \vec{PC} \\ &= 2\mathbf{C} + \mathbf{P} \end{aligned}$$

We note O, U and R as in fig. 2.11 and we call the two end-points of the searched hypotenuse H_0 and H_1, and T a temporary point created for simplicity. Using the schemas in figure 2.12, we can now recover the object-space coordinates of the target hypotenuse by implementing algorithm 4.

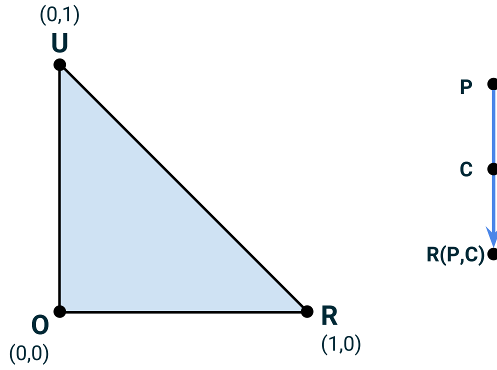


Figure 2.11: Notations for the Post-Mapping Reflect computations

Once we have these object-space coordinates, we don't need to map anything and can compute the neighbour's LoD directly. Special care has to be given in scenarios where the control mesh is displaced, since we should compute the reflected positions *before* displacement, and compute both current and neighbour LoD on *displaced coordinates*.

Algorithm 4 Neighbour Hypotenuse Computation in Object-Space

1: **switch** BranchingID **do**
2: **case** 0
3: $H_0 \leftarrow U$
4: $T \leftarrow Reflect(R, O)$
5: $H_1 \leftarrow Reflect(H_0, T)$
6: **case** 1
7: $H_0 \leftarrow Reflect(R, O)$
8: $T \leftarrow Reflect(U, O)$
9: $H_1 \leftarrow Reflect(H_0, T)$
10: **case** 2
11: $H_0 \leftarrow Reflect(U, O)$
12: $T \leftarrow Reflect(R, O)$
13: $H_1 \leftarrow Reflect(H_0, T)$
14: **case** 3
15: $H_0 \leftarrow R$
16: $T \leftarrow Reflect(U, O)$
17: $H_1 \leftarrow Reflect(H_0, T)$
18: Map the edge $[H_0, H_1]$ to object-space

If the neighbour lies outside the current root triangle, we recover the neighbour's node identifier `nodeID` (method detailed in section 3.3.1), recover the identifier of the root triangle it lies in (`rootID`), use the `nodeID` and `rootID` to map the unit triangle hypotenuse $[(1, 0), (0, 1)]$ from leaf-space to object-space, and finally compute the LoD on the object-space coordinates.

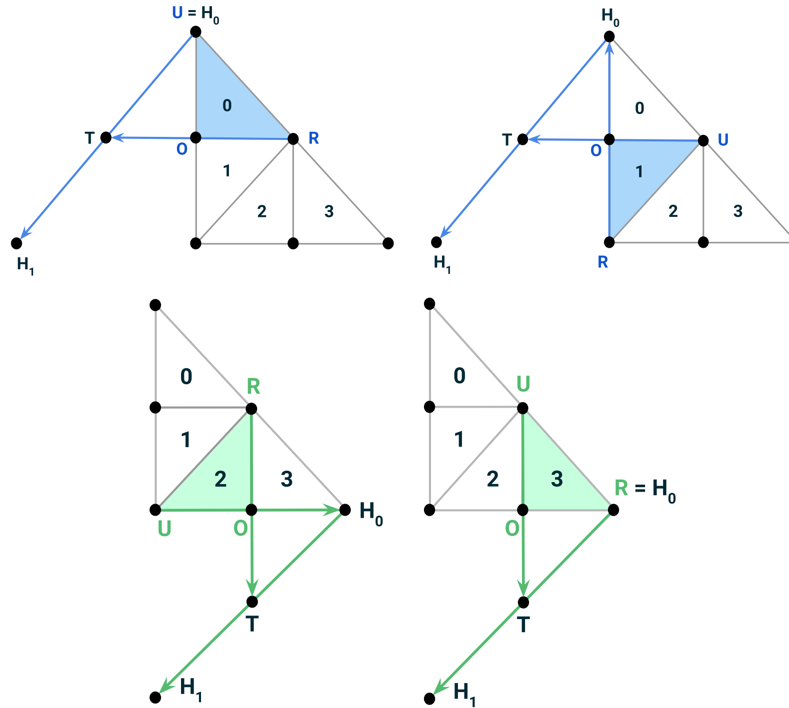


Figure 2.12: Post-Mapping Neighbour Hypotenuse computation for leaves 0 (top-left), 1 (top-right), 2 (bottom-left) and 3 (bottom-right)

T-Junction Removal Now that the LoD of the neighbour is evaluated, we can deterministically detect any T-Junction in the local neighbourhood of the currently treated quadtree leaf. What remains to be determined is the removal of discovered T-Junction.

Recall that in all the neighbour LoD check methods, two leaves sharing the same border of their parent will evaluate the same neighbour, hence detecting the same T-Junction if there should be one. E.g. in figure 2.10 (page 24), if there was a T-Junction between the X-Neighbour and the parent of the four children, both children 2 and 3 will detect it. For this reason, we reserve in the key data-structure 3 bits that serve as flags for X-morph, Y-morph and Destroy. Since leaves that detect a T-Junction always do so in pairs, when detecting a T-Junction in the Compute Pass, we set for the even child the relevant morph flag, and for the odd sibling, the destroy flag. In the render pass, we simply don't render any node with the destroy flag set, and morph appropriately the nodes with the morph flag to take both of their places. To come back to our example, in figure 2.10, the child 3 will get destroyed and the child 2 will be transformed to take both of their places, as illustrated in figure 2.13.

In conclusion, our T-Junction removal approach allows us to

- Detect all T-Junction in the created subdivision

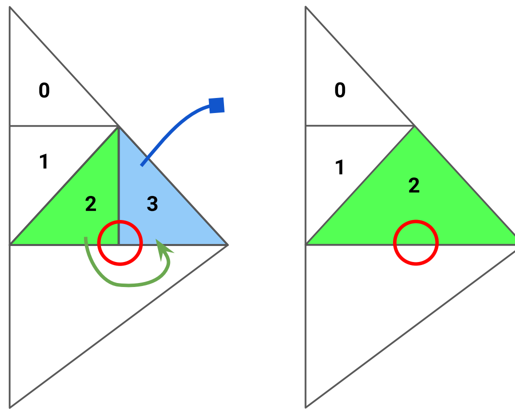


Figure 2.13: T-Junction Removal in the Screen-Space Approach

- Remove them
- Every T-Junction kills an odd quadtree leaf $\Longrightarrow$ 1 less triangle to render per T-Junction
- The morph bit is propagated along with the key, allowing the render pass to morph directly when computing the mapping from unit-space to object-space using the key

2.3.3 Limitations of Screen-Space LoD

The major caveat of our implementation of the screen-space metric resides in the fact that by using the projected length, it subdivide to an extreme level the primitives that are very close to the zNear plane, as shown in figure 4.6 (page 52).

3

Implementation Details

In this section, we will go back to some of the algorithms and aspects of the project presented in the previous chapters, to provide the reader a more in-depth review of our implementations and methods.

3.1 Contribution: Pipeline Structure

The program structure as well as the buffer architecture changed often in the development of this project, and we will only explain the two major variants in their most accomplished form. Before diving in, let us first define a few key concepts.

3.1.1 Notations and definitions

- **Pass:** each frame requires the execution of a few programs sequentially, that we denote on a higher level as passes
- **Compute Shader:** programs that run on the graphics card, outside of the normal rendering pipeline. They can be used for massively parallel GPGPU algorithms, or to accelerate parts of 3D rendering. We use the term *dispatch* to designate the launch of compute shader invocation. They are run in workgroups.
- **Workgroup:** a set of compute shader invocation, that execute in parallel and share the GPU shared memory. The workgroup is defined by a 3D local size, but since we don't take advantage of GPU shared memory, and therefore don't need a conceptual multi-dimensional partition of our working space, we treat the workgroups in 1D (*i.e.* we define the x dimension of the workgroups). It is common practice to have a local workgroup count of at

least 32 for better efficiency. Furthermore, the workgroups themselves are dispatched in a 3D manner, where for each dimension we have a workgroup count, *e.g.* we could have $3 \times 3 \times 3$ worgroups, each of size $32 \times 32 \times 1$, for a total of 27648 compute shader invocations. As before, we just use the x dimension of the workgroup count.

- **Shader Storage Buffer Object (SSBO):** Buffer Object that is used to store and retrieve data from within the OpenGL Shading Language [Segal & Akeley, 2017]. They offer large storage capacity on video memory, while allowing shaders to both read and write in their data-structure.
- **Atomic Counter**: a data-structured proposed by OpenGL that offers atomic operations and that allows the handling of variables in a concurrent manner. The use of a variable as Atomic Counter is defined by the way it is bound to the program from the CPU, and not by the variable itself, meaning that the same variable could be treated as an atomic counter in a given pass, and as a variable stored in an SSBO in a different pass, as long as the bindings are implemented correctly on the CPU side. It is also possible to call atomic functions on variables stored in an SSBO, as it is the case in the first approach detailed below.
- **Indirect Call:** The traditional method for both draw calls and compute dispatches consists in implementing them on the CPU side, and passing all the required parameters as function arguments. To be able to modify these parameters from GPU, OpenGL offers the possibility of storing these parameters in a buffer, stored locally on the graphics card. Then, instead of using the traditional function calls, we bind this buffer appropriately and use the indirect variant of the OpenGL function. The program will then look for the parameters in the buffer and launch the shader invocations as desired.

3.1.2 First Approach: 3 passes, 1 array of command buffer

In its initial design, the program goes through three passes at each frame. We use the `DrawElementsIndirect` command to be able to set the number of instances to render asynchronously on the GPU. To do so, we created the following buffer architecture.

Two lists of keys implemented as two Shader Storage Buffer Objects (SSBOs), respectively corresponding to the quadtree at frames t and $t-1$, and between which we ping-pong at each frame. For example, at a given frame, the program works as follow:

- *Compute Pass*: Read the keys from SSBO 0
- *Compute Pass*: Write the new keys in SSBO 1
- *Cull Pass*: Read the new keys in SSBO 1
- *Cull Pass*: Write the keys that didn't get culled in SSBO 0
- *Render Pass*: Read the keys that didn't get culled in SSBO 0 and render the corresponding triangles
- *CPU*: swap SSBO 0 and SSBO 1

It is important to keep the keys that did get culled since if the camera were to rotate and look at a region of the mesh that was previously culled, the Compute Pass should still be able to treat the keys, and eventually merge / divide the nodes in this area.

An array of DrawElementsIndirectCommand that we will call `command_array`, stored in an SSBO and that has two purposes: keeping track of the number of keys to treat in a given pass, and serving as command buffer for the indirect draw calls. Each element of the array contains a complete `DrawElementsIndirectCommand`, as pictured in fig. 3.1. Each one of these command is composed of 5 variables corresponding to different parameters for the draw call, one of which designates the number of instances of base primitive to draw: `primCount`. By atomically incrementing that variable at the correct index, each pass can then keep track of the number of keys it produces, and OpenGL can later use its final value to render the correct number of primitives. In more details, the `command_array` is managed as follow:

- *Compute Pass*: Read the `primCount` from `command_array[i]`
- *Compute Pass*: At each key write, atomically increment the `primCount` in the `command_array[i+1]`
- *Cull Pass*: Read the `primCount` from `command_array[i+1]`
- *Cull Pass*: At each key write, atomically increment the `primCount` in `command_array[i+2]`
- *Render Pass*: Read the `primCount` from in `command_array[i+2]`
- *CPU*: Reset the `DrawElementsIndirectCommand` in `command_array[i+shift]` and in `command_array[i+shift+1]`
- *CPU*: $i \leftarrow i + 2$

Note that the index incrementation is done modulo the number of elements in the array, such that the executions cycle through it. The resetting is critical to keep a correct count of primitives to treat since we should not increment on the previous `primCount`. We use an array and delete with a shift instead of simply using two commands between which we ping-pong as we want the implementation to stay as asynchronous as possible. If the draw command is managed in a ping-pong manner, we would need at each frame to wait for the target `DrawElementsIndirectCommand` to get reset before being able to start the computation for the next frame. Using a shift, we can start incrementing the next command without being ensured that the value at the end of last frame has been reset. This buffer architecture is illustrated in figure 3.1.

As one can notice, this implementation relies on three passes.

A Compute Pass that implements in a Compute Shader the algorithm 1 (page 10), responsible for updating the Linear Quadtree data-structure for a frame. We call `read_list` the SSBO in which the last iteration stored the keys, and `write_list` the SSBO that will store the new keys. Two elements of the `command_array` are bound as SSBOs, corresponding to `command_array[i]` and `command_array[i+1]` in the algorithm explained in the previous paragraph. In each of these buffers, we denote the variable of interest respectively as `old_primCount` and `primCount`.

At each frame, we dispatch a fixed number of shader invocations running the pseudo-code presented in listing 3.1.

```
int leaf_count = old_primCount;
int invocationID = getUniqueInvocationID();
if (invocationID > leaf_count)
   return;
uvec4 key = read_list[invocationID];
vec3 p_obj_space = mapToObjSpace(p_leaf_space, key);
float current_LOD = getLevelInKey(key);
float target_LOD = computeLoD(p_obj_space);
if (target_LOD < current_LOD - 1) {
   if (isUpperLeftChild(key)) { // avoid storing 4 copies of the parent key
      int idx = atomicAdd(primCount, 1); // returns value before atomic increment
      write_list[idx] = getParentKey(key);
   }
} else if (target_LOD > current_LOD ) {
   uvec4 children[4] = getChildrenKeys(key);
   for (int i = 0; i < 3; ++i) {
      int idx = atomicAdd(primCount, 1);
      write_list[idx] = children[i];
   }
} else {
   uint idx = atomicAdd(primCount, 1);
   write_list[idx] = key;
}
```

Listing 3.1: Compute Pass in the first program architecture

At the end of this pass, we obtain an updated Quadtree that satisfies the restricted tree constraint as long as the LoD function is well defined.

A Cull Pass , also implemented in a Compute Shader, and responsible for copying in the opposite quadtree SSBO the keys that correspond to nodes that are inside the view frustum. The implementation is simple: we bind two elements of the `command_array` as SSBOs, corresponding to `command_array[i+1]` and `command_array[i+2]` in the paragraph focused on the array of `DrawElementsIndirectCommand`. The variables of interest in these buffers are denoted like in the Compute Pass, and the pass pseudo code is detailed in listing 3.2.

```
int leaf_count = old_primCount;
int invocationID = getUniqueInvocationID();
if (invocationID > leaf_count)
   return;
uvec4 key = read_list[invocationID];
vec3 p_obj_space = mapToObjSpace(p_leaf_space, key);
if (isOutOfFrustum(p_obj_space))
   return;
uint idx = atomicAdd(primCount, 1);
write_list[idx] = key;
```

Listing 3.2: Cull Pass in the first program architecture

A Render Pass responsible for rendering the triangles corresponding to the last state of the quadtree. The number of invocations is fixed by the number of vertices in the primitive to render, and by the `primCount` set by the cull pass. By now the render pass implementation should be trivial, but is detailed in Listing 3.3

```
// VERTEX SHADER
in vec3 vertex;
out vec3 v_pos;
// ...
uvec4 key = read_list[gl_InstanceID];
vec3 p_obj_space = mapToObjSpace(vertex, key);
v_pos = p_obj_space;

// FRAGMENT SHADER
in vec3 v_pos;
// ...
color = shade(v_pos);
```

Listing 3.3: Render Pass in the first program architecture

With this implementation, it becomes clear why we don't need to clear the buffer containing the keys: at each pass, the invocations read the key stored at the index corresponding to their invocation identifier. If this ID is greater than the number of keys to treat, the invocation returns without computing anything. Otherwise, the current invocation writes the correct key(s) at the first available index in the list, determined by the use of an atomic counter with value 0 at the beginning of the pass, that is then incremented at each key write. In this manner, the keys are tightly packed, and only the relevant keys get treated. Nevertheless, a major caveat of this approach is that the number of compute shader invocations is fixed. This number should be large enough to tackle the worst case scenario where the key lists are full, but if there only are a few nodes to treat, a very large proportion of these invocations will be launched for nothing, thus greatly

impacting the performance of the program.

We see that this implementation is quite complex and it truly has been hard to maintain and update when we changed some details in our implementation, in particular keeping up with the read, write, full, culled, reset, shift indices of the arrays. The three passes architecture is also wasteful as it requires to compute the world space coordinates three times, once in each pass. This operation is very expensive since we must apply three mappings to each primitive in the quadtree. The resetting of the `primCount` values with a CPU OpenGL command was also not ideal. To palliate to all the caveats of this implementation, we decided to design once again everything from scratch and obtained the following implementation.

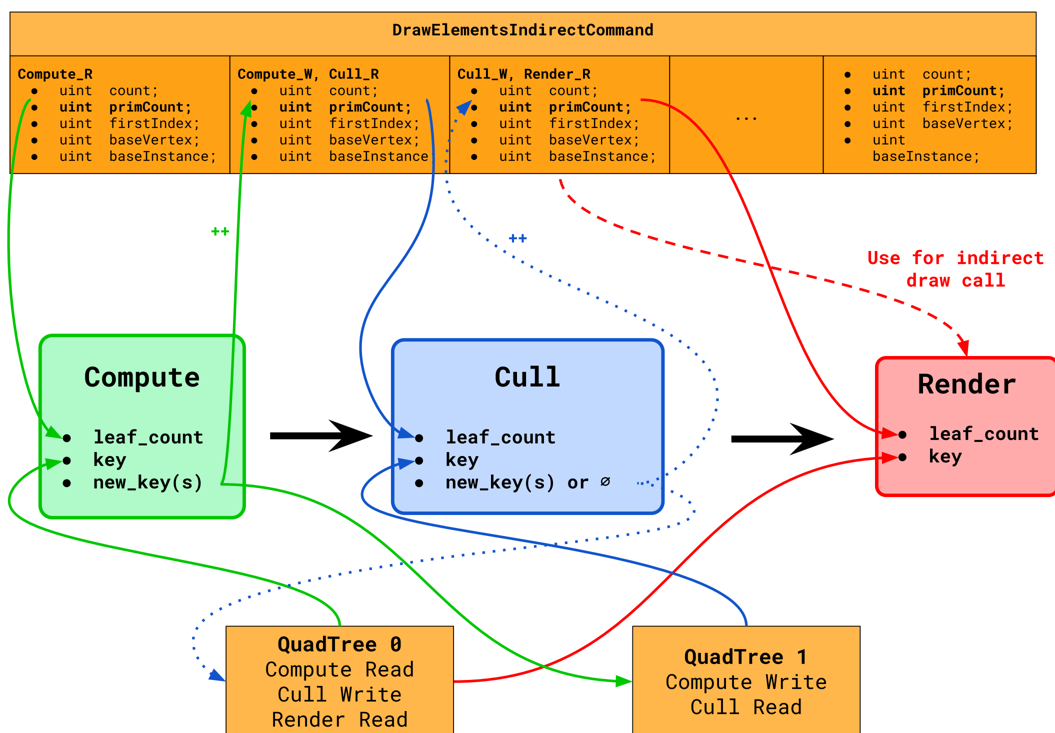


Figure 3.1: First program structure: 3 passes

3.1.3 Second Approach: 2 passes & 1 copy pass

This architecture has been developed with efficiency and ease of use as priorities.

First, we merge the Compute Pass and the Cull Pass to avoid unnecessary mappings and expensive computations.

To store the keys, we use 3 lists of keys instead of two. We denote them as `read_list`, `write_full_list` and `write_culled_list`, and we properly cycle between

these identifier on the CPU.

We switched from direct to indirect compute shader dispatching to be able to dynamically update the number of compute shader instances we launch, and the program now has two (and only two unique) command buffers, one containing the `DrawElementsIndirectCommand`, and the other, the `DispatchIndirectCommand`.

To keep track of the number of keys, we implement two arrays of simple atomic counters: the first one, `primCount_full`, for the full set of keys, and the second, `primCount_culled` for keys that pass the cull test. We associate to these arrays three indices `read_index`, `write_index` and `delete_index` which purpose are self-explanatory, and that are updated, and uploaded as uniform, from the CPU. This last index is equal to the `read_idx` minus some shift, allowing us to keep the transition between two states asynchronous.

With these notation we can detail the execution of the passes as follows:

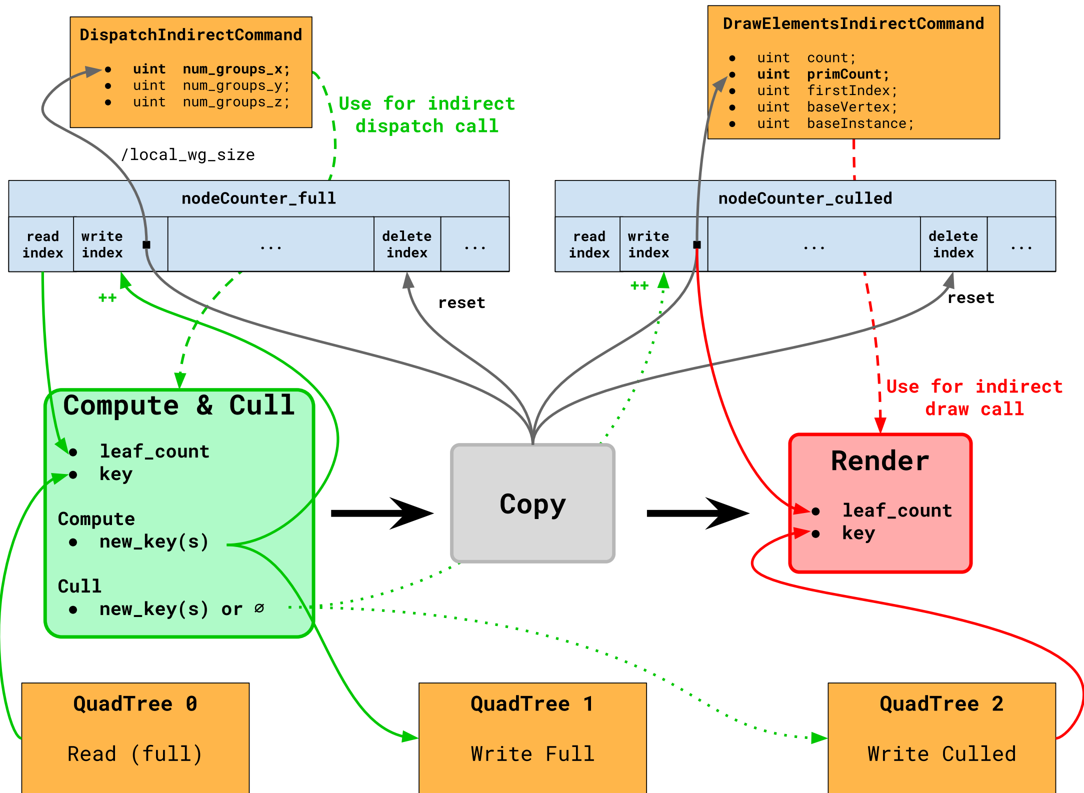


Figure 3.2: Second program structure, compute and cull pass are merged

Compute & Cull Pass A single pass is now in charge of updating the quadtree, and storing in a separate data-structure the keys corresponding to triangles that are visible during the present frame with the current camera settings (stored in the Model-View Matrix). Its execution is detailed in listing 3.4

```
layout (binding = 0) uniform atomic_uint primCount_full[16];
layout (binding = 1) uniform atomic_uint primCount_culled[16];
uniform int read_index, write_index;

// ...
int leaf_count = int(atomicCounter(primCount_full[read_index])); // Returns the
    value
int invocationID = getUniqueInvocationID();
if (invocationID > leaf_count)
   return;
uvec4 key = read_list[invocationID];
vec3 p_obj_space = mapToObjSpace(p_leaf_space, key);

// Compute
float current_LOD = getLevelInKey(key);
float target_LOD = computeLoD(p_obj_space);
if (target_LOD < current_LOD - 1) { // Merge
   if (isUpperLeftChild(key)) { // avoid storing 4 copies of the parent key
      uint idx = atomicCounterIncrement(primCount_full[write_index]); // returns
          value before atomic increment
      write_full_list[idx] = getParentKey(key);
   } else {
      return;
   }
} else if (target_LOD > current_LOD ) { // Divide
   uvec4 children[4] = getChildrenKeys(key);
   for (int i = 0; i < 3; ++i) {
      uint idx = atomicCounterIncrement(primCount_full[write_index]);
      write_full_list[idx] = children[i];
   }
} else { // Simply copy
   uint idx = atomicCounterIncrement(primCount_full[write_index]);
   write_full_list[idx] = key;
}
// Cull
if (isOutOfFrustum(p_obj_space)) // Cull
   return;
uint idx = atomicCounterIncrement(primCount_culled[write_index]);
write_culled_list[idx] = key;
```

Listing 3.4: Compute & Cull Pass in the second program architecture

We see here why the merging of the *Compute & Cull Pass* pass required the use of three lists of keys representing the full quadtree at $t-1$, the full quadtree at t and the culled quadtree at $t-1$.

Copy Pass If we were to set the local workgroup count to 32 and the last *Compute & Cull* pass stored 3200 keys in the `write_full_list`, we should then set the number of workgroups to $3200/32 = 100$ in the `DispatchIndirectCommand`. Nevertheless, this simple integer division cannot be performed from CPU while

copying data from a buffer to another without passing the data to the CPU and upload it again to GPU, hence forcing synchronism. This is the first reason why we investigated the idea of a *Copy Pass*. Note that the arrays of counter are here accessed by a single thread, and so they don't need to be bound as atomic counters.

```
// Only one instance of the Copy program is necessary
layout (local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

// Binding the arrays of primCount as SSBOs as no concurrence is needed
layout (std430, binding = 0) buffer full_buffer {
    uint primCount_full[16];
};
layout (std430, binding = 1) buffer culled_buffer{
    uint primCount_culled[16];
};
// ...
uint full_count = primCount_full[read_index];
uint culled_count = primCount_culled[read_index];

// Set the number of workgroups to instantiate at the next frame
workgroup_size_x = uint(full_count / float(local_wg_count)) + 1;
// Set the number of instances to draw during the following render pass
drawCommand.primCount = culled_count;

// Reset the counters with a shift
nodeCount_full[delete_index] = 0;
nodeCount_culled[delete_index] = 0;
```

Listing 3.5: Copy Pass in the second program architecture

Render Pass Apart from some minor modification to suit the new workflow, the render pass did not need a lot of changes.

This new implementation is illustrated in Figure 3.2

3.2 Key Data-Structure

3.2.1 Format Description

Previous Work: Our first implementation of Linear Quadtrees followed the format presented in [Dupuy et al., 2014]. The keys are stored in `uint` variables. Four bits at the end are reserved to store the subdivision level of the designated leaf, and the remaining 28 bits store the branchings, accumulated as pictured in figure 1.4a (page 9). As each branching is designated by an integer between 0 and 3, each one can be stored in 2 bits, allowing the key to store $28/2 = 14$ branchings, which works well with the maximum level of 15 that can be stored in the least

significant 4 bits. For example, to store the key `30123` at level 5, we set the key to:

$$0000\,0000\,0000\,0000\,0011\,0001\,1011\ -\ 0101_2$$

which in base4 is equivalent to (the level not being in base4).

$$00\,00\,00\,00\,03\,01\,23_4\ -\ 5_{10}$$

Contribution: new way to store the level in the key We realized that using four bits to define the level was quite wasteful, while storing explicitly the level of subdivision remained in itself necessary. Intuitively, if read the key stored in an variable set to 0, we cannot differentiate between leaf `0`, `00`, `000` etc. if we did not have a way to store and read the LoD in the key. Another way to see this problem is how to differentiate between the zeros corresponding to the initial value of the variable, and the zeros corresponding to actual branchings in the quadtree.

By defining the problem as such, we can see that it suffices to set some kind of barrier between the initialization zeros and the ones corresponding to branchings. This is easily done by prepending a `1` before the list of all branchings of the leaf. For example, the key 3 would be stored as

$$0000 \ldots 0000\,0111_2\ =\ 00 \ldots 00\,13_4$$

while the key `03` would be stored as

$$0000 \ldots 0001\,0011_2\ =\ 00 \ldots 01\,03_4$$

We can now define the key `00000`, corresponding to the upper-left leaf of a a quadtree subdivided five times, as

$$0000\,0000\,0000\,0000\,0000\,0100\,0000\,0000_2\ =\ 00\,00\,00\,00\,00\,10\,00\,00_4$$

This evolution allows us to save 2 bits and to reach a maximum level of subdivision of 15, but required the re-implementation of most of the function handling the keys and the level.

Contribution: 31 levels in a 4-integer key To exceed more significantly this number of possible subdivisions, and the ones generally achievable in the usual tessellation implementations on CPU or Tessellation Hardware, we need to find a way to circumvent the limitations induced by the use of `uint` variables.

Using the `long` format would provide us 64 bits to store the branchings and the level, but glsl doesn't offer `long` variables without the use of external extensions. Nevertheless, what is feasible with only native types is using two `uint` stored in a `vec2` that we artificially concatenates by reimplementing the bitwise operations, such as `leftShift`, `rightShift`, `findMSB` etc... Replacing the native implementations of these operation by our custom ones that take `vec2` as input, we easily adapt the algorithms to this new format. Since we still use 2 bits to prepend the 1 that delineate the transition between the zeros of the initial variable value from the zeros that designate branchings, we now have $2 \times 32 - 2 = 62$ bits to store the branching, allowing 31 level of subdivision !

Remember that we create one quadtree per control mesh primitive (or one per root triangle in the screen-space approach). To decode a key and find the position in world space of the currently treated leaf, we need means to recover the position of the control mesh primitive in world space on which we should map the quadtree. For this purpose, the key stores an integer `meshPolygonID` that uniquely designate the face of the control mesh to which the current quadtree corresponds. In other words, all of the nodes of a given quadtree share this identifier, and when mapping from quadtree-space to object-space, we use this ID as index in the list of faces that are uploaded on the video memory.

Furthermore, as the control mesh is triangulated, we also need an identifier that labels which one of the 3 (or 4) root triangles corresponds to our quadtree. An additional integer `rootID` is then used for this purpose.

Finally, recall that we need 3 bits for the X-Morph, Y-Morph and Destroy flags for our T-Junction Removal algorithm to work, as explained in section 2.3.2. As the `rootID` only takes two bits, we can append these flags at the most significant position of the integer used for `rootID`.

The key in its final implementation is then defined in an `uvec4` as depicted in Listing 3.6

```
key[0] = nodeID_MostSignificant32Bits;
key[1] = nodeID_LeastSignificant32Bits;
key[2] = meshPolygonID;
key[3] = (X_Moph_flag, Y_Moph_flag, Destroy_flag, rootID);
```

Listing 3.6: Key data-structure in an uvec4

3.2.2 Contribution: Key Initialization

At initialization, the program should fill the `read_list` with keys such that the quadtree algorithm can start subdividing the mesh. Let n be the number of

primitives (quads or triangles) in the control mesh. Since we triangulate the primitives and create one quadtree per root triangle, we respectively obtain $3n$ quadtrees for a triangle mesh, and $4n$ quadtrees for a quad mesh. At initialization, the quadtrees are not subdivided, and the `nodeID` is simply 0, prepended with the 1 determining the level by its position. Since the level is zero, we have

$$nodeID = 0000\ldots0000\,0001_2 \;=\; 00\ldots00\,01_4$$

The `meshPolygonID` has to correspond to the position of the face in the list. Since all root triangles are equal at initialization and that the faces are stored in a tightly packed data-structure, it suffices to increment the `meshPolygonID` for each face quadtrees.

Finally, the `rootID` is simply an integer between 0 and 2 for triangle meshes, and 0 and 3 for quad meshes, designating which of the root triangles created on the mesh polygon corresponds to the quadtree. Once again, since all quadtrees are similar at creation, we can simply increment `rootID` at creation for each root triangle key of a given mesh primitive. The flags sharing the integer are never set at initialization and so we can ignore them.

As for now, the size of the buffers containing the quadtrees is not dynamic, we reserve as much memory as possible and use the routine in listing 3.7 to initialize it:

```
// Key data-structure:
// key = uvec4(nodeID_MSB, nodeID_LSB, meshPolygonID, rootID)
uvec4* read_list = new uvec4[max_num_nodes];
if (primitive_type == TRIANGLES) {
    init_node_count_ = 3 * mesh_data_->triangle_count;
    for (int ctr = 0; ctr < mesh_data_->triangle_count; ++ctr) {
        read_list[3*ctr+0] = uvec4(0, 1, uint(ctr*3), 0);
        read_list[3*ctr+1] = uvec4(0, 1, uint(ctr*3), 1);
        read_list[3*ctr+2] = uvec4(0, 1, uint(ctr*3), 2);
    }
} else if (primitive_type == QUADS) {
    init_node_count_ = 4 * mesh_data_->quad_count;
    for (int ctr = 0; ctr < mesh_data_->quad_count; ++ctr) {
        read_list[4*ctr+0] = uvec4(0, 1, uint(ctr*4), 0);
        read_list[4*ctr+1] = uvec4(0, 1, uint(ctr*4), 1);
        read_list[4*ctr+2] = uvec4(0, 1, uint(ctr*4), 2);
        read_list[4*ctr+3] = uvec4(0, 1, uint(ctr*4), 3);
    }
}
```

Listing 3.7: Key initialization

For example, if the control mesh was a cube defined by six squares, we would obtain at initialization the array detailed in listing 3.8

```
// Key data-structure:
// key = uvec4(nodeID_MSB, nodeID_LSB, meshPolygonID, rootID)
// Face 0
read_list[0] = uvec4(0, 1, 0, 0);
read_list[1] = uvec4(0, 1, 0, 1);
read_list[2] = uvec4(0, 1, 0, 2);
read_list[3] = uvec4(0, 1, 0, 3);
// Face 1
read_list[4] = uvec4(0, 1, 1, 0);
read_list[5] = uvec4(0, 1, 1, 1);
read_list[6] = uvec4(0, 1, 1, 2);
read_list[7] = uvec4(0, 1, 1, 3);

...

// Face 5
read_list[20] = uvec4(0, 1, 5, 0);
read_list[21] = uvec4(0, 1, 5, 1);
read_list[22] = uvec4(0, 1, 5, 2);
read_list[23] = uvec4(0, 1, 5, 3);
```

Listing 3.8: Example of initialization keys for a quad cube

3.2.3 Contribution: Key Decoding

As mentioned in section 1.3, the implementation of Linear Quadtrees of [Dupuy et al., 2014] is provided with functions that return the translation and scaling that, applied to the base quad, map it to its position in the unit square of the Quadtree. But as our triangle tessellation pattern also required rotation for the children 1 and 2, it was necessary to redesign the recovering of transformations for a given key from scratch.

The idea behind this new design is to consider the key as a pile of branchings, and decode it index by index, from root to leaf, computing each time the associated transformation and accumulating them to finally return the transformation matrix that maps the unit triangle to the position of the leaf in the triangle quadtree. The difficulty of properly accumulating the transformations is due to the fact that, by construction, the mapping from leaf-space to quadtree-space should cycle between scaling, rotation, and translation, at each branching, all of this relative to a parent that is considered to span on the unit triangle $\{(0,0),(0,1),(1,0)\}$. In other words, the complete mapping from leaf-space to quadtree-space is the matrix containing one rotation, one scaling and one translation, that when applied to a point is equivalent to successively applying the relative branching transformation corresponding to each branching in the key. As by definition, the relative branching transformations are defined in function of a parent spanning on the unit triangle,

the routine thus cannot compute the translation and angle of rotation at each branching and simply accumulate them to the previous translation and angle of rotation, since the parent could already be translated and rotated.

The solution is then to adapt the relative branching transformation by taking into account the ones previously computed. Intuitively, we see that:

- The angles of rotation can simply be added up at each iteration
- The scaling is divided by 2 at each iteration and so the scaling can simply be accumulated by multiplying it by $\frac{1}{2}$ for each branching
- The translation is the less trivial transform, as it should have its length equal to the scaling, and its natural direction (given as relative to the parent in the unit triangle) rotated by the previous rotation angle.

More explicitly, we can write this routine as the following pseudo-code (listing 3.9)

```
float theta = 0.0, scale = 1.0;
vec2 tmp, translation;
int current_branching; // integer between 0 and 3
for (int i = level-1; i >= 0, i--) // for each branching, starting from root
{
   current_branching = getBranching(key, i);
   tmp = scale * getTranslation(current_branching);
   translation += rotate(tmp, theta);
   theta += getRotation(current_branching);
   scale *= 0.5;
}
```

Listing 3.9: Overview of the key decoding algorithm

The respective transformation for each possible branching (*i.e.* the ones returned by `getTranslation` and `getRotation`) are detailed in figure 1.7 (page 13).

These functions could have been naively implemented as a switch between four cases, returning in each one the correct transformation, but it would have been very inefficient performance-wise. Indeed, at compile time, each conditional statement for which the condition is not evaluated on an uniform variable gets developed, and each branch gets evaluated even if only the correct ones return or affect a variable. For this reason, we used binary operations that allow us to avoid any kind of condition. Let us first clearly list the respective transforms in table 3.1

Here, b1b2 is a naming convention used to make the binary operations more explicit when using the bit representation of the branching. Stored in uint variables, b1b2, b1 and b2 are easily recovered by using a shifting mask on the `nodeID`. To

Table 3.1: Relative transformation from parent unit right triangle to leaf position. The rotation needs to be multiplied by $\pi/2$ and the translation by $\frac{1}{2}$

Branching	b1b2	b1	b2	rotation	translation.x	translation.y	scale
0	00	0	0	0	0	1	0.5
1	01	0	1	3	0	1	0.5
2	10	1	0	1	1	0	0.5
3	11	1	1	0	1	0	0.5

illustrate:

$$branching = 2_4 \implies b_1 b_2 = 10_2 \implies b_1 = 1 \,\&\, b_2 = 0$$

Thanks to this representation of the relative transformation, the `getTranslation` method can be trivially implemented as shown in listing 3.10.

```
vec2 getTranslation(uint b1)
{
   vec2 translation;
   translation.x = float(b1 & 0x1); // eq. to (b1 == 1)
   translation.y = float(b1 ^ 0x1); // eq. to (b1 == 0)
   return translation * 0.5;
}
```

Listing 3.10: `getTranslation` implementation

In practice, the `getRotation` method has been implemented in a single-liner, but the operations have been decomposed for greater expressive power in the listing 3.11. We report the operations in table 3.2 in which all values are in binary, and we see that we obtain the desired results

```
float getRotation(uint b1b2, uint b1, uint b2)
{
   uint a = (b1b2 ^ 0x2);
   uint b = (a | 0x1);
   uint c = (b1 ^ b2);
   return (b * c) * M_PI * 0.5;
}
```

Listing 3.11: `getTranslation` implementation

A final optimization that can be implemented regards the 'rotate(theta, translation)' method. In its naive variant, it can be defined as shown in listing 3.12

Nevertheless, trigonometric functions are very expensive, and this method is eventually called many times per key, for each key, at each frame. But one can notice that all of the angles are multiples of $\pi/2$, so their their cosine and sine are in $\{-1, 0, 1\}$ and the rotation can be emulated by carefully chosen sign inversions

Table 3.2: Computing the angle of rotation from binary representation of current branching. The final result needs to be multiplied by $\pi/2$

$b1b2$	$b1$	$b2$	$a = b1b2 \oplus 10$	$b = a \vee 01$	$c = b1 \oplus b2$	$r = b \times c$	r_{10}
00	0	0	10	11	0	00	0
01	0	1	10	11	1	11	3
10	1	0	00	01	1	01	1
11	1	1	01	01	0	00	0

on the values of the translation, as we did in our code.

```
vec2 rotate(float theta, vec2 tr)
{
   vec2 r;
   float cosT = cos(theta), sinT = sin(theta);
   r.x = cosT * tr.x - sinT * tr.y;
   r.y = sinT * tr.x + cosT * tr.y;
   return r;
}
```

Listing 3.12: Naive implementation of the `rotate` function

3.3 Contributions: Neighbour LoD Check, two other approaches

In section 2.3.2, we listed the three approaches that we developed for the neighbour LoD check, but only detailed the one used in the final implementation. In this section we will explain the remaining two and discuss our final choice.

3.3.1 Neighbour Key Recovering

As its name suggests, Neighbour Key Recovering consists in recovering the key of the relevant neighbour, and use it to map the leaf-space hypotenuse $[(1,0),(0,1)]$ to object-space, as we usually do in our Compute Pass. Nevertheless, finding the adjacent key from the current key is not trivial, as the pattern appearing in a subdivided quadtree shows no obvious relationship between two adjacent leaves key, as depicted on figure 3.3

As the keys are built recursively, the first approach has been to decode the keys in a recursive routine, during which we treat the last branching of the key. More precisely, our recursive routine N is composed of two recursive subroutine X_n and Y_n that respectively compute the key of the X-Neighbour and the Y-Neighbour, as well as a deterministic subroutine Out that is called when the computations indicate that the neighbour key is in an adjacent root triangle (cf figure 2.8 page 22). Let us define a sub-key A_i from an original key A_0 of level n

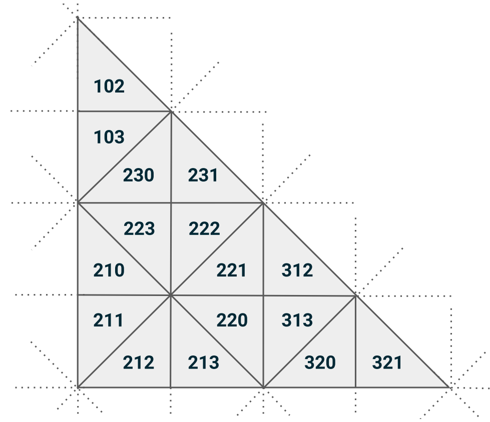


Figure 3.3: Extract from a uniformly subdivided quadtree at level 3

as

$$A_i = q_n q_{n-1} q_{n-2} ... q_{i-2} q_{i-1} q_i \text{ where } q_k \in \{0, 1, 2, 3\}$$

We obtain

$$N(A_i) = (N(A_{i-1}), N(q_i))$$

Here, $N(q_i)$ always returns a base4 value deterministically, but $N(A_{i-1})$ can either return the resulting key directly or continue in a recursive manner, depending on the subroutine the algorithm is currently in and the value of q_i. Note that the keys are considered as a string of base4 elements, and that every operation is computed in base4 (*e.g.* $113_4 + 1_4 = 120_4$). The three subroutines are defined in algorithm 5.

To give an example of execution, let us say that we look for the Y-Neighbour of $A_0 = 312$. We directly call the subroutine Y_n on the key and the recursive routine will execute as follow:

$$\begin{aligned} Y_n(A_0) &= Y_n(312) \\ &= (X_n(31), 1) \\ &= ((Y_n(3), 2), 1) \\ &= (((2), 2), 1) \\ &= 221 \end{aligned}$$

Looking at figure 3.3, we see that 221 is indeed the Y-Neighbour of 312. From this key we can recover the object-space position of the relevant neighbour hypotenuse. Since recursion is not achievable on the GPU, I broke down the subroutines and merged them in a function that follows the same steps and evaluations but in a state machine manner, the next state being dictated by the treated branch-

Algorithm 5 Neighbour key recursive algorithm

```
 1: function Xn(A_0, A_i)
 2:     if A_i has length 1 then
 3:         switch BranchingID do
 4:             case 0 return 1
 5:             case 1 return Out(A_0)
 6:             case 2 return 3
 7:             case 3 return Out(A_0)
 8:     else
 9:         q_i ← last branching of A_i
10:         switch q_i do
11:             case 0 or 2 return A_i + 1_4
12:             case 1 return (Y_n(A_0, A_{i-1}), 2)
13:             case 3 return (X_n(A_0, A_{i-1}), 0)
14: function Yn(A_0, A_i)
15:     if A_i has length 1 then
16:         switch BranchingID do
17:             case 0 return Out(A_0)
18:             case 1 return 0
19:             case 2 return Out(A_0)
20:             case 3 return 2
21:     else
22:         q_i ← last branching of A_i
23:         switch q_i do
24:             case 1 or 3 return A_i − 1_4
25:             case 0 return (Y_n(A_0, A_{i-1}), 3)
26:             case 2 return (X_n(A_0, A_{i-1}), 1)
27: function Out(A_i)
28:     for all Branching q_i in A_i do <text>
29:         q_i ← 3_4 − q_i
        return A_i
```

ing. This method has the advantage of tearing down the isolation constraint of our subdivision, since we can now recover all the informations about neighbouring nodes by first computing their key. This could be very advantageous in the future (*e.g.* when optimizing for textures), but seems a bit overkill for the original purpose.

3.3.2 Pre-Mapping Reflect

Recall that the reflection method consists in reflecting the parent hypotenuse in a given space to recover the coordinates in object-space of the target neighbour hypotenuse in order to compute the associated LoD. By then comparing the resulting LoD with the level of the currently treated leaf, we can detect T-Junctions.

Contrary to the Post-Mapping Reflect, the Pre-Mapping Reflect consists in computing the position of the hypotenuse of interest in unit-space, and from unit-space mapping it to object-space using the key of the currently treated leaf. Depending on the relative position as a child, we choose one of two easily computed reflected hypotenuse and we then map these position to object-space as we would have mapped the current node's parent, and obtain the coordinates from which we can compute the LoD. As with the two previously presented methods, if the neighbour lies on an adjacent root triangle, we compute its key and map the unit triangle hypotenuse to world space. The hypotenuses of interest are pictured in orange in fig.2.10 (page 24) and the Pre-Mapping Reflect can be implemented with the following approach, where we call A_{target} the key used to map the Unit-Space reflected hypotenuse H_u for the current leaf designated by key A_{leaf}:

Algorithm 6 Pre-Mapping Reflect

1: **if** the desired neighbour is outside the current root triangle **then**
2: $\quad A_{target} = Out(A_{node})$
3: $\quad H_u = [(1, 0), (0, 1)]$
4: **else**
5: $\quad A_{target} = Parent(A_{node})$
6: $\quad$ **if** current node is on X Border of its parent (child 2 or 3) **then**
7: $\quad\quad H_u = [(1, 0), (0, -1)]$
8: $\quad$ **else**
9: $\quad\quad H_u = [(-1, 0), (0, 1)]$
10: Map H_u to object-space using A_{target}

4 Results

In this section, we will present a few visual results yielded by our implementations. We will not be able to produce any kind of quantitative measures unfortunately, as I was not able to have a satisfying implementation on Tessellation Hardware that would make comparison relevant.

4.1 Distance Based Pipeline

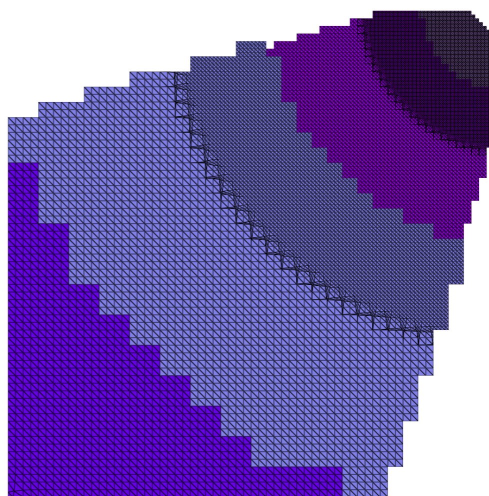
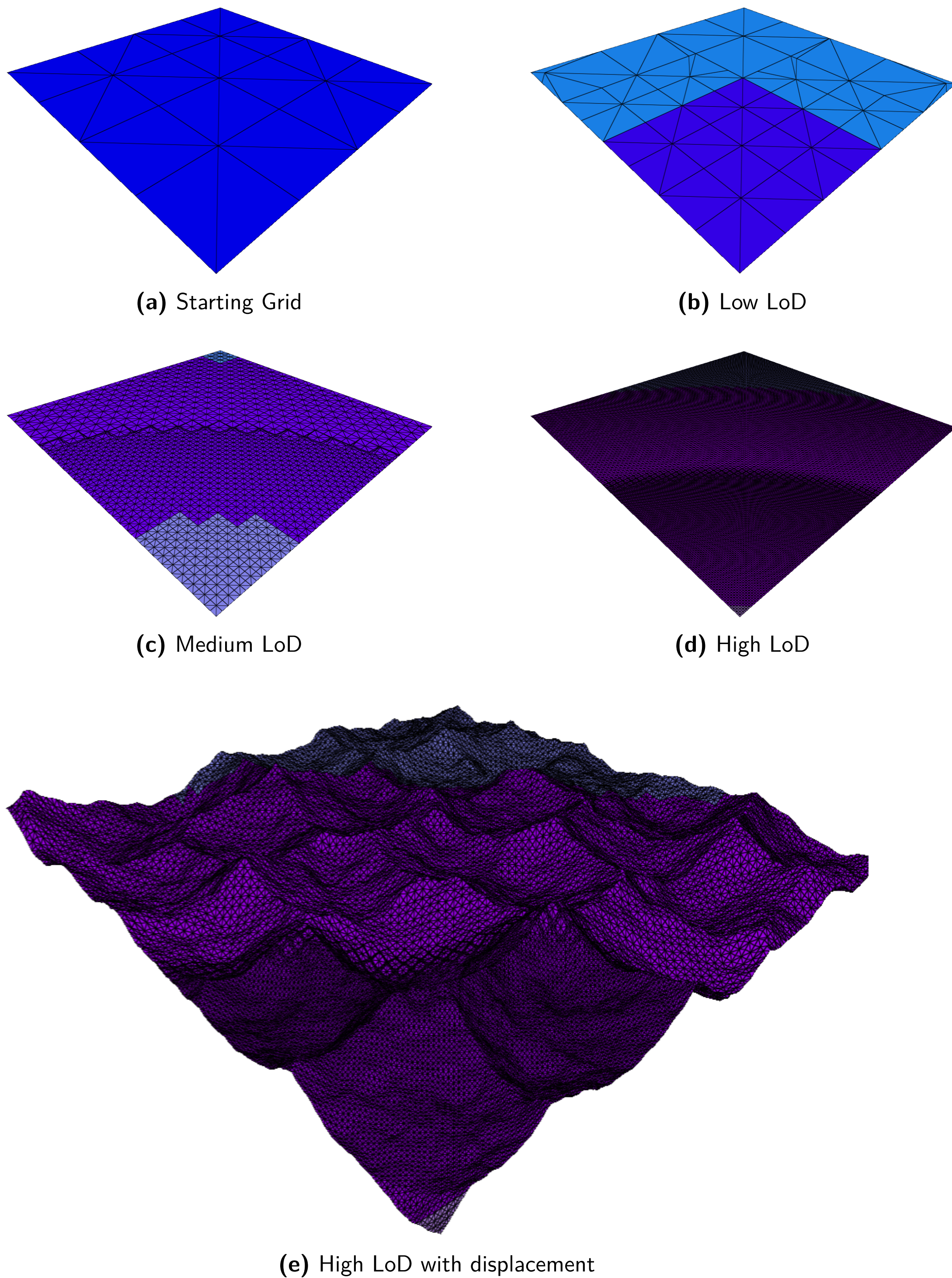


Figure 4.1: Top view of the subdivided terrain. The different colors represent different LoDs, and the morph zones are visible at transition between two grid levels inside the same LoD.



(a) Starting Grid

(b) Low LoD

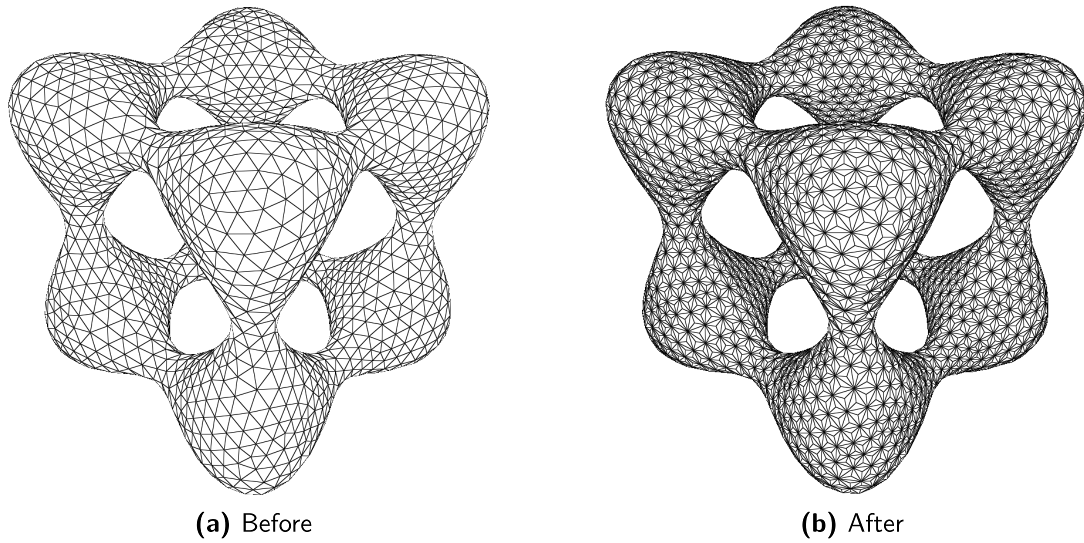
(c) Medium LoD

(d) High LoD

(e) High LoD with displacement

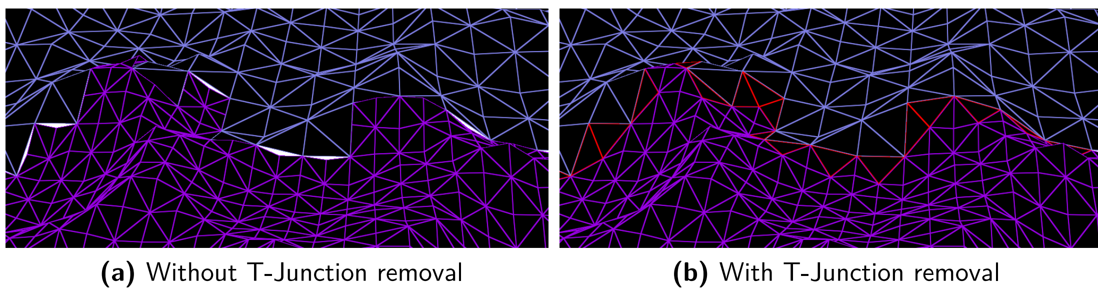
Figure 4.2: Distance-Based Subdivision on a triangle grid. The different colors represent different LoDs.

4.2 Screen-Scape Pipeline



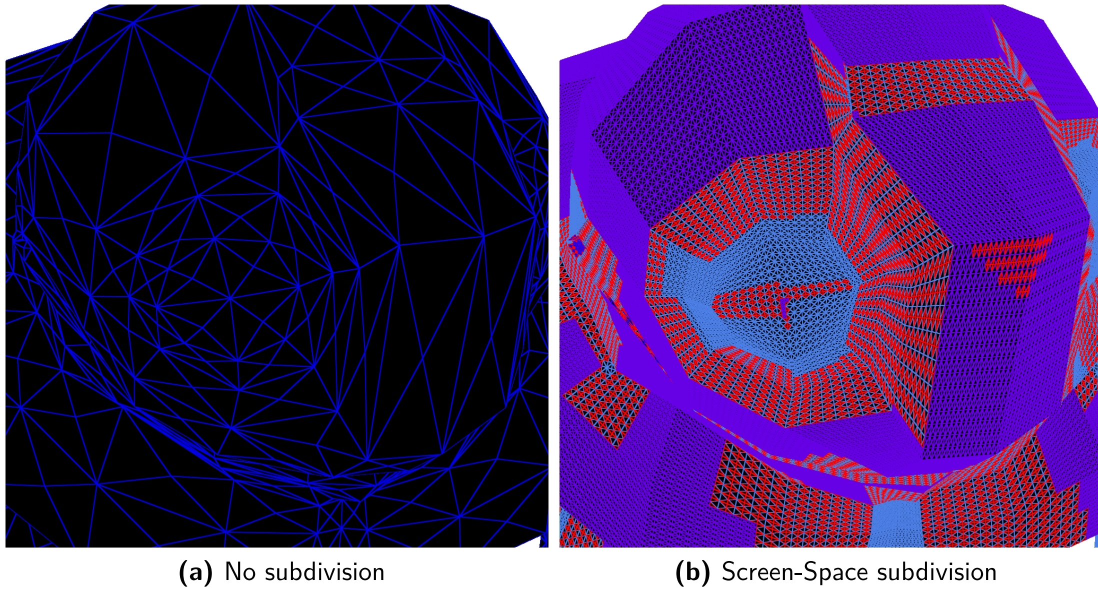
(a) Before **(b)** After

Figure 4.3: Effect of our triangulation on a triangle mesh



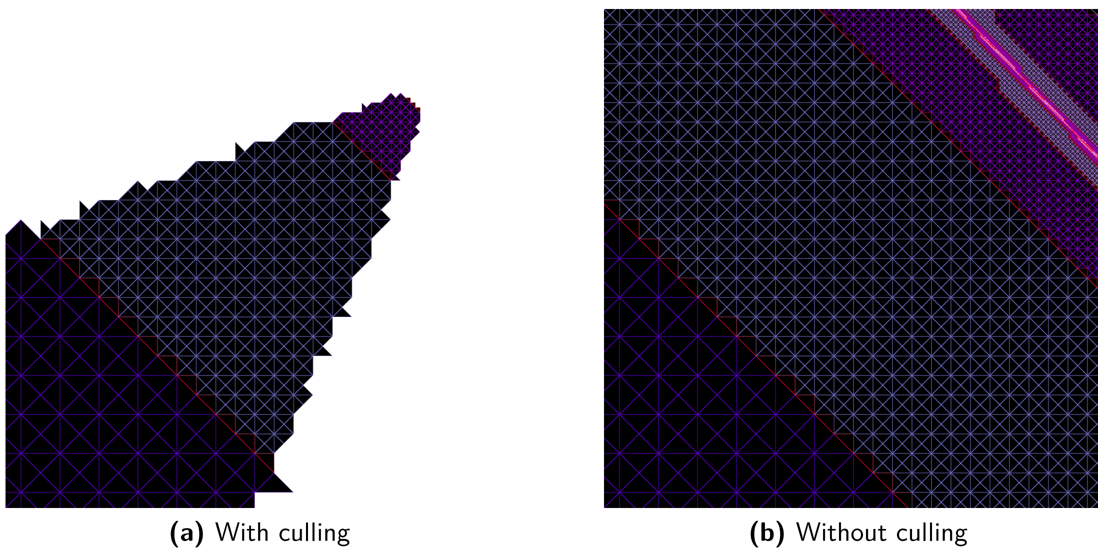
(a) Without T-Junction removal **(b)** With T-Junction removal

Figure 4.4: Visual result of our T-Junction removal algorithms. In shades of blue: LoD level. In red: triangles morphed in the T-Junction removal



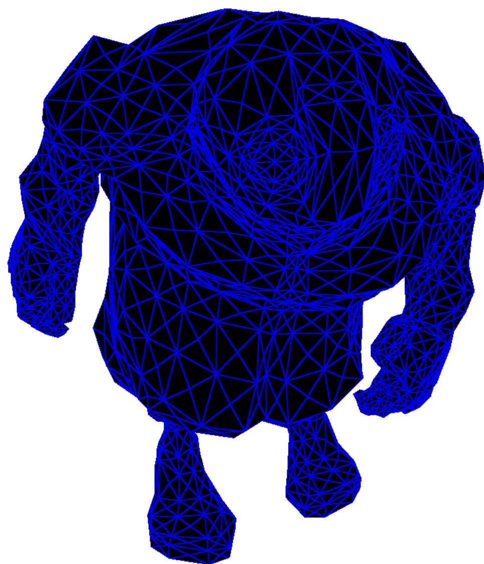
(a) No subdivision **(b)** Screen-Space subdivision

Figure 4.5: Close-up on a detail of BigGuy. Notice the great reduction in triangle size variance in the subdivided mesh. In shades of blue: LoD level. In red: triangles morphed in the T-Junction removal

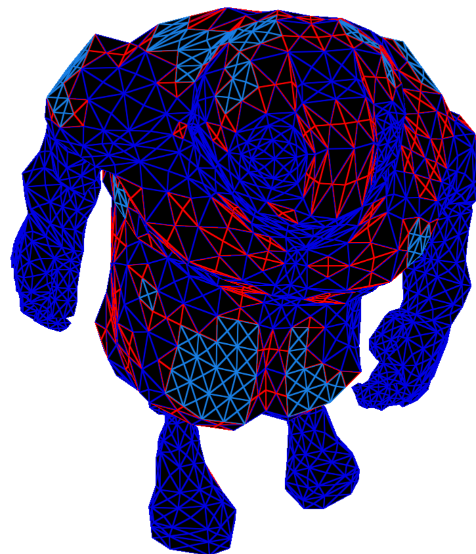


(a) With culling **(b)** Without culling

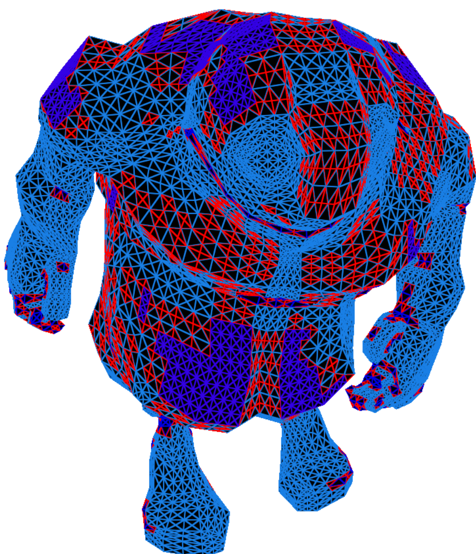
Figure 4.6: Top view of a 2D grid subdivided with the screen-space pipeline. We see in the picture without culling the problem happening at the near plane of the projection matrix



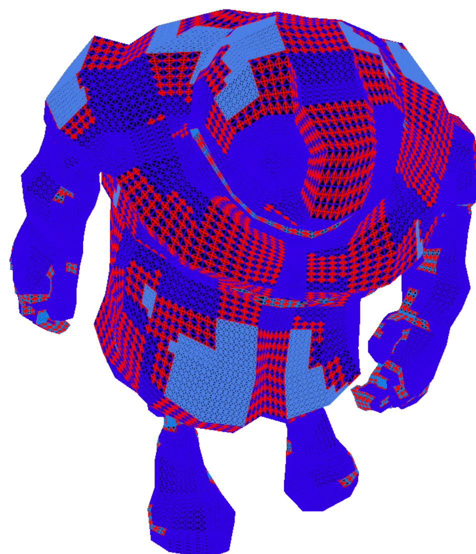
(a) LoD factor 1



(b) LoD factor 3



(c) LoD factor 5



(d) LoD factor 5

Figure 4.7: Showcase of our screen-space pipeline, at the same distance with 4 different LoD factors. In shades of blue: LoD level. In red: triangles morphed in the T-Junction removal

5 Conclusions

In this thesis, we took as starting point the work proposed in [Dupuy et al., 2014], and iterated from this implementation, each time straying further away from both the traditional quadtree usage and the initial implementation of Linear Quadtrees, in the pursuit of a complete rendering pipeline that can seamlessly render subdivision surfaces from any kind of mesh.

We started by using a subdivision criterion often used along quadtrees for terrain rendering: the distance from polygon to camera position. Around this LoD function we build the first prototype of our pipeline, which worked well, but needed many improvements from an architectural point of view. We also realized that a distance metric wasn't ideal in all cases, and decided to try a new approach.

Once our goal was set on using screen-space metrics to define the level of detail, we were confronted with the major constraint of keeping the tree restricted. The solution found required a drastic change in our mesh processing, as the program performs the triangulation of all mesh polygons. Along with this new insight, we rebuild most of the program structure to be as flexible and easy to maintain as possible.

Finally, we managed to obtain a few qualitative results, yielding very clean subdivided meshes, proving that our subdivision scheme does not yield any kind of cracks.

5.1 Future Work

A major evolution to this project that could increase its impact on the industry would be to implement the Catmull Clark Subdivision Surfaces. This

would join together the tessellation used in offline and real-time pipelines, allowing the use of the same assets for a movie and a video-game (for example).

An idea that could be worth investigating resides in using the tessellation hardware to perform the last few subdivision level, taking advantage of both our adaptability and of the performance of the tessellator.

The ROAM article [Duchaineu et al., 1999] proposes a subdivision scheme that is close to the one we chose, while proposing what would be intermediate levels between two levels of our implementation. This could offer a more progressive tessellation.

To optimize the memory footprint of our implementation, we could modify the buffer implementation such that the size of the SSBOs containing the quadtree keys is dynamically updated.

From a commercial point of view, to maximize the spread of real-time tessellation, this implementation could be implemented as a feature in the Unity3D game engine.

Finally, as we triangulate the control mesh polygon, it would be possible to first adapt our program to run on meshes containing both quad and triangles, and in a second time, to define more triangulations for less commonly used polygons (hexagons, octogons, etc...)

5.2 Personal Insight

This project has been extremely fulfilling for me. My understanding of OpenGL, and in a broader sense, of the mechanisms implemented on a graphics card, has increased immensely.

It also gave me the chance to dive in the subject of Tessellation that I previously briefly discovered during the project *Procedural Terrain Rendering* for the Introduction to Computer graphics course at E.P.F.L.. With retrospect, I realize I should have adopted a more scientific work methodology, by implementing aside my work existing algorithm to be able to produce quantitative performance comparisons. But the thrive to make more advanced and efficient algorithms work well, and the time required to implement from scratch all of the technologies presented in this thesis, didn't leave me much breathing room to wonder what kind of existing algorithm I could implement, for the sole purpose of comparison.

From a professional standpoint, working at Unity Labs has truly been a treat, and I cannot thank the team of Grenoble enough for their support, patience and encouragement. Their enthusiasm, passion, scientific culture and curiosity are greatly responsible for my decision to pursue a career in Graphics research.

References

[D3D, 2009] (2009). *Direct3D 11 Features*. Microsoft.

[Brainerd et al., 2016] Brainerd, W., Foley, T., Kraemer, M., Moreton, H., & Nießner, M. (2016). Efficient gpu rendering of subdivision surfaces using adaptive quadtrees. *ACM Transactions on Graphics (TOG)*, 35(4), 113.

[Cantlay, 2011] Cantlay, I. (2011). Directx 11 terrain tessellation. *Nvidia whitepaper*, 8(11).

[Carpenter, 1984] Carpenter, L. (1984). The a-buffer, an antialiased hidden surface method. *ACM Siggraph Computer Graphics*, 18(3), 103–108.

[Catmull, 1974] Catmull, E. (1974). *A subdivision algorithm for computer display of curved surfaces*. Technical report, UTAH UNIV SALT LAKE CITY SCHOOL OF COMPUTING.

[Duchaineu et al., 1999] Duchaineu et al. (1999). Roaming terrain: Real-time optimally adapting meshes. *Proceedings of the SIGGRAPH '99, Denver, USA*, (pp. 56–67).

[Dupuy et al., 2014] Dupuy, J., Iehl, J.-C., & Poulin, P. (2014). Quadtrees on the gpu. *GPU Pro5: Advanced Rendering Techniques*, 5, 439–450.

[Gargantini, 1982] Gargantini, I. (1982). An effective way to represent quadtrees. *Communications of the ACM*, 25(12), 905–910.

[Lindstrom et al., 1996] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., & Turner, G. A. (1996). Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (pp. 109–118).: ACM.

[Nießner et al., 2016] Nießner, M., Keinert, B., Fisher, M., Stamminger, M., Loop, C., & Schäfer, H. (2016). Real-time rendering techniques with hardware tessellation. In *Computer Graphics Forum*, volume 35 (pp. 113–137).: Wiley Online Library.

[Pajarola, 2002] Pajarola, R. (2002). Overview of quadtree-based terrain triangulation and visualization.

[Patney & Owens, 2008] Patney, A. & Owens, J. D. (2008). Real-time reyes-style adaptive surface subdivision. In *ACM Transactions on Graphics (TOG)*, volume 27 (pp. 143).: ACM.

[Riccio, 2012] Riccio, C. (2012). Southern islands in deep dive. SIGGRAPH Tech Talk.

[Segal & Akeley, 2017] Segal & Akeley (2017). *The OpenGL Graphics System: A Specification (version 4.5)*. Khronos Group.

[Strugar, 2009] Strugar, F. (2009). Continuous distance-dependent level of detail for rendering heightmaps. *Journal of graphics, GPU, and game tools*, 14(4), 57–74.